МИЛОШ Е., ДЖОРУПБЕКОВ С.

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Учебное пособие

Рекомендовано Учебно-методическим советом
Кыргызского национального университета
имени Ж. Баласагына

Бишкек – 2009

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Рецензенты:

Директор Института новых информационных технологий
КГУСТА им. Н. Исанова,
академик Международной академии информатизации Укуев Б. Т.

Зав. кафедрой информационных систем Института информационных и коммуникационных технологий Кыргызского национального университета им. Ж. Баласагына, профессор Сабитов Б.Р.

Милош Е., Джорупбеков С.

Парадигмы программирования.
Учебное пособие. Е. Милош, С. Джорупбеков – Б.: 2009. - 209 с.

Учебное пособие подготовлено по программе международного проекта для магистров с двойной компетенцией: «Информатика и социальные науки».

Парадигмы программирования – это способы мышления программиста при программировании задач и способы внутренней организации текста программы. Почти все основные парадигмы программирования нашли своего отражения в современном языке программирования высокого уровня Delphi, как результат усовершенствования языка Pascal.

В учебном пособии подробно излагаются конструкции языка программирования Delphi и их использования, а также показываются способы внутренней организации текста Delphi-программы при использовании различных парадигм программирования.

Рекомендовано к печати Учебно-методическим советом Кыргызского национального университета им. Ж. Баласагына

СОДЕРЖАНИЕ

Введение

Использование компьютеров в деятельности человека и практика программирования различных прикладных задач выработали разные парадигмы программирования.

Термин "парадигма" вначале определялся, как свод норм научного мышления, как правила развития научного знания. Они в течение определенного времени дают научному сообществу модель постановки проблем и их решений. В каком же смысле этот термин понимается в программировании?

Вся история программирования – это попытка совладеть со сложностью решаемой задачи, т.е. со сложностью окружающего мира. Задачи, встающие перед программистом, становятся все более громоздкими, системными, информация, которую необходимо обработать, непрерывно растет. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи для решения. Программисты придумывают новые методы программирования, стили-образцы, создают новые языки. Некоторые методы и стили становятся общепринятыми, образцами и образуют на некоторое время *способ мышления* и *программирования*, или, говорят, образуют так называемую *парадигму программирования*. В итоге все это приводит к определенной схеме внутренней организации текста программы. Так что для *каждой парадигме программирования присуще своя характерная схема внутренней организации текста программы*. Итак, **парадигма программирования – это способ мышления при программировании задач и способ внутренней организации текста программы.**

Все эти способы нашли своего отражения в современном языке программирования высокого уровня Delphi, как результат усовершенствования языка Pascal. А также практически все современные технологии программирования уже вошли в Delphi в той или иной форме. Считается, что система программирования Delphi является одной из самых мощных сред *быстрого проектирования приложений*. В таких средах в результате автоматизации многих процессов подготовки современных приложений (прикладных программ), работы программирования заменяются работами проектирования, что позволяет разработчику больше сосредоточиться на логике решаемой задачи. Поэтому в учебном пособии подробно излагаются конструкции языка программирования Delphi и их использования, а также показываются способы внутренней организации текста Delphi-программы при использовании различных парадигм программирования.

Авторы

	Ельжбета Милош – доктор инженерии программирования, Люблинский политехнический университет (Польша)
	Сатылган Джорупбеков – доцент Кыргызского национального университета им. Ж. Баласагына

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

«Парадигмы программирования»

План лекций и лабораторных занятий

Лекции – 32ч.
Лабораторные занятия – 64ч.

№	Тема	Содержание	Лек-ции	Лаб. зан.
1	Компьютер, языки и парадигмы программирования	Введение Что такое компьютер? Устройство компьютера Языки программирования Парадигмы программирования Системы программирования Система программирования Delphi.	2 Lec1	2 Lab1
2	Язык программирования Delphi. Структура программы.	Алфавит языка Delphi Лексемы Структура программы. Процедуры вывода Процедуры ввода Компиляция и запуск программы на выполнение	2 Lec2	2 Lab2
3	Информация, данные и их типы. Процедуры ввода, вывода.	Информация и данные Именование, объявление и использование переменных Типы данных. Стандартные типы данных Процедуры ввода, вывода.	2 Lec3	2 Lab3
4	Оператор присваивания Выражения.	Концепция действия. Выражения Арифметический оператор присваивания Логический оператор присваивания Символьный оператор присваивания Строковый оператор присваивания Приведение типов и функции преобразования типов	2 Lec4	4 Lab4 Lab5
5	Управление последователь-ностью действий.	Алгоритм с ветвлениями Составной оператор begin-end Условный оператор if-then Условный оператор if-then-else Вложенные условные операторы Оператор выбора case Оператор перехода	2 Lec5	4 Lab6 Lab7
6	Цикл. Структура цикла и операторы цикла	Цикл. Структура цикла. Оператор цикла for Оператор цикла while Оператор цикла repeat Управление работой циклов Вложенные циклы	2 Lec6	6 Lab8 Lab9 Lab10
7	Парадигма процедурно-структурного	Подпрограмма Объявление и вызов процедур и функций Значение, возвращаемое функцией	2 Lec7	4 Lab11 Lab12

	программирования Подпрограммы: Процедуры и функции – часть 1	Параметры подпрограмм Глобальные и локальные переменные Завершение работы подпрограммы Внутренняя организация Delphi-программы, использующей подпрограммы Парадигма процедурно-структурного программ-мирования.		
	Парадигма процедурно-структурного программирования Подпрограммы: Процедуры и функции – часть 2	Вложенные подпрограммы Отложенная реализация подпрограмм Перезагрузка процедур и функций Внешние определения под-программ Рекурсия Встраиваемые подпрограммы	2 Lec8	2 Lab13
9	Пользовательские типы	Пользовательские типы Перечислимые типы Типы поддиапазонов	2 Lec9	2+2 K1 Lab14 Lab15
10	Массивы.	Статические массивы Операции, допустимые над массивами Динамические массивы Передача массивов подпрограммам	2 Lec10	4 Lab16 Lab17
11	Записи. Множества.	Определение записи Операции, допустимые над записями Определение множества Операции, допустимые над множествами	2 Lec11	4 Lab18 Lab19
12	Файлы.	Файлы текстовые Бинарные файлы Файлы типизированные Файлы нетипизированные	2 Lec12	6 Lab20 Lab21 Lab22
13	Парадигма модульного программирования Модули	Недостатки парадигмы процедурно-ориентированного программирования Понятие модуля. Запись модуля Главная программа модульной Delphi-программы Парадигма модульного программирования в среде Delphi	2 Lec13	2+2 K2 Lab23 Lab24
14	Парадигма объектного программирования Классы и объекты.	Недостатки парадигмы модульного программирования Парадигма объектно-ориентированного программирования Понятие класса и объекта Инкапсуляция, наследование, полиморфизм.	2 Lec14	6 Lab25 Lab26 Lab27
15	Парадигма событийного и визуального программирования.	Событийное программирование Визуальное компонентное программирование в среде Delphi Среда разработки Delphi	2 Lec15	4 Lab28 Lab29

	Программы управляемые событиями. Компонентное программирование.	Проект Delphi Компоненты Form, Label, Edit, Button и их использования в визуальном программировании Пример вычислительной программы		
16	Библиотека компонентов VCL Delphi	О библиотеке компонентов VCL Delphi Компоненты карты Standard Пример использования компоненты TMemo в визуальном программировании Заключение – сравнение парадигм программирования	2 Lec16	4+2 К3 Lab30 Lab31 Lab32

Лабораторные занятия

№	Тема	Программы
Lab1	Парадигмы программирования	*Lab1a.exe, Lab1b.exe, Lab1c.exe, Lab1d.exe* *Lab1a.dpr* *Lab1b.dpr* *Lab1c.dpr* и *Unit1c.pas* *Lab1d.dpr* и *Unit1d.pas* *Lab1e.dpr* и *Unit1e.pas*
Lab2	Язык программирования Delphi. Структура программы.	*Lab2_1.dpr*
Lab3	Данные программы (переменные, константы) и их типы. Оператор ввода, вывода	*Lab3_1.dpr*
Lab4	Оператор присваивания. Выражения	*Lab4_1.dpr*
Lab5	Оператор присваивания. Выражения	*Lab5_1.dpr*
Lab6	Условный оператор	*Lab6_1.dpr*
Lab7	Оператор выбора	*Lab7_1.dpr*
Lab8	Цикл. Структура цикла и операторы цикла	*Lab8_1.dpr*
Lab9	Цикл. Структура цикла и операторы цикла	*Lab9_1.dpr*
Lab10	Цикл. Структура цикла и операторы цикла	*Lab10_1.dpr*
Lab11	Парадигма процедурно-структурного программирования Подпрограммы: Процедуры	*Lab11_1.dpr*
Lab12	Парадигма процедурно-структурного программирования Подпрограммы: Функции	*Lab1b.dpr*
Lab13	Парадигма процедурно-структурного программирования Подпрограммы: Процедуры и функции	*Lab13_1.dpr*
Lab14	Пользовательские типы	*Lab14_1.dpr*
Lab15	Контрольная 1	
Lab16	Статические массивы	*Lab16_1.dpr*
Lab17	Динамические массивы	*Lab17_1.dpr*
Lab18	Записи	*Lab18_1.dpr*
Lab19	Множества	*Lab19_1.dpr*
Lab20	Файлы текстовые	*Lab20_1.dpr*
Lab21	Файлы типизированные	*Lab21_1.dpr*

Lab22	Файлы	*Lab22_1.dpr*
Lab23	Парадигма модульного программирование. Модули.	*Lab23_1.dpr*
Lab24	Контрольная 2	
Lab25	Парадигма объектно-ориентированного программирования. Классы и объекты.	*Lab25_1.dpr*
Lab26	Парадигма объектно-ориентированного программирования. Классы и объекты.	*Lab26_1.dpr*
Lab27	Парадигма объектно-ориентированного программирования. Классы и объекты.	*Lab27_1.dpr*
Lab28	Парадигма событийного и визуального программирования. Программы, управляемые событиями. Компонентное программирование.	*Lab28_1.dpr*
Lab29	Парадигма событийного и визуального программирования. Программы, управляемые событиями. Компонентное программирование.	*Lab29_1.dpr*
Lab30	Библиотека компонентов VCL Delphi	*Lab30_1.dpr*
Lab31	Контрольная 3	
Lab32	Зачет лабораторных	

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 1. Компьютер, языки и парадигмы программирования

План
Введение
Информация о компьютере
Устройство компьютера
Языки программирования
Парадигмы программирования
Системы программирования
Интегрированная среда быстрой разработки программ - Delphi.

Введение

Наша изучаемая дисциплина называется “Парадигмы программирования”. В этом курсе изучается что такое язык программирования, основные конструкции языка программирования **Delphi (Дельфи)** и как они используются при написании программы. Этот язык рожден от языка Паскаль, ранее назывался Object Pascal (Объектный Паскаль). Синтаксис языка более прост в восприятии, чем другие языки программирования, например, синтаксис языка С.

Мы также познакомимся с различными парадигмами программирования и их влияниями на внутренней организации текста программы на примере Delphi-программы. Специалист по информационной технологии должен быть знаком с существующими методами разработки программ и уметь использовать их на практике.

Сначала освежим память информацией о компьютере.

Информация о компьютере

Компьютер — это устройство, работающее по программе и способное производить вычисления и принимать логические решения в миллионы или даже миллиарды раз быстрее человека. *Программа* есть *алгоритм* решения задачи или алгоритм выполнения какой-то работы, написанный на языке «понятной» устройству. Сейчас многие из современных персональных компьютеров могут выполнять десятки миллионов (и даже миллиарды) *операций* сложения в секунду. Чтобы повторить этот объем вычислений, сделанных компьютером за одну секунду, человеку, вооруженному калькулятором, потребуется целая жизнь. (Информация к размышлению: как узнать, не сделал ли человек ошибки в ходе вычислений? Как узнать, не ошибся ли компьютер в своих вычислениях?) Сегодняшние самые быстрые *суперкомпьютеры* могут выполнять сотни миллиардов операций сложения в секунду — это примерно столько же, сколько сотни тысяч людей могут выполнить за год. А в исследовательских лабораториях уже созданы компьютеры, выполняющие триллионы операций в секунду!

Компьютеры обрабатывают данные под управлением наборов команд, называемых *компьютерными программами.* Эти компьютерные программы направляют действия компьютера посредством упорядоченных наборов команд, написанных людьми, которых называют *компьютерными программистами.* Меняя выполняемые компьютером программы можно изменять вид выполняемых работ компьютером.

Разнообразные устройства (такие как клавиатура, монитор, диски, память и процессорные блоки), входящие а состав компьютерной системы, называются *аппаратным обеспечением.* Компьютерные программы, исполняемые компьютером, называются *программным обеспечением.* Стоимость аппаратных средств в последние годы существенно снизилась и достигла уровня, когда персональные компьютеры превратились в предмет массового потребления. К сожалению, стоимость разработки программного обеспечения неуклонно росла, так как

программисты создавали все более мощные и сложные прикладные программы, не имея средств улучшения технологии их разработки.

Устройство компьютера

Независимо от особенностей физической реализации каждый компьютер можно разделить на шесть *логических блоков* или частей:

1. Блок *ввода.* Это часть компьютера, принимающая информацию в виде данных и программ от различных *устройств ввода* и передающая ее в распоряжение других блоков компьютера для последующей обработки. В наши дни информация в компьютер вводится обычно при помощи клавиатуры (устройство, напоминающее клавиатуру пишущей машинки), манипулятора мыши и дисков. В будущем информация будет вводиться в компьютер голосом, в виде отсканированных образов и с помощью видеозаписи.

2. Блок *вывода.* Здесь информация, обработанная компьютером, передается на различные *устройства вывода* и становится доступной для использования за пределами компьютера. Обычно данные выводятся на экран монитора, печатаются на бумаге, воспроизводятся через звуковые колонки, записываются на магнитные диски и ленты или передаются для дальнейшей обработки другим устройствам.

3. *Блок памяти.* Это хранилище информации относительно малой емкости, но с высокой скоростью доступа. Сюда поступает информация, введенная через блок ввода, которая может быть немедленно обработана, когда это потребуется. Сюда также поступают обработанные компьютером данные для последующей передачи их блоку вывода, который отправит их на нужное устройство вывода. Обычно блок памяти называют просто *памятью, основной памятью или RAM (Random Access Memory — память с прямым доступом).*

4. *Арифметико-логическое устройство (АЛУ).* Эта часть компьютера выполняет «производственную» задачу; производит арифметические операции сложения, вычитания, деления и умножения. Логическая часть этого блока умеет принимать решения и отвечает за выполнение логических операций, например, сравнения двух элементов в памяти компьютера — совпадают они или нет.

5. *Центральное процессорное устройства (ЦПУ).* Это «административная» часть компьютера, которая координирует работу всех остальных блоков компьютера и осуществляет надзор за их деятельностью. ЦПУ дает команды блоку ввода на ввод данных в память, дает команды АЛУ выполнить вычисления с данными в памяти и указывает блоку вывода, когда данные из памяти должны быть направлены на нужное устройство вывода.

6. *Блок внешней памяти.* Это долговременное, большой емкости хранилище данных компьютера. В этом блоке обычно используются диски. Сюда помещаются на хранение данные и программы, не используемые другими блоками компьютера, возможно, на несколько часов, дней, месяцев или лет, словом, до тех пор, пока они не понадобятся снова. Время доступа к информации на устройствах внешней памяти больше, чем время доступа к основной памяти, а стоимость за единицу элемента памяти у внешней памяти меньше.

Языки программирования

Программисты пишут программы на различных языках программирования; некоторые из них непосредственно понятны компьютеру, программы, написанные на других языках, требуют

выполнения специальных промежуточных действий — *трансляции.* Сегодня используются сотни различных языков программирования. Все они могут быть разделены на три основные типа;

1. Машинные языки
2. Ассемблерные языки
3. Языки высокого уровня

Любой компьютер может понимать непосредственно только свой собственный *машинный язык*. Машинный язык определяется совокупностью операций, выполняемым конкретным компьютером. Он определяется основном аппаратным обеспечением компьютера. Например, процессор фирмы “Intel” может лишь выполнять операции арифметики и, осуществляя сравнения двух величин, может выполнять команды перехода на команды в заданных адресах памяти. Программировать такой компьютер можно в виде прямой записи двоичных команд.

Машинный язык, в общем случае, это последовательность двоичных чисел в двоичном представлении — последовательность нулей и единиц. Каждая такая последовательность — это команда компьютеру на выполнение элементарной операции в единицу времени. Машинные языки являются *машинно-зависимыми,* т.е. эти языки понятны только машинам одного и того же типа. Машинные языки тяжелы для человеческого восприятия. Это видно из приводимого ниже фрагмента программы на машинном языке, в котором выполняется *сложение* базовой ставки заработной платы и платы за сверхурочную работу и *запись* в память полученной итоговой зарплаты (команды даны в не двоичных числах):

+1300042774
+1400503410
+1200274027

Персональный компьютер имеет специальное постоянное запоминающее устройство с программами BIOS. После установки BIOS получается машина с дополнительными командами загрузки программы с дисков, чтения информации из любого сектора дисков, чтения символа с клавиатуры, вывода информации на экран и т.д. Благодаря прерываниям BIOS, становится возможным использование арифметики с плавающей точкой как при наличии, так и отсутствии сопроцессора. Так что, можно сказать, что машинный язык – это совокупность операций процессора и совокупность операций, реализуемых программами BIOS.

По мере развития компьютерной индустрии стало ясно, что программирование непосредственно на машинных языках — дело слишком медленное и утомительное для большинства программистов. Вместо последовательности чисел, которые компьютер может понимать напрямую, программисты для обозначения элементарных операций компьютера стали использовать их англоязычные аббревиатуры.

Эти сокращения и послужили основой для создания *языков ассемблера.* Для преобразования программ на ассемблере в машинные коды были созданы специальные *программы-трансляторы,* названные *ассемблерами,* выполняющие такое преобразование со скоростью, равной быстродействию компьютера. В приводимом ниже фрагменте программы на языке ассемблера выполняется та же задача: базовая ставка заработной платы складывается с платой за сверхурочную работу, а полученная итоговая зарплата записывается в память. Программа при этом становится более понятной, чем ее эквивалент на машинном языке;

```
LOAD  ocnovnajZarplata
ADD   cverxUrochnaj
STORE itogovajZarplata
```

Понятная человеку, эта программа теперь недоступна пониманию компьютера: до тех пор, пока она не будет переведена на машинный язык.

Появление языков ассемблера стимулировало расширение сферы использования компьютеров, несмотря на то, что при программировании простейших задач требовалось закодировать большое количество инструкций. Для сокращения количества инструкций и ускорения процесса разработки программ были придуманы языки программирования *высокого уровня,* один оператор (одно предложение) которых может выполнять целый ряд сложных действий.

Термин «*высокоуровневый*» означает следующее: многие детали обрабатываются автоматически, а программисту для создания своего приложения приходится писать меньшее количество строк. В частности:

- Распределением регистров занимается компилятор, так что программисту не надо писать код, обеспечивающий перемещение данных между регистрами и памятью;
- Последовательности вызова процедур генерируются автоматически; программисту нет необходимости описывать помещение аргументов функции в стек и их извлечение оттуда;
- Для описания структур управления программист может использовать также ключевые слова, как if, while; последовательности машинных команд, соответствующие этим описаниям компилятор генерирует динамически;
- Использует более строгий контроль типов.

Программы-переводчики программ на языке высокого уровня в программы на машинном языке получили название *компиляторов.* Программы на языке высокого уровня содержат операторы, напоминающие конструкции разговорного английского и традиционные математические обозначения. Фрагмент нашей программы начисления заработной платы на языке высокого уровня может уместиться в одну строку и будет выглядеть так:

itogovajZarplata := ocnovnajZarplata + cverxUrochnaj

Очевидно, что, с точки зрения программистов, языки высокого уровня предпочтительнее машинных и ассемблерных языков. Языки C, C++, Visual Basic, Delphi и Java сейчас самые популярные и самые мощные языки высокого уровня.

Процесс компиляции программы на языке высокого уровня в программу на машинном языке может занять значительное время. Поэтому были созданы программы - *интерпретаторы,* с помощью которых программы на языках высокого уровня выполнялись напрямую, без трансляции всей программы целиком в программу на машинном языке. Хотя откомпилированные программы выполняются намного быстрее, чем программы в режиме интерпретации, интерпретаторы нашли себе применение у разработчиков программ: в этом случае не нужно заниматься постоянной перекомпиляцией программы в процессе внесения изменений и исправления ошибок. После того как разработка программы завершена, ее можно откомпилировать и для повышения эффективности работы постоянно работать с откомпилированной версией.

Девяностые годы прошлого века ознаменовались рождением языков 4-го поколения - 4GL (fourth generation languages). В них впервые вместо скучных строк кода программист получил удивительную возможность оперировать графическими, интуитивно понятными образами, а для создания элементарного приложения стало достаточно лишь несколько раз щелкнуть кнопкой мыши. В одно время даже раздавались восторженные возгласы о том, что программирование стало доступным для домохозяек.

Создание профессионального программного продукта и в наши дни по-прежнему требует от человека *глубоких* и *разносторонних знаний*, *терпения* и *внимательности*, *находчивости* и *сообразительности* и, если не таланта, то, по крайней мере, *творческой одаренности*, потому что программирование уже давно перестало быть просто *наукой* - это уже и *искусство*.

Парадигмы программирования

Термин "парадигма" вначале определялся, как свод норм научного мышления, как правило развития научного знания. Оно в течение определенного времени дает научному сообществу

модель постановки проблем и их решений. В каком же смысле этот термин понимается в программировании?

Вся история программирования – это попытка совладеть со сложностью окружающего мира. Задачи, встающие перед программистом, становятся все более громоздкими, информация, которую необходимо обработать, непрерывно растет. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи для решения. Программисты придумывают новые методы программирования, стили-образцы, создают новые языки. Некоторые методы и стили становятся общепринятыми, образцами и образуют на некоторое время *способ мышления* и *программирования*, или, говорят, образуют так называемую *парадигму программирования*. В итоге все это приводит к определенной схеме внутренней организации текста программы. Так что для *каждой парадигме программирования присуще своя характерная схема внутренней организации текста программы*. Итак, **парадигма программирования – это способ мышления в программировании и способ внутренней организации текста программы.**

Можно сказать, практика создания различных программ выработала следующие парадигмы программирования:

1. *Структурное линейное программирование*. При структурном линейном программировании запись текста программы делается линейно (оператор за оператором) с помощью только *трех базовых структур (базовых предложений) алгоритма*: *действие (функция)*, *выбор* и *повторение*. Для моделирования логики алгоритма задачи эти предложения могут комбинироваться только двумя способами: *суперпозицией* и *вложением*. Использование только таких предложений порождают программы, которые проще для понимания.
2. *Процедурно-ориентированное программирование* (точнее процедурно-структурное, кратко процедурное программирование). В этом случае в добавок к структурированной форме записи текста программы, текст программы сложной задачи разбивается еще на *подпрограммы* (*процедуры* и *функции*). Процедурное программирование как методика разработки программного обеспечения разделяет программу на *данные* и *процедуры*, обрабатывающие эти данные. Процедурное программирование имеет последовательную природу. В процессе выполнения процедурной программы последовательно выполняются операторы текста программы, в частности, последовательно вызываются различные процедуры. После выполнения последней из них программа завершает свою работу.
3. *Модульное программирование* (точнее модульно-структурное программирование). Модульное программирование связывает константы, типы данных и процедуры в компоненты программы, называемые модулями. Модули могут быть использованы для разделения большой программы на логически связанные части. В одном модуле могут объединяться несколько подпрограмм. Модули скрывают представление данных и внутренние методы их. Модули относительно самостоятельны, они оформляются по другому чем подпрограммы и могут быть использованы во многих программах. Одного модуля выполнить как программу нельзя. Чтобы использовать содержащие в нем процедуры и функции нужно в программе заложить декларацию данного модуля. Модули могут вызывать друг друга, вводить из внешнего мира дополнительные данные и выводить результат. Такой подход дает возможность разрабатывать структуру программы на концептуальном, поведенческом уровне, а не в терминах данных и процедур, как в процедурном подходе. Однако потребности программирования, поставленного на промышленную основу, невозможно в полной мере удовлетворить с помощью модульного программирования. Возникает необходимость создать такую концепцию программирования, которая повысила бы производительность и надежность работы программистов, позволив применять конвейерные методы работы при разработке сложных программ. Так возникла парадигма объектно-ориентированного программирования.

4. *Объектно-ориентированное программирование.* Основополагающей идеей объектно-ориентрованного подхода является жесткое объединение данных и действий, производимых над этими данными, в единое целое, которое называется классом. На основе класса в программе строятся *объекты*. Такие программные объекты уже более адекватно отображают объекты реального мира, в то время как ни отдельно взятые данные, ни отдельно взятые подпрограммы не способны так адекватно их отображать. Кроме этого в основу объектно-ориентированного программирования положено *принципы инкапсуляции, сокрытия информации, абстракции данных, наследования* и *полиморфизма.* Это метод разработки программного обеспечения, позволяющий при моделировании программы использовать названия объектов реального мира. Такое программирование разбивает программу на набор взаимодействующих объектов.
5. *Событийное программирование.* Программа, построенная таким подходом, является совокупностью действий в виде процедур и функций, обслуживающих внешние события (например нажатие мыши, клавиатуры, действие часов). При таком подходе нельзя предусмотреть очередность выполнения подпрограммы, как делается в процедурном подходе, каждый запуск программы может выглядеть по другому. Исполнение программы это ожидание происхождения события и ответ на них. Большую роль при событийном программированию играет обслуживание ошибок со стороны пользователя, которого действие не всегда совпадает с концепцией автора программы.
6. *Визуальное программирование*. Визуальное программирование состоит в автоматизированной разработке программ с использованием особой диалоговой оболочки. Системы визуального программирования базируются на объектно-ориентированном программировании и являются его логическим продолжением.

В литературе выделяют еще другие виды парадигм программирования такие, как: логически-ориентированное программирование, параллельное программирование, программирование, ориентированное на правила и др., которые ориентированы для программирования определенного класса задач. Мы же рассматриваем парадигмы программирования, ориентированные для решения широкого класса задач.

В этом курсе мы будем подробно знакомиться *парадигмой процедурно-структурного программирования.* По этой парадигме программа внутренне организовывается с использованием процедур (или функций). Тексты вызываемых подпрограмм приводятся в описательной части *основой (вызывающей)* Delphi-программы**.** **Т**ексты *вызывающей программы* и *вызываемых процедур и функций* оформляются по правилам структурного программирования (*структурированного кодирования*), А также познакомимся с *парадигмой модульного программирования* и процессом создания Delphi-программы, состоящей из отдельных модулей. При этом еще познакомимся с начальными понятиями *объектно-ориентированного программирования,* а также принципами парадигмы *визуального компонентного программирования и событийного программирования.*

Системы программирования

Средство создания программы на языке высокого уровни с помощью компьютера принято называть *системой программирования.* Система программирования состоит из следующих компонентов.

1. Программа **Текстовый редактор**, с помощью которого составляется текст программы на компьютере. В результате получается исходный код программы, сохраняемый на компьютере в виде файла. Имя файла исходного кода на языке Delphi имеет расширение **.dpr** (его часть, играющая роль главной подпрограммы) и **.pas** (модули - его отдельные части).

2. Поскольку такой исходный код (например, *.pas-файл*) надо переводить на машинный язык (на машинный код), постольку необходим компонента, называемая **компилятором**. Компилятор порождает текст, называемый объектным кодом, сохраняемый в файлах с расширениями **.obj**. Исходный текст большой программы состоит, как правило, из нескольких текстов, оформленных в специальном виде, называемом *модулями*. Каждый модуль компилируется в отдельный файл с объектным кодом, которые затем надо объединять в одно целое. Кроме того, к ним приходиться добавлять машинный код подпрограмм, реализующих вычисление различных стандартных функций (например, вычисляющие значения математических функций). Такие функции принято содержат в библиотеках (файлах с расширениями **.lib)**
3. **Объектный код** обрабатывается специальной программой, называемой **редактором связей** или **сборщиком**, который выполняет связывание объектных модулей и машинного кода стандартных функций, находя их в библиотеках, и формирует на выходе работоспособное **приложение** (*программу*) – **исполнимый код** (***.exe-файл***) для конкретной компьютерной платформы.
4. **Исполнимый код** – это законченная программа, которую можно запустить на любом компьютере, где установлена операционная система, для которой эта программа создавалась. Как правило, итоговый файл имеет расширение **.exe** или **.com**
5. Современные системы программирования имеют еще один компонент – *отладчик*. Отладчик позволяет анализировать процесс выполнения программы в пошаговом режиме и искать возможные ошибки в тексте программы.

В последнее время созданы и находятся в использовании системы программирования быстрого создания программы, называемые **средами быстрого проектирование (RAD - среды).** В таких средах, в результате автоматизации многих процессов подготовки современных оконных приложений (программ), работы программирования заменяются работами проектирования, что позволяет разработчику больше сосредоточиться на логике решаемой задачи. Такой средой разработки, например, является **Borland Delphi** (просто **Delphi**).

Интегрированная среда быстрой разработки программ - Delphi

Эта система программирования основана на использовании языка программирования Delphi. Система Delphi 1 появилась на рынке в 1995 году. Она объединяет *средства системы программирования* и *средство визуального проектирования окон приложений*. Потому ее еще называют **интегрированной средой программирования**. Среда программирования Delphi является одним из безусловных лидеров среди современных систем программирования. Это глубоко продуманный, высокоэффективный и (что немаловажно) весьма удобный программный продукт, позволяющий создавать приложения практически любой сложности, предназначенные для работы под управлением операционных систем Microsoft® Windows® и Linux.

Изначально Delphi специализировалась только на создании программного обеспечения под Windows. Для этого среда снабжена глубоко проработанной и эффективной библиотекой визуальных компонентов (VCL, Visual Compoents Library), элементы которой не только инкапсулировали в себе функции прикладного программного интерфейса (API, Application Program Interface) Windows, но и внесли существенные усовершенствования. Благодаря этому библиотека VCL успешно конкурирует с библиотекой MFC (Microsoft Foundation Class), разработанной в корпорации Microsoft, и служит фундаментом альтернативным Microsoft Visual Studio средам программирования Borland Delphi и Borland C++.

В условиях жесткой конкуренции фирма Borland постоянно развивает и улучшает возможности среды разработки. Начиная с шестой версии Delphi в состав среды разработки вошел пакет кроссплатформенной разработки CLX (Borland Component Library for Cross-Platform), основанный на идеях, апробированных в VCL. Delphi 2005 впитала идеи создания распределенных программных продуктов, базирующихся на архитектуре Microsoft® .NET Framework.

Особенность пакета CLX в том, что он позволяет строить приложения не только для Windows, но и для набирающей обороты ОС Linux. Тем самым программисты Delphi получили еще одно существенное преимущество - переносимость приложений между разными операционными системами. Однако за универсальность платформы пришлось заплатить - приложения CLX вынуждены отказаться от вызова функций, специфичных для каждой из операционных систем. В связи с этим опора на CLX не столь рациональна в тех случаях, когда вы нацелены только на работу с Windows. Поскольку мы рассматриваем вопросы программирования для Windows, то мы больше не будем возвращаться к CLX и системе Kylix (дополнение к Delphi для работы с CLX).

Перечисляя заслуги Delphi, стоит упомянуть доступность и интуитивную понятность интерфейса среды, наглядность кодовых конструкций языка Object Pascal (Дельфи), надежную систему выявления ошибок, высокоэффективный компилятор, умение поддерживать самые распространенные форматы баз данных и многое другое.

Версия Delphi 7 (2001 год) стала последней и наиболее развитой версией среды, реализованной для платформы Win32. В ней, в частности, появился компилятор для .NET, пока в консольном варианте. Добавилось множество компонентов и Мастеров, автоматизирующих создание сетевых приложений. Была продолжена поддержка кросс-платформной технологии создания программ на базе Delphi и Kylix.

Версия Delphi 8 (2003 год) считается первой версией Delphi для платформы .NET. Переход был кардинальным: из Delphi 8 был полностью исключен компилятор для платформы Win32. В то же время допускалась создание приложений как с помощью стандартного набора компонентов .NET, так и с помощью новой версии библиотеки Borland VCL.NET.

Версия Delphi 2005 (2004 год) вновь включила средства разработки для Win32 — в дополнение к поддержке .NET. В нее также вошел компилятор языка C# — стандартного языка разработки для платформы .NET.

Версия Delphi 2006 и все последующие версии среды развиваются не только наращиванием функциональных возможностей, вводом новых наборов компонентов или включением новых **информационных технологий**. Практически все современные технологии уже вошли в Delphi в той или иной форме — ведь жизненный цикл технологии программирования составляет не менее трех лет, а версии Delphi выпускаются каждый год. Среда Delphi теперь активно расширяется внешними продуктами, охватывающими все большее число этапов создания программного продукта. Автоматизируется работа по анализу требований, модельному проектированию, организации дистанционного доступа к версиям кода, автоматическому тестированию и развертыванию. Таким образом, **система Delphi** становится законченным комплексом, предлагающим все необходимые инструменты для организации **жизненного цикла программного обеспечения**. Современная система Delphi – это система объектно-ориентированного визуального программирования. Приложения (прикладные программы) Delphi являются интерактивными системами, в которых для организации взаимодействия между пользователем и программой используются подпрограммы, управляемые событиями.

Резюме

- Работой компьютера управляет программное обеспечение (компьютеры часто упоминаются под термином аппаратное обеспечение).

- Компьютер — это прибор, работающий по программе и способный производить вычисления и принимать логические решения в миллионы или даже миллиарды раз быстрее человека.
- Компьютеры обрабатывают данные под управлением компьютерных программ. Эти компьютерные программы предписывают компьютеру последовательно выполнять набор действий, которые указали люди, называемые компьютерными программистами.
- Различные устройства (такие как монитор, диски, клавиатура, память и процессор), которые входят в состав компьютера, называются аппаратными средствами.
- Программы, которые выполняет компьютер, называются программным обеспечением.
- Входной блок является «воспринимающей» частью компьютера. Основная информация вводится сегодня в компьютеры с помощью клавиатур, подобных пишущей машинке. Этот блок получает информацию (данные и компьютерные программы) от различных устройств ввода данных и делает эту информацию доступной для других модулей компьютера.
- Выходной блок является частью компьютера, «транспортирующей» информацию на устройства вывода. Большая часть информации выводится из компьютеров путем отображения на экране или печати на бумаге.
- Блок памяти является «складирующей» частью компьютера, к которой имеется быстрый доступ, однако размеры такой памяти обычно невелики. Этот блок хранит информацию, которая была введена через устройства ввода; делает ее немедленно доступной для обработки, когда это необходимо; сохраняет информацию, которая уже была обработана, пока эта информация не будет помещена в устройства вывода блоком вывода.
- Арифметико-логическое устройство (ЛЛУ) выполняет вычисления и принимает решения.
- Центральное процессорное устройство (ЦПУ) является «административным« разделом компьютера. Это координатор компьютера, который ответственен за работу других блоков компьютера.
- Блок внешней памяти — это долговременное, большой емкости хранилище данных компьютера. В этом блоке обычно используются диски. Сюда помещаются на хранений данные и программы, не используемые другими блоками компьютера, возможно, на несколько часов, дней, месяцев или лет, словом, до тех пор, пока они не понадобятся снова.
- Все языки программирования могут быть разделены на три основные типа: машинные языки, ассемблерные языки и языки высокого уровня.
- Любой компьютер может непосредственно понимать только свой собственный машинный язык. Машинные языки в общем случае содержат ряды чисел (в конечном счете сводимых к единицам и нулям), которые являются командами компьютеру на выполнение большинства элементарных операций в тот или иной момент времени. Машинные языки машинно-зависимы.
- Англо-подобные аббревиатуры образуют основу языков ассемблера. Программы трансляторы, называемые ассемблерами, транслируют программы на языке ассемблера в машинные коды.
- Компиляторы транслируют программы на языках высокого уровня в машинные коды. Языки высокого уровня (например, Delphi) используют английские слова и общепринятую математическую нотацию.
- Интерпретирующие программы непосредственно выполняют программы на языках высокого уровня и не требуют их трансляции в машинный код.
- Хотя откомпилированные программы выполняются быстрее, чем интерпретируемые программы, интерпретаторы используются в тех случаях, когда программы часто перекомпилируются для добавления в них новых возможностей и исправления ошибок. Но когда разработка программы завершена, ее откомпилированная версия будет выполняться более эффективно.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Delphi – одна из самых мощных систем программирования, позволяющих на самом современном уровне создавать как отдельные прикладные программы Windows, так и разветвленные комплексы, предназначенные для работы в корпоративных сетях и в Интернет. Создавать в Delphi не слишком сложные приложения проще. Можно создавать мощные системы работы с локальными и удаленными базами данных любых типов; при этом имеются средства автономной отладки приложений с последующим выходом в сеть. Из прекрасного средства создания приложений для Windows Delphi уже превратилась в инструмент создания приложений для многозвенных распределенных кросс-платформенных корпоративных информационных систем.

Вопросы для самопроверки

1. Что такое компьютер?
2. Что такая программа?
3. Что такое алгоритм?
4. В чем различие программы и алгоритма?
5. Классификация языков программирования
6. Что такое машинный язык?
7. Что такое ассемблерный язык?
8. Что такое язык высокого уровня?
9. Чем отличаются языки программирования?
10. Что такое система и среда программирования?
11. В чем различие между компилятором и интерпретатором?
12. Что такое парадигма?
13. Что такое парадигма программирования?
14. Какие парадигмы программирования бывают?
15. Какие парадигмы программирования считаются основными?
16. Отличие системы программирования и интегрированной среды разработки.

Лекция 2. Язык программирования Delphi. Структура программы

План
Алфавит языка Delphi
Лексемы.
Структура программы.
Процедуры вывода
Процедуры ввода
Компиляция и запуск программы на выполнение

Алфавит языка Delphi

Язык Delphi представляет собой существенно улучшенную версию языка Паскаль, созданного в конце 1960-х годов. Основное достоинство Паскаля — удачное сочетание наглядности, компактности и простоты изучения с эффективностью компиляции и исполнения результирующего кода. Кроме того, в Паскаль встроены мощные средства контроля за распространенными ошибками, что позволяет существенно сократить процесс отладки (поиска ошибок). Недаром на основе Паскаля было создано обширное семейство новых языков программирования (Модула, Оберон и другие).

Язык Delphi поддерживает оригинальный синтаксис Паскаля, что позволяет использовать многолетние мировые наработки, сделанные на этом языке. Язык Delphi является объектно-ориентированным расширением языка Паскаль, т.е. он дополняет стандартные возможности Паскаля объектными технологиям, а также и средствами тесной стыковки с интерфейсами вычислительных платформ Win32 и .NET.

Всякий искусственный язык как любой естественный язык имеет свой алфавит.

Алфавит языка Delphi состоит из:

1. букв латинского алфавита;
2. десятичных цифр от 0 до 9 и шестнадцатиричных цифр: 0, 1, 2, 3,4,5,6,7,8,9,A,B,C,D,E,F;
3. специальных символов, знаков. К ним относятся следующие знаки и комбинации:
   - Одиночные символы: + - * / = , ‘ . : ; < > [ ] { } ( ) ^ @ & $ # пробел и знака подчеркивания "_"
   - Пары символов: (* (. *) .) .. // := <= >= <>

Из таких знаков строятся **первичные конструкции** языка, называемые **лексемами** или **словами**. Они являются элементарными (неделимыми) лексическими единицами текста программы. Каждая такая лексема и человеком, и компьютером интерпретируется («понимается») только в определенном, однозначном смысле. Так что лексемы являются синтаксическими единицами языка. В языке затем из таких простых строятся сложные конструкции (предложения) языка, например, *операторы* и *выражения.*

Лексемы

Лексемами мы называем:

1. зарезервированные (или ключевые, или служебные) слова;
2. идентификаторы;
3. константы;
4. разделители.

Зарезервированные слова

Зарезервированные слова – это некоторые слова английского языка или их сокращения. Их еще называют ключевыми словами. Ключевые слова являются синтаксическими элементами языка Delphi. Их нельзя модифицировать или использовать в целях, отличных от заданных стандартом языка. К ключевым относятся, например, слова **begin, end**, **program** и еще несколько десятков других. В окне редактора Delphi ключевые слова выделяются полужирным шрифтом. Их нельзя использовать в качестве идентификаторов.

Идентификаторы

Идентификаторы в Delphi – это имена обрабатываемых данных или имена методов обработки. Идентификаторы делятся на *стандартные идентификаторы* и *пользовательские идентификаторы*.

Стандартные идентификаторы заранее определены разработчиками языка и имеют заданные ими смыслы. Примерами стандартных идентификаторов являются, например, Sin, Pi, Real. Их желательно использовать только для назначенных целей.

Пользовательские идентификаторы применяются в качестве *имен объектов* (сущностей), используемых в программе. Идентификаторы не должны совпадать с ключевыми словами языка. Идентификаторами обозначают переменные величины, константы, процедуры и функции, типы данных, свойства, поля, библиотеки, пакеты, модули и программы. Длина идентификатора может быть любой, но учитываются только первые 255 его символов. Идентификатор может содержать латинские буквы, цифры и символы подчеркивания _, но **не должен начинаться с цифры**. Запись идентификатора не должна разрываться знаками пробела или конца строки. Итак, пользовательские идентификаторы задаются программистом и должны отвечать следующим правилам:

1. идентификатор составляется из букв, цифр и знака подчеркивания;
2. идентификатор не должен начинается с цифры;
3. в идентификаторе можно использовать как строчные, так и прописные буквы, компилятор интерпретирует их одинаково. Как известно, имя на естественном языке может быть состоять из нескольких слов. В программировании такие имена (идентификаторы) задаются их слитным написанием, выделяя начало следующего слова с большой буквы. Например, todayDate (сегоднешняя дата).
4. имена, используемые в программах, должны соответствовать назначению, обладать узнаваемостью, обеспечивать запоминаемость, быть краткими, обладать уникальностью.

Примеры правильных идентификаторов:

A a1 myProgram _1 a_1 my_program ALPHA

Примеры неправильных идентификаторов:

1A - начинается с цифры
a@1 - содержит специальный символ
my program - содержит пробел
end - зарезервированное слово

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Константы

Константы (или **литералы**)– это хранящееся в компьютере число (или строка), значение которого (которой) остается неизменным на протяжении всего времени выполнения программы. В качестве констант в Delphi могут использоваться целые, вещественные и шестнадцатеричные числа, логические константы, символы, строки символов, конструкторы множеств и признак неопределенного указателя nil.

Целые числа записываются со знаком или без него по обычным правилам и могут иметь значение в диапазоне от -2^{62} до $+2^{63}$ - 1. Следует учесть, что если целочисленная константа выходит за указанные границы, компилятор дает сообщение об ошибке. Такие константы должны записываться с десятичной точкой, т. е. определяться как вещественные числа.

Вещественные числа записываются со знаком или без него с использованием десятичной точки и/или экспоненциальной части. Экспоненциальная часть начинается символом *e* или E, за которым могут следовать знаки + или - и десятичный порядок. Символ *e (E)* означает десятичный порядок и имеет смысл "умножить на 10 в степени". Например,

3.14E5 — 3,14 умножить на 10 в степени 5;

-17e-2 — минус 17 умножить на 10 в степени минус 2.

Если в записи вещественного числа присутствует десятичная точка, перед точкой и за ней должно быть хотя бы по одной цифре. Если используется символ экспоненциальной части *e (E)*, за ним должна следовать хотя бы одна цифра десятичного порядка.

Шестнадцатеричное число состоит из шестнадцатеричных цифр, которым предшествует знак доллара $ (код символа 36). Диапазон шестнадцатеричных чисел — от $FETFETFFFFFFFFFF до $7ETFFFFFFFFFFFF (для версий 4—7).

Логическая константа— это либо слово False (Ложь), либо слово True (Истина).

Символьная константа — это любой символ ПК, заключенный в апострофы:

‘z’ — символ "z";
‘Ф’ — символ "Ф".

Если необходимо записать собственно символ апострофа, он удваивается:

‘’’’ — символ ' (апостроф).

Как известно, любой символ в памяти компьютера представляется в виде целого числового кода. Поэтому допускается использование записи символа путем указания его внутреннего кода, которому предшествует символ # (это управляющий символ, его код 35), например:

- #97 — символ a
- #90 — символ Z
- #39 — символ '
- #13 — символ CR

Строковая константа — любая последовательность символов (кроме символа CR — возврат каретки), заключенная в апострофы. Если в строке нужно указать сам символ апострофа, он удваивается. Строка символов может быть пустой, т. е. не иметь никаких символов в обрамляющих ее апострофах. Строку можно составлять из внутренних кодов нужных символов с предшествующими каждому коду символами #, например, строка #83#121#109#98#11#108 эквивалентна строке Symbol. В строковых значениях допускается комбинация символов и кодовых значений. Например, значение строки Delphi может выглядеть так: 'D'#101'l'#112'h'#105

Константы делятся на *именованные* и *неименованные*. В тексте программы запись вида, например, 105 – это неименованная константа, а именованная константа должна быть объявлена с помощью оператора *const*. Синтаксис оператора *const* имеет вид

```
const
  имя_константы1 = выражение1;
  имя_константы2 = выражение2;
.....................................................
```

Чтобы имя константы отличать от других имен можно писать его прописными буквами с символом подчеркивания. Например,

```
Const
    MAX_WORK_WEEKS = 105;
```

При использовании такого объявления в тексте программы можно пользоваться введенным именем константы. Именованная константа может быть определена с помощью любого *математического* или *строкового выражения*.

Разделители

Разделителями являются пробелы, символ конца строки, символ табуляции, а также любой специальный символ алфавита. Каждый символ разделителя используется в программе только для определенной цели. Потому символы разделителя могут использоваться для отделения друг от друга остальных лексем текста. Потому их называют разделителями или еще **ограничителями**. Они указывают начало и конец очередной лексемы (отдельных лексических единиц) в тексте программы.

Структура программы

«Программа - это идея, которую программист изложил на языке программирования». Такое определение во главу угла ставит не сотни строк безликого кода, а ее величество идею, то, без чего немыслимо существование творческой личности. Это определение - достойный ответ спорщикам на тему: «Что такое программирование - ремесло или искусство?»

Программа на языке программирования Delphi записывается с помощью *набора символов в кодировке Unicode,* При этом **строчные и прописные буквы не различаются**. Так, слова program, Program и PROGRAM означают одно и то же.

Текст программы представляет собой последовательность ***предложений*** *(называ… операторами),* разделенных символом **;** (точка с запятой). Каждое предложение представляет собой *команду* (программное действие) или *описание элемента программы*, используемого в командах. Каждое предложение строится из выше названных лексем. Для наглядности слова текста программы отделяют пробелами.

В тексте программы еще могут быть **комментарии** и **директивы**.

Комментарии в тексте программы

Во всех без исключения языках программирования предусмотрена возможность комментирования строк исходного кода. В комментарии программист в сжатом виде описывает, что делается в этих строках, для чего введена данная переменная, что произойдет после вызова процедуры. Другими словами, в комментариях разработчик кода кратко поясняет смысл рожденных в его голове команд. В результате листинг программы становится более понятным, более читаемым и доступным для изучения.

Для того чтобы при компиляции программы текст комментариев не воспринимался Delphi как исходный код программы и не служил источником ошибок, приняты следующие соглашения. Комментарием считается:

1. Отдельная строка, начинающаяся с двух наклонных черт //. Например,

```
//Это одна строка комментария
```

2. Весь текст, заключенный в фигурные скобки {... } или в круглые скобки с символами звездочек (* ... *). Например,

```
1. { Это многострочный текст
   комментария }
2. (* Это также комментарий *)
```

Текст комментария пропускается компилятором и не оказывает влияния на «жизнедеятельность» программы.

Директивы

Если внутри фигурных скобок на первой позиции окажется символ $, то это не что иное, как *директива компилятора*. Ниже дан шаблон программы, где используется такая директива:

```
{$APPTYPE CONSOLE}
```

В данном случае это означает, что наша программа является консольным приложением. Начинающему программисту не следует изменять ни содержимое таких строк, ни место их расположения, в противном случае есть риск привести свой проект в негодность.

Составление и написание программы на языке Delphi можно делать в *интегрированной среде разработки приложений* (IDE) фирмы Borland. Интегрированная среда разработки – это рабочая среда Delphi, в которой совмещены (интегрированы) все стадии процесса программирования,

включая редактирование, компиляцию и отладку. Программы, разрабатываемые такой системой принято называть *приложением* (*проектом*).

Система Delphi предназначена для создания приложений, работающих под управлением Windows, однако с ее помощью можно создать также приложения для DOS. Программы для DOS обычно не имеют графического интерфейса пользователя, вместо пиктограмм в них используются текстовые меню, а результат обработки информации выводится в файлы или на экран в текстовом виде. В системе Delphi программы для DOS называются *консольными приложениями*. Приложения с графическим интерфейсом пользователя, работающие под управлением Windows, принято называть *оконными приложениями*.

Для изучения применений конструкций языка Delphi мы будем использовать консольное приложение. **Консольное приложение** имеет следующую структуру**:**

```
program Project1;  // заголовок программы

// описательная часть программы
$APPTYPE CONSOLE}
uses
  SysUtils;
  ..........

// исполнительная часть программы
begin
  // здесь приводятся предложения (операторы),
  // описывающие действия алгоритма
end.
```

Итак, консольное приложение имеет *заголовок*, *описательную* и *исполнительную части*. Прежде чем разбираться, что содержится в шаблоне кода консольного приложения, научимся сохранять плоды своей деятельности в виде файла. Для этого выберем пункт меню *File -> Save* и в открывшемся диалоговом окне присвоим своему первому проекту имя FirstPrj.dpr. В завершение нажмем кнопку ОК.

Разрабатываемая программа на языке Delphi может включать десятки или даже сотни файлов различного типа и назначения. Совокупность файлов Delphi, используемых для разработки определенной программы, принято называть *проектом*. Проект Delphi состоит из нескольких типов файлов. Из них наиболее важные – *файлы проектов*, *файлы модулей* и *файлы форм*. Файл проекта имеет расширение ***.dpr*** (сокращение от Delphi Project). Он выполняет роль главной подпрограммы из него выполняется вызов файлов модулей и форм, входящие в данный проект. Но как минимум программа состоит из одного файла - главного файла проекта Delphi. Файл такого типа идентифицируется расширением ***.dpr*** (сокращение от Delphi Project).

Теперь, когда наш проект сохранен на жестком диске компьютера, вновь обратите внимание на первую строку кода. Вместо имени проекта по умолчанию, Project1, после ключевого слова program появится предложенное нами название FirstPrj. Следующее ключевое слово uses применяется для подключения к проекту внешних модулей, как правило, содержащих *библиотеки*

...олнительных подпрограмм. В частности, наш проект будет эксплуатировать наиболее ч... используемую библиотеку системных утилит SysUtils. Шаблон завершается составным оператором *begin .. end*, внутри которого размещается выполняемый код. Пока же, в нашем примере, здесь стоит только текст комментария.

Мы в начале курса будем рассматривать только *консольное приложение*. В таком приложении исходный текст программы представляется в виде последовательности строк, при этом текст может начинаться с любой позиции строки. Можно еще сказать, что структурно текст такой программы (приложения) состоит в этом случае из **заголовка** и **блока**. **Заголовок** находится в начале программы и имеет вид:

Program имяПрограммы;

Блок же, как выше сказано, состоит из двух частей: **описательной** и **исполнительной**. В *описательной части* содержатся *описания объектов* текста программы, а в *исполнительной* указываются *действия с объектами* программы, позволяющие получить требуемый результат.
Структуру программы более подробно в общем случае можно представить так:

```
program ИмяПрограммы;
// ниже описательная часть программы
// здесь могут быть еще директивы для компилятора
uses
  список модулей; //, используемых в этой программе;
label
   список меток; //, используемых в этой программе;
const
   список констант; //, используемых в этой программе
type
   список типов;  // пользователя, используемых в этой программе
var
   Объявления переменных; //, используемых в этой программе
Описание процедур;    //, используемых в этой программе;
Описание функций;     //, используемых в этой программе;
// ниже исполнительная часть, ее еще называют телом программы,
// в ней описываются действия алгоритма с помощью операторов в терминах
//вышеописанных объектов
begin
  Оператор1;
  .....
  ОператорN;
end.
```

В описательной части просто описываются, т.е. просто перечисляются какие *обозначения* меток, констант, какого *названия* типы данных, какие *модули* из библиотеки модулей программного обеспечения используются в тексте программы. А также *описываются* (приводятся) *полные тексты процедур* и *функций*, которые используются в тексте данной программы. Итак, процедура или функция, в случае использования в данной программе, должна полным текстом приводиться в описательной части программы. Еще переменные перед использованием в тексте программы *должны быть обязательно объявлены* в предложении **var** описательной части.

программировании следует различать термины **объявление** и **описание**. Объявл… некоторого объекта в программе предполагает выделение ячеек памяти для его размещения в оперативной памяти компьютера, а описание же нет.

В виде *оператор1, ... операторN* записываются действия *алгоритма*. *Программа* – это запись алгоритма решения той или иной задачи на выбранном языке программирования, например, на языке Delphi в терминах операторов языка. Итак, в исполнительной части описываются действия алгоритма с помощью операторов языка. Такие действия алгоритма как вывод результата обработки и ввод обрабатываемых данных в память компьютера в тексте программы принято оформлять в виде процедур.

Процедуры вывода

Процедуры названиями **Write** и **Writeln** выводят информацию на экран. Общий формат этих процедур следующий:

```
Write(выражение1, выражение2,…); //после вывода курсор остаётся в конце
                                 //  строки
Writeln(выражение1, выражение2,…); //после вывода курсор переносится в
                                   // новую строку
```

Если в качестве выражения принять строковую константу, то, например, операторы вывода

```
Writeln('Я учусь программировать в Дельфи');
Write('Я учусь  ');
Writeln('программировать в Дельфи');
```

выводят на экран в двух строках текст

```
Я учусь программировать в Дельфи
```

Процедуры ввода

Процедуры названиями **Read** и **Readln** вводят данные в программу. Общий формат этих процедур следующий:

```
Read(имя1, имя2, имя3, …);
Readln(имя1, имя2, имя3, …);
```

В данных процедурах внутри скобки не могут стоять *выражения*. Процедура Readln без ( ) означает ожидание на нажатие любой клавиши, тем завершается консольное приложение. В лекции № 3 будет больше информации о процедурах ввода и вывода.

Компиляция и запуск программы на выполнение

Теперь научимся компилировать программу. **Компилирование** - это процесс, переводящий программу с языка программирования (в нашем случае с языка Delphi) на язык машинных команд. Как-то неинтересно компилировать пустой проект, поэтому давайте научим его чему-нибудь полезному, например, здороваться. В следующем листинге предложен пример такой исключительно воспитанной программы. А для того чтобы исходный код был понятнее, он буквально насквозь пропитан комментариями.

```
program FirstPrj;
  {это листинг самой короткой и доброжелательной программы на свете}
  {$APPTYPE CONSOLE} {это директива компилятора, которую нельзя изменять}
uses
   SysUtils;  (*  это строка подключения внешних библиотек подпрограмм *)
begin
  WriteLn('Hello, World!');    //выводим текст «Привет, Мир!»
  ReadLn;           //ожидаем ввод - нажатие любой клавиши завершит работу
end.
```

Первый оператор этой программы состоит из следующих лексем: **WriteLn** – идентификатор (стандартный), **(** - разделитель, **'Hello, World!'** – строковая константа, **)** – разделитель, **;** - разделитель. Второй оператор состоит из одной лексемы идентификатор – Readln. Этот оператор используется для того чтобы экран с выданным текстом сразу не погас.

Вставив между begin и end шаблона консольного приложения указанные операторы, выберите пункт главного меню Delphi *Run -> Run*. Если все повторено безошибочно, то за считанные доли секунды на экране появятся плоды нашего коллективного творчества - консольное окно со строкой «Hello, World!» Если же вдруг была допущена ошибка, то компилятор просигнализирует о ней, выделив в листинге строку, содержащую предполагаемую ошибку или следующую за ней строку.

Вместо утомительных поисков необходимого элемента в меню у программистов Delphi наибольшей популярностью пользуется быстрая клавиша запуска программы - функциональная клавиша F9. При нажатии этой клавиши осуществляется проверка синтаксиса проекта, его компиляция и запуск исполняемого exe-файла.

Как видите, вся программная логика сосредоточена внутри, так называемого, составного оператора **begin ... end** и выполняется линейно, т.е. последовательно в соответствии с очередностью следования строк, начиная с первой.

Здесь мы хотим обратить внимание на то, что *умение пользоваться языком программирования и умение программировать - это вовсе не одно и то же*. На самом деле написание программы является всего лишь одним из этапов в ее изготовлении, причем далеко не самым трудным, а скорее наоборот. И далеко не случайно этот этап часто называют не «программированием», а «*кодированием*» - подчеркивая тем самым, что эта работа носит в основном технический характер, хотя качественное его исполнение, разумеется, имеет немаловажное значение.

В самом деле, непосредственное написание фраз на языке программирования (*операторов, описаний, объявлений* и т.д.) – это как бы укладывание кирпичей или других строительных блоков в строящееся здание, каковым в данном случае является программа. Однако известно, что строители никогда не начинают непосредственного строительства здания, пока не будет разработан достаточно детальный его **проект**, т.е. не будет определено его назначение, общая архитектура, назначение и взаимное расположение отдельных помещений и т.д.

Аналогично обстоит дело и при изготовлении программы. К началу непосредственного написания ее текста должны быть четко определены назначение программы, ее исходные данные и требуемые результаты ее выполнения, разработана структура программы, т.е. четко выделены составные части (*процедуры и функции*) будущей программы, точно определено назначение каждого из них и их взаимодействие, т.е. связи по данным и порядок выполнения. Именно разработка *алгоритма*, *детальное проектирование* будущей программы, и является сутью программирования, наиболее важным и ответственным этапом в изготовлении программы. Так что квалификация программиста заключается в умении выполнить именно эту часть работы – точно так же, как при создании того или иного здания (сложного технического изделия) решающую роль играет квалификация его архитектора (конструктора). Строго говоря, *для проектирования программы* и *для ее написания (кодирования)* требуются специалисты разных профилей, поскольку на каждом из этих этапов приходится решать свои, специфичные проблемы. Это обстоятельство следует иметь в виду и в том достаточно распространенном случае, когда

ектирование и написание программы производится одним и тем же лицом (программис… Игнорирование этого обстоятельства является одним из наиболее существенных и распространенных недостатков в работе начинающих программистов. А проявляется этот недостаток в том, что получив задачу, которую необходимо подготовить к решению на компьютере, программист стремится как можно быстрее перейти к непосредственному написанию программы (например, на Delphi), не спроектировав ее предварительно. *В итоге ему приходится параллельно выполнять два совершенно разных вида работ – и разработку программы, и ее кодирование.* Так делать, повторяем еще раз, неправильно.

Итак, приступать к *кодированию* (к записи текста программы на том или ином языке программирования) имеет смысл лишь после того, как завершена *разработка (проектирование)* программы – даже в том случае, если эти два этапа работ выполняются одним и тем же программистом. Если даже программируются очень простые задачи, надо учиться этапы проектирования и кодирования выполнять строго в указанной последовательности.

Резюме

Лексемы первичные конструкции языка, построенные из алфавита и имеющие однозначные смыслы или предназначения. Компилятор «умеет выделять и интерпретировать» отдельные лексемы текста программы. Поэтому для правильного написания предложения текста программы программист должен знать лексемы языка программирования и правильно их использовать строго по назначению.

Консольное приложение Delphi оформляется по правилам (по парадигме) процедурно-ориентированного программирования. В таком случае тексты, используемых в консольном приложении процедур и функций, должны явно приводиться до исполнительной части приложения. Такова будет внутренняя организация Делфи-программы при процедурно-ориентированном программировании.

Вопросы для самопроверки

1. Как выглядит структура программы на языке Delphi?
2. Что такое константа?
3. Что такое именованная константа?
4. Какие лексемы бывают?
5. Знаете ли вы разницу между описанием и объявлением?
6. Что такое интегрированная среда разработки программ?
7. Какие бывают виды комментарий?
8. Что такое проектирование и кодирование программы?
9. Что такое система программирования?
10. Какое различие между системой и интегрированной средой программирования?
11. Какая разница между приложением и программой?
12. Как вы понимаете термин «программирование»?
13. Что такое неименованная константа?
14. Как называется структура консольного приложения Delphi?
15. Что такое парадигма программирования?
16. Какие парадигмы программирования вы знаете?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 3. Информация, данные и их типы. Процедуры ввода, вывода

План
Информация и данные
Именование, объявление и использование переменных
Типы данных.
Стандартные типы данных
Процедуры ввода, вывода.

Информация и данные

С помощью программы в компьютере обрабатываются данные. Под термином данные принято понимать представление фактов и (или) идей в формализованном виде, пригодном для передачи и обработки в некотором процессе, например в процессе, реализуемом аппаратурой компьютера. Итак, д***анное* – это *условное представление информации*, т.е. представление с помощью знаков какого-то алфавита, иначе говоря, данное – это какой-то *код*. *Информация* же есть *смысл*, связанный с этим условным представлением или кодом.** Смысл закладывает человек, который условное представление ввел. Например, 5, пять, беш, V могут быть разными кодами одной и той же информации. Этого знает только тот человек, который ввел этот обозначение или код. Обрабатывая на компьютере данные, можно говорить, что мы обрабатываем информацию. Так говорится в том смысле, что пользователь программы знает какие данные с какими смыслами обрабатывается компьютером по его программе и какие результатные данные со смыслами должны получиться. Так что обработка информации компьютером ведется опосредованно, с помощью обработки данных.

В программах обрабатываемые данные фигурируют в качестве значений (характеристики) тех или иных *программных объектов*. Данные, которые зафиксированы в тексте и не изменяются в процессе ее выполнения, являются значениями таких программных объектов, как *константы*; остальные данные являются значениями объектов, называемых *переменными*, поскольку значения этих объектов возникают и могут изменяться в процессе выполнения программы.

В аппаратуре компьютера, из-за специфики ее элементарной базы, любые данные представляются в виде последовательностей двоичных цифр 0 и 1, изображаемых тем или иным способом их комбинаций.

Одно из преимуществ языков программирования высокого уровня как раз и состоит в том, что они позволяют абстрагироваться от деталей, от конкретного способа представления данных в компьютере, за счет концепции *типа значений.* Каждый такой тип, предусмотренный в языке, определяет как *множество значений этого типа,* так *и набор операций над ними.*

Именование, объявление и использование переменных

При работе программы, обрабатываемые ею данные, хранятся в ячейках оперативной памяти компьютера. Как известно, ячейки памяти имеют только *целые числовые номера*, говорят, *адреса* или, можно сказать, *имена*. При обращении к ячейке в тексте программы указываются адрес ячейки. Сейчас принято эти адреса указывать не числовыми именами, а *буквенными именами*, т.е. говоря, термином программирования *идентификаторами*. Переводящей программе не трудно заменить буквенный адрес числовым. Как известно, буквенные имена (обозначения) в математике принято связывать с переменными. Значит, можно сказать, что в тексте программе для доступа к компьютерным данным используются переменные. ***Компьютерная переменная***

— это представление в языке программирования области памяти, отведенной для хранения значения (т.е. данного). В программе к переменной обращаются по ее имени (идентификатору). Так что, *переменная текста программы для компьютера – это имя ячейки ее памяти*. Значит, используемые в программе ячейки памяти для данных представлены в тексте программе как переменные. Тогда *значение переменной* есть *содержимое ячейки*. Итак, в программировании используется следующая аналогия:

Имя переменной	*Имя ячейки*
Значение переменной	*Содержимое ячейки*
Тип значения	*Тип содержимого*

Значения, содержащиеся в компьютерных и математических переменных, носят разный характер. Если математические переменные могут содержать только числовые данные, а компьютерные переменные могут содержать *как числовые, так и символьные данные* (т.е. комбинацию букв, цифр и других символов). Причем данные этих двух видов представляются в памяти компьютера по-разному, а также обрабатываются по-разному.

Как выше сказано в языке программирования, каждый т*ип данных определяет множество различных значений и их свойства (например, упорядоченность), а также операции, которые могут выполняться над этими значениями.* Итак, множество значений переменной (или содержимого ячейки) определяется ее типом.

Каждый тип должен быть каким-то образом специфицирован для его выделения среди всех возможных типов. В качестве спецификаторов могут использоваться служебные слова или обычные идентификаторы. Для достижения единообразия в этом отношении и для большего удобства, за стандартными типами (не требующими их явного описания в программе) закреплены стандартные имена. В языке паскаль имеются *четыре стандартных типа: целый, вещественный, символьный (литерный) и логический.* Так, термин целый (или целочисленный) употребляется в обычном смысле и обозначается стандартным именем *integer*. Аналогично за каждым стандартным типом закреплено свое стандартное имя. В тексте программы, используя эти имена, пишется инструкция по созданию переменной определенного типа. Компилятор, встречая инструкцию по созданию переменной определенного типа, отводит в оперативной памяти место, т. е. ячейку соответствующей величины, для хранения значения заказанного типа. Как известно, ячейка может состоят из одного или нескольких байтов. Количество байтов определяется типом переменной. Для доступа к этой области памяти используется имя переменной. Переменная должна быть явно декларирована до ее использования в других инструкциях. *Декларация (объявление) переменной - это инструкция компилятору о создании новой переменной заданного типа.* Объявление переменной записывают следующим образом (по правилам структурного программирования следует так писать):

```
var
  имяПеременной : имяТипа;
```

Например,

```
var
  X: integer;
```

Имена переменных должны начинаться с буквы или с символа подчеркивания (_). Имя переменной – это правильно построенный идентификатор. Например, A5 – правильное имя переменной, а 5A – неправильное.

Тип переменной определяет диапазон значений переменной и служит для компилятора показателем того, сколько байтов памяти она занимает.

Переменным следует присваивать описательные имена, чтобы можно было понять назначение переменных. Другой программист, читающий вашу программу (да и вы сами через некоторое время), не сможет понять, что означает имя A5. Для этого ему придется изучать документацию

или отслеживать переменную в программе. В то же время назначение переменной todaysDate (сегодняшняя дата) видно из ее названия, т.е. это имя является описательным.

В языке Delphi имена переменных должны начинаться с буквы английского (но не русского) алфавита или с символа подчеркивания _. За первым символом может следовать любое количество букв английского алфавита, цифр или символов подчеркивания, однако только первые 255 символов принимаются транслятором во внимание. Для компилятора Delphi имена переменных не чувствительны к регистру. Например, если объявить переменную todaysDate, то в исходном коде ее можно использовать под именем todaysDATE; компилятор воспримет их как одну и ту же переменную.

Часто символ подчеркивания в имени переменной используется для отделения различных составляющих ее слов друг от друга. Например, todays_Date. Мы будем вместо этого слова, входящие в переменную, выделяются так: первая буква второго и последующих слов набирается в верхнем регистре, например, todaysDate или costOfGoodsSold.

Итак, чтобы назначить переменной какой-либо тип данных, ее нужно объявить с этим типом. В Delphi каждая переменная должна быть объявлена явно. Для объявления переменной и назначения им типов в Delphi используется ключевое слово *var*. Объявление сообщает компилятору имя и тип переменной, а также количество байтов памяти, которые нужно зарезервировать для объявленной переменной (ниже будет показано соответствие числа байтов типу). Таким образом, *var* является невыполняемым оператором. Он не преобразуется компилятором в выполняемый код и является всего лишь инструкцией компилятору во время компиляции, который выполняется до выполнения программы. Наиболее общий синтаксис оператора *var* имеет вид

```
var
   имя_переменной11, имя_переменной12, … : тип_данных1;
   имя_переменной21, имя_переменной22, … : тип_данных2;
   ……
```

Чтобы исходный код был более понятным, рекомендуется объявлять каждую переменную в отдельной строке кода. Здесь «тип_данных» синоним «имяТипа».

Язык программирования Delphi поддерживает работу только с так называемыми *типизированными данными*. Любая обрабатываемая программой информация должна принадлежать к известному типу. Такой подход обеспечивает ряд преимуществ, Во-первых, несоответствия при обработке данных часто выявляются еще на этапе компиляции. Во-вторых, удается заранее оценить размер оперативной памяти для хранения значений определенного типа, что существенно повышает скорость обработки данных программой во время ее работы.

Компьютерная переменная может хранить два вида данных: числовые и символьные. Данные этих двух видов обрабатываются компьютером по-разному. Хранение символьных данных осуществляется относительно просто, для этого нужны только два типа данных: символы и строки. Строка – это последовательность символов, которая хранится в строковой переменной (т.е. в переменной типа строки), а символ - это неделимая часть строки. Для числовых переменных используется большее количество типов данных, так как числа могут быть целыми или вещественными. Целые – это числа без дробной части, поэтому обычно для их хранения требуется меньше памяти, чем для вещественных. Использование разных типов для одного типа данных позволяет экономнее расходовать оперативную память.

После объявления переменная обязательно должна быть инициализирована, т.е. ей должно быть присвоено какое-либо значение. В Delphi (как и в большинстве языков) переменная после объявления содержит так называемый «мусор» - значения битов, сохранившиеся в ее ячейках памяти от предыдущих операций. Хорошо запомните: перед первым использованием каждая переменная обязательно должна быть инициализирована. Присвоение переменной значения, например, можно делать с помощью операторов ввода или присваивания.

В ряде задач невозможно обойтись без преобразования значений из одного типа в другой. Так, целое число при выводе на экран требуется преобразовать в строковое представление, а дробную величину нередко требуется округлить до целого значения. Кроме того, системные механизмы Windows нередко работают с универсальным представлением данных, когда любой документ или мультимедийный файл обрабатывается как двоичный объект, не имеющий специфических свойств. Поэтому в языке Delphi введены механизмы гибкого преобразования типов и работы с нетипизированными данными.

Типы данных

Типы данных в Delphi можно разделить на *стандартные,* т.е. предопределенные в языке, и *определяемые программистом (пользовательские)*. К стандартным типам в Delphi относятся следующие: *целые, вещественные, символьные, строки, указатели, логические и variant.* В языке Delphi пользователь может определять собственные типы данных. Они могут основываться на уже существующих типах (в таком случае они называются производными типами) или же быть полностью оригинальными.

Каждое значение любого из производных типов в общем случае представляет собой уже нетривиальную структуру, т.е. обычно это значение имеет более чем одну компоненту. При этом каждая компонента структуры может быть как отдельным данным, так и свою очередь значением любого из производных типов. Производным типам относятся: массивы, файлы. Необходимость в массивах возникает всякий раз, когда при решении задачи приходится иметь дело с большим, но конечным количеством однотипных упорядоченных данных. Но существует определенный класс задач и определенные ситуации, когда количество компонент (пусть даже одного и того же любого из уже известных нам типов) заранее определить невозможно, оно выясняется только в процессе решения задачи, т.е. при выполнении программы. Поэтому возникает необходимость в специальном типе значений, которые представляют собой произвольные последовательности элементов одного и того же типа, причем длина этих последовательностей заранее не определяется, а конкретиризируется в процессе выполнения программы. Этот тип значений получил в паскале название файлового.

Пользовательские типы, как правило, определяются в описательной части программы с помощью ключевого слова ***type.*** Стандартные типы не нужно определять с помощью ключевого слова ***type*** (ибо за каждым стандартным типом закреплено свое стандартное имя), потому можно сразу использовать при объявлении переменных с помощью ключевого слова ***var***.

В этой лекции мы познакомимся следующими стандартными типами: *целыми, символьными, логическими, которых еще называют* ***порядковыми****, и вещественными, типом вариант, указателями.*

Порядковые типы характеризуются тем, что соответствующие им значения образуют конечное упорядоченное множество и каждое значение имеет свой порядковый номер. Для данных порядкового типа определены следующие функции:

- ***Ord(x)*** – возвращает порядковый номер значения x. Для целых типов возвращает само значение x, для логического 0 или 1, для символьного – значение в диапазоне от 0 до 255.
- ***Pred(x)*** – возвращает величину, предшествующую значению x.
- ***Succ(x)*** - возвращает величину, следующую за значением x.

Для константы или переменной порядкового типа определены также следующие функции:

- ***High(x)*** - возвращает максимальное возможное значение для аргумента x.
- ***Low(x)*** - возвращает минимальное возможное значение для аргумента x

Отношения между элементами любого из порядковых типов складываются вполне ординарно. Все элементы внутри одного типа могут располагаться в виде упорядоченной последовательности (следующий больше предыдущего). Для всех простых типов (за исключением целых чисел) справедливо утверждение, что первый (самый младший) элемент последовательности имеет индекс 0, второй - 1 и т. д. Целые же числа допускают хранение и отрицательных значений, поэтому здесь самый младший элемент последовательности может начинаться не с нуля. К еще одной особенности порядковых типов данных стоит отнести отсутствие знаков после запятой.

Для обеспечения эффективности и совместимости с внутренними интерфейсами Windows в язык Delphi добавлены такие типы, как *указатель* (указатель на значение в оперативной памяти), *процедурный тип* (описание процедуры как типа, что позволяет создавать переменные, ссылающиеся на процедуры), и *вариант* (неопределенный тип, позволяющий использовать двоичный блок данных как данные произвольного типа).

Стандартные типы

Целые типы

Значениями целого типа являются элементы подмножества целых чисел, зависящего от реализации языка. Поскольку в аппаратуре компьютера для изображения чисел отводится фиксированное количество разрядов (битов), то для каждой конкретного компьютера существует константа с именем maxint такая, что любое представимое в машине целое число N должно удовлетворять условию

-maxint <= N <= maxint

Так что, в первую очередь целые типы, описывающие целые числа, характеризуются пределом границ диапазона хранимых значений и возможностью описывать отрицательные величины. Чем больше предел допустимых значений, тем больший объем памяти будет занимать переменная этого типа. В следующей таблице 3.1 представлены целые типы языка Delphi.

Таблица 3.1. Целые типы языка Delphi

Имя типа	Диапазон значений	Размер в байтах
Shortint	От - 128 до 127	1
Smallint	От –32768 до + 32767	2
Integer (или ***Longint***)	От -2 147 483 648 до 2 147 483 647	4
Int64	От -2^{63} до $2^{63}-1$	8
Cardinal (или ***LongWord***)	От 0 до 4 294 967 245	4
Word	От 0 до 65 535	2
Byte	От 0 до 255	1

Использование разных типов для одного типа данных, например, для целочисленных данных позволяет экономнее расходовать оперативную память при решении задач. Как видно из таблицы, самым серьезным из предлагаемых типов является Int64, «пожирающий» целых 8 байт (64 бит) ОЗУ. Такой тип способен хранить величины, сопоставимые с количеством звезд во Вселенной. Как правило, для решения «земных» задач программисту достаточно диапазона значений типа Integer.

Попытка вычислить на машине выражение, целочисленное значение которого не принадлежит указанному диапазону, приводит либо к не верному результату этого вычисления, либо к

прекращению выполнения программы, в зависимости от особенностей данного компьютера. Это число maxint и определяет упомянутое выше подмножество целых чисел.

Считается, что целые числа (в отличие от вещественных) в компьютере должны представляться точно, и все определенные над ними операции также должны выполняться точно. Для такого точного представления значения целого типа в ячейках памяти используется форма представления, называемая как **форма с фиксированной запятой (точкой).**

Над целочисленными значениями в Delphi определены *пять основных операций*, результатом которого также является целое число (а деление «/» порождает нецелый результат) (таблица 3.2).

Таблица 3.2. Операции над целочисленными значениями

Знак операции	Содержание операции
+	Сложение
-	Вычитание
*	Умножение
div	Деление и «отсечение» (отбрасывание дробной части)
mod	Взятие остатка при делении

Операция div есть целочисленное деление: в качестве результата принимается целочисленное частное, а получающийся при делении остаток игнорируется. Например,
7 div 2 = 3,
3 div 5 = 0,
(-7) div 2 = -3,
(-7) div (-2) = 3.
Значение m mod n определено по формуле:
m mod n = m – ((m div n)*n),
Например,
7 mod 2 = 1,
3 mod 5 = 3,
(-14) mod 3 = -2
(-10) mod 5 = 0.

Логические типы

Сначала рассмотрим кратко некоторые элементы математической логики, из которой и возник этот тип значений. Математической логики является одной из ветвей общей логики – науки о формах и законах мышления, которая развивалась применительно к потребностям математики. Основу математической логики составляет *алгебра логики* или *исчисление высказываний,* причем здесь используется тот же язык формул, который характерен для математики вообще – это освобождает математическую логику от неопределенности в толковании логических выражений, показывающих связи между отдельными суждениями, понятиями и т.д.

Под *высказыванием* понимается любое *предложение*, в отношении которого можно однозначно сказать, *истинно оно* или *ложно*, например, «3>2», «5 – четное число», «Осло – столица Кубы» и т.д. Отвлекаясь от конкретного содержания высказывания, можно сказать, что истинность любого высказывания принимает одно из двух возможных логических (истинностных) значений: «*истина*» (если высказывание истинно) и «*ложь*» (если высказывание ложно).

Значение истинности высказываний могут изменяться в зависимости от обстоятельств, при которых сделаны эти высказывания. Например, истинность высказывания «сегодня - среда» зависит от того, в какой день недели оно сделано, а высказывания «x<0» - от конкретного значения числовой переменной x. В связи с этим возникает понятие *логической* или *булевской*

переменной, которая может принимать одно из двух возможных логических значений «истина» и «ложь» - подобно тому, как в математике имеется понятие переменной, принимающей числовые значения. В частности, каждому высказыванию можно сопоставить определенную логическую переменную. Булевский как синоним слова «логический» и *Boolean* как стандартное имя логического типа появился в честь математика Дж. Буль (Bool)

Итак, множество значений логического типа (***Boolean***) содержат всего два истинностные значения, каковыми являются идентификаторы **true** (истина) и **false** (ложь). Для хранения такого логического значения отводится один байт. В язык Delphi также введены логические типы ***ByteBool*** (*однобайтовая* логическая величина), ***WordBool*** (для хранения логического значения отводятся *два байта*) и ***LongBool*** (для хранения логического значения отводятся *четыре байта*). Основным логическим типом в Delphi является ***Boolean***. Остальные типы нужны для совместимости с логическими данными, используемыми в Windows и некоторых других системах программирования, например, Visual C.

Над логическими значениями в Delphi определены *логические операции*, которые обозначаются следующими служебными словами (операции указаны в порядке убывания их старшинства):

not – отрицание,
and – логическое умножение (логические И),
or – логическое сложение (логическое ИЛИ),
xor – логическое исключающее ИЛИ.

Эти операции, естественно, применимы только к логическим аргументам и дают результат этого типа (таблица 3.3).

Таблица 3.3 Операции над логическими значениями

X	Y	X **and** Y	X **or** Y	X **xor** Y	**not** X
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Вещественные типы

Значениями вещественного типа являются вещественные (действительные) числа, т.е. числа, имеющие дробную часть. Областью значений вещественного типа является определяемое реализацией языка подмножество множества всех вещественных чисел. Множество значений этого типа не относится к числу упорядоченных и потому к значениям этого типа неприменимы функции ***succ*** и ***pred.*** Вещественные числа в компьютере представляются в форме, называемой как *форма с плавающей точкой*, т.е. число в ячейке представляется как комбинация его *цифровой части* числа (мантиссы числа) и его *порядка*. Количество разрядов ячейки, отводимых на изображение мантиссы и порядка, определяется конкретным компьютером. В отличие от целого типа, этот диапазон содержит бесконечное подмножество вещественных чисел. Однако фиксированное количество разрядов, отводимых и для изображения мантиссы, приводит к тому, что в машине точно может быть представлено лишь ограниченное множество вещественных чисел. Таким образом, каждое машинное число представляет с той или иной точностью некоторой диапазон вещественных чисел. Так что вещественные числа представляются, вообще говоря, *неточно,* и арифметические операции над ними выполняются *неточно*, а по правилам действий над приближенными числами. В силу этих причин множество значений

вещественного типа не относится к числу упорядоченных. По этим же причинам операция сравнения значений этого типа на их точное равенство является некорректной и следует избегать ее использования в программах языка Delphi.

Итак, значения вещественного типа в ячейке представляется, в так называемой, **форме с плавающей запятой (точкой)**.

В Delphi имеется следующие вещественные типы (таблица 3.4), которые позволяют получить нужный результат практически с любой заданной степенью точности:

Таблица 3.4. Вещественные типы

Имя типа	*Диапазон значений*	Количество знаков	Размер в байтах
Real 48	От 2.9*lO $^{-39}$ до 1.7*lO^{38}	11-12	6
Single	От 1.5*lO $^{-45}$ до 3.4x10^{38}	7-8	4
Double (или ***Real***)	От 5.0 lO $^{-324}$ до 1.7*lO^{308}	15-16	8
Extended	От 3.6*lO $^{-4951}$ до 1.1*lO^{4932}	19-20	10
Comp	От -2^{63}+1 до 2^{63}-1	19-20	8
Currency	От -922337203685477.5808 до 922337203685477.5807	19-20	8

Над значениями вещественного типа определены *операции* сложения (+), вычитания (-), умножения (*), деления (/). Для вещественных типов операции *div* и *mod* не определены.

Если в программе необходимо производить вычисления действительных чисел, то по возможности объявляйте переменные как *Single* (если, конечно, вас устраивает диапазон данного типа данных). Тип Double используется для хранения более точно заданных данных и для получения результатов с большой (как говорят двойной) точностью, чем для данных типа *Single*.

Типы *Comp* и *Currency* применяются для бухгалтерских расчетов. В типе Comp дробная часть отсутствует, а в типе Currency она ограничена четырьмя знаками. С другой стороны, значения этих типов совместимы со значениями других вещественных типов, т.е. над ними могут выполняться все вещественные операции и им можно присваивать значения переменных и выражений других вещественных типов. Но при этом нельзя забывать о том, что будет происходить усечение значений до четырех знаков в дробной части для типа Currency и полное отбрасывание дробной части для типа Comp.

При проектировании бухгалтерских приложений для обработки денежных величин применяйте переменные типа Currency. Значение, переданное в переменную этого типа данных, физически хранится в формате Int64, при этом Delphi полагает, что четыре последних знака этого значения - знаки после запятой. Таким образом, действительное число 9,9999 обрабатывается как целое 99999, но при выводе на экран и проведении математических операций в тайне от нас Delphi делит его на 10000, тем самым соблюдая статус кво. Вся эта казуистика позволяет избежать ошибок округления, что очень нравится бухгалтерам.

Символьные типы

Символы – это фундаментальные «кирпичики», неделимые элементы исходного текста программы. С каждым таким символом в компьютере ассоциировано числовое значение ASCII-кода (ANSI -кода или Unicode-кода). Кодировка символов по таблице ASCII используется в операционной системе MS DOS, а кодировка символов по таблице ANSI

используется в операционной системе Windows. Согласно таблицам этих кодировок каждому символу сопоставлено целое число. *Кодировка* – это порядок следования (порядковый номер следования) символов в символьном наборе таблицы. При нажатии клавиши клавиатуры с символом в ячейку памяти записывается соответствующий по таблице числовой код – целое число. Если символ не закреплен ни за одной из клавиш клавиатуры, то его код можно ввести так: нажав клавиши ALT и не отпуская ее, набрать числового кода символа (по таблице кодировки) на дополнительной цифровой клавиатуре.

Символьные типы предназначены для хранения числового кода одного символа в памяти компьютера. В Delphi имеется три символьных типа (таблица 3.5).

Таблица 3.5. Символьные типы

Имя типа	Размер в байтах
ANSIChar	1
WideChar	2
Char	1

Тип *ANSIChar* представляет собой так, называемые Ansi-символы, которые используются в ОС Windows. В операционной системе Windows на экран выводятся не все символы ASCII. Поскольку консольное приложение создается в операционной системе Windows, а выполняется как программа MS DOS, то попытки вывести русские символы на экран из-за различия в кодировках ASCII и ANSI обречены на неудачу.

Тип *WideChar* предназначен для хранения так называемых Unicode-символов, которые в отличие от Ansi-символов занимают два байта. Это позволяет кодировать символы числами от 0 до 65535 и поэтому используется для представления символов всех языков мира. Первые 256 символов в стандарте Unicode совпадает с символами Ansi. Поскольку тип WideChar предназначен для использования в ОС Windows, его можно использовать при создании оконных приложений.

Основной тип одиночного символа называется ***Char***. Тип Char существует с первых версий языка Delphi (char — базовый тип Паскаля), предназначен представления восьмиразрядного набора значений ANSI (256 возможных символов) и соответствует типу AnsiChar компилятора Delphi для Win32. Такой восьмизначный двоичный числовой код одного символа позволяет в ячейке величиной байт представлять всего до 256 штук различных символов.

Для преобразования числового значения в символ и обратно используются

Chr(i) - дает символ с порядковым номером **i**;

Ord(c) – дает порядковый номер символа **c**.

Например,

```
Chr(120)  =  'x'
Ord('x')  =  120
```

Если символ не входит в допустимый перечень символов, которые можно отобразить на экране, то его записывают в формате:

#число

Размер данных любого типа можно определить с помощью стандартной функции SizeOf(). В качестве ее аргумента разрешено задавать как имя переменной определенного типа, так и само имя типа. Возвращает она число байтов, отводимых для хранения значения указанного типа.

Символьный тип еще называется *литерным* типом. Над значениями символьного типа в языке не предусмотрены какие-либо операции.

Строковые типы

При решении задач на компьютере задач самых различных классов возникает необходимость в использовании строк, представляющих собой последовательности литер (символов). Например, даже в программах решения вычислительного характера приходится использовать строки для печати заголовков или комментариев к результатам счета. Использование значений символьного типа для этих целей очень неудобно, поэтому желательно иметь инструмент для задания целых последовательностей символов. Таким инструментом в Delphi и является *строки*. Следует отметить, что кроме задач вычислительного характера, где строки играют вспомогательную роль, существует большой класс задач, в которых строки символов являются основными объектами обработки (например, задачи лексического и синтаксического анализа программы, задача трансляции и т. д.). Для использования в программе заранее известных последовательностей символов служат *строки-константы*, о которых уже говорилось во второй лекции. Напомним, что строка-константа представляет собой произвольную последовательность символов, заключенную в одиночные апострофы (одинарные кавычки), например:

```
'Язык_программирования_Delphi'
```

Если внутри строковой константы нужно использовать символ ‘ (апостроф), то она удваивается, например:

```
'Значения_об''явленных_переменных'.
```

В этих примерах вместо пробела вставлены символ подчеркивания (_) для точности их отображения.

В Delphi используются следующие строковые типы (таблица 3.6):

Таблица 3.6 Строковые типы

Тип строки	Область, отводимая для хранения строки	Есть ли нулевой символ в конце
ShortString	От 2 до 256 байт	Нет
AnsiString	От 4 байт до 2 Гбайт	Есть
String	до 256 байт / до 2 Гбайт	Нет / есть
WideString	От 4 байт до 2 Гбайт	есть

Значение типа *ShortString* – это так называемые короткие строки, длина которых не превышает 255 символов. При этом каждый символ занимает один байт, самый первый байт содержит число, указывающее длину строки (количество символов в строке). Заметим, что длина строки будет представлена в символьном коде. Каждый байт имеет свой порядковый номер. Первый байт, содержащий длину строки, имеет номер, равный 0, следующий – 1, и т. д. По номеру символа можно получить доступ к его значению. Память для короткой строки выделяется компилятором до начала выполнения программы. В таком случае говорят, память выделяется *статически*.

Строка типа *AnsiString* располагается в памяти иначе. Сама переменная типа AnsiString занимает в памяти 4 байта и является *указателем* (*ссылкой*), т.е. содержит адрес той ячейки памяти, начиная с которой будет фактически располагаться символьная строка. Выделение места в памяти происходит на этапе выполнения программы, т.е. *динамически.* Программа сама определяет необходимую длину строки по заданному количеству символов и операционная система выделяет нужный участок памяти. В конце строки размещается терминальный (завершающий) нуль – символ #0 и так называемый *счетчик ссылок*, занимающий 4 байта. Нумерация символов в строке AnsiString начинается с единицы, т.е. первый символ такой строки имеет индекс 1. Если число

ссылок на строку станет равным 0, то строка уничтожается и освобождает место в памяти. Счетчик ссылок позволяет экономить память.

Тип *String* интерпретируется компилятором по-разному, в зависимости от значения директивы компилятора $H. Если она включена - {$H+}- то тип String интерпретируется как AnsiString, если нет - {$H-}- то как ShortString. По умолчанию действует директива {$H+}.

Если в разделе *var* указано, например, String[10], независимо от директивы компилятора тип трактуется как ShortString с указанным числом (10) символов.

Тип *WideString* также представляет собой динамически размещаемые в памяти компьютера строки, длина которых ограничена только объемом свободной памяти компьютера. Однако в отличие от строки типа AnsiString каждый символ является Unicode-символом. Для Unicode-символа выделяется два или больше байта.

Над значениями строкового типа предусмотрена единственная *операция* – операция *конкатенации*, обозначаемая символом (+). Конкатенация объединяет строки в указанном порядке слева направо.

Тип Variant

Этот универсальный тип данных основном предназначен для работы с результатами заранее не известного типа. Но за универсальность приходится платить: на переменную вариантного типа дополнительно отводится еще два байта памяти (это не считая байт, необходимых для хранения обычной типизированной переменной).

Вариантный тип переменной полезен при вызове объектов, поддерживающих технологию OLE Automation, хранения значений даты/времени и создания массивов переменной длины.

Вариантный тип — это тип, допускающий хранение данных разных типов и модификацию типов непосредственно во время работы программы.

Например, опишем переменную V типа Variant:

```
var
  V : Variant;
```

Ни один из типов данных не позволит таких вольностей, как variant. К переменной типа `Variant можно` присваивать значения различного типа,

Тип хранимого в вариантной переменной значения будет определен корректно непосредственно во время работы программы — по структуре самого значения.

В состав Delphi 7 входит модуль Variants, содержащий подпрограммы для работы с данными типа variant

Ссылочные типы (Указатели)

Как известно, физически любая компьютерная переменная представляет собой не что иное, как область памяти, содержащую какие-то данные. Когда мы объявим переменную myValue : integer, в памяти компьютера для хранения значения этой переменной будет зарезервировано 4 байта согласно объявленного типа и ячейка получает имя myValue. Содержимое компьютерной переменной myValue можно просмотреть непосредственно в этой области памяти, обратившись по имени myValue. Объявив другую переменную, мы заставим операционную систему отвести под эту переменную новые свободные байты памяти. При этом значения указателей на myValue и на новую переменную будут различны. Под таким указателем понимается компьютерная переменная, содержащая *адрес области памяти. Итак,* ***указатель*** *– это переменная (ячейка), в которой хранится адрес другой переменной (адрес другой ячейки).* Например, пусть значение обычной

целой переменной, например, типа integer, равное 7, хранится в памяти начиная с 5000-го байта. Тогда значение указателя на эту обычную переменную равно не 7, а 5000, т.е. указатель обычной переменной «указывает» на саму обычную переменную - определяет адрес той ячейки, где храниться значение переменной.

Итак, в программировании указатель представляет собой переменную, значением которой является адрес начала размещения некоторых данных в основной памяти. Также говорят, указатель есть ***ссылка*** на размещения самих значений переменной. В программировании могут быть в использовании *обычные переменные* и *указательные переменные*. Если же переменная является не обычной, а указательной, то значение компьютерной указательной переменной нельзя просмотреть непосредственно в этой области памяти, как обычно делается для обычных компьютерных переменных. Если обычной переменной соответствует одна ячейка, то для указательной переменной, можно сказать, соответствует две ячейки: в одной находится адрес размещения значения. переменной, а в другой находятся сами значения переменной.

Указатели могут ссылаться па данные любого типа. Переменные типа указателя являются *динамическими,* а это означает, что размещения соответствующих ей данных в основной памяти определяются во время выполнения программы. С помощью указателей программист может создавать, так называемые *списковые структуры данных*, в памяти компьютера для хранения данных, порождаемых во время выполнения программы.

В Delphi указатели используются в любом типе данных, требующем динамического выделения больших блоков памяти. Например, длинные строки текста и объекты (т.е. экземпляры классов) являются неявными указателями. Они указывают на область памяти, где начинают располагаться связанные с ними значения. Понимание указателей и операций с указателями необходимо для работы в *среде разработки программ Delphi.*

Используя указатели, можно экономнее расходовать память, потому что с их помощью память для переменных можно выделять динамически. Однако, программист должен постоянно помнить о необходимости удаления неиспользуемых динамических объектов, иначе в программе произойдет **утечка памяти.** Интенсивная утечка памяти может вызвать аварийное завершение программы, причем обнаружить источник утечки обычно довольно трудно.

Различают указатели *типизированные* и *нетипизированные. Типизированный* указатель может ссылаться на данные определенного типа, который указывается при объявлении указателя или при описании типа указателя. При этом используется знак ^, который ставится перед именем типа адресуемых данных. Символ «крышка» (^) используется как для обозначения указателя, так и для его **разыменования**. Синтаксис определения типированного указателя имеет вид:

```
type
  имяТипаУказателя = ^ТипАдресуемыхДанных;
```

Например, оператор

```
type
  IntPointer = ^Integer;
```

Определяет тип указателя IntPointer. Переменная этого типа является указателем на целое значение, т.е. переменная типа IntPointer содержит адрес ячейки памяти, в котором хранится значение типа Integer. Неявный вариант определения этого типа: var

Нетипизированный указатель имеет тип ***Pointer*** и может ссылаться на данные любого типа..

Если определен тип указателя, то можно объявить указатель этого типа. Пример объявления переменных-указателей:

```
var
  Pi : Pointer;
  p2 : ^Integer;
```

Здесь переменная *p2* может указывать на данные типа integer, а переменная *p1* — на данные любого типа.

С помощью указателя можно получить доступ к значению адресуемых данных. Для этого служит операция *разыменовывания указателя* — справа от имени указателя приписывается знак ^. Таким образом, выражение

```
имяУказателя^
```

определяет разыменование указателя. Например, если переменная ptr имеет тип IntPointer, то выражение ptr^ возвращает целое значение, которое хранится в памяти по адресу, находящемуся в настоящее время в переменной ptr.

Для определения адреса ссылаемого с помощью указателя объекта можно использовать *операцию @*, записываемая перед именем этого объекта.

В Delphi есть зарезервированное ключевое слово nil, обозначающее *специальную константу*, значение которой можно присвоить любой указательной переменной. Если указателю присвоено значение nil, значит, указатель не ссылается ни на какой объект. Поэтому рекомендуется всегда инициализировать указатели значением nil.

Тип дата-время

Тип дата-время предназначен для одновременного хранения *даты* и *времени*. Определяется этот тип при помощи стандартного идентификатора ***TdataTime***. Значения этого типа представляют собой 8-ми байтовые вещественные числа с фиксированной точкой. Целая часть позволяет определить дату, а дробная - время. Благодаря такому подходу, над данными типа дата-время определены те же операции, что и для вещественных данных. Существуют стандартные подпрограммы для работы с датами и временем.

Процедуры ввода, вывода

В консольном приложении ввод данных с клавиатуры осуществляется при помощи стандартных процедур ***read*** и ***readln***, а вывод на экран дисплея – при помощи процедур ***write*** и ***writeln***.

Процедуры ввода: read и readln.

Общий формат этих процедур следующий:

```
Read(имя1, имя2, имя3, …);
Readln(имя1, имя2, имя3, …);
```

Например,

```
Read(x1, y2, z3);
Readln(name, age, summa);
```

При выполнении этих процедур происходит следующее. Программа приостанавливает свою работу и ждет, пока на клавиатуре будут набраны данные, соответствующие спискам имен, и нажата клавиша Enter. После нажатия клавиши Enter, введенные значения присваиваются переменным, имена которых указаны в списке.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Если в строке экрана число набранных значений больше числа, чем число имен в списке процедуры Read, то лишние будут проигнорированы или обработаны следующей процедурой ввода Read (если такая имеется в тексте программы).

В отличие от Read процедура Readln после ввода значений для всех указанных в списке имен осуществляет переход к следующей строке дисплея (т.е. курсор ввода ставится на первой позиции следующей строки).

Процедуры вывода: write и writeln

Назначение процедур – выводить информацию на экран. Общий формат этих процедур следующий:

Write(выражение1, выражение2,…);
Writeln(выражение1, выражение2,…);

Например,

```
Writeln('5=',  5,  summa, 5 + 56,  5 div 2);
```

При использовании процедуры Writeln сначала вычисляются по очереди значения *выражения* (аналог математического выражения) затем значение печатается на экране в след предыдущему значению. При печати между значениями пробелы не ставятся (так что этого нужно программисту задать самому). После окончания печати курсор устанавливается в начало следующей строки. Если вы хотите оставить курсор на той же строке после последнего значения, то используйте Write.

Для печати значения можно задать ширину полосу печати. Для этого нужно в списке после выражения через двоеточия задать *количество символов для значения* (и *количество символов для дробной части дробного значения)*.

Например,

```
Writeln(15.2,  5+56:15, summa:15:2);
```

Резюме

Данное – это код, т. е. условное представление информации. Информация – это смысл (мысль), связанный с кодом (данным). Условное представление может писаться с помощью знаков различного алфавита. Компьютер хранит в памяти и обрабатывает данные. Данные являются значениями переменных в тексте программы или содержимыми ячеек памяти. Переменная и ячейка имеют одинаковые характеристики, поэтому в программировании переменную определяют еще *как именованная область памяти для хранения данного.*

Переменная текста программы компьютером интерпретируется как ячейка памяти. Ячейка, как *компьютерная переменная*, может содержать (иметь) значения разного типа. Используемая переменная должна быть обязательно прежде объявлена с указанием типа значения переменной. Указание типа для компьютера обусловлено с тем, что компьютер числовые и символьные значения представляют по-разному в ячейках различной длины, а также обрабатывает их по-разному. Использование в языке разных типов для одного и того же вида значения позволяет экономнее расходовать память. Переменная перед использованием должна обязательно получить свое значение.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Вопросы для самопроверки

- Что такое переменная и как она декларируется?
- Что общего между переменной и ячейкой?
- Какие бывают комментарии и как они записываются?
- Что такое указатель?
- Что такое ячейка?
- Что такое адрес ячейки?
- Какая разница между значением переменной и содержимым ячейки?
- Что такое указательная переменная?
- Что означает значение указательной переменной?
- Что за тип вариант?
- Как представляются целые и вещественные числа в ячейке?
- Что означает разыменование указателя?
- Какая разница между стандартными и пользовательскими типами?
- Как обрабатываются целые и вещественные числа компьютером?
- Какая разница между символьными и строковыми типами?
- Как ввести данные в программу?
- Как вывести результаты вычислений на экран?
- Чем отличаются процедуры write и writeln?
- Чем отличаются процедуры read и readln?
- Есть ли различия в параметрах процедур ввода и вывода?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 4. Оператор присваивания. Выражения

План

Концепция действия. Выражения.
Арифметический оператор присваивания
Логический оператор присваивания
Символьный оператор присваивания
Строковый оператор присваивания
Приведение типов и функции преобразования типов

Концепция действия. Выражения

Основное назначение программы состоит в задании тех действий по обработке данных, которые должны быть выполнены для решения поставленной задачи. Напомним, что программа – это запись алгоритма на некотором языке программирования. Как известно, базовыми структурами алгоритма являются: *действие (функция), выбор (ветвление)* и *повторение.* В этой лекции рассмотрим вариант реализации структуры *действия* виде оператора на языке программирования Delphi.

Запись алгоритма на языке программирования есть инструкция для того исполнителя, который фактически будет осуществлять заданный процесс решения задачи. При этом имеется в виду такой исполнитель, который умеет выполнять только заранее фиксированный набор операций над отдельными значениями, причем набор типов таких значений также зафиксирован. Именно таким исполнителем и является компьютер.

Что касается типов значений, допустимых в Delphi, то об этом уже говорилось в предыдущей лекции. Для задания же действий над данными, которые необходимо выполнить для решения той или иной задачи, в языке программирования служит такое понятие, как *оператор*. *Каждый оператор в Delphi-программы определяет некоторый логически законченный, самостоятельный этап процесса обработки данных.* Естественно, что для однозначности понимания и интерпретации программы зафиксирован набор допустимых операторов и четко определены правила их записи, т.е. синтаксис оператора.

Процесс решения задачи распадается на ряд последовательно выполняемых этапов, на каждом из которых по некоторым значениям, известным к началу выполнения этого этапа, вычисляется новое значение. Одни из этих вычисленных значений являются окончательными результатами решения задачи, а другие являются промежуточными результатами, используемых в качестве исходных данных для некоторых из последующих этапов.

Для задания правил вычисления новых значений в Delphi служит такое понятие, как *выражение*, причем каждое выражение задает правила вычисления только одного значения. Заметим, что выражение ничего не говорит о том, что следует делать с этим значением, и потому выражение не задает какого-то логически завершенного этапа вычислений. Наиболее же типичным является ситуация, когда вычисленное значение необходимо запомнить для его использования на последующих этапах вычислительного процесса – такое запоминание достигается путем присваивания вычисленного значения некоторой *переменной*. Для задания такого действия и служит *оператор присваивания*, который относится к числу основных операторов.

Синтаксис записи оператора присваивания следующий:

переменная **:=** выражение;

или

имяФункции **:=** выражение;

Пара символов **:=**, разделяющая правую и левую части оператора, рассматривается как *оператор **присваивания***. В левой части оператора присваивания стоит *идентификатор* — имя декларированной к данному моменту переменной. В правой части записано *выражение*. Результат его вычисления *присваивается в переменную* (***записывается в ячейку***), имя которой указанно в левой части. Старое значение этой переменной (**ячейки**) теряется. *Тип значения выражения должен соответствовать типу переменной в левой части оператора*. В конце оператора присваивания ставится точка с запятой. Оператор присваивания, как и любые другие операторы языка Delphi, в тексте программы может располагаться только между ключевыми словами *begin* и *end*.

Например:

```
var
  X, Y: Integer; // мы объявляем переменные X, Y как переменные с
                 // целочисленными значениями типа Integer.
                 // По такому объявлению компилятор выделить две ячейки
                  // и дасть им соответственно имена X, Y, причем каждая
                  // ячейка будет состоят из четырех байтов (согласно Integer).
begin
  X := 1;     // к  переменной  X  будет присвоено  значение  1. Подобное
              // действие в дальнейшем понимается  как действие:
              // в ячейку с именем X  будет записано целое число 1.
  Y := 2;     //  к  переменной  Y  будет присвоено  значение  2
  X := Y + 3;
    // в переменную X будет присвоено число 5:
    // 5 - это значение выражения Y + 3  то есть  (2+3=5)
  X := X + 1;
    // значение переменной X увеличивается на 1
    // и становится равным 6:
```

При выполнении оператора присваивания сначала всегда рассчитывается значение правой части. Только потом это значение присваивается к переменной левой части как новое значение, т.е. в ячейку записывается новое значение. Вычисление значения *выражения* делается процессором и результат получается в процессоре, после этого результат записывается в указанную ячейку оперативной памяти.

В тексте программы каждой вводимой в употребление переменной предписывается вполне определенный тип значений, которые может принимать эта переменная, а попытка присвоить ей значение какого-либо другого типа расценивается как ошибка в программе. Отсюда видно, что выражение в правой части оператора присваивания не может быть произвольным оно должно задавать правила вычисления значения того же типа, что и и переменной в левой части оператора присваивания. В связи с этим можно говорить и о типе выражения, подразумевая под этим тип того же значения, правила вычисления которого задаются данным выражением. С одной стороны, выражения разных типов (в указанном выше смысле) имеют много общего: все они строятся из *операндов, знаков операций и круглых скобок*, с помощью которых можно задать *любой желаемый порядок выполнения операций*. При этом имеются тир вида операндов: постоянные, переменные и вычисляемые.

Постоянный операнд задает значение, которое известно при составлении программы и не меняется в процессе ее выполнения ,так что постоянный операнд –это константа того или иного типа.

Переменный операнд задает значение ,которое определяется и может изменяться в процессе выполнения программы. Однако к началу вычисления выражения, в котором используется переменный операнд, его значение должно быть определено. Такими операндами являются *переменные* языка Delphi. С точки зрения синтаксиса, в Delphi имеется несколько видов переменных, что обусловлено наличием различных типов значений и их спецификой. Поскольку

ока знакомы только со стандартными скалярными типами, будем считать, что синтаксич переменная- это идентификатор, который используется в качестве имени текущего значения переменной как программного объекта, способного принимать значение. В языке по имени переменной осуществляется непосредственный доступ к ее значению. По мере изучения других типов значений мы познакомимся и с другими видами переменных.

Вычисляемый операнд задает значение, которое не определено даже к началу вычисления того выражения, в котором такой операнд используется- это значение вычисляется в процессе вычисления самого выражения. Вычисляемыми операндами в Delphi являются *вызовы функций* (или, короче, *имя функции*).

С другой стороны, поскольку каждое выражение должно определять значение какого-то определенного типа, то в нем могут фигурировать операнды тоже определенных типов. Кроме того, как мы это уже знаем на примере стандартных скалярных типов, над каждым типом значений в языке определен свой набор операций. В связи с этим на данном этапе изложение Delphi довольно трудно дать единое синтаксическое определение для выражения всех допустимых в Delphi типов, которое было бы и достаточно компактным, и достаточно понятным.

Арифметический оператор присваивания

Арифметический оператор присваивания служит для присваивания значения переменной арифметического типа, т.е. целого и вещественного типов. В связи с этим и в правой части такого оператора должно фигурировать арифметическое выражение, т.е. выражение, задающее правило вычисления значения одного из этих типов.

Если переменная в левой части оператора присваивания имеет целый тип, то арифметическое выражение объязательно должно определять значение этого же типа.

Выражения, в которых операнды имеют разные числовые типы данных, называются смешанными арифметическими выражениями. Их использование в Delphi допускается. Например, следующий фрагмент кода не содержит ошибки:

```
Var
  Num1:  Real;
  Num2:  Integer;
  Result:  Real;
Begin
  Num1 := 7.2;
  Num2 := 3;
  Result := num1 * num2;  // смешанное арифметическое выражение
End;
```

В смешанных арифметических выражениях числовые типы, требующие меньшей памяти, автоматически приводятся к типам, требующим большей памяти, а целые типы приводятся к вещественным. Другими словами, преобразование целого типа в вещественный выполняется неявно.

В рассмотренном примере сначала тип num2 приводится к вещественному. Однако если попытаться перенести это выражение в другой язык, то в нем могут оказаться другие правила приведения типов, в результате чего возникнет ошибка. Поэтому рекомендуется по возможности избегать использования смешанных арифметических выражений.

В качестве операндов в арифметическом выражении используются: константа (число без знака или имя константы), переменная и функция. Напомним, что в языке предусмотрены две категории арифметических операций: *мультипликативные* (*, /, div, mod) и *аддитивные* (+, -). Операции в каждой из этих категорий имеют одинаковый ранг (старшинство), причем мультипликативные операции имеют *более высокий ранг*, чем аддитивные, т.е. выполняются в первую очередь. Операции одного и того же ранга, если они встречаются в выражении подряд, выполняются в

дке их следования слева направо. В случае необходимости желаемый порядок выполнения операций можно задать с помощью круглых скобок: *подвыражения* (части выражения), заключенные в скобки, вычисляются независимо и раньше, чем будут выполняться предшествующие и последующие операции.

В Delphi результат операции деления (/) ***всегда*** имеет тип ***Extended***, независимо от типов операндов. Для других арифметических операций справедливы следующие правила. Если хоть один операнд имеет вещественный тип, то результат выполнения операции имеет тип Extended

Результат выполнения операции над целыми операндами имеет тип Int64, если хоть один операнд имеет тип Int64; в противном случае результат имеет тип Integer. Целочисленное деление (оператор div) используется для деления двух чисел, причем возвращаемый результата содержит только целую часть отношения, дробная часть отбрасывается. Например, оператор 5 div 2 возвращает результат 2, хотя в математике это отношение равно 2,5. Другими словами, результат выполнения операции div равен меньшему по модулю ближайшему целому математического отношения операндов. Оператор деления по модулю mod возвращает целый остаток деления двух целых чисел. Например, оператор 5 mod 2 возвращает результат 1. Математически оператор *x mod y* эквивалентен операции *x-(x div y)*y* как для положительных, так и для отрицательных операндов.

Встроенные математические функции

Язык Delphi содержит многочисленные математические функции. Наиболее часто используемые функции перечислены в табл.4.1.

Таблица 4.1. Встроенные математические функции

Функция	Возвращаемый результат
Abs(x)	Модуль (абсолютное значение) числа *x*
Ceil(x)	Наименьшее целое, большее или равное *x*
Exp(x)	Вещественное значение числа *e*, возведенное в степень *x*, где *e*-основание натуральных логарифмов
Floor(x)	Наибольшее целое, меньшее или равное числу *x*
Frac(x)	Дробная часть числа *x*, имеющего тип Extended
Int(x)	Целая часть числа *x,* имеющего тип Extended
IntPower(основание, степень)	Значение типа Extended, равное основанию (типа Extended), возведенному в степень(типа Integer)
Ldexp(x)	Возвращаемый результат имеет тип Extended и равен $x*2^p$
Ln(x)	Натуральный логарифм вещественного значения *x*
LnXP1(x)	Натуральный логарифм вещественного значения (x+1)
Log10(x)	Десятичный логарифм числа x
Log2(x)	Двоичный логарифм числа x
LogN(n, x)	Логарифм по основанию *n* числа x
Max(x, y)	Большее из двух чисел
Min(x, y)	Меньшее из двух чисел
Pi( )	3,14592653589793238 5
Power(основание, степень)	Значение типа Extended, равное основанию (типа Extended), возведенному в степень(типа Extended)
Round(x)	Вещественное значение *x* округляется до ближайшего

	целого значения типа Int64. Если x находится точно посередине между двумя целыми, то результат всегда является четным числом.
Sqr(x)	Квадрат числа, т.е. x^2
Sqrt(x)	Квадратный корень числа x, т.е. $\sqrt{x}$
Trunc(x)	Отбрасывание дробной части числа x

В отличие от многих компьютерных языков, в Delphi для вычисления степеней используются функции, а не операторы. В других языках для возведения в степень используются операторы ** или ^, однако в Delphi символ ^ зарезервирован для указателей (см. лекцию 2.).

Логический оператор присваивания

Если в левой части оператора присваивания указана переменная типа ***boolean***, то в правой части оператора должно быть задано логическое выражение, задающее правило вычисления логического значения (true или false).

В логическом выражении используются те же виды операндов, что и в арифметическом выражении (константы, переменные и функции), только каждый операнд логической операции должен иметь тип boolean. Специфическим видом операнда какой-либо логической операции в логическом выражении является *отношение.* Два арифметического выражения, соединенные операциями сравнения образуют отношения. *Операциями сравнения* задаются знаками: **<, <=, =, <>, >.**

Отношение имеет значение true, если заданное в нем с помощью операции сравнения соотношение между значениями арифметических выражений действительно имеет место, и значение false - в противном случае. Например, отношение 3 < 5 имеет значение true, а отношение 3 >= 5 – значение false.

В отношении арифметические выражения могут быть как вещественными, так и целочисленными – предполагается, что при сравнении целого числа с вещественным оно предварительно преобразуется в вещественное число.

Примеры логических выражений (d, b, c - логические, x, y – вещественные, k – целочисленная переменные):

```
X<2*y          (отношение)
True           (константа)
D              (переменная)
Odd(k)         (функция)
Not not d
(x>y/2)
d and (x<>y) and b
(c or d) and (x=y) or not b
```

Итак, в логическом выражении из-за наличия в нем отношений могут присутствовать как арифметическое, так и логические операции. При этом самой старшей операцией является операции not, применимая к логическому операнду, затем следуют мультипликативные операции (*, /, div, mod, and), потом – аддитивные операции (+, -, or), и самый низкий приоритет имеют операции сравнения (т.е. они выполняются в последнюю очередь). Операции одинакового старшинства выполняются в порядке их следования в выражении слева направо. Для задания любого желаемого порядка выполнения операций, как обычно, используются круглые скобки.

Примеры логических операторов присваивания:

```
D := true;
B := (x>y) and b;
C := d or b and not (odd(k) and d);
```

Символьный оператор присваивания

Если в левой части оператора присваивания указана переменная типа ***char***, то в правой его части должно быть задано символьное выражение, задающее правило определения значения типа char, т.е. отдельного символа. Как мы уже знаем из рассмотрения типа char, над значениями этого типа не определены какие-либо операции, результатом выполнения которых является значение типа char. Поэтому символьным выражением может быть только константа, переменная или функция этого типа.

Примеры символьных оператор присваивания (sym, alpha, beta – переменные типа char)

```
sym := '+';
alpha := sym;
beta := succ(sym);
```

Строковый оператор присваивания

Как вы помните, строка – это группа символов. Строковые константы обозначаются апострофами. В следующей строке кода переменной today присваивается значение Четверг:

```
today := 'Четверг';
```

Операция ***конкатенации***, обозначаемая символом +, - единственная строковая операция Delphi. Конкатенация объединяет строки в указанном порядке слева направо. Таким образом, оператор + может выполнять различные операции в зависимости от типа операндов. Такие операторы называются перегружаемыми. В следующем фрагменте кода выполняется конкатенация строк:

```
WeekendDays  := 'Суббота'+' и '+'Воскресенье';
```

После выполнения этого кода переменная weekenDays содержит строку *Суббота и Воскресенье.*

В предыдущем примере, как вы заметили, апострофы не являются частью строки. Но что делать, если нужно использовать строку, содержащую апострофы? Следующий фрагмент кода является *неправильным*:

```
WeekendDays := ''Суббота и Воскресенье'';
```

Обрабатывая эту строку кода, компилятор сгенерирует сообщение о синтаксической ошибке. Компилятор Delphi не сможет понять структуру этого оператора . Он видит, что за первым апострофом непосредственно следует второй, и считает это пустой строкой, т.е. строкой, не содержащей символов. Затем оператор встречает слово *Суббота* и не знает, как его понять, поскольку строка уже закончилась. К счастью, существует несколько способов включения апострофа в состав строки. Внутри строки компилятор считает последовательность из двух апострофов не признаком окончания строки, а одним апострофом, входящим в состав строки. В следующем фрагменте кода переменной weekendDays присваивается значение *'Суббота и Воскресенье'*.

```
WeekendDays := '''Суббота и Воскресенье''';
```

Первый и последний апострофы обозначают конец строки. Каждая пара апострофов внутри строки рассматривается компилятором как один апостроф.

Другой способ включения апострофов в состав строки – использование числовых значений символов. В компьютере с каждым символом ассоциировано числовое значение ASCII (American Standard Code for Information Interchange-Американский стандартный код для обмена информацией). В операционной системе Windows на экран выводятся не все символы ASCII, в ней используется более ограниченный набор символов ANSI (American National Standard Institute-

ональный институт стандартизации США). Числовые значения большинства символов ASCII и ANSI совпадают.

Для работы с символами ASCII в Delphi есть две встроенные функции.

Chr(n) - возвращает символ, числовое значение ASCII которого равно *n*

Ord (символ) – возвращает числовое значение ASCII для символа *символ*

Например, Chr (65) возвращает символ A, а Ord (‘A’) возвращает значение 65. Таким образом, числовое значение символа A равно 65. Список числовых значений символов ASCII (ANSI) можете посмотреть по справочной литературе.

Числовое значение ASCII символа “апостроф” равно 39, поэтому строку предыдущего примера можно записать в виде

```
WeekendDays := Chr (39) + `Суббота и Воскресенье ` + Chr (39);
```

Этот оператор присваивания эквивалентен предыдущему: он присваивает переменной WeekendDays значение ‘Суббота и Воскресенье’.

Кроме функции Chr( ) и операции конкатенации, для включения в строку символа в Delphi можно воспользоваться управляющей последовательностью. Она состоит из символа #, за которым следует целое число в диапазоне от 0 до 255, обозначающее символ ASCII. Например, оператор

```
MyString :=  #72#101#108#108#111;
```

эквивалентен оператору

```
MyString := ’Hello’;
```

Еще один пример: оператор

```
WeekendDays := #39`Суббота и Воскресенье`#39;
```

эквивалентен и оператор

```
WeekendDays := `’’Суббота и Воскресенье`’’;
```

и оператору

```
WeekendDays := Chr (39) +’Суббота и Воскресенье` + Chr (39) ;
```

Обратите внимание, что управляющая последовательность в составе сроки не работает. Она управляет только работой компилятора. Например, если переменную `myString := ’Hel #10o’` вывести на экран, то мы увидим `Heloo`, а `Hel#108o`.

Полезный совет

Чтобы включить в строку символ, которого нет на клавиатуре, нужно сначала найти его числовое значение в таблице ASCII, а затем воспользоваться оператором конкатенации (+) и функцией Chr ( ).

Язык Delphi поддерживает также типы данных и функции, необходимые для работы с набором символов Unicode, в котором каждый символ представлен двумя байтами. Строка Unicode состоит из последовательности двухбайтовых символов. Преимуществом набора Unicode является то , что он содержит символы всех языков мира. Это существенно облегчает создание приложений, которые будут использоваться в других странах. Символы и строки Unicode иногда называют широкими символами и широкими строками. Первые 256 символов Unicode совпадают с символами набора ANSI. Для излагаемого в книге материала набора ANSI достаточно, поэтому символы Unicode в ней не рассматриваются и не используются. При необходимости вы можете обратиться к справочной системе Delphi, в которой представлена полная информация о символах Unicode.

Для обработки строк Delphi предоставляет довольно много встроенных процедур, благодаря которым он является одним из лучших языков для работы со строками. В таб. 4.2. перечислены некоторые наиболее полезные встроенные строковые функции.

Таблица 4.2. Встроенные строковые функции

Функция	Назначение
CompareStr( )	Сравнивает две строки с учетом регистра
CompareText( )	Сравнивает две строки по числовым значениям символов без учета регистра
Concat( )	Выполняет конкатенацию (двух или более) строк в одну
Copy( )	Возвращает подстроку заданной строки
IsDelimiter( )	Определяет, является ли указанный символ строки символом-разделителем
LastDelimiter( )	Возвращает позицию последнего символа-разделителя в строке
Length( )	Возвращает количество символов в строке
LowerCase( )	Возвращает копию строки, заменив все буквы верхнего регистра соответствующими буквами нижнего регистра
Pos(подСтрока, Строка)	Возвращает позицию первого вхож-дения первого символа указанной подстроки, содержащиеся в заданной строке
QuotedStr( )	Возвращает строку, выделенную апострофами (т.е. добавляет апострофы в конец и в начало строки)
StringOfChar( )	Возвращает строку с указанным количеством повторяющихся символов
StringReplase( )	Возвращает строку с заменой всех вхождений одной подстройки другой подстройкой
Trim( )	Удаляет из строки пробелы, расположенные в её начале и в конце, и управляющие символы
TrimLeft( )	Удаляет из строки пробелы, расположенные в ее начале, и управляющие символы
TrimRight( )	Удаляет из строки пробелы, расположенные в её конце, и управляющие символы
UpperCase( )	Возвращает копию строки с заменой всех букв нижнего регистра соответствующими буквами верхнего регистра
WrapText	Разбивает строку на несколько строк, имеющих указанный размер
Delete( )	Удаляет подстроку из строки
Insert(подСтрока, Строка, Индекс)	Вставляет подстроку в строку, начиная с указанной позиции
SetLength( )	Устанавливает длину строки
SetString( )	Устанавливает содержимое и длину строки
Str( )	Преобразует числовую переменную в строку
Val( )	Преобразует строку в числовую переменную

Как видно из табл.4.2. функция QuotedStr( ) предоставляет еще один способ размещения апострофов вокруг строки. С её помощью фрагмент кода, присваивающий строке weekendDays значение ‘Суббота и воскресенье‘ можно записать так:

```
...kendDays := QuotedStr('Суббота и воскресенье');
```

Заметим, что в языке паскаль, вообще говоря, нет таких понятий, как арифметическое выражение, логическое выражение и т.д. Мы ввели эти дополнительные понятия из методических соображений.

Приведение типов и функции преобразования типов

Delphi является сильно типизированным языком . Это означает, что типы данных строго различаются и при вычислении выражении строго выполняются определенные правила и ограничения на использование различных типов. Преимущество сильной типизации состоит в том, что транслятор получает возможность правильно обрабатывать данные и тщательно проверять исходный код, исключая таким образом трудно диагностируемые ошибки времени выполнения. Чтобы лучше понять принципы сильной типизации, рассмотрим следующий фрагмент кода:

```
Var
    myChar: Char;
    myByte: Byte;
begin
    myChar := ' A ' ;
    myByte :=  myChar;    // строка с синтаксической ошибкой
end;
```

Переменные обоих типов (Char и Byte) для хранения своих значений занимают по одному байту памяти. Компилятор генерирует сообщение об ошибке потому, что, согласно правилам сильной типизации Delphi, нельзя присваивать значение типа Char переменной типа *Byte* и наоборот.

Однако иногда в программе необходимо выполнить операцию, запрещенную правилами сильной типизации, как в предыдущем примере. В этом случае обойти правила сильной типизации можно с помощью механизма приведения типов. ***Приведением типов*** называется преобразование типов промежуточных результатов вычислений. Обратите внимание, что преобразуются только типы промежуточных результатов, типы объявленных переменных всегда остаются неизменными. Общий синтаксис типов имеет вид

Тип_данных(выражение);

Например, оператор Integer('A') приводит тип символа *A* к целому. Значение любой переменной можно привести к любому типу при условии, что исходный и результирующий типы занимают одинаковые объемы памяти и целые и вещественные числа не смешиваются. Для преобразования числовых типов используются встроенные функции ***Int()*** и ***Trunc().***

Теперь можно исправить наш предыдущий пример так: `myByte := Byte(myChar);`

Операции приведения типов не преобразует типы значений между целыми и вещественными числами и строками. Однако это можно сделать с помощью встроен-ных процедур и функций преобразования типов Delphi, перечисленных в табл. 4.3.

Таблица 4.3. Процедуры и функции преобразования типов

Процедура или функция	Назначение
CompToCurrency()	Преобразует значение Comp в значение Currency
CompToDouble()	Преобразует значение Comp в значение Double
CurrencyToComp()	Преобразует значение Currency в значение Comp
CurrToStr()	Преобразует значение Currency в строку

CurrToStrF()	Преобразует значение Currency в строку в заданном формате
DoubleToComp()	Преобразует значение Double в значение Comp
Int()	Возвращает целую часть вещественного значения
IntToStr()	Преобразует значение Integer в строку
Round()	Округляет вещественное значение до ближайшего целого
Str()	Преобразует числовое значение в строку
StrToCurr()	Преобразует строку в значение Currency
StrToInt()	Преобразует строку, в которой представлено целое значение (в десятичной или шестнадцатеричной форме), в целое значение
StrToInt64()	Преобразует строку, в которой представлено целое значение (в десятичной или шестнадцатеричной форме), в целое значение типа Int64
Trunc()	Усекает вещественное значение до целого (отбрасывает дробную часть)
Val()	Преобразует строку в числовое значение

Резюме

Часто выполняемым действием компьютера является выполнение операций формулы вычисления над операндами, значения которых хранятся в ячейках памяти и запись результата вычисления в соответствующую ячейку памяти. В тексте программы подобные действия задаются операторами присваивания. Содержимые ячейки для выполнения операции над операндами переписываются в соответствующие регистры процессора и процессор выполнив указанную операцию, получает результат в соответствующем регистре, а затем полученный результат перезаписывает в указанную ячейку памяти.

Вопросы для самопроверки

1. Какие операции допустимы с целыми и дробными значениями?
2. Какие операции допустимы с логическими значениями?
3. Что такое приоритет операции и как его можно изменить?
4. Что такое переменная и как она декларируется?
5. Какие бывают комментарии и как они записываются?
6. Как записывается оператор присваивания и как он работает, выполняется?
7. Зачем делается приведение типов?
8. Знаешь формулу приведения типа?
9. Различие между явным и неявным приведением типов.
10. Что такое выражение?
11. Всегда ли компьютер дает точный результат?
12. Почему компьютер делает вычисление с ошибками?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 5. Управление последовательностью действий

План

Алгоритм с ветвлениями
Составной оператор begin-end
Условный оператор if-then
Условный оператор if-then-else
Вложенные условные операторы
Оператор выбора case
Оператор перехода

Алгоритм с ветвлениями

Как известно, базовыми структурами алгоритма являются: *действие (функция), выбор (ветвление)* и *повторение.* В этой лекции рассмотрим, как *структура выбор* реализовывается в виде операторов на языке программирования Delphi.

Как известно, компьютер обрабатывает *исходный код* (т.е. предложения текста программы) строго последовательно, от первого до последовательного. Однако во многих случаях в программе при выполнении операторов должны быть предусмотрены переходы к выполнению других фрагментов операторов при наступлении некоторых условий. Чтобы это было возможным, в программах применяются специальные конструкции, которые называются *структурами принятия решений*. Так что, для программы со сложной логикой нужны *механизмы ветвления* — управления последовательностью выполнения в зависимости от определенных *условий*. Для создания эффективного и компактного кода программы нужны еще *средства многократного повторения групп команд (операторов)* при определенных *условиях*. В любой структуре принятия решений используется *сравнение значений* двух или более выражений для задания *условия*.

Составной оператор begin-end

В программе обработки данных сложной логикой нередко требуется поместить несколько операторов в такое место программы, где синтаксис языка допускает наличие **только одного**. В таких случаях используют **составные операторы**.

Составной оператор начинается с ключевого слова *begin*, а завершается ключевым словом *end*, за которым обычно ставится точка с запятой. В предшествующем операторе точку с запятой перед словом *end* указывать не обязательно.

```
begin
  Оператор1;
  ..................
  ОператорN;
end;
```

Составной оператор удобен тем, что позволяет компоновать в одну группу набор из нескольких команд (операторов) и обращаться к такому набору как *к* одному оператору. Например, запись составного оператора по правилам структурного программирования (содержимое записывается с отступом):

```
begin
  N := 50;
  MyVal := N div 10
end;
```

или

```
  N := 50
end;
```

Допускается также пустой составной оператор:

```
begin
end;
```

Пустой оператор применяется для наглядности или как пустая заготовка для будущего кода.

Составные операторы могут быть вложены друг в друга. Например,

```
begin
  N := 50;
  begin
    MyVal := N div 10;
  end;
  N := MyVal
end;
```

Структура *выбор (ветвление)* реализовывается на языке программирования Delphi в виде *условных операторов.* В Delphi есть два условных оператора: *if* и *case*.

Условный оператор if-then

Условный оператор *if* (если) очень похож на сослагательное наклонение в естественном языке. Синтаксис простой (сокращенной) формы оператора *if*, называемый *if-then*, имеет вид

```
if условие then  оператор;
```

Здесь *условие* является логическим выражением. Если значение выражения *условие* равно *True,* то выполняется оператор, стоящий после *then, в* противном случае этот оператор не выполняется и управление передается на следующий за *if* оператор. В качестве оператора, идущего за ключевым словом then и выполняемого зависимости от условия, может использоваться любая команда или оператор Delphi.

По синтаксису после *then* на место *оператор* должен стоять только один оператор языка. Поэтому оператор *if* рекомендуется использовать совместно с составным оператором *begin-end.*

Например, в следующей структурированной форме записи:

```
if условие then begin
                    операторы
                end;
```

В этом случае между *begin* и *end* можно поставить много операторов. Такая форма записи позволяет избежать многих проблем. Например, если к одному оператору, выполняемому по условию if, нужно добавить еще один, а блок *begin-end* нет, то программист может не заметить, что он добавляет новый оператор «мимо» оператора if. Если же блок *begin-end* есть, то не заметить это трудно.

Итак, ***условный оператор*** *— это оператор, который выполняет* ***команду*** *или* ***группу команд*** *в зависимости от определенного* ***условия***. ***Условие*** *в условном операторе представляется* ***выражением логического типа.***

например:

```
if x = 0 then
  WriteLn('икс равен нулю!');
if(I > 0) and (I < 12)then begin
                                    aI := 5;
                                   I := I + 1;
                                 end;
```

Условный оператор if-then-else

Этот вариант условного оператора (расширенный) имеет следующий синтаксис:

```
if условие then
begin
  операторы-1
end
              else
begin
  операторы-2
end;
```

Группа операторов-1 выполняется, только если выражение условие имеет значение ***true***. Операторы операторы-2 выполняются, если выражение условие имеет значение ***false***

Например:

```
if x = 0 then
begin
   WriteLn('икс  равен  нулю!')
end
       else
begin
    WriteLn('икс не равен нулю!');
end;
```

или

```
if (I > 0) and (I < 12) then
begin
  aI := 5;
  I := I + 1;
end
                       else
begin
  aI : = 0;
  I := I - 1;
end;
```

Вложенные условные операторы

В программировании часто используются вложенные операторы **if**, т.е. расположенные внутри других операторов if, например:

```
if условие-1 then begin
                    оператор-1
                  end
          else if условие-2  then begin
                                    оператор-2
                                  end
  …
  …
  …
                                  else if условие-n then begin
                                                           оператор-N
                                                         end
                                                 else begin
                                                        оператор-X
                                                      end;
```

В таком варианте оператор if может содержат произвольное количество предложений else if и только одно предложение else. Группа операторов операторы-1 выполняется, когда значение выражения условие-1 равно true; операторы-2 выполняются, когда условие-1 ложно, а условие-2 истинно. И наконец, операторы-N выполняются, когда все предыдущие условия ложны, а условие-N истинно. Операторы операторы-X выполняются, только если все условия (от условие-1 до условие-N) ложны.

Вариант этого оператора if-then-else без предложения else:

```
if условие-1 then begin
                    оператор-1
                  end
          else if условие-2  then begin
                                    оператор-2
                                   end
  …
  …
  …
                                  else if условие-n then begin
                                                           оператор-N
                                                         end;
```

Операторы if, содержащие предложения else-if , могут быть записаны как последовательные операторы вида if-then. В этом случае последовательность условий не менее важна и тоже должна быть записана логически правильно. Предыдущий пример кода можно переписать с использованием только последовательных операторов if-then, без применения операторов else-if.

```
if условие-1 then begin
                    оператор-1
                  end;
 if условие-2  then begin
                     оператор-2
                    end;
  …
  …

  …
 if условие-N then begin
                    оператор-N
                  end;
```

любом операторе if- then-else выполняется не более одного блока операторов. Выполняется первого встретившегося оператора if, условие которого истинно. Поэтому последовательность расположения операторов if весьма существенна и должна соответствовать логике алгоритма.

Рассмотрим пример

```
if условие-1 then begin
    if условие-2  then begin
                        оператор-1
                       end
                  else begin
                         оператор-2
                        end;
                 end;
```

В каком случае выполняется оператор-2? В языке Delphi действует ***правило:*** *команда, указанная после слова else, относится к ближайшему ключевому слову if*. В данном случае слово *else* относится к *условию-2*, и *оператор-2* выполнится, если *условие -1* истинно, а *условие-2* ложно. Если же надо связать *оператор-2* с *условием-1*, то вложенный условный оператор надо заключить в логические скобки, т.е. заключить в составной оператор, тем самым «закрыть» для предложения *else* второй оператор *if*:

```
if условие-1 then begin
  if условие-2  then begin
                      оператор-1
                     end
                end
              else begin
                     оператор-2
                    end;
```

При написании исходного кода с вложенными структурами для удобочитаемости кода вложенные операторы выделяются отступами.

Оператор выбора case

Другая структура принятия решений в Delphi – оператор ***case.*** Он наглядно представляет процесс выбора из множества альтернатив (условный оператор предлагает на выбор *не* более двух возможностей). Каждый оператор *case* можно заменить эквивалентным ему оператором *if* , однако обратное неверно – не всякий оператор if можно заменить эквивалентным case. Такое отличие обусловлено тем, что *операторы case работают только с порядковыми типами данных.* Тем не менее оператор case используется довольно часто и поддерживается почти во всех языках высокого уровня.

Оператор выбора *— это оператор, выполняющий одну команду из заданного набора в зависимости от значения выражения.*

Общий *синтаксис* оператор *case* имеет вид (сокращенный):

```
case выражение of
  список_выражений_1:  оператор_1;
  список_выражений_2:  оператор_2;
  список_выражений_3:  оператор_3;
  ..........................................................
end;
```

или (расширенный)

```
case выражение of
 список_выражений_1: оператор_1;
 список_выражений_2: оператор_2;
 список_выражений_3: оператор_3;
 ..........................................
else
 оператор
end;
```

Выражение после слова case должно быть выражением ***порядкового типа***, т.е. типа Integer, Char, Boolean и др. Кроме того, каждое выражение в списках выражений должно быть ***порядковым и вычисляемым*** во время компиляции. Например, в списках допустимы выражения 12, True, 4 – 9*5, Integer('Z'). Переменные и вызовы большинства функций в списках выражений недопустимы. Список выражений может также содержать поддиапазоны.

Выполнение. Сначала вычисляется значение **выражения,** расположенного после ключевого слова *case*. Далее выполняется тот из последующих операторов, в списке значений которого указана величина, равная вычисленному значению. Если подходящего значения не найдено, то выполняется финальная команда после ключевого слова else. Например:

```
x := 2;
case x + 2 of
    1: y := 3;
    4: y := 0;
   -2: y := 10;
end;
```

Сначала в этом операторе рассчитывается выражение x+2 (его значение равно 4). Затем внутри блока case начинается последовательное сравнение этого значения со значениями 1**,** 4 и -2. Совпадение обнаруживается на втором шаге — и в результате выполняется оператор y := 0. Далее управление сразу передается команда следующей за ключевым словом end. Все остальные варианты выбора не анализируются и пропускаются.

В операторе выбора выполняется только один (или вообще ни одного) оператор из списка команд внутри блока case. Если в зависимости от условия на выполнить группу команд, то ее надо заключить в скобки begin и end (представить в виде составного оператора).

В операторе выбора можно задавать непрерывные диапазоны проверочных значений, записывая только начальное и конечное значения этого диапазона через две точки (конечное значение должно быть больше или равным начальному). Это позволяет записывать программу компактнее. Вручную ввести список из, например, 100 значений от 1 до 100 сложно, но можно записать этот диапазон в компактном виде 1..100:

```
case x+2 of
  1..100, 501        : y := 3;
  400..500, 222+300: y := 0;
  -2..0              : y := 10;
end;
```

Оператор перехода

Инструкция перехода предназначена для изменения обычного порядка выполнения инструкций программы. Она используется в случаях, когда после выполнения некоторой инструкции требуется выполнить не следующую по порядку, а какую-либо другую инструкцию. При этом для осуществления перехода инструкция, которой передается

вление, должна быть помечена *меткой*. Метка, стоящая перед инструкцией, отделяется от нее двоеточием.

Напомним, что меткой может быть идентификатор или целое число без знака в диапазоне 0...9999, а все метки должны быть предварительно объявлены в разделе объявления меток того блока процедуры, функции или программы, в котором эти метки используются. Метка представляет собой идентификатор, описанный в разделе описаний следующим образом:

Label список_меток;

В списке меток через запятую представлены идентификаторы, объявляемые как метки.

Формат инструкции перехода:

goto <метка>;

Пример использования инструкции перехода:

```
Label ml;
begin
  ..........
  goto ml;
  ..........
  ml:  Оператор;
  ..........
end;
```

Передавать управление с помощью инструкции перехода можно инструкциям, расположенным в тексте программы выше или ниже инструкции перехода. Запрещается передавать управление инструкциям, находящимся внутри структурированных инструкций, а также инструкциям, находящимся в других блоках (процедурах, функциях).

Широкое использование оператора goto может породить неструктурированную программу. При структурном программировании *исключают* использование оператора goto.

Резюме

Любая вычислительная задача может быть решена путем выполнения некоторой совокупности операций в определенном порядке. Процедура решения задачи в *терминах операций*, которые должны быть выполнены, и *последовательность*, в которой эти операции должны выполняться, называется *алгоритмом*. Обычно операторы программы выполняются один за другим в том порядке, в котором они записаны в программе. Это называется *последовательным выполнением*. Так сказать, последовательная структура управления порядка выполнения операторов встроена в принцип работы компьютера и в язык программирования. Определение порядка, в котором должны выполняться операторы в компьютерной программе, называется *программным управлением*.

В этой лекции мы познакомились с определением порядка с помощью структуры выбора, которая реализуется в языке Dtlphi одним из трех способов:

- Структура *if-then* (одиночный выбор)
- Структура *if-then-else* (выбор из двух)
- Структура *case* (выбор из многих)

самом деле, несложно показать, что одной структуры if-then достаточно для организа программе любой разновидности выбора – все, что может быть сделано структурами if-then-else и case, может быть реализовано при помощи нескольких структур if-then (хотя выглядеть это будет не столь изящно). В тексте программы операторы, управляющие действиями должны писаться с выделением их структур для лучшего их понимания. В лекции предложен один из вариантов структурной записи.

Существует набор *стандартов программирования*, т.е. набор правил, определяющих стиль программирования. *Стиль программирования* – это способ выбора программистом определенных конструкций и программных решений из набора возможных; способ использования программистом в исходном коде *отступов*, *пробелов* и *комментариев* (элементов структурной записи).

Вопросы самопроверки

1. Какие операторы ветвления вы знаете?
2. Как записывается составной оператор?
3. Какие базовые структуры алгоритма вы знаете?
4. Какими способами могут комбинироваться базовые структуры?
5. Как работает сокращенный условный оператор?
6. Как работает расширенный условный оператор?
7. Понятно ли как выполняются вложенные условные операторы?
8. Знаете ли форму записи условного оператора по правилам структурного программирования?
9. Можете ли структурно записывать оператор выбора?
10. Как работает оператор выбора?
11. Укажите недостатки использования оператора перехода goto.
12. Укажите достоинства структурного кодирования.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 6. Цикл. Структура цикла и операторы цикла

План
Цикл.
Структура цикла.
Разработка циклического алгоритма
Оператор цикла for
Оператор цикла while
Оператор цикла repeat
Управление работой циклов
Вложенные циклы

В этой лекции рассмотрим, как из базовых структур алгоритма: *действия (функций), выбора* и *повторения,* именно, как *структура повторение* реализовывается в виде операторов на языке программирования Delphi.

Цикл

Повторяющиеся действия алгоритма на языке программирования Delphi описывается с помощью операторов цикла. *Оператор цикла* - это оператор, организующий многократное выполнение определенной команды или группы команд. Многократно выполняемую команду или группу команд в операторе цикла называют *телом цикла.*

Программа, содержащая оператор цикла, называется *циклической программой*. Как правило, программа для компьютера всегда является циклической. В языке Delphi программисту предоставлены три вида операторов цикла: ***for, while*** и ***repeat***.

Структура цикла

Цикл состоит из четырех характерных видов работ:

4. Блок действий подготовительных работ для организации повторений;
5. Блок проверки условия продолжения (или завершения) повторения;
6. Блок задания самих повторяемых действий;
7. Блок подготовки к очередному выполнению повторяемых действий.

Итак, блок-схема структуры цикла выглядит как в рис 1.

В блоке подготовительных работ, например, задаются действия по присваиванию начальных значений переменным, которые участвуют в повторяемых действиях. Кратко это называется **инициализации переменных**. Второй и четвертый блоки могут быть переставлены местами.

Разработка циклического алгоритма

1. **Постановка задачи.**
Даны несколько целых чисел. Составить алгоритм нахождения суммы этих чисел

2. **Решение**

Постановка задачи задана в, так называемой, содержательной форме, т.е. словесной форме.

Как известно, машинная программа должна быть циклической. Для решения данной задачи тоже нужно составить циклическую программу.

Мы знаем, что цикл состоит из четырех характерных работ и потому можем нарисовать блок-схему шаблона цикла и его надо заполнить.

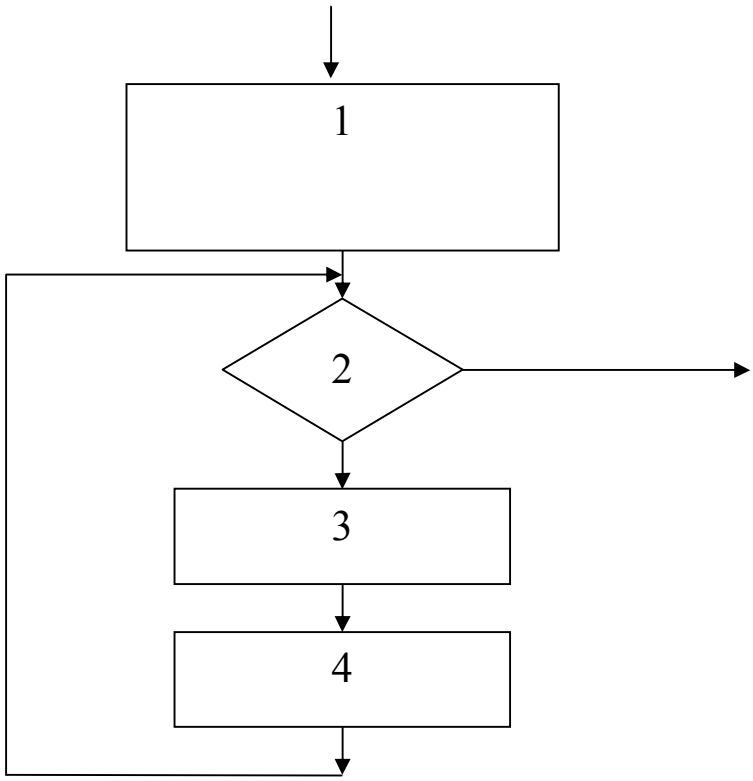


Рис 1. Блок-схема шаблона цикла

В блоке 3 задаются повторяющиеся действия решения данной задачи. Значит, нам надо анализируя текст данной задачи найти эти действия и выразит их кратко формулами.

Как известно, любая задача состоит из двух частей: условия и требования.

На практике это схематически кратко пишется так:

Дано: ……..

Найти: …….

Чтобы так написать, надо тщательно изучить условие и требование задачи и понять их. В результате этого выделяются объекты задачи. Объекты – это данные, которые обрабатываются при решении задачи, и результаты такой обработки. Выделенным анализом объектам надо дать имена (ввести обозначения).

Анализ данной задачи выделяет следующие объекты:

несколько чисел;

сами числа;

их сумма.

Дадим этим объектам соответственно следующие имена:

n;

$x_1, x_2, \ldots, x_n$;

S;

Таким образом, условие и требование данной задачи кратко схематически выражается так:

Дано: n , $x_1, x_2, \ldots, x_n$;

Найти: $S = x_1 + x_2 + \ldots + x_n$

В результате таких работ мы получаем формализованную постановку задачи:

Даны числа n и $x_1, x_2, \ldots, x_n$. Найти $S = x_1 + x_2 + \ldots + x_n$

рмализованной форме постановки задачи в виде формулы часто могут задаваться пра (т.е. алгоритм) нахождения результата решения задачи. В нашем случае так и есть.

Итак, оказывается, что решение данной задачи или алгоритм его решение задается следующей формулой $S = x_1 + x_2 + \ldots + x_n$, но этой формулой не задается кратко повторяющиеся действия процесса решения задачи. Так что нам надо их найти и выразить их кратко формулами..

Для этого изучим процесс выполнения суммирования и найдем там повторяющихся действий. Изучение дает, что:

1. к значению переменной x_1 (кратко к x_1) добавить значение переменной x_2 (кратко x_2) и принят их сумму за искомое значение переменной S (кратко S).
2. к найденному значению S (кратко к S) добавить x_3 и принят их сумму за новое искомое значение S.
3. к S добавить x_4 и принят их сумму за новое искомое значение S

……..……

……

n-1. к S добавить x_n и принят их сумму за конечное значение S.

Чтобы повторение было ровно n раз, добавим одно действие: «за начальное значение S принят x_1». Тогда первое действие перепишем в виде: «к $S + x_2$ и принят их сумму за новое искомое значение S» и сделаем его вторым действием. В качестве первого действия возьмем действие: «за значение S принять значение x_1».

Тогда эти n раз повторяющиеся действия можно кратко записать в виде одной формулы. Для этого применяется абстракция обобщения. При обобщении:
сохраняем то что является общим (а именно, S, x);
отвлекаемся от различных обозначений (т.е. от индексов) и задаем им общее имя (например, k). Тогда новая переменная k будет иметь значения: 1, 2, 3, …, n.

Эти действия позволяют выше перечисленные действия записать кратко в виде следующей формулы, а именно:
$S := S + x_k$, где k будет иметь значения: 1, 2, 3, …, n при условии, что будет предпослана действие $S := 0$.

Такая формула, где от знака присваивания в обоих сторонах стоят имя одной и той же переменной, принято называть реккурентной или рекурсивной формулой. Одной простой рекурсивной формулой можно описывать кратко многократно повторяющиеся действия. Этот факт, наверно, подчеркивает то, что «истина, понятая правильно, может выразиться кратко».

Теперь многократно вычисляющаяся рекурсивная формула вставляется в третий блок шаблона, а действие S := 0 (подготовительная работа для этого действия) нужно поставить в первый блок шаблона.

Здесь переменная k введена искусственно и значение, которой меняется, от 1 до n может, использоваться в программе как параметр цикла и тогда условие завершение цикла будет, следующий k меняется, от 1 до n (кратко k <=n). Оно задается во втором блоке. Каждое увеличение k при повторении задается кратко, как k := k + 1 и оно задается в 4-м блоке.

Таким образом, составлена блок-схема циклического алгоритма решения данной задачи. Теперь этот алгоритм можно переписать в виде программы, используя пока уже изученные операторы.

```
Program Pcikl;
   {$APPTYPE CONSOLE}
uses
    SysUtils;
label mn, mk;
```

```
  k, s, x, n: Integer;          // объявление переменных
begin
  s := 0;  k := 1;  readln(n);   //  инициализация переменной
  mn:if k > n   then  goto mk;
  readln(x);
  s := s + x;
 k := k+1;
 goto mn;
 mk:writeln(s);
 readln
end.
```

Оператор цикла for

Цикл ***for*** называется детерминированным. Это означает, что он может быть использован, только если количество итераций (повторений) можно определить до начала выполнения.

Синтаксис ***инкрементного*** цикла **for** имеет вид

```
for переменная_счетчик : = начало to конец do begin
                                                операторы   // тело цикла
                                                 end;
```

В операторе цикла задействуется специальная переменная, которая хранит текущее число выполнения тела цикла. Она называется счетчик цикла, и просто *счетчик*. Переменная, выступающая и качестве счетчика, должна быть локальной переменной любого порядкового типа. Выражение *начало* и *конец* определяют первое и последнее значения переменной счетчик. Ее значение не может изменяться внутри тела цикла. Тело цикла *один оператор* или *составной оператор*.

Выполнение. Когда в ходе работы программы встречается этот оператор, **счетчик** принимает начальное значение. Начальное значение записывается в заголовке оператора цикл в виде константы или выражения, тип которого совпадает с типом переменной-счетчика. Далее тело цикла выполняется, после чего значение счетчика увеличивается на одну порядковую позицию. Так, если счетчик имеет целый тип, то его значение просто увеличивается на единицу. Затем управление передается в заголовок цикла и значение счетчик сравнивается с выражением конец. Если значение переменной счетчик меньше выражения конец или равно ему, то тело цикла выполняется повторно. Когда значение счетчика превысит конечное значение, тело цикла уже не выполняется и управление передается следующему оператору.

В следующем примере находится сумма целых чисел от 1 до 100:

```
var
  I, sum: Integer;        // объявление переменных
begin
  sum := 0;               //  инициализация переменной
  for I := 1 to 100 do    //  заголовок цикла
    sum := sum + I;       //  тело цикла
end;
```

В некоторых случаях необходимо последовательно менять значение счетчика не в возрастающем, а в убывающем порядке. В этих случаях используется ***декрементный*** цикл *for*, синтаксис которого имеет вид:

for *переменная_счетчик* : = *начало* downto **конец** do begin
операторы // тело цикла
end;

Алгоритм выполнения оператора остается прежним, только на каждой новой итерации счетчик не увеличивается, а уменьшается на один шаг (целочисленные счетчики уменьшаются на единицу). Конечное значение в заголовке цикла должно быть меньше начального или равно ему.

Значение переменной счетчик увеличивается или уменьшается автоматически. Попытка изменить его в теле цикла *for* вызовет сообщение об ошибке компиляции. Еще более опасная ошибка – попытка изменить в теле цикла значение любого из выражений начало или конец. В этом случае компилятор не сообщил об ошибке, однако изменение на цикл не подействует, так как эти выражения были вычислены один раз в начале цикла и больше не вычисляют. Заметить такую ошибку очень трудно.

В следующем примере демонстрируется использование в качестве счетчика переменной символьного типа. Запомните: счетчик не обязательно должен иметь тип Integer, он может иметь любой порядковый тип. В этом примере в строковую переменную alphabet записываются все буквы в нижнем регистре:

```
var
  letter :      Char ;
  alphabet :    String ;
begin
   alphabet  :=  '';
   for letter := 'a' to 'z' do begin
                                alphabet := alphabet + letter ;
                               end ;
end ;
```

Оператор цикла while

Оператор for предназначен для повтора тела цикла заранее определенное число раз. Это число вычисляется в момент первого выполнения заголовка цикла и на протяжении последующих итераций остается неизменным.

Во многих случаях число выполнения тела цикла заранее неизвестно. Прекращение итераций должно происходить во время работы программы в зависимости от определенного условия. Например, такие операторы цикла часто применяют, когда надо ввести данные из файла. Окончание файла служит хорошим критерием завершения цикла. Такие операторы цикла используют, например, для вычисления значения функции с заданной точностью, достижение которой завершает итерационный процесс расчета.

При этом теоретически допустимо, что итерации продолжаются бесконечно.

Оператор цикла ***while*** — это оператор цикла, в котором тело цикла повторяется заранее неизвестное число раз. Работа этого оператора продолжается в зависимости от определенного условия.

Оператор цикла ***while*** (или говорят цикл с предусловием) записывается следующим образом:

```
while условие_продолжения do begin
                                операторы
                              end;
```

Условие_продолжения — это логическое выражение языка Delphi, которое вычисляет логическое значение. Если это значение равно ***True***, то тело цикла выполняется, после чего условие проверяется вновь. Эти операции повторяются, пока значение выражения станет равно ***False***. После этого итерации прекращаются и управление передается следующему оператору за циклом.

В качестве тела оператора цикла указывается *обычный* или *составной оператор* языка Delphi. Пример нахождения суммы целых чисел от 1 до 100 перепишем с использованием оператора *while*:

```
var
  I, sum: Integer;
begin
 sum := 0; I := 1;
 while (I <= 100) do
 begin
   sum := sum + I;
   I := I + 1
 end;
end;
```

Если в этом примере убрать из тела цикла оператор увеличения переменного на единицу, то цикл будет выполняться бесконечно, что приводит к так называемому *зависанию* программы. В этом отличие этого оператора от оператора for, в котором увеличение счетчика происходит автоматически. В операторе while понятия счетчика цикла формально не существует. Программист должен самостоятельно реализовывать подсчет числа повторений.

Оператор цикла repeat

Этот оператор цикла (цикла с постусловием) записывается следующим образом:

```
repeat
  операторы
until условие_завершение;
```

Условие, как и в предыдущем случае, — это логическое выражение, вычисляющее логическое значение. В данной версии оператора цикла оно служит критерием не продолжения цикла, а наоборот, его завершения. Если значение выражения равно True, то оператор цикла прекращается и управление предается далее. Кроме того первая проверка условия завершения происходит не сразу, а только после того, как тело цикла выполнено один раз.

В операторе *while* тело цикла — это одиночный оператор, который может быть составным. В операторе с постусловием между ключевыми словами repeat и until может располагаться произвольное число команд языка Delphi и операторной скобки можно не ставить.

Вот как находится сумма целых чисел от 1 до 100 с помощью цикла с постусловием:

```
var
  I, sum: Integer;
begin
 sum := 0;
 I := 1;
```

```
at
    sum := sum + I;   // запись без операторной
    I := I + 1            // скобки begin и end
  until (I > 100);
end;
```

В этих примерах переменная sum является накопителем. В ней суммируются (накапливаются) значения, образующие окончательный ответ. Обратите внимание, что эти два кода очень похожи. Фактически они отличаются только условиями прерывания циклов. Тело цикла while выполняется, если значение *условия* равно *True*. В то же время тело цикла repeat выполняется, если значение *условия* равно *False*. Если *условие* в теле цикла не изменяется, то while или repeat превращается в бесконечный цикл. Если тело цикла всегда должно быть выполнено хотя бы один раз, то предпочтителен в использовании цикл repeat, в противном случае более удобен цикл while.

Циклы while и repeat являются недетерминированными. Это означает, что количество итераций необязательно должно быть известно до начала выполнения цикла. Циклы while и repeat можно также использовать вместо for в качестве детерминированных. Таким образом, каждый цикл for можно заменить эквивалентным ему циклом while или repeat, но не наоборот.

Если в циклах while и repeat необходимо предусмотреть увеличение или уменьшение значения переменной порядкового типа (например, для выбора следующего и предыдущего значения такой переменной) , то надо использовать встроенных порядковых подпрограмм:

Функция ***Pred()*** – возвращает предыдущее порядковое значение аргумента;
Функция ***Succ()*** - возвращает следующее порядковое значение аргумента;
Функция ***Odd()*** - возвращает True, если аргумент является нечетным числом;
Функция ***Ord()*** - возвращает порядковое значение выражения порядкового типа;
Процедура ***Dec()*** – уменьшает порядковую переменную на 1 или на n;
Процедура ***Inc()*** – увеличивает порядковую переменную на 1 или на n.

Например, для циклического перебора букв алфавита можно написать:

```
var
  letter: Char;
begin
   letter := 'a';
   while (letter <= 'z') do
   begin
      ..................
      letter := Succ(letter);
   end;
.................................................
```

В циклах while, repeat часто используются ***флажки.*** Флажком называется булева (логическая) переменная, обозначающая удовлетворение или неудовлетворение некоторому условию.

Управление работой циклов

Циклы, используемые в программах, могут быть очень сложными. Для подобных случаев в языке Delphi предусмотрены дополнительные средства управления работой циклов. Нередко работу оператора цикла (прежде всего оператора for) требуется прервать раньше, чем выполнятся все итерации. Для этого предназначена команда ***Break***.

Как только команда ***Break*** встречается в теле цикла, цикл немедленно прекращается и управление передается оператору, следующему за ним. Тем самым завершается выполнение цикла.

Другая команда ***Continue*** обеспечивает досрочное завершение очередного прохода цикла. Она эквивалентна передачи управления в самый конец циклического оператора и продолжения повторения.

Вложенные циклы

Тело цикла может быть свою очередь циклом, т.е. цикл может быть вложен в цикл. Подобные циклы называются вложенными циклами.

Счетчик каждого вложенного цикла должен иметь уникальное имя. В следующем фрагменте кода вложенные циклы for используются для вывода в область просмотра memOitput таблицы умножения размером 12x12:

```
{Вывод таблицы умножение размером 12x12}
var
  row:          Integer;
  col:          Integer;
  lineOut:      String;
  product:      String;
begin
  for row :=1 to 12 do
  begin
    lineOut := '';
    for col := 1 to 12 do
    begin
       Str( (row * col) :4, product) ;
       lineOut  := lineOut + product ;
    end ;
 end ;
end ;
```

Мы тексты программ приводим различными формами записи структурного кодирования. Вам самим выбирать вариант оформления текста и выработать свой стиль структурирования.

Резюме

Мы в лекции познакомились как структура повторение реализуется в языке Delphi одним из трех операторов:

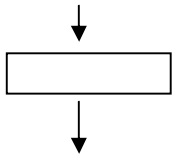
- оператор for
- оператор while
- оператор repeat

Несложно показать, что и в случае структур повторения любая из них может быть реализована на основе одной структуры while. Все, что можно сделать, используя структуры for и repeat, может быть реализовано при помощи структуры while (хотя выглядеть это будет не лучше, чем в случае ограничения структур выбора одной структурой if). Подводя итог всем этим результатам, мы должны сделать вывод, что любая форма управления из тех, что когда-либо могут понадобиться в Delphi, может быть реализована на основе следующих управляющих структур.

- последовательная структура
- структура if
- структура while

Блок-схемы этих структур имеют только один вход и один выход. А это позволяет этим управляющим структурам объединяться только двумя способами – *суперпозицией (последовательным соединением)* и *вложением.* Доказано, что для реализации любого алгоритма требуются только три формы структур управления: *последовательная, выбора, повторения.* Эти принципы положены в основу *парадигмы структурного программирования.* Поэтому правила построения структурированных программ выглядит так:

1. Начинайте разработку с простейшего варианта, например, его блок-схема может иметь следующий вид.



2. Любой прямоугольник (действие) может быть заменен двумя последовательными прямоугольниками (действиями).
3. Любой прямоугольник (действие) может быть заменен любой управляющей структурой (последовательной, if-then, if-then-else, case, for, while, repeat).
4. Правила 2 и 3 могут применяться неограниченное число раз в произвольном порядке.

Применяя эти правила, вы всегда сможете создавать блок-схемы алгоритма с ясной, отчетливой структурой. При написании текста программы форма записи операторов также должны обладать определенной вид, наглядно подчеркивая и выделяя отдельные управляющие структуры. Для этого используют отступы, пробелы, пустые строки и комментарии (элементы структурированного кодирования).

Вопросы для самопроверки

1. Как работает безусловный оператор цикла for?
2. Как работает условный оператор цикла while?
3. Как работает условный оператор цикла repeat?
4. Какой командой прерывается оператор цикла?
5. Какой командой принудительно завершается текущая итерация в операторе цикла?
6. Разница между условием завершения и условием продолжения цикла.
7. Назовите порядковые подпрограммы и опишите их назначение.
8. Что такое счетчик? Что такое флажок?
9. Что такое тело цикла?
4. Что такое заголовок цикла?
5. В чем различие между программированием и кодированием?
6. Что такое структурированное кодирование?
7. Что такое структурное программирование?
8. Отличие между понятиями «структурированное кодирование» и «структурное программирование».
9. Что такое счетчик цикла?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 7. Парадигма процедурно-структурного программирования
Подпрограммы: Процедуры и функции – часть 1

План
Подпрограмма
Объявление и вызов процедур и функций
Значение, возвращаемое функцией
Параметры подпрограмм
Глобальные и локальные переменные
Завершение работы подпрограммы
Внутренняя организация Delphi-программы, использующей подпрограммы
Парадигма процедурно-структурного программирования.

Как известно, базовыми структурами алгоритма являются: *действие (функция), выбор (ветвление)* и *повторение.* Мы уже знаем, что, *структура действие* на языке программирования Delphi может быть реализована в виде операторов присваивания, оператора ввода и оператора вывода. Эта структура на языке программирования может быть еще представлена в виде *оператора подпрограммы*.

Подпрограмма

Обычно человек выполняет сложное дело по частям или решает сложную задачу, разбивая ее на подзадачи. Аналогично создание крупных программ делается разбиением на отдельные фрагменты или, говорят, на *подпрограммы*. В языке Delphi подпрограмма оформляется в двух видах: *процедура и функция*.

Концепция процедур и функций, объединяемых одним словом *подпрограмма* существовала уже в самых первых языках программирования.

Подпрограмма *— это явно выделенный в исходном тексте набор команд и операторов, имеющий уникальное имя. Этот набор вызывают на выполнение обращением к его имени. Выполняемый подпрограммой набор операторов называют* ***телом подпрограммы.***

Использование подпрограмм, во-первых, существенно сокращает размер исходного текста, который к тому же становится более понятным. Во-вторых, появляется возможность создания повторно используемых модулей кода. На идее подпрограмм, в частности, построены **программные библиотеки стандартных функций**. Важнейшая особенность подпрограммы — *настраиваемость.* Достигается она за счет *параметров* подпрограммы. Когда процедура или функция вызывается *по имени*, то конкретные значения, предназначенные для обработки, задаются как параметры в круглых скобках непосредственно за этим именем.

Параметр *— это переменная, значение которой должно быть определено при вызове подпрограммы. Для вычислений значение параметра передается в тело подпрограммы.*

Если, например, вызывается функция расчета синуса Sin(), то величина исходного угла задается в качестве ее параметра. Последовательность расчета остается всегда одной и той же, а значения, используемые в этой последовательности программист задает как параметры в каждом конкретном случае обращения к подпрограмме.

Подпрограммы делятся на две группы: *процедуры* и *функции.*

Процедура *— это подпрограмма, выполняющая определенную последовательность действий. Процедура необязательно возвращает что-либо в вызывающую программу. Процедура может*

передать информацию в вызывающую программу посредством списка параметров (с помощью ***var****-переменных).*

Функция *— это подпрограмма, в теле которой вычисляется значение, доступное вызывающей программе. Функция возвращает хотя бы одно значение в вызывающую программу.*

Эти жесткие принципы назначения и использования функции в очередных версиях Дельфи отменены. Так так в других языках программирования нет процедур а только функции, так в Дельфи функцию можно использовать так же как и процедуру. И так современное понимание функции можно определить как: значение заданного типа, которое вычисляется в теле функции и затем передается в вызвавшую программу, называют **значением, возвращаемым функцией.**

Функция *— это подпрограмма, в теле которой* ***может*** *вычисляться значение, доступное вызывающей программе. Функция* ***может*** *возвращать значение в вызывающую программу посредством своего имени, но может и не возвращать значения и быть использованная как процедура. В каждом случае в заголовку функции объязательно указать тип результата функции.*

Функция схожа как с процедурой, так и с переменной. Она вызывается по имени как процедура, и может иметь параметры. Обращение к имени функции формирует значение, как и обращение к имени переменной. Только это значение не хранится, а вычисляется телом функции.

Объявление и вызов процедур и функций

При описании подпрограммы в исходном тексте программы вводится имя подпрограммы и список параметров, а также ее тело — набор команд. Описание подпрограммы не вызывает никаких действий программы — набор команд в теле подпрограммы сам по себе не выполняется, он просто служит для определения ее содержимого.

Вызов тела подпрограммы происходит только внутри одного из блоков begin .. end. Для этого надо указать имя подпрограммы и, при необходимости, набор значений параметров.

Процедура объявляется следующим образом:

```
procedure имяПроцедуры(формальный_параметр1: тип1;
                       формальный_параметр2: тип2; ….);

локальные-объявления;

begin              //тело-процедуры
   операторы;   // его текст пишется в терминах формальных параметров
end;               //и локальных переменных
```

Функция объявляется так:

```
function имяФункции(формальный_параметр1:тип1; …):тип_функции;
локальные-объявления;
begin
  операторы;                                  // тело-
   имяФункции := возвращаемое_значение;       // функции
end;
```

или:

```
function имяФункции(формальный_параметр1:тип1; …):тип_функции;
локальные-объявления;
begin
  операторы;                              // тело-
  Result := возвращаемое_значение;        // функции
end;
```

или:

```
function имяФункции(формальный_параметр1:тип1; …):тип_функции;
локальные-объявления;
begin
 операторы;                          // тело функции, не возвращаемой ничего
end;
```

Второе объявление функции содержит неявное объявление переменной **Result** того же типа что тип функции. Переменной **Result** можно воспользоваться как обычной переменной, это означает, что может она появится с правой стороны выражения:
Result := Result+a (имя функции не может аналогично использоваться). Имя функции доступно только для чтения.

Тексты тел подпрограмм пишется в терминах формальных параметров и локальных переменных.

Именем процедуры или функции может быть любой допустимый идентификатор языка Delphi. Для функции дополнительно указывается тип возвращаемого ей значения.

В разделе локальных объявлений допускаются любые объявления типов, констант и переменных, начинающиеся с ключевых слов *type, const, var*. Кроме того, разрешается определение процедур и функций. Все определения типов, констант, переменных и подпрограмм в разделе локальных объявлений действуют исключительно в рамках текущей подпрограммы и «не видны» за ее пределами. В языке Паскаль и основанном на нем языке Delphi действует принцип, согласно которому компилятор, встречая при разборе исходного текста очередной идентификатор, ищет, прежде всего, его локальное определение (например, в пределах текущей процедуры, класса или модуля) и лишь потом рассматривает глобальные описания из других модулей.

Пусть, например, определена процедура Print4:

```
Procedure  Print4;
var
   N: Integer;
begin
    N := 2 + 2;
   WriteLn(N);
end;
```

Для вызова процедуры Print4 необходимо указать ее имя в последовательности команд:

```
…….
WriteLn('Вызываем процедуру Print4 ‘);
Print4;    // вызов процедуры
X := 0;
……..
```

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Значение, возвращаемое функцией

Функции, в отличие от процедур, предназначены для расчета значения и возврат его в вызывающую программу. Функцию разрешено вызывать как процедуру: вводом ее имени с параметрами. В этом случае возвращаемое ею значение просто теряется.

Возвращаемое функцией значение формируется внутри ее тела с помощью оператора присваивания. В левой его части ставится идентификатор Result, а в правой — выражение, соответствующее типу возвращаемого функцией значения. Этот оператор может располагаться в любой точке внутри функции (не обязательно последним). Вместо переменной Result можно указывать идентификатор, совпадающий с именем функции.

Следующие две записи функции Kvadrat эквивалентны:

```
function Kvadrat(X: Integer): Integer;
begin
  Result  := X  *  X
end;

function Kvadrat(X: Integer): Integer;
begin
  Kvadrat := X * X
end;
```

Синтаксис вызова функции с присвоением возвращаемого значения некоторой переменной имеет вид:

имя_переменной := имяФункции(список_фактических_параметров);

Например, `Y := Kvadrat(2);`

Вызов функции (имя функции) может использоваться как операнд выражения. Например, `Y := 2 + Kvadrat(2) + a;`

Параметры подпрограмм

Параметры предназначены для передачи в процедуру или функцию набора значений, которые могут варьироваться в момент вызова подпрограммы. Ведь полезность подпрограмм заключается именно в том, что они обрабатывают разные входные значения (параметры), рассчитывая на их основе итоговый результат. Многие стандартные процедуры (например, процедуры вывода на экран) сначала получают выводимый текст в качестве параметра, и лишь потом отображают его на экране.

Подпрограмма может иметь несколько параметров или ни одного. Список параметров заключается в круглые скобки. Каждый параметр записывается в формате:

идентификатор-параметра: тип-параметра

Параметры отделяются друг от друга точкой с запятой. Если список параметров пуст, его можно опустить вместе с круглыми скобками. Если подряд идет несколько параметров одного типа, то их имена можно объединить, записав через запятую. Например:

```
function MySin(X: Integer): Real;
procedure PrintText(Name: String; Address: String);
procedure PrintText2(Name, Address: String);
```

Внутри подпрограммы параметры трактуются как локальные переменные, поэтому пытаться переопределять их в качестве собственных локальных переменных некорректно.

Формальные и фактические параметры

При создании программы значения передаваемых в нее параметров еще неизвестны. При описании в заголовке подпрограммы указываются формальные параметры. При вызове подпрограммы указываются фактические параметры, которые и передаются в нее.

Формальный параметр *— это* ***переменная****, которая определена в заголовке описания подпрограммы, точнее это переменная, которой при вызове подпрограммы присваивается значение соответствующего фактического параметра.*

Фактический параметр *— это значение, передаваемое в подпрограмму при ее вызове. Каждому фактическому параметру соответствует один формальный параметр. Фактические параметры – это* ***переменные*** *и* ***выражения, значения*** *(или* ***адреса****) которых передаются в подпрограмму.*

Пусть имеется описание функции Kvadrat:

```
function Kvadrat( X :  Integer):  Integer;
begin
  Kvadrat  := X  *  X
end;
```

Параметр X — это формальный параметр. Пусть происходит вызов этой функции в операторе присваивания:

```
X  :=  2 + Kvadrat(2) + a;
```

Значение 2, введенное в списке параметров вызываемой на выполнение функции, — это фактический параметр.

При составлении списка фактических параметров необходимо учитывать следующее.

- Количество фактических параметров должно быть равно количеству формальных параметров.
- Первый фактический параметр в списке соответствует первому формальному, второй – второму и т.д.
- Тип каждого фактического параметра должен совпадать с типом соответствующего ему формального параметра.
- Имя фактического параметра никак не связано с именем соответствующего формального параметра.
- Необходимо строго различать способы передачи данных – *по ссылке* или *по значению.*

Передача параметров по значению и по имени

Параметры могут передаваться **по значению** и **по имени**. В случае передачи параметра по значению все фактические параметры при вызове подпрограммы копируются в формальные параметры (*промежуточные внутренние переменные*) и только потом передаются внутрь функции или процедуры. Это гарантирует неизменность переменных, представленных как фактические параметры. Например:

```
// описание процедуры VarX
procedure VarX(X: Integer);
```

```
begin
  X := 1;
end;
.....
// основной текст программы
X := 5;
VarX(X);   // вызов процедуры VarX
Write(X);  // значение переменной X - по-прежнему 5     ,
```

После вызова процедуры VarX(X) значение переменной X в основном коде не изменится, хотя внутри тела VarX происходит явное обращение к содержимое параметра (фактически — к внутренней переменной X). Так как параметр передан по значению, для него предварительно формируется промежуточное хранилище (внутренняя переменная X, формальный параметр процедуры), временное содержимое которого и модифицируется оператором X : = 1. По завершении процедуры это хранилище (формальный параметр X) просто уничтожается.

Передача фактического параметра по имени разрешает модификацию его значения внутри процедуры или функции. Перед идентификатором формального параметра, передающегося по имени, надо поставить ключевое слово **var**.

Например:

```
procedure VarX(var Z:   Integer);
begin
  Z   :=  1;
end;
...........
X   :=  5;
VarX(X);
Write(X); // значение переменной X изменилось с 5 на 1
```

После вызова процедуры VarX значение переменной X станет равным 1, так как она использована (явно подставлена) в тело процедуры VarX везде, где имеется обращение к формальному параметру Z.

В качестве фактического параметра, передаваемого по имени, может быть использовано только имя существующей *переменной*. При передаче фактического параметра по значению можно использовать любое допустимое *выражение*, вычисляющее значение нужного типа. Например, допускается вызов процедур VarX (X + 2), если параметр передается по значению. Если же он передается по имени, то возможны только вызовы вида VarX(X), VarX(i2), VarX(Peremennaya) подобные.

Дополнительные спецификаторы параметров

Способ передачи параметров по имени или значению, придуманный много десятков лет назад, по мере развития программной индустрии был усовершенствован. В языке Delphi, в частности, появились способы уточнения возможности модификации формальных параметров. Так, в ряде случаев желательно запретить модификацию формальных параметров процедуры в ее теле, даже если они переданы по значению. Например, код составляется несколькими разработчиками и необходимо соблюсти требование неизменности значений параметров на всем протяжении работы подпрограммы. Для этого используют так называемые *константные параметры,* значения которых разрешено лишь считывать. Константные параметры формируются вводом ключевого слова const перед именем параметра:

```
function VarX(const Z: Integer): Integer;
begin
  Z := 1;        // запись в Z запрещена! ошибка компиляции
  Result := Z;  // разрешено
end;
```

Формальный параметр может быть предназначен только для записи значения. Считывать значение такого параметра не разрешается. Подобный параметр определяется ключевым словом **out**, вводимым перед идентификатором параметра:

```
function VarX(out  Z:  Integer):  Integer;
begin
  Z  :=  1;        //  разрешено
  Result  :=  Z;  //  считывание  значения  Z  запрещено!  ошибка  компиляции
end;
```

Передача параметров неизвестного типа

Технологии, поддерживаемые системой Delphi, нередко требуют передачи параметров заранее неизвестных типов. В таком случае описание типа параметра можно опускать, однако параметр должен сопровождаться одним из описателей *var, out* или *const*. Обращение к параметру неизвестного типа в теле подпрограммы возможно только после явного приведения его к определенному типу. При этом компилятор не проверяет корректность обработки нетипизированных параметров. Пользоваться данной возможностью надо осторожно, так как, например, следующая запись становится синтаксически корректной:

```
procedure  nt(var x);
begin
  x  : = 5;
  X : = 'abc';
end;
```

Необязательные параметры

В списке параметров допускаются необязательные элементы, которые при вызове подпрограммы можно не указывать. Они всегда следуют в конце списка параметров. Обязательные параметры между ними не допускаются. Необязательные параметры записываются по схеме

идентификатор: тип = значение

Если соответствующий параметр опущен, то вместо него автоматически подставляется заданное значение.

Глобальные и локальные переменные

Подпрограммы могут возвращать результат в основную программу не только при помощи параметры-переменные (***var-***переменные) и выходных параметров (***out***-переменные), но и *непосредственно изменением глобальных переменных*.

Переменные, описанные в основной программе, являются глобальными по отношению к внутренним процедурам и функциям. Это означает, что они доступны в подпрограммах, описанных внутри основной программы.

Переменные, описанные внутри процедур и функций, называются локальными. Они порождаются при каждом входе в подпрограмму и уничтожаются при выходе из этой подпрограммы, т.е. локальные переменные существуют в памяти компьютера только при выполнении подпрограммы и недоступны (невидны) в основной программе. *Формальные параметры подпрограмм являются локальными переменными*. Текст подпрограммы пишется, обычно, в терминах локальных переменных. Если имя локальной переменной совпадает с именем глобальной переменной, то при выполнении подпрограммы такая глобальная переменная становится недоступной, т.е. «закрывается» локальной переменной до завершения работы подпрограммы.

Перепишем текст программы P6 (см. в следующих страницах), используя механизм глобальных параметров. В этом случае процедуру max можно написать без формальных параметров.

```
program P5; // заголовок основной (вызывающей) программы
// ниже описательная часть основной программы
……..
var
  a,b,c,x,y,z,t1,t2,t3,u: real; // это глобальные переменные
procedure max;  // это начало подпрограммы
   begin            // формальных параметров нет и нет локальных переменных
    if x>y then z := x else z := y
   end;             //  это конец подпрограммы

begin         // это начало исполнительная части основной программы
  readln(a,b,c);
  x := a;
  y := a + b;
  max;
  t1 := z;
  y := b + c;
  max;
  t2 := z;
  x := a+b*c;
  y := a*c;
  t3 := z;
  u := (t1 + t2)/(1 + t3);
  writeln('u=', u:7:3);
  readln
end.              //    это конец основной (вызывающей) программы
```

Завершение работы подпрограммы

Работу подпрограммы иногда необходимо прервать до момента ее естественного завершения (достижения ключевого слова end). Так бывает, если, например, все необходимые расчеты уже выполнены. Для мгновенного завершения подпрограммы предназначена команда (стандартная процедура) *Exit*. Как только она встречается в коде любой подпрограммы, та завершается. Например:

```
procedure PrintDiv(A, B: Integer);
begin
  if B = 0 then Exit;
  Write(A div B);
end;
```

В процедуре, выводящей на экран частное от деления параметра A на параметр B, сначала выполняется проверка, равен ли второй параметр нулю. Если это так, работа процедуры немедленно завершается (выполняется команда Exit), в противном случае ее функционирование продолжается в нормальном режиме.

Внутренняя организация Delphi-программы, использующей подпрограммы

Задача

Вычислить u = (max(a, a+b) + max(a, b+c))/(1 + max(a+b*c, a*c))

Решение

В решении задачи можно выделить подзадачу «поиска максимального из двух чисел». Поэтому сначала оформим алгоритм решение подзадачи в виде процедуры с именем *max*. Пусть формальными параметрами процедуры будут: *x, y* - для описания двух чисел и *z* – для возвращаемого процедурой значения. Тогда текст процедуры в терминах этих обозначений будет таким:

```
procedure max(x, y: real; var z: real);
begin
  if x>y then z := x else z := y
end;
```

Теперь можем составить текст Delphi-программы для вычисления u, использующей этой процедуры. При выполнении эта программа будет использовать (вызывать) процедуру max(). Тогда текст **основной (использующей подпрограммы) программы** будет следующим:

```
program P5;   // заголовок основной программы
// ниже описательная часть основной программы
{$APPTYPE CONSOLE}
Uses
  SysUtils;
Var
  a,b,c,t1,t2,t3,u: real;

procedure max(x, y: real; var z: real); // начала текста используемой подпрограммы
  begin
   if x>y then z := x else z := y
  end;   // конец текста используемой (вызываемой) подпрограммы
// ниже исполнительная часть основной программы
```

```
begin
  readln(a,b,c);
  max(a,a+b,t1);        // 1-й вызов подпрограммы, результат получен в t1
  max(a,b+c,t2);        // 2-й вызов подпрограммы, результат получен в t2
  max(a+b*c,a*c,t3);  // 3-й вызов подпрограммы, результат получен в t3
  u := (t1 + t2)/(1 + t3); // использование результатов работы подпрограммы
  writeln('u=', u:7:3);
  readln
end.
```

При процедурно-структурной парадигме программирования пользовательские подпрограммы, использующиеся в вызывающей (основной) Delphi-программе, полными текстами приводятся в описательной части основной программы. Такова внутренняя структура текста Delphi-программы, написанной по правилам процедурно-структурного программирования.

Резюме

Парадигма процедурно-структурного программирования.

Можно сказать, практика создания различных программ выработала следующие первые парадигмы программирования:

Структурное программирование. При структурном программировании запись текста программы делается с помощью только трех базовых структур (предложений) алгоритма: *действие (функция), выбор и повторение*. Для моделирования логики алгоритма задачи эти предложения могут комбинироваться только двумя способами: **суперпозицией** *(последовательным соединением структур)* и **вложением** *одной структуры внутрь другой.* Поэтому можно сказать, что структурное программирование имеет последовательную природу. Из-за этого использование только таких предложений порождают программы, которые проще для чтения и понимания. Такая программа выполняется последовательно одна структура за другой. Это и как раз подчеркивалась в первой лекции называнием структурное линейное программирование.

Процедурно-ориентированное программирование (точнее процедурно-структурное, кратко процедурное программирование). В этом случае в добавок к структурированной форме записи текста программы, текст программы сложной задачи разбивается еще на подпрограммы (процедуры и функции). Процедурное программирование как методика разработки программного обеспечения разделяет программу на *данные и процедуры*, обрабатывающие эти данные. При этом акцент делается на обработке (алгоритме), необходимой для выполнения требуемых вычислений. В этом случае используется такая *парадигма: анализируя задачу реши, какие требуются процедуры и функции; используй лучшие доступные алгоритмы для подпрограмм.*

Процедурное программирование имеет последовательную природу. В процессе выполнения процедурной программы последовательно выполняются операторы текста программы, в частности, последовательно вызываются различные процедуры. После выполнения последней из них программа завершает свою работу.

При процедурно-структурной парадигме программирования на языке Delphi в основной (вызывающей) программе тексты вызываемых, т.е. используемых в ней пользовательских подпрограмм (процедур и функций) приводятся полными текстами описательной ее части. При этом тексты подпрограмм и основной программы пишутся по правилам структурного программирования.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Консольное приложение Delphi как раз пишется по правилам процедурно-структурного программирования.

Вопросы для самопроверки

1. Что такое подпрограмма?
2. Чем процедуры отличаются от функций?
3. Как записываются процедуры и функции?
4. Чем отличаются основная программа и подпрограмма?
5. Как задается возвращаемое функцией значение?
6. Зачем нужны параметры подпрограмм?
7. В чем разница между передачей параметров по имени и по значению?
8. Какой командой экстренно завершается работа подпрограммы?
9. Дайте определения фактических и формальных параметров и опишите их соотношение.
10. Что такое передача параметров по ссылке и по значению? Каким образом в программе можно задать вид передачи параметров?
11. В каких случаях использование функции предпочтительнее использования процедуры?
12. Приведите синтаксисы вызовов процедуры и функции. Выделите их отличия.
13. В чем смысл процедурно-структурной парадигмы?
14. Чем различаются парадигмы структурного и процедурного подходов?
15. Что общего у структурного и процедурного подходов?
16. Что такое локальные и глобальные переменные?
17. Каково время жизни локальных и глобальных переменных?
18. Чем отличаются локальные и глобальные переменные?
19. Какие обозначения используются при написании текста подпрограммы?
20. Какие обозначения используются при написании текста основной программы,
21. спользующей подпрограмм?
22. Что будет, если имена формальных и фактических параметров будут совпадать?
23. Что такое пользовательские и библиотечные подпрограммы?
24. Различие использования библиотечных и пользовательских подпрограмм?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 8. Парадигма процедурно-структурного программирования
Подпрограммы: Процедуры и функции – часть 2.

План

Вложенные подпрограммы
Отложенная реализация подпрограмм
Перезагрузка процедур и функций
Внешние определения подпрограмм
Рекурсия
Встраиваемые подпрограммы

Вложенные подпрограммы

Описания подпрограмм разрешено вкладывать друг в друга. Вложенная подпрограмма считается локальной (доступной только в теле родительской процедуры), как и переменные, определенные внутри процедуры или функции. Внутри вложенной подпрограммы можно свободно использовать локальные переменные родительской процедуры или функции.

Вложенная подпрограмма описывается после раздела определения локальных переменных *var*, но до первого ключевого слова *begin* в родительской подпрограмме. Например:

```
procedure PrintSin2(X: Real);
 var Y: Real;
     procedure Sin2;      // процедура, вложенная
                          // в процедуру PrintSin2
     begin
       Y := Sin(X*2);     // в локальную переменную Y
                          // родительской процедуры
                          // заносится значение синуса двойного угла
     end;
begin            // начало тела процедуры PrintSin2
   Sin2;         // расчет результата вызовом вложенной процедуры
   Write(Y);     // печать результата
end;
```

Отложенная реализация подпрограмм

Процедуру или функцию нельзя использовать, если до обращения к ней она не описана. Язык Паскаль, на основе которого был создан язык Delphi, разработан давно. Требование определять процедуры и функции до их использования связано с ограниченными возможностями существовавших тогда компьютеров. Паскаль задумывался с прицелом на так называемую однопроходную архитектуру компилятора. Текст программы анализировался в один проход, построчно, «забегания вперед» не происходило. Такой принцип позволял создавать быстро работающие транслирующие системы. Последующие языки (например, C) уже изобретались под многопроходную, оптимизирующую технологию компиляции, многократно исследующую исходный текст. В них столь жестких ограничений на структуру исходного текста не существует.

Следующая запись неверна:

```
procedure A;
begin
  C;    // ошибка - процедура C еще не описана!
end;

procedure C;
begin
  A;
end;
```

В процедуре A происходит обращение к процедуре C, однако процедура C описывается лишь далее по тексту. Поэтому компилятор к моменту анализа описания процедуры A еще не знает о существовании процедуры C.

Чтобы обойти данное ограничение, пока еще неопределенную процедуру или функцию можно объявить в виде заголовка с добавлением ключевого слова *forward*. Оно подскажет компилятору, что реализация заголовка расположена далее.

```
// Подсказываем компилятору, что C - это процедура.
// Она будет описана позднее.
procedure C; forward;

procedure A;
begin
  C;    // теперь это уже корректный вызов
end;

procedure C;
begin
  A;
end;
```

Перезагрузка процедур и функций

В программах часто востребованы процедуры и функции одинакового предназначения, различающиеся лишь типами используемых параметров. Например, можно сложить два целых числа и получить целый результат, но можно сложить и два вещественных числа и получить вещественный результат. Алгоритм реализации не всегда одинаков, поэтому требуется разнести обработку данных разных типов по разным процедурам. Задачу подбора подходящей подпрограммы для параметров разного типа можно переложить на компилятор. Разработчику достаточно задать одно и то же имя подпрограммы.

Перезагрузка *(или* ***перегрузка)*** *подпрограммы — это создание новой подпрограммы с именем уже существующей подпрограммы. Заголовок новой подпрограммы отличается от заголовка существующей подпрограммы количеством либо типом параметров (в случае функции тип возвращаемого значения не может быть единственным отличием перезагруженных функций, так так по способу вызова функции должна быть выбрана одна из перезагруженных функции).*

В конце заголовка перегружаемых подпрограмм ставится ключевое слово *overload*. Пусть в программе определены три перегруженные процедуры с одинаковым именем Test:

```
procedure Test(n: Integer); overload;
procedure Test(x: Real); overload;
procedure Test(s: String); overload;
```

Их реализация будет разной:

```
Procedure Test(n: Integer);
begin
   Write('Целое значение');
end;
Procedure Test(x: Real);
  Write('Вещественное значение'
end;
Procedure Test(s: String);
begin
  Write('Текстовая строка');
end;
```

В коде программы разместим вызовы этих процедур, различающиеся типами параметров:

```
Test(1) ;
Test(1.0);
Test('!');
```

В каждом случае корректно вызывается правильный экземпляр процедуры - он автоматически определяется компилятором по типу фактического параметра.

Пусть в программе определены четыре перегруженные функии:

```
Function x(a:byte):byte;overload;//1
begin
result:=2*a;
end ;
{Function x(a:byte):real;overload; //2
begin
result:=3*a;
end ; }
Function x(a:real):real;overload;//3
begin
result:=4*a;
end ;
Function x:real;overload;//4
begin
result:=1;
end ;
```

Из них правильны лиш 1,3 и 4, так как 2 не мохеть отличатся от 1 лиш типом результата. Способ вызова определяет выбор функции:

```
writeln(x(1)); //1
writeln(x(1.00)); //3
writeln(x); //4
```

Внешние определения подпрограмм

Процедуры и функции хранятся не только в исходном тексте текущей программы, но и в библиотеках, которые подключаются к программе на этапе компиляции, во внешних модулях. Внешние модули могут представлять собой *объектные файлы,* полученные после компиляции, но ещё не преобразованные в исполнимый формат .EXE, или динамические библиотеки DLL. Если подпрограмма физически размещена во внешнем модуле, то достаточно указать в исходном тексте ее заголовок и пометить эту подпрограмму как внешнюю с помощью ключевого слова external. Например:

```
procedure MyImport(X: Real); external;
```

Объектные файлы создаются на разных языках программирования. Нередко программам на Delphi требуется подключать объектные файлы с расширения .OBJ, подготовленные компилятором языка C. Для подключения такого файл надо воспользоваться директивой препроцессора $L, следом за которой следуй имя объектного файла:

```
{$L MyImport.obj}
```

Здесь MyImport. obj — имя объектного файла, созданного с помощью компилятора языка C.

Если внешняя процедура или функция хранится в библиотеке DLL, то за ключевым словом *external* через пробел дополнительно вводится текстовое имя файла соответствующей библиотеки.

```
function MyImpFun: Real; external 'MyLib.dll';
```

Рекурсия

Язык Delphi допускает рекурсию. ***Рекурсия*** *— это способ организации подпрограммы, при котором подпрограмма в ходе выполнения своих операторов обращается сама к себе.*

Механизм рекурсивного управления очень мощный. Существуют языки программирования (например, LISP), которые полностью построены на идее рекурсивного управления последовательностью вычислений, без условных операторов и операторов цикла. Рекурсия позволяет компактно записывать алгоритмы, которые при реализации классическим императивным походом требуют гораздо большего числа операторов. Основной недостаток рекурсии — медленная работа рекурсивного процесса и повышенные требования к оперативной памяти.

Рассмотрим классический пример применения рекурсии для расчета факториала числа. Факториал числа N равен произведению чисел 1*2*3*..* (N-1) * N. Так, факториал числа 5 равен 1*2*3*4*5 = 120. С помощью оператора цикла функция расчета факториала запишется так:

```
function F(N: Integer): Integer;
var I, Mul: Integer;
begin
  Mul := 1;
  for I := 2 to N do
   Mul := Mul * I;
  Result := Mul
end;
```

Вот как факториал реализуется с помощью рекурсии:

```
function F(N: Integer): Integer;
begin
  if N = 1 then
    Result := 1   // для этого случая значение рекурсивной функции известно
                  // (базовая задача)
  else
    Result := N * F(N-l);   // для других значений параметра функция вызывает сама
  // себя с новым параметром до известного случая после чего выполняются вычисления
end;
```

Так вызов F(3) автоматически развернется в такую цепочку вызовов и вычислений:

```
F(3)
3*F(2)
3*2*F(1)
3*2*1
3*2
6
```

Рекурсивный метод обычно решает только одну элементарную задачу (или несколько таких задач), называемую *базовой задачей*. Если метод вызывается для решения базовой задачи, то он возвращает результат. Если метод вызывается для более сложной задачи, он делит ее на две концептуальные части: часть, которую метод умеет решать (базовая задача), и часть, которую метод решать не умеет. Чтобы решить эту рекурсивную задачу, последняя часть должна быть похожа на первоначальную задачу, но должна представлять ее упрощенную или уменьшенную версию. Поскольку эта новая задача является вариантом первоначальной задачи, то для решения этой меньшей задачи метод вызывает новую копию себя самого. Чтобы рекурсия успешно завершилась, в каждом последующем вызове метода самого себя, должна решаться все более простая версия первоначальной задачи, и последовательность этих более простых задач должна сойтись к базовой задаче. В конце концов, когда метод дойдет до базовой задачи, он возвратит результат выполнения последней своей копии, и получится обратная последовательность возвращения результатов до тех пор, пока первый вызов метода не вернет окончательный результат в вызывающий метод.

Если повторения действий цикла завершается, когда не выполняется условие продолжения цикла, то рекурсия завершается, когда процесс доходит до базовой задачи. Рекурсия многократно инициирует механизм вызова метода и, в результате, потребляет много вычислительных ресурсов. При этом расходуется много процессорного времени и выделяется большой объем памяти.

Тем не менее рекурсивные методы, например, используются компиляторами, чтобы определить, все ли идентификаторы в исходной программе правильно сформированы.

Встраиваемые подпрограммы

В целях повышения быстродействия результирующего кода подпрограмму можно описать как *встраиваемую в код.* Для этого в ее заголовке ставится ключевое слово *inline*:

```
function F(N: Integer): Integer; inline;
var
 i, Mul: Integer;
begin
  Mul := I;
  for I := 2 to N do
    Mul := Mul * I;
    Result := Mul
end;
```

Компилятор включает исходный текст встраиваемых подпрограмм по месту их вызова. В результирующий набор машинных инструкций попадут не вызовы таких подпрограмм, а напрямую их операторы, наравне с другими оператора окружающими вызов. Пусть, например, имеется текст:

```
……
x := 5;
y := F(x);
z := x+y;
……
```

Если функция F помечена как *inline*, компилятор попробует сформировать примерно такой код (предварительно объявив в текущем блоке локальные переменные i и Mul):

```
….
X := 5;
Mul := 1;
For I := 2 to x do
    Mul := Mul * I;
Y :=Mul;
Z := x + y;
……..
```

Вызов функции F корректно заменен ее телом.

Существует немало ограничений на «раскрытие» встраиваемых подпрограмм в исходном тексте, однако за всеми возможными ограничениями следит компилятор. В случае невыполнения одного из требований он просто не совершает никаких преобразований, игнорируя ключевое слово inline. Поэтому часто вызываем процедуры, влияющие на общую производительность, можно смело объявлять как встраиваемые.

Резюме

Когда в программе встречается вызов подпрограммы, то программное управление передается этой подпрограмме. Когда вызов подпрограммы завершается, подпрограмма возвращает управление вызывающей программе. Фактические параметры, передаваемые в подпрограмму, должны соответствовать типу формальных параметров подпрограммы, должны следовать в том же порядке и по числу совпадать с числом формальных параметров.

Локальные переменные подпрограммы автоматически создаются при входе в подпрограмму и существуют в памяти пока подпрограмма активна. Они автоматически уничтожаются, когда завершается подпрограмма, в которой они объявлены.

Вопросы для самопроверки

1. Можно ли вкладывать описания подпрограмм друг в друга?
2. Что значит, что подпрограмма перегружена?
3. Как компилятор узнает, что подпрограмма перегружена?
4. Что такое ключевое слово?
5. Как компилятор узнает, что подпрограмма внешняя?
6. Каким образом компилятор находит подпрограмму?
7. Что такое рекурсия?
8. В чем особенность рекурсивной программы?
9. Как описываются встраиваемые подпрограммы?
10. В чем отличие подпрограммы и программы?
11. Почему программа разбивается на подпрограммы?
12. Есть ли отличия между основной и вызывающей программой?
13. Для чего используется ключевое слово в программировании?
14. Как компилятор узнает, что в программе есть подпрограмма?

Лекция 9. Пользовательские типы

План
Пользовательские типы
Перечислимые типы
Ограниченные типы

Пользовательские типы

Как известно, типы данных в Delphi можно разделить на *стандартные*, т.е. предопределенные в языке, и определяемые программистом (*пользовательские*).

В языке Delphi пользователь может определять собственные типы данных. Они могут основываться на уже существующих типах (в таком случае они называются *производными типами)* или же быть *полностью оригинальными*. В последнем случае пользовательские типы формируются на основе явно заданного допустимого набора значений.

> *Пользовательский **тип** данных — это тип данных, который разработчик самостоятельно создает в своей программе.*

В языке Delphi пользовательским типам данных относятся перечисления, поддиапазоны, массивы, множества, записи, файлы.

Пользовательские типы, как правило, определяются в описательной части программы с помощью ключевого слова **type:**

```
Type
  имя_типа = описание_типа;
```

Тип, вводимый в употребление программистом по своему усмотрению, определяет лишь множество значений этого типа. Набор же допустимых операций над ними, а также правила упорядочения (если оно имеет место в определяемом типе) зафиксированы в языке и не могут быть изменены по желанию программиста.

Каждый тип должен быть каким-то образом специфицирован для его выделения среди всех возможных типов. В языке Delphi в качестве имени типа (спецификатора типа) используются обычные имена (идентификаторы).

Перечислимые типы (перечисления)

В случае стандартных типов значений мы имели дело с такими понятиями, как «целое число», «вещественное число», «литера» и «логическое значение», подразумевая под каждым таким понятием определенное множество его частных случаев.

Однако на практике нам приходится иметь дело с самыми различными понятиями, каждое из которых включает в себя свое множество частных случаев. Например, понятие «месяц года» объединяет в себе в качестве частных случаев месяцы с именами январь, февраль, …, декабрь; понятие «день» - дни с именами понедельник, вторник,…, воскресенье и т.д. и т.п.

При решении на компьютере задач, связанных и использованием понятий подобного рода, их отдельные частные случаи иногда кодируют в цифровой форме путем отображения на целые числа. В этом случае в записи алгоритма вместо явного указания нужного частного случая понятия указывается его код. Ясно, что при этом снижается наглядность записи алгоритма, затрудняется его понимание и проверка. Поэтому для избежания кодировки естественно принять в качестве значений обычные названия (имена) этих частных случаев.

Один из способов определить новый тип данных состоит в том, чтобы явно указать все значения, которые может принимать переменная этого типа.

Перечислимый тип *— это пользовательский тип данных, для которого явно задан допустимый перечень значений.*

В языке Delphi пользовательские типы данных определяются с помощью ключевого слова *type*. Для определения перечислимого типа используется следующий синтаксис:

```
type
  имяТипа = (значение1,…,значениеN);
```

Имя и значения перечисляемого типа определяются программистом. Имя типа (допустимый идентификатор Delphi) после определения в качестве типа в дальнейшем может использоваться наравне с обозначениями любых других типов данных. В языке Delphi принято начинать имена типов данных с буквы T (от слова Type — тип), чтобы различать от имен переменных.

Например:

```
type
  TAnimal = (Cat, Dog, Hamster);
```

Здесь определен тип данных TAnimal. В состав этого типа входят допустимые значения Cat, Dog и Hamster. Перечисления представляет собой тип данных, определяемый пользователем, значения которого задаются с помощью списка. Каждое такое значение рассматривается как константа перечисляемого типа. Компилятор использует внутреннее представление значений константы перечисляемого типа в виде целых чисел: 0, 1, 2, 3, …. Такое определение называется **явным определением** пользовательского типа.

После объявления типа можно определять переменные, имеющие этот тип, с помощью ключевого слова **var**. Например,

```
var
  MyAnimal: Tanimal;
```

Пользовательский тип можно также определять **неявно**, например, явное определение перечисления можно задать и так образом:

```
var
  MyAnimal: (Cat, Dog, Hamster);
```

Это второй способ ввести в употребление нужный новый тип, который состоит в том, что задание этого типа помещается непосредственно в описании соответствующих переменных. Поскольку в этом случае введенному в употребление типу не дано какого-либо имени (этот тип является «безымянным»), то на него уже невозможно сослаться в других местах программы. По этой, в частности, причине предпочтение следует отдавать первому из указанных выше способов определения типов, т.е. вводить в употребление новый тип с помощью его описания в разделе типов. Предварительное описание типа обеспечивает также значительно более высокую наглядность программы и простоту ее понимания.

Существует полезный совет: создавать пользовательские типы явно с помощью ключевого слова type. Это обеспечивает совместимость типов переменных и предотвратит ошибки компиляции. Например, если возможные значения типа определены, как указано выше (с помощью var), в переменной при ее объявлении, то нельзя будет ввести другую переменную с теми же значениями. Если такая потребность имеется, то перечислимый тип надо определять именно как тип (с помощью type). Тогда можно ввести в программе несколько переменных этого типа.

Можно считать, что базовый тип данных Boolean является перечислимым, хотя на уровне физической реализации это, скорее всего, не так:

```
type
  Boolean = (False, True);
```

Явные числовые значения в соответствие значениям перечислимых типов по умолчанию не ставится. Однако стандартная функция Ord, применимая, как уже говорилось, ко всем базовым типам, вернет порядковый номер значения в списке, начиная с нуля. Так, Ord(Cat) и ord(False)

дут значение 0, а Ord(Hamster) вернет значение 2. Порядок следования значений в описании типа может быть проверен базовыми функциями Pred и Succ для взятия предыдущего и последующего элементов: обращение Pred (Dog) даст значение Cat, а Succ (Dog) даст значение Hamster.

При желании можно задавать конкретные числовые величины для каждого перечисляемого значения. Для этого следом за наименованием значения записывается символ «=» и вводится целое число:

```
type
  TAnimal = (Cat = 1, Dog = 2, Hamster = Dog+4);
```

Если некоторые значения опущены, то они нумеруются, начиная с нуля. Так, в случае описания:

```
type
  TAnimal = (Cat = 1, Dog, Hamster);
```

идентификатор Dog будет эквивалентен значению 0, а Hamster — значению 1 (как и Cat). Соответственно, в рамках типа TAnimal значения Cat и Hamster будут равны.

Итак, перечисление позволяет определять переменные, принимающие фиксированный перечисляемый набор значений.

Задача:
Сотрудники отдела имеют следующие зарплаты: Иванов, Петров – 6000с., Сидоров – 5000с., Федоров, Гаврилов по 4000с. Составит программу для определения суммарной зарплаты сотрудников отдела.

Решение.
При составлении программы будем использовать перечислимый тип.

```
Program P7;
.....
   // сначала объявляем новый тип – тип перечисления
 type
   Tfam =(Ivanov, Petrov, Sidorov, Fedrov, Gavrilov);
 var
    S: real;
    I: Tfam;  // потом объявление переменной этого типа
 begin
   S := 0;
   for I:= Ivanov to Gavrilov do //использование этой переменной
     case I of
        Ivanov, Petrov: S :=  S + 6000;
        Sidorov:        S :=  S + 5000;
        Fedorov, Gavrilov:       S :=  S + 4000;
     end;
  writeln ('Summa =', S:6:0);
  readln
end.
```

В данном примере счетчик цикла является переменной типа перечисления, т.е. порядкового типа. К порядковым типам относятся типы: целые, логические, символьные, перечисления, диапазоны.

Применение переменных типа перечисления повышает наглядность программы и позв автоматически контролировать значения переменных. Так, например, переменная I в программе может принимать только одно из пяти заданных значений.

Ограниченные типы (типы диапазона)

Тип диапазон можно определить, накладывая ограничения на уже определенный некоторый порядковый тип – его называют базовым типом. В любом из базовых типов можно выделить диапазон значений: от начального до конечного (начальное значение должно быть меньше конечного).

Ограниченный тип *— это пользовательский тип данных, для которого явно задан диапазон допустимых значений.*

Диапазон выделяют в отдельный тип данных:

```
type
 имя-типа = начальное-значение .. конечное-значение;
```

Например,

```
type
  T5tol0 = 5 .. 10;  // базовый тип Integer
type
  Tabcdefgh = 'a'.. 'h'; // базовый тип Char
```

Диапазоны можно строить и на основе пользовательских перечислимых типов:

```
type
 TAnimal = (Cat, Dog, Hamster, Mouse);
type
 TMyAnimal = Dog .. Mouse; //базовый тип – тип Tanimal
```

Тип TMyAnimal охватит значения Dog, Hamster и Mouse.

Границы диапазонов могут быть заданы выражениями. Однако компилятор Delphi выдает ошибку, если после знака равенства в объявлении типа следует скобка, поскольку компилятор посчитает, что описывается перечислимый тип:

```
type
  T4tol0 = (1+1)*2 .. 10; //  ошибка
```

Правильно перезаписать это определение так:

```
type
   T4tol0 = 2*(1+1)..10;   //   корректно
```

Диапазоны позволяют строго контролировать набор значений, принимаемых определенной переменной. Если по условиям задачи известно, что переменная X может принимать только значения от -100 до +100, то корректнее не описывать ее как переменную типа Integer, а предварительно сформировать оригинальный тип данных:

```
type
 T100 = -100..100;
```

Теперь переменную X можно объявить как переменную типа T100. Если теперь в ходе работы программы произойдет попытка записи в X значения, большего + 100 или меньшего -100, возникнет ошибка. При использовании же базового типа integer выявить такую ошибку не удастся, и впоследствии неверный ход расчетов приведет к непредсказуемым последствиям. В компиляторе имеется опция, позволяющая включить проверку диапазона при присваивании значения переменной ограниченного типа. Это опция {$R+}. Вы можете ее включить в том месте вашей программы, где хотите начать проверку диапазона, и выключить проверку, где захотите, опцией {$R-}.

Задача. Составить программу для вычисления среднего балла, полученного абитуриентом на вступительных экзаменах по математике, физике и химии.

Будем использовать тип-диапазон для контроля вводимых оценок.

```
Program P8;
…….
{$R+}    // включение проверки диапазона значений оценок при вводе
uses SysUtils;
type Tball = 2..5; // определение диапазона значений оценок
var math, phiz, chem: Tball; //наложение ограничений на оценки
    ave: real;
begin
   readln(math, phiz, chem); //ввод оценок с соблюдением ограничений
   ave := (math + phiz + chem) / 3;
   writeln(ave:5:2);
   readln
end.
```

Резюме

Использование перечислимых типов упрощает работу с данными, поскольку дает возможность работать не с абстрактными числами, а с осмысленными значениями.

Использование ограниченных типов позволяет формулировать алгоритм решения задачи в более понятной и наглядной форме.

Вопросы для самопроверки

1. Какие типы относятся к пользовательским типам?
2. Что значит тип стандартный?
3. Чем отличается стандартные и пользовательские типы?
4. Как формируются значения перечислимых типов?
5. Как формируются значения типов диапазонов?
6. Как в Delphi определить свой собственный тип данных?
7. Какие типы относятся к порядковым типам?
8. Преимущества использования перечисления и диапазона?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 10. Массивы

План
Решение задачи, без использования массива.
Статические массивы
Операции, допустимые над массивами
Динамические массивы
Передача массивов подпрограммам
Строка как массив символов

Решение задачи, без использования массива.

Рассмотрим следующую задачу. Допустим, 5 студентов компьютерных курсов сдают экзамен. Инструктор дал им задание создать программу, выводящую на экран среднюю оценку экзамена и фразу, описывающую оценку каждого студента относительно средней (пусть это будет "Выше среднего", "Средне" и "Ниже среднего"). С помощью 5 переменных вещественного типа (простых структур данных) можно составить программу:

```
program Project10_a;
{$APPTYPE CONSOLE}
Uses
  SysUtils;
var
  score1, score2,score3,score4,score5,scoreavg:real;
procedure compare(student:integer; sc,avg:real);
begin
  if sc<avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
    'ниже чем средней')
  Else
  if sc>avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
    'выше чем средней')
  Else
    writeln(' Студент  ',student,' моя оценка ',sc:5:2,
    'равна средней')
end;
begin
  write('введите оценку 1: '); readln(score1);
  write(' введите оценку  2: '); readln(score2);
  write(' введите оценку  3: '); readln(score3);
  write(' введите оценку  4: '); readln(score4);
  write(' введите оценку  5: '); readln(score5);
  scoreavg:=(score1+score2+score3+score4+score5)/5;
  writeln('Средняя оценка: ', scoreavg:5:2);
  compare(1,score1,scoreavg);
  compare(2,score2,scoreavg);
  compare(3,score3,scoreavg);
   compare(4,score4,scoreavg);
  compare(5,score5,scoreavg);
  readln;
end.
```

Обратите внимание, что отдельная переменная должна быть объявлена для опенки каждого студента. Для ввода опенок студентов должны быть созданы также 5 отдельных полей ввода

Рассмотренный выше метод применим только для небольшого количества студентов: 10 студентах исходный код получился довольно громоздким. А что будет при 100 студентах? А что делать, если количество студентов не постоянно? Delphi предоставляет лучший способ решения таких задач, состоящий в использование массивов – более сложных структурированных типов данных.

Массив представляет собой последовательность индексированных данных ***одного и того же типа***. Каждый элемент массива имеет уникальный индекс.

Массив с одним индексом называется **одномерным.** Математическим эквивалентом одномерного массива является *вектор*. **Двухмерный массив** имеет два индекса; его математический эквивалент — *матрица.* Существуют также **многомерные массивы** (n-мерные). Количество индексов многомерного массива больше двух.

В большинстве языков программирования массивы записываются аналогично тому, как это принято в математике. Например, i-й элемент вектора **x** принято записывать в математике как $\mathbf{x}_i$, а i-й элемент массива **x** в Delphi — как **x[i]**. Редактор кода Delphi использует стандартный набор символов ASCII, в который не входят нижние индексы, поэтому индекс вектора заменен выражением в квадратных скобках.

Статические массивы

Статический массив представляет собой фиксированное количество упорядоченных однотипных элементов, снабженных индексами. Чтобы задать статический массив, используется ключевые слова ***array of***. Синтаксис объявления N-мерного статического массива имеет вид:

```
Type
   Имя_типа =  array [тип_индекса1, …, тип_индексаN ] of  базовый_тип;
```

Например,

```
type
  vector = array[1..10] of Integer;
  table = array [1..10, 1..5] of String;
```

Здесь vector – имя статического одномерного массива, состоящий из десяти целых чисел, table – имя статического двумерного массива, состоящий из десяти строк и пяти столбцов.

После ввода типа-массива, можно задавать переменные этого типа. Так, для описанных выше типов можно задать, например, следующие переменные:

```
var
  m1, m2: vector;
  matr: table;
```

Если использовать второй способ задания пользовательского типа, то синтаксис объявления N-мерного статического массива будет следующим:

```
var
  имя_массива:  array [тип_индекса1, …, тип_индексаN ] of  базовый_тип;
```

Тогда вышеприведенные примеры запишутся следующим образом:

```
var
  m1, m2: array[1..10] of Integer;
  matr: array [1..10, 1..5] of String;
```

Обратите внимание, что в синтаксисе квадратные скобки являются символами объявления массива. Имя массива может быть любым правильным идентификатором. Базовый тип — это тип, назначаемый каждому элементу массива. Каждый индекс должен иметь ***порядковый тип.*** Общее количество элементов массива равно произведению количества элементов по каждому

индексу. В качестве типа индекса чаще всего используется целый диапазон, однако типом индекса может быть также, например, перечислимый тип.

Для одномерного массива в объявлении используется только один тип индекса, поэтому синтаксис объявления одномерного массива имеет вид

```
var
 имя_однамерного_массива:  array [тип_индекса]  of базовый_тип;
```

Например, приведенный ниже оператор объявляет массив sampleArray, содержащий 10 чисел типа Integer, причем значение индексов изменяется от 1 до 10.

```
var
  sampleArray: array[1..10]  of  Integer;
```

Для индексации массива могут использоваться любые ограниченные типы. Например, вы можете определить массив следующим образом:

```
type color = (red, yellow, green) ;
var ACol : array [color] of integer;
```

В качестве типа индекса вы задали введенный вами перечислимый тип. И тогда, например, ко второму элементу массива вы можете обратиться как ACol[yellow].

В следующем фрагменте кода объявляются двухмерный и многомерный массивы:

```
type
    Year = (Freshman, Sophomore, Junior, Senior);
var
    smp2DArray: array[1..10, 1..5] of Integer;
    smp4DArray: array[Year, 1..20, 'A'. .'F', 1..3] of Char;
```

Двухмерный массив можно представить как массив массивов. Аналогично этому многомерный массив — это массив массивов, имеющих размерность на единицу меньше (например, трехмерный массив — это массив массивов массивов). Поэтому следующие объявления эквивалентны предыдущему фрагменту кода:

```
type
  Year =  (Freshman,  Sophomore,  Junior,  Senior);
var
  smp2DArray: array[1..10] of array[1..5] of Integer;
  smp4DArray: array[Year] of array[1..20] of  array['A'..'F'] of array[1..3] of Char;
```

В обоих случаях массив smp2DArray содержит 10*5=50 элементов типа Integer, smp4DArray — 4*20*6*3 = 1440 элементов типа Char. Выбор способа массива определяется стилем программирования. Однако помните: ваш стиль программирования должен быть неизменным на протяжении всего периода разработки программы.

Каждый элемент массива фактически является переменной базового типа, переменной (т.е. к элементу массива) можно обращаться с помощью набора индексов. Для получения доступа к элементу массива можно использовать одну из следующих форм.

```
имя__массива [значение_индекса1, ..., значение_индексаN]
```

ИЛИ

```
имя__массива [значение_индекса1] ... [значение_индексаN]
```

Например, следующий оператор присваивает значение 10 пятому элементу массива sampleArray:

```
sampleArray[5] : = 10;
```

Как и в случае двух разных способов объявления массивов, операторы

```
DArray[5, 2] := 7;
smp4DArray [Junior, 10, 'A', 1] := 'X';
```
эквивалентны операторам
```
smp2DArray[5][2] := 7;
smp4DArray[Junior][10]['A'][1] := 'X';
```

Операции, допустимые над массивами

Используя массивы в программах нужно определить значения его элементов, потом выполнить на них нужные действия и вывести результаты. При этом используют оператор цикла.

Рассмотрим массивы:

```
type
  t1 = array[1..10] of Integer;
  t2 = array [1..10, 1..5] of String;
var
   z1: t1;
   z2: t2;
```

Типичными операциями допустимыми над массивами считаются:

1 Ввод элементов массива

```
for i:=1 to 10 do
   Readln(z1[i]);       //одномерный массив
for i:=1 to 10 do
     for j:=1 to 5 do
        z2[i,j]:='ala';  //двухмерный массив
```

1. Вывод элементов массива

```
for i:=1 to 10 do
   Write(z1[i]:5);
for i:=1 to 10 do
   for j:=1 to 5 do
      Writeln(`z2[`, i, ` , `,j,']=', z2[i,j]);
```

2. Поиск самого большого (маленького) числа в массиве

```
max:=z1[1];
for i:=2 to 10 do if z1[i]>max then max:=z1[i];
min:=z1[1];
for i:=2 to 10 do if z1[i]< min then min:=z1[i];
```

3. Копировка одного массива в другой (такого же типа и размера)

```
var   z3: t1;
…..
z3:=z1;
```

4. Замен местами двух элементов массива (например при сортировке)

```
var   zp: Integer;
….
zp:=z1[i];
z1[i]:=z1[i+1];
z1[i+1]:=zp;
```

Полезный совет.

Если статический массив является формальным или фактическим параметром либо создается больше одного массива данного типа, то используйте для создания массива ключевое слово *type*. Проиллюстрируем это на примере. Приведем следующий фрагмент кода:

```
Var
  intArrayl: array[1..10] of Integer;
  intArray2: array[1..10] of Integer;
  count:          Integer;
begin
  for count := 1 to 10 do begin intArrayl[count] := count; end;
  intArray2 := intArrayl;//неправильно
end;
```

Для определения эквивалентности типов переменных компилятор Delphi использует имена типов. Поэтому типы массивов intArrayl и intArray2 **несовместимы.** Эти массивы имеют одно и то же физическое строение, их **типы разные**. Поэтому компилятор отметит ошибку несовместимости типов в операторе fntArray2 := intArrayl. Чтобы исправить ошибку, нужно записать этот фрагмент кода следующим образом:

```
Type
  TenInts = array[1..10] of Integer;
Var
  intArrayl: TenInts;
  intArray2: TenInts;
  count:          Integer;
begin
  for count := 1 to 10 do begin intArrayl[count] := count; end;
  intArray2 := intArrayl; //правильно
end;
```

Предыдущий пример, в котором вычисляется средняя оценка студентов можно решить с использованием массивов следующим образом:

```
program Project10_b;
{$APPTYPE CONSOLE}
Uses
  SysUtils;
var score:array[1..5] of real;
    scoreavg:real;
    i:integer;
procedure compare(student:integer; sc,avg:real);
begin
  if sc<avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
    'ниже средней')
  Else
    if sc>avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
       'выше средней')
    Else
    writeln('Студент ',student,' моя оценка ',sc:5:2,
      'равна средней')
end;
begin
scoreavg:=0;
for i:=1 to 5 do
begin
  write('введите оценку ', i,': '); readln(score[i]);
  scoreavg:=scoreavg+score[i];
```

```
scoreavg:=scoreavg/5;
writeln('Srednia ocen: ', scoreavg:5:2);
for i:=1 to 5 do
begin
  compare(i,score[i],scoreavg);
end;

readln;

end.
```

Теперь, когда вы знакомы с массивами, рассмотрим, что полезного можно сделать с их помощью. Разработаем программу несложной адресной книги. Ниже приведено объявление констант, типов и переменных такой программы. Для простоты будем использовать статические массивы, а максимальное количество хранимых адресов будем полагать равным 50.

```
const
  MAX_RECORDS = 50;
type
  DataField = array[1..MAX_RECORDS] of String;
var
  firstName: DataField; {Имя}
  lastName:  DataField; {Фамилия}
  address:  DataField;   {Улица и номер дома}
  city:   DataField;       {Город}
  state:  DataField;       {Штат}
  zipCode:   DataField;   {ZIP-код}
  phoneNumber:  DataField;  {Номер телефона}
  number: Integer;{Количество адресов в адресной книге}
```

Объявление каждой переменной типа DataField создает массив, содержащий данные одного вида. В одном массиве хранятся имена, в другом — фамилии и т.д. Кроме переменной number хранится количество адресных карточек, формируемых программой. Максимальное количество адресных карточек определяется значением константы MAX_RECORDS.

Программа адресной книги, разработанная на основе объявленных таким образом структура данных, работает правильно только при полном соответствии значений индексов. Другими словами, человек с именем firstName[1] и фамилией lastName[1] может жить только в городе city[1], в штате state[1] и т.д. Аналогично вся другая информация, относящаяся к этому человеку, может храниться только в 1-м элементе значение индекса равно 1) любого из массивов. Массивы, хранящие информацию в таком формате, называются параллельными массивами. Если данные окажутся отсортированными неправильно или значение индекса по какой-либо причине исказится (вследствие программной ошибки или ошибки ввода-вывода), то данные всей адресной книги будут испорчены. Поэтому надежность такой структуры данных невелика. Лучше использовать записи (лекция 11).

Динамические массивы

Изменим программу вычисления средней оценки таким образом, чтобы количество студентов могло быть произвольным. Один из способов решения такой задачи состоит в объявлении статического массива, размер которого был бы достаточным для приемлемого количества студентов. Однако такой способ приводит к неэкономному расходованию памяти. К тому же может оказаться, что количество студентов все-таки превышает размер массива, и тогда

ется опять изменять исходный код. Более оптимальное решение — использо **динамического массива.**

Для объявления динамического массива используется оператор array of без задания индексов. Синтаксис объявления одномерного динамического массива имеет вид

```
var
имя_динамического_массива: array of базовый_тип;
```

Задание размера массива и выделение для него памяти выполняется с процедуры ***SetLength ( )*.**

SetLength*(имя_динамического_массива, длина);*

В этом синтаксисе выражение *длина* должно быть целого типа. После такого оператора значение индекса динамического массива может изменяться от 0 до *длина-1*. Процедуру SetLength () в процессе выполнения программы можно вызывать произвольное количество раз. Каждый вызов приводит к изменению длины массива, причем содержимое массива сохраняется. Если при вызове SetLength () **длина массива** увеличивается, то добавленные элементы заполняются произвольными значениями, так называемым **мусором.** Если длина массива уменьшается, то содержимое отброшенных элементов теряется. Например:

```
program tablice_dyn;
{$APPTYPE CONSOLE}
uses SysUtils;
var t : array of integer;
   n,i:integer;
begin
  Readln(n);
  SetLength(t,n);
  for i:=0 to n-1 do t[i]:=i;
  SetLength(t,2*n);
  for i:=n to 2*n-1 do t[i]:=i;
  for i:=0 to 2*n-1 do write(t[i]);
  readln
end.
```

Динамические массивы являются динамическими структурами данных, поэтому по окончании работы с ними в программе должно быть предусмотрено их явное удаление из памяти компьютера. Процесс удаления ненужных динамических переменных из памяти компьютера иногда называют сборкой мусора.

В Delphi программисту предоставлены три метода удаления динамических массивов:

1. Путем установки нулевой длины динамического массива:
   SetLength (имя динамического_массива, 0);
2. Путем присвоения массиву (не его элементам, а индексной переменной, носящей имя массива) значения ***nil***. В Delphi зарезервированное слово ***nil*** используется в качестве значения индексных (указательных) переменных. Если указательная переменная имеет значение nil, то она не указывает ни на какой объект. Таким образом, удалить динамический массив из памяти можно, воспользовавшись оператором
   имя_динамическдго_массива := nil;
3. С помощью встроенной процедуры ***Finalize ():***

Finalize (имя_динамического_массива);

Для работы с динамическими массивами Delphi предоставляет еще несколько полезных функций. Функция ***copy ()*** возвращает заданную часть динамического массива:

Copy(имя_динамического_массива,начальное_значение_индекса,
количество_копируемых_элементов);

Функции ***High()*** и ***Low()*** возвращают наибольшее и наименьшее значения. динамического массива, т.е. High() возвращает значение *длина-1,* а Low() — значение 0. Если динамический массив имеет нулевую длину, то функция High() возвращает -1. Синтаксис этих функций имеет вид

High (имя_динамического_массива) ;
Low (имя_динамического_массива);

Для вычисления средней произвольного количества студентов можно использовать динамический массив:

```
program Project10_c;
    {$APPTYPE CONSOLE}
uses
  SysUtils;

var score:array of real;
    scoreavg:real;
    numStudents, i:integer;

procedure compare(student:integer; sc,avg:real);
begin
  if sc<avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
    'ниже средней')
  else
  if sc>avg then writeln('Студент ',student,' моя оценка ',sc:5:2,
    'выше средней')
  else
    writeln('Студент ',student,' моя оценка ',sc:5:2,
    'равна средней')
end;

begin
  scoreavg:=0;
  write('введите кол-во студентов'); readln(numStudents );
  SetLength(score,numStudents );
  for i:=0 to numStudents -1 do
  begin
     write('введите оценку ', i+1,': '); readln(score[i]);
     scoreavg:=scoreavg+score[i];
  end;
  scoreavg:=scoreavg/numStudents ;
  writeln('Средняя оценка: ', scoreavg:5:2);
  for i:=0 to numStudents -1 do
  begin
    compare(i+1,score[i],scoreavg);
  end;
  readln;
end.
```

Мы рассмотрели создание и использование одномерного динамического массива. Но как создать многомерный динамический массив? Как вы помните, многомерный массив — это не что иное, как массив массивов. Поэтому синтаксис объявления: с несколькими индексами имеет вид

```
var
имя_мыогомерного_динамического_массива:array of array of  ...  array of базовый_тип;
```

Задание размера многмерного массива и выделение для него памяти выполняется с процедуры ***SetLength ()*.**

rLength(имя_динамического_массива, длина, длина,...);

Например:

```
var
  smp2DdynArray: array of array of Integer;
  count:       Integer;
begin
   SetLength(smp2DdynArray, 3);
   for count := 1 to 3 do begin
     SetLength(smp2DdynArray[count -1], count*2);
   end;
   {В этом месте массив smp2DdynArray можно использовать}
end;
```

или

```
program tablice_dyn;
{$APPTYPE CONSOLE}
uses SysUtils;
var t:array of array of integer;
     n,i,j:integer;
begin
  Readln(n);
  SetLength(t,n,n);
  for i:=0 to n-1 do
    for j:=0 to n-1 do t[i,j]:=i;
      for i:=Low(t) to High(t) do
      begin
        for j:=Low(t[i]) to High(t[i]) do write(t[i,j]);
        writeln;
      end;
  readln
end.
```

Имя динамического массива обозначает переменную, фактически являющуюся указателем. Это значит, что такая переменная содержит не данные, а адрес первого элемента массива. Говорят, что она указывает на первый элемент массива. Индексы определяют смещение требуемого элемента массива относительно первого. Рассмотрим следующий фрагмент программного кода:

```
var
  arrayl, array2: array of Integer;
  check:       Boolean;
begin
  SetLength(arrayl, );
  SetLength(array2, );
  arrayl[0] : = 5;
  array2[0] := 5;

chheck := (arrayl = array2);
.
    .
end;                                                    . .
```

Какое значение принимает переменная check? Если бы это был статический массив, то значение check было бы равно True, потому что массивы содержат одну и ту же информацию. Однако для динамических массивов (как в данном случае) значение check равно False, поскольку указатели arrayl

ay2 ссылаются на разные области памяти. Значение указателя равно адресу первого по счету мента массива, а так как эти массивы хранятся в разных местах, естественно, указатели имеют разные значения. Для сравнения динамических массивов необходимо сравнивать в цикле каждую пару элементов отдельно.

Переменная типа фактически является динамическим массивом символов.

Передача массивов подпрограммам

В двух предыдущих версиях программы вычисления средней оценки в процедуру Compare() передавался единственный элемент массива. Однако в подпрограмму можно передать и целый массив. В последней версии программы вычисления оценки в процедуру Сошраге () передается весь массив score и цикл его обработки расположен внутри Compare(). Для сравнения: в предыдущих версиях процедура Compare() вызывается в (цикле, но внутри себя цикла не содержит. Код программы, решающей эту задачу:

```
program Project10_d;
{$APPTYPE CONSOLE}
Uses
  SysUtils;
type Tsc= array of real;
var score: Tsc;
    numStudents, i:integer;
procedure compare( scoretab:Tsc);
var avg:real; i:integer;
begin
 avg:=0;
for i:=low(scoretab) to high(scoretab) do avg:=avg+scoretab[i];
avg:=avg/(high(scoretab)+1) ;
writeln('средняя оценка: ', avg:5:2);
for i:=low(scoretab) to high(scoretab) do
begin
  if scoretab[i]<avg then writeln('Студент ',i+1,' моя оценка ',scoretab[i]:5:2,
    'ниже средней')
  else
 if scoretab[i]>avg then writeln('Студент ',i+1,' моя оценка ',scoretab[i]:5:2,
  'выше средней')
 else
   writeln('Студент ',i+1,' моя оценка ',scoretab[i]:5:2,
     'равна средней')
end;
end;

begin
  write('введите кол-во студентов '); readln(numStudents );
  SetLength(score,numStudents );
  for i:=0 to numStudents -1 do
  begin
    write('введите оценку ', i+1,': '); readln(score[i]);
  end;
  compare(score);
  readln;
end.
```

В системе Delphi допускается использование ***открытых индексных параметров***, т.е. массивов без указания индексов. Например, оператор

procedure OpenArrayEx(inArray: **array of** Integer);

ляет процедуру, единственным формальным параметром которой является массив произвол длины. Использование открытых индексных параметров существенно расширяет возможности подпрограмм. Синтаксис открытого индексного параметра напоминает синтаксис динамического массива, однако работают. они по-разному. Чтобы объявить динамический массив в качестве формального параметра, для него необходимо определить пользовательский тип.

Открытые индексные параметры подчиняются следующим правилам.

1. Массив, объявленный как открытый индексный параметр, всегда начинается с нуля, т.е. индекс первого элемента массива всегда равен о.
2. Функции Low () и High () возвращают для такого массива значения 0 и *длина-1*. Функция sizeof () возвращает длину фактического массива, переданного в подпрограмму.
3. В исходном коде можно обращаться только к элементам такого массива. Обращение к целому массиву запрещено.
4. Открытые индексные параметры не могут передаваться в процедуру SetLength (). Они могут передаваться в другие подпрограммы только как открытые индексные параметры или как нетипизированные параметры var.
5. Передаваемая в подпрограмму переменная базового типа открытого индексного параметра внутри подпрограммы полагается массивом единичной длины.

Массивы могут содержать огромное количество информации. При работе с большими массивами пользователю иногда легче вводить данные в файл, а не в поля ввода, предоставляемые в пользовательском интерфейсе. Затем файл данных можно использовать как входной файл программы; Работа с файлами данных рассматривается позже.

В Delphi использование индекса в списке формальных параметров подпрограмм не допускается. Например, следующее объявление функции вызовет сообщение о синтаксической ошибке:

```
Function Example(inArray array[1..20] of Real): Real;
```

Синтаксическая ошибка возникает вследствие задания индексов параметра inArray. *Напоминаем: чтобы обойти это ограничение, следует объявить пользовательский тип данных.* Объявление этой же функции без синтаксической ошибки имеет вид

```
Type
  RealArray20 = array[1..20] of real;
Function Example(inArray:  RealArray20): Real;
```

Строка как массив символов

Строка фактически является массивом символов. С каждым символом строки ассоциировано уникальное значение индекса. Обращение к отдельному символу строки можно сделать с помощью выражения *имя_строки[i],* где *i* – индекс символа, на который вы хотите сослаться. Значение индекса первого символа строковой переменной равно 1, второго – 2 и т. д. Обратите внимание, что выражение *имя_строки[i]* имеет символьный тип, поэтому его значение можно сохранить в любой переменной символьного или строкового типа. Кроме того, выражение *имя_строки[i],* можно использовать в операторе присваивания или в качестве параметра подпрограммы. Его использование иллюстрируется в следующем примере:

```
var
  myString: String;
  myChar:   String;
  aChar:    Char;
Begin
  myString := 'This is an example' ;
  myChar := myString[13];
  myChar := UpperCase(myChar);
  aChar := myChar[1];
  myString[13] := aChar;
writeln(myString);
end;
```

индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
myString	T	h	i	s		i	s		a	n		e	x	a	m	p	l	e

Этим показана структура строковой переменной как массив символов.

Какой текст появится в результате выполнения этого кода? Правильный ответ: This is an e**X**ample'

Резюме

В Delphi *наборы значений* хранятся в массивах. Массивы представляют собой *структуры данных*, состоящие из взаимосвязанных однотипных элементов. В программировании под структурой данных программы в общем случае понимают множество элементов данных, множество связей между ними, а также характер их организованности. Массив объединяет группу ячеек памяти под одним именем и, как правило, имеющих один и тот же тип. Для обращения к определенной ячейке памяти или элементу массива следует указать имя массива и индекс или место данного элемента в массиве. Многомерные массивы с двумя индексами часто используются для представления таблиц, в которых информация организована по строкам и столбцам. Каждый элемент таблицы идентифицируется с помощью двух индексов – первый определяет строку, а второй – столбец, в котором расположен элемент.

Структура данных программы во многом определяет алгоритмы. Одна и та же задача может часто решаться с использованием разных структур данных, что было продемонстрировано в этой лекции.

Вопросы для самопроверки

1 Можете дать определение массива?
2 Как обращаются к элементам массива?
3 Какими должны быть элементы массива?
4 Отличия статических и динамических массивов.
5 Какая разница между одиночной переменной и элементом массива?
6 Отличаются ли элементы массива и строки и чем?
7 Что общего между элементами массива и строки?

Лекция 11. Записи. Множества

План
Определение записи
Операции, допустимые над записями
Определение множества
Операции, допустимые над множествами
Определение записи

Запись – *это пользовательский тип, содержащий группу связанных данных. Запись состоит из произвольного количества полей. Поле содержит неделимую единицу информации. Поля могут быть разных типов.*

Определение типа записи задает имя этого типа, а также имена и типы каждого по записи. В определении используется ключевое слово *record* (запись).
Синтаксис определения типа записи имеет вид:

```
type
имя_типа_записи = record
                    список_полей1:      тип_данных1;
                    список_полей2:      тип_данных2;
                        .......
                        .......
                    список_полейN: тип_данных;
                    end;
```

В этом синтаксисе имя_типа_записи должно быть любым правильным идентификатором. Список полей должен быть любым правильным идентификатором или списком! Параметр *тип_данных* задает тип каждого поля в соответствующем списке полей. Последняя точка с запятой перед ключевым словом end необязательна.

Примером использования этого типа данных может быть тип содержащий информацию о студенте:

```
type
  TStudentInfo = record
                   lastName:   String[25];
                   firstName:  String[15];
                   age:         Integer;
                 end;
var studentInfo: TStudentInfo;
```

Имя, фамилия и возраст студента это данные разных типов.

Другим примером может быть адресная книга, в которой каждая адресная карточка является записью., которая состоит из полей, содержащих имя, фамилию, улицу и номер дома, город, штат, ZIP-код и номер телефона.

```
type
TAddressCard = record
                    FirstName: String;      {Имя}
                    lastName: String;       {Фамилия}
                    address: String;        {Улица и номер дома}
                    city: String;           {Город}
                    state: String;          {Штат}
                    zipCode: String;        {Zip-код}
                    phoneNumber: String; {Номер телефона}
               end;
```

Адресную книгу можно определить как массив записей:

```
c
  MAX_RECORDS = 50;
Var
  DataField1 : array[1..MAX_RECORDS] of TAddressCard; {массив статический}
  DataField2 : array of TAddressCard; {массив динамический}
```

В Delphi допускается создание ***вариантных записей***, содержащих вариантную часть. Другими словами, вариантная запись содержит поля, предназначенные для разных типов данных, причем в одном экземпляре записи никогда не используются все такие поля.

Синтаксис определения типа ***вариантной записи*** имеет вид:

```
type
   имя_типа_записи= record
                          список_полей1:тип1;
                           .
                           .
                          списокполейN: типN
                             case поле_переключатель:порядковый_тип of
                                список_констант1: (вариант1);
                                                    .
                                списокконстантN:  (вариантN);
                       end;
```

Первая половина объявления (вплоть до ключевого слова ***case***) совпадает со стандартными объявлениями типа записи. Фрагмент от ключевого слова case до end содержит вариантную часть объявления. Вариантная часть обязательно должна быть расположена после объявления остальных полей.

Необязательный параметр *поле_переключатель* представляет собой любой правильный идентификатор. Если он опущен, то следует также опустить двоеточие. Если параметр *поле_переключатель* приведен, он является невариантным полем записи указанного порядкового типа. Разделенные запятыми элементы списков констант содержат константы указанного порядкового типа. Ни одно значение константы не может находиться более чем в одном списке констант.

Синтаксис варианта, ассоциированного со списком констант, имеет вид:

```
список_полей1:  тип1;
   .
   .
списокполейN:  типN;
```

Разделенные запятыми имена в списке полей должны быть правильными идентификаторами. Все поля одного списка имеют тип, указанный после двоеточия. Точка с запятой после типN не обязательна. Типами полей не могут быть длинные строки, динамические массивы, вариантные типы, а также любые структуры, содержащие перечисленные типы. Однако тип поля может быть указателем на данных этих типов.

Рассмотрим следующий пример:

```
type
EmployeeData = record
                  firstName:String[30];
                  lastName: String[30];
                  birthDate:TDate;
```

```
    phoneNumber:String[14];
      case citizen:Boolean of
        True: (birthplace: String[30];
        SSN: String[ll]);
        False: (country:  String[30];
        entryPort: String[20];
        entryDate: TDate;
        workID: String[20]);
  end;
```

Запись типа EmployeeData (данные о служащем) может содержать разные поля в зависимости от значения булева поля citizen (гражданство). Предполагается, что и поле citizen истинно, служащий является гражданином данной страны. В случае запись содержит поля, хранящие информацию о его месте рождения и номере карточки социального страхования. В противном случае запись содержит название страны, откуда он прибыл, порт въезда и т.д.

Для каждого экземпляра вариантной записи компилятор выделяет память, достаточную для хранения всех полей наибольшего варианта. Поэтому экземпляр записи содержит несколько вариантов, разделяющих одну и ту же область памяти. Более того, все поля экземпляра вариантной записи всегда видимы, т.е. к любому полю любого варианта можно обращаться в любой момент. Однако помните: если записывать данные сначала в одном варианте, а затем в другом, то можно повредить данные, хранящиеся в полях первого варианта. Более подробная информация о вариантных записях содержится в справочной системе Delphi.

Операции, допустимые над записями

При использовании записи в программе нужно определять значения его полей. Для ссылки на конкретное поле записи нужно привести имена записи и поля, разделенные точкой. Таким образом, выражение

имя_записи.имя_поля

ссылается на заданное поле в заданной записи. В качестве имени записи можно использовать также элемент массива записей или любое выражение, ссылающееся на запись.

Например, присвоить значения полям записи studentInfo можно с помощью следующих операторов:

```
studentInfo.lastName := 'Smith' ;
studentInfo.firstName := 'John';
studentInfo.age  := 19;
```

а присвоить значения полям первой записи массива DataField1 можно с помощью следующих операторов:

```
DataField1[1].lastName := 'Smith' ;
DataField1[1].firstName := 'John';
……
```

Для доступа к полям записи можно также воспользоваться оператором ***with***, который упрощает доступ к полям записи не используя точки. Синтаксис оператора ***with*** имеет вид

```
with запись do begin
                операторы;
              end;
```

Например, предыдущие фрагменты кода могут быть записаны так:

```
with studentInfo  do begin
                       lastName  :=  'Smith';
                       firstName  := 'John';
                       age  := 19;
                     end ;

with DataField1[1]  do begin
                          lastName  :=  'Smith';
                          firstName  := 'John';
                          //……
                        end;
```

Таким образом, оператор with позволяет опускать имя записи во всем блоке. Перед каждым упоминанием элемента записи, указанной в операторе with, компилятор автоматически подставляет имя записи. При этом пространство имен, видимых в программном блоке, расширяется именами элементов записи, указанной в операторе with.

Оператор with используется для квалификации не только элементов записей, но и элементов объектов (лекция 14).

Определение множества

Множество напоминает перечислимый тип, но отличается от него тем, что элементы в множестве не упорядочены. Соответственно, можно лишь проверить принадлежность элемента множеству, но нельзя узнать местонахождение элемента в множестве. Множество представляет и все допустимые подмножества, составленные из его элементов. Уже для множеств из нескольких десятков элементов число подмножеств велико и обрабатывать их сложно. В результате максимальный размер множества ограничен в языке Delphi 256 элементами.

*Тип **множеств**а — это неупорядоченный набор значений одного простого типа.*

Множества удобны тем, что можно быстро проверить их содержимое на наличие определенного значения. Если вместо множеств использовать, например, массивы, то подобная проверка потребует выполнения оператора цикла. В случае с множеством достаточно одной логической операции сравнения по битовой маске, представляющей множество (трансляцию обращений к множествам в эффективный исполнимый код выполняет, конечно, компилятор).

Множество описывается так:

```
type
  имя_множества = set of имя_базового_типа;
```

Параметр *имя_множества* должен быть правильным идентификатором Delphi. Диапазон допустимых значений элементов множества совпадает со всеми допустимыми значениями базового типа, который должен быть порядковым типом данных, например перечислимым. Множество может содержать любые значения базового типа. Оно может быть также пустым множеством, обозначаемым в Delphi как **[ ]**. Кроме того, в Delphi на размер множества наложено следующее ограничение: оно не может содержать более 256 значений. Поэтому порядковые значения базового типа должны находиться в диапазоне от 0 до 255.

; связи с этим ограничением множества иногда определяются с помощью поддиапаз обозначаемых двумя точками (..). Например, приведенный ниже оператор определяет тип множества с именем LowercaseSet, значениями которого являются буквы английского алфавита от a до z в нижнем регистре.

```
type
   LowercaseSet = set of  'a'..'z';
type
  LowercaseLetters = 'a'..'z';
  LowercaseSet = set of LowercaseLetters;
```

Эти два объявления типа множества LowercaseSet эквиваленты.

Множество чисел можно объявить так:

```
type
   Numbers =  0..255;
   NumberSet    = set of Numbers;
```

Множество перечисляемого типа можно объявить так:

```
type  MyAnimals( Cat,   Dog,   Hamster);
       MyAnimalSet   =  set of MyAnimals
```

Определив таким образом тип, можно объявлять переменные, являющиеся множествами этого типа, а затем присваивать этим переменным значения множеств. В Дельфи значение множества записывается как заключенный в квадратные скобки список его элементов, разделенных запятыми.

```
[значение1, значение2, …, значениеN]
```

Значение элементов множества, естественно, должны входить в определенный поддиапазон базового типа. Не путайте значения множества и значения элементов множества! Значение множества - это некоторый набор элементов. В следующем фрагменте кода объявляется и создается множества. Конкретные значения, входящие в множеств, задают перечислением через запятую. Список значений заключают в квадратные скобки:

```
var
    Set1: LowercaseSet;
    Set2, Set3: MyAnimalSet ;
    Set4: NumberSet;
begin
    Set1:= ['a'..'c','m','n','x'..'z'];
    Set2:=[Cat,   Cat,   Hamster];
    Set3:=[];
    Set4:=[1..9, 20..30];
end;
```

Над множествами допустимы операции объединения, пересечения, сравнения и разности.

Операции, допустимые над множествами

Подобно тому как числовые переменные ассоциированы с арифметическими операциями, множества ассоциированы с набором операций над множествами.

- + знак операции объединения множеств. В результате получается множество состоящее из неповторяющихся элементов первого и второго множеств;
- - знак операции вычитания множеств. В результате получается множество состоящее из элементов первого множества, не принадлежащие второму;

* знак операции пересечения множеств. В результате получается множество состо… из элементов общие для обоих множеств;

- = знак операции эквивалентности множеств. В результате получается true, если множество эквивалентны;
- <> знак операции неэквивалентности множеств. В результате получается true, если множество не эквивалентны;
- <= знак операции проверки вхождения. В результате получается true, если первое множество входит во второе множество;
- >= знак операции проверки включения. В результате получается true, если первое множество включает второе множество;
- **in (проверка членства)** операция которая определяет принадлежность выражения порядкового типа (первого операнда) множеству (второму операнду). Результат операции имеет тип boolean и значение True в случае, если значение принадлежит множеству.

Примеры использования множеств:

```
Type MonthDays = Set of 1. . 31;
var  Color:    Set of  (Red,  Blue,  White,  Black);
     Day:      MonthDays;
Color  :=  [Blue];
Color  := Color  -  [Blue,  Red,  White];
Color := Color +  [Black];
Color := Color +  [Black];
Day  :-  [];
Day  := Day -  [1] ;
Day  :=  [2,  4];
Day  := Day +  [5,  12,  1];
Day  := Day  -  [1] ;
```

Альтернативой операциям объединения и вычитания множеств могут послужить стандартные процедуры **Exclude** и ***Include.*** Процедура Exclude служит для удаления элемента из множества, а процедура Include — для добавления элемента. Первым параметром данных функций является переменная типа множества, а вторым — значение, тип которого определяется базовым типом элементов множества. Вот пример их использования.

```
var si:  set  of  byte;
begin
//1. Создаем небольшое множество нечетных чисел
Include(si,1);  //можно использовать оператор s1:=s1+[1];
Include(si,3);
Include(si,4);
//2. Число 4 попало в множество по ошибке
Exclude(si,4);  //можно использовать оператор s1:=s1-[4];
end;
```

Задача. Составить программу, подсчитывающую общее количество цифр и знаков ‘+’, ‘-‘,’*’, входящих в строку S.

Решение

Алгоритм решения задачи состоит в проверке на принадлежность каждого символа строки S указанному множеству.

```
Program P12;

var
  S: String;
  I, k: integer;
begin
  Writeln ('Enter S');
  Readln(s);  //  вводим строку текста, содержащую в том числе вышеназванные знаки
  k: = 0;
  For I: = 1  to  length (S)  do
    if  S [i]  in  ['0'..'9', '+', '-', '*']  then k : =  k+1;
   Writeln('k =', k: 2);
    Readln
end.
```

Резюме

Данные – непременный атрибут любой программы. Ими могут быть отдельные биты, последовательность независимых битов, числа в разных формах представления (с фиксированной или плавающей точкой, обычной или удвоенной точностью и т.д.), байты и группы независимых байтов, представляющие символы в различных системах кодирования, массивы чисел, а также информация на устройствах внешней памяти, организованная в виде отдельных файлов и систем взаимосвязанных файлов. Данные имеют разный уровень сложности, или организованности, начиная с простейшего информационного элемента – бита и кончая множеством, записью, файлами и системами файлов. Характер организованности, множество допустимых значений и набор допустимых операций над данными представляют их структуру. *Структура данного* – общее свойство любого информационного объекта, с которым имеет дело та или иная программа. Простые типы данных (кратко типы данных) – «строительный материал» для более сложных структур данных: векторов, массивов, таблиц, записей, множеств и т.д. Созданные таким образом структуры могут стать, в свою очередь, элементами еще более сложных структур. Этот процесс порождения структур произвольной сложности можно продолжать и дальше. Это процесс приводит к порождению такого сложного по структуре типа данных, определяемым программистом, как *класс.*

Вопросы для самопроверки

1. Чем отличается массив и запись?
2. Может ли запись быть элементом строки?
3. Чем отличается массив и множество?
4. Что такое тип данных?
5. Какие типы данных знаешь?
6. Почему типы данных делятся на стандартные и пользовательские?
7. Как определются типы данных?
8. Что такое ячейка оперативной памяти?
9. Как распределяются ячейки данным разных типов?
10. Что такое структуры данных?
11. Какие структуры данных вы знаете?

Лекция 12. Файлы

План
Файлы текстовые
Бинарные файлы
Файлы типизированные
Файлы нетипизированные

Компьютерные программы должны сохранять данные на диске и читать их с диска. Данные может храниться на диске в виде *файлов*. Поэтому в программе должны быть предусмотрены операции ввода и вывода файлов.

Ввод файла – это операция чтения информации из файла, хранящегося на диске. ***Вывод файла*** – это операция записи информации в хранящийся на диске файл. В Object Pascal поддерживаются три типа файлов данных**: *текстовые, типизированные и нетипизированные.***

Файлы текстовые

Текстовые файлы являются последовательными файлами. Доступ к их элементам может быть получен только последовательно, т.е. сначала предоставляется доступ к первому элементу файла, затем — ко второму и так далее до конца файла. Термин доступ означает как чтение из файла, так и запись в файл. Таким образом, процесс чтения данных из текстового файла или записи в него всегда начинается от начала файла и продолжается общем случае, до конца файла.

Создание текстового файла с помощью редактора кода Delphi

Текстовый файл состоит из ***символов ASCII***. Его можно создавать и редактировать помощью любого текстового редактора, поддерживающего сохранение файлов в форм ASCII, например с помощью редактора кода Delphi либо программ Word, WordPad, Notepad. В качестве примера создадим текстовый файл с помощью редактора Delphi. В главном меню Delphi выберите команду ***File ->New->Other..***. В раскрывшемся диалоговом окне ***New Items*** (Новые элементы) во вкладке ***New*** выделите пиктограмму ***Text*** и щелкните на кнопке ОК.

Введите в появившейся пустой вкладке следующие строки.

```
Anderson 73
Castor 82
Hartley 91
McGuire 89
Ramone 67
Swanson 78
0123456789012     Нумерация позиций
```

Не вводите последние строки в файл. Она приведена лишь для того, чтобы вы могли увидеть номера позиций символов. Если сохранить эту строку, то файл хранения будет содержать некоторые дополнительные символы, невидимые в окне редактора кода. Они называются управляющими символами. Если бы управляющие символы были видны, то в окне редактора кода отображалось бы следующее.

```
Anderson 73<CR><LF>
Castor 82<CR><LF>
Hartley 91<CR><LF>
```

Mcintire 89<CR><LF>
Ramone 67<CR><LF>
Swanson 78<CR><LF>
<EOF>

Каждая пара угловых скобок с заключенными в них буквами обозначает управляющий символ. Символ <CR> (значение ASCII равно 13) обозначает возврат каретки, а символ <LF> (значение ASCII равно 10)- переход на новую строку. Символ <EOF> (***end of file***-конец файла)не позволяет курсору редактора переместиться за пределы введенного текста.

Работа с текстовыми файлами

Теперь, когда вы можете создать текстовый файл и знаете формат хранящихся в нем данных, приступим к изучению операторов языка Delphi предназначенных для ввода-вывода данных в файл. Чтобы получить доступ к файлу, его нужно открыть. По завершении всех необходимых операций с файлом его нужно закрыть.

Для получения доступа к текстовыми файлу нужно сначала объявить переменную типа ***TextFile***, которая называется дескриптором файла.

```
Var myFile : TextFile;
```

В дескрипторе хранится указатель файла, который похож на курсор в текстовом редакторе. Как и курсор, указатель файла обозначает текущую позицию в открытом текстовом файле. В режиме ввода указатель файла определяет следующий элемент данных, который будет считан из файла. В случае текстового файла таким элементом данных является символ.

В режиме вывода указатель файла определяет позицию, в которую будет записан следующий элемент данных.

Объявив дескриптор файла, нужно связать его с файлом данных с помощью процедуры ***AssignFile(),*** синтаксис которой имеет вид

AssignFile(дескриптор_файла, имя_файла) ;

Например

```
AssignFile(myFile, 'data.txt');
```

Строковое выражение имя_файла должно содержать любое правильное имя файла Windows. Если файл находится не в текущем каталоге, в выражении имя_файла должен быть указан его полный путь, включая имя диска и всех подкаталогов.

И наконец, для получения доступа к файлу его нужно открыть. Текстовый файл можно огкрыть или, для ввода, или для вывода, но не для обеих операций, одновременно.

Процедура ***Reset()*** открывает или повторно открывает текстовый файл в режиме ввода, а процедуры ***Rewrite()*** и ***Append()*** открывают или повторно открывают текстовый файл в режиме вывода. Синтаксисы этих процедур имеют вид

Reset(дескриптор_файла);
Rewrite(дескриптор_файла);
Append(дескриптор_файла);

Процедура ***Reset()*** открывает существующий файл данных, ассоциированный с заданным дескриптором, и устанавливает указатель файла в его начало. Если файл уже открыт сначала он закрывается, а затем открывается повторно. Если файла данных с указанным именем не существует, то генерируется ошибка *file not found* (файл не найден)

Процедура ***Rewrite()*** создает новый файл данных с именем, ассоциированным с дескриптором, и устанавливает указатель файла на его начало. Если файл с этим именем шествует, то он удаляется и вместо него создается новый файл с этим же именем, уже открыт, то он закрывается и

ется, а вместо него создается новый файл. Таким образом, процедура Rewrite() или со… новый файл, или записывает существующий файл заного.

Для добавления данных в конец файла используется процедура ***Append().*** Она открывает существующий файл, имя которого ассоциировано с дескриптором, и устанавливает указатель файла в конец файла. Если файл уже открыт, то процедура. Append() закрывает его, а затем открывает повторно. Если файла с этим именем, не существует, то генерируется ошибка *file not found*

Для вариантного открытия файла процедурой ***Rewrite()*** или ***Append()*** используют директивы компилятора:

{$I-} – включена контроль ошибок ввода, вывода

{$I+} - выключена контроль ошибок ввода, вывода.

В зависимости от того есть внешний файл или нет открытие мщжно осуществить следующим образом:

```
AssignFile(myFile, 'data.txt');
{$I-}
APPEND(myFile);
{$I+}
IF IOResult<>0 THEN REWRITE(myFile);
```

При этом ***IOResult*** - это функция без параметров, которая принимает значение 0 если нет ошибки и другое целое число если есть ошибка.

Когда текстовый файл открыт в режиме чтения, из него можно читать данные с помощью. процедуры ***Read (),*** синтаксис которой имеет вид

Read(дескриптор_файла, переменная)

Процедура ***Read ()*** выполняет следующие операции:

1. Считывает из файла, ассоциированного с дескриптором, порцию данных (слово, число), на которую показывает указатель файла. Обычно порции данных отделены друг от друга символами-разделителями (пробел, символ табуляции, символ перехода на новую строку).
2. Сохраняет считанные данные в переменной,
3. Перемещает указатель файла на следующую порцию данных.

Если, например, в текущей позиции файла хранится целое число, то оно должно быть считано в переменную целого типа. Присутствующие в текстовом файле символы-разделители отделяют друг от друга числа, записанные в десятичном формате. Таким образом, процедура Read() одновременно считывает и преобразует числовые данные из десятичного формат в двоичный.

Несколько вызовов процедуры Read () можно объединить в один вызов. Например, группа операторов

```
Read (myFile, variablel);
Read(myFile, Variable2);
Read(myFile, variable3);
```

эквивалентна одному оператору

```
Read(myFile, variablel, variable2, varlable3);
```

Общий синтаксис процедуры ***Read()*** имеет вид

Read (дескриптор_файла, переменная1 [,переменная2, …]);

В отличие от Read (), процедура ***Readln ()*** читает данные с новой строки. Синтаксис процедуры Readln () имеет вид

Readln(дескриптор_файла, переменная);

Процедура ***Readln()*** работает аналогично Read(), но Readln() начинает читать данные не с позиции, заданной указателем файла, а со следующей новой строки. С помощью одного вызова процедуры Readln() можно прочитать несколько переменных причем они не обязательно должны

диться в одной строке. Если строка закончилась, а список переменных еще не исчерпал начинают считываться переменные из сле- дующей строки. Синтаксис такого вызова имеет вид

Readln(дескриптор_файла, переменная1, [, переменная2, ...]);

В отличие от Read (), несколько вызовов Readln {) не эквивалентны одному вызову с объединенным списком переменных. Например, группа операторов

```
Readln(myFile, variablel);
Readln(myFile, variable2);
Readln(myFile, variable3);
```

не эквивалентна одному оператору

```
Readln(inyFile, variablel, variable2, variable3);
```

Это объясняется тем, что один оператор переходит на следующую строку, только если в текущей закончились переменные, а при использовании трех операторов переход на новую строку происходит при каждом вызове Readln ().

Если с помощью процедуры Rewrite() или Append() файл открыт в режиме вывода, то в него можно записывать данные с помощью процедуры ***Write(),*** общий синтаксис которой имеет вид

Write(дескриптор_файла [, выражение [: минимальная_ширина
[: длина_дробной_части]]]);

Параметр выражение может иметь любой числовой или строковый тип, а минимальная__ширина и длина__дробной_части должны иметь целочисленный тип. Необязательный параметр минимальная_ширина задает количество символов текстового файла, отведенных для выводимого выражения. Если длина выводимого выражений меньше указанной, то процедура Write() дополняет его пробелами слева, а если больше, то в файл выводится больше символов, чем задано выражением минималъная_ширина, т.е. выводятся все необходимые символы. Если выводится число вещественного типа, то необязательный параметр длина_дробной_ части задает количество цифр десятичной точки. С помощью одного вызова процедуры Write () в файл можно вывести произвольное количество выражений (включая нулевое). Выводимые выражения отделяются друг от друга запятыми. Например, оператор

```
Write(myFile, 10:5, 10.47589:8:2);
```

Выводит в текстовый файл, ассоциированный с дескриптором myFile, следующий текст: ___10____10.48

Встроенный в DeIphi компилятор Object Pascal поддерживает два стандартных дескриптора текстовых файлов: ***Input*** и ***Output***. Файлом дескриптора ***Input***, работающим в режиме чтения,¬ считается стандартное устройство ввода операционной системы (обычно это клавиатура). Файлом дескриптора ***Output***., работающим в режиме записи, является стандартное устройство вывода операционной системы (обычно эта консоль MS DOS). Перед запуском консольного приложения файлы, ассоциированные с дескрипторами Input и Output, автоматически открываются. Это эквивалентно выполнению следующих операторов:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Любая попытка доступа к файлам, с помощью дескрипторов input и Output из приложения Windows (не консольного) генерирует сообщение об ошибке ввода-вывода.

Некоторые встроенные подпрограммы ввода-вывода текстовых файлов не требуют явного дескриптора файла. Если дескриптор опущен, то по умолчанию предполагается, что дескриптором является Input для процедур ввода и Output для.процедур вывода. Например, вызов Read(value) эквивалентен вызову Read(Input, value). а Write(value) – вызову Write(Output, value).

процедуры ***Writeln()*** работает аналогично Write(), но после вывода всех указанных в ее списке выражений Writeln() записывает в файл управляющие символы возврата каретки и начала новой строки (<CR><LF>). Например, операторы

```
Write(myFile, 'Hello ');
Write(myFile, 'and Good-bye');
Writeln(myFile); {Переход на новую строку}
Writeln(myFile, 'Hello ');
Write(myFile, 'and Good-bye');
```

Выводят в файл следующий текст:

```
Hello and Good-bye
Hello
and Good-bye
```

Как в случае Read() и Readln() операторы Write () можно объединять, однако объединение операторов Writeln () приведет к объединению строк в результирующем файле. Следовательно, предыдущий пример можно переписать так:

```
Write(myFile, 'Hello ', 'and Good-bye');
Writeln(myFile); {Переход на новую строку}
Writeln(myFile,. 'Hello ');
Writeln(myFile, 'and Good-bye');
```

Если программа закончила работу с файлом, его необходимо закрыть. Процедура ***CloseFile ()*** разрывает связь дескриптора с файлом, возвращая эти ресурсы системе. Если файл был в режиме вывода, то процедура CloseFile () перед закрытием файла записывает в его конец символ конца файла <EOF>. Синтаксис процедуры CloseFile() имеет вид

CloseFile (дескриптор_файла);

```
CloseFile(myFile);
```

Подпрограммы обработки текстовых файлов

В Delphi предусмотрены встроенные функции для обработки текстовых файлов:

Eof()	Проверяет, где находится указатель файла: в конце файла или за его пределами
Eoln()	Проверяет, находится ли указатель файла в конце строки
SeekEof()	Возвращает True, если до конца файла остались только символы-разделители
SeekEoln()	Возвращает True, если до конца текущей строки остались только символы-разделители
Append()	Открывает существующий файл для добавления текста в его конец
AssignFile()	Связывает имя файла с дескриптором
AssignPrn()	Присваивает дескриптор текстового файла принтеру
CloseFile()	Разрывает связь между файлом и дескриптором
Erase()	Удаляет файл
Flush()	Очищает буфер текстового файла, открытого для вывода
Read()	Читает данные из файла
Readln()	Читает данные из файла с новой строки
Reset()	Открывает существующий файл для чтения
Rewrite()	Создает и открывает новый файл
SetTextBuf()	Присваивает текстовому файлу буфер ввода-вывода
Write()	Записывает данные в текстовый файл
Writeln()	Записывает в текстовый файл данные и символы конца строки

Функция **Eof()** (*end-of-file* — конец файла) возвращает булево значение, сообщающее о том, достигнут ли конец файла а в режиме чтения. Если значение Eof (дескрилтор_файла) равно True, значит указатель файла находится за последним символом. Функция Eof () используется в в недетерминированных циклах, выход из которых выполняется при достижении конца файла.

Функция ***Eoln()*** (*end-of-line* — конец строки) возвращает булево значение, сообщающее, находится ли указатель файла в конце текущей строки. Для файла в режиме чтения значение Eoln () равно True, если указатель находится на символе конца строки или если Eof() равно True. Функция Eoln() также используется в недетерминированных циклах посимвольной обработки текстовых файлов.

Подпрограммы управления файлами

Система Delphi предоставляет программисту также встроенные подпрограммы, выполняющие различные операции над файлами, такие как создание каталогов или переименование файлов.:

CreateDir()	Создает новый каталог
DeleteFile()	Удаляет файл с диска
DirectoryExits()	Определяет, существует ли заданный каталог
DiskFree()	Возвращает количество байтов свободной памяти на заданном диске
DiskSize()	Возвращает размер в байтах заданного диска
FileDateToDateTime()	Преобразует значение даты/времени системы DOS в значение типа TDateTime
FileExits()	Проверяет, существует ли заданный файл
FileGetAttr ()	Возвращает атрибуты заданного файла
FileGetDate()	Возвращает метку даты/времени DOS заданного файла
FileOpen()	Открывает файл с указанным методом доступа
FileRead()	Читает заданное количество байтов из файла
FileSearch()	Находит путь к файлу
FileSeek()	Размещает указатель текущего файла в предварительно открытом файле
FileSetAttr()	Устанавливает атрибуты заданного файла
FileSetDate()	Устанавливает для заданного файла метку даты/времени DOS
FileWrite()	Записывает содержимое буфера в текущую позицию файла
FindFirst()	Находит первый экземпляр файла с указанными атрибутами в заданном каталоге
FindNext()	Возвращает следующий экземпляр файла с указанными атрибутами в заданном каталоге
ForceDirectories()	Создает все каталоги заданного пути, если их еще не существует
GetCurrentDir()	Возвращает имя текущего каталога
RemoveDir()	Удаляет существующий пустой каталог
RenameFile()	Изменяет имя файла
SetCurrentDir()	Устанавливает текущий каталог
ChDir()	Изменяет текущий каталог
FileClose()	Закрывает указанный файл
FindClose()	Освобождает память, выделенную процедурой
GetDir()	Возвращает текущий каталог заданного диска

Пример использования текстовых файлов

И качестве примера рассмотрим программу состоящую из процедур, которые позволяют на запись и чтение из файла.

```
program pliktekst;

procedure pisz(var plik:textfile; n:string);//запись одной строки в начале
                                            //или в конце файла
var   linia:string;
begin
   assignfile(plik,n);
   if FileExists(n) then append(plik) else rewrite(plik);
   writeln(Введите текст);
   readln(linia);
   writeln(plik,linia);
   closefile(plik)
end;

procedure czytaj(var plik:textfile; n:string);  //чтение содержимого файла
var   linia:string;
begin
assignfile(plik,n);
if FileExists(n) then
begin
    reset(plik);
    writeln('Распечатка содержимого файла ',n);
    while not Eof(plik) do
        begin
           readln(plik,linia);
           writeln(linia)
        end;
     closefile(plik)
end
           else writeln(` Нет файла `, n)
 end;

var plik1:textfile;
   nazwa:string;

BEGIN
   write(Введите название дискового файла ');
   readln(nazwa);
   nazwa:=nazwa+'.txt';
   czytaj(plik1, nazwa);
   pisz(plik1, nazwa);
   czytaj(plik1, nazwa);
   readln;
END.
```

Неправильное форматирование входного файла часто является причиной ошибок ввода-вывода. В частности, распространенной причиной такой ошибки является наличие дополнительных пустых строк, перед концом файла. Прежде чем искать ошибки в программе, убедитесь, что вводимый файл действительно имеет ожидаемый формат.

некоторых текстовых файлах элементы данных отделены друг от друга запятыми. Для ра с такими файлами процессор баз данных компании Borland (BDE-Borland Database Engine) использует специальный драйвер ASCII. Например, присвоение свойству TableType объекта "таблица" значения ttASCII означает, что файл данных таблицы является текстовым файлом ASCII с элементами, разделенными запятыми.

Бинарные файлы

До сих пор рассматривались текстовые файлы, т.е. файлы последовательного доступа. Раннее рассматривались сложные структуры данных, включая типы записей. Теперь вы узнаете, как сохранять записи в файле данных. Если применить для этого текстовый файл, то сохранение или чтение каждого поля потребует нескольких строк исходного кода. Кроме того, доступ к отдельному полю записи в текстовом файле — процесс весьма неэффективный. Для преодоления этих проблем в Object Pascal предусмотрен еще один вид файлов — ***бинарные файлы***.

В отличие от текстовых файлов, доступ к элементам структур данных, хранящихся в бинарных файлах, выполняется не последовательно, а в произвольном порядке. Поэтому бинарные файлы иногда называют ***файлами произвольного доступа***. Кроме того, хранящуюся в бинарном файле структуру данных можно полностью прочитать или записать с помощью одного оператора исходного кода. Таким образом, использование бинарных файлов представляет собой более удобный и быстродействующий способ хранения информации, организованной в структуры данных.

Текстовые файлы хранятся в формате ASCII, а данные, хранящиеся в бинарных файлах, с форматом ASCII никак не связаны. Если бинарный файл прочитать как текстовый, то большинство символов окажется бессмысленным. Чтобы правильно обрабатывать бинарные файлы, программа должна учитывать структуру хранящихся в них данных.

Существует два вида бинарных файлов: ***типизированные*** и ***нетипизированные***.

Файлы типизированные

В типизированном файле хранится последовательность элементов одного и того же типа. Возможно, вы заметили схожесть с определением массива, который согласно определению является последовательностью элементов одного типа. Фактически типизированный файл можно рассматривать как массив, имеющий форму файла. Для доступа к элементам типизированного файла можно в качестве индекса применять номер записи.

Для определения типа файла используются ключевые слова ***file of*:**

```
type
    имя_типа_файла = file of тип_данных;
```

Именем типа файла может быть любой правильный идентификатор. Параметр *тип_данных* должен иметь фиксированный размер, а потому он не может быть ни явным, ни неявным указательным типом. Это значит, что в типизированном файле не могут храниться динамические массивы, длинные строки, классы, объекты, указатели, вариантные типы, другие файлы, а также любые структуры данных, содержащие эти типы.

Рассмотрим следующий фрагмент кода:

```
type
StudentRec = record
            lastName: String[30];
            firstName: String[20];
            ID : String[12];
            GPA : Real;
            CrdtHrs : Real;
```

```
            end;
StudentDB = file of StudentRec;
Var
studentFile: StudentDB;
```

В этом фрагменте объявляется тип StudentDB типизированного файла, хранящего записи StudentRec. Переменная studentFile является дескриптором файла типа StudentDB, она будет использована в операторах обращения к файлу. Таким образом, в файле будут храниться записи с информацией о студентах: их имена, фамилии, идентификационные номера ID, средние оценки GPA и количество зачетных часов crdtHrs.

Как и в случае применения текстовых файлов, для привязки дескриптора бинарного ному файлу, хранящемуся на диске, используется процедура ***AssignFile()***. Процедуры ***Reset()*** и ***Rewrite()*** с бинарными файлами работают так же, как и с текстовым. По умолчанию к бинарному файлу можно обращаться как для ввода, так и для вывода независимо от того, какая из этих процедур используется. Сравните: процедура Reset() обращается к текстовому файлу только в режиме чтения, а Rewrite ()— только в режиме записи. Для бинарных файлов обе эти процедуры устанавливают указатель файла в его начало. Процедура Append () используется только для текстовых файлов, ее применение с бинарными файлами недопустимо.

Значение глобальной переменной ***FileMode*** определяет режим доступа к бинарному файлу в момент его открытия процедурой Reset (). Допустимыми значениями FileMode являются: 0 — только чтение, 1 — только запись, 2 — чтение и запись. По умолчанию значение FileMode равно 2. После того как значение переменной FileMode будет присвоено заново, все последующие вызовы Reset () будут выполняться в указанном режиме доступа.

Используйте процедуру Rewrite() для создания бинарного файла или для очистки его содержимого. Во всех остальных случаях используйте процедуру Reset().

Процедура ***Read()*** читает элемент данных из бинарного файла в переменную совме- стимого типа. Аналогично процедура ***Write()*** записывает содержимое переменной в би- нарный файл совместимого типа. Обе операции (чтение и запись) выполняются над элементом, на который в текущий момент ссылается указатель файла. После выполнения соответствующей операции ввода-вывода процедуры Read() и Write() автоматически увеличивают значение указателя файла на 1, в результате чего он ссылается на следую щий элемент данных.

Упрощенные синтаксисы этих процедур имеют вид

```
Read(дескриптор_файяа,переменная);
Write(дескриптор_файла, переменная);
```

Как и для текстовых файлов, один оператор Read() или Write() может содержать несколько переменных. Например, операторы

```
Read(binFlle, data1);
Read(binFlle, data2);
Read(binFlle, data3);
```

эквивалентны оператору

```
Read(binFile, datal, data2, data3);
```

Аналогично этому операторы

```
Write(binFile, datal);
Write(binFile, data2);
Write(binFile, data3);
```

Выполняют те же операции, что и

```
Write(binFile, datal, data2, data3);
```

Полные синтаксисы процедур Read () и Write() имеют следующий вид:

```
Read (дескриптор, переменна1 [, переменная, ..]);
Write(дескриптор, переменна1 [, переменная, ..]);
```

В отличие от текстовых, бинарные файлы не состоят из строк, поэтому попытка использовать с ними процедур Readln() и Writeln() вызовет синтаксическую ошибку Эти процедуры используются только с текстовыми файлами. Аналогично с бинарными работает процедура Eof(), но не Eoln().

Процедура ***Seek()*** перемещает указатель бинарного файла на заданную запись или на элемент данных. Вот ее синтаксис:

```
Seek(дескриптор, номер_записи);
```

Номер записи должен быть целочисленным выражением. Значение номера записи первого элемента данных равно 0.

Функция ***Filesize()*** возвращает количество записей (элементов), хранящихся в указанном бинарном файле. Значение номера записи бинарного файла находится в диапазоне от 0 до FileSize(дескриптор) -1. Для установки указателя бинарного файла в его конец используется оператор

```
Seek(дескриптор, FileSize(дескриптор)) ;
```

Вызов процедуры Write() сразу после этого оператора увеличивает размер бинарного файла на столько элементов данных, сколько выводимых переменных входит в список фактических параметров процедуры Write(). Процедура ***Truncate()*** удаляет из бинарного файла все элементы данных от текущей позиции указателя файла до его конца причем текущей позицией указателя становится конец файла.

По завершении всех операций бинарный файл необходимо закрыть. Это выполняется с помощью процедуры ***CloseFile(),*** которая разрывает связь между дескриптором бинарного файла и внешним файлом, хранящимся на диске.

Файлы нетипизированные

Для объявления дескриптора нетипизированного файла используется ключевое слово ***file. Нетипизированный файл*** фактически является низкоуровневым каналом ввода-вывода, используемым для прямого доступа к хранящемуся на диске файлу, независимо его структуры или содержимого. Синтаксис объявления дескриптора нетипизированного файла имеет вид

```
Var
  Имя_дескриптора_нетипизированного_файла : file;
```

Процедуры Reset () и Rewrite () позволяют включить дополнительный параметр определяющий размер записи в операциях ввода-вывода нетипизированного файла. Для нетипизированного файла размер записи не имеет никакого отношения к структуре хранящихся в нем данных; под записью здесь подразумевается фрагмент файла записываемый на диск за одно обращение к нему. Этот параметр влияет только на эффективность выполнения программы. По умолчанию размер записи равен 128 байт. Размер 1 байт может быть использован для работы с любым нетипизированным файлом.

Для выполнения высокоскоростных операций обмена с нетипизированными файлами в место стандартных процедур Read() и Write() используются процедуры ***BlockRead()*** и ***BlockWrite ().*** В остальном для нетипизированных файлов используются те же стандартные процедуры ввода-вывода, что и для типизированных.

Наиболее употребительные подпрограммы Delphi, предназначенные для работы с бинарными файлами это:

Eof() Проверяет, находится ли указатель файла в его конце
FilePos() Возвращает текущую позицию в файле (номер записи)
FileSize() Возвращает размер файла (в записях — для типизированных и в байтах — для нетипизированных файлов)
IOResult() Возвращает статус последней выполненной операции ввода-вывода
AssignFile() Связывает имя внешнего файла с дескриптором бинарного файла
BlockRead() Читает одну или несколько записей из бинарного файла в переменные
BlockWrite() Записывает одну или несколько записей из переменных в бинарный файл
CloseFile() Разрывает связь между дескриптором бинарного файла и внешним файлом
Read() Читает элемент данных из бинарного файла
Reset() Открывает существующий бинарный файл
Rewrite() Создает и открывает новый бинарный файл
Seek() Устанавливает указатель файла на запись с заданным номером
Truncate() Удаляет все элементы данных начиная от текущей позиции файла
Write() Записывает элемент данных в бинарный файл

Пример использования бинарного типизированного файла

```
program pliki1;
{$APPTYPE CONSOLE}
uses SysUtils;
type
  dni = 1..31;
  miesiace = 1..12;
  tdata = record
            dzien   : dni;
            miesiac : miesiace;
            rok     : word;
          end;
  twpis = record
            imie     : string[20];
            nazwisko : string[40];
            dataur   : tdata;
          end;
const
  cnazwapliku='dane.dat';
var
  fplik : file of twpis;
  vwpis : twpis;
  k     : string[1];
  poz   : word;
procedure czytajwpis(nr : word); //чтение записи с номером nr
begin
  seek(fplik,nr);
  {$I-} read(fplik,vwpis); {$I+}
  if ioresult<>0 then
    begin
      writeln(Нет такой позиции в файле.');
      writeln;
    end
  else
    begin
      writeln('Имя: ',vwpis.imie);
```

```
        writeln('Фамилия: ',vwpis.nazwisko);
        writeln('День рождения: ',vwpis.dataur.dzien,'-',vwpis.dataur.miesiac,'-
                ',vwpis.dataur.rok);
      writeln;
    end
end;
procedure dodajwpis; //введение записи в конце файла
begin
  write('Имя: '); readln(vwpis.imie);
  write('Фамилия: '); readln(vwpis.nazwisko);
  write('День рождения: '); readln(vwpis.dataur.dzien);
  write('Месяц рождения: '); readln(vwpis.dataur.miesiac);
  write('Год рождения: '); readln(vwpis.dataur.rok);
  seek(fplik,filesize(fplik));
  write(fplik,vwpis);
  writeln('Записано под номером ',filepos(fplik)-1);
end;

begin
  assignfile(fplik,cnazwapliku);
  {$I-} reset(fplik); {$I+}
  if ioresult<>0 then rewrite(fplik);
  repeat
    write('W – введение данных, C – чтение данных, K - конец: ');
    readln(k);
    k:=uppercase(k);
    if k='W' then dodajwpis;
    if k='C' then
      begin
        write('Номер позиции: '); readln(poz);
        czytajwpis(poz);
      end;
  until k='K';
  closefile(fplik);
end.
```

Резюме

Стандартные типы данных – это простейшие структуры данных, реализуемые языком программирования. Структуры данных, которые создаются, хранятся и обрабатываются в основной памяти компьютера, называют *оперативными структурами*. Компилятор связывает каждый идентификатор с определенным адресом памяти, при этом он учитывает информацию о типе каждой именованной величины с целью проверки совместимости типов. Современные системы программирования позволяют формировать и эффективно обрабатывать широкий спектр оперативных структур.

Существует другой важный класс структур – структура данных, характерные для устройств внешней памяти компьютера. Такие структуры данных называют *файловыми структурами*. На самом нижнем уровне к ним относятся *файлы* – поименованные совокупности однородных данных, располагающиеся на устройствах внешней памяти. Файл – упорядоченный набор информации на внешнем носителе. Множество связанных друг с другом файлов, в свою очередь, могут быть организованы в виде *базы данных*, являющейся центральным информационным ядром системы управления базами данных. Элементами файловой структуры служат *записи*.

Процессор может обрабатывать только данные, находящиеся в основной памяти (а также в регистровой и кэшевой памяти). Поэтому данные, находящиеся во внешней памяти и подлежащие

работке, должны быть сначала перемещены в основную память. Как правило, все подлежащие обработке данные перемещаются из внешней памяти в основную и после обработки из основной памяти во внешнюю небольшими порциями. Минимальная единица данных, которая может быть передана между внешней и основной памятью, называется *физической записью* или *блоком.*

Над всеми структурами данных могут выполняться пять операций: создание, уничтожение, выбор (доступ), обновление, копирование. Структуры данных и алгоритмы служат основой построения программ. Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачному программированию и может в большей степени сказываться на производительности программы, чем детали используемого алгоритма.

Рассмотрение структуры данных без учета ее представления в машинной памяти называют *абстрактной*, или *логической*, *структурой данных.* Способ физического представления данных в памяти машины называется *физической структурой данных* или еще структурой хранения, внутренней структурой, структурой памяти или дамп.

Вопросы для самопроверки

1. Для чего используются файлы данных?
2. Чем отличается файл от массива?
3. Назовите три типа файлов данных в Object Pascal.?
4. Что такое последовательный файл? Как его создать?
5. Что означает файл произвольного доступа?
6. Что понимается под термином «порция данных» для файла?
7. Что такое символы-разделители?
8. Что такое управляющие символы?
9. Что такое указатель файла?
10. Что за дескрипторы Input, Output?
11. Можете перечислить последовательность действий работы с текстовым файлом?
12. Имеются ли для диска ячейки (несколько байтов, рассматриваемые как одно цельное)?
13. Что такое структура данных?
14. В чем различие между физической и логической структурами данных?
15. Какие операции могут выполняться под структурами данных?
16. Какие средства содержат файловая система?
17. Какие стандартные типы данных вы знаете?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 13. Парадигма модульного программирования. Модули

План

Недостатки парадигмы процедурно-ориентированного программирования
Понятие модуля.
Запись модуля
Главная программа модульной Delphi-программы
Парадигма модульного программирования в среде Delphi

Недостатки парадигмы процедурно-ориентированного программирования

Стандартный Паскаль не предусматривает механизмов раздельной компиляции частей программы с последующей их сборкой перед выполнением. Более того, последовательное проведение в жизнь принципа обязательного описания любого объекта перед его использованием делает фактически невозможным разработку разнообразных библиотек прикладных программ. Точнее, такие библиотеки в рамках стандартного Паскаля могут существовать *только в виде исходных текстов* и программист должен *сам включать в программу подчас весьма обширные тексты различных поддерживающих процедур*, таких, как процедуры матричной алгебры, численного интегрирования, математической статистики и т.п.

Вполне понятно, поэтому стремление разработчиков коммерческих компиляторов Паскаля включать в язык средства, повышающие его модульность. Чаще всего таким средством является разрешение использовать внешние процедуры и функции, тело которых заменяется зарезервированным словом EXTERNAL. Разработчики Delphi пошли в этом направлении еще дальше, включив в язык механизм так называемых модулей.

Модуль в начале понимался как подпрограмма (процедура или функция), оформленная определенным образом и выполняющая строго одно действие. Сейчас **модуль** - *это автономно компилируемая программная единица, включающая в себя различные компоненты раздела описаний (типы, константы, переменные, процедуры и функции) и, возможно, некоторые исполняемые операторы инициирующей части.* В модулях явным образом выделяется некоторая *«видимая» интерфейсная часть*, в которой сконцентрированы описания глобальных типов, констант и переменных, а также приводятся *заголовки глобальных процедур и функций*. Так что модуль может содержать много подпрограмм. Появление объектов в интерфейсной части делает их доступными для других модулей и основной программы. Тела процедур и функций располагаются в, так называемой, *исполняемой части модуля или части реализации*, которая может быть скрыта от пользователя.

Модули представляют собой прекрасный инструмент для разработки библиотек прикладных программ и мощное средство модульного программирования. Важная особенность модулей заключается в том, что компилятор размещает их программный код в отдельном сегменте памяти. Максимальная длина сегмента не может превышать 64 Кбайта, однако количество одновременно используемых модулей ограничивается лишь доступной памятью, что дает возможность создавать весьма крупные программы.

Термин модульное программирование означает разработку программ, состоящих из отдельных модулей исходного кода. ***Структурирование программ*** – это ее разбивка на отдельные подпрограммы, содержащие такие управляющие структуры, как условные операторы или цикла. В каждом модуле может находиться, повторяем, произвольное количество подпрограмм. Программы, написанные в Delphi, структурированы по своей природе, поскольку сам язык строго структурирован. Более того: *все приложения, разработанные в среде Delphi, являются модульными программами, потому что среда разработки создает отдельные модули исходного кода.*

Объем современной программы обычно настолько велик, что, как правило, невозможно бессмысленно представлять ее себе как целое. Программа формируется из отдельных фрагментов исходного текста, представленных в автономных модулях.

Разработка программ на основе принципов модульного программирования предоставляет ряд существенных преимуществ. Во-первых, средства модульного программирования позволяют логически организовать программу путем разделения задачи на более простые подзадачи. Такое разделение задачи называется *пошаговым решением (декомпозицией)* и является составной частью *методологии разработки программы сверху вниз*. Например, программист может создать программу, состоящую из трех подпрограмм: одна подпрограмма содержит процедуры ввода, другая выполняет вычисления, а третья выводит результаты вычислений на экран или в файлы. Во-вторых, модульное программирование облегчает отладку и делает код более понятным. В-третьих, как и в случае со стереосистемой, программу можно модернизировать путем замены отдельных модулей, не затрагивая другие модули.

И наконец, концепция модульного программирования позволяет использовать разработанные коды повторно. ***Повторное использование кодов*** - важнейший принцип компьютерного программирования.

Модули бывают ***стандартные*** (встроенные в Дельфи) и ***пользовательские***. К числу стандартных модулей принадлежат: SysUtils, Math, Windows, ...

Понятие модуля

Модуль *- это набор переменных и подпрограмм ,заданных в интерфейсном разделе и реализованных в разделе реализации.*

В ***интерфейсном разделе*** сосредоточены определения переменных и заголовки подпрограмм, доступных для использования во внешних программах.

В ***разделе реализации*** находится полный код подпрограмм, указанных в интерфейсном разделе. Подобное разделение модуля на две части позволяет скрыть от пользователей способы реализации находящихся в нем подпрограмм, в теле которых, возможно, задействованы оригинальные алгоритмы.

Скомпилированный модуль может использоваться в других проектах в качестве поставщика функциональных возможностей, заданных в интерфейсном разделе. Головная программа, выполненная в подобной модульной архитектуре, представляет собой небольшой набор команд с вызовами подпрограмм из модулей проекта.

Запись модуля

Модуль начинается с ключевого слова ***unit***, за которым следует произвольный идентификатор и точка с запятой. Идентификатор задает имя модуля. Язык Delphi требует, чтобы оно совпадало с именем файла, в котором хранится модуль. Пусть, например, модуль описан следующим образом:

```
unit Unit1;
…
```

Данный модуль должен храниться в файле `unit1.pas`.

За строкой с ключевым словом ***unit*** следует описание интерфейсного раздела. Этот раздел начинается с ключевого слова ***interface*** и завершается перед ключевым словом ***implementation***. В интерфейсном разделе описываются типы данных, константы, переменные, классы, указываются заголовки процедур и функций. Они доступны внешним программам и другим модулям, использующим данный модуль. Чтобы подпрограмма была доступна в других модулях, копия заголовка подпрограммы должна быть записана в интерфейсном разделе модуля, в котором подпрограмма определяется. Тогда подпрограмма будет иметь ***открытую*** область видимости. Это

чает, что она сможет использоваться в любом модуле, содержащего определ подпрограммы. Обычно все заготовки пользовательских подпрограмм в интерфейсном разделе располагаются последовательно один за другим. Таким образом, интерфейсный раздел модуля должен содержать заготовки всех открытых пользовательских подпрограмм, которые будут вызываться в других модулях.

Если подпрограмма *определена* в модуле и вызывается в разделе реализации, но заголовок подпрограммы в интерфейсном разделе не приведен, то она называется ***закрытой***. Ее можно вызывать только в том модуле, в котором она определена.

В Delphi не накладываются какие – либо ограничения на последовательность расположения подпрограмм. *Заголовки открытых подпрограмм*, определенных в модуле, могут быть записаны в интерфейсном разделе в любой последовательности. В разделе реализации определения открытых подпрограмм также могут быть расположены в любом порядке, независимо от действительной последовательности вызовов подпрограмм в модуле. Однако определения *закрытых подпрограмм* должны быть расположены перед их вызовом из другой подпрограммы. Другими словами, определение вызываемое закрытой подпрограммы должно быть расположено выше ее первого вызова. В этом случае заголовком закрытой подпрограммы служит ее определение (точнее, заголовок определения).

Раздел реализации содержит реализации указанных в интерфейсном разделе процедур, функций и методов классов. Он начинается с ключевого слова ***implementation*** и завершается ключевым словом ***end*** с точкой в конце. Этим словом завершается и весь модуль. Все, что описано внутри раздела реализации, в другим модулям недоступно, «невидимо».

Общая структура модуля такова:

```
unit Unit1;
 // вместо Unit1 можно указать другое
 // подходящее имя модуля

interface          // описание интерфейсного раздела модуля:
   uses   список подключаемых модулей для раздела интерфейса
   //описание типов, констант, переменных, классов
   //заголовки процедур и функций

implementation      // описание раздела реализации модуля:
  uses  список подключаемых модулей для раздела реализации
  //реализация классов и подпрограмм,
  //описанных в интерфейсном разделе

initialization
  //инициализация переменных
finalization
  //действия при завершении работы программы
end.
```

За разделом реализации указываются еще два раздела, каждый из которых необязателен. Первый раздел начинается с ключевого слова ***initialization*****,** второй - с ключевого слова ***finalization***. Если в модуле имеется хотя бы один из этих разделов, считается, что раздел реализации заканчивается перед ним.

Ключевое слово ***initialization*** задает группу команд, которые вызываются один раз, в момент загрузки модуля при запуске программы. Обычно в разделе инициализации формируют начальные значения переменных модуля и вызывают процедуры инициализации системных ресурсов (например, для генерации случайных чисел). Раздел завершения модуля задается группой команд после ключевого слова ***finalization*** и продолжается до заключительного слова ***end.*** Эти команды гарантированно вызываются при завершении программы. Такая возможность полезна, если требуется, например, освободить системные ресурсы, захваченные в процессе работы приложения.

рядок вызова разделов завершения противоположен порядку их инициализации. Так, если первым был вызван на инициализацию модуль *Unit1*, за ним модуль *Unit2* и потом *Unit3*, то их разделы завершения вызывались бы в порядке *Unit3, Unit2 и Unit1*.

Доступ к интерфейсной части модуля из других модулей проекта (приложения) Delphi реализуется в языке Delphi с помощью ключевого слова ***uses*** **(**использовать). Оно может следовать за любым из ключевых слов *interface* и *implementation*. За словом *uses* через запятую приводится список внешних модулей. Если внешний модуль с помощью ключевого слова *uses* подключен к интерфейсному разделу, то он доступен и в разделе реализации. Модуль, подключенный в разделе реализации, в интерфейсном разделе невидим. Ключевое слово *uses* позволяет распределять логику работы крупных приложений по разным модулям. Каждый программист независимо от других сосредотачивает усилия на реализации набора функций в конкретном модуле. Впоследствии этот модуль подключается к другим в качестве готового «кирпичика» и используется многократно, в том числе и в иных проектах. При подключении одного модуля к другому содержимое его раздела *interface* доступно, а содержимое раздела *implementation* скрывается.

На основе модуля реализовывается классическая концепция программирования, известная как ***сокрытие информации***. Каждый идентификатор, который объявлен в разделе ***interface*** (интерфейс) модуля, становится видимым в других модулях программы, что обеспечивается использованием выражения ***uses***, в котором помещается ссылка на модуль, в котором объявлен идентификатор. А идентификаторы, объявленные в разделе ***implementation*** (реализации) модуля, являются локальными для этого модуля.

Пример простого модуля:

```
UNIT dzialania;
INTERFACE
Function summa(a,b:Integer):Integer;
Function roznica(a,b:Integer):Integer;
Function umnoj(a,b:Integer):Integer;
Function deli(a,b:Integer):Real;
IMPLEMENTATION
Function summa (a,b:Integer):Integer;
Begin
        Result:=a+b;
End;
Function roznica (a,b:Integer):Integer;
Begin
        Result:=a-b;
End;
Function umnoj (a,b:Integer):Integer;
Begin
        Result:=a*b;
End;
Function deli (a,b:Integer):Real;
Begin
        Result:=a/b;
End;

END.
```

Главная программа модульной Delphi-программы

Итак, модули - это своеобразные библиотеки, к интерфейсным разделам которых возможен свободный доступ. При модульном программировании программа разбита на модули, реализующие разные алгоритмы. Модули получают данные, обрабатывают их и передают другим

дулям. Координация работы осуществляется главным модулем: он вызывает подпрогра передает им данные и возвращает результат программы. Главный модуль или главная программа исполняется по алгоритму, заданному в главном файле проекта с расширением . DPR (сокращение от Delphi Project). Такой файл в проекте всегда один, в отличие от неограниченного числа модулей с расширением .PAS.

Первым в главном файле проекта следует ключевое слово ***program***. За ним указывается произвольный идентификатор, задающий имя программы. Например:

```
Program MyTest;
```

Идентификатор MyTest служит именем программы и одновременно именем файла проекта с расширением .DPR (MyTest.dpr). Следом за именем программы располагается пара логических скобок *begin* и *end*. Между ними находятся команды программы. Выполнение программы начнется с первой команды этой последовательности.

Создать файл. DPR вручную не требуется. При создании пустой заготовки любого проекта среда Delphi автоматически генерирует работоспособный код пустой заготовки и файл проекта.

Использование процедур и функции содержимых в модуле происходит благодаря директиве ***uses.*** И так программа MyTest использующая модуль dzialania может выглядет так:

```
Program MyTest;
{$APPTYPE CONSOLE}
USES dzialania;
VAR x,y:Integer;
BEGIN
X:=1;
Y:=2;
WRITELN('сумма=', summa(x,y));
WRITELN('разница=', roznica(x,y));
READLN;
END.
```

Парадигма модульного программирования в среде Delphi

Идея разбиения программы на независимые модули реализована в системах программирования Borland, начинания с самых первых сред разработки Turbo Pascal.

Модуль есть часть программы, расположенная в *отдельном файле*. Размещенный в отдельном файле набор связанных процедур вместе с данными, которые они обрабатывают, называют программной единицей (Unit). Часто слово “Unit” переводят как модуль. Так возник термин «модульное программирование». В модульном программировании акцент сместился от проектирования процедур в сторону организации данных. Помимо прочего, это явилось отражением факта увеличения размеров программ. ***Парадигма:*** *анализируя задачу реши, какие требуются модули; разбей программу так, чтобы скрыть данные в модулях.*

В языке Delphi модулем обычно называется файл исходного кода (Unit), с которым ассоциирован файл формы. *Парадигма модульного программирования в среде Delphi* предполагает разработку программ на основе *отдельных модулей* и *компонентов*. В Delphi под компонентом чаще всего подразумевается *форма* или *элемент управления* в среде разработки. Такая технология программирования помогает логически организовать большие программы, повышает наглядность исходного кода, упрощает отладку и *способствует повторному использованию кодов*.

Повторяем, что *все (оконные) приложения, разработанные в среде Delphi, являются модульными программами*, потому что среда разработки создает отдельные модули (Unitы) исходного кода.

Резюме

Модуль – фундаментальное понятие и функциональный элемент технологии структурного программирования. В технологии обектно-ориентированного программирования это файл (*unit*) с описаниями родственных классов. Модульность программ – основной принцип технологии структурного программирования, характеризуется тем, что вся программа состоит из модулей.

Модуль - подпрограмма, но оформленная в соответствии с особыми правилами.

Правило 1. Модуль должен иметь один вход и один выход и выполнять строго однозначную функцию, которая описывается простым распространенным предложением естественного языка или даже предложением без сказуемого.

Правило 2. Модуль должен обеспечивать компиляцию, независимую от других модулей, с «забыванием» всех внутренних обозначений модулей.

Правило 3. Модуль может вызывать другие модули по их именам.

Правило 4. Хороший модуль не использует глобальные переменные для общения с другим модулем, так как потом трудно отыскать модуль, который портит данные. Если же используются глобальные переменные, то нужно четко комментировать те модули, которые только читают, и те модули, которые могут менять данные.

Правило 5. Модуль кодируется только базовыми структурами алгоритма и тщательно комментируется.

В понятие структуры программы включаются состав и описание связей всех модулей, которые реализуют самостоятельные функции программы и описание носителей данных, участвующих в обмене как между отдельными подпрограммами, так и вводимые и выводимые с/на внешних устройств.

Для того, чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры, которые будут вызываться другими модулями, – *открытые* (public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот модуль, - *закрытых* (private) процедур. Первые перечисляются в отдельной части модуля – *интерфейсе* (interfaice), вторые участвуют только в *реализации* (implementation) модуля. Данные, занесенные в модуль, тоже делятся на открытые, указанные в интерфейсе и доступные для других модулей, и закрытые, доступные только для процедур того же модуля. В различных языках программирования это деление производится по-разному. В языке Delphi модуль специально делится на интерфейс и реализацию, в языке С интерфейс выносится в отдельные «головные» (header) файлы. В языке С++, кроме того, для описания интерфейса можно воспользоваться абстрактными классами. В языке Java есть специальная конструкция для описания интерфейсов, которая так и называется- *interfaice,* но можно и написать и абстрактные классы.

Так возникла идея о скрытии, *инкапсуляции* данных, и методов их обработки Скрытие производится не для того, чтобы спрятать от другого модуля что-то любопытное. Здесь преследуются две основные цели. Первая – обеспечить безопасность использования модуля, вынести в интерфейс, сделать общедоступными только те методы обработки информации, которые не могут испортить или удалить исходные данные. Вторая цель – уменьшить сложность, скрыв от внешнего мира ненужные детали реализации.

Вопросы для самопроверки

1. Суть недостатки процедурного подхода программирования?
2. Чем отличается модуль от подпрограммы?
3. Что такое модульное программирование?
4. Консольное приложение является ли модульным?
5. Что такое главная программа модульного приложения?
6. В чем суть разделения модуля на интерфейсной части и на части реализации?
7. Для чего используется предложение uses?
8. Чем отличаются парадигмы процедурного и модульного программирований?

Лекция 14. Парадигма объектного программирования. Классы и объекты

План
Недостатки парадигмы модульного программирования
Парадигма объектно-ориентированного программирования
Понятие класса и объекта
Инкапсуляция, наследование, полиморфизм.

Недостатки парадигмы модульного программирования

Потребности программирования, поставленного на промышленную основу, невозможно в полной мере удовлетворить с помощью парадигм процедурно-ориентированного и модульного программирования. Даже внутри одной команды разработчиков программных продуктов необходимо согласовать детали представления данных и реализации алгоритмов, направленных на решение общей задачи. Программисты, не согласовавшие свои действия, рискуют создать несовместимые модули. В силу этих причин возникла необходимость создать такую концепцию программирования, которая повысила бы производительность и надежность работы программистов, позволив применить конвейерные методы работы. Так возникла парадигма обьектно-ориентированного программирования.

К возникновению этой парадигме, а также способствовало то, что ни отдельно взятые данные и ни отдельно взятые подпрограммы, являющиеся основой структурно-процедурного и модульного подходов, плохо отражают картину реального мира, плохо моделируют реальные объекты задачи. В сущностях реального мира они, наоборот, объединены в одном цельном реальном объекте. Необходим был программный объект, более адекватно моделирующий ***реальные объекты программируемой задачи.*** Такой программный объект кратко называется **объектом**.

Парадигма обьектно-ориентированного программирования

Объектно-ориентированное программирование – это в наше время совершенно естественный подход к построению сложных программ и систем. В основу обектно-ориентированного программирования положено понятия *класса, объекта* и принципы *инкапсуляции, сокрытия информации, абстракции данных, наследования и полиморфизма.*

Принципы объектно-ориентированного программирования взяты из жизни. Они являются принципами формирования человеческого понимания сложности объективной среды проживания. Человечество управляет сложностью через *абстракцию.* Например, люди не представляют себе автомобиль как набор десятков тысяч индивидуальных частей (деталей). В их воображении автомобиль – хорошо определенный объект со своим собственным уникальным поведением. Эта абстракция позволяет людям использовать автомобиль для поездки, не задумываясь над сложностью частей, из которых он состоит. Игнорируя подробности работы двигателя, трансмиссионной и тормозной системы, они могут свободно пользоваться объектом в целом.

Мощным способом управления абстракцией является применение *иерархических классификаций.* Они позволяют расслоить семантику сложных систем, разбивая их на более управляемые части. Извне автомобиль представляется как единый объект. Изнутри же он содержит несколько систем – рулевого управления, тормозной и звуковой систем, пристяжных ремней, нагревателей и т. п. В свою очередь каждая из этих подсистем состоит из более специализированных узлов. Суть здесь в том, что вы управляете сложностью автомобиля (или любой сложной системой) через использование иерархических абстракций.

Основная идея парадигмы обьектно-ориентированного программирования заключается в том, что все вычисления происходят внутри *объектов*, обменивающихся между собой *сообщениями*. В некотором смысле обьектно-ориентированное программирование можно считать высшей ступенью развития модульного программирования; только роль модулей здесь играют объекты. В модульном программировании вся информация обрабатывалась внутри модулей, причем данные не зависели от них. Теперь объекты, наделены возможностью не только хранить, но и обрабатывать данные. Например, если в модульных программах структура представляет собой простую совокупность записей, то в обьектно-ориентированных – это *активный объект*, который не просто *хранить информацию*, но и *обрабатывает* ее. Чтение и обновление данных должно производиться посредством вызова соответствующих методов. В этом и заключается *принцип инкапсуляции*. *Инкапсуляция* – это механизм, который связывает код вместе с обрабатываемыми им данными и сохраняет их в безопасности как от *внешнего влияния*, так от *ошибочного использования*. Можно представить инкапсуляцию как *защитную оболочку,* которая предохраняет код и данные от произвольного доступа из других кодов, определенных вне этой оболочки. Доступ к коду и данным *внутри* оболочки строго контролируется через хорошо определенный *интерфейс*. Интерфейс определяется совокупностью методов и свойств. До сих пор идея инкапсуляции внедрялась в программирование только посредством призывов и примеров в документации, но в языке же Object Pascal появилась соответствующая конструкция. В объектах Object Pascal пользователь объекта может быть полностью отгорожен от его данных при помощи *свойств*.

Итак, *объект – это объединение данных и алгоритмов, обрабатывающих эти данные*. Теперь мы можем, например, погрузить массив и метод его сортировки в некий объект, а теперь обратиться к этому объекту с просьбой упорядочить элементы массива и выдать результат. Основное преимущество такого подхода – гибкость. Если данный модуль является частью более сложной конструкции, необходимо лишь согласовать способы его сообщения с внешним миром – то что называют интерфейсом. Его внутренний мир других программистов не касается. Как хранятся данные внутри модуля и как они обрабатываются, должен знать только создатель данного объекта. А если другие программисты окажутся слишком бесцеремонными и захотят изменить внутреннюю структуру объекта? Как защитить его от постороннего вмешательства? Для этого используется принцип *сокрытия информации.* В обьектно-ориентированном программировании внутреннюю структуру объекта можно разделить на три раздела: *открытый, закрытый и защищенный*. Не вдаваясь в преждевременные подробности, заметим, что открытый раздел доступен всем сущностям программы (иначе говоря, виден из любой ее точки и из внешнего мира), закрытый раздел - лишь внутренним сущностям (подпрограммам) объекта, а защищенный – внутренним сущностям данного объекта и объекта - наследника.

Для того чтобы программа была эффективной, необходимо, чтобы объекты программы хорошо соответствовали объектам решаемой задаче. Однако со временем объект может морально устареть. Например, может появиться новый сверхскоростной алгоритм сортировки. Что делать? Создать новый объект? Изменить старый? Для того чтобы облегчить задачу, решили разделить *схему объекта (т.е.* ***класса)*** и *его реализацию (т.е.* ***объекта)***. *Класс* содержит внутри себя (инкапсулирует) набор данных и методов (подпрограмм), реализованных при помощи процедур или функций и обычно обобщает свойства ряда однородных объектов. Класс – это *тип данных*, определяемый пользователем, который включает в себя набор допустимых *значений* (описание типа) и набор возможных *операций* над ними (поведение типа). *Объект* является *экземпляром класса* или, по-другому, переменной типа данных, объявленного классом (*объектной переменной*). Такое разделение позволяет программисту описать, *что* может делать объект, не конкретизируя, *как* это делать. В этом заключается принцип *абстракции данных*. Класс – это абстрактный тип данных. Объект – это реальная сущность. Во время работы программы объекты резервируют память для своего внутреннего представления. Соотношение между объектом и классом аналогично соотношению между переменной и стандартным типом данных.

принципиальная схема объекта не должна зависеть от ее конкретного наполн Необходимо лишь указать, что объект должен содержать такую-то операцию. Если алгоритм понадобится изменить, мы модифицируем его *реализацию*, но *общая структура объекта* (класс) от этого не пострадает.

Представим себе, что нам понадобилось создать новый проект, который обладал бы возможностями старого, но умел бы делать и что-то новое. Что делали в прошлом? Копировали содержание старого модуля и добавляли него новые функции. Теперь все это можно делать на много проще! Мы просто создаем объект, наследующий все свойства своего предшественника, и добавляем в него новые возможности. Этот механизм называется *наследованием. Наследование* есть процесс, с помощью которого один объект приобретает свойства другого объекта. Оно важно потому, что поддерживает концепцию иерархической классификации. Как уже говорилось выше, наибольшая часть знаний становится управляемой только с помощью иерархических (т. е. организованных «сверху вниз») классификаций. Без применения классификаций каждый объект нуждался бы в явном определении всех своих характеристик. При использовании наследования объект нуждается в определении только тех качеств, которые делают его уникальным в собственном классе. Он может наследовать общие свойства от своего родителя. Поэтому именно механизм наследования дает возможность одному объекту быть специфическим экземпляром более общего случая.

Объект может получать разнообразную информацию. Разумеется, для ее обработки можно заранее предусмотреть его реакцию, написав соответствующие функции. Однако группа связанных объектов может иметь еще *общий класс действий.* Это означает, что, возможно использовать *один интерфейс для общего класса действий* (т. е. написать для всех таких объектов одну общую команду для выполнения общего действия). Специфическое действие определяется точной природой ситуации, а именно ситуацией, связанной с конкретно используемым объектом. Забота компилятора – выбрать специфическое действие (т. е. метод конкретно используемого объекта) для его использования в каждой конкретной ситуации. В обьектно-ориентированных языках существует еще возможность перегружать и наследовать функции, процедуры, что позволяет объекту самому конкретизировать их смысл в ходе выполнения программы. В этом (и не только) заключается сущность *полиморфизма.* В принципе полиморфизм проявляется при, так называемом *позднем связывании*. Итак, *полиморфизм – механизм*, который позволяет использовать *один интерфейс для общего класса действий*.

Понятие класса и объекта

Описывая принципы парадигмы обьектно-ориентированного программирования, мы оперировали понятием объект. Однако ***объект*** – это физическая сущность, возникающая при выполнении программы, т.е. совокупность ячеек памяти, хранящих данные и код. Для того чтобы создать объект, необходима *его схема*: какие данные он содержит, какие подпрограммы обрабатывают эти данные, как организован доступ к этим данным и подпрограммам, которые называются членами объекта. В языке Delphi схемой объекта называется ***класс***.

Класс – это тип данных, образец похожих объектов, состоящий из двух составляющих:

1. ***Данных*** (полей, свойств) - это: переменные (***var***), константы (***const***), свойства (***property)***
2. ***Методов*** (действий) - это: процедуры (***procedure***), функции (***function***), конструкторы (***constructor***) и деструкторы (***destructor***).

В процессе работы программы объекты могут создаваться и уничтожаться. Таким образом, структура программы является динамическим образованием, меняющимся в процессе выполнения. Конструктор создает объект, процедуры и функции обрабатывают данные объекта, а деструктор уничтожает объект. Объекты также могут реагировать на *события*. События наступают, прежде всего, вследствие действий пользователя, и в результате работы самих объектов. На каждом объекте определено множество событий, на которые он может реагировать.

Синтаксис объявления класса имеет вид:

```
TYPE имя_класса =CLASS[(имя_предка)]
                    [список_полей:  тип];
                    [список_заголовков_метод];
                  END;
```

Например, класс, позволяющий решать основные операции на двух целых числах можно определит как:

```
Type Tdzialania=class
                  //данные (или переменные)
                  a, b:Integer;
                  //методы
                   Procedure dane;
                   Function summa:Integer;
                   Function roznica:Integer;
                   Function umnoj:Integer;
                   Function deli:Real;
                end;
```

Классом в Delphi называется структура языка, которая может иметь в своем составе переменные, функции и процедуры. Класс есть сложный пользовательский тип. Переменные в зависимости от предназначения именуются *полями* или *свойствами* (см. ниже). Процедуры и функции класса — *методами*. Соответствующий классу тип будем называть *объектным типом*, а переменную такого типа *объектной переменной*.

Класс с данными разного вида можно определить следующим образом:

```
type Tfigura=class
  private
    Fkolor:byte; //поле переменное для записи и чтения

    procedure SetColor(k:byte);
    function GetColor:byte;
  public
    a:integer;        //поле переменное для записи и чтения
    var x,y:integer;  //поле переменное для записи и чтения (var opcjonalne)
    const autor='Kowalski';  //поле переменное только для чтения
    property kolor:byte read GetColor write SetColor;  //свойство
//... тут методы
end;
```

Класс с методами разного вида можно определить следующим образом:

```
type Tfigura=class
  //... тут данные
  constructor Create; //конструктор
  constructor Tworz1; overload; // конструктор перегруженный
  constructor Tworz1(xx,yy:integer);overload; // конструктор перегруженный
  procedure WyswietlAutora; //метод статический;
  procedure dane;overload;
  procedure dane(aa, xx, yy:integer); overload;
  procedure rysuj; virtual; //метод виртуальный
  function pole: real; dynamic;  //метод динамический
  procedure obliczenia; virtual; abstract;  //метод абстрактный
  procedure wyswietl;
  destructor Destroy; override;//деструктор
  end;
```

```
Function имя_класса.имя_функции (параметры);  //заголовок функции
Begin
  //операторы
End;

Procedure имя_класса.имя_процедуры(параметры);  //заголовок процедуры
Begin
  // операторы
End;
```

Например, описание методов dane и summa в классе Tdzialania имеет вид:

```
Procedure Tdzialania. dane;
begin
  write('a= '); readln(a);
  write('b= '); readln(b);

end;

Function Tdzialania. summa:Integer;
begin
  result:= a+b;
end;
```

Объект - это экземпляр класса. Синтаксис объявления объекта имеет вид:

```
Var имя_объекта : имя_класса;
```

Например:

```
Var dzialania : Tdzialania;
```

В этом случае переменная dzialania называется *объектной переменной* (кратко *объект*) типа класса (или объектного типа) Tdzialania.

nСоздание объекта методом конструктора

```
имя_объекта := имя_класса . имя_конструктора;
```

Например:

```
dzialania := Tdzialania.Create;
```

- Произведение действий на объекте - использование процедур и функции объекта

```
имя_объекта . имя_метода(параметры);
```

Например:

```
dzialania . dane; // команда объекту dzialania выполнить свое
                  // действие (процедуру) - dane
writeln('сумма=',  dzialania.summa); // команда распечатать на экране монитора
                                    // результата выполнения объектом dzialania своего
                                    // действие (функцию) – summa
```

```
with объект do begin
                    имя_метода(параметры);
                    имя_метода(параметры);
                    .......
                  end;
```

Например:

```
With dzialania .do
begin
    dane;
    writeln('сумма=', summa);
end;
```

- Уничтожение объекта деструктором

```
имя_объекта . имя_деструктора;
```

Например:

```
dzialania . Destroy;
```

```
Program MyObiekt;
{$APPTYPE CONSOLE}  // вариант программы без использования модуля, где еще класс
                    // может быть объявлен
USES SysUtils;
Type Tdzialania=class  // 1. объявление класса
                       //данные
                       a,b:Integer;
                       //методы
                       Procedure dane;
                       Function summa:Integer;
                       Function roznica:Integer;
                       Function umnoj:Integer;
                       Function deli:Real;
                  end;
Procedure Tdzialania. dane;  // описания методов (подпрограмм) решения задачи
                             // полными текстами
        begin
          write('a= '); readln(a);
          write('b= '); readln(b);
        end;

Function Tdzialania. summa:Integer;
        begin
          result:= a+b;
        end;

Function Tdzialania. roznica:Integer;
        begin
          result:= a-b;
        end;

Function Tdzialania. umnoj:Integer;
        begin
          result:= a*b;
        end;
```

```
Function Tdzialania. deli:Real;
        begin
          result:= a/b;
        end;
Var dzialania : Tdzialania;  // 2. ОБЪЯВЛЕНИЕ ОБЪЕКТНОЙ ПЕРЕМЕННОЙ ТИПА КЛАССА

BEGIN
dzialania := Tdzialania.Create;  // 3. создание объекта – экземпляра класса
                                 // 4. выполнение объектом своих методов – методов
                                 //    решения задачи, предопределенных в классе
                                  //    программистом
dzialania . dane;                    // команда объекту выполнить свой  метод dane
writeln('сумма= ', dzialania.summa);   //команда напечатать результат
writeln('разница=', dzialania. roznica); // выполнения метода summa и roznica
writeln('умножение= ', dzialania. umnoj); // метода umnoj
writeln('деление=', dzialania. deli);     // метода deli
dzialania .Destroy;                                  //5. уничтожение объекта
READLN;
END.
```

Объектно-ориентированная программа - это совокупность объектов и способов их взаимодействия. В нашей простой программе используется только один объект, поэтому нет явного взаимодействие объектов. Отдельным (и главным) объектом при таком подходе во многих случаях можно считать пользователя программы. Он же служит и основным, но не единственным, источником событий, управляющих программой.

Класс – это сложный тип данных, определяемый пользователем. Потому имя созданного класса начинается с буквы T (Tdzialania). Класс может иметь в своем составе *поля*, *методы* и *свойств*. Класс, как тип, определяет *объектную переменную*. Объектная переменная (кратко *объект*) есть *экземпляр класса*. Таким образом, класс – это логическая конструкция, а объект – это физическая реальность, программный объект, хранимый в памяти компьютера. Класс должен быть определен до того, как будет объявлена хотя бы одна переменная этого класса (т.е. объект). Переменная, обозначающая объект (т.е. объектная переменная), фактически является указателем, ссылающимся на размещенные в памяти данные, которые принадлежат этому объекту. Следовательно, на один и тот же объект могут ссылаться несколько объектных переменных. Поскольку объектные переменные являются указателями, они могут содержать значение nil, указывающее, что объектная переменная не ссылается ни на какой объект. Однако, в отличие от указательных переменных, объектная переменная для доступа к объекту не требует явного разыменования. Поэтому мы выше пользовались записью вида:

```
  dzialania.summa
```

где к имени объекта оператор разыменования ^ не используется.

Поля объекта аналогичны полям записи (record). Поля класса являются переменными, объявленными внутри класса. Они предназначены для хранения данных во время работы экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны предшествовать методам и свойствам. Обычно поля используются для обеспечения выполнения операций внутри класса. При объявлении имен полей принято к названию добавлять заглавную букву F. Например FsomeField (у нас этого не было сделано). Итак, поля предназначены для использования внутри класса. Однако класс (объект) должен каким-либо образом взаимодействовать с другими классами или программными элементами приложения. В подавляющем большинстве случаев класс должен выполнить с некоторыми данными определенные действия и представить результат. Для получения и передачи данных в классе применяются *свойства.* Для объявления свойств в классе используется зарезервированное слово **property.** *Свойство* можно определить как поле, доступное для чтения и записи не напрямую, а через соответствующие методы. *Методы* — это процедуры и функции, описанные внутри класса

предназначенные для операций над его полями. От обычных процедур и функций методы отличаются тем, что им при вызове передается *указатель на тот объект, который их вызвал*. Поэтому обрабатываться будут поля именно того объекта, который вызвал метод. Внутри метода указатель на вызвавший его объект доступен под зарезервированным именем *self*.

Инкапсуляция, наследование, полиморфизм

Цель класса – инкапсуляция сложности, поэтому существуют механизмы скрытия сложности реализации внутри класса.

Инкапсуляция (encapsulation) означает объединение всех данных об объекте и характеристик его поведения в одном пакете. Существенным достоинством инкапсуляции является то, что она предоставляет средства сокрытия данных, т.е. инкапсулированные данные остаются недоступными для пользователя и посторонних фрагментов программы. Инкапсуляция - это сокрытие реализации **класса** перед его пользователем посредством отделения внутреннего представления класса от внешнего (его интерфейса). При использовании объектно-ориентированного программирования не принято применять прямой доступ к данным **класса**. Пользователь класса может быть полностью отгорожен от его данных при помощи *свойств **(property)*** класса. Обычно свойство определяется тремя своими элементами: полем и двумя методами, которые осуществляют запись и чтение значений поля (данного). Значит, в классе языка Object Pascal доступ к полю может быть непосредственно напрямую и не напрямую, а через методов свойства, связанного с полем.

В языке Object Pascal введен механизм доступа к составным частям объекта, определяющий области, где ими можно пользоваться (т.е. области видимости). В Delphi это делается посредством слов: ***private, public, protected, published,*** размещенных внутри объявления класса.

Private – доступ к данным (свойствам) и методам, объявленных ниже этой директивы возможен только в модуле (*.pas), где объявлен класс

Public - доступ к данным (свойствам) и методам, объявленных ниже этой директивы возможен везде: в модулях, в главной программе

Protected - доступ к данным (свойствам) и методам, объявленных ниже этой директивы возможен только в модуле (*.pas), где объявлен класс и в модулях с наследованными типами

Published - доступ к данным (свойствам) и методам, объявленных ниже этой директивы возможен везде: в модулях, в главной программе (как и ***Private)***, но дополнительно эти составляющие класса видны в программной среде Delphi в оконном приложении в окне ***Object Inspector*** и используются для изменения свойств объектов.

Внимание:

Инкапсуляция действует только тогда, когда ***классы объявлены в отдельных модулях*** в следующем виде.

```
TYPE имя_класса =CLASS[(имя_предка)]
                        private
                            [список_полей: тип];
                            [список_заголовков_метод];
                        public
                            [список_полей: тип];
                            [список_заголовков_метод];
                        protected
                            [список_полей: тип];
                            [список_заголовков_метод];
                        published
                            [список_полей: тип];
                            [список_заголовков_метод];
                    END;
```

Например, в классе Tdzialania (**unit** Unit1) все данные с запрещенным доступом (доступны в модуле, недоступны в главной программе), а методы с произвольным доступом (доступны везде).

```
unit Unit1; //модуль

interface
Type Tdzialania=class
private
a,b:Integer;
public
Procedure dane;
Function summa:Integer;
Function roznica:Integer;
Function umnoj:Integer;
Function deli:Real;
  end;   // класс без свойства (property)

implementation
Procedure Tdzialania. dane;
begin
  write('a= '); readln(a);                  // данные доступны (private)
  write('b= '); readln(b);
end;
// тут описание остальных методов
end.

program Project2; //программа

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Unit1 in 'Unit1.pas'; //модуль с описанием класса

Var dzialania : Tdzialania;

BEGIN
dzialania := Tdzialania.Create;
dzialania . dane;                                  //методы доступны (public)
writeln(`сумма= `, dzialania.summa); //методы доступны (public)
// dzialania a:=1;                                 //данные недоступны (private)
// dzialania b:=2;                                 //данные недоступны (private)
//.....
READLN;
END.
```

Наследование (inheritance) - это создание новых классов (***класс-потомок***) на основе существующих (***класс-предок)***. ***Класс-потомок*** использует структуру или *поведение* другого ***класса-предка*** (одиночное наследование), или других (множественное наследование*)* классов. Наследование вводит иерархию "общее/частное", в которой ***подкласс*** наследует от одного или нескольких более общих ***суперклассов***. Подклассы обычно дополняют или переопределяют унаследованную структуру (данные) и поведение (процедуры и функции). Наследование важно потому, что поддерживает концепцию иерархической классификации. Как известно, наибольшая часть знаний становится управляемой только с помощью иерархических (т.е. организованных

рху-вниз») классификаций. Люди видят мир, состоящим из объектов, которые связаны д… другом иерархическим способом.

Синтаксис объявления класса-потомка имеет вид:

```
TYPE имя_класса =CLASS(имя_предка) //наследование
                        [список_полей_потомка:  тип];
                        [список_заголовков_метод_потомка];
                    END;
```

Например:

```
Type Tdzialania2=class(Tdzialania)
                        Function deli:Real;
                        Procedure rezultat;
                    end;
```

Класс Tdzialania2 является потомком класса Tdzialania. Его структура состоит из данных a, b и метод: dane, summa, roznica, umnoj (наследованных от предка) и метод deli (*другой алгорифм метода с проверкой деления через нуль*) и rezultat (вывод всех действий над числами).

```
Function Tdzialania.deli:Real;
        begin
          if b<>0 then result:= a/b else writeln('ошибка – деление на 0');
    end;
Procedure Tdzialania. rezultat;
        begin
        writeln('Результаты для a= ', a, 'b= ', b);
        writeln('сумма= ',summa);
        writeln('разница= ',roznica);
        writeln('умножение= ',umnoj);
        writeln('деление=',deli);
       end;
```

Полиморфизм (polymorphism – дословно-способность появляться во многих формах) означает, что программа может обрабатывать объект по-разному в зависимости от его класса. Это достигается путем *переопределения методов* родительского класса в его дочерних классах. Полиморфизм в Дельфи реализуются директивами ***virtual*** и ***override,*** следующими за объявлениями метода в классах: ***virtual*** в классе-предка и ***override*** в классе-потомка. Например, имея классы:

```
TYPE
  TSquare= class
              a:Real;
              PROCEDURE dane(x:Real);
              FUNCTION pole:Real; VIRTUAL; //вычисляет площадь квадрата
              PROCEDURE rezultat;
            END;
 TCircle= class(TSquare)  // TCircle класс-потомок, для класса-предка TSquare
 FUNCTION pole:Real; OVERRIDE;  //чтобы перекрыть аналогичный метод в
 // классе-потомке, нужно использовать ключевое слово OVERRIDE. Это возможно
 //только когда метод был определен как VIRTUAL  в классе-предке.
 // Поскольку метод перекрыть (т.е. метод pole стал полиморфным),
 // он будет вычислять площадь окружности
END;
```

принципу полиморфизма мы определили для этих классов метод pole, вычисляющий площадь фигуры.

Имея следующую процедуру

```
//……
PROCEDURE TSquare.rezultat;
Begin
  Writeln('Площадь=', pole:5:2);
End;
//……
```

можно объявить объекты :

```
var Square : TSquare;
    Circle: TCircle;
```

Тогда исполнение одного и того же метода rezultat разными объектами

```
//……
Square. rezultat;
Circle. rezultat;
……
```

для обоих объектов дает ожидаемые (разные) результаты. Для объекта каждого класса площадь вычисляется по-разному. Поскольку метод pole переопределяется, программист может вызывать метод pole, не заботясь о виде фигуры. Этим полиморфизм похож на *перегрузку операторов*, например, оператор + может выполнять различные операции в зависимости от типов операндов: суммирование для чисел или конкатенацию для строк. Однако между полиморфизмом и перегрузкой операторов есть и существенное отличие: до момента вызова метода в программе вид фигуры может быть вообще неизвестен. Это позволяет считать, что вызов метода rezultat будет работать для любого потомка класса TSquare, которые сейчас могут быть еще не созданы, но будут иметь метод pole, надлежащего определения.

Резюме

Все языки программирования построены на абстракции. Язык ассемблера есть небольшая абстракция от лежащего в основе компьютера, его машинного языка. Языки высокого уровня были абстракциями от ассемблера. Эти языки дают значительное преимущество по сравнению с ассемблером, но их основная абстракция по-прежнему заставляют вас думать о структуре компьютера, а не о решаемой задаче. Объектно-ориентированный язык высокого уровня (объектный подход) делает шаг вперед в абстракции, предоставляя программисту средства для решения задачи в ее пространстве, в ее понятиях. Мы обращаемся к элементам пространства задачи и их представлениям в пространстве решения как к «объектам». Идея состоит в том, что программа может адаптироваться к трудностям задачи, создавая новые типы объектов по мере необходимости. Таким образом, объектно-ориентированное программирование (ООП) позволяет описывать проблему в ее выражениях, а не в терминах компьютера.

Для объектно-ориентированного программирования характерно следующие этапные работы:

- *Осуществление объектно-ориентированного анализа условий и требований задачи.* Анализом задачи формируется необходимый набор суперклассов, подклассов. Делается моделирование в терминах задачи, т.е. формализованное описание задачи в ее пространстве, а не в терминах компьютера. Делается постановка задачи в виде совокупности действующих экземпляров классов - объектов.

- *Осуществление объектно-ориентированного проектирования*. Проектирование программной системы на основе модели объектно-ориентированного анализа. Проектирование программы, как программной системы, в виде взаимодействующих программных объектов (экземпляров классов).

Таким образом, ООП– это процесс реализации программ, основанной на представлении программы в виде совокупности объектов некоторых классов. ООП предполагает, что любая подпрограмма (функция или процедура) в программе представляет собой метод объекта некоторого класса, причем класс должен формироваться в программе естественным образом, как только в программе возникает необходимость описания новых физических предметов или их абстрактных понятий (объектов программирования). Классы из предметной области непосредственно отражают понятия, которые использует конечный пользователь для описания своих задач и методов их решения. Каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих классов, т.е. технология ООП иначе может быть названа как программирование «*от класса к классу*». Важно осознавать, что разработка программы является пошаговым, последовательным процессом, так же, как, например, обучение людей. Вы можете провести столько анализа и планирования, сколько вам заблагорассудится, но пока вы не приступите к делу, все до конца ясно не будет. У вас будет больше успехов – и больше немедленных результатов, - если вы начнете «выращивать» ваш проект в качестве органичного, эволюционирующего существа, а не сконструируете его сразу, как небоскреб из стеклянного каркаса. ООП позволяет описать проблему в ее выражениях (в виде набора объектов, взаимодействующих между собой) и , а не в терминах компьютера, на котором будет исполнено решение.

Основные технологии ООП – *наследование и полиморфизм* – создают возможности многократного использования программного обеспечения. Вновь создаваемые классы, через механизм наследования, поглощают характеристики существующих суперклассов и добавляют свои собственные уникальные характеристики. Новые классы – подклассы – обычно используются свои собственные переменные и методы, поэтому подкласс, вообще говоря, «больше по размеру», чем его суперкласс. Подкласс является более специфическим классом, чем его суперкласс, и представляет меньшую, но более специализированную группу объектов.

Использование *полиморфизма* открывает возможность создания программных систем, расширение которых производится существенно проще. При разработке программ в них можно предусмотреть алгоритмы обработки объектов всех существующих классов в иерархии в *общем виде* – как объектов суперкласса. Классы, которые не существовали в период разработки программы, могут быть добавлены в программу с внесением в нее небольших изменений, либо эти изменения могут и не потребоваться.

ООП инкапсулирует данные и подпрограммы в совокупности называемые объектами; данные и подпрограммы объекта тесно связаны друг с другом. Объекты обладают способностью сокрытия информации. Это значит, что объекты хотя и могут знать, как связываться друг с другом посредством хорошо определенного интерфейса, им не позволено знать, как реализуются другие объекты – детали реализации спрятаны внутри самих объектов.

Объект в целом определяется как совокупность *свойств* и *методов*, а также *событий*, на которые он может реагировать. Программа, состоящая из отдельных объектов, отлично приспособлена к реагированию на события, происходящие в операционной системе. К другим преимуществам ООП можно отнести большую надежность кода и возможность повторного использования отработанных объектов.

Поэтому ООП помогает справиться с такими сложными проблемами, как:

- Уменьшение сложности разработки больших программ в результате реализации технологии инкапсуляции;

Повышение надежности программного обеспечения. Наследование и полиморфизм эффективные методы, используемые при разработке сложных программных проектов. Поскольку сложность современных программных систем постоянно возрастает, использование ООП позволяет разрабатывать сложные иерархические системы, используя возможности наследования и полиморфизма.

- Обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
- Обеспечение возможности повторного использования отдельных компонентов программного обеспечения;
- Обеспечивает пошаговое программирование больших систем путем многократной конкретизации классов

Вопросы для самопроверки

1. Что такое класс?
2. Как описывается класс?
3. Что такое объект?
4. Как объявляется объект?
5. Какие действия надо провести в программе над объектом?
6. Что такое инкапсуляция?
7. Что такое наследование?
8. Для чего полезен полиморфизм?
9. Какие переменные есть в языке Delphi?
10. В чем заключается важность наследования?
11. Какие преимущества дает объектная модель?
12. Что такое модуль программы и какими характеристиками он должен обладать?
13. Дайте определение понятию «структура программы».
14. Какая разница между переменной математики и переменной текста программы?
15. Какая разница между компьютерной переменной и ячейкой памяти?
16. Какая разница между переменной и объектной переменной текста программы?
17. Какая разница между указателем и объектной переменной?
18. Какая разница между стандартным и пользовательским типами?
19. Какая разница между стандартным типом и классом?
20. Какая разница между полем и свойством класса?
21. Какая разница между подпрограммой и методом класса?
22. Каким образом взаимодействуют объекты между собой?
23. Что такое объектно-ориентированная программа?
24. Какая разница между объектно-ориентированной и процедурной программой?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 15. Парадигма событийного и визуального программирования. Программы управляемые событиями. Компонентное программирование.

План

Событийное программирование
Визуальное компонентное программирование в среде Delphi
Среда разработки Delphi
Проект Delphi
Компоненты Form, Label, Edit, Button
Примеры использования в визуальном программировании часто используемых простых компонентов TButton, Tlabel, TEdit
Пример вычислительной программы

Событийное программирование

Событие это акция узнаваемая объектом, для которой можно определить ответ. Примером событий может быть нажатие конкретного клавиша мыши, клавиатуры, действие часов. Чаще всего источником события является пользователь, который при выполнении программы производит определенные действия. При событийном программировании программа является совокупностью действий (процедур и функций), обслуживающих внешние события. Нельзя предусмотреть очередность выполнения функций, каждый запуск программы может выглядеть по другому. Исполнение программы это ожидание на события и ответ на них. Очередность операции: ввод данных, вычисления, вывод результатов может быть не соблюден пользователем.

После запуска на выполнение такая ***событийно-управляемая программа*** будет ждать наступление тех событий, для которых в тексте данной программы есть соответствующие процедуры обработки событий. Иначе говоря, набор событий и определяет то, что может делать приложение. Программа большую часть времени ожидает только эти события (на другие события просто реагировать не может) и обеспечивает выполнение процедуры обработки этих событий. Код программы, реагирующий на события, получает это событие, обрабатывает его и реагирует нужным образом. Когда программа завершит реакцию на событие, она переходит в состояние ожидания.

Большую роль при событийном программированию играет обслуживание ошибок. При этом используется оператор обслуживания исключений ***try... except... end***; как одна из возможностей обслуживания ошибок:

```
try
  {операторы – тут могут возникнуть ошибки}
  {это блок охраняемый}
except
  {код, который исполняется в случае ошибки }
  {это блок обслуживания исключений}
end;
```

Например

```
try
  Reset(MyFile);
except
  Rewrite(MyFile);
end;
```

```
try
  Result:=a/b;
except
  Writeln('Ошибка'); {код, который исполняется в случае ошибки }
end;
```

Для различения ошибок разного типа в блоке обслуживания исключений используется оператор:
On тип_ошибки **do** оператор;

Например:
```
On EConvertError do writeln('Ошибка преобразования типов');
On EZeroDivide do writeln('Ошибка деления на нуль);
On EMathError do writeln('Ошибка математическая);
```

Другим оператором используемым для обслуживания ошибок и правильных действий с переменными некоторого типа (например Файлы, объекты) является оператор освобождения ресурсов ***try... finally...end***;

```
try
  {операторы}
  {это блок охраняемый}
finally
  {код, который исполняется всегда -в случае ошибки и без нее}
  {это блок освобождения ресурсов }
end;
```

Например

```
...
Rewrite(MyFile);
Try
  {любые действия на файлах}
Finally
  Closefile(MyFile); {код, который исполняется всегда }
End;
```

Визуальное компонентное программирование в среде Delphi

Визуальное программирование – одна из самых популярных парадигм программирования на данный момент. Визуальное программирование состоит в автоматизированной разработке программ с использованием особой диалоговой оболочки. Рассматривая системы визуального программирования легко увидеть, что все они базируются на объектно-ориентированном программировании и являются его логическим продолжением. Наиболее часто визуальное программирование используется для создания интерфейса программ и систем управления базами данных. Основным элементом в средствах визуального программирования в средах Delphi является компонента. *Компонента* представляет собой разновидность *объекта,* который можно перенести (*агрегировать*) в приложение из специальной *Палитры компонент*. Компонента имеет набор свойств, которые можно изменять, не изменяя исходный код приложения (программы).

Компоненты бывают визуальными и невизуальными. Первые предназначены для организации интерфейса с пользователем. Это различные кнопки, списки, статический и редактируемый текст, изображения и многое другое. Эти компоненты отображаются при выполнении разрабатываемого приложения. Невизуальные компоненты отвечают за доступ к системным ресурсам: драйверам баз данных, таймерам и т.д. Во время разработки они отображаются своей пиктограммой, но при выполнении приложения, как правило, невидимы. Компонента может принадлежать либо другой компоненте, либо форме.

Технология визуального программирования состоит в следующем: *создание экранных форм, нанесение визуальных и невизуальных компонент, программирование событий и методов оконных форм.*

При визуальном компонентном программировании программа строится на основе визуальных компонентов – объектов о определенных свойствах и действиях размещаемых на прямоугольной *форме*. Таким образом построенная программа называется *оконным приложением*. Роль окна играет форма. *Формой* называется визуальная компонента, обладающая свойством окна Windows. При разработке программы на форме помещается необходимые компоненты (например, элементы требуемого диалога).

Создавая приложения в среде Delphi разработчик сначала создает экранные формы. Для этого из меню палитры компонент выбирает необходимую компоненту, например, кнопку, и буксирует ее при помощи мыши в нужное место окна разрабатываемой формы. При этом кнопке автоматически присваивается название *(*идентификатор или имя), и она описывается в модуле форм. Если компонента не выделена, то ее надо *выделить* (*сделать ее активной*) щелкнув по изображению компоненты на форме. Затем в *Инспекторе объектов* производится настройка свойств компоненты, путем заполнения отдельных полей. На форму помимо визуальных компонент наносятся невизуальные компоненты. Формы объединяются в единый проект. Далее в соответствии со сценарием диалога программируются методы события основной и подчиненных форм. Программы «пустых» методов событий появляются в окне Редактора после нажатия соответствующих клавиш или действий мыши. Так что, визуальное программирование во многом автоматизирует труд программиста по написанию программы, позволяя в диалоговом режиме создавать «скелет» программы. Затем скелеты пустых методов дополняются разработчиком определенными операторами.

Дальнейшая разработка программы ведется по технологии объектно-ориентрованного программирования. Можно часть программы реализовать по технологии процедурно-структурного программирования. Некоторые недостающие визуальные и невизуальные компоненты получаются модификацией исходных текстов наиболее близких прототипов имеющихся компонент. Рекомендуется новые компоненты помещать в палитру компонент. Это облегчит их повторное использование в данной или последующих разработках.

Как известно, операционная система Windows предоставляет простой и удобный графический интерфейс своему пользователя. Графический пользовательский интерфейс – окно, прямоугольной формы (т.е. оконная форма), содержащий различные графическими элементами (кнопки, меню, пиктограммы и пр.) и управляемый с помощью мыши и клавиатуры. С его помощью пользователь управляет работой компьютера путем передвижения указателя мыши и выбора пиктограмм. Пиктограмма, имеющая вид кнопки с рисунком, может представлять или программу (приложение), или каталог файловой системы.

Оконное Delphi–приложение тоже – это *приложение со своим графическим пользовательским интерфейсом в виде окна.* Пользователь приложения (программы) работает с таким приложением с помощью его оконного пользовательского интерфейса.

Таким образом полученная программа (или оконное приложение) будет событийно-управляемой программой. В среде Windows в каждый момент времени только один объект может принимать входной сигнал (щелчок мышью или нажатие клавиши). Говорят, что этот объект *«имеет фокус»*. Форма (или окно), который принадлежит объект, имеющий фокус (или же «фокус ввода» - это синоним термина «фокус»), выделяется более ярким цветом ее строки заголовка и главного меню, если, конечно, таковое имеется. Пользователь может перенести фокус ввода на другой объект. Для этого достаточно щелкнуть на нем кнопкой мыши. Переключать фокус между объектами пользователь может также с помощью клавиши «Tab». Кроме того, переключать фокус можно программно.

Среда разработки Delphi

После запуска системы Delphi на экране появляется ее окно графического пользовательского интерфейса, содержащее главное меню, несколько панелей инструментов, палитру компонентов и несколько окон - это окно формы (Form1), окно редактора кода (Unit1.pas), окно инспектора объектов (Object Inspector) и др.

Палитра компонентов представляет собой панель с вкладками, обеспечивающую быстрый доступ к компонентам VCL. Компоненты VCL размещаются в формах. Форма, собственно, и является разрабатываемым пользовательским интерфейсом. В окне формы (по умолчанию она называется Form1) ее можно редактировать, т.е. изменять размер и цвет, можно размещать на ней компоненты и т.д.

В среде разработки Delphi используется объектно-ориентированный язык Delphi. Все используемые в Delphi компоненты являются *объектами.* С каждым объектом ассоциирован набор *свойств.* Например, форма является объектом с такими свойствами, как Name (Имя), Caption (Заголовок), ClientHeight (Высота), ClientWidth (Ширина), Color (Цвет) и т.д. При создании новой формы Delphi автоматически присваивает каждому ее свойству *значение по умолчанию*. При разработке программы (и графического пользовательского интерфейса) свойства компонентов можно изменять с помощью диалогового окна Object Inspector (Инспектор объектов). Для этого нужно всего лишь щелкнуть на свойстве кнопкой мыши и заменить его текущее значение желаемым.

Редактор кода представляет собой полнофункциональный текстовой редактор, с помощью которого можно просматривать и редактировать исходный код программы. Кроме того, редактор кода содержит многочисленные средства, упрощающие создание исходного кода на Delphi. В окне редактора кода каждый модуль кода вводится в отдельной вкладке. Чтобы открыть модуль в редакторе кода, выберите команду View> Units (Просмотр> Модуль) или нажмите комбинацию клавиш <Ctrl+ F12>. Затем в окне View Unit выделите имя нужного модуля и щелкните на кнопке OK.

Размещение, изменение размеров и передвижение компонентов

Разместить компонент в форме можно одним из двух способов. Первый – дважды щелкнуть кнопкой мыши на пиктограмме компонента, расположенной на палитре компонентов. Однако при этом компонента попадает не в то место, куда вам требуется, а в середину формы. Второй – щелкнуть на пиктограмме компонента (при этом она выделится) и в форме. Подобным образом компонент можно сразу поместить в требуемое место в форме – точка, в который выполнен щелчок, задает расположение его верхнего левого угла.

Перемещение компонентов в форме выполняется с помощью стандартного для приложений Windows метода перетаскивания. Для этого нужно выполнить следующие: поместить указатель мыши на компонент, нажать левую кнопку мыши и, не отпуская ее, переместить компонент в другое место, после чего отпустить кнопку мыши. Во время передвижения компонента в форме под указателем мыши появляется подсказка с координатами компонента в пикселях относительно левого верхнего угла формы.

Для изменения размеров требуемый компонент должен быть сначала выделен. Для этого нужно щелкнуть на нем левой кнопкой мыши. (Ни в коем случае не щелкните два раза! Для многих компонентов двойной щелчок создает обработчик события, который вам сейчас не нужен). Выделенный компонент отмечается по периметру черными квадратиками. Теперь нужно поместить указатель мыши на один из этих квадратиков, перетащить его (указатель вместе с квадратиком) в новое место и отпустить кнопку мыши. Во время передвижения под указателем мыши появится подсказка с текущими размерами компонента в пикселях.

Работая в Delphi, представьте себе, что вы художник, а форма-это ваш холст. Вы рисуете пользовательский интерфейс приложения путем размещения в форме компонентов. Затем, изменяя свойства компонентов, изменяете их внешний вид. После этого разрабатываете средства взаимодействия компонентов с пользователем. Взаимодействие компонентов формы с пользователем означает некоторую реакцию компонентов на действия пользователя.

Проект Delphi

Программу, создаваемую в среде Delphi, принято называть *проектом* (или *приложением*), из-за сложности ее организации. *Проект Delphi* состоит из *нескольких файлов*. Наиболее важные файлы – это файла проекта (Delphi Project) с расширением **.dpr,* одного или нескольких файлов модулей (Unit) с расширением *.pas и файлов дизайнера экранных форм с расширением *.dfm. Файл проекта содержит текст основной программы Program, с которой начинается выполнение всей программы. Тексты вызываемых подпрограмм и используемых объектов находятся в файлах модулей. Такой проект или приложение еще называется *оконным приложением*, ибо общение пользователя такой программой осуществляется через экранной формы, обладающей свойством окна Windows. Основная программа проекта выглядит так:

```
program Project1;//P roject1.dpr

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Главная программа содержит действия объекта ***Application***:

- ***Initialize*** – инициализация приложения
- ***CreateForm(TForm1, Form1);*** - создание объекта Form1 класса TForm1
- ***Run*** – ожидание события и ответ на них, т.е. выполнение приложения.

Таким образом, файл проекта связывает вместе все остальные файлы, входящие в данный проект. Между файлами модулей и форм существует примерно однозначное соответствие: с каждым файлом модуля ассоциирован один файл формы и наоборот. Файл формы имеет расширение **.dfm*; в нем перечислены объекты формы и значения свойств объектов. Файл модуля имеет расширение **.pas*, в нем находится исходный код, ассоциированный с формой. При выполнении команды File- New – Application на экране будет выведен следующий текст модуля unit Unit1.

```
unit Unit1;   // файл Unit1.pas
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
Type
  TForm1 = class(TForm)  //  класс по имени TForm1 является потомком класса TForm
```

```
                              //  и потому наследует все поля и методы предка, иначе
                              //  говоря, обеспечена возможность повторного
                              //  использования программного кода, соответствующий
                              //  классу TForm
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;              // переменная Form1 объявляется как переменная типа
                              // TForm1, точнее является объектом, т.е. экземпляром
                              // класса TForm1. А это означает, что объект Form1
                              // будет обеспечен программным кодом, соответствующий
                              // классу TForm при создании объекта Form1. Потому если
                              // выполнить эту программу,
                              // то на экране будет напечатана пустая форма (окно) с
                              // кнопками, хотя нами не был написан никакой код. Это
                              //  является результатом использования принципа
                              // наследования;

implementation

{$R *.dfm}

end.
```

Программа (проект) создается в среде Delphi в *визуальном режиме* – с помощью встроенного *Проектировщика форм*. Разработчик размещает на форме компоненты (элементы управления) из обширной библиотеки визуальных компонентов, настраивает их свойства в наглядном *Инспекторе объектов* и при необходимости задает реакции на различные действия пользователя, например на нажатие кнопки, выбор пункта меню, изменение состояния переключателей. Реакция программы на подобные действия вводится записью непосредственно на языке программирования Delphi, в *окне редакторе кода*, имеющем специальные средства наглядной визуализации исходного кода и контекстной помощи. Запись делается на автоматически созданном шаблоне процедуры – *процедуры обработчика события* – которая будет выполняться, как реакция программы на действие пользователя.

Компиляция, сборка и запуск программы выполняются полностью в автоматическом режиме. При обнаружении ошибок среда подскажет сомнительные места в исходном тексте, а при сбое в работающей программе выдаст подробный протокол, раскрывающий возможную неисправность.

Скомпилированная программа является *выполняемым модулем* (т.е. кодом на машинном языке) и хранится в файле с расширением *.exe.* Она может быть выполнена на компьютере без среды разработки Delphi.

Вы должны периодически сохранять файлы проектов, модулей и форм разрабатываемого вами приложения Delphi. При сохранении файла формы на диск записываются и сохраняются как файл формы, так и файл модуля, причем под одним и тем же именем, но с соответствующими расширениями. Чтобы сохранить форму, нужно в главном меню выбрать команду File=>Save(Файл=>Сохранить) или File=>Save As..(Файл=>Сохранить как..). Другой способ – щелкнуть кнопкой мыши на пиктограмме Save стандартной панели инструментов. Еще один способ - нажать комбинацию клавиш <Ctrl+S>. Для сохранения файла проекта нужно в главном меню выбрать команду File=>Save Project As..(Файл => Сохранить проект как..). Чтобы быстро сохранить все файлы, составляющие проект, нужно в главном меню выбрать команду File=>Save

All (Файл=>Сохранить все) или щелкнуть кнопкой мыши на пиктограмме Save All стандартной панели инструментов.

Компоненты Form, Label, Edit, Button

Каждый компонент из палитры компонентов представляет собой разновидность класса (объекта), который можно перенести в приложение из палитры. Компонент обладает набором свойств, методов и событий. Свойства можно изменять, не изменяя исходный код программы. Рассмотрим основные компоненты: ***Form, Label, Edit и Button.***

Компонент ***Form*** **–** выбранные **свойства:**

- **Name -**имя
- **Width, Height –** ширина и высота в пикселях
- **Caption –** заголовок
- **Color** - цвет
- **Menu, PopupMenu -** главное и подсобное меню
- **Constraints –** минимальный и максимальный размеры формы
- **Font –** шрифт всех компонентов на форме

Компонент ***Form*** – выбранные методы:

- **Create –** создание формы
- **Show –** показание формы
- **Close –** закрытие формы

Компонент ***Form*** **–** выбранные **события:**

- **OnCreate** – при создании формы
- **OnClick** – при нажатии клавиша мыши на форме

Компонент ***Label*** **–** выбранные **свойства:**

- **Caption** – заголовок
- **Color** - цвет
- **Font –** шрифт

Компонент ***Edit*** **–** выбранные **свойства:**

- **Text** – текст внутри окна
- **Color** - цвет
- **Font –** шрифт
- **ReadOnly** – только для чтения
- **Enabled** – доступность

Компонент ***Edit*** **–** выбранные **методы:**

- **Clear** – очистка окна
- **SelectAll** – выбор всего текста
- **CopyToClipBoard, CutToClipboard, PasteFromClipboard** – копировка, уничтожение и вставка текста из буфера.

Компонент ***Edit*** **–** выбранные **события:**

- **OnChange** – при изменении содержимого текста в окне

- **OnKeyPress –** при нажатии клавиша клавиатуры

Компонент ***Button*** **–** выбранные **свойства:**

- **Caption** – заголовок кнопки

Компонент ***Button*** **–** выбранные **события:**

- **OnClick -** при нажатии клавиша мыши на кнопке

Примеры использования в визуальном программировании часто используемых простых компонентов типа TButton, TLabel, TEdit

Использование компонента Button

Обычно с помощью компонента кнопка (типа Tbutton) пользователь инициирует выполнение какого – либо фрагмента кода или целой программы. Другими словами, если щелкнуть на элементе управления Tbutton, то программа выполнит определенное действие. При этом кнопка примет такой вид, будто она действительно была нажата. Кнопкам можно присваивать комбинации клавиш быстрого вызова. Во время выполнения программы нажатие такой комбинации клавиш эквивалентно щелкнуть на кнопке левой кнопкой мыши. Выполните следующие действие.

1. Выбрав команду File > New > Application, создайте новый проект.
2. В инспекторе объектов измените значение свойства формы Name на frmButtonxample, а свойства Caption – на Пример кнопки.
3. Поместите кнопку в форму. Для этого дважды щелкните на ее пиктограмме и в любом месте формы.
4. Измените значение свойства Name кнопки на btnMyButton. Для этого в инспекторе объектов щелкните на свойство Name и введите btnMyButton. Убедитесь, что вы изменили свойство кнопки, а не формы. В заголовке раскрывающегося списка в верхней части инспектора объектов должно быть написано Button1: Tbutton, а после изменение имени – btnMyButton: Tbutton.
5. Измените значение свойство Caption кнопки на &Щелкните здесь. Обратите внимание: в заголовке кнопки буква, перед которой стоит символ &, оказалась подчеркнутой. В данном случае это буква Щ. Это означает, что теперь кнопке присвоена комбинации клавиш быстрого вызова < Alt + Щ >.
6. Измените размер и положение кнопки.
7. Нажав клавишу < F9 >, запустите программу на выполнение. При этом на экране появиться изображение, показанное на рис.2.28.
8. Щелкните на кнопке. Обратите внимание, что в момент щелчка кнопка принимает такой вид, как будто она “ вдавлена “.
9. Активизируйте кнопку, нажав комбинацию клавиш < Alt + Щ >. Как видите, при активизации с помощью клавиш быстрого вызова кнопка не принимает вид вдавленной. Пока еще с кнопкой не связен какой – либо фрагмент кода, поэтому никакой реакции кнопки на ее активизацию не видно. Тем не менее, можете нам поверить, она активизирована.
10. Щелкнув на кнопке с крестиком в правом верхнем углу формы, завершите работу программы.
11. Заголовок кнопки btnMyButton выглядит как Щелкните здесь, а не Щелкните здесь. Символ g, расположенный перед любой буквой значения свойства Caption,

присваивает кнопке комбинацию клавиш быстрого вызова. В заголовке кнопке буква, перед которой стоит символ g, подчеркнута. Это сообщают пользователю о том, что с кнопкой связана комбинация клавиш быстрого вызова. Во время выполнения пользователь может активизировать кнопку с помощью клавиатуры. Для этого нужно нажать клавишу <Alt>и, удерживая ее, нажать клавишу с подчеркнутой буквой.

В данном примере клавиатура должна быть переключена на русский язык (ведь буква “ Щ “ – русская). Кроме того, на клавиатуре должна быть предварительно нажата клавиша < Caps Lock > (лампочка соответствующего режима включена), так как подчеркнута прописная буква. Активизировать кнопку можно также, нажав комбинацию клавиш < Alt + Shift + Щ >. Во многих приложениях русская буква в верхнем регистре срабатывает без включения режима CapsLock или нажатия клавиши < Shift >, однако для компонента Delphi это недоступно (по крайне мере, без специальных ухищрений). Кроме того, если в данный момент курсор ввода не находится в одном из текстовых полей формы, срабатывание кнопки происходит просто поле нажатия клавиш < Щ > (или < Shift + Щ >), без использования клавиши < Alt >.

Что делать, если в заголовке кнопки должен присутствовать символ & ? Ведь если поместить его в заголовок, то он сделает следующую букву подчеркнутой, а сам виден не будет. Чтобы решить эту проблему, используйте следующие правило: символ & отображения в заголовке кнопки, если в свойстве Caption записаны два стоящих подряд символа - &&. Например, чтобы заголовок кнопки имел вид This & That, в свойстве Caption должно быть написано This && That. При этом никакая комбинации клавиш быстрого вызова кнопке не присваивается.

Как видите, с помощью среды разработки Delphi можно буквально за минуту создать простую форму (и программу). Аналогично создаются самые сложные пользовательские интерфейсы для Windows. Теперь вы знакомы со свойствами некоторых наиболее распространенных компонентов Delphi – формами, надписями, полями ввода, областями просмотрам и кнопками. Чтобы закрепить полученные знания, поэкспериментируйте с этими компонентами. Попробуйте изменять другие их свойства. Не бойтесь: самое худшее, что вы можете сделать, - это испортить среду Delphi (хотя и это крайне маловероятно). Тогда вам придется всего лишь установить ее заново.

Использование компонента Label

Компонент надпись (типа Tlabel) используется для помещения в форму текста, который пользователь не может изменить непосредственно. Рассмотрим методику работы с надписями на конкретном примере. Выполните следующие действия.

1. Выбрав команду File =>New =>Application, создайте новый проект.
2. Поместите компонент надписи в формую. Для этого просто дважды щелкните кнопкой мыши на пиктограмме надписи на палитре компонентов. Другой способ размещения надписи в форме – однократно щелкнуть кнопкой мыши на пиктограмме надписи на палитре компонентов, а затем однократно щелкнуть в произвольном месте формы. Такой способ удобен тем, что надпись в этом случае размещается там, где вы хотите. Если вместо щелчка в требуемом месте формы только нажать кнопку мыши, а затем перетащить курсор по диагонали, то надписи можно сразу же придать желаемый размер. Чтобы удалить надпись из формы нужно выделить ее (щелкнуть на ней левой кнопкой мыши; при этом она выделится черными квадратиками) и нажать клавишу <Delete> . Чтобы отменить выделение, нужно щелкнуть кнопкой мыши в любом свободном месте формы за пределами надписи. Поэкспериментируйте с размещением и удалением надписей.
3. Переместить надпись в другое место в формате методом перетаскивания. Для этого установите указатель мыши на надписи, нажмите кнопку мыши и, удерживая ее нажатой, передвиньте надпись в другое место формы. Когда надпись займет нужное положение, отпустите кнопку мыши.

4. Измените свойство надписи Name (Имя надписи) на lbMyLabel (по умолчанию она называлась Label1. Для этого в инспекторе объектов щелкните на свойстве Name и введите lblMyLabel. Убедитесь, что вы изменяете свойство надписи, а не формы (это типичная ошибка новичков). Для этого надпись в форме должна быть выделена, а в заголовке раскрывающегося списка в верхней части инспектора объектов должно быть написано Label1: Tlabel (когда вы измените имя надписи, там будет написаноlblMyLabel: Tlabel). После ввода нужного имени надписи зафиксируйте его, нажав клавишу <Enter>.
5. Обратите внимание: текст надписи в форме изменился на IBIMyLabel. Это объясняется тем, что по умолчанию текст надписи совпадает с ее именем. Измените текст надписи явно. Для этого выберите в инспекторе объектов свойство Caption (Заголовок надписи), которое определяет содержании надписи при ее отображении в форме, введите в него новое значение Это моя первая надпись! и нажмите клавишу < Enter > . Введенный текст должен появиться в форме.
6. Измените цвет фона надписи. Для этого выберите свойство Color (Цвет фона), щелкните на стрелке, выберите в раскрывшемся списке желтый цвет и щелкните на нем.
7. Измените шрифт и цвет надписи. Для этого выберите свойство Font(Шрифт) и щелкните на кнопке с тремя точками. В окне Font измените шрифт на Arial, начертание - на Bold Italic, а размер – на 20. В раскрывающемся списке Цвет выберите красный цвет и щелкните на кнопке ОК.
8. Добавьте в форму еще одну надпись. На этот раз воспользуйтесь другим методом- щелкните на пиктограмме надписи на палитре компонентов, переместите указатель мыши в произвольное место формы и еще раз щелкните кнопкой мыши. При этом в форме в указанном вами месте должна появиться новая надпись.
9. Измените свойство Name новой надписи на lblAnother, а свойство Caption- на *Еще одна надпись.*
10. Добавив в форму две надписи, выделите теперь саму форму-объект Form1. Сделать это можно одним из двух способов: щелкнуть в любом свободном месте формы за пределами надписей или выбрать значение Form1 в раскрывающемся списке в окне инспектора объектов. Если форма видна, то первый способ, конечно, удобнее, однако если в проекте есть много форм, причем нужная форма закрыта другими окнами, то более удобен второй способ.
11. Теперь измените свойства формы Form1: свойству Name присвойте значение frmLabelExample, а свойству Caption- Пример надписи.
12. Итак, сами того не заметив, вы создали простое приложение, которое, правда, пока что ничего не делает. Выполните его. Это можно сделать одним из трех способов: щелкнув на кнопку Run (Выполнить) панели инструментов отладки, выбрав в главном меню команду Run=>Run или нажав клавишу<F9>.При этом на экране должна появиться форма.
13. Щелкнув на кнопке с крестиком в верхнем правом углу формы, завершите приложение. То же самое можно сделать и в среде Delphi. Для этого запустите приложение еще раз, активизируйте любое окно Delphi (однократно щелкнув на нем кнопкой мыши) и выберите Run=>Program Reset или нажмите комбинацию клавиш Ctrl+F2.

Использование компоненты Edit

В компоненте поле ввода (типа TEdit) хранится текст, который можно помещать в данный компонент как во время разработки, так и во время выполнения. Текст, видимый в поле ввода, находится в свойстве Text этого объекта. Свойство MaxLength определяется максимальное количество символов в поле ввода. Если значения свойства MaxLegth равно 0, то количество символов ничем не ограничено. С помощью свойства Font можно устанавливать шрифт текста (

свойство Font является дочерним объектом объекта класса TEdit). Если свойство ReadOnly (Только чтение) установить равным True (Истина), то во время выполнения программы пользователь не сможет изменять текст в этом поле ввода. Чтобы лучше усвоить приемы работы с полями ввода, выполните следующие действия.

1. Выбрав команду File>New>Application, создай новый проект.
2. Поместите поле ввода в форму. Как и в случае надписи, это можно сделать одним из двух способов: дважды щелкнуть на пиктограмме поля ввода на палитре компонентов или щелкнуть на ней один раз, а затем щелкнуть в произвольном месте формы.
3. Измените размер поля ввода. Для этого установите указатель мыши на одном из черных квадратиков и, удерживая кнопку мыши нажатой, перетащите этот квадратик (а с ним и границу поля ввода) в нужном направлении. Установив необходимый размер, отпустите кнопку мыши. (Если черных квадратиков вокруг поля ввода нет, значит, оно не выделено. В этом случае сначала выделите поле ввода, щелкнув на нем кнопкой мыши.)
4. Переместите поле ввода в другое место методом **перетаскивания.** Для этого установите указатель мыши на поле ввода, нажмите кнопку мыши и, удерживая ее нажатой, передвиньте поле ввода в новое место. Когда поле ввода займет нужное положение, отпустите кнопку мыши.
5. Установите значение свойства Name равным edtMyText. Для этого в инспекторе объектов щелкните на свойстве Name и введите edtMyText. Как и в случае с надписью, убедитесь, что вы изменяете свойство *поля ввода*, а не формы. В заголовке раскрывающегося списка в верхней части инспектора объектов должно быть написано Edit1:TEdit (после изменения свойства Name здесь будет написано edtMyText:TEdit).
6. Выберите в инспекторе объектов свойство Text и введите его новое значение: Это элемент управления “Поля ввода”. Нажав клавишу <Enter>,зафиксируйте введенный текст. Обратите внимание: во время ввода одновременно изменяется содержимое поля ввода в форме.
7. Измените цвет текста в поле ввода на синий. Для этого в инспекторе объектов щелкните на значке (+) слева от имени свойства Font. Значок(+) изменится на (-) и раскроется список свойств объекта Font. Выберите свойство Color, щелкните на стрелке, расположенной в этом поле, и раскроется список доступных цветов. Найдите в нем синий цвет и щелкните на нем.
8. Выделите форму. Это можно сделать одним из двух способов: щелкнув в любом месте формы за пределами поля ввода или выбрав имя формы в раскрывающемся списке в верхней части инспектора объектов. Измените свойство формы Name на frmEditBoxExample, а свойство Caption- на *Пример поля ввода*.
9. Нажав клавишу<F9>, запустите разработанную программу на выполнение. На экране появится изображение. Поэкспериментируйте с полем ввода. Введите в него какой-либо текст.
10. Для завершения программы щелкните на кнопке с крестиком в правом верхнем углу формы.
11. Установите значение свойства RealOnly поля ввода равным True.
12. Нажав клавишу<F9>, запустите программу на выполнение еще раз. Попытайтесь изменить содержимое поля ввода : как видите, теперь изменить его содержимое во время выполнения нельзя. Возможно, вы удивитесь: зачем может понадобиться поле ввода, в которое нельзя ничего ввести? Однако в дальнейшем вы увидите, что это довольно полезное средство, так как значение свойства ReadOnli можно менять из программы, запрещая или разрешая таким образом пользователю вводит данные.

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Пример вычислительной программы

Ввод данных в оконном приложении происходит в виде строки, поэтому, вводя и выводя числа нужно пользоваться функциями преобразования типа. Поэтому используя оконное приложение для расчета нужно:

1. Ввести строку данных (тип string)
2. Преобразовать данные строкового типа на численный тип
3. Провести вычисления
4. Результат вычислении числового типа преобразовать в строковый тип
5. Вывести результаты, используя строковые свойства компонентов

Задача.

Рассмотрим пример вычисления суммы, разницы, умножения и деления двух целых чисел.

1. Графический интерфейс пользователя

Создание оконного приложения в среде Delphi начинается с разработки интерфейса пользователя. Разместив в форме компоненты, следует установить значения их свойств.

Для ввода данных этой задачи применим два компонента ***Edit*1, *Edit*2** типа ***TEdit***, для вывода результатов четыре компонента ***Label1, Label2, Label3, Label4*** типа ***TLabel.*** Кнопка ***Button1*** типа ***Tbutton*** будет использована для вычисления результатов.

С помощью окна ***инспектора объектов*** (карта *Properties*) свойство *Text* всех компонентов типа ***TEdit*** становим равным пустой строке, свойство *Caption* компонента ***Button1*** становим равным тексту *Результаты*

2. Процедуры обслуживания событий

С каждым компонентом может быть ассоциировано некоторое событие. Для этих и других событий операционная система должна соответствующим образом реагировать. Программист должен создавать подпрограммы, выполняющиеся при наступлении этих событий. Поэтому после создания формы и установки новых значений свойств компонентов создается исходный код этих подпрограмм. Такие подпрограммы называются *обработчиками событий*. Обработчик – это подпрограмма специального вида: ее вызывает не оператор исходного кода, а операционная система при наступлении ассоциированного с обработчиком события.

В этой задаче с помощью окна ***инспектора объектов*** (карта *Events*) для события **OnClick** компонента ***Button1*** создаем процедуру ***Button1Click*** для вычисления результатов***:***

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  label1.Caption:='сумма='+IntToStr(StrToInt(Edit1.Text)+ StrToInt(Edit2.Text));
  label2.Caption:='разница='+IntToStr(StrToInt(Edit1.Text)- StrToInt(Edit2.Text));
  label3.Caption:='умножение='+IntToStr(StrToInt(Edit1.Text)*StrToInt(Edit2.Text));
  label4.Caption:='деление='+FloatToStr(StrToInt(Edit1.Text)/ StrToInt(Edit2.Text));
end;
```

Для обеспечения программы от ошибок ввода плохих данных используем оператор обслуживания исключений ***try... except... end***:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
try
  label1.Caption:='сумма='+IntToStr(StrToInt(Edit1.Text)+ StrToInt(Edit2.Text));
  label2.Caption:='разница='+IntToStr(StrToInt(Edit1.Text)- StrToInt(Edit2.Text));
  label3.Caption:='умножение='+IntToStr(StrToInt(Edit1.Text)*StrToInt(Edit2.Text));
```

```
  label4.Caption:='деление='+FloatToStr(StrToInt(Edit1.Text)/ StrToInt(Edit2.Text));
except
  ShowMessage('ошибка');
end;
end;
```

Процедура ShowMessage('сообщение') выводит текст сообщения в отдельном окне. Для различения ошибок разного типа используем оператор ***on...do***:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
try
  label1.Caption:='сумма='+IntToStr(StrToInt(Edit1.Text)+ StrToInt(Edit2.Text));
  label2.Caption:='разница='+IntToStr(StrToInt(Edit1.Text)- StrToInt(Edit2.Text));
  label3.Caption:='умножение='+IntToStr(StrToInt(Edit1.Text)*StrToInt(Edit2.Text));
  label4.Caption:='деление='+FloatToStr(StrToInt(Edit1.Text)/ StrToInt(Edit2.Text));
except
  On EConvertError do ShowMessage ('Ошибка преобразования типов');
  On EZeroDivide do ShowMessage ('Ошибка деления на нуль');
end;
end;
```

Этот обработчик события *OnClick* компоненты Button1 автоматически вызывается операционной системой при нажатии клавиши Button1. После выполнения на графическом пользовательском интерфейсе будут выданы результаты решения задачи.

Резюме

В современном объектно-ориентированном программировании большое значение имеют понятия «*события*» (которое приводит, так называемому событийно-ориентированному программированию) и «*компонента*» (которое приводит, так называемому компонентному программированию).

Событие возникает при воздействии на компонент какими-нибудь манипуляциями мышью, при вводе с клавиатуры, при перемещении окна, изменения его размеров. Объект, в котором произошло событие, называется источником события. Подпрограмма, автоматически вызываемая операционной системой при наступлении события называется *обработчиком события.*

В операционной системе Windows событие – это какое-либо действие пользователя: щелчок кнопкой мыши, нажатие клавиши и т.д. При запуске и завершении приложения автоматически генерируются определенная последовательность событий, которые можно использовать в программе.

Когда приложение Delphi начинает работу, одно за другим происходят события OnCreate, OnShow, OnPaint, OnActivate в указанной последовательности. Событие OnCreate происходит при создании формы, OnShow – при выводе формы на экран (когда ее свойство Visible принимает значение true), OnPaint – при получении формой сообщения Windows о необходимости заполнения формы рисунком, OnActivate – когда форма становится активной.

По завершении работы приложения Delphi происходят события OnClose, OnDestroy в указанной последовательности. Событие OnClose происходит перед закрытием формы, а OnDestroy – перед удалением формы из памяти.

Визуальное программирование во многом автоматизирует труд программиста по написанию программ. Визуальное программирование – одна из самых популярных парадигм программирования на данный момент. Оно базируется на технологии объектно-ориентированного программирования. Основным элементом в средствах визуального программирования является

компонент. Компонент бывают визуальными и невизуальными. Технология визуального программирования состоит в следующем: *создание экранных форм, нанесение визуальных и невизуальных компонент, программирование событий и методов оконных форм.*

Вопросы для самопроверки

1. В чем смысл событийного программирования?
2. Назовите этапные работы визуального программирования?
3. Как делаются вычисления в оконном приложении?
4. Чем отличается консольное и оконное приложения?
5. Какая разница между программой и приложением?
6. В чем суть автоматизации разработки при визуальном программировании?
7. Что такое повторное использование кода?
8. Что такое графический пользовательский интерфейс?
9. Имеет ли консольное приложение графический пользовательский интерфейс?
10. Что такое компонента?
11. Чем отличается компонента от подпрограммы?
12. Что значит событийно-управляемая программа?
13. Что такое «скелет» метода?
14. Каким образом строится «скелет» метода?
15. Какая разница между классом и компонентом?

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Лекция 16. Библиотека компонентов VCL Delphi

План

О библиотеке компонентов VCL Delphi
Компоненты карты Standard
Пример использования компоненты TMemo в визуальном программировании
Заключение – сравнение парадигм программирования

О библиотеке компонентов VCL Delphi

При изучении языка программирования высокого уровня необходимо освоить две вещи. Первое, это познакомиться с самим языком, чтобы вы могли создавать свои собственные программы и подпрограммы (в виде компоненты, классы), а второе, это научиться использовать огромное количество разработанных другими подпрограмм (классов, компонент). В практике программирования разработанные подпрограммы оформлялись для использования другими в виде библиотек процедур и функций, библиотек модулей, библиотек классов и компонент.

Процесс создания программы должен выглядеть как построение здания из отдельных кирпичей. Не изобретайте велосипед, используйте фрагменты существующих программ. Называется такой *подход повторным использованием кода* и является одной из основных идей объектно-ориентированного программирования. Основная идея улучшения процесса разработки программного обеспечения состоит в *повторном использовании программного кода*.

Библиотека компонентов VCL Delphi (Visual Component Library – библиотека визуальных компонентов), одна из многих мощных и полезных библиотек классов и компонент, в которых максимально используются выгоды многократного использования программного обеспечения благодаря использованию механизма наследования.

Компоненты, размещаемые на экранной форме, являются к*омпонентами библиотеки VCL* Delphi. Библиотека визуальных компонентов Delphi состоит достаточно из большого количества компонентов и мы не состоянии все их здесь рассмотреть.

Компоненты VCL служат строительными блоками графического пользовательского интерфейса любого приложения Delphi, а форма является основой разрабатываемого пользовательского интерфейса. Во *время выполнения* приложения компоненты VCL появляются на экране *как элементы управления* - кнопки, флажки, списки, поля ввода и т.д. Элементы управления составляют подмножество компонентов VCL: каждый элемент управления является компонентом, но не каждый компонент является элементом управления.

Разработка программы, основывающаяся *на выборе числа компонентов* и *определения их взаимодействия,* является сутью *визуального программирования в Delphi*. Реальная сила визуального программирования заложена в огромной библиотеке компонентов. Визуальное программирование с использованием компонентов библиотеки VCL – ключевая функциональная возможность среды разработки Delphi.

Итак, среда программирования Delphi поставляется пользователю с большим набором готовых к использованию компонентов. В среде Delphi эти компоненты размещаются на, так называемой, Палитре компонентов. *Палитра компонентов* состоит из, так называемых, *панелей с вкладками* или *карт.* По этим картам распределены все компоненты библиотеки VCL.

Компоненты карты Standard

Палитра компонентов представляет собой панель с вкладками, обеспечивающую быстрый доступ к разным компонентам VCL. Рассмотрим только некоторые выбранные компоненты VCL карты ***Standard*** (Стандарт) с некоторыми выбранными свойствами.

Компонент ***Memo*** **–** выбранные **свойства:**

- **Name –**имя
- ***Text*** **–** текст окна компоненты в виде одной строки
- ***Lines*** **–** строки текста
- ***Lines[i]*** **–** i-я строка в Memo
- ***Lines.Count*** **–** количество строк
- **Color** - цвет
- **Font –** шрифт всех компонентов на форме

Компонент ***Memo*** – выбранные методы:

- **Clear** – очистка всего окна компоненты
- **ClearSelection** – очистка выделенного текста
- **CopyToClipBoard, CutToClipboard, PasteFromClipboard** – копировка выделенного текста в **ClipBoard** , перенос выделенного текста в **ClipBoard** и уничтожение его в окне, переносит в окно в позицию SelStart текст из буфера **ClipBoard**.
- **Lines.LoadFromFile(nazwa:string)** – загрузка в Memo содержимого текстового файла с указанным названием.
- **Lines.SaveToFile(nazw:string)** – запись текста из Memo в текстовой файл с указанным названием.
- **Lines.Add(s:string) –** добавление строки **s** в ***Memo***
- **Lines.Delete(i) –** удаление **i-**той строки

Компонент ***Memo*** **–** выбранные **события:**

- **OnChange** – наступает, когда текст в окне может быть изменился. Свойство Modified показывает, действительно ли произошло изменение.

Компонент ***ComboBox*** **–** выбранные **свойства:**

- **Items** – массив строк списка**.** Свойство позволяет формировать и изменять список.
- **Items[i] -** I –я строка
- **ItemIndex –** указывает порядковый номер элемента, выделенного в списке.
- **Count –** количество строк в списке
- **Sorted –** указывает, должны ли строки в списке автоматически сортироваться в алфавитном порядке.
- **Text –** текст выбранный или написанный пользователем строки.
- **Style –** определяет стиль отображения списка.

Компонент ***ComboBox*** – выбранные методы:

- **Items.Add(tekst) –** tekst добавляется в список.
- **Items.Delete(i) –** удаление i –й строки списка.
- **Items.Insert(tekst, i) –** вставка tekst в i-ю строку списка

Компонент ***ComboBox*** **–** выбранные **события:**

- **OnChange** – наступает при изменении текста в окне редактирования в результате прямого редактирования теста или в результате выбора из списка. В обработчике можно прочитать текст Text и индекс выбранного элемента ItemIndex (-1, если был не выбор, а редактирование).

OnClick – наступает при щелчке на элементе списка.

Компонент ***ListBox*** **–** выбранные **свойства:**

- **Items –** массив строк списка**.**
- **Items[i] –** i –я строка
- **ItemIndex –** указывает порядковый номер элемента, выделенного в списке.
- **Count –** количество строк в списке
- **Sorted –** указывает, должны ли строки в списке автоматически сортироваться в алфавитном порядке.
- **Columns** – определяет число столбцов, в которых будет отображаться список, если он не помещается целиком в окне компонента.
- **MultiSelect –** указывает, можно ли выбрать в окне списка несколько элементов одновременно.
- **ExtendedSelect –** определяет, может ли пользователь при MultiSelect=true выбрать несколько последовательно расположенных элементов, держа нажатой клавишу Shift.

Компонент ***ListBox*** – выбранные методы:

- **Items.Add(tekst) -** tekst добавляется в список.
- **Items.Delete(i), -** удаление i –й строки списка.
- **Items.Insert(tekst, i) -** вставка tekst в i-ю строку списка

Компонент ***ListBox*** **–** выбранные **события:**

- **OnClick -** наступает при щелчке на элементе списка.

Компоненты (индикаторы) ***CheckBox, RadioButton*** **–** выбранные **свойства:**

- **Caption –** надпись индикатора
- **Checked** – указывает выбран ли индикатор (содержит ли флажок или точка)

Компонент (панель) ***RadioGroup*** **–** выбранные **свойства:**

- **Caption –** надпись в левом верхнем углу панели
- **Items –** список радиокнопок группы.
- **ItemIndex –** указывает, какая из радиокнопок выбрана в данный момент.
- **Columns -** определяет количество столбцов кнопок в радиогруппе.

Компонент (групповая панель) ***GroupBox*****–** выбранные **свойства:**

- **Caption –** надпись в углу рамки панели.

Компонент ***Panel*****–** выбранные **свойства:**

- ***Caption*** **-** текст, отображаемый в панели.
- ***Width, Height*** – ширина, высота панели.
- ***Color*** **–** цвет панели.

Компоненты ***CheckBox, RadioButton, RadioGroup, GroupBox, Panel*** **-** выбранные **события:**

- **OnClick -** наступает при щелчке на компоненте.

Компонент ***ScrollBar*****–** выбранные **свойства:**

- **Position –** определяет текущую позицию ползунка в диапазоне

- **Min –** задает минимальное значение изменения возможных значений
- **Max** – задает максимальное значение изменения возможных значений

Компонент ***ScrollBar –*** выбранные **события:**

- **OnChange** – наступает при изменении позиции ползунка.

Пример использования компонента Memo в визуальном программировании

Компонент область просмотра (типа TMemo) предназначен для вывода на экран сразу нескольких строк текста. Свойства MaxLength,Font и ReadOnly области просмотра аналогичны этим же свойствам компонента поля ввода. Свойство Text содержит весь текст области просмотра, однако это свойство доступно только по время выполнения. Свойство Lines содержит отдельные строки текста области просмотра, оно доступно как во время разработки, так и во время выполнения. Свойство WordWarp определяет, будут ли переноситься строки, выходящие за пределы области просмотра, или выступающие части строк останутся невидимыми.

При выводе в область просмотра текста на русском языке необходимо учитывать, что не все шрифты оснащены кириллицей. Кроме того, некириллической может оказаться текущая кодовая таблица. Если вместо русского текста на экране появилась абракадабра, измените значение свойства Charset (Набор символов) объекта Font (Шрифт). Для большинства шрифтов удовлетворительными значениями свойства Charset являются DEFAULT_CHARSET и RUSSIAN_CHARSET.

Для практического усвоения приемов работы с областью просмотра выполните следующие действия.

1. Выбрав команду File>New>Application, создайте новый проект.
2. Поместите область просмотра в форму. Это делается точно так же, как и в случае надписи или поля ввода.
3. Измените размер области просмотра. Для этого установите указатель мыши на одном из черных квадратиков по периметру области просмотра, нажмите кнопку мыши, перетащите квадратик в нужном месте и отпустите кнопку мыши.
4. Переместите область просмотра в другое место методом перетаскивания. Для этого поместите указатель мыши в область просмотра, нажмите кнопку и, удерживая ее нажатой, передвиньте область просмотра в новое место. После этого отпустите кнопку мыши.
5. Измените свойство Name области просмотра на memSample. Для этого в инспекторе объектов щелкните на свойстве Name и введите memSample. Как и в случае надписи или поля ввода, убедитесь, что вы изменили свойство области просмотра, а не формы. В заголовке раскрывающегося списка в верхнем части инспектора объектов должно быть написано Memo1: TMemo (после изменения имени области просмотра там будет написано memoSaple: TMemo).
6. Выберите свойство Lines и щелкните на кнопке с тремя точками. В результате раскроется окно редактора строк String List Edutor. Введите какой-нибудь текст. Закончив ввод текста, щелкните на кнопке OK.
7. Выделите форму. Для этого щелкните на ней левой кнопкой мыши или щелкните на имени формы в раскрывающемся списке в верхнем части инспектора объектов. Измените значение свойства Name на frmMemoBoxExample, а свойства Caption- на Пример области просмотра.
8. Нажав клавишу <F9>, запустите программу на выполнение. На экране должно появится изображение. Как видите, в области просмотра отображается не три строки, а четыре.

Заключение – сравнение парадигм программирования

Задача языка высокого уровня - создать представление, удобное для использования программистом. Не существует единственного верного стандартного представления, и это – одна из причин того, что существует так много языков программирования. Объектно-ориентированное программирование на Java дает еще одно представление.

Процедурные языки программирования делает акцент на действия. В этих языках данные существуют для поддержки действий, необходимых программе. Так или иначе, данные «менее интересны», чем действия. Данные в этих языках «негибки». Существует всего несколько встроенных типов данных, и создание программистом своих собственных новых типов данных представляет определенные трудности.

Такой взгляд на вещи изменился с появлением объектно-ориентированного стиля программирования. Объектно-ориентированный стиль программирования повышает значение данных. Основная деятельность при работе с языком такого стиля заключается в создании новых типов данных (т.е. классов) и представлении взаимодействия между объектами этих типов данных.

Для продвижения в этом направлении среда языков программирования нуждается в формализации некоторых соглашений, относящихся к данным. Рассматривается формализация как расширение использования абстрактных типов данных (АТД). АТД уделяют сейчас так же много внимания, как уделяли структурному программированию два десятилетия назад. АТД не заменяют структурное программирование. Скорее, АТД обеспечивают дополнительную формализацию, которая может улучшить процесс разработки программ. Абстрактные типы данных на самом деле охватывают два понятия, а именно, представление данных и операции, которые разрешены над этими данными. В объектно-ориентированном языке (Java) программист для реализации абстрактных типов данных использует классы. Java имеет небольшой набор встроенных типов данных. Абстрактные типы данных расширяют базу языка программирования. Программист имеет возможность создавать новые типы, используя формализм классов. Эти новые типы можно применять так же, как и встроенные типы данных.

Для проведения сравнительного анализа парадигм программирования разработана специальная методика, основанная на таких объектных принципах, как арифметический подсчет элементов текста программы, анализе алгоритмов. Сравнительный анализ делается на основе демонстрационных функционально похожих программ, но реализованных на основе разных парадигм.

Сравнительный анализ установил, что для этих парадигм наблюдается практически полное совпадение:

- Процентного состава описательных операторов;
- Процентного состава количества комментариев;
- Процентного состава описательных операторов процедур;
- Процентного состава операторов кода программы.

Однако при проведении разработки по парадигме объектно-ориентированного программирования по сравнению с парадигмой процедурно-ориентированного программирования объем кода несколько раз увеличивается. Примерно во столько же раз увеличивается трудоемкость разработки. Такое увеличение трудоемкости разработки объясняется платой за организацию самостоятельности поведения объектов и их завершенную функциональность для повторного использования.

Процедурно-ориентированный и объектно-ориентированный подходы к программированию различаются по своей сути и обычно ведут к совершенно разным решениям одной задачи.

Объектно-ориентированная парадигма является наиболее приемлемой для широкого круга задач, связанных с большими промышленными программными системами, в которых основной проблемой является *сложность*.

Объектно-ориентированное программирование развивается уже более двадцати лет. Им несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий, с которыми мы вкратце познакомились.

Как известно, язык Pascal был языком обучения структурному (*процедурно-структурному*) программированию. Поэтому в языке Delphi реализация принципов объектно-ориентированного программирования, кажется, в некоторой мере, искусственным, усложненным. В недавно появившемся языке программирования Java, наоборот, все эти принципы легко воплощены. Поэтому язык Java становится языком обучения объектно-ориентированного программирования. С другой стороны, язык Delphi позволяет на примере Delphi-программы показать реализации всевозможных парадигм программирования, что и было сделано по возможности в этом курсе.

Вопросы для самопроверки

1. Какие бывают компоненты?
2. Что такое палитра компонент?
3. Опишите функциональные возможности инспектора объектов?
4. Назовите основные шаги технологии визуального программирования?
5. Каким образом взаимодействуют между собой объекты в программе?
6. Что такое окно редактора кода?
7. При решении каких проблем лучше использовать объектно-ориентированный подход?
8. Какие характеристики являются фундаментальными в объектно-ориентированном мышлении?
9. Что такое объект? Чем он отличается от других структур данных?
10. В каких случаях применяется объектно-ориентированное программирование?
11. Чем отличаются класс и объект?
12. Чем объект отличается от указателя?
13. Что такое объектная переменная? Какие еще компьютерные переменные вы знаете?

Лабораторное занятие № 1
Тема: *Парадигмы программирования*

План:
Запуск программ, написанных на языке Delphi, по правилам различных парадигм
Выполнение программ при правильных и неправильных данных
Запуск Delphi
Просмотр кода программ

Задача 1.1.
Программы *Lab1a.exe, Lab1b.exe, Lab1c.exe, Lab1d.exe, Lab1e.exe* вычисляют сумму, разницу, умножение и деление двух целых чисел и написаны разным образом, используя следующие парадигмы программирования:
Lab1a.exe – Структурное линейное программирование
Lab1b.exe – Процедурно-ориентированное программирование
Lab1c.exe - Модульное программирование
Lab1d.exe - Объектно-ориентированное программирование
Lab1e.exe – Событийное и визуальное программирование

- ✓ Запустить программы в исполнимом коде и сравнить их. Ввод исходных данных провести правильно (целые числа) и неправильно.

Задача 1.2.
- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файлы с проектом ***File- Open Project*** (**.dpr*) и ***File- Open*** (*.pas)* и в окне редактора кода ознакомиться с исходными кодами программ, сравнить их и запомнить:

Lab1a.dpr
Lab1b.dpr
Lab1c.dpr и *Unit1c.pas*
Lab1d.dpr и *Unit1d.pas*
Lab1e.dpr и *Unit1e.pas*

Лабораторное занятие № 2
Тема: *Язык программирования Delphi. Структура программы.*

План:
Структура программы.
Компиляция и запуск программы на выполнение

Задача 2.1.
1. Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
2. Открыть файл с проектом *Lab2_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой Delphi-программы и дать комментарий к некоторым строчкам, т.е. уточнить текст каждой строки, обозначенной //???????

```
program Lab2_1; //???????
{$APPTYPE CONSOLE}    //dyrektywa kompilatora
                              //- открытие окна DOS-сеанса
uses
  SysUtils;                   //???????
const
```

```
  komunikat1 = 'Moj programma rabotaet na '; //???????
   ckomunikat2 = 5;                          //???????
var komunikat:String; //???????
Begin//???????
  komunikat:='Ispolzuetca  konsolnoe prilojenie';
  Writeln(komunikat);     //???????
  writeln('Privet!');
  writeln;
  write(ckomunikat1);     //???????
  writeln(ckomunikat2);
  write('Najmi klavichu  ENTER');       //???????
  Readln;     //???????
end.  //???????
```

3. Нажав клавишу <F9>, выполнить программу.
4. Нажав клавишу <ENTER>, завершить консольное приложение

Задача 2.2.

- Создать новый проект консольного приложения ***File-New-Other – Console Application***
- В окне редактора кода между begin и end ввести следующий текст:

```
Writeln('Hello, World, from Delphi');
   Writeln('This is a console application');
   Writeln;
   Writeln('Press ENTER to Quit');
   Readln;
```

- Записать код программы на диске ***File-.....***под именем *Lab2_2. dpr*
- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение
- Расширение программы:

Используя константы (const), вывести на экран свои данные: имя, фамилия, отчество, адрес

- Проверить как среда Delphi сообщает ошибки в коде, например, удаляя знак ; в конце строки или меняя имя константы в главном блоке программы

Задача 2.3.

- Используя изученные средства, написать программу выдачи на экране текста структуры программы на языке Delphi и выполнить ее на компьютере

Лабораторное занятие № 3

Тема: *Данные программы (**переменные, константы)** и их типы. **Оператор ввода, вывода***

План:
Именование, объявление и использование переменных
Типы данных.
Стандартные типы данных
Оператор ввода, вывода.

Задача 3.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab3_1.dpr* ***File- Open Project*** (*.*dpr*) и в окне редактора кода ознакомиться со структурой программы в Delphi, данными в виде констант и переменных разных типов

```
program Lab3_1;
{$APPTYPE CONSOLE}
uses
  SysUtils; //deklaracja modulu
const paradygmat='Programmirovanie strukturnoe';
      x=1;
      y=2.5;
      z=True;
var autor:String;
    vozrast:integer;
    pol :char;
    a,b,c:integer;
    d,e,f:real;
begin
  Write('Vvedite autora '); Readln(autor);
  Write('Vvedite vozrast '); Readln(vozrast);
  Write('Vvedite pol (J/M) '); Readln(pol );
  Writeln('autor programu: ', autor, ' vozrast ', vozrast, '  pol  ', pol );
  Write('Vvedite  a= ');Readln(a);
  Write(' Vvedite   b= ');Readln(b);
  Write(' Vvedite   c= ');Readln(c);
  Writeln(a,b,c);
  Writeln(a:10,b:10,c:10);
  Write(' Vvedite   d e f ');Readln(d,e,f);
  Writeln(d,e,f);
  Writeln(d:10:2, e:10:2, f:10:2);
  Readln;                                          //najmi Enter
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 3.2.

- Создать новый проект консольного приложения ***File-New-Other – Console Application***
- Объявить переменные, с именем *целого типа*, выбранного из таблицы 1, оператором readln ввести значения из диапазона значений, и из вне диапазона., а оператором writeln вывести на экран эти значения.

Таблица 1

Имя типа	**Диапазон значений**	**Размер в байтах**
Shortint	От - 128 до 127	1
Smallint	От –32768 до + 32767	2
Interger (или Longint)	От -2 147 483 648 до 2 147 483 647	4
Int64	От -2^{63} до $2^{63}-1$	8
Cardinal (или LongWord)	От 0 до 4 294 967 245	4
Word	От 0 до 65 535	2
Byte	От 0 до 255	1

Объявить переменные, с именем *вещественного типа*, выбранного из таблицы 2 оператором readln ввести значения из диапазона значений, и из вне диапазона., а оператором writeln вывести на экран эти значения.

Таблица 2

Имя типа	*Диапазон значений*	Количество знаков	Размер в байтах
Real48	От $2.9*10^{-39}$ до $1.7*10^{38}$	11-12	6
Single	От $1.5*10^{-45}$ до $3.4x10^{38}$	7-8	4
Double (или Real)	От $5.0\ 10^{-324}$ до $1.7*10^{308}$	15-16	8
Extended	От $3.6*10^{-4951}$ до $1.1*10^{4932}$	19-20	10
Comp	От $-2^{63}+1$ до $2^{63}-1$	19-20	8
Currency	От -922337203685477.5808 до 922337203685477.5807	19-20	8

- Объявить переменные, с именем *символьного типа*, выбранного из таблицы 3 ; оператором readln ввести значения из диапазона значений, и из вне диапазона., а оператором writeln вывести на экран эти значения.

Таблица 3

Имя типа	Размер в байтах
ANSIChar	1
WideChar	2
Char	1

- Объявить переменные, с именем *строкового типа*, выбранного из таблицы 4; оператором readln ввести значения из диапазона значений, и из вне диапазона., а оператором writeln вывести на экран эти значения.

Таблица 4

Тип строки	Область, отводимая для хранения строки	Есть ли нулевой символ в конце
ShortString	От 2 до 256 байт	Нет
AnsiString	От 4 байт до 2 Гбайт	Есть
String	до 256 байт / до 2 Гбайт	Нет / есть
WideString	От 4 байт до 2 Гбайт	есть

Задача 3.3.
Используя изученные средства, написать программу выдачи на экране информации о себе (имя, фамилия, адрес, ...) на основе введенных данных (переменные и константы) разных стандартных типов и выполнить ее на компьютере

Лабораторное занятие № 4
Тема: *Оператор присваивания. Выражения*

План:
Концепция действия. Выражения
Арифметический оператор присваивания

Задача 4.1.
- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***

Открыть файл с проектом *Lab4_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора ознакомиться со структурой программы в Delphi и оператором присваивания

```
program Lab4_1;
{$APPTYPE CONSOLE}
uses
  SysUtils; //deklaracja modulu
var
    a, b, c: real;
    alfa, beta, gamma: real;
    sinus_alfa, cosinus_alfa: real;
begin
  Write('Vvedite dlinu storoni a= ');Readln(a);
  Write(' Vvedite dlinu storoni  b= ');Readln(b);
  Write('Vvedite velichinu ugla gamma mejdu etimi storonami 0<ugol<180)= ');
  Readln(gamma);
  c:=sqrt(sqr(a)+sqr(b)-2*a*b*cos(gamma*Pi/180));
   //systemnai postojnnai  Pi=3.14
  sinus_alfa:=a*sin(gamma*Pi/180)/c;
  cosinus_alfa:=sqrt(1-sqr(sinus_alfa));
  alfa:=arctan(sinus_alfa/cosinus_alfa);
  alfa:=alfa*180/Pi;
  beta:=180-(alfa+gamma);
  Writeln('Dlina storoni c= ',c:0:2);
  Writeln('Velichina ugla alfa= ',alfa:0:2);
  Writeln('Velichina ugla beta= ',beta:0:2);
  Readln;                                          //najmi Enter
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 4.2.

- Открыть файл с проектом *Lab3_2.dpr* (Лабораторные занятия № 3) ***File- Open Project*** (**.dpr*)
- Вывести на печать результаты выполнения операций из таблицы 4.1. на переменных целого типа с использованием оператора присваивания

Таблица 4.1

Знак операции	Содержание операции
+	Сложение
-	Вычитание
*	Умножение
div	Деление и «отсечение» (отбрасывание дробной части)
mod	Взятие остатка при делении

Задача 4.3.

- Создать новый проект консольного приложения ***File-New-Other – Console Application***
- Вычислить значение выражений, используя встроенные математические (таблица 4.2) функции и оператор присваивания:

y=Succ(round(x /2) – pred(x)) для x целого типа

$$y = \frac{1}{\alpha + \beta}\sqrt{\frac{\alpha}{4}}\left(\sqrt{\alpha^2 + \beta^2} - \frac{1}{2}\alpha\beta\right) + \sin\alpha + 2tg\beta$$ для α, β вещественного типа

Таблица 4.2

Функция	**Возвращаемый результат**
Abs(x)	Модуль (абсолютное значение) числа *x*
Ceil(x)	Наименьшее целое, большее или равное *x*
Exp(x)	Вещественное значение числа *e*, возведенное в степень *x*, где *e*-основание натуральных логарифмов
Floor(x)	Наибольшее целое, меньшее или равное числу *x*
Frac(x)	Дробная часть числа *x*, имеющего тип Extended
Int(x)	Целая часть числа *x,* имеющего тип Extended
IntPower(основание, степень)	Значение типа Extended, равное основанию, возведенному в степень
Ldexp(x)	Возвращаемый результат имеет тип Extended и равен $x*2^p$
Ln(x)	Натуральный логарифм вещественного значения *x*
LnXP1(x)	Натуральный логарифм вещественного значения (x+1)
Log10(x)	Десятичный логарифм числа x
Log2(x)	Двоичный логарифм числа x
LogN(n, x)	Логарифм по основанию *n* числа x
Max(x, y)	Большее из двух чисел
Min(x, y)	Меньшее из двух чисел
Pi( )	3,145926535897932385
Power(основание, степень)	Число, равное основание
Round(x)	Вещественное значение *x* округляется до ближайшего целого значения типа Int64. Если *x* находится точно посередине между двумя целыми, то результат всегда является четным числом.
Sgr(x)	Квадрат числа, т.е.*x2*
Sgrt(x)	Квадратный корень числа *x*, т.е. *x*1/2

Trunc(x)	Отбрасывание дробной части числа *x*

Домашнее задание

- Вычислить значение математической функции, выбранной из таблицы 4.2
- Вычислить площадь фигур: квадрата, круга, прямоугольника
- Переменной d присвоить дробную часть положительного числа x
- Вычислить s – сумму порядковых номеров всех букв, входящих в слово SUM.

Лабораторное занятие № 5
Тема: *Оператор присваивания. Выражения*

План:
Логический оператор присваивания
Символьный оператор присваивания
Строковый оператор присваивания
Приведение типов и функции преобразования типов

Задача 5.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab5_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой программы в Delphi и оператором присваивания

```
program Lab5_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var a,b:char;
    c,d:string;
    e:boolean;
begin
  a:='A';
  b:=chr(ord(a)+10);
  writeln (b);
  c:='  ALA   MA   KOTA    ';
  writeln(c);
  d:=LowerCase(Trim(c));
   writeln(d);
   Writeln('dlina c=', length(c));
   Writeln('dlina d=', length(d));
   e:= a<b;
   Writeln(e);
   e:= c=d;
   Writeln(e);
   readln
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 5.2.

- Составить консольное приложение с вводом данных и выводом результата выполнения оператора присваивания с использованием символьных и строковых функций (таблица 5.1). Параметры функции уточнить по Справке Delphi.

Ввести свое имя, фамилию, отчество и вывести переменную, которая складывает строки в одну и печатает их большими буквами.

- Подсчитать сколько букв имеет имя, сколько фамилия, сколько отчество.
- Логической переменной присвоить значение отношения: количество букв имени больше чем фамилии?

Таблица 5.1

Функция	**Назначение**
CompareStr( )	Сравнивает две строки с учетом регистра
CompareText( )	Сравнивает две строки по числовым значениям символов без учета регистра
Concat( )	Выполняет конкатенацию (двух или более) строк в одну
Copy( )	Возвращает подстроку заданной строки
IsDelimiter( )	Определяет, является ли указанный символ строки символом-разделителем
LastDelimiter( )	Возвращает позицию последнего символа-разделителя в строке
Length( )	Возвращает количество символов в строке
LowerCase( )	Возвращает копию строки, заменив все буквы верхнего регистра соответствующими буквами нижнего регистра
Pos( )	Возвращает позицию первого символа указанной подстроки, содержащиеся в заданной строке
QuotedStr( )	Возвращает строку, выделенную апострофами (т.е. добавляет апострофы в конец и в начало строки)
StringOfChar( )	Возвращает строку с указанным количеством повторяющихся символов
StringReplase( )	Возвращает строку с заменой всех вхождений одной подстройки другой подстройкой
Trim( )	Удаляет из строки пробелы, расположенные в её начале и в конце, и управляющие символы
TrimLeft( )	Удаляет из строки пробелы, расположенные в ее начале, и управляющие символы
TrimRinght( )	Удаляет из строки пробелы, расположенные в её конце, и управляющие символы
UpperCase	Возвращает копию строки с заменой всех букв нижнего регистра соответствующими буквами верхнего регистра
WrapText	Разбивает строку на несколько строк, имеющих указанный размер
Delete( )	Удаляет подстроку из строки
Insert( )	Вставляет подстроку в строку, начиная с указанной позиции
SetLength( )	Устанавливает длину строки
SetString( )	Устанавливает содержимое и длину строки
Str( )	Преобразует числовую переменную в строку
Val( )	Преобразует строку в числовую переменную

Задача 5.3.

- Ввести произвольный символ из клавиатуры и вывести его порядковый номер
- Напечатать текст, образованный литерами (символами) с порядковыми номерами 65, 71, 69.
- Записать на Delphi выражение, истинное при выполнении указанного условия и ложное в противном случае:

А) x принадлежить отрезку [2, 5] или [-1, 1];

лежит вне отрезков [2, 5] и [-1, 1]

Лабораторное занятие № 6
Тема: *Условный оператор*

План:
Алгорифм с ветвлениями
Составной оператор begin-end
Условный оператор if-then
Условный оператор if-then-else

Задача 6.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab6_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с условным оператором

```
program Lab6_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var ochki, ocenka: integer;
begin
 write('Vvedite ochki '); readln(ochki);
 if ochki<10 then ocenka:=2;
 if (ochki>=10) and (ochki<15) then ocenka:=3;
 if (ochki>=15) and (ochki<20) then ocenka:=4;
 if ochki>=20 then ocenka:=5;
 writeln('ochki= ', ochki, ' ocenka= ', ocenka);
 readln;
 if ochki<10 then ocenka:=2
   else if ochki<15 then ocenka:=3
        else if ochki<20 then ocenka:=4
             else ocenka:=5;
 writeln('ochki= ', ochki, ' ocenka= ', ocenka);
 readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

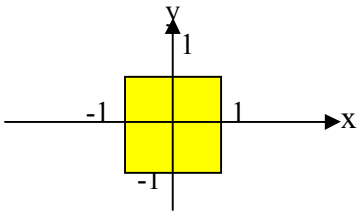
Задача 6.2.

Создать новый проект консольного приложения

- Составить программу следующего вычисления y = max(a, в)
- Составить программу следующего вычисления
- $$y = \begin{cases} 2x, & \text{если } x>0; \\ x+2, & \text{если } x<=0 \end{cases}$$
- Посчитать площадь треугольника на основе его сторон a, b, c. Использовать формулу:

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

- Вводя координаты точки, проверить попадает ли она в указанное пространство:
- Вводя число, определить положительное оно или отрицательное или равно нулю?



Домашнее задание.
Записать последовательность операторов для решения указанной задачи:
по номеру у (y>0) некоторого года определить с – номер его столетия (учесть, что, к примеру, началом XX столетия был 1901, а не 1900 год)

Лабораторное занятие № 7
Тема: *Оператор выбора*

План:
Вложенные условные операторы
Оператор выбора case

Задача 7.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab7_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой программы в Delphi и оператором присваивания

```
program Lab7_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var ochki, ocenka:integer;
begin
  write('Podaj ochki '); readln(ochki);
  if ochki>0 then
  begin
    case ochki of
      0..9: ocenka:=2;
     10..14: ocenka:=3;
     15..19: ocenka:=4;
    else
      ocenka:=5
    end;
    writeln('ochki= ', ochki, ' ocenka= ', ocenka);
 end
 else
   writeln('oschibka dannix');
   readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 7.2.
Составить программу, имитирующую работу микрокалькулятора.
Программа вводит два числа в первой строке и один из знаков +, -, *, / - во второй и выводит на экран результат соответствующего арифметического действия.

```
Label   2, 3;
Var
   Operation : char;    // знак операции
```

```
  , z      : real;    // операнды и результат
  stop       : boolean;  // признак ошибочной операции и останова
Begin
  Stop := false;
  //  пустая строка разделитель
2: writeln;
  write('x, y  ');
  readln(x, y);
  write('операция :   ');
  readln(Operation);
  case  Operation  of
    '+' : z := x + y;
    '-'  : z := x – y;
    '*' : z := x * y;
    '/'  : z := x / y;
  else
    stop := true;
  end;
  if not stop then
    begin
      writeln('     результат =  ', z);
      goto 2;
    end
  else
     goto 3;
3: end.
```

- Оформить консольное приложение и выполнить на компьютере.
- Какова роль переменной stop? Почему в начале этой переменной было присвоено false, а затем присваивается true? Когда останавливается программа?

Задача 7.3.

Создать новый проект консольного приложения

- Составить программу следующего вычисления

$$y = \begin{cases} \sin x, & \text{если } a=1; \\ \cos x, & \text{если } a=2; \\ \operatorname{tg} x, & \text{если } a=3 \\ 0 & \text{в остальных случаях} \end{cases}$$

Лабораторное занятие № 8

Тема: *Цикл. Структура цикла и операторы цикла*

План:

Цикл.
Структура цикла.
Оператор цикла for

Задача 8.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab8_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой цикла и оператором цикла **FOR**

```
program Lab8_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var i:byte;
```

```
    x,s:real;
begin
s:=0;
FOR i:=1 TO 10 DO
      BEGIN
        Write ('x',i,'=');
        Readln(x);
        s:=s+x;
       END;
writeln ('Summa=', s:5:2);
readln;
s:=0;
FOR i:=10 DOWNTO 1 DO
      BEGIN
        Write ('x',i,'=');
        Readln(x);
        s:=s+x;
       END;
writeln ('Summa=', s:5:2);
readln;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 8.2.

Составить консольное приложение для решения задачи:

Студент сдает N экзаменов. Вычислить среднюю его оценок и определить является ли он отличником (средняя выше 4.5), используя оператор FOR

Задача 8.3.

Составить консольное приложение для решения задачи:

Вывести все порядковые номера букв алфавита от A до Z, используя оператор FOR

Домашнее задание.

Составить консольное приложение для решения задачи:

Если среди чисел $\sin x^n$ (n = 1, 2, …, 30) есть хотя бы одно отрицательное число, то логической переменной t присвоить значение true , а иначе – значение false.

Лабораторное занятие № 9

Тема: *Цикл. Структура цикла и операторы цикла*

План:

Оператор цикла while

Оператор цикла repeat

Задача 9.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab9_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой цикла и оператором цикла ***while*** и ***repeat***

```
program Lab9_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var licz:byte;
    x,s:real;
begin
s:=0; licz:=0;
WHILE licz<10 DO BEGIN
```

```
            Write ('x',licz+1,'=');
            Readln(x);
            s:=s+x;
            licz:=licz+1;    //inc(licz)
                END;
    writeln ('Summa=', s:5:2);
    readln;
    s:=0; licz:=0;
    REPEAT
        Write ('x',licz+1,'=');
        Readln(x);
        s:=s+x;
        licz:=licz+1;  //inc(licz)
    UNTIL licz=10;
    writeln ('Summa=', s:5:2);
    readln;
    end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 9.2.

Машинный нуль. Это программа, которая отыскивает так называемое «машинное эпсилон» - такое минимальное, не равное нулю вещественное число, которое после прибавления его к 1,0 еще дает результат, отличный от 1,0.

- ✓ Оформить текст в виде консольного приложения и выполнить:

```
.........
var
  Epsilon:   Real;
begin
  Epsilon := 1;
while  1+Epsilon/2>1 do
  Epsilon := Epsilon/2;
  Writeln('Машинное эпсилон =' , Epsilon)
end;
.....
```

У читателя, привыкшего к непрерывной вещественной арифметике, может вызвать недоумение утверждение о том, что в *дискретной машинной арифметике* всегда существуют такие числа 0<X<epsilon, что 1.0 + X = 1.0. Дело в том, что внутреннее представление типа *Real* может дать лишь конечное множество чисел. Аппроксимация бесконечного непрерывного множества вещественных чисел конечным (пусть даже и очень большим) множеством их внутреннего машинного представления и приводит к появлению «машинного эпсилона». Машинный нуль – это последовательность знаков, воспринимаемая машиной как нуль.

Задача 9.3.

Составить консольное приложение для решения задачи:

Для последовательности целых чисел (отличных от нуля) законченых нулем $a_1, a_2, a_3, ...0$

вычислить среднюю арифметическую четных и среднюю арифметическую нечетных чисел в этой последовательности

Лабораторное занятие № 10
Тема: Цикл. Структура цикла и операторы цикла

План:

Управление работой циклов
Вложенные циклы

Задача 10.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab10_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться со структурой вложенного цикла – программа печатает треугольник, построенный из выбранного знака

```
program Lab10_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var x:char;
    i,j,n:byte;
begin
  write('Vvedite znak postroenia ');readln(x);
   write('Vvedite  n ');readln(n);
   for i:=1 to n do
       begin
       for j:=1 to i do write(x);
       writeln;
       end;
  readln;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 10.2.

Изменить алгоритм задачи 10.1 так, чтобы:

- ✓ выбранным знаком печатать квадрат или прямоугольник в указанных размерах
- ✓ после вывода знаков пользователь решал повторить эти действия еще раз или нет (использовать оператор REPEAT)

Задача 10.3.

Составить консольное приложение для решения задачи:

Вычислить сумму N начальных членов последовательности 1 + 2/3 + 3/5 + 4/7 + ...

Домашнее задание.

- ✓ Вычислить s – сумму квадратов всех целых чисел, попадающих в интервал (ln x, e^x), x>1.
- ✓ Определить k – количество трехзначных натуральных чисел, сумма цифр которых равна n (1<=n<=27). Операции деления (/, div, mod) не использовать

Лабораторное занятие № 11

Тема: *Парадигма процедурно-структурного программированияПодпрограммы: Процедуры*

План:

Подпрограмма
Объявление и вызов процедуы
Параметры подпрограмм
Глобальные и локальные переменные
Завершение работы подпрограммы

Задача 11.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab11_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с подпрограммами, объявлением и вызовом процедуры. Каждую строку обозначенную //??????? уточнить как текст комментария.

```
program Lab11_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
```

```
var a,b:Integer; //?????
Procedure linia; //?????
var i:integer;   //?????
begin
for i:=1 to 80 do write('x');
writeln;
end;
Procedure dane;
begin
Write('Vvedite a '); Readln(a);
Write('Vvedite b '); Readln(b);
end;
Procedure kalkulator(a,b:integer); //?????
begin
WRITELN('summa=', a+b);
WRITELN('roznica=', a-b);
WRITELN('proizvedenie=', a*b);
if b<>0 then WRITELN('chastnoe=', a/b:5:2)
        else Writeln('Oschibka delenia na 0');
end;
BEGIN
linia; //?????
dane;
linia;
kalkulator(a,b); //?????
linia;
READLN;
END.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 11.2.

Составить консольное приложение для решения задачи:

- ✓ Вычислить зарплату рабочего, который проработал N часов по К долларов за час – алгоритм оформить как процедуру
- ✓ Используя процедуру вычислить зарплату для 3 рабочих

Задача 11.3.

Составить консольное приложение для сортировки 3 чисел. Программа содержит процедуру с параметрами

- ✓ Ввести код программы и исполнить ее
- ✓ Убрать слово **var** из параметров процедуры, передавая параметры по значению – сравнить действия программы

```
program Lab11_3;
{$APPTYPE CONSOLE}
uses
  SysUtils;
var a, b, c : integer;
procedure swap(var xx, yy : integer);
var t : integer;
begin
t:=xx; xx:=yy; yy:=t
end;
begin
write('vvedite 3 chisla ');
readln(a, b, c) ;
if a>b then swap(a, b);
if b>c then swap(b, c);
```

```
if b then swap(a, b);
writeln('Rezultat sortirovki: ', a, ' ', b, ' ', c);
readln;
end.
```

Домашнее задание. Составить два варианта программы вычисления

У = (max(a, c+d) + min(c, a * d)) / (2 + max(a, d, c)),

используя подпрограмм (стандартных и пользовательских) нахождения максимального и минимального из двух чисел для заданных a, c, d..

Лабораторное занятие № 12

Тема: ***Парадигма процедурно-структурного программирования Подпрограммы: Функции***

План:

Объявление и вызов функций

Значение, возвращаемое функцией

Задача 12.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab1b.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с объявлением и вызовом функций

```
program Lab1b;
{$APPTYPE CONSOLE}
uses
  SysUtils;
Function summa (a,b:Integer):Integer;
Begin
        Result:=a+b;
End;
Function roznica (a,b:Integer):Integer;
Begin
        Result:=a-b;
End;
Function umnoj (a,b:Integer):Integer;
Begin
        Result:=a*b;
End;
Function deli (a,b:Integer):Real;
Begin
        Result:=a/b;
End;
VAR x,y:Integer;
begin
Write('vvedite  x '); Readln(x);
Write('vvedite y '); Readln(y);
WRITELN('summa=', summa(x,y));
WRITELN('roznica=', roznica(x,y));
WRITELN('proizvedenie=', umnoj(x,y));
if y<>0 then WRITELN('chastnoe=', deli (x,y):5:2)
 else Writeln('Oschibka delenia na 0');
READLN;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 12.2.

Составить консольное приложение для вычиления функции:

a) $\frac{a^2b+b^2a-a}{|a|+|b|+7}-\frac{a-b}{ab}$

b) $\frac{a\sin\pi b}{bc}-\frac{ab}{2\cos^2 c^2}$

Задача 12.3. Составить консольное приложение, содержащие функции для вычисления емкости фигур: шар, кубик, параллелепипеда

Домашнее задание var k : integer; a,b : real;

Описать функцию Stepen(x,n) от вещественного x и натурального n, вычисляющую (через умножение) величину x^n, и использовать ее для вычисления $b = 2.7^k + (a+1)^{-k}$

Лабораторное занятие № 13

Тема: ***Парадигма процедурно-структурного программирования Подпрограммы: Процедуры и функции***

План:
Перегрузка функций
Рекурсия
Встраиваемые подпрограммы

Задача 13.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab13_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с перегрузкой функций

```
program Lab13_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
procedure Test(n: Integer); overload;
begin
    Writeln('integer');
end;
procedure Test(x: Real); overload;
begin
   Writeln('real',x);
end;
procedure Test(s: String);overload;
begin
   Writeln('string',s);
end;
procedure Test(s: String; k:byte);overload;
begin
   Writeln('string byte ', s,k);
end;
Begin
  Test(1);
  Test(1.5);
  Test('ala');
  Test('agent ', 007);
  readln;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задание.

Задача 13.2. Оформить консольное приложение для использования следующих функций:

```
Function Divide(X, Y: Real): Real;
Begin
  Result := X/Y;
End;
Function Divide(X, Y: Integer): Integer;
Begin
  Result := X div Y;
End;
```

как перегруженных функций.

Домашнее задание. Составить консольное приложение, реализующее пример пере грузки операции «+» для числовых и строковых операндов.

Лабораторные занятия № 14
Тема: *Пользовательские типы*

План:
Пользовательские типы
Перечислимые типы
Ограниченные типы (тип диапазона)

Задача 14.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab14_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с пользовательскими типами, а также каждую строку обозначенную //??????? уточнить как текст комментария.

```
program Lab14_1;
{$APPTYPE CONSOLE}
{$R+}
uses
  SysUtils;
type tt=(ala,ola,ela);//?????
var i:tt; //?????
procedure data;
var d:1..31;
   m:1..12;
   y:2000..2010;
begin
write ('den:'); readln(d);
write ('miesiac:'); readln(m);
write ('god:'); readln(y);
writeln(' data ', y,'-',m,'-',d);
end;
begin
for i:=ala to ela do
begin
writeln(ord(i));
data;
end;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 14.2.

оставить программу для вычисления среднего балла, полученного абитуриентом на экзам по математики, физики, химии

- ✓ использовать тип – диапазон для контроля вводимых оценок.
- ✓ добавить код в место, обозначенной как //.....

```
Program Lab14_2;
{$APPTYPE CONSOLE}
{$R+} //Включения директивы проверки диапазона при присваивании
      // значений переменной типа-диапазона.
uses SysUtils;
type
  Tball = 2..5;
var
  math,phiz,chem: Tball;
  ave: real;
begin
  writeln('Enter math, phiz, chem');
  readln(math, phiz, chem);
  //.....
  readln
end.
```

:

Задача 14.3.

Составить консольное приложение

Пусть

```
Type strana = (Австрия, Гана, Италия, Перу, Сша, Швеция);
    Kontinent = (Америка, Африка, Европа);
Var x, y : strana;
   t : boolean;
```

- ✓ Описать функцию kont(s), определяющую континент, на котором находится страна s, и использовать ее для изменения значения переменной t на противоположное, если страны x и y находятся на разных континентах

Лабораторное занятие № 16
Тема: *Статические массивы*

План:

Статические массивы
Операции, допустимые над массивами

Задача 16.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab16_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с использованием статических массивов - программа считает среднюю и самую лучшую оценку для группы n=5 студентов

```
program Lab16_1;
{$APPTYPE CONSOLE}
const n=5;
type dane=array[1..n]of real;
var oceny:dane;
procedure chitaj;
var i:integer;
begin
for i:=1 to n do
  begin
  Write('ocenka ', i,' = ');
```

```
  readln(oceny[i]);
 end;
end;
function srednia:real;
var i:integer;
 s:real;
begin
s:=0;
for i:=1 to n do
  s:=s+oceny[i];
srednia:=s/n;
end;
function max:real;
var i:integer;
    m:real;
begin
m:=oceny[1];
for i:=2 to n do
  if oceny[i]>m then m:=oceny[i];
max:=m;
end;
begin
  writeln('Vvedite ocenky studenta');
  chitaj;
  writeln('Srednia studenta=',srednia:2:2);
  writeln('Luchaj ocenka = ', max:2:2);
  readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 16.2.

Расширить программу Lab16_1 о :

- поиск самой плохой оценки
- вывод оценки каждого студента с комментарием: "Выше среднего", "Средне" или "Ниже среднего" - сравнение оценки студента со средней оценкой

Задача 16.3.

Составить программу для работы на двухмерном статическом массиве MS:
var MS: array[1..5, 1..5] of integer;

- элементам массива присвоить случайное целое число из диапазона 1-99 (использовать процедуру *Randomize* и функцию *Random()*)
- распечатать массив
- посчитать сумму всех элементов и сумму элементов на диагонали
- посчитать сколько раз в элементе массива попалось число 1

Домашнее задание

Составить программу для вычисления средней зарплаты учителя иностранного языка, работающего по часам, за 12 месяцев (зарплата в каждом месяце разная).

Лабораторные занятия № 17
Тема: *Динамические массивы*

План:

Динамические массивы
Операции, допустимые над массивами
Передача массивов подпрограммам

Задача 14.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab17_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с с использованием динамических массивов - программа считает среднюю и самую лучшую оценку для группы n студентов

```
program Lab17_1;
{$APPTYPE CONSOLE}
type dane=array of real;
var oceny:dane;
    n:integer;
procedure chitaj;
  var i:integer;
begin
  write('Vvedite n '); readln(n);
  SetLength(oceny, n);
  for i:=0 to n-1 do
  begin
     Write('ocenka ', i+1,' = ');
     Readln(oceny[i]);
  end;
end;
function srednia:real;
var i:integer;
 s:real;
begin
  s:=0;
  for i:=0 to n-1 do
    s:=s+oceny[i];
  srednia:=s/n;
end;
function max:real;
var i:integer;
   m:real;
begin
  m:=oceny[0];
  for i:=1 to n-1 do
    if oceny[i]>m then m:=oceny[i];
  max:=m;
end;
begin
  writeln('Vvedite ocenky studenta');
  chitaj;
  writeln('Srednia studenta=',srednia:2:2);
  writeln('Luchaj ocenka =', max:2:2);
  readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 17.2.

- Составить программу для вычисления стоимости корзины с покупками какого- то количества разных товаров

Задача 17.3.

Составить программу для работы на двухмерном динамическом массиве MD:
var MD: array of array of integer;

- ввести данные в элементы массива

распечатать массив

- заменить все отрицательные числа на нули и подсчитать сколько этих чисел

Лабораторное занятие № 18
Тема: *Записи.*

План:
Определение записи
Операции, допустимые над записями

Задача 18.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab18_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с с использованием записи

```
program Lab18_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  TStudentInfo = record
                    lastName:   String[25];
                    firstName:  String[15];
                   age: Integer;
                  end;
var studentInfo: TStudentInfo;
begin
  write('Vvedite lastName= ');readln(studentInfo.lastName);
  write(' Vvedite  lfirstName= ');readln(studentInfo.firstName);
  write(' Vvedite  age= ');readln(studentInfo.age);
   with studentInfo do
     begin
       writeln('Hello ', lastName, '  ',firstName);
       if  age>=18 then writeln('Vzroslyj' )
              else writeln('Malolet');
     end;
  readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 18.2.

- Составить программу для вычисления стоимости корзины с покупками какого- то количества разных товаров. Для каждого товара определить название и цену.
- Напечатать название самого дорогого товара

Задача 18.3.

- Составить программу создания плана работ на неделю. План должен содержать день, название работы, место, стоимость.
- Подсчитать стоимость всех запланированных работ

Домашнее задание

- Описать тип записи для представления следующего понятия:
  А) время в часах, минутах и секундах;
  Б) экзаменационная ведомость (предмет, номер группы, дата экзамена, 25 строчек с полями: фамилия студента, номер его зачетной книжки, дата экзамена)

Лабораторное занятие № 19
Тема: *Множества.*

План:
Определение множества
Операции, допустимые над множествами

Задача 19.1.

- Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- Открыть файл с проектом *Lab19_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с использованиями записи и множества.

```
program Lab19_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
const
  pol = ['j','J','m','M'];
var
  vdane : record
            family : string[30];
             imie     : string[20];
             pol      : char;
             vozrast      : byte;
          end;
  koniec : boolean;
  NN:set of byte;
  w:byte;
procedure vvedennye_dannie;
begin
  writeln;
  write('Vvedite family: '); readln(vdane.family);
  if vdane.family='' then koniec:=true;
  if not koniec then
  begin
    write('Podaj imie: '); readln(vdane.imie);
    repeat
      write('Podaj plec: '); readln(vdane.pol);
    until vdane.pol in pol;
    write('Podaj vozrast: '); readln(vdane.vozrast);
    Include(NN, vdane.vozrast);
  end;
end;
procedure prezentacjanye_dannie;
begin
  if not koniec then
  begin
    with vdane do
    begin
      writeln;
      writeln('Family: ',family);
      writeln('Imie: ',imie);
      writeln('Pol: ',pol);
      writeln('Vozrast: ',vozrast);
    end
  end;
end;
begin
  koniec:=false;
```

```
  repeat
    vvedennye_dannie;
    prezentacjanye_dannie;
  until koniec;
  write('Podaj vozrast dlj proverki ');readln(w);
  if w in NN then writeln('Ect chelovek takogo vozrasta')
             else writeln('Net cheloveka takogo vozrasta');
  write('Najmi klawisy ENTER'); readln;
end.
```

- Нажав клавишу <F9>, выполнить программу.
- Нажав клавишу <ENTER>, завершить консольное приложение

Задача 19.2.

- Составить программу для вычисления стоимости корзины с покупками какого- то количества разных товаров. Для каждого товара определить название и цену.
- Напечатать название самого дорогого товара

Задача 19.3.

Составить программу, подсчитывающую общее количество букв A или a входящих в строку S.

Лабораторное занятие № 20
Тема: *Файлы текстовые*

План:

Файлы текстовые
Функции и процедуры для обработки текстовых файлов

Задача 20.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab20_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с использованием текстовых файлов

```
program Lab20_1;
{$APPTYPE CONSOLE}
uses  SysUtils;
const chet= 'liczby1.txt';
var plik:textfile;
    l:byte;
Procedure zapis;
var i:byte;
begin
  assignfile(plik, chet );
  {$I-} Append(plik); {$I+}
  if IOResult<>0 then Rewrite(plik);
  Randomize;
  For i:=1 to 50 do
    begin l:=Random(99);
          Write(plik, l:3);
    end;
  closefile(plik);
end;
Procedure;
var k:integer;
begin
  assignfile(plik, nazwa);
  {$I-} Reset(plik); {$I+}
```

```
if IOResult=0 then
  begin
     k:=0;
     while not eof(plik) do
       begin read(plik,l);
        write(l:3);
        k:=k+1;
       end;
  writeln;
  closefile(plik);
  writeln('Elementow: ',k);
  end;
end;
begin
  zapis;
  writeln('soderjimie plik:');
  chtenie;
  readln;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 20.2.

- ✓ Добавить к программе *Lab20_1.dpr* функцию, которая вычислить сумму всех чисел содержимых и файле
- ✓ Переделать процедуры записи и чтения так, чтобы числа записывались каждый в отдельной строке

Задача 20.3.

Составить программу, которая

- ✓ в текстовом файле запишет фамилии студентов, которые приступили к экзамену, а в конце напечатает все фамилии
- ✓ проверить есть ли в списке данная фамилия

Лабораторное занятие № 21
Тема: *Файлы типизированные*

План:

Бинарные файлы
Файлы типизированные
Функции и процедуры для обработки типизированных файлов

Задача 21.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab21_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с использованием типизированных файлов

```
program Lab21_1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type adres = record
                family:string[25];
                imie: string[15];
                ulica:string[30];
                kod:string[6];
                gorod:string[20]
```

```
                end;
        plikrek=file of adres;
var plik1:plikrek;
    dane:adres;
    nazwa:string[12];
procedure podajnazwe(var plik:plikrek);
 begin
      write('Podaj nazwe dyskowegoimie ');
      readln(nazwa);
      if nazwa='' then assignfile(plik, 'adresy.tmp')
                  else assignfile(plik,nazwa);
 end;
procedure pisz(var plik:plikrek);
var MaxNumerRekordu:Integer;
    koniec:boolean;
begin
    {$I-}
    Reset(plik);
    {$I+}
    if IOResult<>0 then
                rewrite(plik);
    MaxNumerRekordu:=Filesize(plik);
    seek(plik, MaxNumerRekordu);
    koniec:=False;
    while not koniec do
    begin
    writeln('Vvedite family, esli koniec to * ');
    readln(dane.family);
if dane. family <>'*' then
       begin
    writeln(' Vvedite imie');
    readln(dane.imie);
    writeln(' Vvedite ulice');
    readln(dane.ulica);
    writeln(' Vvedite kod');
    readln(dane.kod);
    writeln(' Vvedite gorod');
    readln(dane.gorod);
    write(plik,dane);
      end              else
      koniec:=True;
    end;
    closefile(plik)
end;
procedure czytaj(var plik:plikrek);
begin
     reset(plik);
     writeln('Napechataj coderjimoe fajl', nazwa);
     while not Eof(plik) do
          begin
             read(plik,dane);
             with dane do
             writeln(family+' '+imie+ ' '+ulica+' '+kod +' '+gorod)
           end;
      closefile(plik)
 end;
BEGIN
    podajnazwe(plik1);
```

```
    pisz(plik1);
    czytaj(plik1);
    readln;
END.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 21.2.

Составить программу, которая:

- ✓ Создасть телефонную книгу (имя, фамилия, телефон)
- ✓ Напечатает все телефоны
- ✓ Найдет телефон персоны о данной фамилии

Задача 21.3.

Составить программу для хранения в типизированном файле вещественных чисел – внешнюю температуру. Программа должна добавлятm числа и считать количество отрицательных температур

Лабораторное занятие № 22
Тема: *Файлы.*

План:

Бинарные файлы
Файлы типизированные
Файлы нетипизированные

Задача 22.1.

✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***

✓ Открыть файл с проектом *Lab22_1.dpr* ***File- Open Project*** (**.dpr*) и в окне редактора кода ознакомиться с использованием нетипизированных файлов

```
program Lab22_1;
{$APPTYPE CONSOLE}
var nachal_mnojestvo, con_mnojestvo:file;
      nazwa_nachal, nazwa_con:string[63];
      bufor:array[0..127] of word;
      wynik:integer;
begin
   Write('Podaj nazwe zbioru nachal. mnojestvo  ');
   Readln(nazwa_ nachal);
   Assignfile(nachal_mnojestvo, nazwa_nachal );
   Reset(nachal_mnojestvo);
   Write('Podaj nazwe con. mnojestvo ');
   Readln(nazwa_con );
   Assignfile(con_mnojestvo, nazwa_con );
   Rewrite(con_mnojestvo );
   Repeat
      BlockRead(nachal_mnojestvo, bufor,1, wynik);
      BlockWrite(con_mnojestvo, bufor, wynik);
    until wynik=0;
   Closefile(nachal_mnojestvo);
   Closefile(con_mnojestvo);
   Readln;
end.
```

✓ Нажав клавишу <F9>, выполнить программу.

✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 22.2.

Составить программу для переписи данных (оценки) с одного файла в другой с удале двоек – использовать типизированный файл с целыми числами (2, 3, 4, 5)

Задача 22.3.

Составить программу, которая удаляет выбранный элемент из внутрии файла созданного программой 22.2

Лабораторное занятие № 23

Тема: ***Парадигма модульного программирования. Модули***

План:

Понятие модуля.
Запись модуля
Главная программа модульной Delphi-программы
Парадигма модульного программирования в среде Delphi

Задача 23.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab1c.dpr* ***File- Open Project*** (**.dpr*) и файл с модулем *Unit1c.pas* (View – Units) в окне редактора кода ознакомиться с использованием модулей

Unit1c.pas – модуль

```
unit Unit1c;
interface
 Function summa (a,b:Integer):Integer;
 Function roznica (a,b:Integer):Integer;
 Function umnoj (a,b:Integer):Integer;
 Function deli(a,b:Integer):Real;
implementation
 Function summa (a,b:Integer):Integer;
Begin
        Result:=a+b;
End;
Function roznica (a,b:Integer):Integer;
Begin
        Result:=a-b;
End;
Function umnoj (a,b:Integer):Integer;
Begin
        Result:=a*b;
End;
Function deli(a,b:Integer):Real;
Begin
        Result:=a/b;
End;
end.
```

Lab1c.dpr – главная программа

```
program Lab1c;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Unit1c in 'Unit1c.pas';
VAR x,y:Integer;
Begin
  Write('Podaj x '); Readln(x);
  Write('Podaj y '); Readln(y);
```

```
  WRITELN('suma=', summa(x,y));
  WRITELN('roznica=', roznica(x,y));
  WRITELN('proizvedenie=', umnoj(x,y));
  if y<>0 then WRITELN('chastnoe=', deli(x,y):5:2)
   else Writeln(Ochibka delenie na 0');
READLN;
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

***Задача* 23.2.**

Создать модуль с функцией вычисления синуса двойного угла

- ✓ Создать новый модуль командой ***File-New- Other*** (Файл> Создать > Другое). Выберем значок ***Unit*** в категории ***New Fil***es (Новые файлы).
- ✓ Командой ***File-Save As*** файл с новым модулем назовем *MyMath.pas* – автоматически меняется идентификатор модуля ***unit*** *MyMath*
- ✓ В интерфейсном разделе нового модуля введем заголовок функции *Sin2*.
- ✓ В разделе реализации введем полное описание этой функции. Общий текст модуля станет следующим:

```
 unit MyMath;
interface
// заголовок функции
function  Sin2(a: Real ): Real;
implementation
// реализация функции
function Sin2(a: Real ): Real;
begin
  Result := Sin(2 * a);
end;
end.
```

Вызов модуля в главной программе:

- ✓ Создаем новое консольное приложение ***File-New- Other –Console Applicatin***
- ✓ Добавляем модуль к проекту ***Project – Add to Project*** выбираем *MyMath.pas*- автоматически добавляется в программе **uses** MyMath in ' MyMath.pas';
- ✓ Проводим вычисления: ввод угла, вывод синуса двойного угла

Задача 23.3.

Создать модуль, содержащий функции вычисления поля фигур: круга, квадрата, прямоугольника, треугольника,. Составить программу для вычисления поля поверхности фигуры сложенной из какого-то количества этих фигур с использованием этого модуля.

Домашнее задание.

Оформить алгоритм нахождения максимального из трех чисел, как модуль и использовать ее в консольном приложении.

Лабораторное занятие № 25

Тема: *Парадигма объектного программирования. Классы и объекты.*

План:

Понятие класса и объекта

Парадигма объектно-ориентированного программирования

Задача 25.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***

Открыть файл с проектом *Lab1d.dpr* ***File- Open Project*** (**.dpr*) и файл с модулем *Unit1* (View – Units) в окне редактора кода ознакомиться с использованием классов и объектов – объяснить значение строк обозначенных //???????

Unit1d.pas – модуль

```
unit Unit1d;
interface
 Type Tdzialania=class //???????
       Private          //???????
         a,b:Integer;   //???????
       Public           //???????
         Procedure dane;//???????
         Function suma:Integer; //???????
         Function roznica:Integer;
         Function umnoj:Integer;
         Function deli:Real;
   end;
implementation
Procedure Tdzialania. dane;   //???????
begin
  write('a= '); readln(a);
  write('b= '); readln(b);
 end;
Function Tdzialania.summa :Integer;
Begin
        Result:=a+b;
End;
Function Tdzialania.roznica :Integer;
Begin
        Result:=a-b;
End;
Function Tdzialania.umnoj :Integer;
Begin
        Result:=a*b;
End;
Function Tdzialania.deli :Real;
Begin
        if b<>0 then Result:=a/b
             else begin Result:=0; writeln ('Oschibka delenia na 0');end;
End;
end.
```

Lab1d.dpr – главная программа

```
program Lab1d;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  Unit1d in 'Unit1d.pas';
Var dzialania : Tdzialania;       //???????
BEGIN
  dzialania := Tdzialania.Create;         //???????
  dzialania . dane;                       //???????
  writeln('suma= ',  dzialania.suma);   //???????
  writeln('roznica=',  dzialania. roznica);
```

```
    ein('proizvedenie= ',  dzialania. umnoj);
  writeln('chastnoe=',  dzialania. deli);
  dzialania.Destroy;     //???????
  READLN;
END.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение
- ✓ Возможна также версия программы без модуля, как Lab1d2 (тогда принципы инкапсуляции не работают):

```
program Lab1d2;
{$APPTYPE CONSOLE}
Uses
  SysUtils;
Type Tdzialania=class  //???????
       Private                    //???????
         a,b:Integer;             //???????
       Public                     //???????
         Procedure dane;          //???????
         Function summa:Integer;        //???????
         Function roznica:Integer;
         Function umnoj:Integer;
         Function deli:Real;
   end;
Procedure Tdzialania. dane;   //???????
Begin
  write('a= '); readln(a);
  write('b= '); readln(b);
end;
Function Tdzialania.summa :Integer;
Begin
        Result:=a+b;
End;
Function Tdzialania.roznica :Integer;
Begin
        Result:=a-b;
End;
Function Tdzialania.umnoj:Integer;
Begin
        Result:=a*b;
End;
Function Tdzialania.deli :Real;
Begin
        if b<>0 then Result:=a/b
             else begin Result:=0; writeln (Ochibka delenie na 0');end;
End;
Var dzialania : Tdzialania;       //???????
BEGIN
  dzialania := Tdzialania.Create;         //???????
  dzialania . dane;                       //???????
  writeln('summa= ',  dzialania.summa);        //???????
  writeln('roznica=',  dzialania.roznica);
  writeln('proizvedenie= ',  dzialania.umnoj);
  writeln('chastnoe=',  dzialania.deli);
  dzialania.Destroy;     //???????
  READLN;
END.
```

- ○ Набрать текст программы и сохранить, как консольное приложение Lab1d2.

Задача 25.2.

Составить программу для вычисления поля поверхности фигур: круга, квадрата.

- ✓ Создать новое консольное приложение ***File-New-Other-Console Application*** и записать под именем *Lab23_2.dpr*
- ✓ Создать модуль ***File-New-Unit*** и записать под именем *Unit23_2.pas*
- ✓ В части ***interface*** модуля описать классы Tkolo, Tkwadrat для вычисления площади фигур: круга, квадрата

```
TYPE Tkolo= class
              a:Real;
              PROCEDURE SetA;
              FUNCTION pole:Real;
              FUNCTION GetA:Real;
            END;
    Tkwadrat=.....        // дописать

    ..............
```

- ✓ В части ***implementation*** модуля описать методы каждого класса

```
  PROCEDURE Tkolo . SetA;
Begin
  Write('a='); readln(a);
End;
FUNCTION Tkolo . pole:Real;
Begin
   Result:=pi*sqr(a);
End;
FUNCTION Tkolo . GetA:Real;

egin
  Result:=a;
End;
   ............
```

- ✓ Составить код главной программу содержащей 2 объекта

```
Var kolo:Tkolo;//1
        .........          //2
```

- ✓ Создать объекты, провести на них действия (использовать методы) и уничтожить объекты

```
BEGIN
  kolo := Tkolo.Create;
  kolo . GetA;
  ...........
  kolo.Destroy;
  READLN;
END.
```

Задача 25.3.

Составить объектную программу для вычисления средней оценок студента определенной имени и фамилии, сдающего в сессии N экзаменов

Лабораторное занятие № 26

Тема: *Парадигма объектного программирования. Классы и объекты.*

План:

Понятие класса и объекта
Инкапсуляция.
Парадигма объектно-ориентированного программирования

Задача 26.1.

✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***

✓ Открыть файл с проектом *Lab1d.dpr* ***File- Open Project*** (**.dpr*) и файл с модулем *Unit1d.pas* (View – Units)

✓ В тело главной программы добавьте инструкции: **writeln(dzialania.a, dzialania.b);** и `dzialania.Destroy;` в следующем виде:

```
……:
BEGIN
  dzialania := Tdzialania.Create;        //???????
  dzialania . dane;                      //???????
  writeln('summa= ',  dzialania.summa);       //???????
  writeln('roznica=',  dzialania. roznica);
  writeln('proizvedenie= ',  dzialania. umnoj);
  writeln('chastnoe=',  dzialania.deli);
  writeln(dzialania.a,dzialania.b);
  dzialania.Destroy;     //???????
  READLN;
END.
```

- Затем выполните. Попытка вывести значений полей объекта вызовет ошибку (результат ошибки это действие директивы ***Private*** внутри класса – результат *инкапсуляции)*
- Открыть файл с проектом *Lab1d2.dpr* и ввести аналогическую инструкцию writeln(dzialania.a, dzialania.b) в текст кода; Выполнив, вы ошибку не обнаружите, ибо *инкапсуляция* не действует, так как класс не в отдельном модуле

Задача 26.2.

Для задач 25.2 и 25.3 применить свойство инкапсуляции

Задача 26.3.

Создать объективно-ориентированную программу для вычисления зарплаты 2-х работников. Каждый из них отработал по сколько-то часов, каждому за час своя зарплата. Вывести фамилию работника, у которого большая зарплата

✓ Создать консольное приложение с модулем содержащим класс TRobotnik

Лабораторное занятие № 27

Тема: *Парадигма объектного программирования. Классы и объекты.*

План:

Понятие класса и объекта
Инкапсуляция, наследование, полиморфизм.
Парадигма объектно-ориентированного программирования

Задача 27.1.

✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***

✓ Открыть файл с проектом *Lab27_1.dpr* ***File- Open Project*** (**.dpr*) и файл с модулемями *Unit27_1a.pas*, *Unit127_1b* (View – Units) в окне редактора кода ознакомиться со свойствами *наследования, полиморфизма.*

Unit27_1a.pas – модуль с классом предка

```
unit Unit27_1a;
interface
 Type Tdzialania=class
      Protected                              //инкапсуляция
```

```
        a,b:Integer;
         Function summa:Integer;
         Function roznica:Integer;
         Function umnoj:Integer;
         Function deli:Real;virtual;                    //полиморфизм
       Public                                                  //инкапсуляция
         Procedure dane; overload;                        //перегруженный метод
         Procedure dane(x,y:integer);overload;     //перегрузка метода
         Procedure rezultaty;
      end;

implementation
Procedure Tdzialania. dane;
Begin
  write('a= '); readln(a);
  write('b= '); readln(b);
end;
 Function Tdzialania.summa :Integer;
Begin
        Result:=a+b;
End;
Function Tdzialania.roznica :Integer;
Begin
        Result:=a-b;
End;
Function Tdzialania.umnoj :Integer;
Begin
        Result:=a*b;
End;
Function Tdzialania.deli:Real;
Begin
     writeln('klasa Tdzialania');
     Result:=a/b;
End;
procedure Tdzialania.dane(x, y: integer);
begin
  a:=x;
  b:=y;
end;
procedure Tdzialania.rezultaty;
begin
  writeln('Rezultaty obliczen dla a=', a ,' b=',b);
  writeln('suma= ',  summa);
  writeln('roznica=',  roznica);
  writeln('proizvedenie= ', umnoj);
  writeln('chastnoe=', deli);
end;
end.
```

Unit127_1b - модуль с классом потомка

```
unit Unit27_1b;
interface
uses Unit27_1a;
Type Tdzialania2=class(Tdzialania)                  //наследование
       Protected
         Function deli:Real; override;      //полиморфизм
```

```
nd;
implementation
{ Tdzialania2 }
function Tdzialania2.deli: Real;
begin
  writeln('klasa Tdzialania2');
  writeln('-----------------');
        if b<>0 then Result:=a/b
            else begin Result:=0; writeln (Ochibka delenie na 0');end;
end;
end.
```

Lab27_1.dpr – главная программа

```
program Lab27_1;
{$APPTYPE CONSOLE}
Uses
  SysUtils,
  Unit27_1a in 'Unit27_1a.pas',
  Unit27_1b in 'Unit27_1b.pas';
Var dzialania : Tdzialania;
    dzialania2 : Tdzialania2;
BEGIN
  dzialania := Tdzialania.Create;
  dzialania . dane;
  //dzialania . dane(1,0);
  dzialania .rezultaty;
  dzialania.Destroy;
  dzialania2 := Tdzialania2.Create;
  dzialania2 . dane;
  //dzialania . dane(1,0);
  dzialania2 .rezultaty;
  dzialania2.Destroy;
  READLN;
END.
```

- ✓ Нажав клавишу <F9>, выполнить программу для произвольных данных и другой раз для 1,0 (деление через нуль – метода rezultaty запускает соответствующую методу iloraz)
- ✓ Нажав клавишу <ENTER>, завершить консольное приложение

Задача 27.2.

Составить программу для вычисления поля поверхности фигур: круга, квадрата, прямоугольника с использованием свойств инкапсуляции и наследования

- ✓ В одном модуле пуст находится класс Tkolo, как класс предка
- ✓ В другом класс Tkwadrat как класс потомка от Tkolo
- ✓ В третьем класс Tprostokat как класс потомка от Tkolo

Задача 27.3.

Создать объективно-ориентированную программу для вычисления зарплаты 2-х работников как в задаче 26.3, но с учетом премии для каждого работника

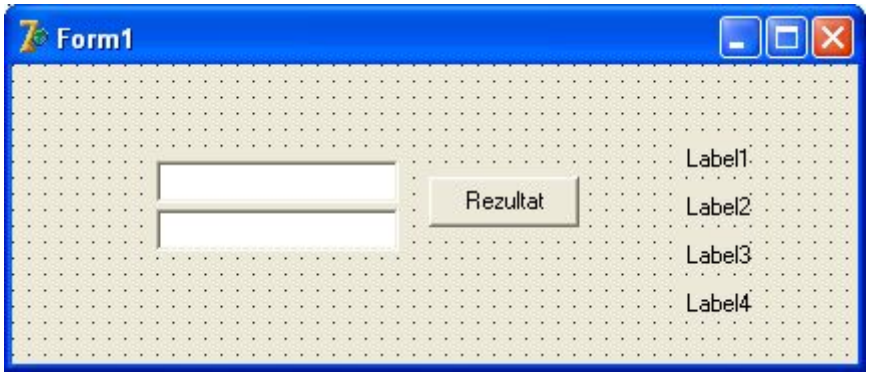
- ✓ Создать консольное приложение с модулем содержащим класс TRobotnik (задача 26.3) и с модулем содержащим класс TRobotnik2 будущим потомком класса TRobotnik

Лабораторное занятие № 28
Тема: *Парадигма событийного и визуального программирования. Программы, управляемые событиями. Компонентное программирование.*

План:
Событийное программирование
Визуальное компонентное программирование в среде Delphi
Среда разработки Delphi
Проект Delphi
Компоненты Form, Label, Edit, Button
Вычислительные программы

Задача 28.1.

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Открыть файл с проектом *Lab1e.dpr* ***File- Open Project*** (**.dpr*)



- ✓ и файл с модулем *Unit1e.pas* (View – Units) в окне редактора кода ознакомиться со структурой программы и с использованием компонентов как объектов данных классов – объяснить значение строк обозначенных //???????

Unit1e.pas – модуль

```
unit Unit1e;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  Dialogs, StdCtrls;
type
  TForm1 = class(TForm)        //???????
    Button1: TButton;          //???????
    Edit1: TEdit;          //???????
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure Button1Click(Sender: TObject);  //???????
  Private
    { Private declarations }
  Public
    { Public declarations }
  end;
var
  Form1: TForm1;               //???????
```

```
Implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);
begin
  try
  label1.Caption:='summa='+IntToStr(StrToInt(Edit1.Text)+     StrToInt(Edit2.Text));//???????
  label2.Caption:='roznica='+IntToStr(StrToInt(Edit1.Text)- StrToInt(Edit2.Text));
  label3.Caption:='proizvedenie='+IntToStr(StrToInt(Edit1.Text)*StrToInt(Edit2.Text));
  label4.Caption:='chastnoe='+FloatToStr(StrToInt(Edit1.Text)/ StrToInt(Edit2.Text));
  except
  //ShowMessage('Ochibka');              //???????
  on EConvertError do ShowMessage ('Ochibka preobrazovanie);                  //???????
  On EZeroDivide do ShowMessage ('Ochibka delenie na 0');    //???????
  end;
end;
end.
```

Lab1e.dpr – главная программа

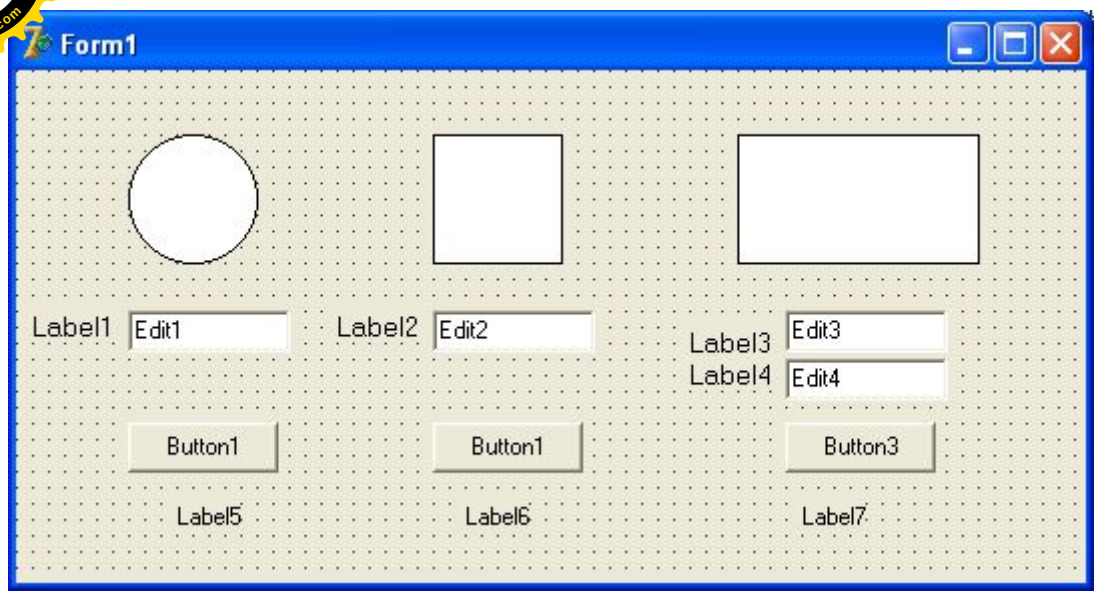
```
program Lab1e;
uses
  Forms,
  Unit1e in 'Unit1e.pas' {Form1};       //???????

{$R *.res}

begin
  Application.Initialize;                          //???????
  Application.CreateForm(TForm1, Form1);     //???????
  Application.Run;                                        //???????
end.
```

- ✓ Нажав клавишу <F9>, выполнить программу.
- ✓ Щелкнув на кнопке с крестиком в верхнем правом углу формы, завершите оконное приложение

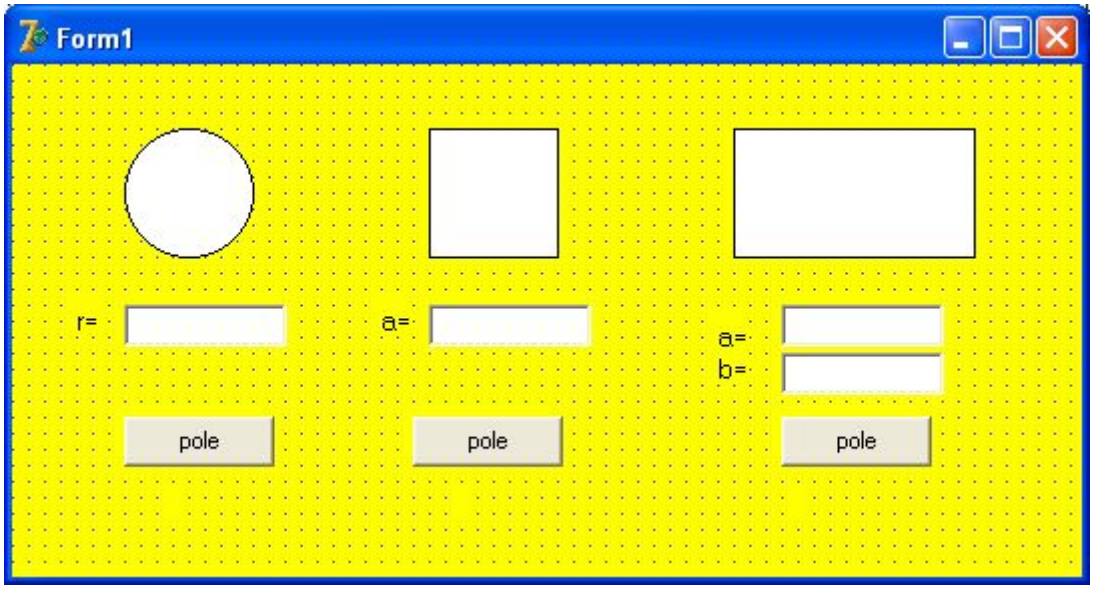
Задача 28.2.

Составить программу для вычисления площади поверхности фигур: круга, квадрата, прямоугольника

- ✓ Создать новое оконное приложение ***File-New-Application***
- ✓ Разместить на форме компоненты Edit, Label, Button (Standard) и Shape (Additional) следующим образом
- ✓ В Окне инспектора объектов изменить свойства компонентов:
  - Form: Color, Width, Heigth
  - Shape: Shape
  - Label: Caption, Font
  - Edit: Text,
  - Button: Caption
- ✓ При щелчке на кнопке Button1 2 раза, автоматически в коде модуля появляется пустая процедура, обслуживания события OnClick, которую заполним кодом вычисления площади круга:



○




```
procedure TForm1.Button1Click(Sender: TObject);
var r, pole:real;
begin
  r:= StrToFloat(Edit1.Text);
  pole:=pi*sqr(r);
  Label5.Caption:=FloatToStrF(pole, ffFixed, 5,2);
end;
```

- ✓ Выполним программу, обращая внимание на вид результата
- ✓ Изменим вид результата

```
procedure TForm1.Button1Click(Sender: TObject);
var r, pole:real;
begin
  r:= StrToFloat(Edit1.Text);
```

```
  ::=pi*sqr(r);
  //Label5.Caption:=FloatToStr(pole);
  Label5.Caption:=FloatToStrF(pole, ffFixed, 5,2);
end;
```

- ✓ Аналогично добавим к программе процедуры для остальных кнопок.
- ✓ Выполним программу
- ✓ Обеспечим программу от ввода плохих данных, используя оператор ***try...except...end***

```
procedure TForm1.Button1Click(Sender: TObject);
var r, pole:real;
begin
  try
  r:= StrToFloat(Edit1.Text);
  pole:=pi*sqr(r);
  //Label5.Caption:=FloatToStr(pole);
  Label5.Caption:=FloatToStrF(pole, ffFixed, 5,2);
  except
  ShowMessage('Ochibka dannyx');
  end;
end;
```

- ✓ Чтобы оператор ***try...except...end*** правильно задействовал в среде Дельфи нужно изменить опции среды ***Tools- Debugger Option*** – нужно выключить поле выбора ***Integrated Debugging***
- ✓ Выполним программу при правильных и неправильных данных

Лабораторное занятие № 29

Тема: *Парадигма собитийного и визуального программирования. Программы, управляемые событиями. Компонентное программирование.*

План:

Событийное программирование
Визуальное компонентное программирование в среде Delphi
Среда разработки Delphi
Проект Delphi
Компоненты Form, Label, Edit, Button
Вычислительные программы

Задача 29.1.

Создать оконное приложение программы вычисляющей корни квадратного уравнения

- ✓ Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7***
- ✓ Создать новое оконное приложение ***File-New-Application***
- ✓ Разместить на форме компоненты и в окне инспектора кода придать им значения свойств рис 2:
- ✓ Записать программу в папке Lab29 под именем Lab29_1.dpr, а модуль с формой Unit29_1.pas
- ✓ Ввести код процедуры обслуживания события нажатия кнопки koniec (конец) и oblicz (считай)

Модуль

```
unit Unit29_1.pas   // Модуль решения квадратного уравнения

unit Unit29_1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  StdCtrls, ExtCtrls;
Type
```

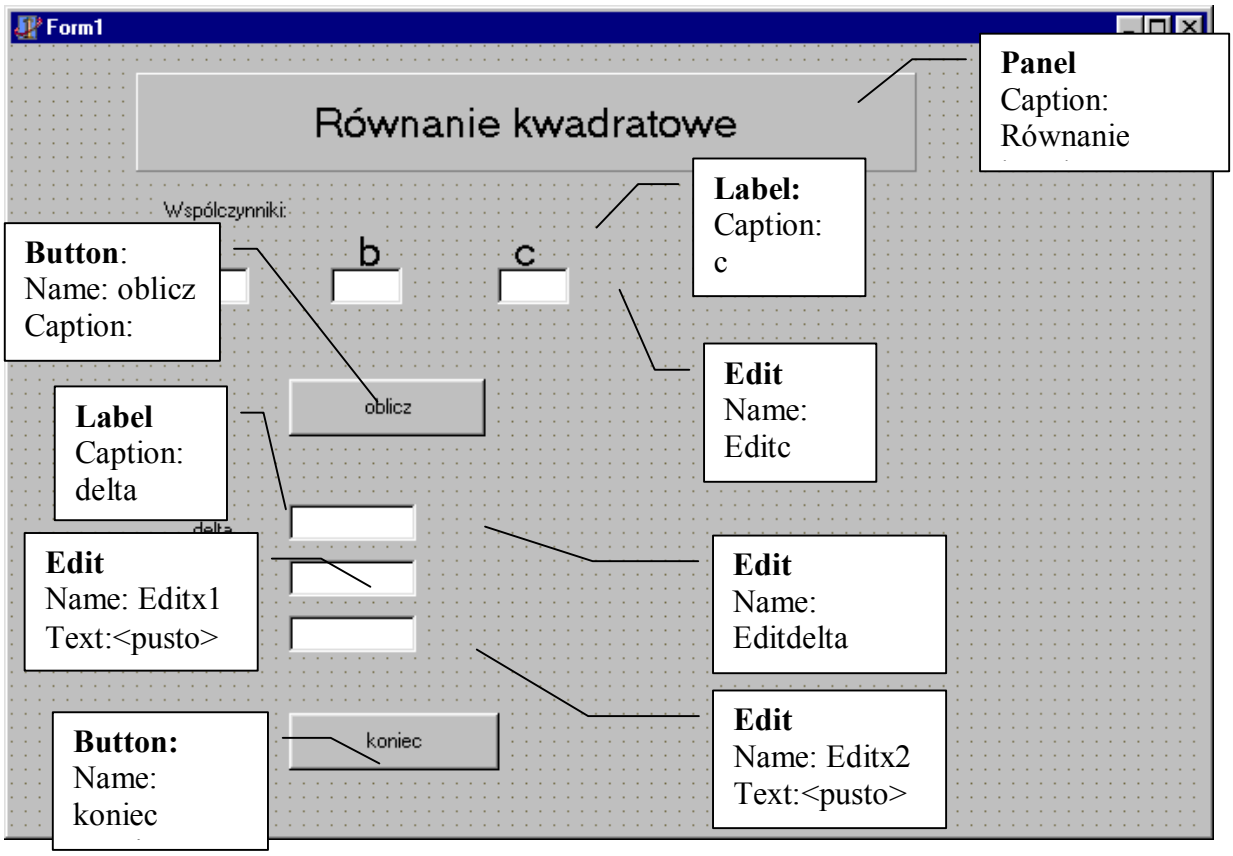


Рис 2

```
TForm1 = class(TForm)
   Panel1: TPanel;
   a: TLabel;
   b: TLabel;
   c: TLabel;
   Edita: TEdit;
   Editb: TEdit;
   Editc: TEdit;
   wsp: TLabel;
   oblicz: TButton;
   x1: TLabel;
   x2: TLabel;
   delta: TLabel;
   Editdelta: TEdit;
   Editx1: TEdit;
   Editx2: TEdit;
   koniec: TButton;
   procedure obliczClick(Sender: TObject);
   procedure koniecClick(Sender: TObject);
 private
   { Private declarations }
 public
   { Public declarations }
 end;
```

```
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.obliczClick(Sender: TObject);
var d:Real;
begin
  d:=sqr(strtofloat(editb.text)) - 4* strtofloat(edita.text)*strtofloat(editc.text);
  editdelta.text:=floattostr(d);
  if d>0 then
  begin
     editx1.text:=floattostr((-strtofloat(editb.text) - sqrt(d))/2*strtofloat(edita.text));
    editx2.text:=floattostr((-strtofloat(editb.text) + sqrt(d))/2*strtofloat(edita.text));
 end
 else
   if d=0 then
   begin
     editx1.text:=floattostr(-strtofloat(editb.text) / 2*strtofloat(edita.text));
     editx2.text:=floattostr(-strtofloat(editb.text) / 2*strtofloat(edita.text));
   end
   else
   begin
     editx1.text:='brak';
     editx2.text:='brak';
   end;
end;
procedure TForm1.koniecClick(Sender: TObject);
begin
  Close;
end;
end.
```

Project Lab29_1.dpr

```
program Lab29_1;
uses
  Forms,
  Unit29_1 in 'Unit29_1.pas ' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Задача 29.2.
Создать оконное приложение, вычисляющего зарплату рабочего (плата за час, количество часов и премия)
Задача 29.3.
Создать оконное приложение, вычисляющего среднего из 5 оценок студента

Лабораторное занятие № 30
Тема: *Библиотека компонентов VCL Delphi*

План:
Компоненты карты Standard
Примеры использования компонентов в визуальном программировании

Задача 30.1.

Создать оконное приложение программы демонструющей выбранные компоненты ***VCL*** карты *Standard*, которая:

- ✓ позваляет ввести разные данные о студенте с использованием разных компонентов
- ✓ выводит все данные в компоненте *Memo*
- ✓ меняет цвет и шрифт компонента *Memo*

1. Запустить среду программирования Delphi ***Пуск-Программы-Borland Delphi 7 - Delphi 7.***
   Создать новое оконное приложение ***File-New-Application***
   Записать в фолдере ***Lab30*** файл с модулем ***Unit30.pas*** и файл с проектом ***Lab30.dpr***
2. Разместить на форме компоненты и в окне инспектора объектов придать им значения свойств (не все компоненты описаны) рис 3:
3. Напиши код для событий:
   - ✓ Выбор роста студента компонентом *Scrollbar1* (событие ***OnChange***, надо отнестись к ***Scrollbar1.Position –*** при этом расположение движка видно в окне ***Edit2)***
   - ✓ Нажатие кнопки *wyswietl* – в строках компонента Memo1должны печататся данные студента (событие ***OnClick***: добавление строки: процедура ***Memo1.Lines.Add(<текст>),*** если текстом является выбранная строка *ListBoxa:* ***Memo1.Lines.Add (ListBox1.Items [ListBox1. ItemIndex])***
   - ✓ Нажатие кнопки wyczysc (*Memo* пустое, цвет белый) – процедура ***Memo1.Clear***
   - ✓ Выбор цвета компонента *Memo* с помощью *Radiogroup2* (событие ***OnChange***, свойство ***ItemIndex*** (начинается с нуля), обозначение цветов: красный: *clRed*, зеленый: *clLime)*
   - ✓ Изменение шрифта в компоненте *Memo1* (курсива***:[fsItalic]***, простой шрифт ***[]***)
4. Добавить к программе:
   - ✓ Два новых цвета для *Memo1 (RadioGroup2*) и два новых вида шрифта *Memo1(CheckBox*)
   - ✓ Оценку студента (ввести в *Edit* и вывести в *Memo1)*

```
unit Unit30;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, ExtCtrls, StdCtrls;
Type
  TForm1 = class(TForm)
        Label1: TLabel;
        Edit1: TEdit;
        Label2: TLabel;
        ComboBox1: TComboBox;
        Label3: TLabel;
        ListBox1: TListBox;
        ScrollBar1: TScrollBar;
        Label4: TLabel;
        Label5: TLabel;
        Label6: TLabel;
        RadioGroup1: TRadioGroup;
        GroupBox1: TGroupBox;
        CheckBox3: TCheckBox;
        CheckBox4: TCheckBox;
        CheckBox5: TCheckBox;
        Button1: TButton;
        Button2: TButton;
        Panel1: TPanel;
        RadioGroup2: TRadioGroup;
```

```
                CheckBox1: TCheckBox;
                Label7: TLabel;
                Edit2: TEdit;
                memo1: TMemo;
                procedure ScrollBar1Change(Sender: TObject);
                procedure Button1Click(Sender: TObject);
                procedure Button2Click(Sender: TObject);
                procedure CheckBox1Click(Sender: TObject);
                procedure RadioGroup2Click(Sender: TObject);
                procedure wyswietl1Click(Sender: TObject);
                procedure wyczysc1Click(Sender: TObject);
                procedure zmienkolor2Click(Sender: TObject);
                procedure czcionkaBold1Click(Sender: TObject);
               private
                { Private declarations }
                 public
                   { Public declarations }
               end;
      var
        Form1: TForm1;
implementation
{$R *.DFM}
```

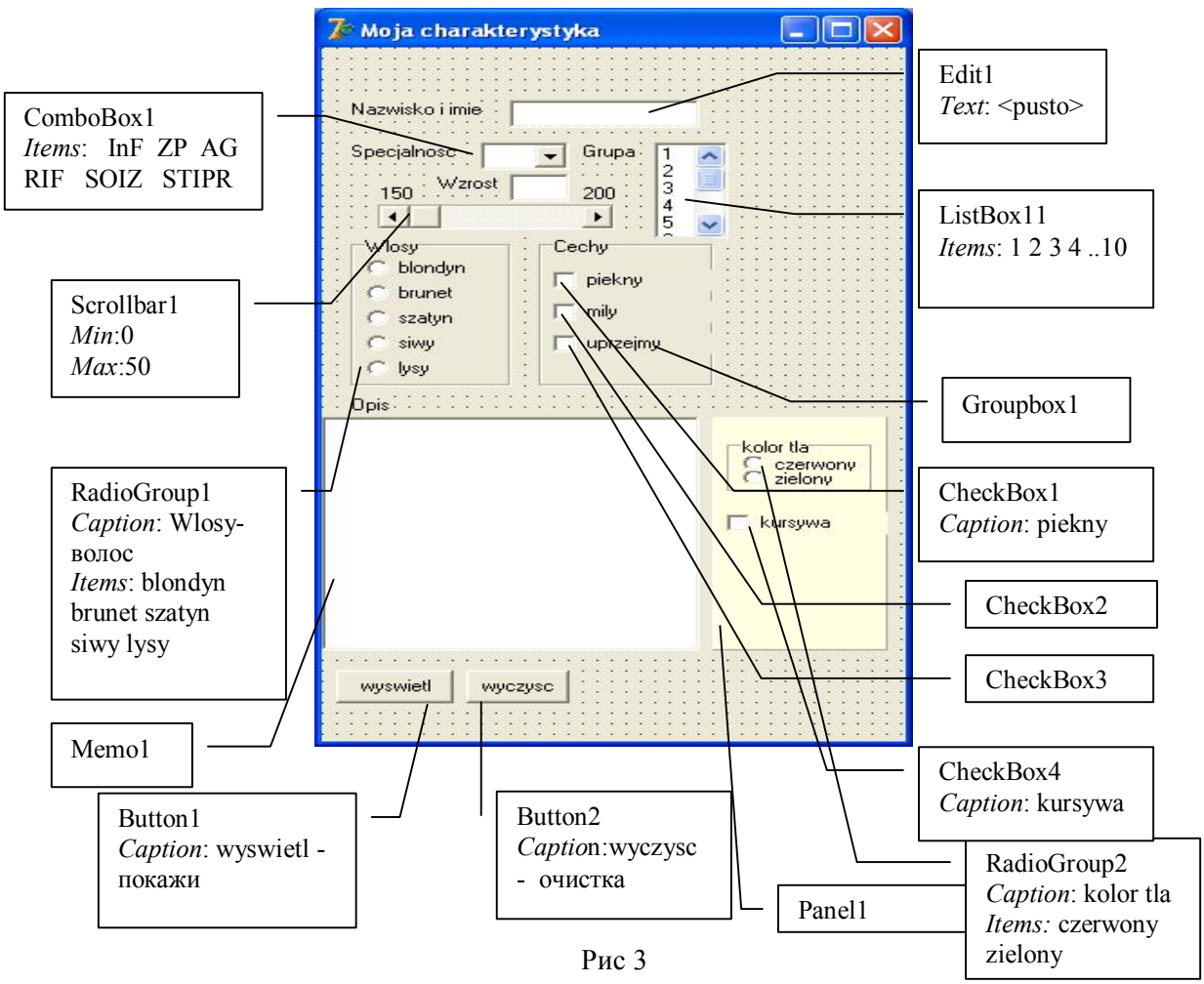
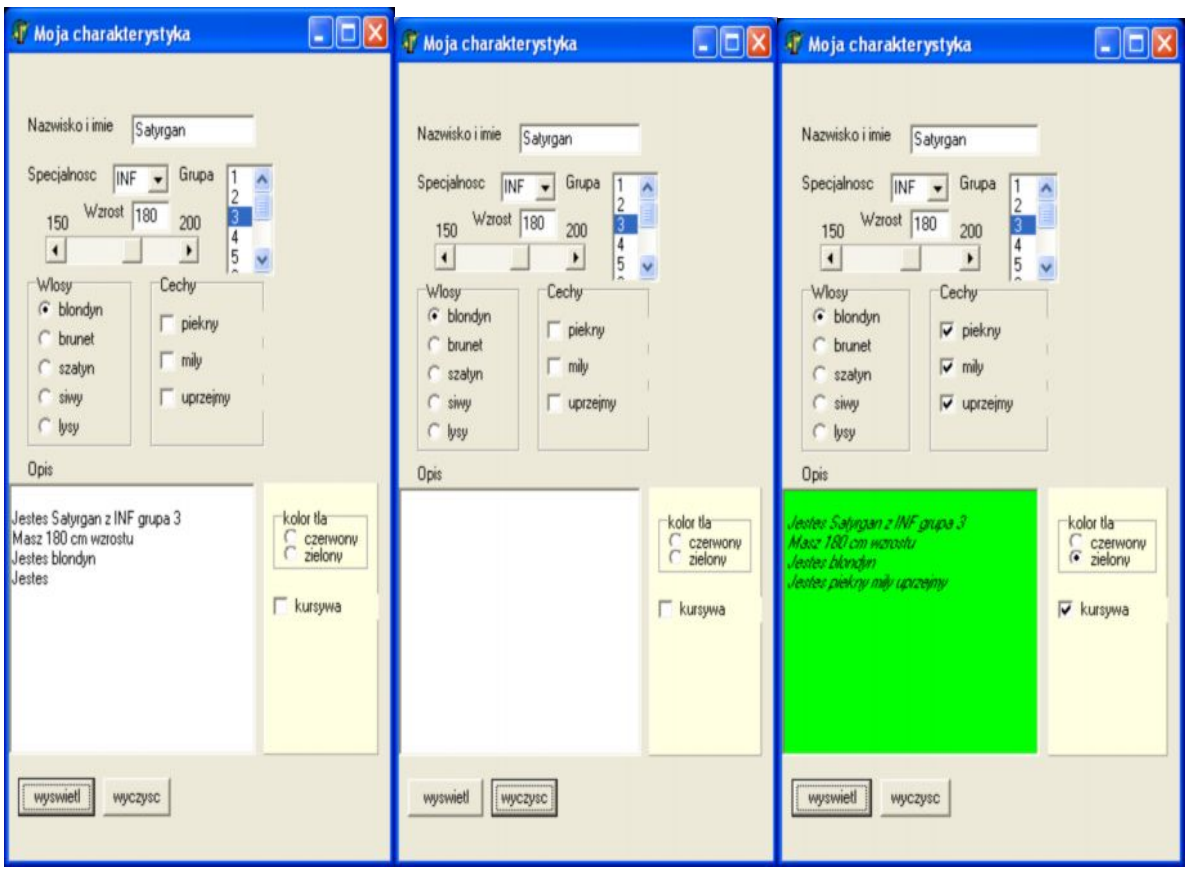


Рис 3

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  Edit2.text:=IntToStr(ScrollBar1.Position+150)
end;

procedure TForm1.Button1Click(Sender: TObject);
var cecha:string;
begin
  Memo1.Lines.Add('Jestes '+Edit1.text + ' z ' + Combobox1.text + ' grupa ' +
ListBox1.Items[ListBox1.ItemIndex]);
  Memo1.Lines.Add('Masz ' +  Edit2.text + ' cm wzrostu');
  case Radiogroup1.ItemIndex of
    0:  Memo1.Lines.Add('Jestes blondyn');  // Jestes - ты
    1:  Memo1.Lines.Add('Jestes brunet');
    2:  Memo1.Lines.Add('Jestes szatyn');
    3:  Memo1.Lines.Add('Jestes siwy');       // siwy - седой
    4:  Memo1.Lines.Add('Jestes lysy');       // lysy - лысый
  end;
cecha:='Jestes ';
  if CheckBox3.Checked then cecha:=cecha+'piekny';       // piekny - красив
  if CheckBox4.Checked then cecha:=cecha+' mily';        // mily - милый
  if CheckBox5.Checked then cecha:=cecha+' uprzejmy';  // uprzejmy - вежлив
  Memo1.Lines.Add(cecha);
end;
```



```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo1.Clear;
  Memo1.color:=clWhite;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then Memo1.Font.Style:=[fsItalic]
                  else memo1.Font.Style:=[];
end;
procedure TForm1.RadioGroup2Click(Sender: TObject);
begin
  case Radiogroup2.ItemIndex of
    0: Memo1.color:=clRed;
    1: Memo1.Color:=clLime;
  end;
end;

procedure TForm1.wyswietl1Click(Sender: TObject);
var cecha:string;
begin
  Memo1.Lines.Add('Jestes '+Edit1.text+ ' z '+Combobox1.text+' grupa '+
  ListBox1.Items[ListBox1.ItemIndex]);
  Memo1.Lines.Add('Masz '+ Edit2.text+' cm wzrostu');
  case Radiogroup1.ItemIndex of
    0:  Memo1.Lines.Add('Jestes blondyn');
    1:  Memo1.Lines.Add('Jestes brunet');
    2:  Memo1.Lines.Add('Jestes szatyn');
    3:  Memo1.Lines.Add('Jestes siwy');
    4:  Memo1.Lines.Add('Jestes lysy');
  end;
  cecha:='Jestes ';
  if CheckBox3.Checked then cecha:=cecha+'piekny';
  if CheckBox4.Checked then cecha:=cecha+' mily';
  if CheckBox5.Checked then cecha:=cecha+' uprzejmy';
  Memo1.Lines.Add(cecha);
end;

procedure TForm1.wyczysc1Click(Sender: TObject);
begin
  Memo1.Clear;
end;

procedure TForm1.zmienkolor2Click(Sender: TObject);
begin
  Memo1.Color:=clBlue;
end;

procedure TForm1.czcionkaBold1Click(Sender: TObject);
begin
  Memo1.Font.Style:=[fsBold]
end;

end.
```

ABBYY PDF Transformer 2.0 Click here to buy www.ABBYY.com

Литература

1. Абрамов В. Г., Трифонов Н. Н., Трифонова Г. Н. Введение в язык паскаль. –М., 1988г.
2. Вирт Н. Алгоритмы+структура данных=программа. –М. Мир, 1985г.
3. Керман М. Программирование и отладка в Delphi. –М., 2004г.
4. Архангельский А. Я. Delphi 7. Справочное пособие. –М., 2003г.
5. Камаев В. А., Костерин В. В. Технологии программирования. –Высшая школа, М., 2006г.
6. Фаронов В. В. Система программирования Delphi. –СП., 2003г.
7. Хомоненко А. и др. Delphi. –СП., 2004г.