Министерство образования, науки и молодежи Республики Крым

Государственное бюджетное профессиональное образовательное учреждение

Республики Крым

«Феодосийский политехнический техникум»

«Утверждаю»

Заместитель директора

по учебной работе

\_\_\_\_\_\_\_\_О.Г. Сердюкова

« » \_\_\_\_\_\_\_\_\_\_ 201\_г.

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ОРГАНИЗАЦИИ**

**САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ**

**УЧЕБНОЙ ДИСЦИПЛИНЫ ОП.05 ОСНОВЫ ПРОГРАММИРОВАНИЯ**

для специальности среднего профессионального образования

09.02.03 «Программирование в компьютерных системах»

201\_\_г.

Методические указания по организации самостоятельной работы обучающихся учебной дисциплины ОП.05«Основы программирования» разработаны на основе Рабочей программы учебной дисциплины ОП.05«Основы программирования» и в соответствии с учебным планом специальности 09.02.03 Программирование в компьютерных системах

Организация-разработчик: Государственное бюджетное профессиональное образовательное учреждение Республики Крым «Феодосийский политехнический техникум»

Разработчик Дворянова Т.Н., преподаватель компьютерных дисциплин

Методические указания по организации самостоятельной работы обучающихся учебной дисциплины ОП.05«Основы программирования» рассмотрены и одобрены на заседании цикловой комиссии компьютерных дисциплиндисциплин.

Протокол № от « »\_\_\_\_\_\_ 201\_\_ года

Председатель цикловой комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Т.Н. Дворянова

СОДЕРЖАНИЕ

[ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 4](#_Toc467066721)

[Тематический план и содержание самостоятельной работы 7](#_Toc467066722)

[**Тема.** Способы записи алгоритмов 9](#_Toc467066723)

[**Тема.** Компоненты среды. Общие свойства компонентов. 12](#_Toc467066724)

[**Тема.** Типы данных 16](#_Toc467066725)

[**Тема.** Выражения и операции 20](#_Toc467066726)

[**Тема.** Примеры использования основных и специальных классов. 27](#_Toc467066727)

[**Тема.** Структура фрагмента программы обработки исключений 37](#_Toc467066736)

[**Тема.** Операции над множествами. 40](#_Toc467066737)

[**Тема.** Динамический массивВыделение и освобождение динамической памяти 45](#_Toc467066740)

[**Тема.** Процедуры и функции для работы с файлами. 48](#_Toc467066741)

[**Тема.** Библиотеки подпрограмм. Рекурсивные подпрограммы 53](#_Toc467066742)

[**Тема.** Программирование графики. Вывод текстовой информации 59](#_Toc467066743)

[**Тема.** Диаграммы 89](#_Toc467066746)

[**Тема.** Элементы управления. Отображение заголовков 91](#_Toc467066747)

[**Тема.** Организация взаимодействия форм. Создание многостраничных окон. 98](#_Toc467066748)

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Согласно учебного плана специальности09.02.03 Программирование в компьютерных системах по дисциплине ОП.05Основы программирования объем самостоятельной работы студентов предусматривает 98 часов.

Самостоятельная работа выполняется обучающимися в рамках изучения дисциплины под контролем преподавателя, как в аудиторное, так и внеаудиторное время. Самостоятельная работа играет важную роль в воспитании сознательного отношения обучающихся к овладению теоретическими знаниями и практическими умениями, формирования привычки к направленному интеллектуальному труду. Самостоятельная работа направлена на формирование умений и навыков практического решения задач, на развитие логического мышления, творческой активности, исследовательского подхода в освоении учебного материала, развития познавательных способностей. Самостоятельная работа выполняет не только развивающие и воспитательные функции, но и способствует формированию **профессиональных компетенций:**

ПК 1.1. Выполнять разработку спецификаций отдельных компонент.

ПК 1.2. Осуществлять разработку кода программного продукта на основе готовых спецификаций на уровне модуля.

ПК 1.3. Выполнять отладку программных модулей с использованием специализированных программный средств.

ПК 1.4. Выполнять тестирование программных модулей.

ПК 1.5. Осуществлять оптимизацию программного кода модуля.

ПК 3.1. Анализировать проектную и техническую документацию на уровне взаимодействия компонент программного обеспечения.

Способствует формированию у обучающегося **общих компетенций:**

ОК1. Понимать сущность и социальную значимость своей будущей профессии, проявлять к ней устойчивый интерес.

ОК2. Организовывать собственную деятельность, определять методы и способы выполнения профессиональных задач, оценивать их эффективность и качество.

ОК 3. Решать проблемы, оценивать риски и принимать решения в нестандартных ситуациях.

ОК 4. Осуществлять поиск, анализ и оценку информации, необходимой для постановки и решения профессиональных задач, профессионального и личностного развития.

ОК5. Использовать информационно-коммуникационные технологии для совершенствования профессиональной деятельности.

ОК6. Работать в коллективе и команде, обеспечивать ее сплочение, эффективно общаться с коллегами, руководством, потребителями.

ОК 7. Ставить цели, мотивировать деятельность подчиненных, организовывать и контролировать их работу с принятием на себя ответственности за результат выполнения заданий.

ОК 8. Самостоятельно определять задачи профессионального и личностного развития, заниматься самообразованием, осознанно планировать повышение квалификации.

ОК 9. Быть готовым к смене технологий в профессиональной деятельности.

Целью методических указаний по организации самостоятельной работы студентов учебной дисциплины ОП.05Основы программирования является обеспечение эффективности самостоятельной работы обучающихся с литературой и интернет - ресурсами на основе организации их изучения.

Задачами методических рекомендаций по организации самостоятельной работы являются:

- активизация самостоятельной работы студентов;

- содействие развития творческого отношения к учебной дисциплине;

- выработка умений и навыков рациональной работы с литературой, источниками;

- управление познавательной деятельностью студентов.

Средства обучения, необходимые для организации самостоятельной работы:

1.Дидактические средства (сборники задач, интернет – источники, электронная библиотека).

2.Технические средства, при помощи которых предъявляется учебная информация (компьютеры, мультимедия).

3.Средства, которые используют для руководства самостоятельной деятельностью студентов (методические указания по организации самостоятельной работы, методические указания по выполнению лабораторных и практических работ).

**Критерии оценки результатов самостоятельной работы.**

Критериями оценки результатов внеаудиторной самостоятельной работы студентов являются:

- уровень освоения учебного материала;

- уровень умения использовать теоретические знания при выполнении практических задач;

- уровень сформированных обще учебных умений;

- уровень умения активно использовать электронные образовательные ресурсы, находить требующуюся информацию, изучать ее и применять на практике;

- обоснованность и четкость изложения материала;

- оформление материала в соответствии с требованиями стандарта оформления;

- уровень умения ориентироваться в потоке информации, выделять главное;

- уровень умения четко сформулировать проблему, предложив ее решение, критически оценить решение и его последствия;

- уровень умения определить, проанализировать альтернативные возможности, варианты действий;

- уровень умения сформулировать собственную позицию, оценку и аргументировать ее.

**Критерии оценки учебного конспекта:**

«Отлично» - Учебный материал описан полностью и логично по плану. Текст конспекта записан аккуратно и грамотно, при необходимости, проиллюстрирован. Текст конспекта составлен самостоятельно

«Хорошо» - Учебный материал описан не полностью или не логично, но по плану. Текст конспекта записан не аккуратно или не грамотно, при необходимости, проиллюстрирован. Текст конспекта составлен самостоятельно

«Удовлетворительно» - Учебный материал описан не полностью или не логично, но по плану. Текст конспекта записан не аккуратно или не грамотно. Нет необходимых, иллюстраций. Текст конспекта составлен самостоятельно

«Неудовлетворительно» Учебный материал описан не полностью или не логично, но по плану. Текст конспекта записан не аккуратно или не грамотно. Нет необходимых, иллюстраций. Текст конспекта составлен не самостоятельно.

**Критерии оценки составления отчета по лабораторным и практическим работам:**

|  |  |
| --- | --- |
| отлично – | Отчет оформлен согласно требованиям ГОСТ 7.32-2001 и содержит выполненное практическое задание и ответы на контрольные вопросы. Выполнение работы описано полно, точно, подробно. Ответы на контрольные вопросы даны полные и, если нужно, проиллюстрированы примерами. |
| хорошо– | Отчет оформлен согласно требованиям, ГОСТ 7.32-2001, но имеет незначительные нарушения и содержит выполненное практическое задание и ответы на контрольные вопросы. Выполнение работы описано полно, точно, не достаточно подробно. Ответы на контрольные вопросы даны полные, но не всегда, проиллюстрированы примерами. |
| удовлетворительно– | Отчет оформлен согласно требованиям, ГОСТ 7.32-2001, но имеет незначительные нарушения и содержит выполненное практическое задание и ответы на контрольные вопросы. Выполнение работы описано недостаточно полно и точно, не достаточно подробно. Ответы на контрольные вопросы даны не полные, и не всегда, проиллюстрированы примерами. |
| неудовлетворительно– | Отчет оформлен согласно требованиям, ГОСТ 7.32-2001, но имеет значительные нарушения и содержит выполненное практическое задание в неполном объеме и ответы не на все контрольные вопросы. Выполнение работы описано недостаточно полно и точно, не достаточно подробно. Ответы на контрольные вопросы даны не полные, и не всегда, проиллюстрированы примерами |

# Тематический план и содержание самостоятельной работы

| №  п/п | Название темы | Вид деятельности обучаемого | Формируемые ПК + ОК | Объем часов | Форма контроля |
| --- | --- | --- | --- | --- | --- |
| 1 | Способы записи алгоритмов | Составление тезисного конспекта | ОК.1 – ОК.9  ПК 1.1. ПК 3.1 | 2 | Оценка составленного конспекта |
| 2 | Компоненты среды. Общие свойства компонентов. | Составление тезисного конспекта | ОК.1 – ОК.9  ПК 1.1, ПК 1.2 | 6 | Оценка составленного конспекта |
| 3 | Типы данных. | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.2  ПК3.1 | 4 | Оценка составленного конспекта |
| 3 | Выражения и операции | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.2 | 6 | Оценка составленного конспекта |
| 4 | Особенности применения инструкций цикла. Вложенные циклы | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 6 | Оценка составленного конспекта Проверка отчета |
| 5 | Примеры использования основных и специальных классов. | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.2 | 6 | Оценка составленного конспекта |
| 6 | Структура фрагмента программы обработки исключений | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 4 | Оценка составленного конспекта  Проверка отчета |
| 7 | Операции над множествами | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.2 | 4 | Оценка составленного конспекта |
| 8 | Динамический массивВыделение и освобождение динамической памяти | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.2 | 4 | Оценка составленного конспекта Проверка отчета |
| 9 | Процедуры и функции для работы с файлами. | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 6 | Оценка составленного конспекта Проверка отчета |
| 10 | Библиотеки подпрограмм. Рекурсивные подпрограммы | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 6 | Оценка составленного конспекта Проверка отчета |
| 11 | Программирование графики. Вывод текста | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 10 | Оценка составленного конспекта Проверка отчета |
| 12 | Диаграммы | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 4 | Оценка составленного конспекта Проверка отчета |
| 13 | Элементы управления. Отображение заголовков | Составление тезисного конспекта | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 8 | Оценка составленного конспекта Проверка отчета |
| 14 | Организация взаимодействия форм. Создание многостраничных окон. | Составление тезисного конспекта Составление отчета по лабораторной работе | ОК.1 – ОК.9 ПК1.1 - ПК1.5  ПК 3.1 | 6 | Оценка составленного конспекта Проверка отчета |
| Всего часов самостоятельной работы студентов по дисциплине: | | | | 82 |  |

# **Тема.** Способы записи алгоритмов

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 словесная и словесно-формульная форма записи алгоритма

2 запись алгоритма в виде псевдокода

3 запись алгоритма на языке программирования

**Вопросы для самоконтроля:**

1. Что общего и в чем различия представления алгоритма в словесной и словесно- формульной форме
2. Что представляет собой псевдокод

**Теоретический материал**

Название **"алгоритм"** произошло от латинской формы имени величайшего среднеазиатского математика **Мухаммеда ибн Муса ал-Хорезми** (Alhorithmi), жившего в 783—850 гг. В своей книге "Об индийском счете" он изложил правила записи натуральных чисел с помощью арабских цифр и правила действий над ними "столбиком", знакомые теперь каждому школьнику. В XII веке эта книга была переведена на латынь и получила широкое распространение в Европе.

Человек ежедневно встречается с необходимостью следовать тем или иным правилам, выполнять различные инструкции и указания. Например, переходя через дорогу на перекрестке без светофора надо сначала посмотреть направо. Если машин нет, то перейти полдороги, а если машины есть, ждать, пока они пройдут, затем перейти полдороги. После этого посмотреть налево и, если машин нет, то перейти дорогу до конца, а если машины есть, ждать, пока они пройдут, а затем перейти дорогу до конца.

В математике для решения типовых задач мы используем определенные правила, описывающие последовательности действий. Например, правила сложения дробных чисел, решения квадратных уравнений и т. д. Обычно любые инструкции и правила представляют собой последовательность действий, которые необходимо выполнить в определенном порядке. Для решения задачи надо знать, что дано, что следует получить и какие действия и в каком порядке следует для этого выполнить. Предписание, определяющее порядок выполнения действий над данными с целью получения искомых результатов, и есть алгоритм.

|  |
| --- |
| **алгоритм — заранее заданное понятное и точное предписание возможному исполнителю совершить определенную последовательность действий для получения решения задачи за конечное число шагов.** |

Это — не определение в математическом смысле слова, а, скорее, **описание** интуитивного понятия алгоритма, раскрывающее его сущность.

Свойства алгоритма

**1.   Понятность** для исполнителя — исполнитель алгоритма должен понимать, как его выполнять. Иными словами, имея алгоритм и произвольный вариант исходных данных, исполнитель должен знать, как надо действовать для выполнения этого алгоритма.

**2.   дискретность** (прерывность, раздельность) — алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов (этапов).

**3.   Определенность** — каждое правило алгоpитма должно быть четким, однозначным и не оставлять места для пpоизвола. Благодаpя этому свойству выполнение алгоpитма носит механический хаpактеp и не тpебует никаких дополнительных указаний или сведений о pешаемой задаче.

**4.   Pезультативность** (или конечность) состоит в том, что за конечное число шагов алгоpитм либо должен пpиводить к pешению задачи, либо после конечного числа шагов останавливаться из-за невозможности получить решение с выдачей соответствующего сообщения, либо неограниченно продолжаться в течение времени, отведенного для исполнения алгоритма, с выдачей промежуточных результатов.

**5.   Массовость** означает, что алгоpитм pешения задачи pазpабатывается в общем виде, т.е. он должен быть пpименим для некотоpого класса задач, pазличающихся лишь исходными данными. Пpи этом исходные данные могут выбиpаться из некотоpой области, котоpая называется областью пpименимости алгоpитма.

4. Формы записи алгоритмов.

Формы записи алгоритма

На практике наиболее распространены следующие формы представления алгоритмов:

* [**словесная**](E:informaticatheorychapter71_7_5.html) **и словесно-формульная** (запись на естественном языке);
* [**графическая**](E:informaticatheorychapter71_7_6.html) (изображения из графических символов);
* [**псевдокоды**](E:informaticatheorychapter71_7_7.html) (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
* [**программная**](E:informaticatheorychapter71_7_12.html) (тексты на языках программирования).

|  |
| --- |
| **Словесный способ** записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке. |

Например. Записать алгоритм нахождения **наибольшего общего делителя (НОД)** двух натуральных чисел (алгоритм Эвклида).

Алгоритм может быть следующим:

1. задать два числа;
2. если числа равны, то взять любое из них в качестве ответа и остановиться, в противном случае продолжить выполнение алгоритма;
3. определить большее из чисел;
4. заменить большее из чисел разностью большего и меньшего из чисел;
5. повторить алгоритм с шага 2.

Описанный алгоритм применим к любым натуральным числам и должен приводить к решению поставленной задачи. Убедитесь в этом самостоятельно, определив с помощью этого алгоритма наибольший общий делитель чисел 125 и 75.

Словесный способ не имеет широкого распространения, так как такие описания:

* строго не формализуемы;
* страдают многословностью записей;
* допускают неоднозначность толкования отдельных предписаний.

Иногда используется разновидность словесной формы описания – **словесно – формульное** представления алгоритма, где словесное описание уточняется расчетными формулами.

**Графический** способ представления алгоритмов является более компактным и наглядным по сравнению со словесным.

|  |
| --- |
| **При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий.** |

Такое графическое представление называется схемой алгоритма или **блок-схемой**.

**Псевдокод**

|  |
| --- |
| Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов. |

Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой строны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

Примеры подобрать самостоятельно

# **Тема.** Компоненты среды. Общие свойства компонентов.

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Общие свойства компонент

2 Свойства компонента Форма

3 Свойства простых объектов

**Вопросы для самоконтроля:**

1Чем свойство **name** отличается от свойства **caption**

2 С помощью какого свойства можно сделать компонент невидимым

3 Опишите основные свойства формы

4 Каким будет состояние окна в момент запуска приложения

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

**Визуальные компоненты имеют следующий ряд общих свойств:**

**Left и Тор** - местоположение визуального компонента внутри формы.

**Width и Height** - горизонтальный и вертикальный размеры компонента соответственно.

**Anchors** - позволяет привязать границы компонента к границам формы. Привязанная граница компонента будет следовать за соответствующей границей формы при изменении размеров формы.

**BiDiMode** - позволяет сделать так, чтобы текст читался справа налево.

**Caption** - надпись компонента. Установленная в свойстве текстовая строка может содержать специальный символ (амперсант). Если в строке встречается амперсант, то следующий за ним символ отображается подчеркнутым (амперсант при этом не отображается).

**Color** - цвет компонента. Компонент либо использует свой собственный цвет, либо копирует цвет содержащего компонента.

**Cursor** - определяет, какой вид принимает указатель мыши, когда пользователь наводит его на компонент.

**Enabled** - определяет, доступен ли компонент для пользователя. Если свойство имеет значение True, то компонент доступен, а если значение False, то недоступен. Недоступный компонент обычно имеет блеклый вид.

**Font** - шрифт надписи на компоненте. Для выбора шрифта в окне свойств необходимо установить курсор напротив выбранного свойства. В диалоговом окне Выбор шрифта можно установить название, начертание и размер шрифта. Параметры шрифта можно задать с помощью вложенных свойств Color, Name, Size, Style и др.Hint ~ подсказка, появляющаяся над компонентом, когда пользователь временно задерживает над ним указатель мыши. Появление подсказки может быть разрешено или запрещено с помощью свойства ShowHint.

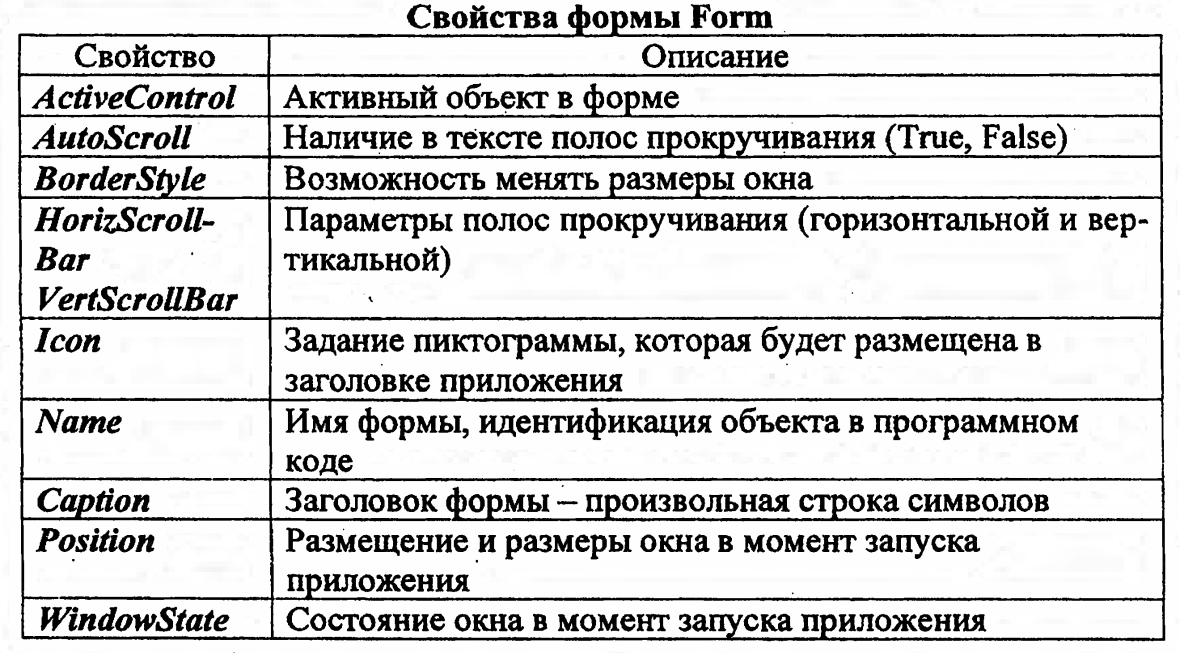
**РорuрМеnu** - используется для привязки контекстного меню к компоненту. Это меню вызывается щелчком правой кнопки мыши по компоненту (практическая работа 4).

**TabStop** - определяет, может ли компонент получать фокус ввода. Если свойство имеет значение True, то компонент находится в очереди на фокус ввода, а если значение False, то - нет. Свойство TabStop присутствует только в оконных компонентах.

**Visible** - определяет видимость компонента на экране. Если свойство имеет значение True, то компонент виден, а если значение False, то - не виден. Окно формы. Работа с формой

Форма - это окно Windows. Форма является контейнером объектов, в то же время форма сама является объектом. Все внутреннее пространство формы называется рабочей областью, на которую условно нанесена сетка выравнивания для удобного расположения на ней компонентов. При размещении объекта на форме основные пара¬ метры объекта сохраняются в исполняемом программном коде. При выполнении групповых операций для нескольких компонентов их29можно объединять с помощью указателя. Основные свойства формы представлены в табл. 3.

Таблица 3



**Состояние окна в момент запуска приложения**

Другая возможность для выравнивания позволяет задавать определенные числовые значения свойствам Left и Тор.В момент выполнения окно приложения появляется перед пользователем в том месте экрана, в котором проектировалась форма. Чтобы окно располагалось в центре экрана, необходимо в инспекторе объектов найти свойство Position. По умолчанию оно имеет значение ро Designed - спроектированное. Можно изменить это значение на ро Screen Center - центр экрана. Чтобы в процессе манипуляций с проектом невозможно было испортить окно приложения, можно запретить изменять размер окна. Для этого свойство Border Style со значением bs Sizeable (окно с изменяемыми размерами) изменить на значение bs Dialog.

Таким образом, окно будет сохранять те пропорции, которые были заданы в процессе проектирования. Для привязки компонента к определенному положению на форме используется свойство Anchors. По умолчанию привязка осуществляется к левому и верхнему краям родительского компонента.

Тор - верхний правый край привязан к верхнему краю родительского компонента.

Left - Верхний левый край привязан к левому краю родительского компонента.

Right - Правый край привязан к правому краю родительского компонента.

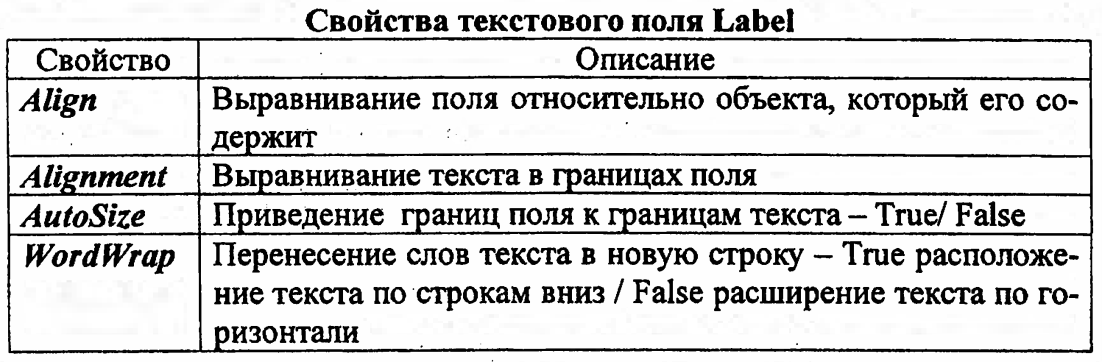
Button - Нижний край привязан к нижнему краю родительского компонента.

**Свойства простых объектов**

**Надпись**

Визуальный элемент управления - надпись или метка - Label используется для размещения на форме текстовой информации, которая содержится в специальных окнах типа надпись. Текст надписи задается свойством Caption. Это свойство можно установить в окне Properties или в программном коде. Шрифт надписи определяется свойством Font. Оформление надписи можно изменить в зависимости от значения свойств, некоторые представлены в табл. 4.

Таблица 4



Текст, который задан свойством Caption, может иметь достаточно большой размер и занимать несколько строк. Если надпись имеет не¬ большой и известный размер, то размер можно изменять с помощью мыши, клавиш управления курсором при нажатой клавише Shift, свойств Height и Width в окне свойств.

Если размер текстового объекта заранее неизвестен, то можно использовать свойства Autosize и Wordwrap. Если для свойства AutoSize установлено значение True, то происходит расширение площади для размещения текста, который вводится в свойстве Caption.

При AutoSize = False размер метки не зависит от длины текста, введенного в Caption. Если длина текста превышает длину объекта Label, часть информации, не поместившаяся в объект, будет не видна.

Если установлено Wordwrap = True, то происходит расширение надписи вниз по строкам и изменяется высота объекта Label, чтобы поместить весь текст. Ширина надписи остается постоянной. При этом осуществляется автоматический перенос слов. Установка значения Wordwrap = False размер объекта Label изменяется только в горизонтальном направлении.

**Текстовое поле**

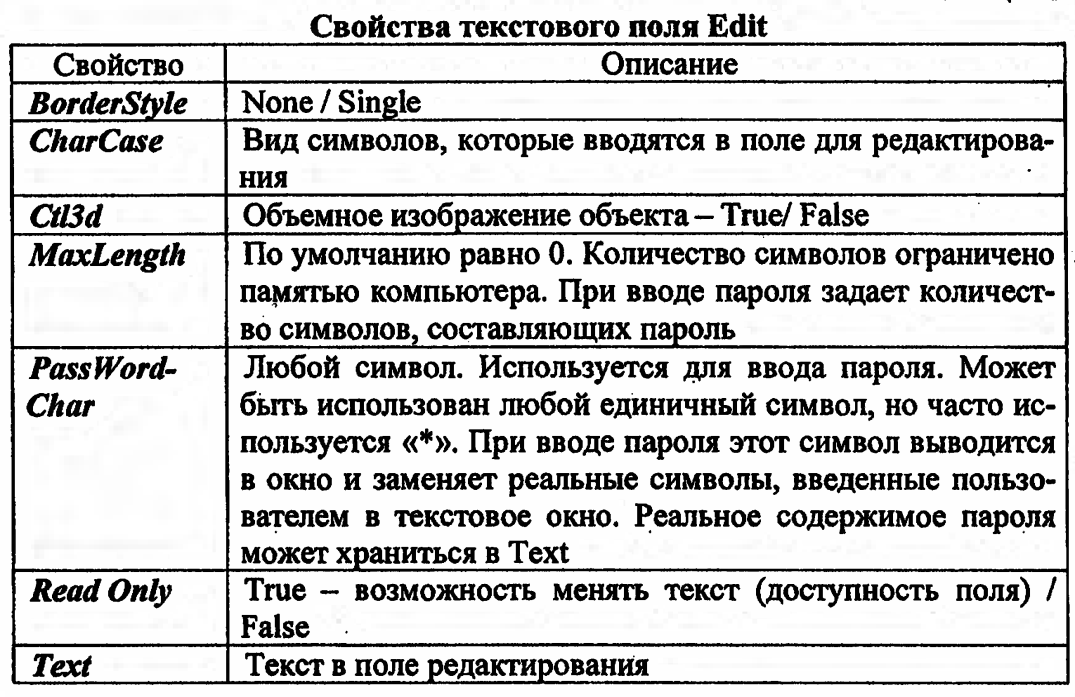
Визуальный элемент управления - текстовое окно или поле ввода- Edit, помещенный на форму, применяется для ввода пользователем данных во время работы приложения или вывода данных (табл. 5). Заполняется либо в программном коде, либо на этапе проектирования приложения. Если объект активизирован, то в текстовом окне появляется курсор для ввода данных. Числовые данные будут введены в текстовом формате, поэтому для выполнения вычислений их следует привести к числовому формату.

Текстовые окна прежде всего предназначены для ввода информации пользователями приложения. Пользователь может только просматривать данные без возможности их редактирования. В этом случае изменение информации в текстовом поле может выполняться только в программном коде.

В режиме выполнения приложения один из объектов на форме находится в фокусе или активизирован. Такой объект - текстовое окно, доступно для ввода с клавиатуры, курсор находится внутри текстового окна и оно готово для ввода текста. В результате ввода или просмотра информации курсор может перемещаться в пределах окна.

В случае, если фокус впервые переходит в текстовое окно, курсор , по умолчанию устанавливается слева от находящегося в поле текста. При последующем возвращении фокуса на поле курсор устанавливается в то место, куда он был установлен в последний раз.

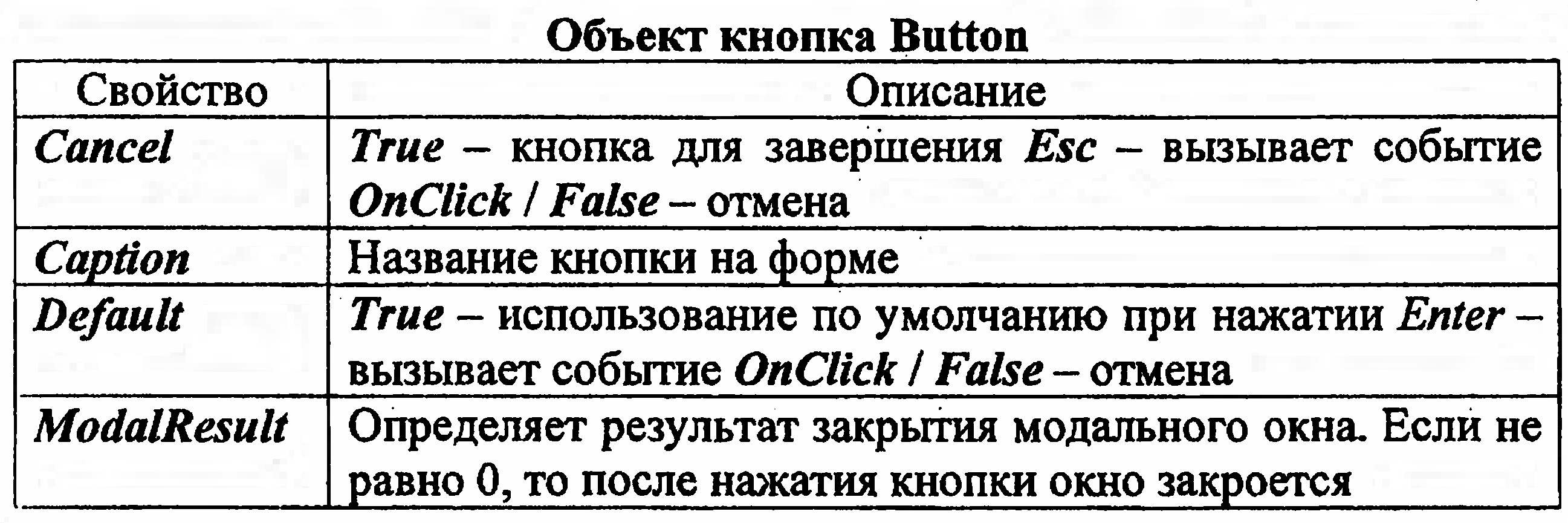
Таблица 5



**Командная кнопка**

Командная кнопка Button применяется для выполнения на форме запрограммированных действий, позволяет одним кликом запустить '« некоторый процесс (табл. 6).При щелчке по кнопке инициируется событие Click для данной кнопки и выполняется процедура, которая написана пользователем для данного события. Это может быть, например, выполнение некоторых вычислений. Кнопка всегда содержит надпись, которая задается свойством Caption и обозначает назначение кнопки. Количество символов в названии не может быть более 255. Если длина надписи больше ширины кнопки, автоматически осуществляется перенос надписи на следующую строку. Если размер кнопки не позволяет поместить всю надпись, то остальная часть отсекается.

Таблица 6



После того как кнопка помещена на форму и имеет название, следует определить действия, которые реализуются при нажатии на эту кнопку. Для этого необходимо кликнуть два раза по кнопке и в открывшемся окне редактора кода задать заготовку процедуры (заголовок процедуры, содержащий событие Click, и последний оператор процедуры), соответствующую этой кнопке.

Часто на форме располагаются две кнопки: одна для активизации некоторых действий (Ок), другая - для отмены каких-либо действий (Cancel).При работе с клавиатурой подобные действия традиционно выполняют клавиши Enter и Esc. Чтобы задействовать применение этих клавиш, можно использовать свойства Default и Cancel для командных кнопок.

Поскольку оба эти свойства могут принимать одно из двух значений - True / False, то во избежание проблем и путаницы при назначении кнопок в Delphi предусмотрены следующие правила:

1. Свойство Default или Cancel могут быть установлены как True только для одной кнопки на данной форме.

2. Одна из кнопок может иметь одновременно только одно значение True либо свойство Default, либо свойство Cancel.

# **Тема.** Типы данных

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Классификация типов данных

2 Описание типов данных

3 Форма представления вещественных чисел

4 Специальные управляющие символы

5 Порядковые типы

**Вопросы для самоконтроля:**

1 Поясните классификацию типов данных

2 С какой целью используются типы данных

3 Какие типы данных называют порядковыми и почему

4 Чем отличаются формы представления вещественных чисел

5 С какой целью используется символ #

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

*Тип* определяет множество значений, которые могут принимать объекты программы (константы и переменные), а также совокупность операций, допустимых над этими значениями.

Например, значения 1 и 3 относятся к целочисленному типу, и над ними можно выполнять любые арифметические операции. Значения «отличная» и «учеба» принадлежат к строковому типу и над ними можно выполнять только одну операцию — *склеивания, сцепления, или конкатенации* текста (обозначается через +).

Все типы данных, используемые в Turbo Pascal, можно разделить на две большие группы: скалярные (простые) и структурированные (составные). *Скалярные типы* в свою очередь подразделяются на стандартные и пользовательские (перечисляемый и интервальный). *Стандартные типы* предлагаются программисту разработчиками Turbo Pascal. К ним относятся: целочисленные, вещественные, символьный (литерный), логический (булевский) и указатели. *Структурированные типы* имеют в своей основе скалярные типы данных. К структурированным относятся: строки, массивы, множества, записи и файлы.

Целочисленные типы, символьный, логический и пользовательские типы данных (перечисляемый и интервальный) образуют группу так называемых *порядковых типов,* имеющих большое значение.

Тип данных очень важен при выделении памяти под переменные, поскольку каждому типу соответствует строго определенный размер ячейки памяти. В любом случае этот размер ограничен, следовательно, *все типы данных имеют ограниченный диапазон значений* Этот факт не согласуется с нашими математическими представлениями о числовых множествах. Тем не менее, с ним приходится считаться.

**Стандартные типы**

*Целые* и *вещественные* типы предназначены для представления числовых данных. В математике рассматривается *бесконечное* множество целых чисел. Целый тип в языке Turbo Pascal — это *интервал* целых чисел . Операции над целыми числами (см. табл. 3.1) определены лишь тогда, когда исходные данные *(операнды)* и результат лежат в этом интервале. Иначе возникает ситуация, называемая *переполнением.* За исключением переполнения все операции над аргументами целого типа выполняются *точно .*

***Таблица 1-*** *Целочисленные типы данных*

|  |  |  |
| --- | --- | --- |
| **Название целого типа** | **Диапазон возможных значений** | **Память, байт** |
| byte (байтовый) | 0—255 | *1* |
| shortint (короткий целый) | -128—127 | *1* |
| integer (целый) | -32 768-32 767 | 2 |
| word (слово) | 0—65 535 | 2 |
| longint (длинный целый) | -2 147 483 648-2 147 483 647 | *4* |

В математике вещественные числа — это *бесконечное непрерывное* множество чисел. В вычислительных машинах вещественные числа представляются *конечным* множеством значений (табл. 2).

Например, внутреннее представление типа real может дать 248 = = 281 474 976 710 656 (более чем 10) возможных комбинаций значащих разрядов в отведенных для него 6 байтах, или 48 битах. Это очень большое число, но все же оно не сопоставимо с множеством вещественных чисел.

***Таблица 2-*** *Вещественные типы данных*

|  |  |  |  |
| --- | --- | --- | --- |
| **Название вещественного типа** | **Диапазон возможных значений (плюс-минус)** | **Количество значащих цифр** | **Память, байт** |
| single(с одинарной точностью) | 1,5е-45—3,4е38 | 7—8 | 4 |
| real (вещественный) | 2,9е-39—1,7е38 | 11—12 | 6 |
| Double (с двойной точностью) | 5,Ое-324-1,7е308 | 15-16 | 8 |
| Extended (с повышенной точностью) | 3,4e-4932-1,1e4932 | 19-20 | 10 |
| соmр (сложный) | -2е63+1— 2е63-1 | 19—20 | *8* |

*Логический (булевский)* тип имеет всего два значения: true (да — истина, 1) и false (нет — ложь, 0), причем данные значения упорядочены, т. е. в опе­рациях сравнения true > false (табл. 3).

*Символьный (литерный) и строковый* типы представляют данные, являющиеся *символами* и их последовательностями — *строками* (см. табл.3). В памяти компьютера символы хранятся в виде их *числовых кодов.* Числовые коды преобразуются в буквы и другие символы лишь в момент их вывода на экран или принтер. Соответствие между символом и его кодом задается при помощи *кодовой таблицы,* которая находится в памяти компьютера и используется при выводе символов

***Таблица 3-*** *Символьный и логический (булевский) типы данных*

| **Тип** | **Диапазон возможных значений** | **Память, байт** |
| --- | --- | --- |
| char (символьный, литерный) | Символы кодовой таблицы | 1 |
| boolean (булевский) | true, false | 1 |

Переменные, описываемые любым из типов byte, shortint, integer, **wortL** longint, принимают только целые значения. Типы byte, word — **беззнаковые.**

Переменные, описываемые любым из типов single, real, double, extended, comp принимают только вещественные значения — положительные и отрицательные.

**Формы записи вещественных чисел**

Вещественные числа могут записываться двумя способами - в общепринятой и экспоненциальной форме. Общепринятая форма предполагает запись по обычным правилам арифметики. Целая часть от дробной отделяется *десятичной точкой,* а не запятой, как в математике. Если точка отсутствует число считается *целым.* Запись вещественного числа *в экспоненциальной форме* (в форме с *мантиссой* и *порядком)* использует степень десяти и удобна для записи очень больших и очень маленьких чисел. При этом число изображается так: пишется мантисса, знак умножения опускается, вместо основания 10 пишется буква *е,* а следом указывается порядок (показатель степени). буква *е,* предшествующая порядку, читается как "умножить на 10 в степени".

Например, 123,456 или -11,9 - общепринятая форма, а 5.18е+02 (518) или 10е-03 (0,01) — экспоненциальная.

Любое вещественное число хранится в памяти компьютера в экспоненциальной форме: отдельно — мантисса и отдельно — порядок. При этом под мантиссу и порядок отводится строго определенное количество двоичных разрядов.

**Запись символов. Специальные и управляющие символы**

В том случае, если в программе требуется использовать значение символьной переменной или константы, его необходимо заключить в апострофы или записать с использованием знака #, за которым следует код символа.

Например, 'А' обозначает букву А, ';' — точку с запятой, ' ' — пробел, #32 или #$20 являются также символом пробела (32 — это код, соответствующий пробелу, а шестнадцатеричное число 20 равно десятичному 32).

Символьные константы упорядочены по кодам.

Рекомендуется применять # (знак номера) только для *специальных (служебных) символов,* которые не отображаются на экране и имеют мнемонические сокращения, унаследованные из прошлого. Некоторые из них могут исполь­зоваться программистом для выполнения определенных действий:

* #07 (BEL) — подача короткого звукового сигнала;
* #08 (BS) — смещение курсора на одну позицию назад;
* #09 (НТ) — горизонтальная табуляция: смещение курсора в позицию кратную 8, плюс 1 (9, 17, 25 и т. д.);
* #10 (LF) — перевод строки, курсор смещается по вертикали вниз на одну строку;
* #11 (VT) — вертикальная табуляция; - #12 (FF) — прогон страницы;
* #13 (CR) — возврат каретки или перевод строки, выполняет перемеще­ние курсора в начало следующей строки экрана (соответствует клавише <Enter>);
* #26 (SUB) — конец файла, вводится нажатием комбинации клавиш <Ctrl>+<Z>;
* #27 (ESC) — конец работы, символ соответствует клавише <Esc>;
* -#32 (BL) — пробел и т. д.

**Запись строк символов**

последовательность символов, заключенная в апострофы, является строкой иотносится к типу string. Причем сами апострофы не входят в состав троки, а лишь указывают на то, что все заключенные в них символы следу-т рассматривать как единое целое — *строковую константу.* Если в состав

**Например,** 'Язык программирования Turbo Pascal, '123451, 'А+В'. Болееподробно строки и действия над ними будут рассматриваться далее *(см. гл. 9).*

**Порядковые типы**

Следующие типы данных — целые, символьный и логический имеют ограниченное количество значений, идущих по порядку, поэтому эти типы принято называть *порядковыми типами.* Общим для них является то, что в компьютере они представляются *целым* числом.

*Все вещественные типы данных не являются порядковыми.*

В Turbo Pascal имеются два дополнительных пользовательских порядковых типа:

* интервальный (ограниченный) тип или диапазон;
* перечисляемый тип.

Они используются для того, чтобы еще больше ограничить количество значений, принимаемых переменными этого типа.

*Интервальный тип* задается своим минимальным и максимальным значениями и может быть определен на основе любого порядкового типа:

Минимальное Значение.. Максимальное значение

Например: 1. .12 (номер месяца может принимать значения от 1 до 12) или 'а1.. • z' (буквы латинского алфавита — от а до z).

*Перечисляемый тип* ограничен больше, он задается перечислением своих значений.

Например, в виде *строковых констант:* color=(red,blue,green,black).

# **Тема.** Выражения и операции

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Порядок записи арифметических выражений

2 Арифметические операции

3 Арифметические процедуры и функции

**Вопросы для самоконтроля:**

1 Поясните правила записи арифметических выражений

2 Какие функции позволяют округлять числа

3 Какие виды арифметического деления существуют

4 Какими арифметическими процедурами можно заменить операции сложения и вычитания

5 Поясните алгоритм работы генератора случайных чисел

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Рассмотрим арифметические выражения, т. к. именно с их помощью выполняются все вычисления в программе.

Результатом *арифметического выражения* является целое или вещественное значение. *Выражение* задает порядок действий над элементами данных и состоит из:

* операндов (констант, переменных, функций);
* круглых скобок;
* знаков операций.

**Арифметические операции**

*Операции* определяют действия, которые надо выполнить над операндами. В отличие от традиционной математической записи обязательно указывать все знаки операций.

Например, в выражении (х+у) \*5-10 операндами являются переменные х и у, а также константы 5 и 10; а +, \* — знаки арифметических операций сложе­ния и умножения соответственно. Символ операции умножения — \* (звездочка) должен присутствовать в явном виде. Если все объекты, входящие в выражение, определены в момент их использования (в нашем примере — это переменные х и у), то значение выражения считается определенным.

В простейшем случае выражение может состоять из одной переменной или константы. Круглые скобки ставятся так же, как и в обычных математических выражениях для *управления порядком выполнения операций*. *Унарные* операции, такие как смена знака, относятся к одному операнду, *бинарные* — связывают два.

Например, -а — унарная операция, а+b — бинарная. При использовании двух знаков операций нежелательно, чтобы они стояли рядом: а\*-b. Лучше заключить второй операнд в скобки: а\*(-b).

В табл. 3. 1 приведены сведения об основных арифметических операциях. Зля упрощения использованы только два основных числовых типа: *integer* и *real*.

**Таблица 3. 1.** Основные арифметические операции, типы операндов и результата

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Знак** | **Тип** | |
|  |  | **Операндов** | **Результата** |
| *Бинарные операции* | | | |
| Сложение | + | real | real |
| integer | integer |
| Вычитание | - | real | real |
| integer | integer |
| Умножение | \* | real | real |
| integer | integer |
| Деление | / | integer | real |
| real | real |
| Целочисленное деление | Div | integer | integer |
| Остаток от деления | Mod | integer | integer |
| Арифметическое И | And | integer | integer |
| Побитовый сдвиг влево | Shi | integer | integer |
| Побитовый сдвиг вправо | Shr | integer | integer |
| Арифметическое ИЛИ | Or | integer | integer |
| Ариф-е побитовое сложение по модулю 2 | Xor | integer | integer |
| *Унарные операции* | | | |
| Сохранение знака | + | real | real |
| integer | integer |
| Отрицание знака | - | real | real |
| integer | integer |
| Арифметическое отрицание | Not | integer | integer |

Операции DIV и MOD

Целочисленное деление div (от *division*, деление) отличается от обычной опе­рации деления тем, что возвращает целую часть частного, а дробная часть отбрасывается — 13 div 3 = 4, а не 4,(3). Результат div всегда равен нулю, если делимое меньше делителя.

Например: х

11 div 5=2 10 div 3=3 2 div 3=0 123 div 4 = 30

17 div -5 = -3 -17 div 5 = -3 -17 div -5=3

*Взятие остатка от деления* mod (от *modulus*, мера) вычисляет остаток, полу­ченный при выполнении целочисленного деления.

Например:

10 mod 5=0 11 mod 5=1 10 mod 3=1 14 mod 5=4

22 mod 5=2 31 mod 16 = 15 17 mod -5=2 -17 mod 5 = -2

-17 mod -5 = -2

Аргументы операций *div* и *mod* — целые числа. Взаимосвязь между операциями div и mod проста. Для а>о и b>о справедливо:

a mod b = а — (a div b) \* b (a div b) \* b + (a mod b) = a

*Обратите внимание* — операцию mod можно использовать, чтобы узнать, кратно ли целое а целому b. А именно, а кратно b тогда и только тогда, когда *a mod b = 0* .

**Арифметические процедуры и функции**

В арифметических выражениях часто используются следующие стандартные функции (табл. 3.2).

Таблица 3.2. Некоторые стандартные функции, типы значений аргумента и результата

|  |  |  |  |
| --- | --- | --- | --- |
| **Стандартная функция** | **Выполняемое действие** | **Тип** | |
| **аргумента** | **Результата** |
| abs(x) | | x | | real | real |
| integer | integer |
| sqr(x) | X | real | real |
| integer | integer |
| sqrt(x) |  | real | real |
| integer | real |
| exp(x) |  | real | real |
| integer | real |
| ln(x) | ln(x) | real | real |
| integer | real |
| pi | число пи | - | real |
| sin(x) | sin(x) | real | real |
| integer | real |
| cos(x) | cos(x) | real | real |
| integer | real |
| arctan(x) | arctg(x) | real | real |
| integer | real |

Вызов стандартной функции осуществляется путем указания в нужном месте программы имени функции (abs, in, exp и др.) и ее аргумента, заключенного в круглые скобки. После вычисления значения функции ее вызов заменяется результатом, и расчет содержащего ее выражения продолжается дальше.

*Следует знать*:

- аргумент прямых тригонометрических функций sin и cos задается в радианах. Для преобразования значения угла из радианной меры в градусную необходимо умножить величину угла на число pi/180. Для перевода значения угла из градусной меры в радианную необходимо умножить величину угла на число pi/180;

- результат функции arctan получается в радианах.

Кроме приведенных в табл. 3.2 также используются следующие стандартные процедуры и функции:

- функция *random* (диапазон) возвращает случайное число х, удовлетворяющее условию о<=х<диапазон. Тип аргумента и результата — word. В том случае, если нам необходимы целые случайные числа из диапазона а<=х<b, мы можем получить их, используя выражение *random (b-а)+а*. Если параметр диапазон не указан, то random возвращает число х в диапазоне 0<=х<1. Тип результата — real. В том случае, если нам необходимы вещественные случайные числа из другого диапазона: а<=х<b, мы можем задать его при помощи *random\*b+a*. Перед первым обращением к функции random необходимо с помощью вызова процедуры randomize инициализировать программный генератор случайных чисел. В противном случае при каждом запуске программы датчик будет выдавать одни и те же числа.

- процедура \ dec (х, n) уменьшает значение целочисленной переменной х на n. Например, х: =10; dec (х, 2); {результат: 8}. При отсутствии необязательного параметра n процедура принимает вид dec(x), а значение х уменьшается на единицу;

процедура inc(x, n) увеличивает значение целочисленной переменной х на п. Например, х: =10; inc(x, 3); {результат: 13}. При отсутствии необязательного параметра п процедура принимает вид inc (х), а значение х увеличивается на единицу;

функция frac(x) вычисляет дробную часть х. Аргумент и результат — real.

Например *write(frac(0. 25\*11): 4: 2)*; {результат 0. 75};

функция int(x) вычисляет целую часть х. Аргумент и результат— real.

Например, *write(int(422.117):4:2)*; {результат 422. 00). Таким образом, x=int(x) +frac(x).Тема. Особенности применения инструкций цикла. Вложенные циклы

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Основные особенности циклов

2 Порядок выполнения операций во вложенном цикле

3 Организация досрочного выхода из цикла

**Вопросы для самоконтроля:**

1 Поясните как выбрать оператор для организации цикла

2 Каким образом организовать досрочный выход из цикла в разных типах циклов

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

С помощью условных операторов и операторов присваивания можно реализовать самый сложный алгоритм. Однако в программах, связанных с обработкой данных или вычислениями, часто выполняются циклически повторяющиеся действия.

Цикл представляет собой последовательность операторов, которая выполняется неоднократно.

В языке программирования имеется три разновидности цикла — цикл с постусловием (инструкция repeat), цикл с предусловием (инструкция while) и цикл со счетчиком (инструкция for).

***Следует знать:***

-подавляющее большинство задач с циклами можно решить разными способами, используя при этом любой из трех операторов цикла; часто решения, использующие разные операторы цикла, оказываются равноценными;

-в некоторых случаях все же предпочтительнее использовать какой-то один из операторов;

-самым универсальным из всех операторов цикла считается while, поэтому в случае затруднений с выбором можно отдать предпочтение ему; цикл repeat имеет очень простой и понятный синтаксис, поэтому с него удобно начинать изучение циклов;

-цикл for обеспечивает удобную запись циклов с заранее известным числом повторений;

-при неумелом использовании циклов любого типа возможна ситуация, когда компьютер не сможет нормально закончить цикл (в таком случае говорят, что программа "зациклилась").

**Оператор REPEAT**

Оператор повтора repeat состоит из заголовка(repeat), тела цикла и условия окончания (until). Ключевые слова repeat, until обозначают "повторяй" и "пока" соответственно.

repeat

{ Инструкции }

until Условие выхода из цикла;

Вначале выполняется тело цикла — инструкции, которые находятся между repeat и until, затем проверяется значение Условия выхода из цикла. В том случае, если оно равно false (ложь), т. е. не выполняется — инструкции цикла повторяются еще раз. Так продолжается до тех пор, пока условие не станет true (истина).

***При работе оператора:***

– число повторений операторов (инструкций) цикла repeat определяется в ходе работы программы и во многих случаях заранее неизвестно;

- инструкции цикла repeat будут выполняться, пока условие, стоящее после until, будет оставаться ложным;

- после слова until записывается условие завершения цикла;

- условие — это выражение логического типа: простое выражение отношения или сложное логическое выражение;

- для успешного завершения цикла repeat в его теле обязательно должны быть инструкции, выполнение которых влияет на условие завершения цикла, иначе цикл будет выполняться бесконечно — программа зациклится. Другими словами, переменная, которая участвует в условии выхода из цикла, обязательно должна изменяться в теле цикла.

- цикл repeat — это цикл с постусловием (условие проверяется после выполнения тела цикла), т. е. инструкции тела цикла будут выполнены хотя бы один раз.

-поэтому цикл repeat удобно использовать в тех случаях, когда тело цикла гарантированно должно выполниться хотя бы один раз;

- нижняя граница операторов тела цикла четко обозначена словом until, поэтому нет необходимости заключать эти операторы в операторные скобки begin и end.

**Оператор WHILE**

Оператор повтора while состоит из *заголовка и тела цикла.* Ключевые слова while и do обозначают "до тех пор, пока" и "выполняй" соответственно.

**while** Условие выполнения цикла **do begin**

{ Инструкции } end;

Оператор while аналогичен оператору repeat, но проверка Условие выполнения цикла производится в самом начале оператора — если значение условия равно true (истина), то *выполняются инструкции цикла,* находящиеся между *begin* и *end* и снова вычисляется выражение *Условие выполнения цикла*. Так продолжается до тех пор, пока значение *Условие выполнения цикла* не станет равно false (ложь).

***Взаимосвязь операторов while и repeat***:

\* оператор while *Условие do инструкция*; *эквивалентен* оператору

*if Условие* then repeat *Инструкция* until Not *Условие*;

***При работе оператора****:*

\* *число повторений* операторов (инструкций) цикла while определяется в ходе работы программы и, как правило, заранее неизвестно;

\* после слова while записывается условие *продолжения выполнения* инструкций цикла, в этом отличие цикла while от цикла repeat;

\* *условие —* это выражение логического типа: *простое выражение отношения* или *сложное выражение отношения (логическое выражение),* которое может принимать одно из двух значений: true или false;

\* для успешного завершения цикла while в его теле обязательно должны присутствовать инструкции, оказывающие *влияние* на условие выполнения инструкций цикла.

\* цикл while — это цикл с *предусловием,* т. е. инструкции тела цикла *вообще могут быть не выполнены,* если проверяемое условие ложно с самого начала;

\* исходя из последнего утверждения цикл while считают самым универсальным видом цикла;

\* цикл while обычно применяется в тех же задачах, что и repeat (в зависимости от личного вкуса программиста). Удобнее всего использовать его в тех случаях, когда возможны ситуации невыполнения цикла;

\* в операторе цикла while точка с запятой никогда не ставится после зарезервированного слова do.

**Оператор FOR**

Этот вид оператора цикла называют *циклом со счетчиком* или *циклом с параметром.* В нем важную роль играет *переменная-параметр,* которая на каждом шаге цикла автоматически изменяет свое значение ровно на *единицу* — поэтому ее и называют *счетчиком.*

Инструкцию for можно реализовать двумя способами.

**Вариант 1** (с увеличением счетчика).

for Счетчик := НачальноеЗначение to КонечноеЗначение do

begin

{ Инструкции }

end;

Ключевые слова for, do обозначают "для", "выполняй" соответственно. Строка, содержащая for... do, называется *заголовком цикла,* оператор, стоящий после do образует его *тело.* Очень часто тело цикла — составной оператор. Если тело цикла представлено одиночным оператором, то begin и end не пишутся.

Инструкции между begin и end выполняются столько раз, сколько определяет выражение [(КонечноеЗначение - НачальноеЗначение) + 1].

Это соответствует всем значениям счетчика от начального до конечного включительно.

Если НачальноеЗначение больше, чем КонечноеЗначение, то инструкции между begin и end не выполняются ни разу

**Вариант 2** (с уменьшением счетчика).

for Счетчик := НачальноеЗначение downto КонечнооЗцачение do

begin

{ Инструкции }

end;

Инструкции между begin и end выполняются столько раз, сколько определяет выражение [(НачальноеЗначение — КонечноеЗначение) + 1].

Если НачальноеЗначение меньше, чем КонечноеЗначение, то инструкции между begin и end не выполняются ни разу.

Например, выполнение цикла — фрагмента программы:

*Обратите внимание —* цикл for удобно использовать для организации вывода данных программы.

***При работе оператора:***

\* оператор (инструкция) for используется для организации циклов с *фиксированным,* заранее известным или определяемым во время выполнения программы числом повторений;

\* *количество повторений* цикла определяется *начальным и конечным значениями переменной-счетчика.* Оператор for обеспечивает выполнение тела цикла до тех пор, пока не будут перебраны *все значения* параметра цикла от начального до конечного;

\* переменная-счетчик должна быть *порядкового типа-* чаще — integer, реже — char, boolean или одного из пользовательских типов*. Использование вещественного типа недопустимо;*

*\* начальное* и *конечное* значения параметра цикла могут быть константами переменными, выражениями и должны принадлежать к *одному и том уже типу* данных. Начальное и конечное значение *нельзя изменять* во время выполнения цикла;

\* после нормального выполнения оператора for значение параметра цикла равно конечному значению. Если оператор for не выполняется, значение параметра цикла не определено;

\* параметр цикла for может изменяться (увеличиваться или уменьшаться ) каждый раз при выполнении тела цикла только на единицу. Если нужен другой шаг изменения параметра, предпочтительнее циклы repeat и while.

***Замечание***

Для **досрочного выхода из цикла** можно использовать оператор goto или оператор break. Но лучше использовать циклы repeat и while. Процедура continue позволяет прервать выполнение любого цикла и передает управление на его заголовок.

Например, рассмотрим фрагмент программы с досрочным выходом из цикла:

…

for I:=1 to 45 do

begin

f:=f+I;

if (f>100)or (I=39) then break;

end;

…

**Вложенные циклы**

В теле любого оператора цикла могут находиться другие операторы цикла. При этом цикл, содержащий в себе другой, называют *внешним,* а цикл, находящийся в теле первого — *внутренним (вложенным).* Правила организации внешнего и внутреннего циклов такие же, как и для простого цикла. *Обратите внимание —* при программировании вложенных циклов необходимо соблюдать следующее дополнительное условие: *все операторы внутреннего цикла должны полностью располагаться в теле внешнего цикла.* Рассмотрим задачу вывода на экран таблицы умножения, решение которой предполагает применение вложенных циклов. С использованием цикла for соответствующий фрагмент программы имеет вид:

var i, j : byte;

begin

for i:=l to 10 do

for j:=l to 10 do

вывод(i,' \* ', j, ' = ',i\*j); end.

При вложении циклов внутренний цикл выполняется *полностью* от начального до конечного значения параметра, при *неизменном* значении параметра *внешнего* цикла. Затем значение параметра *внешнего* цикла изменяется на *единицу,* и опять от начала и до конца выполняется *вложенный* цикл. И так до тех пор, пока значение параметра внешнего цикла не станет больше конечного значения, определенного в операторе for внешнего цикла.

# **Тема.** Примеры использования основных и специальных классов.

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Понятие объекта, класса, класса-предка, класса-наследника

2 Наследование, инкапсуляция, полиформизм

3 Иерархия классов

**Вопросы для самоконтроля:**

1 Класс – это …?

2 Как в программе определяются методы

1. С помощью чего в Object Pascal реализуется ограничение доступа к полям объекта?

4 Как называется возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам

5 DELPHI содержит сложную иерархию классов. В начале этой иерархии стоят классы, называемые \_\_\_\_\_\_\_\_\_\_\_\_\_\_ классами.

6 Класс \_\_\_\_\_\_\_\_\_ является предком всех других классов, используемых в DELPHI.

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Любая программа в широком смысле может рассматриваться как *модель***[[1]](#footnote-1)**(воспроизведение или отражение) части реального мира (внешней среды, предметной области).

Различные образования реального мира могут быть определены единым термином: объекты. К ним относятся окружающие нас в быту предметы, технические устройства, сооружения, животные и, наконец, сами люди. Каждый объект характеризуется только присущим ему набором (множеством) параметров (свойств), а также множеством действий, которые может выполнить сам объект и которые могут быть выполнены над данным объектом другими объектами.

Ко всем объектам реального мира может быть применена единая по смыслу функция классифицирования, результатом выполнения которой является разбиение объектов на группы по различным признакам, видам и свойствам. В связи с этим можно ввести в рассмотрение понятие классов (групп) объектов. Это понятие определяет множество различных свойств объектов и множество различных действий (операций) над ними.

Любая схема (или система) классифицирования обычно строится по принципу от “общего” к “частному” и, как правило, имеет многоуровневую структуру отношений, в которой элементы высших ступеней порождают (производят) элементы низших уровней. Эти элементы имеют свои конкретные особенности и одновременно получают все основные свойства и действия, то есть “**наследуют**” их от элементов верхнего уровня. Например, реальный объект стол обладает множеством различных свойств, таких как тип (обеденный, книжный, кухонный), габариты, цвет, тип древесины и т. д. К нему применимы различные действия: стол можно изготовить, смонтировать (собрать), ремонтировать, передвигать, демонтировать и т. д. Очевидно, что конкретный стол, как объект реального мира, можно считать принадлежащим к классу столов, который определяет общие характеристики всех столов. В свою очередь класс столов можно считать порождением элемента верхнего уровня, а именно класса мебели, все свойства которого одновременно являются и свойствами класса столов. Это означает, что класс столов “наследует” свойства и действия класса мебели. Далее можно рассмотреть класс изделий, в который будет включен класс мебели. О классе изделий и классе мебели можно сделать такие же заключения, что и о классах мебели и столов.

Продолжая аналогичные рассуждения, можно создать сложную иерархическую структуру отношений классов различных объектов реального мира.

В основе ООП лежат понятия класса и его физической реализации – объекта.

Класс – это сложная структура (аналогичная типу), включающая в себя, помимо описания данных, описание подпрограмм (процедур и функций), которые могут быть выполнены над объектом, а также характеристики данных и подпрограмм.

**Данные класса называются полями, подпрограммы – методами, а характеристики данных и подпрограмм – свойствами.**

Обычно классы задаются в разделе объявления типов тех или иных модулей, и, чтобы их можно было использовать в программе и других модулях, эти объявления должны быть сделаны в интерфейсе модуля.

Класс объявляется на основе общего правила задания типов. Описание структуры класса в языке ОbjеctPascal начинается с зарезервированного слова **class**, после которого в круглых скобках указывается имя родительского класса. Если он не указан, то предполагается, что родительским является класс TОbjеct, который в ООП-модели языка Objec Pascal по умолчанию считается предком всех объявленных классов. Далее в виде отдельных строк записываются поля данных, методы и свойства. Завершается описание класса зарезервированным словом **end**. Если ни одного поля, метода или свойства у объявляемого класса нет, но указан непосредственный предок класса, зарезервированное слово **end** в конце объявления можно не писать. Последовательность записи отдельных элементов (поля, методы, свойства) класса безразлична (с учетом возможности использования одними элементами других), однако чаще всего сначала записываются поля, затем методы и, наконец, свойства.

Пример описания простого класса **[[2]](#footnote-2)**:

TTPerson=**class**

**private**

fname:string[15];

faddress:string[35];

**public**

**procedure** show;

**end**;

В приведенном примере TTPerson – это имя класса, fname и faddress – имена полей, show – имя метода.

Отдельные элементы класса могут иметь различные возможности по их использованию вне рассматриваемого класса (иметь разные области доступности или, иначе, видимости). В ООП DELPHI имеется несколько вариантов задания областей видимости, которые определяют разделы (секции) в описании класса и начинаются с ключевых слов **private**, **public**, **protected**, **published** и **automated**. Количество и порядок следования этих разделов могут быть произвольными.

Общую структуру описания класса можно представить следующим образом:

**type**

ИмяКласса = **class**(ИмяРодительскогоКласса)

Опубликованные поля, методы, свойства;

**private**

Скрытые поля, методы, свойства;

**public**

Общедоступные поля, методы, свойства;

**protected**

Защищенные поля, методы, свойства;

**published**

Опубликованные поля, методы, свойства;

**automated**

Поля, методы, свойства для OLE-объектов;

**end**;

В программе представители класса — объекты, объявляются в разделе **var**.

Например, так:

**var**

student:TTPerson;

professor:TTPerson;

Следует обратить особое внимание на то, что в ObjectPascal объект — **динамическая** структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных и задании для них начальных значений.

Выделение памяти и инициализация объекта осуществляются с помощью специального метода класса — конструктора, которому обычно присваивают имя create (создать). Чтобы подчеркнуть особую роль и поведение конструктора, вместо ключевого слова **procedure** в заголовке конструктора используется слово **constructor**.

Ниже приведено описание класса TTPerson, в состав которого введен конструктор.

TTPerson = **class**

**private**

fname:string[15];

faddress:string[35];

**constructor** create; // конструктор

**public**

**procedure**show; // метод

**end;**

Выделение памяти для данных объекта происходит присваиванием результата применения метода-конструктора к типу (классу) объекта.

Например, после выполнения оператора

professor:=TTPerson.create;

выделяется необходимая память для данных объекта professor.

Помимо выделения памяти, конструктор решает задачу присваивания полям объекта начальных значений, т. е. осуществляет инициализацию объекта.

Пример реализации конструктора для объекта TTPerson.

**constructor**TTPerson.create;

**begin**

fname:=’’;

faddress:=’’;

**end**;

После объявления и инициализации объект можно использовать, например, установить значение поля объекта. Доступ к полю объекта осуществляется указанием имени объекта и имени поля, которые отделяются друг от друга точкой.

Например, для доступа к полю fname объекта professor вместо следует писать professor.fname

Если в программе некоторый объект больше не используется, то можно освободить память, занимаемую полями этого объекта. Для выполнения этого действия используют метод-деструктор free. Например, чтобы освободить память, занимаемую полями объекта professor, достаточно записать professor.free;

**Методы** - это процедуры и функции, объявление которых включено в описание класса и которые выполняют действия над объектами класса. Объявляются и описываются они в интерфейсной части модуля внутри описания класса записывается лишь заголовок подпрограммы, а в исполнительной части – само содержание метода. Чтобы метод был выполнен, надо указать имя объекта и имя метода, отделив одно имя от другого точкой. Например, оператор - professor.show;

вызывает применение метода show к объекту professor. Фактически оператор применения метода к объекту — это специфический способ записи оператора вызова процедуры.

В программе методы определяются точно так же, как обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, состоит из двух частей: имени класса, к которому принадлежит метод, и имени метода. Имя класса от имени метода отделяется точкой.

Ниже приведен пример определения метода show описанного выше класса ТТРегson.

// метод show классаTTPerson

**procedure**ТТРегson.show;

**begin**

ShowMessage(‘Имя:’+fname+#13+'Адрес:'+faddress);

**end**;

Следует обратить внимание на то, что в операторах метода доступ к полям объекта осуществляется без указания имени объекта.

Обычно методы применяются к ранее созданным объектам программы. Такие методы можно считать методами объектов. В DELPHI существует возможность создания методов, которые можно использовать даже тогда, когда ни один объект класса не создан. К таким методам относятся, например, конструкторы, которые вызываются для создания объектов. Подобные методы называются методами класса и особым образом оформляются (за исключением конструкторов, о которых сказано выше). Объявление такого метода начинается с зарезервированного слова **class**, после которого следует обычное объявление метода, например:

TMyForm = **class**(TForm)

**class procedure**MessageCreate;

**end**;

…

**class procedure**TMyForm.MessageCreate;

**begin**

…

**end;**

Вызывается такой метод либо как метод класса, тогда перед именем метода в качестве квалификатора записывается тип класса, либо как метод объекта.

Под **инкапсуляцией** понимается скрытие полей объекта с целью обеспечения доступа к ним только посредством методов класса.

В ObjectPascal ограничение доступа к полям объекта реализуется с помощью свойств объекта. Свойство объекта характеризуется полем, хранящим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства. Метод установки значения свойства называется методом записи свойства (write), метод получения значения свойства называется методом чтения свойства (read).

В описании класса перед именем свойства записывают ключевое слово **property** (свойство). После имени свойства указываются его тип, затем имена методов, обеспечивающих доступ к значению свойства. После слова read указывается имя метода, обеспечивающего чтение свойства, после слова write —имя метода, обеспечивающего запись свойства. Ниже приведен пример описания класса TTPerson, содержащего два свойства: Name и Address.

**type**

TName=string[15];

TAddress=string[35];

TTPerson = **class**

**private**

FName:Tname; //значениесвойства Name

FAddress:TAddress; // значениесвойства Address

**Constructor**Create(Name:Tname);

**Procedure** Show;

**Function**GetName:TName;

**Function**GetAddress:TAddress;

**Procedure**SetAddress(NewAddress:TAddress);

**public**

**Property** Name: Tname

readGetName;

**Property** Address: TAddress

readGetAddress;

writeSetAddress;

**end;**

В программе для установки значения свойства необязательно записывать оператор применения к объекту метода установки значения свойства, можно записать обычный оператор присваивания значения свойству. Например, чтобы присвоить значение свойству Address объекта student, достаточно записать

student.Address:=’С.Петербург, ул. Садовая, 21, кв.3’;

Компилятор перетранслирует приведенный оператор присваивания значения свойству в оператор вызова метода:

student.SetAddress('С.Петербург, ул. Садовая, 21, кв.3');

Внешне использование свойств в программе ничем не отличается от использования полей объекта. Вместе с тем между свойством и полем объекта существует принципиальное отличие: при присвоении и чтении значения свойства автоматически вызывается процедура, которая выполняет некоторую работу.

В программе на методы свойства можно возложить некоторые дополнительные задачи. Например, с помощью метода можно проверить корректность присваиваемых свойству значений, установить значения других, логически связанных со свойством, полей, вызвать вспомогательную процедуру.

Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта, например можно разрешить только чтение. Чтобы операторы программы не могли изменить значение свойства, в описании свойства надо указать только имя метода чтения. Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку времени компиляции Вприведенном выше описании класса TTPerson свойство Name доступно только для чтения, а свойство Address — для чтения и записи.

Установить значение свойства, защищенного от записи, можно во время инициализации объекта. Ниже приведены методы класса TTPerson, обеспечивающие создание объекта класса TTPerson и доступ к его свойствам.

//конструкторобъектаTTPerson

**Constructor**TTPerson.Create(Name:TName);

**begin**

Fname:=Name;

**end;**

//метод получения значения свойства Name

**Function**TTPerson.GetName;

**begin**

Result:=FName;

**end;**

//метод получения значения свойства Address

**Function**TTPerson.GetAddress;

**begin**

Result:=FAddress;

**end;**

//метод изменения значения свойства Address

**Procedure**TTPerson.SetAddress(NewAddress:TAddress);

**begin**

**if**FAddress =’’

**then**FAddress:=NewAddress;

**end;**

Приведенный конструктор объекта TTPerson создает объект и устанавливает значение поля FName, определяющего значение свойства Name.

Операторы программы, обеспечивающие создание объекта класса TTPerson и установку его свойства, могут быть, например, такими:

student:=TTPerson.create('Иванов');

student.address:=’ул. Садовая, д.21, кв. 3’;

Концепция объектно-ориентированного программирования предполагает возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам. Такой механизм получения новых классов называется **порождением**. При этом новый, порожденный, класс (класс-потомок) наследует свойства и методы своего базового, родительского класса.

В объявлении класса-потомка указывается класс родителя. Например, класс TEmployee (сотрудник) может быть порожден от рассмотренного выше класса TTPerson путем добавления поля Department (отдел). Объявление класса TEmployee в этом случае может выглядеть так:

TEmployee = **class**(TTPerson)

FDepartment:**integer**; // номеротдела

**constructor** Create(Name:TName;Dep:**integer**);

end;

Заключенное в скобки имя класса TTPerson показывает, что класс TEmployee является производным от класса TTPerson. В свою очередь, класс TTPerson является базовым для класса TEmployee.

Класс TEmployee имеет свой собственный конструктор, который обеспечивает инициализацию класса родителя и своих полей. Пример реализации конструктора классаTEmployee:

**constructor**TEmployee.Create(Name:Tname;Dep:integer);

**begin**

**inherited** Create(Name);

FDepartment:=Dep;

**end**;

В приведенном примере директивой **inherited** вызывается конструктор родительского класса, затем присваивается значение полю класса потомка.

После создания объекта производного класса в программе можно использовать поля и методы родительского класса. Ниже приведен фрагмент программы, демонстрирующей эту возможность:

engineer:=TEmployee.create('Сидоров',413);

engineer.address:='ул.Блохина, д.8, кв.10';

Первый оператор создает объект типа TEmployee. Второй устанавливает значение свойства, которое относится к родительскому классу.

Помимо объявлений элементов класса (полей, методов, свойств) описание класса, как правило, содержит несколько директив, которые устанавливают степень видимости элементов класса в программе.

Рассмотрим семантику директив видимости.

Директива **private.**

Поля, свойства и методы, описанные в разделе **private,** называются личными или закрытыми. Сюда помещаются элементы (чаще всего поля), которые выполняют в объекте специфичные функции и которые поэтому целесообразно скрыть от других частей программы, либо такие элементы, для которых по ряду причин не следует разрешать доступ извне объекта.

Директива **public**.

Элементы, описанные в разделе **public**, называются общедоступными. Они могут быть использованы всюду в программе. Поля, свойства и методы, расположенные сразу после заголовка класса, при выключенной директиве компилятора {$M-}, по умолчанию принимаются общедоступными.

Директива **protected**.

Элементы класса, объявленные в разделе **protected**(защищенный), доступны только в классах, порожденных от исходного. Здесь размещаются элементы, которые важны лишь для функционирования объектов данного класса и его потомков. Обычно в секцию **protected** помещаются описания методов класса.

Директива **published**.

Поля, свойства и методы, описанные в разделе **published**, называются опубликованными. Их область видимости эквивалентна области видимости общедоступных описаний. Отличие состоит в том, что информация о них, за исключением ряда типов, например **real**, на этапе проектирования программы помещается в инспектор объектов. Описания, расположенные сразу после заголовка класса, при включенной директиве компилятора {$M+}, по умолчанию принимаются опубликованными.

Директива **automated**.

Элементы класса, объявленные в разделе **automated**, называются автоматическими. Их область видимости эквивалентна области видимости общедоступных описаний. Отличие состоит в том, что для автоматических свойств и методов генерируется дополнительная информация, которая используется для реализации OLE-механизма. Использовать директиву **automated** имеет смысл при объявлении потомков стандартного класса TAutoObject.

Ниже приведено описание класса TTPerson, в которое включены директивы видимости.

ТТРегsоп = class

**private**

FName:TName; { значение свойства Name }

FAddress:TAddress; ( значениесвойства Address}

**protected**

**Constructor**Create(Name:TName);

**Function**GetName:TName;

**Function**GetAddress:TAddress;

**Procedure**SetAddress (NewAddress: TAddress);

**Property**Name:TName

readGetName;

**Property**Address:TAddress

readGetAddress

writeSetAddress;

**end;**

***Полиморфизм*** — это возможность использовать одинаковые имена для методов, входящих в различные классы.

Концепция полиморфизма обеспечивает при применении метода к объекту использование именно того метода, который соответствует классу объекта.

Пусть определены три класса, один из них является базовым для двух других:

**type**

// базовыйкласс

TPerson=**class**

fname:**string**;{ имя }

**constructor** Create(name:**string**);

**function** info: **string**; **virtual;**

**end**;

// производныйотбазовогоTPerson

TStud=**class**(TPerson)

fgr:**integer**; { номергруппы }

**constructor** Create(name:**string**;gr:**integer**);

**function** info: **string**; **override**;

**end;**

// производный от базового TPerson

TProf=**class**(TPerson)

fdep:**string**; { названиекафедры }

**constructor** Create(name:**string**;dep:**string**);

**function** info: **string**; **override**;

**end**;

В каждом из этих классов определен метод info. В базовом классе с помощью директивы **virtual** метод info объявлен виртуальным. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным. В каждом дочернем классе определен свой метод info, который замещает соответствующий метод родительского класса. При этом метод порожденного класса, замещающий виртуальный метод родительского класса, помечается директивой **override**. Ниже приведено определение метода info для каждого класса.

**function**TPerson.info:**string;**

**begin**

result:=’’;

**end;**

**function**TStud.info:**string**;

**begin**

result:=fname+' гp.'+IntTostr(fgr);

**end**;

**function**TProf.info:**string**;

**begin**

result:=fname+'каф.'+fdep;

**end;**

В программе список людей можно представить массивом объектов класса TPerson, например

1ist:**array**[1..SZL] **of** TPerson; // SZL — размермассива

Здесь следует отметить, что объект list— это массив указателей. Объявить подобным образом список можно потому, что ObjectPascal позволяет указателю на родительский класс присвоить значение указателя на дочерний класс. Поэтому элементами массива list могут быть как объекты класса TStud, так и объекты класса TProf.

Вывод списка можно осуществить применением метода info к элементам массива, например, так:

st:=’’;

**for** i:=l **to** SZL **do**

**if** list[i] <> NIL **then** st:=st+info.list[i]+#13;

ShowMessage(st);

Во время работы программы каждый элемент массива может содержать как объект типа TStud, так и объект типа TProf. Концепция полиморфизма обеспечивает возможность применения к объекту именно того метода, который соответствует типу объекта.

**Классы и объекты DELPHI**

DELPHI содержит сложную иерархию классов, которые можно использовать в программе, создавая объекты этих классов или формируя классы-потомки. В начале этой иерархии стоят классы, называемые абстрактными классами. Для них нельзя создать полноценные работающие объекты, но они являются родоначальниками целых семейств классов, для которых такие объекты уже могут быть созданы. Исходя из свойства наследования, в абстрактных классах помещены характеристики, присущие всем классам их семейств.

Структура некоторых абстрактных классов, находящихся в начале иерархии классов, приведена на рисунке.**[[3]](#footnote-3)**

## TObject

## TPersistent

## TComponent

## TCanvas

## TPicture

## TGraficControl

## TWinControl

## TControl

Класс TObject является предком всех других классов, используемых в DELPHI. Он включает в себя характеристики, свойственные всем используемым классам. Некоторые методы класса TObject могут использоваться без создания соответствующих объектов с учетом того, что реального объекта такого класса может и не быть. Эти методы позволяют получить общие характеристики класса – адрес таблицы, содержащей характеристики класса, имя класса, имя предка класса, характеристики методов и т. д. Примеры некоторых методов класса TObject:

ClassName – функция класса (типа ShortString) формирует строку, содержащую имя класса, данное ему при создании;

ClassParent – функция, определяющая класс непосредственного предка данного класса;

ClassType – функция возвращает класс конкретного объекта;

InstanceSize – функция (типа Longint) возвращает размер класса или объекта в байтах;

FieldAddress(Name) – функция типа Pointerвозвращает адрес поля объекта с именем Name типа ShortString.

Класс TPersistent (Постоянный) является потомком класса TObject и предком всех классов, объекты которых могут быть помещены в память и взяты из памяти. Основными потомками класса TPersistent являются классы TComponent (Компонента) – предок всех компонент проекта; TStrings (Строки) – предок всех списков строк; TCollection (Коллекция) – коллекция (список) элементов; TGraphicObject (Графический объект), TCanvas (Канва – основа для рисования), TGraphic (Графический элемент), TPicture (Изображение) – классы, образующие так называемый графический инструментарий DELPHI.

Класс TControl является родоначальником всех элементов управления, с помощью которых выводится информация на экран и с помощью которых можно вводить информацию в программу, используя клавиатуру и мышь. Его потомок класс TWinControl служит для создания окон Windows. Класс TGraphicControl отличается от класса TWinControl отсутствием у объектов его семейства оконной функции, в связи с чем такие элементы либо служат для вывода на экран информации, либо являются чисто декоративными.

В ObjectPascal можно задавать указатель на класс, называемый метаклассом. Для этого в объявлении записываются ключевые слова **classof**, после чего указывается имя класса, ссылка на который формируется:

**type**

<имя типа>=**classof**<базовый класс>;

Примером может быть следующее описание:

**type**

TControlClass= **class of** TControl;

Этот тип является стандартным в DELPHI и именно так он определяется в модуле Controls. Введя такой тип данных, далее можно задавать переменные этого типа, например:

**var**ControlClass: TControlClass;

Значениями такой переменной могут быть классы, входящие в семейство исходного класса TControl: сам класс TControl, класс TForm, класс TButton и т. д. Таким образом, метакласс может быть указателем на любой класс, входящий в семейство исходного класса.

Введение метаклассов удобно для создания универсальных подпрограмм, которые могут работать с объектами из целого семейства классов, причем заранее неизвестно, какой из этих объектов будет использоваться. В ряде случаев это существенно упрощает создаваемую программу.

# **Тема.** Структура фрагмента программы обработки исключений

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Свойства и методы базового класса исключений

2 Два направления обработки исключений

3 Виды исключительных ситуаций

**Вопросы для самоконтроля:**

1 Какую ситуацию называют исключительной

2 Какой класс является базовым для всех исключений

3 Синтаксис обработчиков исключений

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Исключительная ситуация – это нарушение условий выполнения программы, вызывающее прерывание или полное прекращение ее работы. Обработка исключительных ситуаций состоит в нейтрализации динамической ошибки, вызвавшей ее.

Базовым классом для всех исключений является класс Exception. Объекты данного класса имеют следующие свойства и методы:

* Message – описание исключительной ситуации;
* HelpContext – номер идентификатора контекстной помощи для объекта исключения;
* Create – метод, который служит для создания объекта исключительной ситуации.

Класс Exception имеет множество потомков, каждый из которых служит для обработки определенный динамической ошибки.

Обработка исключительных ситуаций может происходить по двум направлениям:

1. Глобальная обработка исключений. Механизм ее реализуется через объект Application. При получении от системы сообщения об исключении объект Application генерирует событие OnException. Программист может создать свой глобальный обработчик. С этой целью удобно использовать компонент ApplicationEvents, для которого пишется обработчик события OnException/
2. Локальная обработка исключений. Для обеспечения возможности использования специализированных обработчиков исключений, в состав языка введены две конструкции try… finally и try…exept.

Try

//операторы, выполнение которых может вызвать ошибку

Finally

//операторы, которые должны быть выполнены в случае ошибки

End;

Данная конструкция применяется для выполнения всех необходимых действий перед передачей управления на следующий уровень обработки ошибки или глобальному обработчику. Если в любом из операторов секции try возникает исключительная ситуация, то управление передается первому оператору секции finally для выполнения всех операторов секции. Если исключительная ситуация не возникла, то последовательно выполняются все операторы обеих секций.

Try

//операторы, выполнение которых может вызвать ошибку

Except

//операторы, которые должны быть выполнены в случае ошибки

End;

Эта конструкция применяется для перехвата исключительной ситуации и предоставляет возможность ее обработки. Если в операторах секции try возникла исключительная ситуация, то управление передается первому оператору секции except. Если исключительная ситуация не возникла, то операторы секции Except не выполняются. Секция except может быть разбита на несколько частей конструкциями on…do. Это позволяет анализировать класс исключительной ситуации с целью ее обработки. При необходимости исключительную ситуацию можно вызвать программно.

Для этого используется оператор Raise, выполнение которого приводит к созданию объекта исключения класса Exception или одного из его потомков. Оператор имеет следующий синтаксис:

Raise ClassException.Method;

Процесс отладки исключительных ситуаций в Delphi имеет некоторые особенности. По умолчанию при возникновении динамической ошибки Delphi перехватывает исключительную ситуацию и выдает соответствующее сообщение. Если работу приложения надо продолжить, то необходимо выбрать из меню Run команду Run и возникшее исключение будет обработано средствами приложения.

Исключительные ситуации:

* EConvertError - происходит в случае возникновения ошибки при выполнении функций StrToInt и StrToFloat, когда конвертация строки в соответствующий числовой тип невозможна.
* EIntError - предок исключений, случающихся при выполнении целочисленных операций.
* EDivByZero - вызывается в случае деления на ноль, как результат RunTime Error 200.
* EIntOverflow - вызывается при попытке выполнения операций, приводящих к переполнению целых переменных, как результат RunTime Error 215 при включенной директиве {$Q+}.
* ERangeError - вызывается при попытке обращения к элементам массива по индексу, выходящему за пределы массива, как результат RunTime Error 201 при включенной директиве {$R+}.
* EInvalidCast - происходит при попытке приведения переменных одного класса к другому классу, не совместимому с первым (например, приведение переменной типа TListBox к TMemo).
* EListError - вызывается при обращении к элементу наследника TList по индексу, выходящему за пределы допустимых значений (например, объект TStringList содержит только 10 строк, а происходит обращение к одиннадцатому).
* EMathError - предок исключений, случающихся при выполнении операций с плавающей точкой.
* EOverflow - происходит как результат переполнения операций с плавающей точкой при слишком больших величинах. Соответствует RunTime Error 205.
* Underflow - происходит как результат переполнения операций с плавающей точкой при слишком малых величинах. Соответствует RunTime Error 206.
* EZeroDivide - вызывается в результате деления на ноль.

EMenuError - вызывается в случае любых ошибок при работе с пунктами меню для компонент TMenu, TMenuItem, TPopupMenu и их наследников

**Пример**

Необходимо заполнить таблицу числами по следующему правилу: генерируется случайное число, а затем в ячейке компонента *StringGrid1* сохраняется обратное значение. Заранее не известно, был ли сгенерирован 0, и будет ли допустима операция деления. Для корректной работы данной программы необходимо задавать защищенный блок; процедура формирования таблицы будет иметь вид:

procedure TForm1.Button1Click(Sender: TObject);

var i,j:Integer;

a:real;

begin

with StringGrid1 do

begin

ColCount:=SpinEdit1.Value+1;

RowCount:=SpinEdit2.Value+1;

for i :=1 to ColCount do

for j:=1 to RowCount do

Try {Начало блока}

a:=Random(10);

Cells[i,j]:=FloatToStr(1/a);

except

On EZeroDivide do ShowMessage('Деление на ноль в ячейке '+IntToStr(i)+’,’ +IntToStr(j)); {Проверка класса ошибки}

end;

end;

end;

# **Тема.** Операции над множествами.

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Понятие множества

2 Операции над множествами

3 Описание множеств в программе

4 Применение множеств

**Вопросы для самоконтроля:**

1Чем отличаются понятия «множество» в математике и программировании

2 Приведите примеры описания множеств в программе

3 Перечислите операции, которые можно выполнять над множествами

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

**Множество**- это конечный или бесконечный набор определённых объектов, мыслимый как единое целое. Множество характеризуется своими элементами, а элементы имеют лишь одно свойство - принадлежность к данному множеству. Таким образом, мы можем только сказать, принадлежит элемент данному множеству или не принадлежит. Порядок расположения элементов в множестве никакой роли не играет.

**Множества в Delphi**

Понятие множества в языке программирования несколько отличается от математического определения этого понятия, но смысл сохраняется. Основное отличие в том, что в программировании множество может содержать только конечное число элементов, т.е. не может состоять из бесконечного числа объектов. В математике же последнее допустимо. Например, мы можем определить множество натуральных чисел, которое бесконечно: N = {1, 2, 3, ...}

Следует понимать, что множество не предназначено для хранения каких-либо значений (чисел, символов и т.д.) - оно лишь может дать нам ответ на вопрос: присутствует конкретный элемент в множестве или его там нет.

Множество может быть построено на основе перечислимого типа данных

Например, на основе символьного типа Char. По-английски множество называется **set** (*набор*) и именно этим словом описывается в Delphi:

var A: set of Char;

В данном примере мы объявили множество A на основе символьного типа Char.

**Запомните: множество не может состоять более чем из 255 элементов!**

Например, следующее описание:

var N: set of Integer;

приведёт к ошибке "*Set base type out of range*".

**Задание множеств**

В математике множество обычно записывается в виде фигурных скобок, в которых через запятую перечисляются элементы, принадлежащие множеству. В Delphi вместо фигурных скобок используются квадратные.

Чтобы задать множество, мы можем воспользоваться операцией присваивания, где слева стоит переменная-множество, а справа - нужный нам набор. Например, в описанное выше множество A мы хотим поместить элементы-символы A, B, C, D. Тогда это запишется так:

A:=['A','B','C','D'];

Теперь множество A содержит 4 элемента.

Если вспомнить, что тип данных Char упорядочен, то данную запись можно сократить следующим образом:

A:=['A'..'D'];

Мы просто указали диапазон значений, который должен находиться во множестве. Результаты одинаковый, но вторая запись короче и красивее. Допустимы, конечно же, комбинации диапазонов и отдельных значений:

A:=['A','B','K'..'N','R','X'..'Z'];

Помните, что множество - это виртуальный набор элементов: множество нельзя ввести с клавиатуры и точно так же нельзя вывести на экран. Поэтому добавление элементов во множество делается только программным путём. Безусловно, вы каким-либо образом можете связать элементы интерфейса программы и операцию добавления элементов во множество, но напрямую ввести множество нельзя. Аналогично, вы можете показать множество на экране с помощью каких-либо элементов (например, флажков TCheckBox), но само множество "в чистом виде" вывести нельзя.

**Операции над множествами**

В программировании, как и в математике, над множествами допустимы некоторые операции. Рассмотрим их.

**Находится ли элемент во множестве?**

Самая простая операция, для понятия смысла которой даже не нужно задумываться. Чтобы проверить, входит ли элемент во множество, следует использовать специальную конструкцию с оператором **in**. Слева от него указывается элемент, справа - множество. Результатом, как несложно догадаться, является логичное значение - истина или ложь. True - элемент принадлежит множеству, False - не принадлежит:

var A: set of Char;

{...}

A:=['A'..'E','X'];

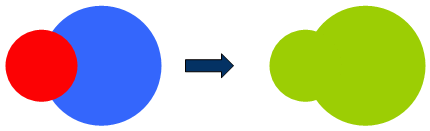
if 'D' in A then ShowMessage('Элемент В находится во множестве A.');

Несложно проверить, что сообщение в данном случае появится на экране.

**Объединение множеств**

Если есть два множества, определённые на одном и том же типе данных, то их можно объединить и получить таким образом новое множество.

Если изобразить множества в виде кругов, причём круги пересекаются в том случае, если у множеств есть одинаковые элементы, то объединение можно изобразить следующим образом:



В словесном описании операция объединения - результирующее множество содержит все те элементы, которые есть хотя бы в одном из двух исходных множеств.

Объединение записывается знаком плюс "**+**". Пример:

var A,B,C: set of Char;

{...}

A:=['A','B','C'];

B:=['X','Y','Z'];

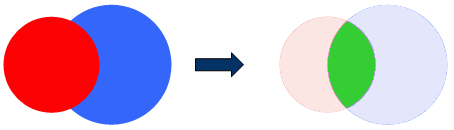
C:=A+B;

//C = ['A','B','C','X','Y','Z']

Включение одного элемента во множество делается точно таким же образом, просто в этом случае включаемое множество содержит всего один элемент.

**Пересечение множеств**

Операция пересечения формирует множество только из тех элементов, которые одновременное присутствуют как в первом, так и во втором исходном множестве. Операция пересечения графически:



Пересечение обозначается звёздочкой "**\***". Пример:

var X,Y,Z: set of Byte;

{...}

X:=[1,2,3,4,5];

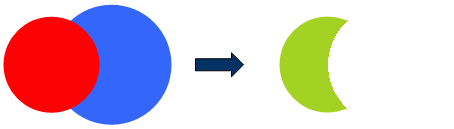
Y:=[4,5,6,7,8];

Z:=X\*Y;

//Z = [4,5]

**Разность множеств**

Операция вычитания удаляет из первого множества те элементы, которые есть во втором множестве:



Пример:

var X,Y,Z: set of Char;

{...}

X:=['A'..'D'];

Y:=['D'..'F'];

Z:=X-Y;

//Z = ['A'..'C']

Следует обратить внимание, что порядок множеств в данном случае важен, т.е. X-Y и Y-X - это разные множества.

**Применение множеств**

Множества находят широкое применение. С помощью множеств удобно задавать набор опций, каждая из которых либо включена, либо выключена. К примеру, поместите на форму кнопку (TButton), перейдите в инспектор объектов, разверните свойство Font (шрифт) и найдите свойство Style. Вот это свойство как раз и реализовано множеством. Во множестве 4 элемента: fsBold, fsItalic, fsUnderline и fsStrikeOut, каждый из которых отвечает за стиль шрифта. Принадлежность элементов ко множеству задаётся указанием значения True или False для каждого из этих пунктов. В строке "Style" находится описание данного множества. Попробуйте изменять стиль и посмотреть, как меняется описание множества Style.

А теперь давайте сделаем простенький интерфейс для доступа к этому свойству. Пусть будет меняться стиль шрифта у этой кнопки (Button1). Поместим на форму 4 TCheckBox - для доступа ко всем значениям и дадим им соответствующие имена. Изменение стиля будем делать при нажатии на саму эту кнопку. Пример реализации:

procedure TForm1.Button1Click(Sender: TObject);

begin

Button1.Font.Style:=[]; //Сделали множество пустым

//Теперь смотрим состояния флажков и добавляем нужные стили

if CheckBox1.Checked then Button1.Font.Style:=Button1.Font.Style+[fsBold];

if CheckBox2.Checked then Button1.Font.Style:=Button1.Font.Style+[fsItalic];

if CheckBox3.Checked then Button1.Font.Style:=Button1.Font.Style+[fsUnderline];

if CheckBox4.Checked then Button1.Font.Style:=Button1.Font.Style+[fsStrikeOut];

end;

Чтобы не повторять везде одно и то же "Button1.Font.", эту часть кода можно, что называется, вынести за скобку при помощи специального оператора **with**. Ранее речь о нём не шла, однако этот оператор очень удобен. Смысл его прост: то, что вынесено вперёд, автоматически применяется ко всему, что находится внутри данного блока. В нашем случае будет так:

procedure TForm1.Button1Click(Sender: TObject);

begin

with Button1.Font do

begin

Style:=[]; //Сделали множество пустым

//Теперь смотрим состояния флажков и добавляем нужные стили

if CheckBox1.Checked then Style:=Style+[fsBold];

if CheckBox2.Checked then Style:=Style+[fsItalic];

if CheckBox3.Checked then Style:=Style+[fsUnderline];

if CheckBox4.Checked then Style:=Style+[fsStrikeOut];

end

end;

Согласитесь, так гораздо удобнее. Используйте оператор *with* как можно чаще - с его помощью и код по объёму становится меньше и скорость работы увеличивается.



С помощью множеств реализованы свойства многих компонент. Множества не дают возможности хранить какие-либо значения - они просто позволяют создавать наборы из значений перечислимых типов данных. Но во многих случаях использование множеств - лучший и самый быстрый способ хранения некоторого набора параметров.

|  |  |
| --- | --- |
| Свойство Options у TOpenDialog - множество | У большинства компонент среди свойств можно найти множества. Например, у диалога открытия файла TOpenDialog (вкладка Dialogs) множеством представлено свойство Options, которое содержит приличное число элементов:  Вы спросите - а что это за названия элементов у множеств? Ответ прост - это специально объявленный перечислимый тип данных, на основе которого и создано множество. Если немного покопать, то можно найти описание этого типа:  TOpenOption = (ofReadOnly, ofOverwritePrompt, ofHideReadOnly,  ofNoChangeDir, ofShowHelp, ofNoValidate, ofAllowMultiSelect,  ofExtensionDifferent, ofPathMustExist, ofFileMustExist, ofCreatePrompt,  ofShareAware, ofNoReadOnlyReturn, ofNoTestFileCreate, ofNoNetworkButton,  ofNoLongNames, ofOldStyleDialog, ofNoDereferenceLinks, ofEnableIncludeNotify,  ofEnableSizing, ofDontAddToRecent, ofForceShowHidden);  TOpenOptions = set of TOpenOption;  Как видите, ничего сверхестественного здесь нет - вам всё уже знакомо. |

# **Тема.** Динамический массивВыделение и освобождение динамической памяти

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Объявление массива

2 Выделение памяти под массив

3 Операции с массивами

3 Освобождение занимаемой памяти

**Вопросы для самоконтроля:**

1В чем отличие статических и динамических массивов

2 Какая процедура используется для выделения памяти

3 Можно ли выполнять действия над массивом целиком

4 Какая процедура используется для освобождения памяти

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Динамические массивы в Delphi не имеют фиксированного размера . Чтобы объявить такой массив необходимо  записать:

**var** da\_MyArray : array of integer;

Как видим , мы просто говорим Delphi, что нам нужен одномерный массив типа Integer, а об его размере мы скажем  когда нибудь потом.

При таком объявлении память не выделяется, поэтому мы можем объявить много таких динамических массивов, не особо беспокоясь об объеме занимаемой памяти, и использовать любой динамический массив по мере надобности.

**2**. Для выделения памяти для динамического массива в Delphi используется процедура **SetLength**: SetLength(da\_MyArray,20);

После вызова этой процедуры будет выделена память для 20 элементов массива, которые будут проиндексированы от 0 до 19 (обратите внимание: индексирование начинается с нуля, а не с единицы!). После этого можно работать с динамическим массивом- присваивать ему значения, производить с элементами различные вычисления, вывод на печать и т.д.

Например

da\_MyArray[0] :=  5 ; da\_MyArray[9] :=  9 ;

da\_MyArray[1] :=  da\_MyArray[0]+ da\_MyArray[9] ;

**3.** Как только динамический массив был распределен, вы можете передавать массив стандартным функциям **Length**, **High**, **Low** и **SizeOf** Функция **Length** возвращает число элементов в динамическом массиве, **High** возвращает самый высокий индекс массива (то есть **Length**- 1), **Low**возвращает 0. В случае с массивом нулевой длины наблюдается интересная ситуация: **High**возвращает -1, а **Low** - 0, получается, что **High**меньше  **Low.**  :) Функция **SizeOf**всегда возвращает 4 - длина в байтах  памяти указателя на динамический массив

iHigh   := High (da\_MyArray3);  
iLow    := Low  (da\_MyArray3);

iLength := Length (da\_MyArray3);

iSizeOf := SizeOf (da\_MyArray3);

1. Доступ к данным динамических массивов с помощью низкоуровневых процедур типа **ReadFile**или **WriteFile** , или любых других подпрограмм, получающих доступ сразу ко всему массиву, часто выполняется неправильно. Для обычного массива (его часто называют также *статическим* массивом - в противоположность *динамическому* массиву) переменная массива тождественна его данным. Для динамического массива это не так - переменная  это указатель. Так что если вы хотите получить доступ к данным динамического массива - вы не должны использовать саму переменную массива, а использовать вместо неё первый элемент массива.

правильно

*WriteFile(FHandle, da\_MyArray02[0], Length(da\_MyArray02), dwTemp, nil)*

неправильно

WriteFile(FHandle, da\_MyArray02, Length(da\_MyArray02), dwTemp, nil)

**5.** Рассмотрим  пример присваивания динамических массивов одного другому

**var** da\_A,da\_B: **array** **of** integer;

**begin**

SetLength(da\_A,2);

SetLength(da\_B,2);

da\_A[0]:=2;

da\_B[0]:=3;

da\_A:=da\_B;

da\_B[0]:=4;

**end**;

После этих манипуляций da\_A[0] равно 4. Дело в том , что  при присвоении da\_A:=da\_B не происходит копирование т.к. da\_A, da\_B, это всего лишь указатели на область памяти. Для копирования необходимо использовать функцию **Copy.**

**6.**  Рассмотрим  пример копирования динамических массивов с использованием функции **Copy**

**Var**  da\_A,da\_B: **array** **of** integer;

**begin**

SetLength(da\_A,2);

SetLength(da\_B,2);

da\_A[0]:=2;

da\_B[0]:=3;

da\_A:=Copy (da\_B);

da\_B[0]:=4;

**end**;

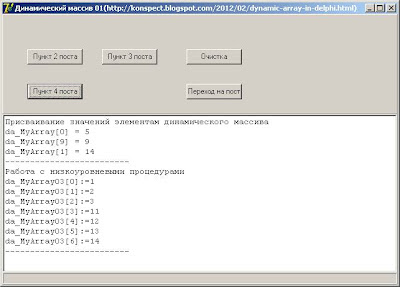
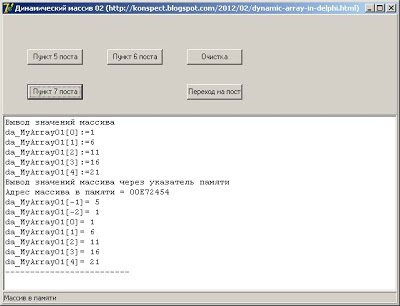
После этих манипуляций da\_A[0] равно 3. После функции **Copy**da\_A и da\_B указывают на разные области памяти, поэтому при изменении da\_B в da\_A ничего не происходит и его значения остаются неизменными**.**

**7.**Динамические массивы (например, *array of Integer*) в Delphi в памяти расположены следующим образом. Библиотека runtime добавляет специальный код, который управляет доступом и присваиваниями. В участке памяти **ниже** адреса, на который указывает ссылка динамического массива, располагаются служебные данные массива: два поля - число выделенных элементов и *счётчик ссылок* (reference count).

|  |
| --- |
| [Расположение динамического массива в памяти](http://4.bp.blogspot.com/-jilcCFolsYc/T2j2b-L1QWI/AAAAAAAAARk/ynvgzGDDPjE/s1600/Dynamic-Array-In-Memory.png) |
| Расположение динамического массива в памяти |

Если, как на диаграмме выше, *N* - это адрес в переменной динамического массива, то *счётчик ссылок* массива лежит по адресу N - 8, а число выделенных элементов (*указатель длины*) лежит по адресу N - 4. Первый элемент массива (сами данные) лежит по адресу N. Для каждой добавляемой ссылки (т.е. при присваивании, передаче как параметр в подпрограмму и т.п.) увеличивается счётчик ссылок, а для каждой удаляемой ссылки (т.е. когда переменная выходит из области видимости или при переприсваивании или присваивании nil) счётчик уменьшается.

**8.** Программы, которые иллюстрируют теоретические сведения по динамическим массивам.

[](http://2.bp.blogspot.com/-fBYP4Eb9wd8/T2o6dRXO35I/AAAAAAAAARs/NdkQY3bYfAo/s1600/Programm01.jpg)[](http://4.bp.blogspot.com/-tNTO3tjsL0w/T2o69SyaIGI/AAAAAAAAAR0/AI2EETkhplM/s1600/Programm02.jpg)

Окно программы пример 1 Окно программы пример 2

# **Тема.** Процедуры и функции для работы с файлами.

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Технология работы с файлами

2 Типы файлов

3 Процедуры и функции для работы с файлами

**Вопросы для самоконтроля:**

1Опишите последовательность действий при работе с файлами

2 Что общего и в чем отличия при работе с текстовыми и типизированными файлами

3 Как задать физическое местоположение файла

1. Какая функция свидетельствует об окончании файла
2. Что общего и в чем отличия способов открытия файла

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Технология **работы с файлами** в системе **Delphi** требует определённого порядка действий:

1. Прежде всего для файла должна быть определена файловая переменная (логическое имя файла, используемое в программе)
2. **файл** должен быть **открыт**. Система следит, чтобы другие приложения не мешали **работе с файлом**. При этом определяется, в каком режиме открывается **файл** - для изменения или только считывания информации. После открытия **файла** в программу возвращается его идентификатор, который будет использоваться для указания на этот **файл** во всех процедурах обработки.
3. Начинается **работа с файлом**. Это могут быть запись, считывание, поиск и другие операции.
4. **Файл** закрывается. Теперь он опять доступен другим приложениям без ограничений. Закрытие **файла** гарантирует, что все внесённые изменения будут сохранены, так как для увеличения скорости работы изменения предварительно сохраняются в специальных буферах операционной системы.

В **Delphi** реализовано несколько способов **работы с файлами**. Познакомимся со классическим способом, связанным с использованием **файловых переменных**. Файловая переменная вводится для указания на файл. Делается это с помощью ключевого слова **File** :

var F**:**  **File** **;**

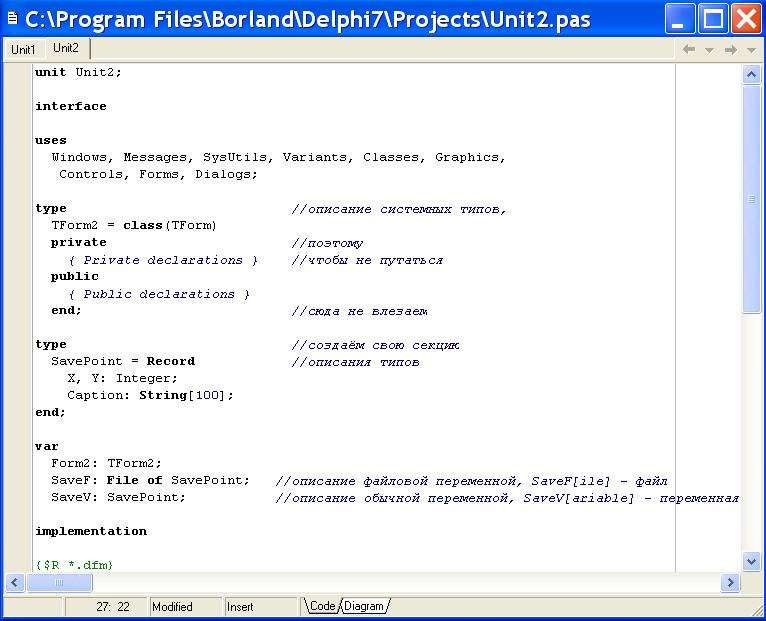
Описанная таким образом файловая переменная считается ***нетипизированной***, и позволяет работать с файлами с неизвестной структурой. Данные считываются и записываются побайтно блоками, размер которых указывается при открытии файла, вплоть от 1 байт.

Но чаще используются файлы, состоящие из последовательности одинаковых записей. Для описания такого файла к предыдущему описанию добавляется указание типа записи:

var F**:**  **File of** тип\_записи **;**

В качестве типа могут использоваться базовае типы, или создаваться свои. Важно только, чтобы для типа был точно известен фиксированный размер в байтах, поэтому, например, тип **String** в чистом виде применяться не может, а только в виде **String[N**].

Данные, считанные из файла или записываемые в файл, содержатся в обычной переменной, которая должна быть того же типа, что и файловая. Поэтому сначала в программе следует описать нужный тип, а затем ввести две переменные этого типа - файловую и обычную:



Для текстовых файлов отдельно укажу, что тип файловой переменной в этом случае **TextFile**, а тип обычной - **String**.

Для открытия файла нужно указать, где он расположен. Для этого файловая переменная должна быть ассоциирована с нужным файлом, который определяется его адресом. Адрес файла может быть абсолютным, с указанием диска и каталогов ('C:\Мои документы\Мои рисунки\FileName.ini'), или относительным, тогда он создаётся в папке с **.exe** файлом программы. Для задания относительного адреса достаточно указать имя файла с нужным расширением. Делается это оператором **AssignFile :**

*AssignFile(SaveF, 'C:\Мои документы\Мои рисунки\FileName.ini');*

*AssignFile(SaveF, 'FileName.ini');*

Теперь файл должен быть открыт.

Открытие файла оператором **Rewrite** приведёт воссозданию файла заново, т.е. существующий файл будет ***без предупреждения*** уничтожен, и на его месте будет создан новый пустой файл заданного типа, готовый к записи данных. Если же файла не было, то он будет создан.

Открытие файла оператором **Reset** откроет существующий файл к считыванию или записи данных, и его указатель будет установлен на начало файла**:**

*Rewrite(SaveF);*

*Reset(SaveF);*

Каждый из этих операторов может иметь второй необязательный параметр, имеющий смысл для нетипизированных файлов, и указывающий длину записи нетипизированного файла в байтах:

*Rewrite(SaveF,1);*

*Reset(SaveF, 1);*

*Чтение файла производится оператором****Read :***

*Read(SaveF, SaveV);*

*Запись в файл производится оператором****Write :***

*Write(SaveF, SaveV);*

При этом чтение и запись производится с текущей позиции указателя, затем указатель устанавливается на следующую запись. Можно проверить, существует ли нужный файл, оператором **FileExists :**

**If** FileExists('FileName.ini') **then** Read(SaveF, SaveV);

Принудительно установить указатель на нужную запись можно оператором **Seek(SaveF, N)**, где N - номер нужной записи, который, как и почти всё в программировании, отсчитывается от нуля:

Seek(SaveF, 49); - установка указателя на 50-ю запись.

При последовательном чтении из файла рано или поздно будет достигнут конец файла, и при дальнейшем чтении произойдёт ошибка. Проверить, не достигнут ли конец файла, можно оператором **EOF** (аббревиатура **E**nd **O**f **F**ile), который равен **true**, если прочитана последняя запись и указатель находится в конце файла:

**while** (not EOF(SaveF)) **do** Read(SaveF, SaveV);

Для текстовых файлов вместо **Read** и **Write** используются операторы **Readln** и **Writeln**, умеющие определять конец строки

Оператор **Truncate(SaveF)** позволяет отсечь (стереть или, если хотите, удалить!) все записи файла, начиная от текущей позиции указателя, и до конца файла.

В конце работы с файлом его необходимо закрыть. Это делается оператором **CloseFile(SaveF) ;**

Создаём обработчик события Формы **OnCreate** со следующим содержимым:

**procedure** TForm1.FormCreate(Sender: TObject) ;

**begin**  
AssignFile(SaveF, 'Init.ini') ;

**if** FileExists('Init.ini') **then begin** Reset(SaveF) ;

Read(SaveF, SaveV) ;

m1.Left := SaveV.X ;

Form1.Top := SaveV.Y ;

Form1.Caption:=SaveV.Caption ;//Наши переменные дополнительно сохраняют заголовок Формы!

**end ;**

**end ;**

Теперь необходимо создать обработчик события **OnClose :**

**procedure** TForm1.FormClose(Sender: TObject; var Action: TCloseAction) ;

**begin**

Rewrite(SaveF) ; //Нет необходимости проверять наличие файла, создадим его заново!

SaveV.X := Form1.Left ;

SaveV.Y := Form1.Top ;

SaveV.Caption := Form1.Caption ;

Write(SaveF, SaveV) ;

CloseFile(SaveF) ;

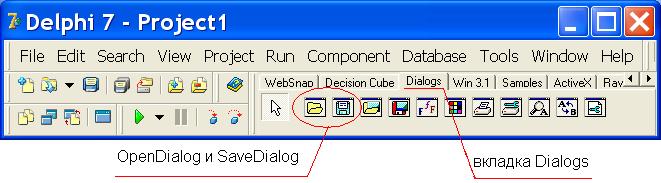
**end ;**

В данном случае файл считывается и записывается туда, куда мы ему указали. Но необходимо также уметь выбрать нужный файл в работающей программе

 То, что мы узнали в предыдущей части урока, позволяет работать с файлами по адресу, жёстко записанному в тексте программы.

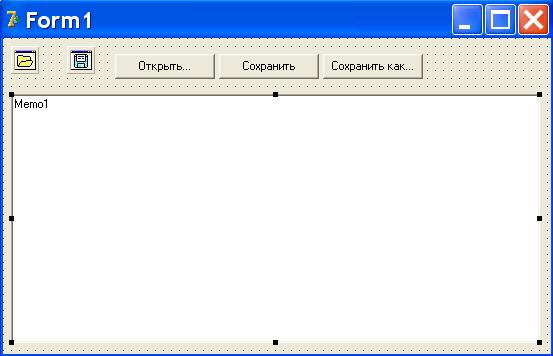
Рассмотрим компоненты, позволяющие в работающей программе осуществлять выбор файлов. **Delphi диалоги** выбора файла позволяют указать програме, с каким файлом мы хотим работать.

На вкладке палитры компонентов **Dialogs** находятся компонент **Delphi OpenDialog** и компонент **Delphi SaveDialog**. Все **Delphi диалоги**, находящиеся на этой вкладке, в том числе и **Delphi диалоги** выбора файла, невизуальные, т.е. при переносе их на Форму в работающей программе их не видно, они видны только на этапе конструирования. Компонент **Delphi OpenDialog** позволяет открыть в нашей программе стандартное Windows-окно диалога открытия файла, компонент **Delphi SaveDialog** - окно диалога сохранения.



**Delphi диалоги** выбора файла сами по себе ничего не делают, а только предоставляют настройки, сделанные пользователем при выборе файла. Самый важный метод **Delphi диалогов** - **Execute**. Он срабатывает в момент нажатия кнопки "**открыть**" или "**сохранить**" в окне выбора файла. Для примера давайте введём в программу возможность выбора файла для загрузки в редактор **Memo**, и сохранения после редактирования.

Итак, кидаем на Форму оба **Delphi диалога**, текстовый редактор **Memo**, и три кнопки **Button**. В свойство **Caption** одной из них записываем "**Открыть...**", другой - "**Сохранить**", третьей - "**Сохранить как...**"



В обработчике **OnClick** кнопки "**Открыть...**" пишем:

**if** OpenDialog1.Execute **then** Memo1.Lines.LoadFromFile(OpenDialog1.FileName);

В результате выбора файла свойство **FileName** компонента **OpenDialog** получает значение полного адреса выбранного файла, который мы и вставляем в функцию загрузки файла компонента **Memo**.

Всё это хорошо, но только в данном случае, когда записанное выражение записывается в одну строку. Если программа использует несколько раз выражение **OpenDialog1.FileName**, то писать руками устанешь. В **Delphi** для такого случая есть так называемый "оператор присоединения" **with**. Он используется для любых объектов, имеющих длинный "хвост" из свойств, которые приходится записывать многократно. Вот как он записывается:

**with** Объект **do begin**  **end;**

Свойства Объекта внутри логических скобок **begin/end** можно записывать непосредственно. Допускается перечислять через запятую несколько объектов. Естественно, в случае, когда внутри скобок находится один оператор, они необязательны. Перепишем фрагмент загрузки файла с использованием оператора присоединения:

**with** OpenDialog1, Memo1 **do if** Execute **then** Lines.LoadFromFile(FileName);

Запись получается более компактной.

Так как свойства компонентов **OpenDialog** и **SaveDialog** одинаковы, сохранение текста выглядит абсолютно аналогично. Создаём обработчик нажатия кнопки "**Сохранить как...**" и пишем:

**with** SaveDialog1, Memo1 **do if** Execute **then begin** Lines.SaveToFile(FileName);

OpenDialog1.FileName:=FileName; // Чтобы исправленный текст не затёр источник

**end;**

Наконец, для кнопки "**Сохранить**" пишем:

Memo1.Lines.SaveToFile(OpenDialog1.FileName); // Сохраняем туда, откуда считали (В предыдущей строчке была ошибка.

При работе этих фрагментов можно заметить, что выбирать приходится из всех файлов в нужной директории. Удобнее видеть только, например, текстовые файлы, или другой тип файлов по нашему выбору. Для этого используются фильтры, свойство **Filter** в наших компонентах. Настраивается оно в **Инспекторе Объектов**. При выборе его можно перейти в редактор фильтров:

|  |  |  |
| --- | --- | --- |
| Редактор фильтров |  | В колонке **FilterName** записываем имена фильтров, в колонке **Filter** - список масок файлов, разделённых точкой с запятой. Маска файла в данном случае выглядит как  **\*.расширение\_файла;**  Звёздочка означает, что выбираются файлы с любыми именами, подходящие по расширению. |

Свойство Delphi диалогов **Title** позволяет записать в заголовок нужную нам фразу. Если оставить его пустым, то в заголовке будут стандартные "**открыть**" или "**сохранить"**  
   Свойство **InitialDir** позволяет в момент открытия оказаться в нужной нам директории. Оно доступно как на этапе "конструирования", так и программно.

# **Тема.** Библиотеки подпрограмм. Рекурсивные подпрограммы

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Понятие процедуры и функции. Общее и различия

2 Локальные и глобальные данные

3 Понятие рекурсии, в каких случаях ее применяют

**Вопросы для самоконтроля:**

1Чем заголовок функции отличается от заголовка процедуры

2 В каких случаях желательно использовать процедуру, а в каких функцию и почему

3 В чем отличия локальных и глобальных данных

4 Как организовать рекурсивную подпрограмму

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Подпрограммы - процедуры и функции в языке Delphi служат для выполнения специализированных операций. Delphi имеет множество стандартных подпрограмм, но всё равно приходится создавать собственные для выполнения часто повторяющихся операций с данными, которые могут меняться.

Вообще, существует **методика программирования** "сверху вниз". **Методика программирования** сверху вниз" разбивает задачу на несколько более простых, которые оформляются в виде подпрограмм. Те, в свою очередь, при необходимости также делятся до тех пор, пока стоящие перед программистом проблемы не достигнут приемлемого уровня сложности (то есть простоты!). Таким образом, эта **методика программирования** облегчает написание программ за счёт создания так называемого скелета, состоящего из описателей подпрограмм, которые в дальнейшем наполняются конкретными алгоритмами. Пустое описание подпрограммы иначе называется "заглушкой".

И **процедуры**, и **функции** позволяют добиться одинаковых результатов. Но разница всё же есть.

**Процедура Delphi** просто выполняет требуемые операции, но никаких результатов своих действий не возвращает. Результат - в тех изменениях, которые произошли в программе в процессе выполнения этой процедуры. В частности, процедура может поменять значения переменных, записать новые значения в ячейки компонентов, сделать запись в файл и т.д.

**Функция Delphi** также позволяет выполнить всё перечисленное, но дополнительно возвращает результат в присвоенном ***ей самой*** значении. То есть вызов функции может **присутствовать в выражении** справа от оператора присваивания. Таким образом, функция - более универсальный объект!

Описание подпрограммы состоит из ключевого слова **procedure** или **function**, за которым следует имя подпрограммы со списком параметров, заключённых в скобки. В случае **функции** далее ставится двоеточие и указывается тип возвращаемого значения. Обычная точка с запятой далее - обязательна! Сам код подпрограммы заключается в "логические скобки" **begin/end**. Для функции необходимо в коде присвоить переменной с именем функции или специальной зарезервированной переменной **Result** (предпочтительно) возвращаемое функцией значение.

Примеры:

|  |  |
| --- | --- |
| **procedure**Имя\_процедуры(параметры); **begin**  Код процедуры; **end;** | **function** Имя\_функции(параметры): тип\_результ;  **begin**     Код функции;     Result:=***результат***;    **end;** |

Описанная таким образом подпрограмма должна быть размещена в основной программе до первого её вызова. Иначе при компиляции получите извещение о том, что "неизвестный идентификатор..." Следить за этим не всегда удобно. Есть выход - разместить только заголовок подпрограммы там, где размещают [описания всех данных программы](http://www.delphi-manual.ru/lesson4.php).

Параметры - это список идентификаторов, разделённых запятой, за которым через двоеточие указывается тип. Если списков идентификаторов разных типов несколько, то они разделяются точкой с запятой. Всё, как и в случае обычного описания данных. Это так называемые **формальные** параметры. При вызове подпрограммы они заменяются на **фактические** - следующие через запятую данные того же типа, что и формальные.

Параметры в описании подпрограммы **могут и отсутствовать**, тогда она оперирует данными прямо из основной программы.

Теперь нужно ввести понятие локальных данных. Это данные - переменные, константы, подпрограммы, которые используются и существуют только в момент вызова данной подпрограммы. Они так же должны быть описаны в этой подпрограмме. Место их описания - между заголовком и началом логического блока - ключевым словом **begin**. Имена **локальных** данных могут совпадать с именами **глобальных**. В этом случае используется локальная переменная, причём её изменение **не скажется** на глобальной с тем же именем.

Совершенно аналогично локальным типам, переменным, константам могут быть введены и локальные процедуры и функции, которые могут быть описаны и использованы только внутри данной подпрограммы.

Теперь пример. Напишем программу суммирования двух чисел. Она будет состоять из Формы, на которой будет кнопка (компонент **Button**), по нажатию на которую будет выполняться наша подпрограмма, и двух строк ввода (компоненты **Edit**), куда будем вводить операнды. Начнём **с процедуры**.

|  |  |
| --- | --- |
| **Var**    Form1: TForm1;    A, B, Summa: Integer;  **procedure** Sum(A, B: Integer);  **implementation**  {$R \*.dfm}  **procedure** TForm1.Button1Click(Sender: TObject);  **begin**   A:=StrToInt(Edit1.Text);   B:=StrToInt(Edit2.Text);   Sum(A, B);   Caption:=IntToStr(Summa);  **end;**  **procedure** Sum(A, B: Integer);  **begin**   Summa:=A+B;  **end;** | Программа суммирования двух чисел |

Наша процедура находится после обработчика нажатия кнопки, где осуществляется её вызов. И программа работает именно потому, что заголовок процедуры вынесен в блок описания данных. Но всё же операция суммирования в данном случае производится как-то невнятно.

Теперь сделаем то же самое с помощью **функции**.

**Var** Form1: TForm1;

A, B, Summa: Integer;

**function** Sum(A, B: Integer): Integer;

**implementation**

{$R \*.dfm}

**procedure** TForm1.Button1Click(Sender: TObject);

**begin**

 A:=StrToInt(Edit1.Text); B:=StrToInt(Edit2.Text);

 Summa:=Sum(A, B);        // На мой взгляд, сейчас более понятно, откуда что берётся

 Caption:=IntToStr(Summa);

**end;**

**function** Sum(A, B: Integer): Integer;

**begin**

 Result:=A+B;

**end;**

Есть особенности в использовании в качестве параметров больших по объёму структур данных, например, массивов, состоящих из нескольких тысяч (и больше) элементов. При передаче в подпрограмму данных большого объёма могут быть большие расходы ресурсов и времени системы. Поэтому используется передача не самих значений элементов (передача **"по значению"**, как в предыдущих примерах), а ссылки на имя переменной или константы (передача **"по имени"**). Достигается это вставкой перед теми параметрами, которые мы хотим передать по имени, ключевого слова **var**.

**function** Sum(A, B: Integer; **var** Arr: **array**[1..1000000] of Integer): Integer;

Если взглянуть на описание нашей подпрограммы и описание обработчика нажатия кнопки (это тоже подпрограмма!), который был создан Delphi, то видим, что перед именем обработчика (Button1Click) стоит **TForm1**. В Delphi точкой разделяется объект и его атрибуты (свойства и методы). Таким образом, Delphi создаёт Button1Click как метод объекта Form1. Причём, буква **T** перед объектом говорит о том, что Button1Click не просто метод объекта, а метод класса объекта. Не будем этим пока заморачиваться, а **просто будем поступать также**. ***Описав свою процедуру или функцию как метод класса TForm1, мы получаем возможность использовать в ней объекты класса без указания его имени***, что гораздо удобнее. То есть, если мы используем в нашей подпрограмме какие-либо компоненты, размещённые на Форме (например, Button1), то мы пишем

Button1.Width:=100;   //Ширина кнопки а не Form1.Button1.Width:=100;

Также появляется возможность использовать встроенные переменные, такие как [параметр Sender](http://www.delphi-manual.ru/sender.php). В каждом обработчике этот объект указывает на источник, то есть тот объект, который вызывает данную подпрограмму. Например, в нашей процедуре суммирования **Sender = Button1**. Проанализировав эту переменную, можно принять решение о тех или иных действиях.

Описав подпрограмму как метод класса, её описание мы должны поместить туда же, куда их помещает Delphi - в описание класса TForm1. Смотрите сами, где находится описание процедуры Button1Click. **Для этого**, поставив курсор внутрь подпрограммы Button1Click, нажмите **CTRL+Shift** и кнопку управления курсором "**Вверх**" или "**Вниз**" одновременно. Произойдёт переход к описанию подпрограммы (чтобы вернуться обратно, повторите это действие ещё раз). Ставьте описание своей подпрограммы рядом, с новой строки. Обратите внимание, что TForm1 уже не пишется.

**Понятие рекурсии**

**Рекурсия** - важное и мощное **свойство процедур и функций** в Delphi.

**Рекурсия** это возможность подпрограммы в процессе работы обращаться к самой себе. Без использования рекурсии приходилось бы применять циклы, а это усложняет чтение программы. Рекурсивный вызов подпрограммы сразу проясняет смысл происходящего. Естественно, приходится следить за тем, чтобы в подпрограмме обязательно было условие, при выполнении которого дальнейшая рекурсия прекращается, иначе подпрограмма зациклится.

**Рекурсивным** называется объект, частично состоящий или определяемый с помощью самого себя. Факториал - это классический пример рекурсивного объекта. Факториал числа п - это произведение целых чисел от 1 до п. Обозначается факториал числа п так: n!.

Согласно определению

n! = 1 х 2 х 3 х ... х (п - 1) х п. Приведенное выражение можно переписать так:

n! = nх ((n - 1) х (n - 2) х ...х 3 х 2 х 1) = n х (n - 1)!

То есть, факториал числа п равен произведению числа п на факториал числа (п - 1). В свою очередь, факториал числа («-!) - это произведение числа (п - 1) на факториал числа (п - 2) и т. д.

Таким образом, если вычисление факториала п реализовать как функцию, то в теле этой функции будет инструкция вызова функции вычисления факториала числа (п - 1), т. е. функция будет вызывать сама себя. Такой способ вызова называется рекурсией, а функция, которая обращается сама к себе, называется рекурсивной функцией.

**Листинг** *Рекурсивная функция вычисления факториала*

**function**factorial(n: integer): integer;

**begin**

**if**n <> 1

**then**factorials n \* factorial(n-1)

// функция вызывает сама себя

**else**factorial := 1; // рекурсивный процесс закончен

**end;**

Обратите внимание, что функция вызывает сама себя только в том случае, если значение полученного параметра k не равно единице. Если значение параметра равно единице, то функция сама себя не вызывает, а возвращает значение, и рекурсивный процесс завершается.

На рисунке приведен вид диалогового окна программы, которая для вычисления факториала числа использует рекурсивную функцию factorial.



Окно программы вычисления факториала

**Листинг** *Использование рекурсивной функции*

**Unit** factor ;

**interface**

**uses** Windows, Messages, SysUtils, Classes,Graphics, Controls, Forms, Dialogs, StdCtrls;

**type**

TForm1 = **class**(TForm)

Label1: TLabel;

Edit1: TEdit;

Button1: TButton;

Label2: TLabel;

**procedure**ButtonlClick(Sender: TObject) ;

**private**

{ Private declarations } **public**

{ Public declarations } **end;**

**Var** Form1: TForm1;

**implementation**

{$R \*.DFM}

// рекурсивная функция

**Function** factorial(n: integer): integer;

**begin**

if **n > 1 then** factorial := n \* factorial(n-1) // функция вызывает сама себя

**else** factorial:= 1; // факториал 1 равен 1

**end;**

**procedure** TForml.ButtonlClick(Sender: TObject);

**var**

k:integer; // число, факториал которого надо вычислить

f:integer; // значение факториала числа k

**begin**

k := StrToInt(Edit1.Text);

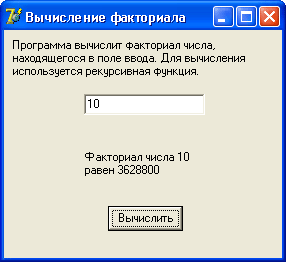
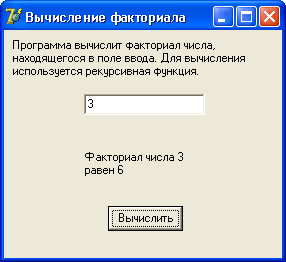
f := factorial(k);

label2.caption:='Факториал числа '+Edit1.Text

+ ' равен '+IntToStr(f);

**end; end.**

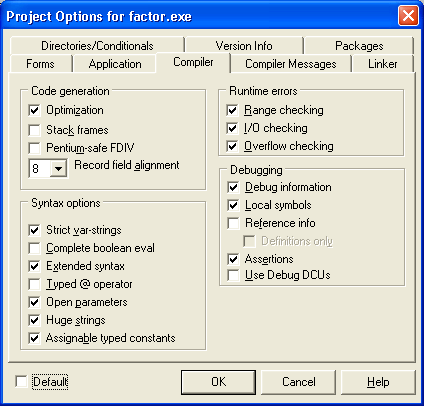
На рисунке приведены два диалоговых окна. Результат вычисления факториала, представленный соответствует ожидаемому.



Примеры работы программы вычисления факториала

Результат не соответствует ожидаемому, если он очень велик. Факториал числа 44 равен нулю! Произошло это потому, что факториал числа 44 настолько велик, что превысил максимальное значение для переменной типа integer, и, как говорят программисты, произошло переполнение с потерей значения.

Delphi может включить в исполняемую программу инструкции контроля диапазона значений переменных. Чтобы инструкции контроля были добавлены в программу, нужно во вкладке **Compiler** диалогового окна **Project Options** установить флажок **Overflow checking**(Контроль переполнения), который находится в группе **Runtime errors** (Ошибки времени выполнения).



Вкладка **Compiler**диалогового окна **Project Options**

# **Тема.** Программирование графики. Вывод текстовой информации

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Компоненты, используемые при работе с гафикой и их свойства.

Дать краткую характеристику: холст, карандаш, кисть

2 Вывод текста.

**Вопросы для самоконтроля:**

1Назовите основные компоненты, используемые при работе с гафикой

2 Из чего состоит холст

3 В чем отличия карандаша и кисти

**Форма контроля:** Оценка составленного конспекта.

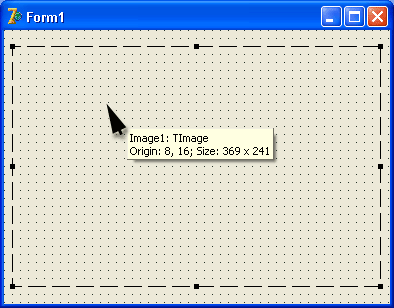
**Теоретический материал**

Программа выводит графику на поверхность объекта (формы или компонента Image). Поверхности объекта соответствует свойство canvas. Для того чтобы вывести на поверхность объекта графический элемент (прямую линию, окружность, прямоугольник и т. д.), необходимо применить к свойству canvas этого объекта соответствующий метод. Например, инструкция Form1.Canvas.Rectangle (10,10,100,100) вычерчивает в окне программы прямоугольник.

**Холст**

Как было сказано ранее, поверхности, на которую программа может выводить графику, соответствует свойство Canvas. В свою очередь, свойство canvas — это объект типа TCanvas. Методы этого типа обеспечивают вывод графических примитивов (точек, линий, окружностей, прямоугольников и т. д.), а свойства позволяют задать характеристики выводимых графических примитивов: цвет, толщину и стиль линий; цвет и вид заполнения областей; характеристики шрифта при выводе текстовой информации.

Методы вывода графических примитивов рассматривают свойство Canvas как некоторый абстрактный холст, на котором они могут рисовать (canvas переводится как "поверхность", "холст для рисования"). Холст состоит из отдельных точек — пикселов. Положение пиксела характеризуется его горизонтальной (X) и вертикальной (Y) координатами. Левый верхний пиксел имеет координаты (0, 0). Координаты возрастают сверху вниз и слева направо (рис. 1). Значения координат правой нижней точки холста зависят от размера холста.



**Рис. 1.**Координаты точек холста

**Размер холста можно получить, обратившись к свойствам**Height и width **области иллюстрации**(image) или к **свойствам формы:**ClientHeight и Clientwidth.

**Карандаш и кисть**

Методы, обеспечивающие вычерчивание на поверхности холста графических примитивов, тоже используют карандаш и кисть. Карандаш применяется для вычерчивания линий и контуров, а кисть — для закрашивания областей, ограниченных контурами.

Карандашу и кисти, используемым для вывода графики на холсте, соответствуют свойства Реn (карандаш) и Brush (кисть), которые представляют собой объекты типа треп и TBrush, соответственно. Значения свойств этих объектов определяют вид выводимых графических элементов.

**Карандаш**

Карандаш используется для вычерчивания точек, линий, контуров геометрических фигур: прямоугольников, окружностей, эллипсов, дуг и др. Вид линии, которую оставляет карандаш на поверхности холста, определяют свойства объекта треп, которые перечислены в табл. 1.

**Таблица 1.**Свойства объекта треп (карандаш)

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| Color | Цвет линии |
| Width | Толщину линии |
| Style | Вид линии |
| Mode | Режим отображения |

Свойство Color задает цвет линии, вычерчиваемой карандашом. В табл. 2 перечислены именованные константы (тип TCoior), которые можно использовать в качестве значения свойства color.

**Таблица 2.**Значение свойства Color определяет цвет линии

|  |  |  |  |
| --- | --- | --- | --- |
| **Константа** | **Цвет** | **Константа** | **Цвет** |
| clBlack | Черный | clSilver | Серебристый |
| clMaroon | Каштановый | clRed | Красный |
| clGreen | Зеленый | clLime | Салатный |
| clOlive | Оливковый | clBlue | Синий |
| clNavy | Темно-синий | clFuchsia | Ярко-розовый |
| clPurple | Розовый | clAqua | Бирюзовый |
| clTeal | Зелено-голубой | clWhite | Белый |
| clGray | Серый |  |  |

Свойство width задает толщину линии (в пикселах). Например, инструкция Canvas. Pen. width: =2 устанавливает толщину линии в 2 пиксела.

Свойство style определяет вид (стиль) линии, которая может быть непрерывной или прерывистой, состоящей из штрихов различной длины. В табл. 3 перечислены именованные константы, позволяющие задать стиль линии. Толщина пунктирной линии не может быть больше 1. Если значение свойства Pen.width больше единицы, то пунктирная линия будет выведена как сплошная.

**Таблица 3.**Значение свойства Реn. туре определяет вид линии

|  |  |
| --- | --- |
| **Константа** | **Вид линии** |
| psSolid | Сплошная линия |
| psDash | Пунктирная линия, длинные штрихи |
| psDot | Пунктирная линия, короткие штрихи |
| psDashDot | Пунктирная линия, чередование длинного и короткого штрихов |
| psDashDotDot | Пунктирная линия, чередование одного длинного и двух коротких штрихов |
| psClear | Линия не отображается (используется, если не надо изображать границу области, например, прямоугольника) |

Свойство Mode определяет, как будет формироваться цвет точек линии в зависимости от цвета точек холста, через которые эта линия прочерчивается. По умолчанию вся линия вычерчивается цветом, определяемым значением свойства Pen.Color.

Однако программист может задать инверсный цвет линии по отношению к цвету фона. Это гарантирует, что независимо от цвета фона все участки линии будут видны, даже в том случае, если цвет линии и цвет фона совпадают.

В табл. 4 перечислены некоторые константы, которые можно использовать в качестве значения свойства Pen.Mode.

**Таблица 4** Значение свойства Реп. Mode влияет на цвет линии

|  |  |
| --- | --- |
| **Константа** | **Цвет линии** |
| pmBlack | Черный, не зависит от значения свойства Pen. Color |
| pmWhite | Белый, не зависит от значения свойства Pen. Color |
| pmCopy | Цвет линии определяется значением свойства Pen . Color |
| pmNotCopy | Цвет линии является инверсным по отношению к значению свойства Pen. Color |
| pmNot | Цвет точки линии определяется как инверсный по отношению к цвету точки холста, в которую выводится точка линии |

**Кисть**

Кисть (canvas.Brush) используется методами, обеспечивающими вычерчивание замкнутых областей, например геометрических фигур, для заливки (закрашивания) этих областей. Кисть, как объект, обладает двумя свойствами, перечисленными в табл.5.

**Таблица 5.**Свойства объекта TBrush (кисть)

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| Color  Style | Цвет закрашивания замкнутой области  Стиль (тип) заполнения области |

Область внутри контура может быть закрашена или заштрихована. В первом случае область полностью перекрывает фон, а во втором — сквозь незаштрихованные участки области будет виден фон.

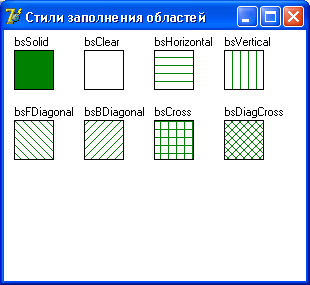
В качестве значения свойства Color можно использовать любую из констант типа TColor (см. список констант для свойства Pen.color в табл. 2).

Константы, позволяющие задать стиль заполнения области, приведены в табл. 6.

**Таблица 6.**Значения свойства Brush, style определяют тип закрашивания

|  |  |
| --- | --- |
| **Константа** | **Тип заполнения (заливки) области** |
| bsSolid | Сплошная заливка |
| bsClear | Область не закрашивается |
| bsHorizontal | Горизонтальная штриховка |
| bsVertical | Вертикальная штриховка |
| bsFDiagonal | Диагональная штриховка с наклоном линий вперед |
| bsBDiagonal | Диагональная штриховка с наклоном линий назад |
| bsCross | Горизонтально-вертикальная штриховка, в клетку |
| bsDiagCross | Диагональная штриховка, в клетку |

В качестве примера в листинге 1 приведена программа **Стили заполнения областей,**которая в окно (рис. 2) выводит восемь прямоугольников, закрашенных черным цветом с использованием разных стилей.



**Рис. 2.**Окно программы **Стили заполнения областей**

**Листинг 1. Стили заполнения областей**

**Unit** brustyle\_; **interface**

**Uses** Windows, Messages, SysUtils, Classes,Graphics, Controls, Forms, Dialogs, ExtCtrls;

**type**

TForm1 = **class**(TForm)

**procedure**FormPaint(Sender: TObject);

**private**

{ Private declarations}

**public**

{ Public declarations )

**end;**

**var** Form1: TForm1;

**implementation**

{$R \*.DFM}

// перерисовка формы

**procedure**TForm1.FormPaint(Sender: TObject);

**const**

bsName: **array**[1..8]**of string =**('bsSolid', 'bsClear', 'bsHorizontal', 'bsVertical', 'bsFDiagonal', 'bsBDiagonal', 'bsCross','bsDiagCross');

**var** x,y: integer // координаты левого верхнего угла прямоугольника

w,h: integer; // ширина и высота прямоугольника

bs: TBrushStyle;// стиль заполнения области

k: integer; // номер стиля заполнения

i,j: integer;

**begin**

w:=40; h:=40; // размер области(прямоугольника)

у:=20;

**for i:=l to 2 do begin** х:=10;

**for**j:=1 **to**4 **do begin** k:=j+(i-1)\*4; // номер стиля заполнения

**case k of**

1: bs = bsSolid;

2: bs = bsClear;

3: bs = bsHorizontal;

4: bs = bsVertical;

5: bs = bsFDiagonal;

6: bs = bsBDiagonal;

7: bs = bsCross;

8: bs = bsDiagCross; end;

// вывод прямоугольника

Canvas.Brush.Color := clGreen;

// цвет закрашивания — зеленый

Canvas.Brush.Style := bs;

// стиль закрашивания

Canvas . Rectangle (x, y, x+w, y-t-h) ;

// вывод названия стиля

Canvas.Brush.Style := bsClear;

Canvas.TextOut(x, y-15, bsName[k]);

// вывод названия стиля

x := x+w+30;

**end;**

у := y+h+30;

**end;end;end.**

**Вывод текста**

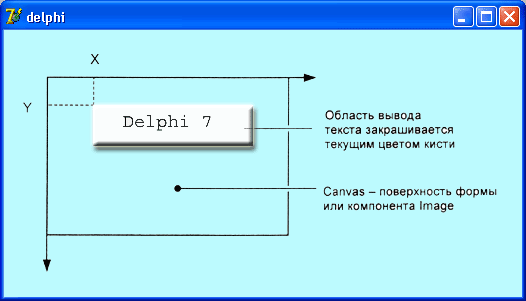
Для вывода текста на поверхность графического объекта используется метод TextOut. Инструкция вызова метода TextOut в общем виде выглядит следующим образом:

*Объект.Canvas.TextOut(x, у, Текст)* где:

объект — имя объекта, на поверхность которого выводится текст;

х, у — координаты точки графической поверхности, от которой выполняется вывод текста (рис. 10.3);

Текст — переменная или константа символьного типа, значение которой определяет выводимый методом текст.



**Рис. 3.**Координаты области вывода текста

Шрифт, который используется для вывода текста, определяется значением свойства Font соответствующего объекта canvas. Свойство Font представляет собой объект типа TFont. В табл. 7 перечислены свойства объекта TFont, позволяющие задать характеристики шрифта, используемого методами TextOut и TextRect для вывода текста.

**Таблица 7.**Свойства объекта TFont

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| Name    Size  Style | Используемый шрифт. В качестве значения следует использовать название шрифта, например Arial  Размер шрифта в пунктах (points). Пункт— это единица измерения размера шрифта, используемая в полиграфии. Один пункт равен 1/72 дюйма  Стиль начертания символов. Может быть: нормальным, полужирным, курсивным, подчеркнутым, перечеркнутым. Стиль задается при помощи следующих констант: fsBold (полужирный), fsltalic (курсив), f sUnderline (подчеркнутый), f sStrikeOut (перечеркнутый). |
| style    Color | Свойство style является множеством, что позволяет комбинировать необходимые стили. Например, инструкция программы, устанавливающая стиль "полужирный курсив", выглядит так: Объект. Canvas . Font : = [fsBold, fs Italic]  Цвет символов. В качестве значения можно использовать константу типа Tcolor |

**Внимание!**

Область вывода текста закрашивается текущим цветом кисти. Поэтому перед выводом текста свойству Brush.Color нужно присвоить значение bsClear или задать цвет кисти, совпадающий с цветом поверхности, на которую выводится текст.

Следующий фрагмент программы демонстрирует использование функции Textout для вывода текста на поверхность формы:

**with**Form1.Canvas **do begin**

// установить характеристики шрифта

Font.Name := 'Tahoma';

Font.Size := 20;

Font.Style := [fsltalic, fsBold] ;

Brush.Style := bsClear; // область вывода текста не закраши-

TextOut(0, 10, 'Borland Delphi 7');

**end;**

После вывода текста методом Textout указатель вывода (карандаш) перемещается в правый верхний угол области вывода текста.

Иногда требуется вывести какой-либо текст после сообщения, длина которого во время разработки программы неизвестна. Например, это может быть слово "руб." после значения числа, записанного прописью. В этом случае необходимо знать координаты правой границы уже выведенного текста. Координаты правой границы текста, выведенного методом Textout, можно получить, обратившись к свойству PenPos.

Следующий фрагмент программы демонстрирует возможность вывода строки текста при помощи двух инструкций Textout.

**with**Form1.Canvas **do begin**

TextOut(0, 10, 'Borland ') ;

TextOut(PenPos.X, PenPos.Y, 'Delphi 7');

**end;**

**Методы вычерчивания графических примитивов**

Любая картинка, чертеж, схема могут рассматриваться как совокупность графических примитивов: точек, линий, окружностей, дуг и др.

Вычерчивание графических примитивов на поверхности компонента (формы или области вывода иллюстрации) осуществляется применением соответствующих методов к свойству Canvas этого компонента.

**Линия**

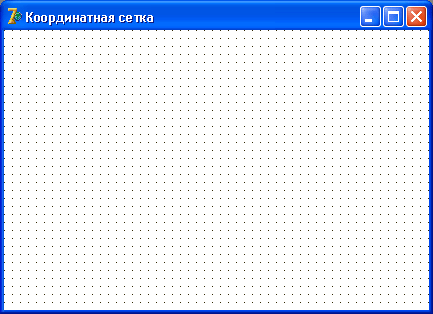
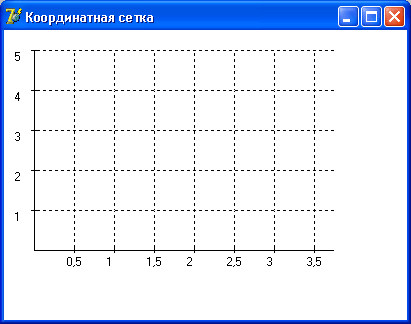
Вычерчивание прямой линии осуществляет метод LinеТо, инструкция вызова которого в общем виде выглядит следующим образом: *Компонент.Canvas.LineTo(x,у)*

Метод LinеТо вычерчивает прямую линию от текущей позиции карандаша в точку с координатами, указанными при вызове метода.

Начальную точку линии можно задать, переместив карандаш в нужную точку графической поверхности. Сделать это можно при помощи метода MoveTo, указав в качестве параметров координаты нового положения карандаша.

Вид линии (цвет, толщина и стиль) определяется значениями свойств объекта Реп графической поверхности, на которой вычерчивается линия.

Довольно часто результаты расчетов удобно представить в виде графика. Для большей информативности и наглядности графики изображают на фоне координатных осей и оцифрованной сетки. В листинге 2 приведен текст программы, которая на поверхность формы выводит координатные оси и оцифрованную сетку (рис. 4).



**Рис. 4.**Форма приложения **Координатная сетка**

**Листинг 2. Оси координат и оцифрованная сетка**

**unit**grid\_;

**interface**

uses Windows, Messages, SysUtils, Classes,

Graphics, Controls, Forms, Dialogs, StdCtrls;

**type**

TForm1 = class(TForm)

**procedure**FormPaint(Sender: TObject);

private

{ Private declarations }

**public**

{ Public declarations }

**end;**

**var** Form1: TForm1; **implementation**

{$R \*.DFM}

**procedure**TForm1.FormPaint(Sender: TObject);

**var** x0,y0:integer; // координаты начала координатных осей

dx,dy:integer; // шаг координатной сетки (в пикселах)

h,w:integer; // высота и ширина области вывода координатной сетки

х,у:integer;

lx,ly:real; // метки (оцифровка) линий сетки по X и Y

dlx,dly:real; // шаг меток (оцифровки) линий сетки по X и Y

cross:integer; // счетчик неоцифрованных линий сетки

dcross:integer;// количество неоцифрованных линий между оцифрованными

**begin**

х0:=30; у0:=220; // оси начинаются в точке (40,250)

dx:=40; dy:=40; // шар координатной сетки 40 пикселов

dcross:=1; // помечать линии сетки X: 1 — каждую;

// 2 — через одну;

// 3 — через две;

dlx:=0.5; // шаг меток оси X

dly:=1.0; // шаг меток оси Y, метками будут: 1, 2, 3 и т. д.

h:=200; w:=300;

**with**forml.Canvas **do begin**

cross:=dcross;

MoveTo(x0,v0); LineTo(x0,y0-h); // ось X

MoveTo(x0,y0); LineTo(x0+w, y0); // ось Y

// засечки, сетка и оцифровка по оси X

x:=x0+dx; lx:=dlx;

**repeat**

MoveTo(x,y0-3);LineTo(x,yO+3); // засечка

cross:=cross-l;

if cross = 0 **then**// оцифровка

**begin**

TextOut(x-8,y0+5,FloatToStr(lx));

cross:=dcross ; end;

Pen.Style:=psDot;

MoveTo(x,y0-3);LineTo(x,y0-h); // линия сетки

Pen.Style:=psSolid; lx:=lx+dlx; x:=x+dx;

until (x>x0+w);

// засечки, сетка и оцифровка по оси Y

y:=y0-dy; ly:=dly;

**repeat**

MoveTo(х0-3,у);LineTo(х0+3,у); // засечка

TextOut(х0-20,у,FloatToStr(1у)); // оцифровка

Pen.Style:=psDot;

MoveTo(х0+3,у); LineTo(x0+w,у); // линия сетки

Pen.Style:=psSolid; y:=y-dy; ly:=ly+dly;

until (y<y0-h);

**end;end;**

**end.**

Особенность приведенной программы заключается в том, что она позволяет задавать шаг сетки и оцифровку. Кроме того, программа дает возможность оцифровывать не каждую линию сетки оси х, а через одну, две, три и т. д. Сделано это для того, чтобы предотвратить возможные наложения изображений чисел оцифровки друг на друга в случае, если эти числа состоят из нескольких цифр.

**Ломаная линия**

Метод polyline вычерчивает ломаную линию. В качестве параметра метод получает массив типа TPoint. Каждый элемент массива представляет собой запись, поля х и у которой содержат координаты точки перегиба ломаной. Метод Polyline вычерчивает ломаную линию, последовательно соединяя прямыми точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д.

В качестве примера использования метода Polyline в листинге 3 приведена процедура, которая выводит график изменения некоторой величины. Предполагается, что исходные данные находятся в доступном процедуре массиве Data (тип Integer).

**Листинг 3. График функции (использование метода Polyline)**

**procedure**TForml.Button1Click(Sender: TObject);

**var**

gr: array[1..50] **of**TPoint; // график — ломаная линия

x0,y0: integer; // координаты точки начала координат

dx,dy: integer; // шаг координатной сетки по осям X и Y

i: integer; **begin**

х0 := 10; у0 := 200; dx **:=**5**;**dy := 5;

// заполним массив gr

**for**i:=l **to**50 **do begin**

gr[i].x := x0 + (i-l)\*dx; gr[i].y := y0 - Data[i]\*dy;

**end;**

// строим график

**with**forml.Canvas **do begin**

MoveTo(x0,y0); LineTo(x0,10); // ось Y

MoveTo(x0,y0); LineTo(200,y0); // ось X

Polyline(gr); // график

**end; end;**

Метод Polyline можно использовать для вычерчивания замкнутых контуров. Для этого надо, чтобы первый и последний элементы массива содержали координаты одной и той же точки. В качестве примера использования метода Polybine для вычерчивания замкнутого контура в листинге 4 приведена программа, которая на поверхности диалогового окна, в точке нажатия кнопки мыши, вычерчивает контур пятиконечной звезды (рис. 5).

Цвет, которым вычерчивается звезда, зависит от того, какая из кнопок мыши была нажата. Процедура обработки нажатия кнопки мыши (событие MouseDown) вызывает процедуру рисования звезды starLine и передает ей в качестве параметра координаты точки, в которой была нажата кнопка. Звезду вычерчивает процедура starLine, которая в качестве параметров получает координаты центра звезды и холст, на котором звезда должна быть выведена. Сначала вычисляются координаты концов и впадин звезды, которые записываются в массив р. Затем этот массив передается в качестве параметра методу Polyline. При вычислении координат лучей и впадин звезды используются функции sin и cos. Так как аргумент этих функций должен быть выражен в радианах, то значение угла в градусах домножается на величину pi/18о, где pi — это стандартная именованная константа равная числу пи.

**Листинг 4. Вычерчивание замкнутого контура (звезды) в точке нажатия кнопки мыши**

**unit**Stars\_; **interface**

**uses** Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

**type**

TForm1 = **class**(TForm)

**procedure**FormMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

**private**

{ Private declarations }

**public**

{ Public declarations }

**end;**

**var** Forml: TForml;

**implementation**

f$R \*.dfm}

// вычерчивает звезду

**procedure**StarLine(x0,y0,r: integer; Canvas: TCanvas);

// x0,y0 — координаты центра звезды, r — радиус заезды

**var**

р : array [1.. 11] **of**TPoint; // массив координат лучей и впадин

a: integer; // угол между осью ОХ и прямой, соединяющей

// центр звезды и конец луча или впадину

i: integer;

**begin**

а := 18; // строим от правого гор. луча

**for**i:=l **to**10 **do begin**

**if**(i **mod**2=0) **then begin**// впадина

p[i].x := x0+Round(r/2\*cos(a\*pi/180) ) ;

p[i] .y:=y0-Round(r/2\*sin(a\*pi/180) ) ;

**end**

**else begin // луч**

[i] .x:=x0+Round(r\*cos (a\*pi/180) ) ;

[i] .y:=y0-Round(r\*sin(a\*pi/180) ) ;

**end;**

a := a+36;

**end;**

p[ll].X := p[l].X; // чтобы замкнуть контур звезды

Canvas. Polyline (р) ; // начертить звезду

**end;**

// нажатие кнопки мыши

**procedure**TForm1 . FormMouseDown { Sender : TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

**begin**

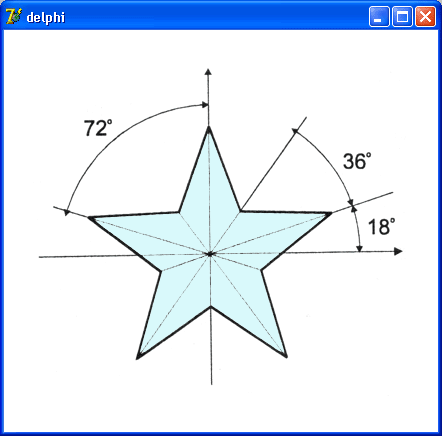
**if**Button = mbLeft // нажата левая кнопка?

**then**Form1. Canvas . Pen . Color : = clRed

**else**Form1. Canvas. Pen. Color := clGreen;

StarLine(x, y, 30, Forml. Canvas );

**end;**

**end.**

**Рис. 5.**Звезда

**Примечание**

Обратите внимание, что размер массива р на единицу больше, чем количество концов и впадин звезды, и что значения первого и последнего элементов массива совпадают.

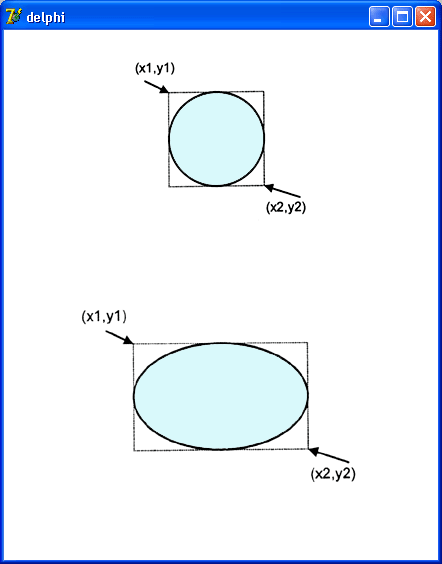
**Окружность и эллипс**

Метод Ellipse вычерчивает эллипс или окружность, в зависимости от значений параметров. Инструкция вызова метода в общем виде выглядит следующим образом:

*Объект.Canvas.Ellipse(x1,y1, х2,у2]*

где:

* объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;
* x1, y1, х2, у2 — координаты прямоугольника, внутри которого вычерчивается эллипс или, если прямоугольник является квадратом, окружность (рис. 6).



**Рис. 6.**Значения параметров метода Ellipse определяют вид геометрической фигуры

Цвет, толщина и стиль линии эллипса определяются значениями свойства Реп, а цвет и стиль заливки области внутри эллипса — значениями свойства Brush поверхности (canvas), на которую выполняется вывод.

**Дуга**

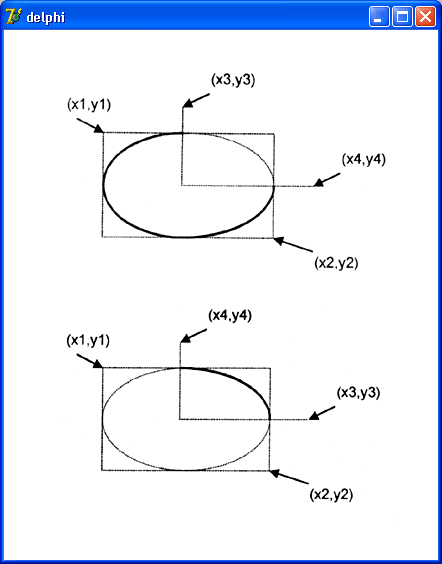
Вычерчивание дуги выполняет метод Arc, инструкция вызова которого в общем виде выглядит следующим образом: *Объект.Canvas.Arc(x1,y1,х2,у2,х3,у3,х4,у4)*

где:

* x1, y1, х2, у2 — параметры, определяющие эллипс (окружность), частью которого является вычерчиваемая дуга;
* х3, у3 — параметры, определяющие начальную точку дуги; П х4, у4 — параметры, определяющие конечную точку дуги.

Начальная (конечная) точка — это точка пересечения границы эллипса и прямой, проведенной из центра эллипса в точку с координатами х3 и у3 (х4, у4). Дуга вычерчивается против часовой стрелки от начальной точки к конечной (рис. 7).

Цвет, толщина и стиль линии, которой вычерчивается дуга, определяются значениями свойства Реп поверхности (canvas), на которую выполняется вывод.



**Рис. 7.**Значения параметров метода Arc определяют дугу как часть эллипса (окружности)

**Прямоугольник**

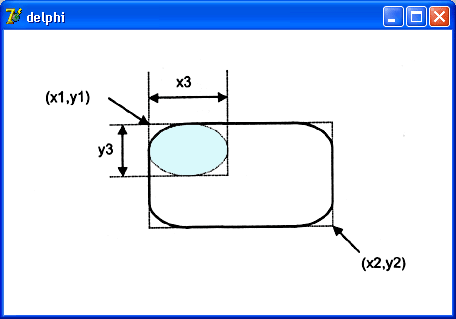
Прямоугольник вычерчивается методом Rectangle, инструкция вызова которого в общем виде выглядит следующим образом: *Объект.Canvas.Rectangle(x1, y1,x2, y2)*

где:

* объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;
* x1, y1 и х2, у2 — координаты левого верхнего и правого нижнего углов прямоугольника.

Метод RoundRec тоже вычерчивает прямоугольник, но со скругленными углами. Инструкция вызова метода RoundRec выглядит так: *Объект.Canvas.RoundRec(x1,y1,х2, у2, х3, у3),* где:

* x1, y1, х2, у2 -- параметры, определяющие положение углов прямоугольника, в который вписывается прямоугольник со скругленными углами;
* х3 и у3 — размер эллипса, одна четверть которого используется для вычерчивания скругленного угла (рис. 8).



**Рис. 8.**Метод RoundRec вычерчивает прямоугольник со скругленными углами

Вид линии контура (цвет, ширина и стиль) определяется значениями свойства Реп, а цвет и стиль заливки области внутри прямоугольника — значениями свойства Brush поверхности (canvas), на которой прямоугольник вычерчивается.

Есть еще два метода, которые вычерчивают прямоугольник, используя в качестве инструмента только кисть (Brush). Метод FillRect вычерчивает закрашенный прямоугольник, а метод FrameRect — только контур. У каждого из этих методов лишь один параметр — структура типа TRect. Поля структуры TRect содержат координаты прямоугольной области, они могут быть заполнены при помощи функции Rect.

Ниже в качестве примера использования методов FillRect и FrameRect приведена процедура, которая на поверхности формы вычерчивает прямоугольник с красной заливкой и прямоугольник с зеленым контуром.

**procedure**TForm1.Button1Click(Sender: TObject);

**var** r1, r2: TRect; // координаты углов прямоугольников

**begin**

// заполнение полей структуры

// зададим координаты углов прямоугольников

r1 := Rect(20,20,60,40); r2 := Rect(10,10,40,50);

**with** fоrm1.Canvas **do begin**

Brush.Color := clRed; FillRect(r1); // закрашенный прямоугольник

Brush.Color := clGreen; FrameRect(r2}; // только граница прямоугольника

**end; end;**

**Многоугольник**

Метод Polygon вычерчивает многоугольник. В качестве параметра метод получает массив типа TPoint. Каждый элемент массива представляет собой запись, поля (х,у) которой содержат координаты одной вершины многоугольника. Метод Polygon вычерчивает многоугольник, последовательно соединяя прямыми линиями точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д. Затем соединяются последняя и первая точки.

Цвет и стиль границы многоугольника определяются значениями свойства Реп, а цвет и стиль заливки области, ограниченной линией границы, — значениями свойства Brush, причем область закрашивается с использованием текущего цвета и стиля кисти.

Ниже приведена процедура, которая, используя метод polygon, вычерчивает треугольник:

**procedure**TForm1.Button2Click(Sender: TObject);

**var** pol: array[1..3] **of**TPoint; // координаты точек треугольника

**begin**

pol[1].x := 10;

polf1].y := 50;

pol[2].x := 40;

pol[2].y := 10;

pol[3].х := 70;

pol[3].у := 50;

Form1.Canvas.Polygon(pol);

**end;**

**Сектор**

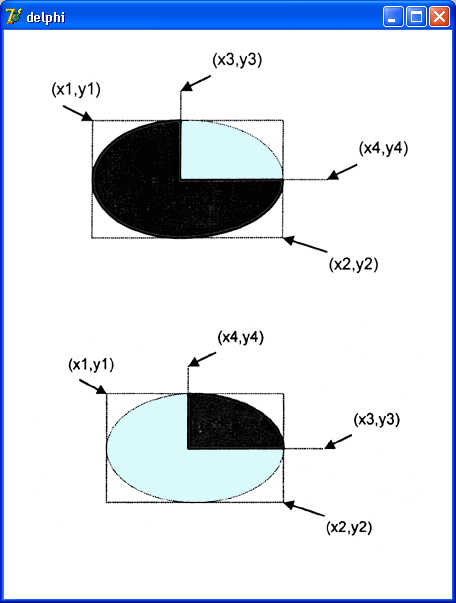
Метод pie вычерчивает сектор эллипса или круга. Инструкция вызова метода в общем виде выглядит следующим образом:

Объект. Canvas.Pie(x1,y1,x2,y2,х3,у3,х4,у4)

где:

* x1, y1, х2, у2 — параметры, определяющие эллипс (окружность), частью которого является сектор;
* х3, у3, х4, у4 — параметры, определяющие координаты конечных точек прямых, являющихся границами сектора.

Начальные точки прямых совпадают с центром эллипса (окружности). Сектор вырезается против часовой стрелки от прямой, заданной точкой с координатами (хЗ, уз), к прямой, заданной точкой с координатами (х4, у4) (рис. 9).



**Рис. 9.**Значения параметров метода Pie определяют сектор как часть эллипса (окружности)

**Точка**

Поверхности, на которую программа может осуществлять вывод графики, соответствует объект Canvas. Свойство pixels, представляющее собой двумерный массив типа TColor, содержит информацию о цвете каждой точки графической поверхности. Используя свойство Pixels, можно задать требуемый цвет для любой точки графической поверхности, т. е. "нарисовать" точку. Например, инструкция

*Form1.Canvas.Pixels[10,10]:=clRed* окрашивает точку поверхности формы в красный цвет.

Размерность массива pixels определяется размером графической поверхности. Размер графической поверхности формы (рабочей области, которую также называют клиентской) задается значениями свойств ciientwidth и ClientHeight, а размер графической поверхности компонента image — значениями свойств width и Height. Левой верхней точке рабочей области формы соответствует элемент pixels [0,0], а правой нижней -Pixels[Ciientwidth - 1,ClientHeight - 1].

Свойство Pixels можно использовать для построения графиков. График строится, как правило, на основе вычислений по формуле. Границы диапазона изменения аргумента функции являются исходными данными. Диапазон изменения значения функции может быть вычислен. На основании этих данных можно вычислить масштаб, позволяющий построить график таким образом, чтобы он занимал всю область формы, предназначенную для вывода графика.

Например, если некоторая функция f(x) может принимать значения от нуля до 1000, и для вывода ее графика используется область формы высотой в 250 пикселов, то масштаб оси Y вычисляется по формуле: т = 250/1000. Таким образом, значению f(x) = 70 будет соответствовать точка с координатой Y =233. Значение координаты Y вычислено по формуле

*Y= h -f(x) х т = 250 - 70х(250/1000),* где h - высота области построения графика.

Обратите внимание на то, что точное значение выражения 250 - 70х(250/1000) равно 232,5. Но т. к. индексом свойства pixels, которое используется для вывода точки на поверхность Canvas, может быть только целое значение, то число 232,5 округляется к ближайшему целому, которым является число 233.

Следующая программа, текст которой приведен в листинге 5, используя свойство pixels, выводит график функции у = 2 sin(jc) e\*/5. Для построения графика используется вся доступная область формы, причем если во время работы программы пользователь изменит размер окна, то график будет выведен заново с учетом реальных размеров окна.

**Листинг 5. График функции**

**unit**grfunc\_;

**interface**

Windows, Messages, SysUtils, Classes,

Graphics, Controls, Forms, Dialogs;

**type**

TForm1 = class(TForm)

**procedure**FormPaint(Sender: TObject);

**procedure**FormResize(Sender: TObject);

private

{ Private declarations }

**public**

{Public declarations }

**end;**

**var** Forml: TForml;

**implementation**

{$R \*.DFM}

// Функция, график которой надо построить

**Function**f(x:real):real;

**begin**

f:=2\*Sin(x)\*exp(x/5) ;

**end;**

// строит график функции

**procedure**GrOfFunc;

**var** x1,x2:real; // границы изменения аргумента функции

y1,y2:real; // границы изменения значения функции

х:real; // аргумент функции

у:real; // значение функции в точке х

dx:real; // приращение аргумента

l,b:integer; // левый нижний угол области вывода графика

w,h:integer; // ширина и высота области вывода графика

mx,my:real; // масштаб по осям X и Y

х0,у0:integer; // точка — начало координат

**begin**

// область вывода графика

l:=10; // X — координата левого верхнего угла

b:=Forml.ClientHeight-20;

//У — координата левого верхнего угла

h:=Forml.ClientHeight-40; // высота

w:=Forml.Width-40; // ширина

x1:=0; // нижняя граница диапазона аргумента

х2:=25; // верхняя граница диапазона аргумента

dx:=0.01; // шаг аргумента

// найдем максимальное и минимальное значения

// функции на отрезке [x1,x2]

y1:=f(xl); // минимум

y2:=f(xl); //максимум

x:=x1;

**repeat**

У := f (х);

**if у < yl then**yl:=y;

**if у > у2 then**y2:=y;

х:=x+dx; until (x >= х2);

// вычислим масштаб

my:=h/abs(y2-yl); // масштаб по оси Y

mx:=w/abs(x2-xl); // масштаб по оси X

х0:=1; у0:=b-Abs(Round(y1\*my)) ;

**with**form1.Canvas **do begin**

// оси

MoveTo(l,b);LineTo(l,b-h); MoveTo(x0,y0);LineTo(x0+w,y0);

TextOut(l+5,b-h,FloatToStrF(y2,ffGeneral,6,3)); TextOut(l+5,b,FloatToStrF(yl,ffGeneral,6,3));

// построение графика

**x:=xl; repeat**

y:=f(x); Pixels[x0+Round(x\*mx),y0-Round(y\*my)]:=clRed;

x:=x+dx;

**until**(x >= x2);

**end; end;**

**procedure**TForm1.FormPaint(Sender: TObject);

**begin** GrOfFunc; **end;**

// изменился размер окна программы

**procedure**TForm1.FormResize(Sender: TObject);

**begin**

// очистить форму

forml.Canvas.FillRect(Rect(0,0,ClientWidth,ClientHeight));

// построить график

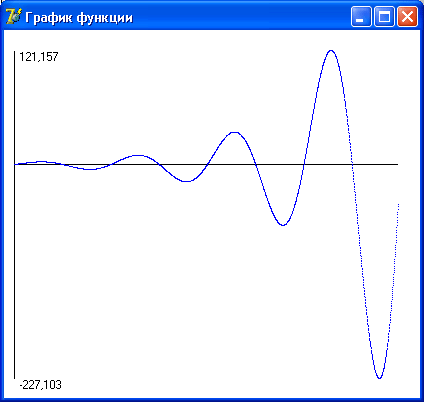
GrOfFunc;

**end; end.**

Основную работу выполняет процедура GrOfFunc, которая сначала вычисляет максимальное (у2) и минимальное (yl) значения функции на отрезке [x1l,x2]. Затем, используя информацию о ширине (Forml.Clientwidth -40) и высоте (Form1.ClientHeight - 40) области вывода графика, вычисляет масштаб по осям X (mх) иY(mу).

Высота и ширина области вывода графика определяется размерами рабочей (клиентской) области формы, т. е. без учета области заголовка и границ. После вычисления масштаба процедура вычисляет координату у горизонтальной оси (уо) и вычерчивает координатные оси графика. Затем выполняется непосредственное построение графика (рис.10).

Вызов процедуры GrOfFunc выполняют процедуры обработки событий onPaint и onFormResize. Процедура TForm1. FormPaint обеспечивает вычерчивание графика после появления формы на экране в результате запуска программы, а также после появления формы во время работы программы, например, в результате удаления или перемещения других окон, полностью или частично перекрывающих окно программы. Процедура TForm1.FormResize обеспечивает вычерчивание графика после изменения размера формы.



**Рис. 10.**График, построенный процедурой GrOfFunc

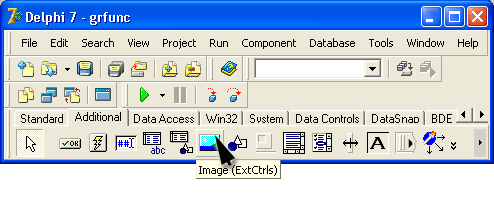
Приведенная программа довольно универсальна. Заменив инструкции в теле функции f (х), можно получить график другой функции. Причем независимо от вида функции ее график будет занимать всю область, предназначенную для вывода.

**Примечание**

Рассмотренная программа работает корректно, если функция, график которой надо построить, принимает как положительные, так и отрицательные значения. Если функция во всем диапазоне только положительная или только отрицательная, то в программу следует внести изменения. Какие — пусть это будет упражнением для читателя.

**Вывод иллюстраций**

Наиболее просто вывести иллюстрацию, которая находится в файле с расширением bmp, jpg или ico, можно при помощи компонента image, значок которого находится на вкладке **Additional**палитры (рис. 11).



**Рис. 11.**Значок компонента Image

В табл. 8 перечислены основные свойства компонента image.

**Таблица 8.**Свойства компонента image

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| Picture Width, Height  AutoSize  Strech  Visible | Иллюстрацию, которая отображается в поле компонента  Размер компонента. Если размер компонента меньше размера иллюстрации, и значение свойств AutoSize и strech равно False, то отображается часть иллюстрации  Признак автоматического изменения размера компонента в соответствии с реальным размером иллюстрации  Признак автоматического масштабирования иллюстрации в соответствии с реальным размером компонента. Чтобы было выполнено масштабирование, значение свойства AutoSize должно быть False  Отображается ли компонент, и, соответственно, иллюстрация, на поверхности формы |

Иллюстрацию, которая будет выведена в поле компонента image, можно задать как во время разработки формы приложения, так и во время работы программы.

Во время разработки формы иллюстрация задается установкой значения свойства picture путем выбора файла иллюстрации в стандартном диалоговом окне, которое появляется в результате щелчка на командной кнопке **Load**окна **Picture Editor**(рис. 12). Чтобы запустить Image Editor, нужно в окне **Object Inspector**выбрать свойство Picture и щелкнуть на кнопке с тремя точками.

Если размер иллюстрации больше размера компонента, то свойству strech нужно присвоить значение True и установить значения свойств width и Height пропорционально реальным размерам иллюстрации.

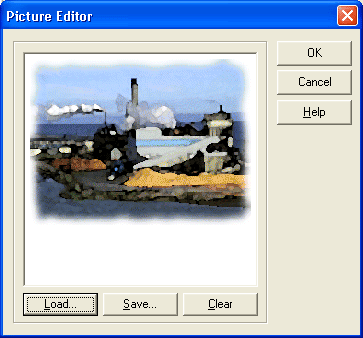
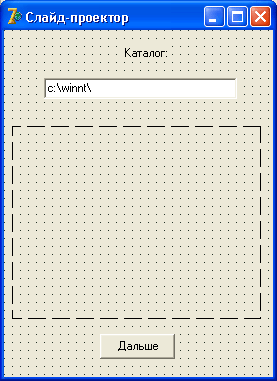
Чтобы вывести иллюстрацию в поле компонента image во время работы программы, нужно применить метод LoadFromFile к свойству Picture, указав в качестве параметра имя файла иллюстрации. Например, инструкция

Form1.Image1.Picture.LoadFromFile('e:\temp\bart.bmp')

загружает иллюстрацию из файла bart.bmp и выводит ее в поле вывода иллюстрации (imagel).

Метод LoadFromFile позволяет отображать иллюстрации различных графических форматов: BMP, WMF, JPEG (файлы с расширением jpg).

Следующая программа, ее текст приведен в листинге 6, использует компонент image для просмотра иллюстраций, которые находятся в указанном пользователем каталоге. Диалоговое окно программы приведено на рис. 13.



**Рис. 12.** Окно **Picture Editor Рис. 13.** Слайд-проектор

**Листинг 6. Слайд-проектор**

**Unit** shpic\_;

**interface**

**uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,

Dialogs, ExtCtrls, StdCtrls, Menu

**type**

TForm1 = **class**(TForm) Image1: ТImage;

Button1: TButton;

**procedure**FormActivate(Sender: TObject);

**procedure**ButtonlClick(Sender: TObject);

**private**

{ Private declarations }

**public**

{ Public declarations }

**end;**

**var** Form1: TForm1;

aSearchRec : TSearchRec;

aPath : **String;**// каталог, в котором находятся иллюстрации

aFile : **String;**// файл иллюстрации

iw,ih: integer; // первоначальный размер компонента Image

**implementation**

$R \*.DFM}

// изменение размера области вывода иллюстрации

// пропорционально размеру иллюстрации

**Procedure**Scalelmage;

**Var** pw, ph : integer; // размер иллюстрации

scaleX, scaleY : real; // масштаб по Х и Y

scale : real; // общий масштаб

**begin**

// иллюстрация уже загружена, получим ее размеры

pw := Form1.Image1.Picture.Width; ph := Form1.Image1.Picture.Height;

**if** pw > iw // ширина иллюстрации больше ширины компонента Image

**then** scaleX := iw/pw // нужно масштабировать

**else** scaleX := 1;

**if** ph > ih // высота иллюстрации больше высоты компонента

**then** scaleY := ih/ph // нужно масштабировать

**else**scaleY := 1;

// выберем наименьший коэффициент

**if** scaleX < scaleY **then**scale := scaleX **else**scale := scaleY;

// изменим размер области вывода иллюстрации

Form1.Image1.Height := Round(Form1.Image1.Picture.Height\*scale)

Form1.Image1.Width := Round(Form1.Image1.Picture.Width\*scale);

// т. к. Strech = True и размер области пропорционален

// размеру картинки, то картинка масштабируется без искажений

**end;**

// вывести первую иллюстрацию

**Procedure** FirstPicture;

var r : integer; // результат поиска файла

**begin**

aPath := 'f:\temp\';

r := FindFirst(aPath+'\*.bmp',faAnyFile,aSearchRec);

if **г = 0 then begin** // в указанном каталоге есть bmp-файл

aFile := aPath + aSearchRec.Name;

Form1.Image1.Picture.LoadFromFile(aFile); // загрузить иллюстрацию

Scalelmage; //-установить размер компонента

Image r := FindNext(aSearchRec); // найти следующий файл

**If** r = 0 **then** // еще есть файлы иллюстраций

Forml.Button1.Enabled := True;

**end; end;**

// вывести следующую иллюстрацию

**Procedure**NextPicture();

Var r : integer;

**begin**

aFile := aPath + aSearchRec.Name;

Forml.Image1.Picture.LoadFromFile(aFile);

Scalelmage;

// подготовим вывод следующей иллюстрации

r := FindNext(aSearchRec); // найти следующий файл

**if**r<>0 **then** Forml.Buttonl.Enabled := False; // больше нет иллюстраций

**end;**

**procedure**TForml.FormActivate(Sender: TObject);

**begin**

Image1.AutoSize := False; // запрет автоизменения размера компонента

Image1.Stretch := True; // разрешим масштабирование

// запомним первоначальный размер области вывода иллюстрации

iw := Imagel.Width; in := imagel.Height;

Button1.Enabled := False; // сделаем недоступной кнопку Дальше

FirstPicture; // вывести первую иллюстрацию

**end;**

//щелчок на кнопке Дальше

**Procedure** TForm1.Button1Click(Sender: TObject);

**begin**

NextPicture;

**end;end.**

Программа выполняет масштабирование выводимых иллюстраций без искажения, чего нельзя добиться простым присвоением значения True свойству strech. Загрузку и вывод первой и остальных иллюстраций выполняют соответственно процедуры FirstPicture и NextPicture. Процедура FrirstPicture использует функцию FindFirst для того, чтобы получить имя первого BMP-файла. В качестве параметров функции FindFirst передаются:

* имя каталога, в котором должны находиться иллюстрации;
* структура asearchRec, поле Name которой, в случае успеха, будет содержать имя файла, удовлетворяющего критерию поиска;
* маска файла иллюстрации.

Если в указанном при вызове функции FindFirst каталоге есть хотя бы один BMP-файл, значение функции будет равно нулю. В этом случае метод LoadFromFiie загружает файл иллюстрации, после чего вызывается функция scaieimage, которая устанавливает размер компонента пропорционально размеру иллюстрации. Размер загруженной иллюстрации можно получить, обратившись к свойствам*Form1.Image1.Picture.Width* и *Form1.Шmage1.Picture.Height*, значения которых не зависят от размера компонента Image.

**Битовые образы**

При работе с графикой удобно использовать объекты типа TBitMap (битовый образ). Битовый образ представляет собой находящуюся в памяти компьютера, и, следовательно, невидимую графическую поверхность, на которой программа может сформировать изображение. Содержимое битового образа (картинка) легко и, что особенно важно, быстро может быть выведено на поверхность формы или области вывода иллюстрации (image). Поэтому в программах битовые образы обычно используются для хранения небольших изображений, например, картинок командных кнопок.

Загрузить в битовый образ нужную картинку можно при помощи метода LoadFromFlie, указав в качестве параметра имя BMP-файла, в котором находится нужная иллюстрация.

Например, если в программе объявлена переменная pic типа TBitMap, то после выполнения инструкции *pic.LoadFromFiie('е:\images\aplane.bmp')*

битовый образ pic будет содержать изображение самолета.

Вывести содержимое битового образа (картинку) на поверхность формы или области вывода иллюстрации можно путем применения метода Draw к соответствующему свойству поверхности (canvas). Например, инструкция *Image1.Canvas.Draw(x,у, bm)*

выводит картинку битового образа bm на поверхность компонента image 1 (параметры х и у определяют положение левого верхнего угла картинки на поверхности компонента).

Если перед применением метода Draw свойству Transparent объекта TBitMap присвоить значение True, то фрагменты рисунка, окрашенные цветом, совпадающим с цветом левого нижнего угла картинки, не будут выведены — через них будет как бы проглядывать фон. Если в качестве "прозрачного" нужно использовать цвет, отличный от цвета левой нижней точки рисунка, то свойству Transparentcoior следует присвоить значение символьной константы, обозначающей необходимый цвет.

Следующая программа, текст которой приведен в листинге 7, демонстрирует использование битовых образов для формирования изображения из нескольких элементов.

**Листинг 7. Использование битовых образов**

**unit**aplanes\_; **interface**

uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

**type**

TForml = **class**(TForm)

**procedure**FormPaint(Sender: TObject);

**private**

{ Private declarations }

**public**

{ Public declarations }

**end;**

**var** Forml: TForm1;

sky,aplane: TBitMap; // битовые образы: небо и самолет

**implementation**

($R \*.DFM}

**procedure**TForm1.FormPaint(Sender: TObject);

**begin**

// создать битовые образы

sky := TBitMap.Create; aplane := TBitMap.Create;

// загрузить картинки

sky.LoadFromFile('sky.bmp');

aplane.LoadFromFile('aplane.bmp') ;

Form1.Canvas.Draw(0,0,sky); // отрисовка фона

Form1.Canvas.Draw(20,20,aplane); // отрисовка левого самолета

aplane.Transparent:=True;

// теперь элементы рисунка, цвет которых совпадает с цветом левой нижней точки

// битового образа, не отрисовываются

Form1.Canvas.Draw(120,20,aplane); // отрисовка правого самолета

// освободить память

sky.free; aplane.free;

**end;**

**end.**

После запуска программы в окне приложения (рис. 14) появляется изображение летящих на фоне неба самолетов. Фон и изображение самолета -битовые образы, загружаемые из файлов. Белое поле вокруг левого самолета показывает истинный размер картинки битового образа aplane. Белое поле вокруг правого самолета отсутствует, т. к. перед его выводом свойству Transparent битового образа было присвоено значение True.



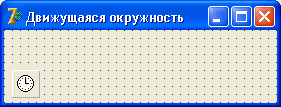
**Рис. 14.**Влияние значение свойства **Transparent** на вывод изображения

**Мультипликация**

Под мультипликацией обычно понимается движущийся и меняющийся рисунок. В простейшем случае рисунок может только двигаться или только меняться.

Как было показано выше, рисунок может быть сформирован из графических примитивов (линий, окружностей, дуг, многоугольников и т. д.). Обеспечить перемещение рисунка довольно просто: надо сначала вывести рисунок на экран, затем через некоторое время стереть его и снова вывести этот же рисунок, но уже на некотором расстоянии от его первоначального положения. Подбором времени между выводом и удалением рисунка, а также расстояния между старым и новым положением рисунка (шага перемещения), можно добиться того, что у наблюдателя будет складываться впечатление, что рисунок равномерно движется по экрану.

Следующая простая программа, текст которой приведен в листинге 8, а вид формы — на рис. 15, демонстрирует движение окружности от левой к правой границе окна программы.



**Рис. 15.**Форма программы **Движущаяся окружность**

**Листинг 8. Движущаяся окружность**

**unit** mcircle\_;

**interface**

**uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ExtCtrls, StdCtrls;

**type**

TForm1 = class(TForm) Timer1: TTimer;

procedure Timer1Timer(Sender: TObject};

procedure FormActivate(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

implementation

{$R \*.DFM}

**Var** Form1: TForml;

x,y: byte; // координаты центра окружности

dx: byte; // приращение координаты x при движении окружности

// стирает и рисует окружность на новом месте

**procedure** Ris;

begin

// стереть окружность

form1.Canvas.Pen.Color:=form1.Color; form1.Canvas.Ellipse(x,y,x+10,y+10);

x:=x+dx;

// нарисовать окружность на новом месте

form1.Canvas.Pen.Color:=clBlack; form1.Canvas.Ellipse(x,y, x+10, y+10) ;

end;

// сигнал от таймера

**procedure** TForm1.Timer1Timer(Sender: TObject);

begin Ris; end;

**procedure** TForm1.FormActivate(Sender: TObject);

begin

x:=0; y:=10; dx:=5; timer1.Interval:=50;

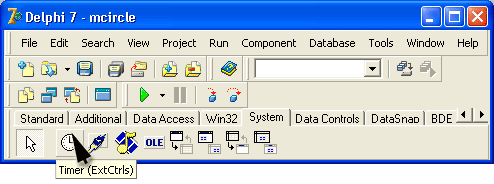
// период возникновения события OnTimer —0.5 сек

form1.canvas.brush.color:=forml.color;

end; end.

Основную работу выполняет процедура Ris, которая стирает окружность и выводит ее на новом месте. Стирание окружности выполняется путем перерисовки окружности поверх нарисованной, но цветом фона.

Для обеспечения периодического вызова процедуры Ris в форму программы добавлен невизуальный компонент Timer (таймер), значок которого находится на вкладке System палитры компонентов (рис. 16). Свойства компонента Timer, перечислены в табл. 9.



**Рис. 16.**Значок компонента Timer

**Таблица 9.**Свойства компонента Timer

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| Name Interval    Enabled | Имя компонента. Используется для доступа к компоненту Период генерации события OnTimer. Задается в миллисекундах  Разрешение работы. Разрешает (значение True) или запрещает (значение False) генерацию события OnTimer |

Добавляется компонент Timer к форме обычным образом, однако, поскольку компонент Timer является невизуальным, т. е. во время работы программы не отображается на форме, его значок можно поместить в любое место формы.

Компонент Timer генерирует событие OnTimer. Период возникновения события OnTimer измеряется в миллисекундах и определяется значением свойства Interval. Следует обратить внимание на свойство Enabled. Оно дает возможность программе "запустить" или "остановить" таймер. Если значение свойства Enabled равно False, то событие OnTimer не возникает.

Событие onTimer в рассматриваемой программе обрабатывается процедурой TimeriTimer, которая, в свою очередь, вызывает процедуру Ris. Таким образом, в программе реализован механизм периодического вызова процедуры Ris.

**Примечание**

Переменные х, у (координаты центра окружности) и dx (приращение координаты х при движении окружности) объявлены вне процедуры Ris, т. е. они являются глобальными. Поэтому надо не забыть выполнить их инициализацию (в программе инициализацию глобальных переменных реализует процедура FormActivate).

**Метод базовой точки**

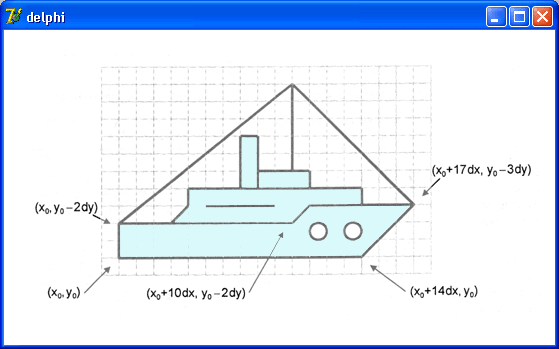
При программировании сложных изображений, состоящих из множества элементов, используется метод, который называется методом базовой точки. Суть этого метода заключается в следующем:

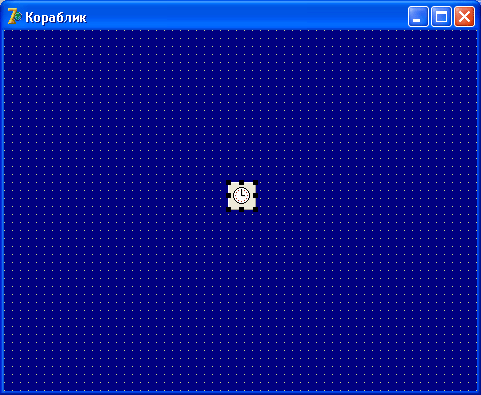
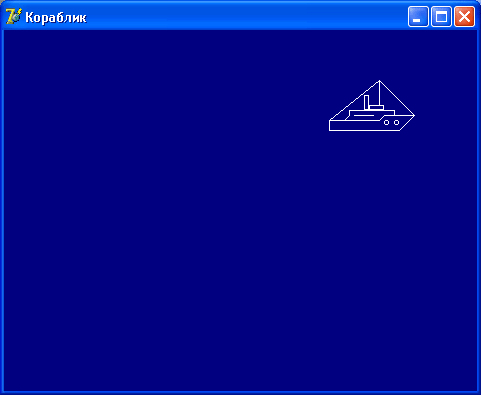
1. Выбирается некоторая точка изображения, которая принимается за базовую.

2. Координаты остальных точек отсчитываются от базовой точки.

3. Если координаты точек изображения отсчитывать от базовой в относительных единицах, а не в пикселах, то обеспечивается возможность масштабирования изображения.

На рис. 17 приведено изображение кораблика. Базовой точкой является точка с координатами (X0 Y0). Координаты остальных точек отсчитываются именно от этой точки.





**Рис. 17.**Определение координат изображения относительно базовой точки

В листинге 9 приведен текст программы, которая выводит на экран изображение перемещающегося кораблика.

**Листинг 9. Кораблик**

**unit** ship\_;

**interface**

**uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, ExtCtrls;

**type**

TForm1 = class(TForm)

Timer1: TTimer;

**procedure**Timer1Timer(Sender: TObject);

**procedure**FormActivate(Sender: TObject);

**private**

{ Private declarations } **public**

{ Public declarations } **end;**

**Var** Form1: TForm1;

x,y: integer; // координаты корабля (базовой точки)

**implementation**

{$R \*.DFM}

// вычерчивает кораблик

**procedure**Titanik(x,y: integer; // координаты базовой точки

color: TColor); // цвет корабля

**const**dx = 5; dy = 5;

**var** buf: TColor;

**begin**

**with**form1.canvas **do begin**

buf:=pen.Color; // сохраним текущий цвет

pen.Color:=color;

// установим нужный цвет

// рисуем . . .

// корпус

MoveTo(x,y);

LineTo(x,y-2\*dy) ; LineTo (x+10\*dx, y-2\*dy) ;

LineTo (x+ll\*dx, y-3\*dy) ; LineTo (x+17\*dx,y-3\*dy) ;

LineTo (x+14\*dx, y) ; LineTo (x,y) ;

// надстройка

MoveTo(x+3\*dx,y-2\*dy) ;

LineTo (x+4\*dx, y-3\*dy) ; LineTo (x+4\*dx, y-4\*dy) ;

LineTo (x+13\*dx,y-4\*dy) ; LineTo (x+13\*dx, y-3\*dy) ;

MoveTo(x+5\*dx,y-3\*dy) ; LineTo (x+9\*dx, y-3\*dy) ;

// капитанский мостик

Rectangle (x+8\*dx, y-4\*dy, x+ll\*dx, y-5\*dy)

// труба

Rectangle (x+7\*dx, y-4\*dy, x+8\*dx, y-7\*dy) ;

// иллюминаторы

Ellipse (x+ll\*dx,y-2\*dy,x+12\*dx,y-l\*dy) ;

Ellipse (x+13\*dx, y-2\*dy, x+14\*dx, y-l\*dy) ;

// мачта

MoveTo(.x+10\*dx,y-5\*dy) ; LineTo(x+10\*dx,y-10\*dy);

// оснастка

MoveTo(x+17\*dx,y-3\*dy);

LineTo(x+10\*dx,y-10\*dy); LineTo(x,y-2\*dy);

pen.Color:=buf; // восстановим старый цвет карандаша

**end; end;**

// обработка сигнала таймера

**procedure**TForm1.Timer1Timer(Sender: TObject);

**begin**

Titanik(x,y,form1.color); // стереть рисунок

**if**x < Form1.ClientWidth **then**x := x+5

**else begin**// новый рейс x := 0;

у := Random(50) + 100;

**end;**

Titanik(x,у,clWhite); // нарисовать в новой точке end;

**procedure**TForml.FormActivate(Sender: TObject);

**begin**

x:=0; y:=100;

Form1.Color:=clNavy;

Timerl.Interval := 50; // сигнал таймера каждые 50 миллисекунд

**end;**

**end.**

Отрисовку и стирание изображения кораблика выполняет процедура Titanik, которая получает в качестве параметров координаты базовой точки и цвет, которым надо вычертить изображение кораблика. Если при вызове процедуры цвет отличается от цвета фона формы, то процедура рисует кораблик, а если совпадает — то "стирает". В процедуре Titanik объявлены константы dx и dy, определяющие шаг (в пикселах), используемый при вычислении координат точек изображения. Меняя значения этих констант, можно проводить масштабирование изображения.

**Использование битовых образов**

В предыдущем примере изображение формировалось из графических примитивов. Теперь рассмотрим, как можно реализовать перемещение одного сложного изображения на фоне другого, например перемещение самолета на фоне городского пейзажа.

Эффект перемещения картинки может быть создан путем периодической перерисовки картинки с некоторым смещением относительно ее прежнего положения. При этом предполагается, что перед выводом картинки в новой точке сначала удаляется предыдущее изображение. Удаление картинки может быть выполнено путем перерисовки всей фоновой картинки или только той ее части, которая перекрыта битовым образом движущегося объекта.

В рассматриваемой программе используется второй подход. Картинка выводится применением метода Draw к свойству canvas компонента Image, a стирается путем копирования (метод copyRect) нужной части фона из буфера на поверхность компонента Image.

Форма программы приведена на рис. 10.18, а текст — в листинге 10.10.

Компонент image используется для вывода фона, а компонент Timer — для организации задержки между циклами удаления и вывода на новом месте изображения самолета.

**Листинг 10. Летящий самолет**

**unit**anim\_;

**interface**

**uses** Windows, Messages, SysUtils, Classes, Graphics, Controls,Forms, Dialogs, ExtCtrls, StdCtrls, Buttons;

**type**

TForm1 = **class**(TForm)

Timer1: TTimer;

Image1: Tlmage;

**procedure**FormActivate(Sender: TObject);

**procedure**Timer1Timer(Sender: TObject);

**procedure**FormClose(Sender: TObject;

**var** Action: TCloseAction); **private**

{ Private declarations } **public**

{ Public declarations } **end;**

**Var** Form1: TForm1;

**implementation**

{$R \*.DFM}

**var**

Back, bitmap, Buf : TBitMap; // фон, картинка, буфер

BackRct : TRect; // область фона, которая должна быть

// восстановлена из буфера

BufRet: Trect; // область буфера, которая используется для

// восстановления фона

х,у:integer; // текущее положение картинки

W,H: integer; // размеры картинки

**procedure**TForm1.FormActivate(Sender: TObject);

**begin**

// создать три объекта — битовых образа

Back := TBitmap.Create; // фон

bitmap := TBitmap.Create; // картинка

Buf := TBitmap.Create; // буфер

// загрузить и вывести фон

Back.LoadFromFile('factory.bmp');

Form1.Image1.canvas.Draw(0,0,Back);

// загрузить картинку, которая будет двигаться

bitmap.LoadFromFile('aplane.bmp');

// определим "прозрачный" цвет

bitmap.Transparent := True;

bitmap.TransParentColor := bitmap.canvas.pixels[1,1];

// создать буфер для сохранения копии области фона,

// на которую накладывается картинка

W:= bitmap.Width; Н:= bitmap.Height;

Buf.Width:= W; Buf.Height:=H;

Buf.Palette:=Back.Palette;

// Чтобы обеспечить соответствие палитр //

Buf.Canvas.CopyMode:=cmSrcCopy;

// определим область буфера, которая

// будет использоваться

// для восстановления фона

BufRct:=Bounds(0,0,W,H);

// начальное положение картинки

х := -W; у := 20;

// определим сохраняемую область фона

BackRct:=Bounds(x,y,W,H); // и сохраним ее

Buf.Canvas.CopyRect(BufRet,Back.Canvas,BackRct);

**end;**

// обработка сигнала таймера

**procedure**TForm1.Timer1Timer(Sender: TObject);

**begin**

// восстановлением фона (из буфера) удалим рисунок

Forml.image1.canvas.Draw(x,у,Buf);

x:=x+2;

**if** x>fоrm1.Image1.Width **then**x:=-W;

// определим сохраняемую область фона

BackRct:=Bounds(x,у,W,H);

// сохраним ее копию

Buf.Canvas.CopyRect(BufRct,Back.Canvas,BackRct);

// выведем рисунок

Forml.image1.canvas.Draw(x,y,bitmap);

**end;**

// завершение работы программы

**procedure**TForm1.FormClose(Sender: TObject; var Action: TCloseAction);

**begin**

// освободим память, выделенную

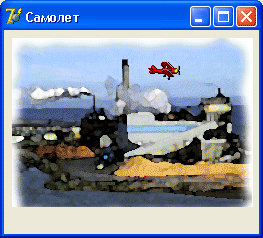
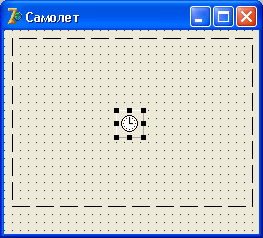
// для хранения битовых образов

Back.Free;

bitmap.Free;

Buf.Free;

**end; end.**



**Рис. 18.**Форма программы **Самолет**

Для хранения битовых образов (картинок) фона и самолета, а также копии области фона, перекрываемой изображением самолета, используются объекты типа TBitMap, которые создаются динамически процедурой FormActivate. Эта же процедура загружает из файлов картинки фона (factory.bmp) и самолета (aplane.bmp), а также сохраняет область фона, на которую первый раз будет накладываться картинка.

Сохранение копии фона выполняется при помощи метода CopyRect, который позволяет выполнить копирование прямоугольного фрагмента одного битового образа в другой. Объект, к которому применяется метод CopyRect, является приемником копии битового образа. В качестве параметров методу передаются координаты и размер области, куда должно быть выполнено копирование, поверхность, откуда должно быть выполнено копирование, а также положение и размер копируемой области. Информация о положении и размере копируемой в буфер области фона, на которую будет наложено изображение самолета и которая впоследствии должна быть восстановлена из буфера, находится в структуре BackRct типа TRect. Для заполнения этой структуры используется функция Bounds.

Следует обратить внимание на то, что начальное значение переменной х, которая определяет положение левой верхней точки битового образа движущейся картинки, — отрицательное число, равное ширине битового образа картинки. Поэтому в начале работы программы изображение самолета не появляется, картинка отрисовывается за границей видимой области. С каждым событием OnTimer значение координаты х увеличивается, и на экране появляется та часть битового образа, координаты которой больше нуля. Таким образом, у наблюдателя создается впечатление, что самолет вылетает из-за левой границы окна.

**Просмотр мультика**

Теперь рассмотрим, как можно реализовать вывод в диалоговом окне программы простого "мультика", подобного тому, который можно видеть в диалоговом окне **Установка связи**при подключении к Internet .

Эффект бегущего между телефоном и компьютером красного квадратика достигается за счет того, что в диалоговое окно выводятся сменяющие друг друга картинки.

Кадры мультика обычно находятся в одном файле или в одном ресурсе. Перед началом работы программы они загружаются в буфер, в качестве которого удобно использовать объект типа TBitMap. Задача процедуры, реализующей вывод мультика, состоит в том, чтобы выделить очередной кадр и вывести его в нужное место формы.

Вывести кадр на поверхность формы можно применением метода copyRect к свойству canvas этой формы. Метод CopyRect копирует прямоугольную область одной графической поверхности на другую.

Инструкция применения метода CopyRect в общем виде выглядит так:

Canvas1.CopyRect(Область1, Canvas2, 06ласть2)

где:

canvas1 — графическая поверхность, на которую выполняется копирование;

Canvas2 — графическая поверхность, с которой выполняется копирование;

параметр Область2 —- задает положение и размер копируемой прямоугольной области, а параметр областьi — положение копии на поверхности Canvas1.

В качестве параметров область! и область2 используются структуры типа TRect, поля которых определяют положение и размер области.

Заполнить поля структуры TRect можно при помощи функции Bounds, инструкция обращения к которой в общем виде выглядит так:

Bounds(x,у,Width,Height)

где:

х и у — координаты левого верхнего угла области;

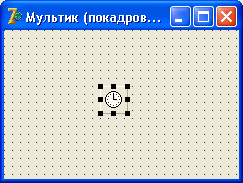
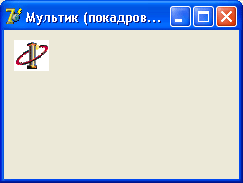
width и Height — ширина и высота области.

Следующая программа, текст которой приведен в листинге 12, выводит в диалоговое окно простой мультик — дельфийскую колонну, вокруг которой "летает" некоторый объект. На рис. 19 приведены кадры этого мультика (содержимое файла film.bmp).

Диалоговое окно программы приведено на рис. 20, оно содержит один единственный компонент — таймер.

http://www.bourabai.kz/einf/Delphi/Glava10/27a.gifhttp://www.bourabai.kz/einf/Delphi/Glava10/27b.gifhttp://www.bourabai.kz/einf/Delphi/Glava10/27c.gif

**Рис. 19.**Кадры мультика



**Рис. 20.** Форма программы

**Листинг 12. Мультик (использование метода CopRect)**

**unit** multik ;

interface

**uses** Windows, Messages, SysUtils, Classes,Graphics, Controls, Forms, Dialogs,ExtCtrls, StdCtrls;

**type**

TForm1 = class(TForm)

Timer1: TTimer;

**procedure**FormActivate(Sender: TObject);

**procedure**Timer1Timer(Sender: TObject);

**private**

{ Private declarations }

**public**

{ Public declarations }

**end;**

**var** Form1l: TForm1;

**implementation**

($R \*.DFM}

**const**

FILMFILE = 'film2.bmp'; // фильм — bmp-файл

N\_KADR=12; // кадров в фильме (для данного файла)

**var**

Film: TBitMap; // фильм — все кадры

WKadr,HKadr: integer; // ширина и высота кадра

CKadr: integer; // номер текущего кадра

RectKadr: TRect; // положение и размер кадра в фильме

Rect1 : Trect; // координаты и размер области отображения фильма

**procedure**TForm1.FormActivate(Sender: TObject);

**begin**

Film := TBitMap.Create;

Film.LoadFromFile(FILMFILE);

WKadr := Round(Film.Width/N\_Kadr);

HKadr := Film.Height;

Rect1 := Bounds(10,10,WKadr,HKadr);

Ckadr:=0;

Form1.Timerl.Interval := 150; // период обновления кадров — 0.15 с

Form1.Timerl.Enabled:=True; // запустить таймер

**end;**

// отрисовка кадра **procedure**DrawKadr;

**begin**

// определим положение текущего кадра в фильме

RectKadr:=Bounds(WKadr\*CKadr,0,WKadr,HKadr);

// вывод кадра из фильма

Form1.Canvas.CopyRect(Rect1,Film\*.Canvas,RectKadr);

// подготовимся к выводу следующего кадра

CKadr := CKadr+1;

**if** CKadr = N\_KADR **then**CKadr:=0;:

**end;**

// обработка сигнала от таймера

**procedure**TForm1.Timer1Timer(Sender: TObject);

**begin**

DrawKadr;

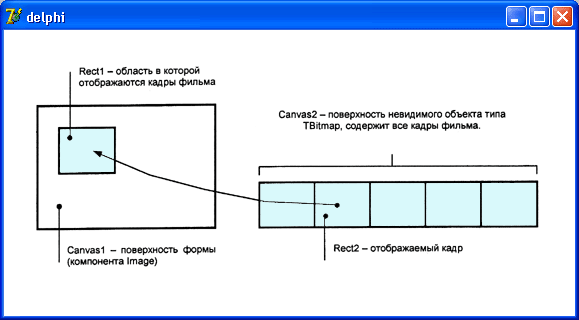
**end;**

**end.**

Программа состоит из трех процедур. Процедура TForm1. FormActivate создает объект Film и загружает в него фильм — BMP-файл, в котором находятся кадры фильма. Затем, используя информацию о размере загруженного битового образа, процедура устанавливает значения характеристик кадра: высоту и ширину.

После этого создается объект Kadr (типа TBitMap), предназначенный для хранения текущего кадра. Следует обратить внимание, что после создания объекта Kadr принудительно устанавливаются значения свойств width и Height. Если этого не сделать, то созданный объект будет существовать, однако память для хранения битового образа не будет выделена. В конце своей работы процедура TForml. FormActivate устанавливает номер текущего кадра и запускает таймер.

Основную работу в программе выполняет процедура DrawKadr, которая выделяет из фильма очередной кадр и выводит его в форму. Выделение кадра и его отрисовку путем копирования фрагмента картинки с одной поверхности на другую выполняет метод copyRect (рис. 21), которому в качестве параметров передаются координаты области, куда нужно копировать, поверхность и положение области, откуда нужно копировать. Положение фрагмента в фильме, т. е. координата х левого верхнего угла, определяется умножением ширины кадра на номер текущего кадра. Запускает процедуру DrawKadr процедура TForm1.Timer1Timer, обрабатывающая событие OnTiner.



**Рис. 21.** Инструкция Canvas1. CopyRect (Rect1, Canvas2, Rect2) **копирует в область** Rectl **поверхности** Canvasl **область** Rect2 **с поверхности** Canvas2

# **Тема.** Диаграммы

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Технология создания диаграммы

2 Настройка свойств компонента

**Вопросы для самоконтроля:**

1Как выбрать тип диаграммы

3 Опишите основные свойства диаграммы

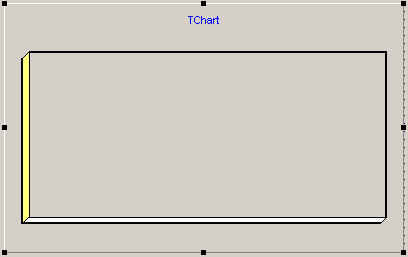
**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Компонент Delphi **Chart**располагается на странице Additional Палитры компонентов. Его разработал Дэвид Бернеда. Это богатый возможностями, очень мощный компонент, позволяющий строить двух и трехмерные красивые диаграммы на основе разнообразных данных. Он имеет большое количество разнообразных свойств. Часть из них, в свою очередь, являются объектами и обладают собственными свойствами.

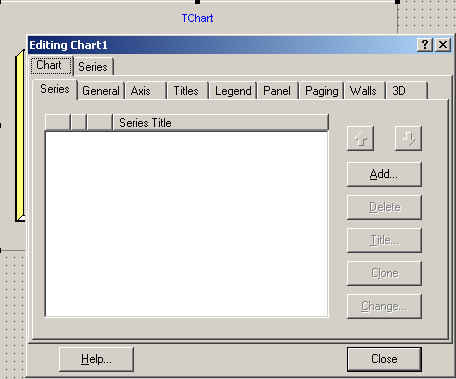
Значок на Палитре компонентов имеет вид. [значок компонента Chart](http://delphi-prg.ru/wp-content/uploads/2012/05/chart1.jpg)

Расположенный компонент на форме будет выглядеть подобно рисунку:

[](http://delphi-prg.ru/wp-content/uploads/2012/05/chart2.jpg)

Создать диаграмму можно, также с помощью мастера. Для запуска мастера из меню запускаем File -> New -> Other  и попадаем в окно New Items, там на вкладке Business выбираем TeeChartWisard. Мастер потребует уточнения, на основе базы данных или нет будет строиться график. Пусть он генерируется программой (переключатель Non Database Chart - не на основе базы данных). Следующий шаг - выбор вида диаграммы, она может быть двух или трех мерной (выбор переключателя 2D или 3D). Нажав Next определяем нужна ли нам легенда (опция Show Legend), желтые подсказки, рядом с диаграммой, включает флажок Show Marks. Нажав на кнопку Finish мы получим диаграмму, заполненную случайно сгенерированными числами.

Настройка свойств компонента Chart происходит в редакторе Editing Chart. Вызвать его можно двойным щелчком по компоненту или используя свойство SeriesList Инспектора объектов.

[](http://delphi-prg.ru/wp-content/uploads/2012/05/editingchart1.jpg)

Основные параметры диаграммы определяются на вкладке Chart (диаграмма), она, в свою очередь, состоит из набора дополнительных панелей.

Панель Ряд данных (Series) важная, в ней можно добавить на один график несколько диаграмм с помощью кнопки Add. При этом над значениями данных можно выполнять операции задаваемые во вкладке Series -> Data Source выбрав function выбрав функцию из списка function.

Панель Общие (General) содержит следующие элементы управления:

* кнопка Export - экспорт изображения в файл,
* кнопка Print Preview - предварительный просмотр и печать диаграммы,
* панель Zoom - масштабирование,
* панель AllowScroll - отвечает за прокрутку изображения.

Средства панели Оси (Axis) регулируют настройку координатных осей, заголовков, их масштаб, шаг пунктирной сетки и многое другое.

Панель Заголовков (Titles) помогает оформить заголовки.

Панель Легенда (Legend) отвечает за внешний вид и содержание легенды.

Панель Панель (Panel) задает оформление панели основы: цвет и форму границы панели (можно сделать так, что диаграмма будет расположена на панели имеющей градиентную заливку).

Панель Страницы (Paging), позволяет разделить диаграммы на страницы. Для этого необходимо в поле (точки на странице) Points per Page подобрать соответствующее значение.

Панель Границы (Walls) отвечает за цвет и границы диаграммы.

Панель 3D дает возможность настройки 3D эффектов, (изменение масштаба, положения в пространстве), простым передвиганием ползунков управления.

Вкладка Ряды данных (Series) отвечает за оформление каждого ряда данных (графиков добавленных с помощью вкладки Chart). Текущий ряд данных задается с помощью раскрывающегося списка. Наиболее важная панель Источник данных данных (Data Sourece). В ней можно выбрать отказ от генерации значений (No Data), создать случайные  значения (Random Values) или сформировать значения, как результат применения функции (список Function) к значениям рядов.

# **Тема.** Элементы управления. Отображение заголовков

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Управляющие элементы для разных окно

2 Заголовок окна

**Вопросы для самоконтроля:**

1Перечислите элементы управления, используемые в окнах приложений

2 Какие элементы управления можно использовать в диалоговых окнах

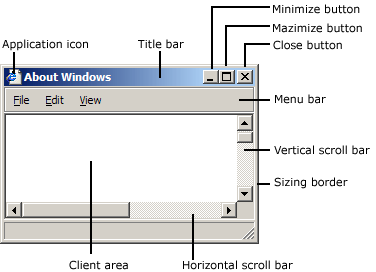
3 Для чего может быть использован заголовок окна

4 Перечислите типы окон

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

Окно приложения может включать в себя элементы вроде заголовка (title bar), оконного меню (window menu, ранее известного как "системное меню"), меню (menu bar), кнопок минимизации и максимизации, кнопки восстановления (restore button), кнопки закрытия (close button), бордюра (border), клиентской области (client area), горизонтальной и вертикальной полосок прокрутки (scroll bar). Главное окно приложений часто включает в себя все эти элементы сразу. Рисунок ниже иллюстрирует эти элементы в типичном главном окне программы:

[](http://2.bp.blogspot.com/-LR9iRydjEa8/T1iQYftKeuI/AAAAAAAADf8/nwAzhLLPWxI/s1600/ms632597.cswin_02(en-us,VS.85).png)

**Клиентская область**

*Клиентская область* (client area) является той частью окна, где приложение показывает основную информацию (текст или графику). Например, текстовый редактор отображает документ в клиентской области главного окна. Чтобы управлять клиентской областью окна (принимать ввод пользователя и отображать в ней информацию), приложение должно указать функцию обратного вызова, называемую оконной процедурой (window procedure). См. [Window Procedures](http://msdn.microsoft.com/en-us/library/ms632593(VS.85).aspx) для дальнейшего обсуждения.

**Неклиентская область**

Заголовок, меню, оконное меню, кнопки минимизации, максимизации, закрытия и восстановления, бордюр и полосы прокрутки собирательно называются *неклиентской областью* окна (nonclient area). Система автоматически управляет большинством аспектов неклиентской области; приложение же управляет поведением клиентской области окна.

*Заголовок окна* (title bar) показывает значок окна (указывается приложением) и строку текста; обычно текст указывает название приложения и цель окна. Приложение указывает значок и текст при создании окна. Кроме того заголовок также позволяет перемещать окно по экрану, используя мышь или иное указующее устройство.

Большинство приложений добавляют окнам *меню* (menu bar), которое включает в себя команды, поддерживаемые приложением. Элементы в полоске меню соответствуют категориям команд меню. Щелчок по элементу меню обычно приводит к его "раскрытию": показу всплывающего меню с элементами, соответствующими заданной категории. Щёлкая по элементу-команде, пользователь просит приложение выполнить эту задачу.

*Оконное меню* (window menu) создаётся и управляется системой (и поэтому оно ранее называлось системным меню). Оно содержит стандартный набор команд по управлению окном - вроде изменения размера и положения окна, его закрытия.

Кнопки в верхнем-правом углу окна влияют на размер и положение окна. Когда пользователь щёлкает по *кнопке максимизации* (maximize button), система увеличивает окно до размера текущего монитора, так что оно займёт весь рабочий стол минус панель задач (в одномониторной конфигурации). Одновременно с этим система заменит кнопку максимизации кнопкой восстановления. Когда вы щёлкаете по *кнопке восстановления* (restore button), система восстанавливает окно на его исходной позиции и с исходным размером. Когда вы щёлкаете по *кнопке минимизации* (minimize button), система уменьшает окно до размера её кнопки на панели задач, размещает окно поверх его кнопки на панели задач и показывает кнопку на панели задач в поднятом (нормальном) состоянии (прим. пер.: [это описание - упрощение](http://www.transl-gunsmoker.ru/2009/10/blog-post.html)). Чтобы восстановить приложение с его исходными положением и размерами - щёлкните по кнопке в панели задач. Когда вы щёлкаете по *кнопке закрытия*(close button), приложение выходит.

*Бордюр* (border) - это область по периметру окна, которая позволяет пользователю изменять размер окна, используя мышь или иное указующее устройство.

*Полосы прокрутки* (scroll bar) переводят ввод с клавиатуры и мыши в смещения, которые используются приложением для сдвига содержимого клиентской области. К примеру, приложение текстовый редактор может показывать большой документ с вертикальной полосой прокрутки, чтобы пользователь мог использовать её для перемещения по документу.

**Элементы управления и диалоговые окна**

Приложение может создавать несколько типов окон в дополнение к его главному окну, включая элементы управления (controls) и диалоговые окна (dialog box).

*Элемент управления* (control). - это окно, которое приложение использует для получения информации от пользователя. Приложения также используют элементы управления, чтобы управлять определённой возможностью приложения. К примеру, элементами управления являются кнопки, списки, поля ввода, дополнительные полосы прокрутки, панели, панели инструментов и т.п.

Элементы управления не могут использовать сами по себе и всегда используются с другим окном - обычно, диалоговым окном. *Диалоговое окно* (dialog box) - это окно, которое содержит один или более элементов управления. Приложение использует диалоговые окна, чтобы запросить у пользователя информацию для выполнения команды. К примеру, приложение, которое имеет команду открытия файла, может показывать диалоговое окно, которое включает в себя элементы управления для указания имени и пути файла. Диалоговые окна обычно не используют те же компоненты окна, что и главные окна приложения. Большинство диалоговых окон имеют заголовок, системное меню, кнопки минимизации/максимизации и закрытия, бордюр (часто - не позволяющий менять размеры окна) и клиентскую область, но они обычно не имеют меню и полос прокрутки. Для получения дополнительной информации - см. [Dialog Boxes](http://msdn.microsoft.com/en-us/library/ms632588(VS.85).aspx).

*Окно-сообщение* (message box) - это специальное диалоговое окно, которое показывает сообщение, предупреждение или сообщение об ошибке пользователю. К примеру, окно-сообщение может использоваться, чтобы сообщить пользователю о невозможности выполнения запрошенной им операции. Для дальнейшей информации - см. [Message Boxes](http://msdn.microsoft.com/en-us/library/ms644994(VS.85).aspx#message_boxes#message_boxes).

**Оконные атрибуты**

Приложение должно указывать следующую информацию при создании окна (за исключением [оконного описателя](http://www.transl-gunsmoker.ru/2012/03/windows.html#window_handle), который создаёт сама функция создания окон):

* [Имя класса](http://www.transl-gunsmoker.ru/2012/03/windows.html#class_name)
* [Имя окна](http://www.transl-gunsmoker.ru/2012/03/windows.html#window_name)
* [Стиль окна](http://www.transl-gunsmoker.ru/2012/03/windows.html#window_style)
* [Дополнительный стиль окна](http://www.transl-gunsmoker.ru/2012/03/windows.html#extended_window_style)
* [Положение](http://www.transl-gunsmoker.ru/2012/03/windows.html#position)
* [Размер](http://www.transl-gunsmoker.ru/2012/03/windows.html#Size)
* [Описатель родительского окна **или** окна-владельца](http://www.transl-gunsmoker.ru/2012/03/windows.html#parent_owner_handle)
* [Описатель меню **или** идентификатор дочернего окна](http://www.transl-gunsmoker.ru/2012/03/windows.html#menu_handle_child_id)
* [Описатель экземпляра](http://www.transl-gunsmoker.ru/2012/03/windows.html#app_instance_handle)
* [Пользовательские данные](http://www.transl-gunsmoker.ru/2012/03/windows.html#creation_data)
* [Оконный описатель](http://www.transl-gunsmoker.ru/2012/03/windows.html#window_handle)

Эти оконные атрибуты описываются ниже.

**Имя класса**

Каждое окно принадлежит оконному классу. Приложение должно зарегистрировать оконный класс до создания окна этого класса. *Оконный класс* (window class) определяет большинство аспектов поведения и отображения окна. Главным компонентом оконного класса является *оконная процедура* (window procedure) - функция обратного вызова, которая принимает весь ввод пользователя и все запросы, отправленные окну. Система предоставляет ей ввод и запросы в виде *оконных сообщений* (window message).

*Имя окна* (window name) - это текстовая строка, которая идентифицирует окно для пользователя. Главное окно, диалоговое окно или окно-сообщение обычно показывают это имя в заголовке окна. Элемент управления также может показывать имя окна, в зависимости от своего оконного класса. К примеру, кнопки, поля ввода и статический текст показывают имя окна в прямоугольнике элемента управления. С другой стороны, элементы вроде списков и списков выбора не показывают имена окон.Чтобы изменить имя окна после создания окна, используйте [функцию SetWindowText](http://msdn.microsoft.com/en-us/library/ms633546(VS.85).aspx). Используйте функции [GetWindowTextLength](http://msdn.microsoft.com/en-us/library/ms633521(VS.85).aspx) и [GetWindowText](http://msdn.microsoft.com/en-us/library/ms633520(VS.85).aspx) для получения имени окна.

**Стиль окна**

Каждое окно имеет один или несколько оконных стилей. *Оконный стиль* (window style) - это числовая константа, которая определяет аспект поведения или отображения, не указанный в оконном классе. Приложение обычно устанавливает стили во время создания окна. Но оно также может изменить некоторые стили после создания окна, используя [функцию SetWindowLong](http://msdn.microsoft.com/en-us/library/ms633591(VS.85).aspx).

Стили интерпретируются системой и, в некотором роде, оконной процедурой класса окна.

Некоторые оконные стили применимы к любым окнам, но большинство стилей специфичны для оконных классов. Общие стили представлены константами, которые начинаются с префикса WS\_; вы можете комбинировать стили, используя оператор or. Стили окна, специфичные для оконного класса, определяют поведение и отображение окон, принадлежащих этому классу. К примеру, класс SCROLLBAR создаёт элемент управления "полоса прокрутки". При этом стили [SBS\_HORZ](http://msdn.microsoft.com/en-us/library/bb787533(VS.85).aspx) и SBS\_VERT указывают создаётся ли вертикальная или горизонтальная полоса прокрутки.

Вот описание стилей для некоторых стандартных системных оконных классов:

* [Window Styles](http://msdn.microsoft.com/en-us/library/ms632600(VS.85).aspx)
* [Button Styles](http://msdn.microsoft.com/en-us/library/bb775951(VS.85).aspx)
* [Combo Box Styles](http://msdn.microsoft.com/en-us/library/bb775796(VS.85).aspx)
* [Edit Control Styles](http://msdn.microsoft.com/en-us/library/bb775464(VS.85).aspx)
* [List Box Styles](http://msdn.microsoft.com/en-us/library/bb775149(VS.85).aspx)
* [Rich Edit Control Styles](http://msdn.microsoft.com/en-us/library/bb774367(VS.85).aspx)
* [Scroll Bar Control Styles](http://msdn.microsoft.com/en-us/library/bb787533(VS.85).aspx)
* [Static Control Styles](http://msdn.microsoft.com/en-us/library/bb760773(VS.85).aspx)

**Расширенный стиль окна**

Каждое окно опционально может иметь один или несколько дополнительных (расширенных) оконных стилей. *Расширенный оконный стиль* (extended window style) - это числовая константа, которая определяет аспект окна, не указанный в классе окна или простом стиле окна. Приложение обычно устанавливает расширенные стили во время создания окна. Но оно также может изменить некоторые стили после создания окна, используя [функцию SetWindowLong](http://msdn.microsoft.com/en-us/library/ms633591(VS.85).aspx).

**Положение**

Положение окна определяется как координаты его левого-верхнего угла, измеряемые в пикселях экрана. Эти координаты (иногда называемые оконными координатами (window coordinates)) всегда относительны левого-верхнего угла экрана или (для child окон) - левого-верхнего угла клиентской области родительского окна. К примеру, окно верхнего уровня с координатами (10,10) размещается на 10 пикселей правее левого-верхнего угла экрана и на 10 пикселей ниже его. С другой стороны, child окно с координатами (10,10) размещается на 10 пикселей правее и на 10 пикселей ниже левого-верхнего угла клиентской области родительского окна.

[Функция WindowFromPoint](http://msdn.microsoft.com/en-us/library/ms633558(VS.85).aspx) ищет окно, занимающее указанную точку на экране, и возвращает его описатель. Аналогичным образом, [функция ChildWindowFromPoint](http://msdn.microsoft.com/en-us/library/ms632676(VS.85).aspx) и [функция ChildWindowFromPointEx](http://msdn.microsoft.com/en-us/library/ms632677(VS.85).aspx) возвращают описатель дочернего окна, занимающего указанную точку в родительском окне. Хотя ChildWindowFromPointEx может игнорировать невидимые, отключенные и прозрачные - ChildWindowFromPoint перечисляет их все.

**Размер**

Размер окна (высота и ширина) измеряется в пикселях. Окно может иметь нулевую высоту или нулевую ширину. Если приложение задаст нулевые высоту или ширину, то система установит размер в минимально допустимый. Чтобы определить минимально возможный размер окон, приложение может вызвать [функцию GetSystemMetrics](http://msdn.microsoft.com/en-us/library/ms724385(VS.85).aspx) с флагами SM\_CXMIN и SM\_CYMIN.

Приложению может потребоваться создать окно с клиентской областью определённого размера (обычно приложение задаёт размеры окна, а клиентская область получается по окну за вычетом размера рамки, заголовка и т.п.). [Функция AdjustWindowRect](http://msdn.microsoft.com/en-us/library/ms632665(VS.85).aspx) и [функция AdjustWindowRectEx](http://msdn.microsoft.com/en-us/library/ms632667(VS.85).aspx) могут быть использованы для вычисления нужного размера окна, основываясь на желаемом размере клиентской области окна. Приложение может передать вычисленные значения размера в [функцию CreateWindowEx](http://msdn.microsoft.com/en-us/library/ms632680(VS.85).aspx).

Приложение может изменить размер окна на чрезмерно большой; однако ему не следует изменять размер окна на размер, больший экрана. До изменения размеров окна, приложение может узнать размеры экрана, используя [функцию GetSystemMetrics](http://msdn.microsoft.com/en-us/library/ms724385(VS.85).aspx) с флагами SM\_CXSCREEN и SM\_CYSCREEN.

**Описатель родительского окна или окна-владельца**

Окно может иметь родительское окно. Окно, которое имеет родительское окно, называется *дочерним окном* (child window). *Родительское окно* (parent window) предоставляет своим дочерним окнам систему координат для относительного позиционирования. Если у окна есть окно-родитель, то это влияет на его отображение; к примеру, дочернее окно обрезается так, чтобы оно не выходило за пределы родительского окна и (опционально) за пределы клиентской области родительского окна. Окно, у которого нет родителя, либо же родительским окном является окно рабочего стола, называется *окном верхнего уровня* (top-level window). Приложение может использовать [функцию EnumWindows](http://msdn.microsoft.com/en-us/library/ms633497(VS.85).aspx), чтобы получить описатель каждого окна верхнего уровня в системе.

Кроме того, окно может владеть другим окном или иметь окно-владельца (owner window). Владетельное окно всегда появляется поверх своего владельца, оно также сворачивается, когда сворачивается его владелец, и уничтожается, когда уничтожается его владелец.

**Описатель меню или идентификатор дочернего окна**

Child окно может иметь *идентификатор дочернего окна* (child-window identifier) - это уникальное число, присваиваемое приложением. Идентификатор дочернего окна особенно полезен для приложений, создающих несколько дочерних окон. Когда приложение создаёт дочернее окно, оно может указать идентификатор для этого окна. После создания окна приложение может поменять идентификатор [функцией SetWindowLong](http://msdn.microsoft.com/en-us/library/ms633591(VS.85).aspx) или получить его [функцией GetWindowLong](http://msdn.microsoft.com/en-us/library/ms633584(VS.85).aspx).

Каждое окно, кроме child-окон, может иметь меню. Приложение может добавить меню в окно, указав описатель меню либо при регистрации оконного класса, либо непосредственно при создании окна.

**Описатель экземпляра**

Каждый исполняемый модуль (exe или DLL) имеет описатель, ассоциированный с ним. [Система использует этот идентификатор, чтобы отличать оконные классы, создаваемые разными модулями](http://www.transl-gunsmoker.ru/2010/04/hinstance-createwindow-registerclass.html).

**Пользовательские данные**

Каждое окно может иметь ассоциированные с ним пользовательские данные, не используемые системой. Когда создаётся окно, система передаёт указатель на пользовательские данные в оконную процедуру создаваемого окна. Оконная процедура может использовать эти данные для инициализации окна и (опционально) для сохранения данных с окном.

**Описатель окна**

После создания окна функция создания возвращает *оконный описатель* (window handle), который уникально идентифицирует окно в системе. Оконный описатель имеет тип HWND. Приложение может использовать этот описатель в других функциях работы с окнами.

**Окна в VCL**

В Delphi вся работа с окнами заключается в класс-оболочку [TWinControl](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl). Формы, кнопки, списки - всё это классы, наследуемые от TWinControl. К примеру: TForm1 -> TForm -> TCustomForm -> TScrollingWinControl -> TWinControl или TButton -> TCustomButton -> TButtonControl -> TWinControl.

У каждого объекта типа TWinControl есть описатель - [свойство Handle типа HWND](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.Handle). Этот описатель создаётся по запросу - при первом обращении к свойству Handle. Эту проверку выполняет метод HandleNeeded. Создание описателя выполняет [метод CreateHandle](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.CreateHandle), который является обёрткой к [методу CreateWnd](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.CreateWnd). Фактически, метод CreateHandle просто вызывает CreateWnd, выполняя пост-настройку объекта после создания окна (настройка якорей и ассоциация объекта TWinControl с окном).

Само создание окна выполняется так: сперва метод CreateWnd строит параметры окна, используя [метод CreateParams](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.CreateParams), который заполняет [запись типа TCreateParams](http://docwiki.embarcadero.com/VCL/en/Controls.TCreateParams). Эти параметры проверяются на допустимость, в них записывается оконная процедура и вызывается [метод CreateWindowHandle](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.CreateWindowHandle), который, собственно, и создаёт окно (и является простой обёрткой к функции CreateWindowEx).

Если вы заинтересованы в изменении создаваемого окна, то вы можете заместить метод CreateParams (если вам нужно просто поменять параметры создания окна), CreateWindowHandle (если вы хотите задавать параметры явно вручную), либо CreateWnd (если вы хотите сделать пост-обработку создания окна).

Оконной процедурой (точкой входа) в TWinControl является метод MainWndProc. Этот метод не виртуален, но всё, что он делает - вызывает настоящую оконную процедуру [WindowProc](http://docwiki.embarcadero.com/VCL/en/Controls.TControl.WindowProc), обернув вызов в try/except и вызывая [Application.HandleException](http://docwiki.embarcadero.com/VCL/en/Forms.TApplication.HandleException) при ошибке. Именно это - то самое место, в котором вы получаете сообщение об ошибке при исключении в обработчике сообщения (например, нажатия на кнопку) вместо вылета программы. Сама же оконная процедура WindowProc является и вовсе свойством, которое можно присвоить в любое значение (для установки своей оконной процедуры). По умолчанию, WindowProc указывает на (виртуальный) [метод WndProc](http://docwiki.embarcadero.com/VCL/en/Controls.TControl.WndProc) того же объекта. Как правило, если вы хотите дополнить или заменить оконную процедуру, то вы должны заметить метод WndProc в классе-наследнике (если вы хотите выполнить замещение в "вашем" классе), либо изменить свойство WindowProc (если вы хотите изменить поведение внешнего объекта).

Метод WndProc является конечной реализацией оконной процедуры. Собственно, это даже не "изобретение" TWinControl - этот метод появился у предка TWinControl: [TControl](http://docwiki.embarcadero.com/VCL/en/Controls.TControl). TControl является базовым классом для всех элементов управления, необязательно оконных. Оконные элементы управления наследуются от TWinControl.

Таким образом, цепочка вызовов при обработке сообщений выглядит так: MainWndProc -> WindowProc -> WndProc. WndProc может сама обработать сообщение, а может оставить его на обработку по умолчанию. В последнем случае сообщение дополнительно проходит такой путь: WndProc -> [Dispatch](http://docwiki.embarcadero.com/VCL/en/System.TObject.Dispatch) -> [DefaultHandler](http://docwiki.embarcadero.com/VCL/en/System.TObject.DefaultHandler) -> [DefWndProc](http://docwiki.embarcadero.com/VCL/en/Controls.TWinControl.DefWndProc). Метод Dispatch сначала пытается направить сообщение [message-методам](http://www.transl-gunsmoker.ru/2011/07/hack-9-dynamic-method-table-structure.html). И только при их отсутствии - вызывает DefaultHandler. DefWndProc вызывается только для оконных контролов при наличии описателя. Таким образом, если вы заинтересованы в необработанных сообщениях - то вам нужно использовать DefWndProc для окон и DefaultHandler для неоконных контролов.

Заметьте, что Dispatch, DefaultHandler и message-методы вводятся аж в [TObject](http://docwiki.embarcadero.com/VCL/en/System.TObject), что означает доступность диспетчеризации сообщений для **любого** объекта Delphi.

Почти все вышеупомянутые методы являются виртуальными - т.е. доступными для замещения и модификации в классах-наследниках, что позволяет нам менять стандартное поведение в широких пределах.

[TComponent](http://docwiki.embarcadero.com/VCL/en/Classes.TComponent) (который является предком TControl, а, следовательно и TWinControl) вводит понятие "владельца" компонента (Owner). Заметьте, что это не то же самое понятие, что окно-владелец в терминах системы, как обсуждается выше. Окно-владелец и окно-родитель (в терминах системы) указываются одинаковым образом - как поле WndParent в записи TCreateParams, параметр hWndParent в [функции CreateWindowEx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms632680(v=vs.85).aspx) и аналогичных местах. То, как трактуется это окно (как родитель или владелец) зависит от типа создаваемого окна (дочернее или верхнего уровня). Поведение этих окон описано выше.

Delphi же вводит понятие владельца в ином смысле. [Владелец (в терминах Delphi)](http://docwiki.embarcadero.com/VCL/en/Classes.TComponent.Owner) - это компонент, который отвечает за удаление владеемого компонента. Его можно указать при создании компонента, либо сменить/задать уже после создания (что бывает редко). При удалении компонента-владельца удаляются и все компоненты, которыми владеет владелец. Вы можете не указывать владельца, если хотите удалять компонент сами, вручную. Как правило, форма является владельцем всех компонентов, размещённых на ней.

Вам не следует путать владельца в терминах системы и владельца в терминах Delphi. Владелец в терминах системы отвечает за взаимодействие окон. Владелец в терминах Delphi не связан с окнами, он контролирует время жизни объекта. Фактически, он применим также и к неоконным элементам управления и даже невизуальным компонентам.

**Примечание:** Delphi также позволяет вам изменить родительское окно уже после создания объекта. Это позволяет вам сделать "перескакивание" дочернего контрола с одного контейнера (формы, панели) на другой.

**Окно Application**

Следующий тонкий момент связан со специальным окном Application в Delphi. Как вам должно быть известно, в Delphi есть глобальный объект [Application](http://docwiki.embarcadero.com/VCL/en/Forms.TApplication), представляющий собой "приложение". В нём содержатся глобальные параметры и некоторые служебные методы-помощники. Сейчас нас интересует тот факт, что в Application есть окно. Это - служебное окно. Его описатель доступен через [свойство Application.Handle](http://docwiki.embarcadero.com/VCL/en/Forms.TApplication.Handle). Это окно является обычным окном. Оно видимо, находится на экране (в центре), но... имеет нулевой размер! Так что реально вы его увидеть не можете, не можете и щёлкнуть по нему.

В Delphi принята несколько иная идеология. Кнопка в панели задач представляет собой "приложение", а не отдельное окно. У неё свой заголовок, своя иконка, она сворачивает все окна приложения - и так далее. Поскольку в системе окна работают не так, то Delphi эмулирует такое поведение. Кнопка в панели задач - это, на самом деле, окно Application, а реальные окна программы там не появляются. Все окна приложения имеют владельца - окно Application. Сворачивание приложения на самом деле заключается в скрытии окон (реально сворачивается только окно Application). При этом, конечно же, куча сил уходит на синхронизацию различных состояний между участниками подобной системы. Отсюда идёт множество вопросов вроде "как мне свернуть/развернуть приложение в трей", вопросов про правильную связку окон и им подобных.

**Модальные окна**

Ещё одним моментом, где Delphi эмулирует нетипичное поведение в системе, являются модальные окна. В дизайне системы модальность предназначена для диалоговых окон, которые создаются специальными функциями по шаблону из ресурса. Причём модальность диалога указывается специальным флагом - DS\_MODALFRAME. Иными словами, это означает, что одно и то же окно не может показываться в разных режимах - модальном и не модальном.

Разумеется, такая модель не подходит для Delphi, где вы можете показывать окно различными способами - вызывая [Show](http://docwiki.embarcadero.com/VCL/en/Forms.TCustomForm.Show) или [ShowModal](http://docwiki.embarcadero.com/VCL/en/Forms.TCustomForm.ShowModal). Поэтому Delphi не использует системный механизм модальных окон. Вместо этого она эмулирует поведение модальных окон. Когда вы показываете "модальное" окно, VCL отключает все прочие окна в программе, а после окончания показа окна - восстанавливает их доступность. Отключение окна блокирует взаимодействие с пользователем - вы не можете переключиться в окно и вводить в него информацию. Как правило, отключение окна приводит к изменению его вида: окно-контрол показывается "затенённым". Однако окна-формы не изменяют свой вид. Поэтому отключенное окно визуально не отличается от обычного окна. Так что у вас создаётся впечатление, что окно не отключено, а просто сейчас показывается другое (модальное) окно.

# **Тема.** Организация взаимодействия форм. Создание многостраничных окон.

**Задание:** Составить конспект.

**План работы:**

1 Ознакомиться с перечнем вопросов, подлежащих рассмотрению

2 Ознакомиться с представленным теоретическим материалам

3 Ответить на вопросы для самопроверки

4 Законспектировать ответы на вопросы, подлежащие рассмотрению, привести примеры

Включайте в конспект не только основные положения, но и обосновывающие их выводы, конкретные факты и примеры (без подробного описания).

5.Составляя конспект, записывайте отдельные слова сокращённо, выписывайте только ключевые слова, делайте ссылки на страницы конспектируемой литературы, применяйте условные обозначения.

6.Чтобы форма конспекта отражала его содержание, располагайте абзацы «ступеньками», подобно пунктам и подпунктам плана, применяйте разнообразные способы подчеркивания, используйте карандаши и ручки разного цвета.

**Вопросы для самостоятельной работы**

1 Добавление новой формы

2 Подключение новой формы

**Вопросы для самоконтроля:**

1Какой командой создается новое окно

2 Как подключить модуль нового окна к главному окну

**Форма контроля:** Оценка составленного конспекта.

**Теоретический материал**

В приложении сложно обойтись одной главной формой. Поэтому приходиться добавлять другие формы и взаимодействовать с ними.

[новая форма](http://delphi-prg.ru/wp-content/uploads/2012/01/newform1.jpg)

Для того чтобы добавить новую форму в проект заходим в пункт меню File -> New -> Form. Или нажимаем на значок создания новой формы в панели инструментов.

Добавив новую форму, она будет выглядеть также как и основная. При создании форма получит имя Form2 (число обозначает номер создаваемой формы) и связанный модуль формы Unit2. В новую форму можно добавлять компоненты также как и в главную. Но для чтобы дополнительная форма отобразилась в программе нам нужно будет создавать обработчик события.

Мы можем отображать формы в Delphi двумя способами:

* Модально - пользователь имеет возможность работать только в одной форме и прежде чем перейти к другой, необходимо закрыть модальную форму.
* Немодально - пользователь может одновременно работать с несколькими формами.

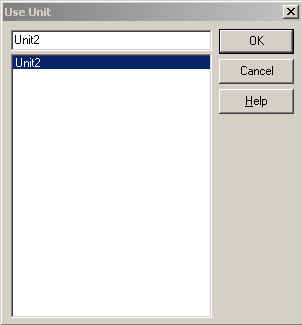
Для того чтобы отобразить форму модально нужно записать такой код:

 Form2.ShowModal;

для не модальной:

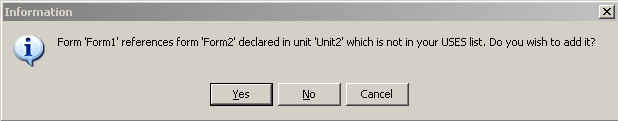
 Form2.Show;

При ссылке на другую форму необходимо помнить о взаимосвязи между формами и модулями. Любая форма имеет свой модуль, при создании формы Delphi автоматически создает код модуля и в процессе работы добавляются разнообразные участки кода или пользователем или автоматически средой Delphi.

[](http://delphi-prg.ru/wp-content/uploads/2012/01/formadobavl2.jpg)

Поэтому модуль второй формы должен быть включен с помощью зарезервированного слова **uses** текущего модуля. Проще это сделать выбрав команду File -> Use Unit (файл использовать модуль). Выбрав нужный модуль ссылка на него автоматически добавиться.

Но если мы  забудем это сделать то при компиляции программы появиться    диалоговое окно. В нем будет сказано, что первая форма использует вторую, но модуль второй формы отсутствует в списке uses модуля первой формы. Нам достаточно нажать на кнопку Yes и необходимая ссылка будет добавлена.

[](http://delphi-prg.ru/wp-content/uploads/2012/01/formadobavl.jpg)

Можно также производить изменения с отдельными компонентами и свойствами другой формы.

Для закрепления материала создадим вторую форму. Присоединим модуль второй формы к первой, описанным выше способом. В первой форме расположим простую кнопку и компонент Edit. В обработчике кнопки запишем следующий код:

Form2.Caption := Edit1.Text;

Form2.ShowModal;

Запустив программу и нажав на кнопку, мы откроем вторую форму модально, заголовок у нее будет взят из свойства Text компонента Edit первой формы.

1. Модель – некоторое физическое или логическое (искусственное или естественное) образование (предмет или явление), которое обладает свойствами и поведением, подобными (сходственными) свойствам и поведению другого образования, называемого оригиналом (аналогом). [↑](#footnote-ref-1)
2. Согласно принятому в DELPHI соглашению имена классов обычно (но необязательно!) начинаются с префикса **T**, а имена полей – с префикса **F** (или **f**). Смысл ключевых слов **private, public** поясняется ниже. [↑](#footnote-ref-2)
3. Всю объектную базу DELPHI можно просмотреть в окне BrowseObjectsкомандой системы программирования **View|Browser**после компиляции проекта. [↑](#footnote-ref-3)