Материалы к экзамену. Курс «Программирование»

Вопросы к экзамену «Программирование» 2 курс 3 семестр

1.Понятие алгоритма. Свойства и классификация алгоритмов.алгоритмов. Основные алгоритмические структуры. Методы описания алгоритмов.Язык блок-схем.

2.История развития языков программирования. Парадигмы программирования. Операциональное и структурное программирование. Принципы структурного программирования.

3.Понятие об объектно-ориентированном (ООП)программировании. Визуальное программирование. Общие принципы построения ООП – инкапсуляция, полиморфизм, наследование.

4.Рекурсия и ее использование в программировании. Итерационные вычисления и формулы.

5.Интегрированная среда delphi. Состав и назначение интегрированной среды разработки (ИСР).

6.Язык программирования Object Pascal. Типы данных и операции языка Паскаль. Константы и переменные Паскаля. Построение выражений. Основные простые типы данных.

7.Операторы языка Object Pascal. Операторы ветвления. Операторы ветвления if и case. Составной оператор.

8.Операторы языка Object Pascal. Циклы и их организация. Циклы с известным числом повторений. Циклы с не известным числом повторений.

9.Язык программирования Object Pascal. Массивы и работа с ними. Массивы и их использование в программах Паскаля.

10.Задачи обработки массивов. Поиск и выборка данных. Сортировка данных. Простые и. быстрые алгоритмы.

11.Работа со строковыми переменными. Строковый и знаковый типы данных Паскаля. Процедуры и функции для работы со строками.

12. Стандартные функции и процедуры Паскаля. Подпрограммы пользователя в Паскале. Процедуры и функции Delphi.

13.Стандартные классы и компоненты Delphi. Палитра компонентов.

14.Работа с деревьями и списками в Delphi. Компоненты работающие со списками.

15.Компонент StringGrid. Использование StringGrid в проектах Delphi.

16.Компоненты диалогов. Диалоги выбора шрифта, выбора цвета. Диалоги открытия и сохранения файлов.

17.Основы компьютерной графики. Графические построения и моделирование цветов.

18.Классы графики в Delphi. Класс Tcanvas.

19.Компоненты для графических построений. Компоненты Image, Paintbox, Shape, Timer.

20.Классы графики в Delphi. Классы TPen, TBrash, TPoint.

21.Алгоритмы графических построений. Построение графика функции.

22.Алгоритмы графических построений. Понятие о геометрическом моделировании.

23.Классы и объекты в Delphi. Структура класса.

24.Свойства и методы класса. Конструктор и деструктор класса.

25.Полиморфизм и наследование классов в Delphi.

26.События и их обработка. Обработка исключительных ситуаций. Событийное управление работой проекта.

27.Логические основы алгоритмизации. Логические операции и выражения. Логический тип данных языка Паскаль. Логические выражения в Паскале.

28.Эффективность и скорость алгоритмов. Вспомогательные алгоритмы.

29.Языки и средства программирования.

30.Особенности языка Паскаль. Структура программы и ее разделы. Ввод и вывод данных.

1.Понятие алгоритма. Свойства и классификация алгоритмов. Основные алгоритмические структуры. Методы описания алгоритмов.Язык блок-схем.

Понятие алгоритма

Алгоритм относится к основным понятиям математики, а потому не имеет точного определения. Часто это понятие формулируют так: *«точное предписание о порядке выполнения действий из заданного фиксированного множества действий направленных для решения всех задач заданного класса»*.

Рассмотрим подробнее ключевые слова в этой формулировке:

− ***«точное предписание»*** означает, что предписание однозначно и одинаково понимается всеми исполнителями алгоритма и при одних и тех же исходных данных любой исполнитель получает один и тот же результат;

− ***«из заданного фиксированного множества»*** означает, что множество действий, используемых в предписании, оговорено заранее и не может меняться в ходе исполнения алгоритма;

− ***«решения всех задач заданного класса»*** означает, что это предписание предназначено для решения класса задач, а не для одной отдельной задачи.

Алгоритм всегда определяет однозначно, какое действие должно быть выполняется на каждом шаге алгоритма, так и то, какой шаг следует за текущим.

Применительно к ЭВМ алгоритм определяет вычислительный процесс, начинающийся с обработки некоторой совокупности возможных исходных данных и направленный на получение определённых этими исходными данными результатов.

Для задания алгоритма необходимо описать следующие его элементы:

- набор объектов, составляющих совокупность возможных исходных данных, промежуточных и конечных результатов;
- правило начала;
- правило непосредственной переработки информации (описание последовательности действий);
- правило окончания;
- правило извлечения результатов.

Понятие исполнителя алгоритма

Алгоритм всегда рассчитан на конкретного исполнителя. **Исполнитель** в информатике – человек или автоматическое устройство, которому поручается исполнить алгоритм или программу. Исполнителем может быть человек, группа людей, робот, станок, компьютер, язык программирования и т. д. Важнейшим свойством, характеризующим любого из исполнителей, является умение исполнителя выполнять некоторые команды. Так, исполнитель-человек умеет выполнять такие команды, как «встать», «сесть», «включить компьютер» и т. д., а исполнитель-язык программирования Бейсик – команды PRINT, END, LIST и другие аналогичные операторы языка. Вся совокупность команд, которые данный исполнитель умеет выполнять, называется **системой команд исполнителя** (СКИ). Область, в пределах которой действует исполнитель, называется **средой исполнителя**. Одно из принципиальных обстоятельств состоит в том, что исполнитель не вникает в смысл того, что делает, но получает необходимый результат. В таком случае говорят, что исполнитель действует **формально**, то есть отвлекается от поставленной задачи и только строго выполняет некоторые требования, инструкции. Это важная особенность алгоритмов. Наличие алгоритма формализует процесс решение задачи, исключает неоднозначность поведения исполнителя. Использование алгоритма даёт возможность решать задачу формально, механически исполняя команды алгоритма в указанной последовательности. Целесообразность предусматриваемых алгоритмом действий обеспечивается точным анализом со стороны того, кто составляет этот алгоритм.

Введение в рассмотрение понятия «исполнитель» позволяет определить **алгоритм** как понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение поставленной цели. Наиболее распространенными и привычными являются алгоритмы работы с величинами – числовыми, символьными, логическими и т. д.

Классификация алгоритмов

Так как алгоритмов очень много существует множество вариантов их классификации. На практике наиболее употребительна классификация по типу решаемых задач.

1. Вычислительный алгоритм (обработка некоторой совокупности возможных исходных данных и получение результата).
2. Логический алгоритм (проверка условия).
3. Моделирующий алгоритм (алгоритм создания и заданного функционирования математической модели). Данный тип алгоритмов используется для описания поведения модели объекта.
4. Адаптивный алгоритм (обладает способностью настраиваться на решаемую задачу).
5. Вероятностный алгоритм (использует случайные данные, результат его так же в каком-то смысле случайный).
6. Алгоритм формирования и функционирования объекта, объектно-ориентированное программирование. Описывает объект какого-то класса. От моделирующего отличается тем, что объект реальный (в моделирующем алгоритме) и объект класса различны по своей природе. Реальный объект – это объект, существующий в природе, для которого создаётся математическая модель и затем данная модель и её функционирование реализуются с помощью алгоритма. Объект класса – это просто структура данных.

Любой возможный алгоритм может быть представлен в виде совокупности трёх основных конструкций: линейной, разветвлённой, циклической.

Линейные алгоритмы

Линейным называется алгоритм, в котором все этапы решения задачи выполняются последовательно. Каждая операция является самостоятельной, независимой от каких- либо условий. На схеме блоки, отображающие эти операции, располагаются в линейной последовательности.

Данный алгоритм имеет следующий формат:

Начало <последовательность выполняемых команд>; Конец.

Разветвленные алгоритмы

Ветвящимся называется такой алгоритм, в котором выбирается один из нескольких возможных путей (вариантов) вычислительного процесса. Каждый подобный путь называется **ветвью алгоритма**. Признаком ветвящегося алгоритма является наличие операций проверки условия. Обычно различают два вида условий – простые и составные.

Простым условием (отношением) называется выражение, составленное из двух арифметических выражений или двух текстовых величин, связанных одним из знаков: >, <, $\leq$, $\geq$, =, $\neq$.

Из отношений можно образовать **сложные (составные)** выражения, с помощью логических операций И, ИЛИ, НЕ.

Например, x<7 и x>2; y>0 или (y<0 и x<0).

В схеме алгоритма операцию проверки выполняет **логический блок**. Он изображается ромбом, внутри которого указывается проверяемое условие (отношение), и имеет два выхода: Да и Нет.

Если условие (отношение) истинно (выполняется), то выходим из блока по выходу «Да»; если ложно (не выполняется) – по выходу «Нет».

Ветвящийся алгоритм, включающий в себя две ветви, называется простым, более двух ветвей – сложным. Сложный ветвящийся алгоритм можно представить с помощью простых ветвящихся алгоритмов.

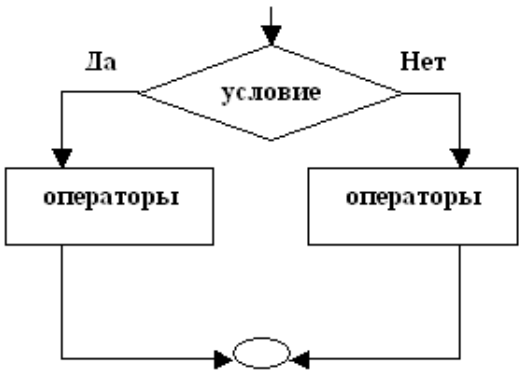
Ветвление реализуется в виде команды:

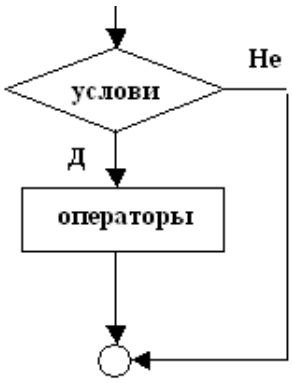
если <ЛВ> то <Серия 1> иначе <Серия 2>.

Здесь **<ЛВ>** – это логическое выражение;

<Серия 1> – описание последовательности действий, которые должны выполниться, когда **<ЛВ>** принимает значение «истина»;

<Серия 2> – описание последовательности действий, которые должны выполниться, когда **<ЛВ>** принимает значение «ложь».





полное не полное ветвление

Любая из этих серий может быть пустой. В этом случае ветвление называется неполным. В противном случае ветвление называется полным.

Циклические алгоритмы

Циклом называется последовательность действий, выполняемых многократно, каждый раз при новых значениях параметров.

Алгоритмы, включающие в себя циклы, называются **циклическими**.

Различают циклические алгоритмы с параметром, с предусловием и с постусловием.

1. **Цикл с параметром** (со счётчиком) применяется в тех случаях, когда в программе какие-то действия (операторы) повторяются и при этом некоторая величина меняется с постоянным шагом.

Формат команды на алгоритмическом языке имеет следующий вид:

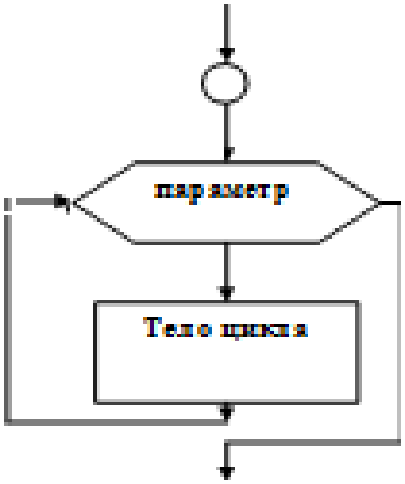
НЦ

Для <параметр цикла> от <начальное значение> до <конечное значение>

Выполнить <оператор>

КЦ

Схема, иллюстрирующая работу оператора цикла с параметром, выглядит следующим образом:



2. Оператор **цикла с предусловием** используется в тех случаях, когда число повторений действий в программе неизвестно. Его общий вид:

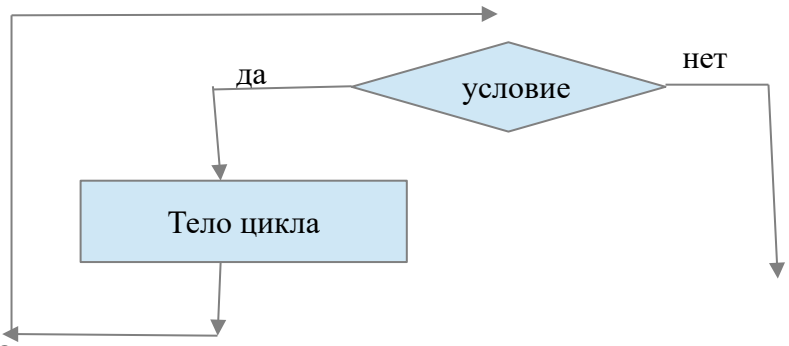
НЦ

Пока <условие>

<тело оператора цикла>

КЦ

Где <условие> – условие, при котором выполняется <тело оператора цикла>. Это условие продолжения цикла. Схема, иллюстрирующая работу данного, цикла имеет вид:



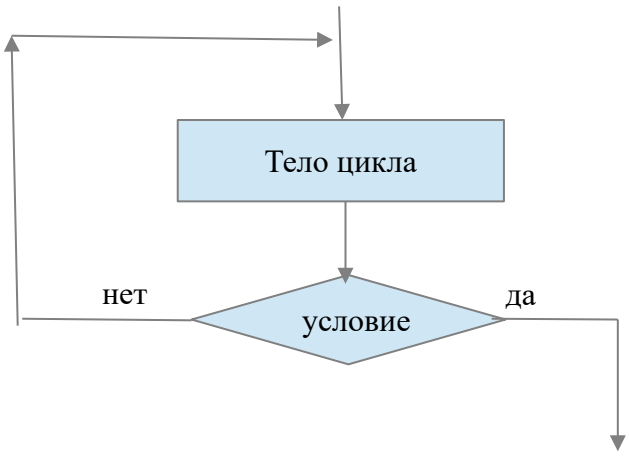
3. Оператор **цикла с постусловием** применяется тогда, когда число повторений действий заранее не известно. Данный оператор отличается от предыдущего тем, что тело цикла повторяется не менее одного раза.

Общий вид этого оператора:

НЦ

<тело оператора цикла>

КЦ при <условие>, где <условие> – условие, при котором оператор прекращает свою работу. Это условие выхода из цикла. Схема, иллюстрирующая работу оператора с постусловием:



Цикл называется **детерминированным**, если число повторений тела цикла заранее известно или оговорено. Примером может служить цикл с использованием оператора FOR.

Цикл называется **итерационным,** если число повторений тела цикла заранее не известно, а зависит от значений параметров (некоторых переменных), участвующих в вычислениях. Например, цикл с оператором WHILE DO или REPEAT UNTIL.

Формы записи алгоритмов. Язык блок-схем

Среди форм записи алгоритмов можно выделить:

- словесная (запись обычным текстом, пример — рецепт);
- блок-схемы (запись с помощью языка блок-схем, примеры — выше приведены схемы циклов);
- языки программирования (запись с помощью операторов языка программирования);
- специальные автоматы (запись команды автомата, машины Тьюринга, машины Поста и т. д.).

Язык блок-схем использует ряд блоков для указания элементов алгоритма:

- прямоугольник, используется для указания блока простого процесса (присваивание, вычисление);
- ромб, используется для блока условия с 2 вариантами (да/нет) исходящих из концов ромба:
- шестиугольник, используется для блока цикла с параметром, внутри указывают имя параметра=начальное значение параметра, конечное значение параметра, шаг изменения параметра (по умолчанию шаг равен 1), пример — К=12,25,2;
- овал (прямоугольник со скругленными угдами), используется для блока начала, конца,

возврата, внутри имеет соответствующую текстовую запись, используется в начале или конце блок-схемы;

- прямоугольник с двойными боковыми линиями, используется для блока вызова предопределен-ного процесса (подпрограммы, процедуры, функции), внутри записывается имя и параметры предопределенного процесса;
- окружность, используется для блока-соединителя, который показывает соединение частей блок-схемы, расположенных в разных частях или листах схемы, внутри указывается номер соединения (так что-бы соединялись те блоки-соединители, которые имеют одинаковые номера), иногда блоки нумеруют, тогда внутри указывают номер блока, куда ведет соединение.

Блоки схемы соединяются линиями со стрелками, которые показывают направление порядка выполнения блоков в алгоритме. В схеме обязательно есть блок начало и блок конец, которые показывают начало и завершение алгоритма. В блок-схеме не должно быть блоков не имеющих входных и выходных линий («висящих»). От начального до конечного блока должна быть непрерывная цепь блоков. Если в алгоритме используется предопределенный процесс, то указывается блок его вызова и отдельно строится блок-схема этого процесса. Она подобна блок-схеме алгоритма, но завершается не блоком конца, а блоком возврат (так как по завершению этого процесса мы должны вернуться в основной алгоритм).

2.История развития языков программирования. Парадигмы программирования. Операциональное и структурное программирование. Принципы структурного программирования.

Основы программирования

Программирование — процесс создания структуры программы, который происходит по следующим этапам:

1. Постановка задачи (выделяются цель, средства ее достижения и условия решения задачи).
2. Создание алгоритма решения задачи.
3. Запись алгоритма в виде программы на языке программирования.
4. Отладка программы.
5. Тестирование программы.

Разработка программы состоит из двух различных действий – создания алгоритма и представления его в виде программы. При этом, создание алгоритма – это, как правило, наиболее сложный этап, т.к. создать алгоритм – значит, найти метод решения задачи. Поэтому, чтобы понять, как создать алгоритм, необходимо понять процесс решения задачи.

Тестирование алгоритма – это проверка правильности или неправильности работы алгоритма на специально заданных *тестах* или тестовых примерах – задачах с известными входными данными и результатами (иногда достаточны их приближения). Тестовый набор должен быть минимальным и полным, то есть обеспечивающим проверку каждого отдельного типа наборов входных данных, особенно исключительных случаев.

Трассировка – это метод пошаговой фиксации динамического состояния алгоритма на некотором *тесте*. *Трассировка* облегчает отладку и понимание алгоритма.

Процесс поиска и исправления (явных или неявных) ошибок в алгоритме называется **отладкой алгоритма**. Как правило отладка происходит совместно с процессом **трансляции** программы (преобразования из кода языка программирования в команды процессора ЭВМ). Трансляция выполняется либо компилятором, либо интерпретатором языка программирования.

Перед программированием задачи может быть составлена графическая **блок-схема алгоритма**, которая составляется по специальным правилам из специальных графических блоков.

По одной из классификаций методы программирования делятся на следующие типы:

Операциональные

Процедурные, называемые также директивные (directive) или императивными (imperative),

Декларативные (declarative) языки,

Объектно-ориентированные (object-oriented).

Операциональное программирование обычно рассматривается на примере машинно-ориентированного языка Ассемблер (хотя оно присутствовало и в первых языках высокого уровня).

Замечание: При создании компьютерных средств программирования исторически первым явилось использование программ написанных в двоичных кодах (программирование в кодах). В дальнейшем были созданы первые ассемблеры, в которых были специальные текстовые конструкции (команды), которые требовалось переводить (транслировать) в двоичные коды. Так появились трансляторы языков программирования (интерпретаторы и компиляторы). Последующее развитие операционального программирование встретило большие трудности, которые были разрешены появлением декларативного и структурного подхода. Теоретическое развитие структурного подхода (Вирт, Якопини, Тьюринг, Бем) позволило создать процедурные зыки (классическим примером является язык Паскаль, созданный Н.Виртом как образец языка структурного/процедурного программирования).

К процедурным языкам относятся такие классические языки программирования, как Algol, Fortran, Basic, Pascal, C.

Наиболее существенными классами декларативных языков являются **функциональные** (functional), и реляционные - **логические** (logic) языки. К категории функциональных языков относятся, например, Lisp и Haskell. Самым известным языком логического программирования является Prolog (Пролог).

Среди объектно-ориентированных языков программирования (языков ООП) отметим C++, Java, Object Pascal, Python.

Особенность операционального программирования – ориентация на построение инструкции для исполнителя – процессора ЭВМ в терминах его операций. Главный недостаток – сложность программ, злоупотребление оператором перехода GOTO.

Особенность процедурного программирования – переход к более строгому, логически структурному программированию. Основные принципы процедурного программирования – использование стандартных технологий проектирования, структурного и модульного принципов, исключение GOTO, максимальное использование стандартных алгоритмов. Главный недостаток – постепенное загромождение модулей процедурами и функциями, недостаточная читабельность и понимаемость больших программ.

Особенность декларативного программирования – программист только описывает (создает декларацию) задачу, а методы и средства ее решения подбирает специальная программа – ядро языка программирования. Главный недостаток – недостаточное развитие среды и средств программирования для данных языков.

3.Понятие об объектно-ориентированном (ООП)программировании. Визуальное программирование. Общие принципы построения ООП – инкапсуляция, полиморфизм, наследование.

Объектно-ориентированное программирование (ООП) - это результат естественной эволюции более ранних методологий программирования. Потребность в ООП связана со стремительным усложнением разрабатываемых программ и, как следствие, их недостаточной надежностью. Модульное программирование, рассмотренное выше, оказалось не способным решить эту проблему. Можно сказать, что **ООП** - это моделирование объектов посредством иерархически связанных классов. При этом малозначащие детали объекта скрыты от нас, и если мы даем команду какому-то объекту, то он "знает", как ее выполнить. Фундаментальной концепцией в ООП является понятие обязанности или **ответственности** за выполнение действия. Все объекты являются представителями, или **экземплярами**, классов. Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения. Классы представляются в виде иерархической древовидной струк-

туры, в которой классы с более общими чертами располагаются в корне дерева, а специализированные классы и в конечном итоге индивидумы располагаются в ветвях. На рисунке показана одна из возможных иерархий классов, включающая в себя собак Белку и Стрелку, кошку Мурку и утконоса Фросю.

Классы собак, кошек и утконосов являются дочерними по отношению к классу млекопитающих, следовательно наследуют его свойства. При программной реализации этой иерархии логично метод "кормление детенышей" реализовывать в родительском классе, вместо того, чтобы несколько раз дублировать его в каждом из подклассов. **Наследование** свойств родительского класса позволяет использовать их в дочерних классах. Немного другая ситуация с рождением детенышей, ведь утконосы откладывают яйца, а не являются живородящими животными? В этой ситуации выручает свойство **полиформизма**: различные реализации методов могут носить одинаковые имена, а система сама определит какую из реализаций использовать в том или ином случае. В нашем примере следует в классе млекопитающих реализовать метод "потомство" (родить детеныша), в классах собак и кошек этот метод будет отсутствовать (система будет искать его в родительском классе и найдет его там), а в классе утконосов нужно написать новый метод, с тем же именем, но другой реализацией (отложить яйца).
Особенность ООП – использование иерархии классов, соединение в объектах полей и методов работы с полями, полиморфизм используемых методов. Появление ООП стимулировало развитие визуальных сред программирования и переход к событийно-управляемым программам.

В основе ООП лежат три основных понятия:

- инкапсуляция (сокрытие данных в классе или методе);
- наследование;
- полиморфизм.

Инкапсуляцию можно представить, как защитную оболочку вокруг кода данных, с которыми этот код работает. Оболочка задает поведение и защищает код от произвольного доступа извне. Кроме того, инкапсуляция объединяет в одном объекте данные (поля) и действия (методы), что и характеризует главную особенность объекта перед другими типами данных.
Наследование - это процесс, в результате которого один тип (класс) наследует свойства другого типа (класса). Наследование фактически обозначает передачу полей и методов от одного класса (предка) к другому (наследнику). Этот процесс упрощает создание новых классов на основе уже созданных.
Полиморфизм - это концепция, позволяющая иметь различные реализации для одного и того же метода, которые будут выбираться в зависимости от типа объекта, переданного методу при вызове.
Особенность ООП – использование иерархии классов, соединение в объектах полей и методов работы с полями, полиморфизм используемых методов. Появление ООП стимулировало развитие визуальных сред программирования и переход к событийно-управляемым программам.

Основы визуального программирования.

Как известно, сложность разработки больших программных систем заставила разработать

теорию структурно-модульного программирования, а затем и объектно-ориентированного программирования. При этом выявилась потребность автоматизации процесса программирования, так как фактически он создавал очень много рутинной, не творческой деятельности, которую достаточно легко автоматизировать. Развитие таких технологий можно проследить по следующим этапам:

1) Появление развитых сред языка программирования, включающих специальные редакторы, отладчики, сложное меню, панели инструментов и т.д..
2) Создание и подключение библиотек процедур и функций, модулей и т.д..
3) Создание утилит-генераторов, которые могут генерировать значительную часть кода приложения.
4) При создании объектно-ориентированного подхода к программированию, появились и специальные средства визуализации и работы с объектами, классами.
5) Появилась методика событийного управления работой проекта, когда объекты программы функционируют автономно, до наступления определенного события (щелчка кнопки мышки, нажатия клавиши и т.д.).
6) На завершающем этапе появляются специализированные среды визуального программирования. В этих средах основные средства интерфейса, различные объекты и элементы представлены как видимые объекты, которые можно интерактивно располагать в проекте и настраивать. Такие объекты стали называть визуальными компонентами, а процесс проектирования был значительно упрощен и ускорен. Характерным примером такой среды является среда Delphi. Принято называть такие технологические решения технологией RAD. Многие новшества RAD технологии появились в средах разработки СУБД раньше чем в традиционных системах. RAD (Rapid Application Development) – методология быстрой разработки приложений характерна не только для разработки приложений СУБД, но и повсеместно применяется при программировании различных задач.

При этом следует учитывать 2 важных момента – во-первых развитие традиционных средств визуального программирования (Delphi, VisualBasic, Visual C, C#, JBuilder) во многом определяется их использованием для разработки СУБД, во-вторых аналогичным образом строятся и многие среды разработки СУБД (Visual FoxPro, различные Case-системы). При этом нужно учитывать, что многие новшества RAD технологии определяются прежде всего требованиями в средах разработки СУБД.

В настоящее время визуальное программирование принято считать особым направлением развития технологий уже не просто программирования, а более широкого понятия — проектирования компьютерных систем. Разработки постоянно совершенствуются в рамках установленных принципов RAD.

RAD (Rapid Application Development) – методология быстрой разработки приложений, повсеместно применяется при программировании различных задач.

Основные принципы RAD

- Инструментарий должен быть нацелен на минимизацию времени разработки.
- Возможно создание *прототипа* для уточнения требований заказчика. Прототип — упрощенная версия продукта, не имеющая полной функциональности.
- Цикличность разработки: каждая новая версия продукта основывается на оценке заказчика результата работы предыдущей версии.
- Минимизация времени разработки версии, за счёт переноса уже готовых модулей и добавления функциональности в новую версию.
- Команда разработчиков должна тесно сотрудничать, каждый участник должен быть готов выполнять несколько обязанностей (взаимозаменяемость).
- Наличие управления проектом должно минимизировать длительность цикла разработки.

Технология RAD опирается на ряд технологий программирования задач (проектирования задач) таких как:

- Объектно-ориентированный подход.

- Подход визуального программирования.
- Подход событийно-управляемого программирования.

4.Рекурсия и ее использование в программировании. Итерационные вычисления и формулы.

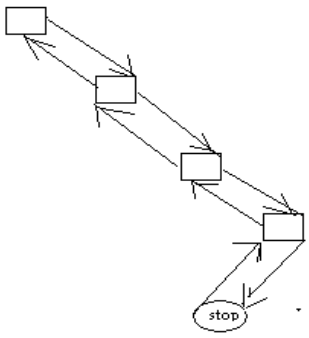
В теории программирования есть еще один особый подход к построению алгоритмов – рекурсивный.

Рекурсия – вызов процедурой или функцией самой себя. Существует много задач легко решаемых рекурсией, более того, доказано, что любую алгоритмическую задачу можно решить рекурсивно. Это методика рекурсивного программирования. Хотя отдельных (рекурсивных) языков программирования так и не было создано, но большинство языков может использовать рекурсию в своих программах.

Рекурсия и ее использование

Рекурсия – вызов процедуры или функции самой себя. Сущаствует много задач легко решаемых рекурсией, более того, доказано, что любую алгоритмическую задачу можно решить рекурсивно. Это методика рекурсивного программирования.

Действие рекурсии можно представлять как вызов процедуры 1 , затем такой же процедуры 2 и т. д. этот вызов должен быть конечным, следовательно, должно быть условие остановки рекурсии. Перед каждым вызовом новой итерации рекурсии это условие должно проверяться. После остановки возвращаемся назад, таким образом, идет движение по кругу: вниз потом вверх (спуск и подъем рекурсии).



Рекурсия необходима для выполнения каких-либо действий, поэтому выделяют следующие варианты использования рекурсии:

- Рекурсия с выполнением действий на спуске (до вызова рекурсии).
- Рекурсия с выполнением действий на подъеме (после вызова рекурсии).
- Рекурсия с выполнением действий на спуске и подъеме.

Если сравнить рекурсию и цикл, то главное отличие в том, что цикл – это повторение действий, рекурсия – новый вызов (новый вариант). Это два принципиально разных подхода к выполнению действий. Теоретически доказано, что оба подхода эквивалентны, но на самом деле простые задачи проще понимаются циклом, а сложные – рекурсией. Таким образом, рекурсивный подход к логическому решению более удобен и логичен.

Следует учитывать, что рекурсивное программирование и циклическое программирование настраивает программистов на разное понимание алгоритмизации. Поэтому навыки программирования в процедурном языке могут быть даже вредны для рекурсивного программирования.

Пример рекурсивного варианта построения функции, осуществляющей вычисление факториала:

Воспользуемся следующим (рекурсивным) представлением факториала:

$$N!=(N-1)!*N = (N-2)!*N*(N-1) = \ldots => F(N) = F(N-1)*N$$

Из него следует, что функцию факториала следует вычислять через себя с порядком на 1 меньше. При этом варианты F(0)=F(1) =1 и F(-K)=0.

```
Function F(N:integer):real;
Var X:real; K:integer;
begin
If N<0 then F:=0
   else
      If N<2 then F:=1
         else
    F:=F(N-1)*N;
end;
```

Итерационные вычисления

Итерационные вычисления производятся по итерационным формулам, которые бывают двух видов:

- явная формула имеет в общем случае вид $x=f(x)$
- неявная формула всегда может быть представлена в виде $F(x)=0$.

Неявные формулы могут быть преобразованы к явному виду. Например линейное преобразование: $x=C*F(x)+x$, где С – некоторая константа, которая выбирается из условия сходимости получаемой итерационной последовательности. Явная формула далее может быть представлена в виде рекуррентной формулы, когда элемент итерационной последовательности x_k выражается в виде

$$x_k = f(x_{k-1}, x_{k-2} \ldots)$$

. Результатом вычисления по такой формуле является итерационная последовательность x_k. Если эта последовательность имеет предел, то он, как правило, является решением задачи итерационного вычисления. Другим вариантом задачи может быть вычисление суммы или произведения (конечных или бесконечных) элементов итерационной последовательности. Количество элементов x_{k-j} в правой части формулы $x_k = F(x_{k-1}, x_{k-2} \ldots x_{k-m})$ определяет m - число начальных значений элементов последовательности, которые потребуются для вычисления. Обычно m=1 и задача требует одно начальное условие, но иногда это не так:

Пример. Вычисление последовательности чисел Фибоначчи по формуле $x_k = x_{k-1} + x_{k-2}$, требует использования 2-х начальных значений $x_0 = 1$ и $x_1 = 1$.

Алгоритм вычисления предела по итерационной формуле:

- Определение начального приближения, которое либо задается, либо определяется из дополнительного условия (например условия сходимости последовательности).
- Организация вычисления последовательности по правилу: $x_1 = f(x_0) \Rightarrow x_2 = f(x_1) \Rightarrow \ldots x_n = f(x_{n-1})$
- Прерывание вычислений производится при достижении косвенной оценки заданной погрешности ε: $|x_{k+1} - x_k| < \varepsilon$

Вычисленный предел является приближенной величиной с заданной погрешностью. Примеры методов и алгоритмов, использующих вычисление предела очень многообразны – итерационные методы решения линейных и нелинейных уравнений, сеточных уравнений, вычисление бесконечных сумм, произведений, комбинаторные вычисления и т. д.. В более сложных случаях последовательность содержит не числа, а некие структуры – вектора, матрицы, строки и т. д.. Однако общий принцип вычисления остается неизменным. Следует учитывать, что итерационная последовательность может сходиться к некоторому пределу или расходится (уход значений на бесконечность). Поэтому вычисление предела последовательности содержит опасность «зацикливания» программы.

Кроме вычисления предела, задача может требовать вычисления суммы, произведения элементов последовательности и т. д.. Характерно, что эти задачи сводятся к тем же итерационным. Например, сумма последовательности может вычисляться по итерационной формуле для сумм $S_{k+1} = S_k + x_k$, начальное условие $S_0 = 0$. Аналогично произведение $P_{k+1} = P_k * x_k$, начальное условие $P_0 = 1$. В этом случае мы вычисляем сразу 2 последовательности S_k и x_k , вторая последовательность называется подпоследовательностью. Выделение

подпоследовательностей часто дает простое решение для сложных задач:

Примеры:

1) Вычисление функции $sin(x) = \sum(-1)^n x^{2n+1} / (2n+1)!$

Здесь выделяется подпоследовательность элементов ряда.

Обозначим элемент ряда $A_k = A_{k-1} * g$ тогда => $g = A_k / A_{k-1} = (-1) x^2 / [(2n+1)*2*n]$. В результате имеем: $S_{k+1} = S_k + A_k$, $A_k = A_{k-1} * g$, $g = (-1) x^2 / [(2n+1)*2*n]$, $S_0 = 0$, $A_0 = x$.

2) Вычисление функции $cos(x) = \sum(-1)^n x^{2n} / (2n)!$

Здесь выделяется подпоследовательность элементов ряда.

Обозначим элемент ряда $A_k = A_{k-1} * g$ тогда => $g = A_k / A_{k-1} = (-1) x^2 / [(2n-1)*2*n]$. В результате имеем: $S_{k+1} = S_k + A_k$, $A_k = A_{k-1} * g$, $g = (-1) x^2 / [(2n-1)*2*n]$, $S_0 = 1$, $A_0 = 1$.

3) Вычисление функции $exp(x) = \sum x^n / (n)!$

Здесь выделяется подпоследовательность элементов ряда.

Обозначим элемент ряда $A_k = A_{k-1} * g$ тогда => $g = A_k / A_{k-1} = x / n$. В результате имеем: $S_{k+1} = S_k + A_k$, $A_k = A_{k-1} * g$, $g = x / n$, $S_0 = 1$, $A_0 = 1$.

4) Вычисление корня $sqrt(x)$. Здесь проще воспользоваться рекуррентным представлением корня. Если $y = sqrt(x)$ => $y^2 = x$ => $y = x/y$. => $y_{k+1} = x/y_k$

Здесь нет необходимости выделять подпоследовательность элементов ряда. Однако эта формула не обладает условием устойчивости, поэтому перестроим ее получив устойчивый вариант: $y_{k+1} = x/(2*y_k) + y_k/2$

В результате имеем: $S_{k+1} = x/(2*S_k) + S_k/2$, $S_0 = 1$.

5) Вычисление полинома по формуле Горнера. Составим алгоритм вычисления *значения* полинома $P_n(x) = a_0x^n + a_1x^{n-1} + ... + a_n$ для заданного *значения* x. Метод решения – метод (схема) Горнера. Опишем его. Заметим, что:

1) при n = 0, $P_0(x) = a_0$;

2) при n = 1, $P_1(x) = a_0x + a_1 = P_0(x)\,x + a_1$;

3) при n = 2, $P_2(x) = a_0x^2 + a_1x + a_2 = (a_0x + a_1)x + a_2 = P_1(x)x + a_2$;

. . .

n) $P_n(x) = a_0x^n + a_1^{xn-1} + ... + a_{n-1}x + a_n = (a_0x^{n-}1 + a_1x^{n-2} + ... + a_{n-1})x + a_n = P_{n-1}(x)x + a_n$.

Таким образом, всегда верно рекуррентное соотношение Горнера:

$$\mathbf{P_{n+1} = P_n * x + a_n}$$

5.Интегрированная среда delphi. Состав и назначение интегрированной среды разработки (ИСР).

Среда пакета Turbo Delphi

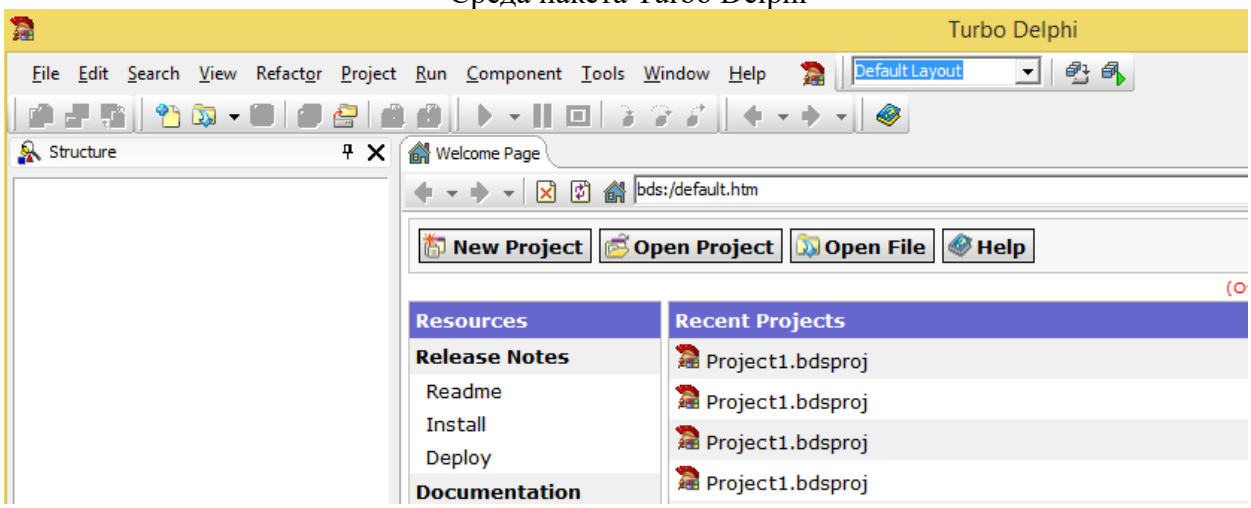


Рисунок 1. Среда Turbo Delphi.

В верхней части окна расположено меню (с выпадающими пунктами). Ниже располагаются пиктограммы панели инструментов. Пока проекта нет, пиктограммы панели инструментов не активизированы (см. Рисунок 1).

Если выбрать пункт меню File/New/VCL Forms Application (см. Рисунок 2), то

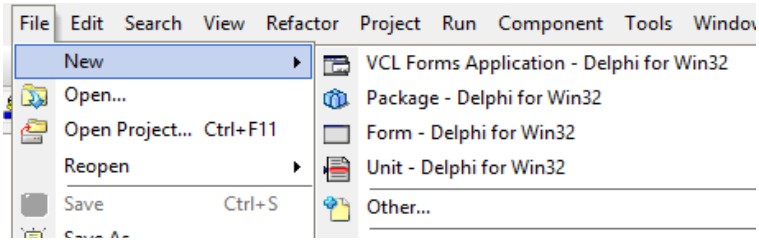


Рисунок 2. Создание проекта VCL

в результате будет создан стандартный VCL проект, который содержит 1 форму Form1 и один модуль Unit 1. Ниже располагаются пункты подменю создания проектов пакета, еще одной формы, еще одного модуля и выбор готового шаблона проекта из репозитория (пункт Other). Примерный внешний вид проекта VCL:

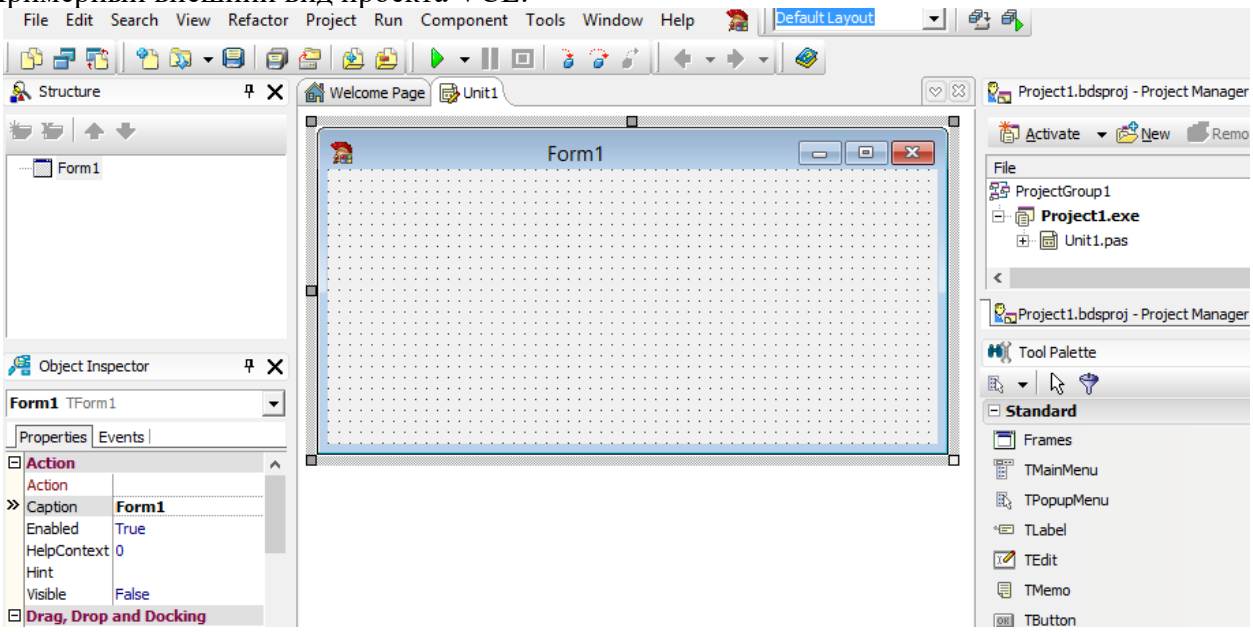


Рисунок 3. Внешний вид проекта VCL

На рисунке 3 виден макет формы проекта и 4 окна: структуры (Structure), инспектора объектов (Object Inspector), менеджера проекта (Project Manager), палитры компонентов (Tool Palette). Теперь пиктограммы панели инструментов уже активны (см. Рисунок 4) и работают. Среди них есть важные для работы (перечислим функции по порядку расположения):

Рисунок 4 — Пиктограммы панели инструментов

Открыть окно модуля, отобразить форму, поменять местами форму и модуль, открыть репозиторий, открыть один из файлов проекта, сохранить все, открыть проект, добавить файл в проект, удалить файл из проекта, запустить проект на исполнение (Run), приостановить работу проекта (Pause). Самая важная здесь пиктограмма запуска проекта (зеленый треугольник), которая заставляет проект пройти компиляцию и запуститься на исполнение. Поясним назначение остальных пиктограмм.

Проект состоит как минимум из формы и модуля. Форма обеспечивает визуальное представление и размещение компонентов. Модуль хранит текст программы Object Pascal для описания содержимого формы проекта (т. е. модуль и форма связаны друг с другом, добавление формы добавляет и модуль для нее). Проект может иметь несколько форм, могут быть и модули без формы (но не наоборот!). Поэтому в проект можно добавлять формы и модули или наоборот удалять. Главное окно проекта устроено так, что в нем есть в страницы — сверху страница с формой, под ней страница модуля. Для переключения между ними внизу окна есть закладки (Code/ Design) :

При щелчке мыши по закладке открывается окно дизайна формы (Design) или кода модуля (Code). Но такое переключение возможно и с помощью пиктограмм, описанных выше. Репозиторий содержит готовые проекты (шаблоны), которые можно включить в проект или сделать основой проекта. Поскольку при создании проекта формируется множество файлов, то можно посмотреть содержимое каждого в отдельности. Пиктограмма с изображением дискеты позволяет наоборот сохранить сразу во всех файлах проекта изменения (сохранить все). Нужно только помнить, что в первый раз нужно сохранять именно проект (пункт меню Save Project As), причем хранить проект лучше в собственном каталоге (иначе потом его нужно искать).

Создание, компиляция, запуск приложения проекта.

Пиктограмма запуска проекта (зеленый треугольник), которая заставляет проект пройти компиляцию и запуститься на исполнение.

Для того чтобы проект начал работать, его нужно откомпилировать и запустить (зеленая кнопка). Если зеленая кнопка не работает, то это значит, что проект не сформирован (чего-то не хватает). После запуска проекта активизируется пиктограмма приостановки работы проекта (Pause). Она дает возможность прервать (временно) работу проекта.

Программа, представленная в виде инструкций языка программирования, называется исходной программой. Она состоит из инструкций, понятных человеку, но не понятных процессору компьютера. Чтобы процессор смог выполнить работу в соответствии с инструкциями исходной программы, исходная программа должна быть переведена на машинный язык — язык команд процессора. Задачу преобразования исходной программы в машинный код выполняет специальная программа — компилятор.

Компилятор выполняет последовательно две задачи:

1. Проверяет текст исходной программы на отсутствие синтаксических ошибок.

2. Создает (генерирует) исполняемую программу — машинный код.

Созданный проект нужно сохранить в виде набора файлов с раширением (dpr — головной файл проект, pas - модуль, dfm — файл описания формы, res — файл ресурсов). Пиктограмма с изображением дискеты позволяет наоборот сохранить сразу во всех файлах проекта изменения (сохранить все). Нужно только помнить, что в первый раз нужно сохранять именно проект (пункт меню Save Project As), причем хранить проект лучше в отдельном собственном каталоге (иначе потом его нужно искать).

Если при создании проекта выбрать пункт меню File/New/VCL Forms Application, то в результате будет создан стандартный VCL проект, который содержит 1 форму Form1 и один модуль Unit 1. Существуют и другие варианты проектов (напрмер консольный). Варианту можно выбрать в репозитории. Репозиторий содержит готовые проекты (шаблоны), которые можно включить в проект или сделать основой проекта.

Инспектор объектов и палитра компонентов.

Для настройки компонентов служит окно инспектора объектов. Оно имеет 2 вкладки — свойств (Properties) и событий (Events). Свойства в инспекторе объектов делятся на текстовые (которые нужно вводить как текст), числовые (которые тоже вводятся, но исключительно как натуральные числа), константные (значения которых выбираются из списка возможных констант, тогда рядом с окном ввода есть знак раскрывающегося списка), особо редактируемые (которые редактируются в специальном окне редактора свойства, у таких свойств рядом с окном ввода есть знак с несколькими точками). С помощью окна палитры компонент можно добавить на форму компонент. Компоненты располагаются на отдельных вкладках палитры.

Для настройки свойств и обработчиков событий можно использовать инспектор объектов. Рассмотрим пример настройки свойства и обработчиков событий компонентов проекта. Свойство Caption для Label, Button и Form нужно использовать для вывода текстов. Свойство text для Edit нужно очистить (это свойство используется для ввода или вывода

текста и лищний текст там не нужен). Для реакции компонентов на события нужно определить действие для процедур вызываемых при событии (например щелчке по кнопке). Эти процедуры принято называть обработчиками событий (щелчка по кнопке или buttonclick). Это событие можно найти в инспекторе объектов на вкладке Events. Для преобразования типов используются специальные функции преобразования типов floattotsr (преобразует действительное число в текст) и strtofloat (преобразует текст в действительное число), inttotsr (преобразует целое число в текст) и strtoint (преобразует текст в целое число). Процедура Form Create - метод формы, который запускается сразу после создания формы (обработчик события создания формы). Его используют для установки начальных параметров проекта. Процедура showmessage('Сообщение …') - используется для вывода сообщения с остановкой работы программы.

6.Язык программирования Object Pascal. Типы данных и операции языка Паскаль. Константы и переменные Паскаля. Построение выражений. Основные простые типы данных.

Типы и структуры данных

Суть всех алгоритмов (и компьютерных программ) состоит в том, что они описывают преобразование некоторых начальных данных в конечные. Какие-то данные алгоритм (программа) может использовать как промежуточные. Естественно, что представление и организация данных имеет при разработке алгоритма первостепенное значение.

Решая конкретную задачу, необходимо выбрать множество данных, представляющих реальную ситуацию. Затем надлежит выбрать способ представления этой информации. Представление данных определяется исходя из средств и возможностей, допускаемых компьютером и его программным обеспечением. Однако очень важную роль играют и свойства самих данных, операции, которые должны выполняться над ними. С развитием вычислительной техники и программирования средства и возможности представления данных получили большое развитие и теперь позволяют использовать как простейшие неструктурированные данные, так и данные более сложных типов, полученные с помощью комбинации простейших данных. Такие данные называют структурированными, поскольку они обладают некоторой организацией. Современные средства программирования позволяют оперировать с множествами, массивами, записями, файлами, динамическими структурами (стеки, очереди, списки, деревья).

К основным типам данных языка Object Pascal в Delphi относятся: целые числа, действительные числа, символы (char), строки (string); логический тип (boolean). Тип переменной описывается в области Var проекта. Пользователь так же может создать свой тип данных и описать его в области Type.

Целые числа и числа с плавающей точкой (действительные) могут быть представлены в различных форматах. Целые числа могут быть без знака (строго положительные или равны нулю) или диапазон объединяет и положительные и отрицательные целые числа. Действительные числа могут быть с фиксированной запятой (например денежный формат — доллары и дробная часть центы, практически не используется в Паскале) или представляют из себя числа образованные из мантиссы и порядка $X=M*2^p$, где $|M|<1$ — мантисса, p — порядок. Порядок может быть и положительным и отрицательным целым числом. Чем больше диапазон порядка, тем больше диапазон значений числа. Наибольший по модулю отрицательный порядок определяет «машинный ноль» (все меньшие по модулю числа считаются или округляются до нуля). Наибольший положительный порядок определяет «бесконечность» (все большие по модулю числа считаются бесконечно большими и приводят к ошибке переполнения). Диапазон битов для мантиссы определяет точность числа, так как действительное число обязательно округляется и имеет конечное число двоичных цифр, при переводе в десятичную систему это приводит к тому, что число хранимых десятичных цифр не однозначно. Для чисел, начинающихся с 1,2,3,4 диапазон больше, для чисел, начинающихся с 6,7,8,9 диапазон меньше.

Типы данных. Целые числа

Формат	**Диапазон**
Shortint	-128..127
Integer	-32768..32767
Longint	-2 147483648..2147483647
Byte	0..255
Word	0..65535

Две точки подряд в Паскале обычно обозначает диапазон.

Типы данных. Действительные числа (с плавающей точкой)

Формат	**Диапазон**	**Кол-во значащих цифр**
Real	$2{,}9*10^{-39}$.. $1{,}7*10^{38}$	11-12
Single	$1{,}5*10^{-45}$.. $3{,}4*10^{38}$	7-8
Double	$5{,}0*10^{-324}$.. $1{,}7*10^{308}$	15-16
Extended	$3{,}4*10^{-4932}$.. $1{,}1*10^{4932}$	19-20

Строки Object Pascal — множества элементов типа char и могут быть 2-х видов. Первый вид традиционный для Паскаля — фактически является динамическим массивом ограниченной длины (255 знаков). Второй вид — строки с завершающим нулем традиционны для языка С. Особенность этой строки то, что она не определяет длину строки и может быть очень длинной, конец строки определяется завершающим знаком кода ноль.

Объявление переменной-строки длиной 255 символов в области Var : Имя:string;

•Объявление переменной-строки указанной длины: Имя:string [ДлинаСтроки].

7.Операторы языка Object Pascal. Операторы ветвления. Операторы ветвления if и case. Составной оператор.

Основные конструкции языка Паскаль используют константы, переменные, выражения и операторы. Переменные имеют собственные имена (идентификаторы) и описываются в области Var программы. Константы могут иметь собственные имена (идентификаторы) и описываться в области Const программы, однако чаще используются константы без имен — числа, знаки, строки, логические значения и т. д. Разница между переменными и константами в возможности изменения своего значения в процессе работы программы. Выражения в Паскале строятся из переменных, констант, функций с помощью операций подобно математическим выражениям). Особое значение имеют логические выражения, использующие логические отношения, операции, константы (true-истина, false-ложь). Для изменения порядка выполнения операций могут использоваться круглые скобки. Квадратные и фигурные скобки имеют в Паскале особое значение и в выражениях не используются. Операторы Паскаля — отдельные специальные команды, разделяемые знаком «точка с запятой». Операторы используют специальные идентификаторы, которые нельзя использовать для других целей.

Константы без имени — числа (23, -23.45), знаки '/' или 'п' кодовой таблицы ЭВМ, тексты или строки 'пример строки', true или false.

Пример выражения: (x+2)/(7*x+f)

Операции

1.Математические +, -, *, /, div, mod

2.Логические and, or, not, xor

3.Специальные in, is, ^, @, +, *

Отношения

<- - меньше. > - больше, <= - меньше или равно, >= - больше или равно, <> - не равно.

'g' > 'H' 'a' < 'м' ord('k') – код знака 'k', chr(78) – знак кода 78, #78=chr(78)

Примеры :

```
 if x>5 then
  begin
    y:=3;
    z:=y*y;
   end
   else z:=0;
   case ocenka of
     2: S:='двоечник';
     3: S:='троечник';
     4: S:='хорошист';
     5: S:='отличник'
 else
   S:='странная оценка'
end;
```

Составной оператор - это последовательность произвольных операторов программы, заключенная в операторные скобки - зарезервированные слова **begin ... end** . Составные операторы - важный инструмент Object Pascal, дающий возможность писать программы по современной технологии структурного программирования (без операторов перехода goto). Object Pascal не накладывает никаких ограничений на характер операторов, входящих в составной оператор. Среди них могут быть и другие составные операторы — язык Object Pascal допускает произвольную глубину их вложенности.

Операторы ввода/вывода на экран (только для старых версий Паскаля и консольного приложения Delphi):

write(список вывода) или writeln(список вывода) — оператор вывода на экран

read(список ввода) или readln(список ввода) — оператор ввода с клавиатуры

Операторы выбора

Оператор простого выбора if

Вариант 1. Полный вариант оператора if-then-else.

```
if Условие then
begin { Инструкции, которые выполняются, если условие истинно. } end
 else  begin { Инструкции, которые выполняются,  если условие ложно } end ;
```

В этом операторе предусмотрено реагирование на 2 альтернативы.

Вариант 2. Неполный вариант оператора if-then.

```
if Условие then
begin { Инструкции, которые выполняются, если условие истинно. } end;
```

Примечание

Если между begin и end находится только одна инструкция, то слова begin и end можно не писать.

В данном операторе показан комментарий (текст, который игнорируется компилятором языка Паскаль). Он размещается в любом месте программы внутри фигурных скобок)

Оператор множественного выбора case позволяет провести анализ значения некоторого выражения и в зависимости от его значения выполнять те или иные действия. Для этого используется выражение-селектор. Вычислив это выражение Паскаль сравнивает полученное значение с вариантами значений (представленными в виде 1 значения, диапазона, списка значений). Если значение соответствует варианту, то выполняется оператор или блок операторов, заданных для данного списка. Кроме того есть вариант, который выполняется при не совпадении значения со всеми списками, он указывается в конце после else. Нужно помнить, что оператор проверяет списки последовательно и если он нашел значение в списке выше, то нижележащие списки проверяться уже не будут. В общем случае формат записи оператора case следующий:

case <выражение> of
<список значений 1>: <оператор 1>;
...
<список значений n>: <оператор n>
else
<оператор>
end;

Инструкция case может так же использоваться в 2-х вариантах.

Варианты оператора множественного выбора Case

Вариант 1: Без else.

case Выражение **of**

Список1_Констант: **begin** { инструкции } end;

Список2_Констант: **begin** { инструкции } end;

...

СписокJ_Констант: **begin** { инструкции } end; **end;**

Вариант 2. Полный вариант.

case Выражение **of**

Список1_Констант: **begin** { инструкции } end;

Список2_Констант: **begin** { инструкции } end;

...

СписокJ_Констант: **begin** { инструкции J} **end**

else begin { инструкции } end; **end;**

Инструкции между begin и end выполняются, если значение выражения, записанного после case, совпадает с константой из соответствующего списка. Если это не так, то выполняются инструкции, находящиеся после else, между begin и end. Общее правило всех операторов — перед else точки с запятой не бывает.

8.Операторы языка Object Pascal. Циклы и их организация. Циклы с известным числом повторений. Циклы с не известным числом повторений.

Циклы

Как уже говорилось, алгоритмы могут использовать различные варианты циклов. **В Паскале есть несколько операторов для реализации циклов. Это циклы с известным числом повторений (счетчиком) и циклы с неизвестным числом повторений (с предусловием и постусловием).**

Инструкция for циклы с известным числом повторений (счетчиком)

Вариант 1 (с увеличением счетчика):

for Счетчик:=НачальноеЗначение to КонечноеЗначение do begin { операторы } end;

Операторы между begin и end выполняются (КонечноеЗначение - НачальноеЗначение) + 1 раз.

ЕСЛИ НачальноеЗначение > КонечноеЗначение, то операторы между begin и end не выполняются.

Вариант 2 (с уменьшением счетчика)'.

for Счетчик:=НачальноеЗначение downto КонечноеЗначение do begin { операторы } end;

Операторы между begin и end выполняется (НачальноеЗначение - КонечноеЗначение) + 1 раз.

Если НачальноеЗначение < КонечноеЗначение, то операторы между begin и end не выполняются.

Инструкция repeat. Цикл с постусловием.

Repeat { инструкции } **until** Условие;

Сначала выполняются инструкции цикла, которые расположены между repeat и until. Затем вычисляется значение выражения Условие, и если оно равно False, то инструкции цикла выполняются еще раз. И так до тех пор, пока значение выражения Условие не станет равным True.

Инструкция while. Цикл с предусловием.

while Условие **do begin** { инструкции) end;

Сначала проверяется Условие, если оно истинно, то выполняются инструкции между begin и end. Затем снова проверяется Условие. Если оно выполняется, то инструкции цикла выполняются еще раз. И так до тех пор, пока Условие не станет ложным.

Оператор for обеспечивает циклическое повторение некоторого оператора - тела цикла заданное число раз. Повторение цикла определяется некоторой управляющей переменной (счетчиком), которая увеличивается или уменьшается на единицу при каждом выполнении тела цикла. Повторение завершается, когда управляющая переменная достигает заданного значения. Это циклы с известным числом повторения. Оператор for записывается в одной из следующих форм:

for <счетчик>:=<начальное значение> to <конечное значение>
do <оператор>;

или

for <счетчик>:=<начальное значение> downto <конечное значение>
do <оператор>;

<счетчик> - локальная управляющая переменная порядкового типа. В начале выполнения оператора for ей присваивается <начальное значение>. После каждого очередного выполнения тела цикла <оператор> ее значение увеличивается (в первой форме с to) или уменьшается (во

второй форме с downto) на единицу. Когда значение управляющей переменной достигает значения <конечное значение>, тело цикла выполняется последний раз, после чего управление передается оператору, следующему за структурой for. <начальное значение> и <конечное значение> являются выражениями, совместимыми по типу с управляющей переменной.

Если заданные начальное и конечное значения равны друг другу, тело цикла выполняется только один раз. Если в форме с to начальное значение больше конечного или в форме с downto начальное значение меньше конечного, то тело цикла не выполняется ни разу.Внутри цикла значение управляющей переменной может использоваться в выражениях. Однако, после окончания выполнения структуры for значение управляющей переменной не определено.

Циклы с неизвестным числом повторений

Структура repeat...until используется для организации циклического выполнения совокупности операторов, называемой телом цикла, до тех пор, пока не выполнится некоторое условие. Синтаксис управляющей структуры repeat...until:

repeat
<операторы тела цикла>
until <выражение условия>;

Точка с запятой после последнего оператора тела цикла (перед ключевым словом until) может опускаться. Структура работает следующим образом. Выполняются операторы тела цикла. Затем вычисляется <выражение условия>, которое должно возвращать результат булева типа. Если выражение возвращает false, то повторяется выполнение операторов тела цикла и после этого снова вычисляется выражение. Такое циклическое повторение цикла продолжается до тех пор, пока проверяемое выражение не вернет true. После этого цикл завершается и управление передается оператору, следующему за структурой repeat...until. Это цикл в постусловием.Поскольку проверка выражения осуществляется после выполнения операторов тела цикла, то эти операторы заведомо будут выполнены хотя бы один раз, даже если выражение сразу истинно.

Структура while...do используется для организации циклического выполнения оператора, называемого телом цикла, пока выполняется некоторое условие. Синтаксис управляющей структуры while...do: while <выражение условия> do <оператор>;

Структура работает следующим образом. Сначала вычисляется <выражение условия>, которое должно возвращать результат булева типа. Если выражение возвращает true, то выполняется оператор тела цикла, после чего опять вычисляется выражение, определяющее условие. Такое циклическое повторение выполнения оператора и проверки условия продолжается до тех пор, пока условие не вернет false. После этого цикл завершается и управление передается оператору, следующему за структурой while...do. Это цикл с предусловием.

Поскольку проверка выражения осуществляется перед выполнением оператора тела цикла, то, если условие сразу ложно, оператор не будет выполнен ни одного раза. В этом основное отличие структуры while...do от структуры repeat...until, в котором тело цикла заведомо выполняется хотя бы один раз.

9.Язык программирования Object Pascal. Массивы и работа с ними. Массивы и их использование в программах Паскаля.

Типы и структуры данных

Суть всех алгоритмов (и компьютерных программ) состоит в том, что они описывают преобразование некоторых начальных данных в конечные. Какие-то данные алгоритм (программа) может использовать как промежуточные. Естественно, что представление и организация данных имеет при разработке алгоритма первостепенное значение.

Решая конкретную задачу, необходимо выбрать множество данных, представляющих реальную ситуацию. Затем надлежит выбрать способ представления этой информации. Представление данных определяется исходя из средств и возможностей, допускаемых компьютером и его программным обеспечением. Однако очень важную роль играют и свойства самих данных, операции, которые должны выполняться над ними. С развитием вычислительной техники и программирования средства и возможности представления

данных получили большое развитие и теперь позволяют использовать как простейшие неструктурированные данные, так и данные более сложных типов, полученные с помощью комбинации простейших данных. Такие данные называют структурированными, поскольку они обладают некоторой организацией. Современные средства программирования позволяют оперировать с множествами, массивами, записями, файлами, динамическими структурами (стеки, очереди, списки, деревья).

Массивы

Объявление одномерного массива:

•ИмяМассива: **array** [НижнийИндекс .. ВерхнийИндекс] of ТипЭлементов;

•Пример : Var m:array [2..25] of integer;

•Объявление двумерного массива:

•ИмяМассива: **array** [НижнийИндекс1 .. ВерхнийИндекс1,

НижнийИндекс2 .. ВерхнийИкдекс2] of ТипЭлементов;

•Пример : Var mк:array [1..5, 2..6] of real;

Основные алгоритмы работы с массивами — инициализация, поиск и сортировка элементов.
Инициализация массива — заполнение массива данными или по другому, присвоение элементам массива конкретных значений. Поскольку после создания массива значения его элементов не определены, то в принципе они могут быть любыми, чтобы это не влияло на работу программы массивы часто инициализируют, например заполняя нулями или другими исходными данными.

Алгоритмы поиска и сортировки

Алгоритм поиск информации заключается в выделении в общем объеме информации элементов, удовлетворяющих некоторому критерию поиска. Поиск может быть однократным, когда находится первый подходящий элемент или множественным, когда находятся все подходящие элементы. Поиск может быть простым (последовательным) или сложным. Последовательный поиск предполагает последовательный просмотр и проверку всей информации. Последовательный поиск достаточно прост, но время его работы прямо пропорционально количеству данных, которые нужно просмотреть; удвоение количества элементов приведет к удвоению времени на поиск, если искомого элемента в массиве нет. Это линейное соотношение (время выполнения является линейной функцией от размера данных), поэтому такой метод также называется линейным поиском. Очевидно, что если объем информации (массив) не упорядочен, то возможен только линейный поиск. Наличие структуры позволяет ускорить поиск с помощью различных алгоритмов сложного поиска.

10.Задачи обработки массивов. Поиск и выборка данных. Сортировка данных. Простые и. быстрые алгоритмы.

Алгоритмы поиска и сортировки

Алгоритм поиск информации заключается в выделении в общем объеме информации элементов, удовлетворяющих некоторому критерию поиска. Поиск может быть однократным, когда находится первый подходящий элемент или множественным, когда находятся все подходящие элементы. Поиск может быть простым (последовательным) или сложным. Последовательный поиск предполагает последовательный просмотр и проверку всей информации. Последовательный поиск достаточно прост, но время его работы прямо пропорционально количеству данных, которые нужно просмотреть; удвоение количества элементов приведет к удвоению времени на поиск, если искомого элемента в массиве нет. Это линейное соотношение (время выполнения является линейной функцией от размера данных), поэтому такой метод также называется **линейным поиском**. Очевидно, что **если** объем информации (**массив) не упорядочен, то возможен только линейный поиск**. Наличие структуры позволяет ускорить поиск с помощью различных алгоритмов сложного поиска.

Сложность алгоритма, оценка сложности алгоритма

Известно, что правильность — далеко не единственное качество, которым должна обладать хорошая программа. Одним из важнейших является эффективность. Эффективность – сравнительная характеристика алгоритма и определяется неким критерием эффективности. Это может быть время выполнения алгоритма, число операций процессора, размер памяти используемой программы и т.д.. Этот критерий определяется как зависимость или функция от различных входных данных (параметра N).

В тоже время можно говорить о эффективности алгоритма и сложности задачи. Можно сравнивать разные алгоритмы для одной задачи, а можно для разных задач. Поэтому принято говорить о сложности или скорости алгоритма и о сложности задачи. Так как эти величины есть функции, то можно их сравнивать по асимптотическим оценкам при стремлении параметра $N->\infty$.

Нахождение точной зависимости критерия эффективности для конкретной программы — задача достаточно сложная. По этой причине обычно ограничиваются **асимптотическими оценками** этой функции, то есть описанием ее примерного поведения при больших значениях параметра N. При этом для асимптотических оценок используют традиционное отношение O (читается "O большое") между двумя функциями f(n)=O(n).

Асимптотические оценки функции при $N->\infty$ можно представить выражением O(f(N)), которое означает const* f(N). Пусть заданы 2 оценки O(f(N)) и O(g(N)). Будем считать, что:

Функция f(N) растет медленнее g(N), если $\lim_{N->\infty} [O(f(N)) / O(g(N))] = 0$.

Функция f(N) растет быстрее g(N), если $\lim_{N->\infty} [O(g(N)) / O(f(N))] = 0$.

Функции f(N) и g(N) имеют одинаковую скорость роста, если $\lim_{N->\infty} [O(f(N)) / O(g(N))] = const \neq 0$.

Аналогично по асимптотике роста критерия эффективности можно говорить о скорости роста алгоритма. Причем, чем больше скорость роста алгоритма, тем он фактически медленнее и наоборот. Соответственно по функции асимптотики можно различать линейные O(N), квадратичные $O(N^2)$, кубические $O(N^3)$ экспоненциальные $O(e^N)$ и др. алгоритмы.

Сложность задачи можно оценить по скорости наиболее эффективного для нее алгоритма. При этом, может оказаться, что наиболее эффективный алгоритм нам пока не известен. Поэтому реальная сложность задачи может оказаться ниже чем представляемая нами, но не может оказаться больше чем текущая.

В качестве примера рассмотрим алгоритм нахождения факториала числа. Легко видеть, что количество операций, которые должны быть выполнены для нахождения факториала N! числа N в первом приближении прямо пропорционально этому числу, ибо количество повторений цикла (итераций) при вычислении по формуле N!=(N-1)!*N программе равно N. В подобной ситуации принято говорить, что алгоритм имеет ***линейную сложность*** (сложность O(N)).

Можно ли вычислить факториал быстрее? Оказывается, да. Можно написать такую программу, которая будет давать правильный результат для тех же значений N с логарифмической скоростью. Про алгоритмы, в которых количество операций примерно пропорционально log(N) (в информатике обычно используют для основания двоичный логарифм) говорят, что они имеет ***логарифмическую сложность*** (O(log(N)).

Сортировка информации имеет смысл, если информация структурирована. Тогда сортировка предполагает изменение структуры так, чтобы она располагалась по направлению возрастания\убывания некоторого числового критерия сортировки. Соответственно можно говорить о сортировке линейных структур (массивы, таблицы, линейные списки), сортировки деревьев, сортировки различных сетей и т.д.. В простейшем случае сортировки линейной структуры алгоритмы принято делить на простые и быстрые алгоритмы. Простые (медленные) алгоритмы имеют квадратичную по числу элементов скорость и называются квадратичными. К ним относят методы «пузырька» или обменной сортировки, выбора, вставки, дополнительного массива и т.д..

Простые варианты сортировок.

Метод обменной сортировки предполагает по парное сравнение рядом расположенных элементов с последующим обменом местами, если окажется, что элементы расположены «не правильно». Очевидно, что для сортировки по возрастанию первый элемент должен иметь меньшее значение критерия чем следующий, если это не так, то элементы переставляют. Если эту проверку включить в цикл, то можно проверить все пары элементов и переставить при необходимости. К сожалению, за один цикл сортировка может не получиться, поэтому цикл повторяют до тех пор, пока есть перестановки, когда перестановок в цикле нет, то это и значит, что массив уже отсортирован:

```
repeat
k:=0;
For i := 2 to N do
If A[i-1] > A[i] then  Begin  Am:= A[i-1]; k := k+1; A[i-1] := A[i]; A[i] = Am; end;
until k=0;
```

Здесь 2 цикла и число операций уже пропорционально квадрату N.

Метод выбора использует алгоритм поиска max\min. Например, для сортировки по возрастанию используем поиск минимального элемента. Найдем min элемент и поменяем его местами с первым, затем повторим это действие с частью массива без 1-го элемента и т.д.. Когда останется 1 элемент, прекратим вычисления – массив отсортирован:

```
For k:=1 to N do begin
Amin := A[k]; p:=k; {Сортировка по возрастанию}
For i := k+1 to N do
If A[i] < Amin then  Begin  Amin := A[i]; p := i  end;
A[p]:=A[k];  A[k]:=Amin;
end;
```

Особенность алгоритмов сортировки в том, что они являются шаблонами для целого класса алгоритмов. Если в условиях If алгоритмов поменять только условие, то смысл задачи сильно изменится. Например, при замене < на > получается сортировка по убыванию.

Алгоритм вставки определяется делением исходного массива на две логические части. Первая часть массива «отсортированная», вторая – «остаток». Последовательно элементы из остатка вставляют в отсортированную часть в нужное место, «раздвигая» его элементы. В самом начале первая часть содержит только один – первый элемент. Затем добавляют второй элемент (справа или слева от первого).

Алгоритм дополнительного массива использует подсчет номера места для каждого элемента в отсортированном массиве. Пусть из 10 элементов исходного массива 6 элементов больше данного, а 3 меньше. Тогда при сортировке по возрастанию элемент будет 4-м, а по убыванию 7-м в отсортированном массиве. Таким образом, мы можем параллельно строить этот отсортированный массив в качестве дополнительного.

Алгоритмы поиска.

Приведем пример линейного поиска максимального и минимального элементов массива:

```
Amax := A[1]; k:=1; {Поиск максимального элемента}
Amin := A[1]; p:=1; {Поиск минимального элемента}
For i := 2 to N do   begin
If A[i] > Amax then  Begin  Amax := A[i]; k := i end;
If A[i] < Amin then  Begin  Amin := A[i]; p := i  end;
end;
```

Как видно из примера, поиск максимального отличается от поиска минимального только знаком отношения (больше или меньше) в условии if. Аналогичный вид будет иметь программа и для любого линейного поиска, разница будет только в выражении в условии if, которое определяется критерием поиска.

В случае сортировки массива, таблицы реляционной базы данных для поиска используют «быстрые» (или логарифмические) алгоритмы. Рассмотрим в качестве примера алгоритм бинарного поиска в отсортированном по условию поиска массиве:

- Выберем элемент лежащий максимально близко к середине массива. Проверим его по критерию поиска, очевидно, что результат проверки даст возможность отбросить половину массива, как заведомо неподходящую (если критерий элемента больше нужного, то все элементы с большим значением критерия отбрасываются, аналогично для противоположного случая).
- Отбросим ненужную часть массива и в оставшейся части вновь найдем средний элемент.
- П.2 повторяем до тех пор, пока не останутся только те элементы массива, которые удовлетворяют критерию поиска.

Скорость данного алгоритма пропорциональна $\log_2 N$, где N – количество элементов массива. Для больших значений N данный вариант сортировки гораздо выгоднее линейной.

Быстрая сортировка

Быстрая сортировка имеет более сложные алгоритмы, но при этом скорость порядка $N^* \log_2 N$. Примерами быстрой сортировки являются бинарный метод и метод Хоара. Бинарный метод основан на простом принципе – деления массива пополам. Если сложность сортировки массива N^2 , то для половины массива $N^2/4$. Вместе 2 половинки дают $2*N^2/4= N^2/2$, для объединения массива потребуется еще N операций. Всего $N^2/2+N < N^2$. При больших N такой простой ход дает уменьшение числа операций почти вдвое. Однако можно усовершенствовать этот подход и вновь разделить каждый из 2-х массивов пополам, затем вновь пополам и т.д.. В пределе массив уменьшается до 2 или 3 элементов. В результате фактически строится бинарное дерево и поиск ведется по ветвям этого дерева. Этот алгоритм и называется быстрой бинарной сортировкой.

Сортировка Хоара – улучшение сортировки вставкой. Хоар делит массив примерно на 2 равных части, выбирает элемент в середине и меняет местами элементы, так чтобы с одной стороны были большие элементы, а с другой меньшие. Теперь место среднего элемента найдено и можно сортировать каждую часть в отдельности. Как и в бинарной сортировке получаем выгоду. Особенно выигрышен алгоритм Хоара в рекурсивной форме. Скорость роста быстрой сортировки больше чем у линейных алгоритмов, но меньше чем у квадратичных. Для таких «не кратных» степеням скоростей роста принято общее название – полиномиальные скорости роста (т.е. ограниченные полиномом).

11.Работа со строковыми переменными. Строковый и знаковый типы данных Паскаля. Процедуры и функции для работы со строками.

Строки Object Pascal — множества элементов типа char и могут быть 2-х видов. Первый вид традиционный для Паскаля — фактически является динамическим массивом ограниченной длины (255 знаков). Второй вид — строки с завершающим нулем традиционны для языка С. Особенность этой строки то, что она не определяет длину строки и может быть очень длинной, конец строки определяется завершающим знаком кода ноль.

Строки

•Объявление переменной-строки длиной 255 символов:

Имя:string;

•Объявление переменной-строки указанной длины:

Имя:string [ДлинаСтроки].

Строковые процедуры и функции

Рассмотри только строки в стиле Паскаля. **Для них есть только одна операция +. Это соединение (сцепление) строк**. Остальное нужно делать с помощью стандартных подпрограмм.

Задачи преобразования текстовой информации включают в себя прежде всего задачи поиска,

выделения и композиции текстовой информации. Текст как правило делится на строки и представляется массивом строк. **Обработка строк связана с использованием специальных функций. В Паскале это функции Pos – поиск подстроки, Copy – выделение подстроки, Length – длина строки, Delete – удаление подстроки.** Пользователь может создать свои процедуры обработки, однако, на практике использования стандартных процедур вполне достаточно. Особое значение для работы с текстами играют специализированные классы списков (VCL Delphi) – TLisp, TStrings, TStringList и т.д.. Данные классы имеют специальные методы поиска, сортировки строк.

Замечание: Следует помнить, что в программировании используются строки 2-х типов. В первом варианте (строки типа Паскаля) строка имеет вид динамического массива целых чисел с кодами букв строки. При этом первый элемент массива с индексом 0 содержит число – длину строки. Во втором варианте (строки с завершающим 0 в стиле языка Си) строка так же динамический массив чисел с кодами букв строки, но при этом длина строки не задается, а просто в последнем элементе указывается код равный нулю. В первом случае строка имеет ограниченную длину (ограничена 255 для обычной кодировки или 65535 знаков), но пользоваться такой строкой легче. Во втором случае строка может иметь миллиарды знаков, но приходится пользоваться более сложными алгоритмами для работы с указателями.

При создании интерфейса часто приходится преобразовывать числа в текст и обратно. Для этого есть много специальных функций (например strtoint, strtofloat, inttostr и т.д.).

Процедуры и функции обработки строк в стиле Pascal (без нулевого символа в конце)

CompareStr(const S1, S2: string): Integer	**Сравнивает две строки S1 и S2 с учетом регистра. Возвращает значение < 0, если S1 < S2, 0, если S1 = S2, и > 0, если S1 > S2.**
CompareText(const S1, S2: string): Integer	**Сравнивает две строки S1 и S2 без учета регистра. Возвращает значение < 0, если S1 < S2, 0, если S1 = S2, и > 0, если S1 > S2.**
Concat(s1 [, s2,..., sn]: string): string	**Возвращает строку, склеенную из строк s1, ..., sn. Идентична операции "+" для строк.**
Copy(S: string; Index, Count: Integer): string	**Возвращает подстроку строки S, начинающуюся с S[Index] и содержащую до Count символов.**
Delete(var S: string;Index,Count: Integer)	**Удаляет из S подстроку, начинающуюся с S[Index] и содержащую до Count символов.**
FloatToStrF(Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): string	**Преобразует Value в строку, используя формат Format с точностью Precision и числом цифр Digits. Возможные значения Format: ffGeneral - формат g, ffExponent - формат e, ffFixed - формат f, ffNumber - формат n, ffCurrency - формат m.**
FloatToStr(Value: Extended):string	**Преобразует Value в строку 15 ц.**
FormatFloat(const Format:string; Value: Extended):string	**Преобразует Value в строку, используя строку форматирования Format**
Insert(Source: string; var S: string; Index: Integer)	**Вставляет строку Source в S, начиная с S[index]**
IntToStr(Value: Integer): string	**Возвращает строку, содержащую преобразованное целое значение Value**
Length(S: string): Integer	**Возвращает число символов в S**
Pos(Sb: string; S: string): Integer	**Возвращает позицию (индекс) первого вхождения Sb в S. Если Sb нет в S, возвращается 0**
Str(X [:Width [: Decimals]]; var S)	**Преобразует целое или действительное значение X в строку S. Не обязательные**

	параметры: Width - ширина поля, Decimals - число цифр
StrToFloat(const S: string):Extended	**Преобразует строку S в действительное число**
StrToInt(const S: string): Integer	**Преобразует строку S в целое число**
StrToIntDef(const S: string; Default: Integer): Integer	**Преобразует строку S в целое число. Если строка не является допустимым целым, возвращается значение по умолчанию Default**
Trim(const S: string): string	**Удаляет из строки S начальные и завершающие пробелы и управляющие символы**
TrimLeft(const S: string):string	**Удаляет из строки S начальные пробелы и управляющие символы**
TrimRight(const S: string): string	**Удаляет из строки S завершающие пробелы и управляющие символы**
Val(S; var V; var Code: Integer)	**Преобразует строку S в целое число Code**

12. Стандартные функции и процедуры Паскаля. Подпрограммы пользователя в Паскале. Процедуры и функции Delphi.

Процедуры и функции Delphi

Подпрограмма – часть программы, выделенная как особая, имеющая собственный заголовок и параметры. Подпрограммы в Паскале делятся на процедуры и функции. Описание подпрограммы состоит из заголовка и тела подпрограммы.

Заголовок процедуры имеет вид: **PROCEDURE** <имя> [(<сп.ф.п.>)] ;

Заголовок функции: **FUNCTION** <имя> [(<сп.ф.п.>)] : <тип>;

Здесь <имя> - имя подпрограммы, <сп.ф.п.> - список формальных параметров; <тип> - тип возвращаемого функцией результата. Функция в Паскале отличается от процедуры тем, что она должна возвращать результат — значение функции, поэтому в конце заголовка функции указывается (через двоеточие) тип значения функции. Процедура не должна возвращать результат.

У подпрограмм есть параметры, которые делят на параметры-переменные и параметры-значения. Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например: **Procedure** SB(a: Real; b: Integer; c: Char); Как видно из примера, параметры в списке отделяются друг от друга точками с запятой.

В заголовке подпрограммы параметры-переменные записываются после Var. Например:

procedure ddd(x,y:real; var z:real); - здесь x,y — параметры-значения, а z- параметр-переменная. Параметр-переменная может изменять свое значение внутри подпрограммы, а параметр-значение нет. Поэтому параметр-переменную можно использовать для передачи результата из процедуры. Параметр-значение — формальный, т. е. он действует только внутри подпрограммы как переменная, а вне ее это фиксированное значение (константа). Локальными являются и переменные, которые описываются внутри попрограммы, вне ее они не действует. Глобальными переменными называют те, которые описаны в области Var модуля Unit, они действуют и могут меняться везде, и в подпрограмме, и вне ее.

Стандартные процедуры и функции Delphi

Стандартные процедуры и функции (модуля System)

Модуль SYSTEM автоматически связывается с любой программой, поэтому объявленные в его интерфейсной части типы, константы, переменные и подпрограммы доступны программисту в любой момент. Принято эти подпрограммы называть стандартными процедурами и функциями

паскаля. Их делят на несколько групп. Рассмотрим подпрограммы для работы с математическими вычислениями и для работы со строками.

Математические процедуры и функции

Для математических вычислений используются математические операции * / + - обычные, и особые div — целочисленное деление и mod — остаток после такого деления. Выражения могут использовать скобки. Дальнейшее усложнение формул возможно при использовании стандартных процедур и функций.

function Cos(X: Extended): Extended	Возвращает косинус аргумента X, заданного в радианах
function ArcTan(X: Extended):Extended;	Возвращает Арктангенс (в радианах) x
procedure Dec(var X [; N: LongInt]);	Уменьшает x на n, а если n опущено - на 1. x, n - любые порядковые типы, в том числе Int64
function Frac(X: Extended): Extended;	Возвращает дробную часть x
function Exp(X: Real): Real;	Возвращает x°, где e - основание натурального логарифма
procedure Inc(var X [; N: LongInt]);	Наращивает x на n, а если n отсутствует - на единицу
function Int(X: Extended): Extended;	Возвращает целую часть вещественной переменной
function Ln(X: Real): Real;	Возвращает натуральный логарифм x
function Odd(X: Longint): Boolean;	Возвращает True, если аргумент - нечетное число
function Pi: Extended;	Возвращает число it=3,141592653589793
procedure Randomize;	Инициирует генератор псевдослучайных последовательностей.
function Round(X: Extended):Int64;	Округляет вещественное число до ближайшего целого
function Random [(Range: Integer)] ;	Возвращает очередное псевдослучайное число.
function Sin(X: Extended): Extended;	Возвращает синус аргумента (в радианах)
function Sqr(X: Extended): Extended;	Возвращает квадрат аргумента
function Sqrt(X: Extended): Extended;	Возвращает корень квадратный из аргумента
function Trunc(X: Extended): Int64;	Преобразует вещественное число к целому путем отбрасывания дробной части

13.Стандартные классы и компоненты Delphi. Палитра компонентов.

Стандартные классы и компоненты Delphi

Все компоненты Delphi порождены от класса TComponent, в котором инкапсулированы самые общие свойства и методы компонентов. Предком TComponent является класс TPersistent, который произошел непосредственно от базового класса TObject. Класс TComponent служит базой для создания как видимых, так и невидимых компонентов. Большинство видимых компонентов происходит от класса TControl. Два наследника этого класса - TWinControl И TGraphicControoпределяют две группы компонентов: имеющие оконный ресурс rwincontroi и его потомки) и не имеющие этого ресурса TGraphicControl и его потомкам и). Оконный ресурс - это специальный ресурс Windows, предназначенный для создания и обслуживания окон. Только оконные компоненты способны получать и обрабатывать сообщения Windows. Оконный компонент в момент своего создания обращается к Windows с требованием выделения оконного ресурса и, если требование удовлетворено, получает так называемый дескриптор окна. TWinControl и его потомки хранят дескриптор окна в свойстве Handle.

Класс TControl со своими наследниками образуют всю палитру видимых компонентов Delphi. Терминологически они называются элементами управления, так как на их основе

прежде всего реализуются управляющие элементы Windows - кнопки, переключатели, списки и т. п. Как уже отмечалось, некоторые из наследников TControl обладают дескрипторами окон и способны получать и обрабатывать *Wwdows-сообщения,* другие окон не имеют, но обязательно включаются в состав оконных компонентов, которые управляют ими, согласуясь с требованиями (сообщениями) Windows. Оконные элементы управления обладают специальной оконной функцией, в которую Windows посылает управляющие сообщения (например, извещения о манипуляции пользователя с мышью или о нажатии клавиш клавиатуры). В терминологии Windows такие элементы называются родительскими, а связанные с ними неоконные элементы - дочерними. Замечу, что оконный компонент может выступать как родительский не только по отношению к неоконным компонентам, но и к оконным. В этом случае он просто транслирует управляющие сообщения Windows в оконные функции дочерних компонентов. Обязательным требованием Windows является визуальная синхронизация дочерних элементов: они не могут выходить из границ своего родителя и появляются и исчезают вместе с ним. Иными словами, родитель с дочерними элементами рассматривается Windows как единое целое. Класс TControl определяет свойство parent, которое содержит ссылку на родительский компонент: **property** Parent: TWinControl; Это свойство не следует путать с собственником owner: owner создал компонент (не обязательно - видимый), а parent управляет видимым компонентом. поскольку конструктор TComponent. Create не изменяет свойства parent (в родительском классе TComponent такого свойства нет), при создании видимых компонентов на этапе прогона программы это свойство необходимо изменять программно.

Положение и размеры компонента определяются четырьмя его свойствами (в пикселях):

property Height: Integer; // Высота

property Left: Integer;// Положение левой кромки

property Top: Integer;// *Положение верхней кромки*

property Width: Integer;// *Ширина*

Для всех компонентов, кроме форм, эти свойства задаются в координатах клиентской части родительского компонента. Для формы - в координатах экрана. Клиентская часть компонента - это внутренняя его область за исключением заголовка, рамки и меню. Свойства обычно определяются на стадии конструирования формы, но они доступны также и на этапе прогона программы. Изменение любого из них приводит к немедленному изменению положения или размера компонента как на этапе конструирования, так и при прогоне программы. Все четыре числовые величины содержатся также в единственном свойстве **property** BoundsRect: TRect;

Важную роль играет свойство Align, определяющее выравнивание положения компонента относительно границ своего родителя:

type TAlign = (aINone, alTop, alBottom, alLeft, alRight, alClient) ;

property Align: TAlign;

Если это свойство не равно aiNone, компонент прижимается к верхней (alTop), нижней (alBottom), левой (alLeft) или правой (alRight) границе своего родителя. При этом размеры компонента по соседним с границей измерениям игнорируются, и компонент “растекается” по границе. Например, если Align=alTop, значения свойств компонента Left и width игнорируются и его прямоугольник будет занимать всю верхнюю часть клиентской области родителя высотой Height пикселей; если Align=alLeft, свойства тор и Height игнорируются и прямоугольник занимает левую часть родителя шириной width пикселей и т. д. Если несколько компонентов имеют одинаковое выравнивание, они последовательно прижимаются друг к другу в порядке их перечисления в свойстве controls: первый прижимается к границе родителя, второй - к границе первого и т. д. Вся не заполненная другими компонентами клиентская область родителя заполняется компонентами со свойствами Align=alClient, которые в этом случае накладываются друг на друга. Замечательной особенностью свойства является его постоянство при изменении размеров клиентской части родителя. Если,

например, компонент прижат к верхней границе формы, он будет неизменно занимать верхнюю часть клиентской области при любых изменениях размеров окна. Таким способом можно легко реализовать панели с инструментальными кнопками, панели статуса и т. п. Временное отключение и затем включение эффекта от свойства Align обеспечивается методами

procedure DisableAlign;

procedure EnableAlign;

Любой видимый компонент можно спрятать или показать с помощью свойства visible или методами Hide и show:

property Visible: Boolean; // *True - показывает*

procedure Hide; // *Прячет компонент*

procedure Show; // *Показывает компонент*

Спрятанный компонент не реагирует на события от мыши или клавиатуры, он не участвует в дележе клиентской области родителя, ему нельзя передать фокус ввода клавишей *Tab.*

Если компонент частично или полностью перекрывается другими компонентами, его можно расположить над всеми компонентами и убрать обратно с помощью методов

procedure BringToFront; // *Сделать верхним*

procedure SendToBack; // *Сделать нижним*

Свойство

property Enabled: Boolean;

определяет возможность активизации компонента. Если оно имеет значение False, компонент запрещен для выбора. Такие компоненты (точнее, надписи на них) обычно отображаются серым цветом.

Некоторые компоненты имеют плоское изображение (например, метка TLabel), другие - всегда объемное (например, кнопка TButton).

Для остальных элементов объемность изображения регулируется свойством

property Ctl3D: Boolean;

С каждым управляющим компонентом связывается текстовая строка, которая становится доступна либо через свойство Caption, либо через свойство Text (альтернативой свойству Text, которое имеет тип string, является свойство TControi.windowsText типа pchar). Независимо от того, какое свойство хранит эту строку, ее можно установить и получить соответственно методами setTextBuf и *GetTextBuf,* при этом метод GetTextLen возвращает длину строки:

procedure SetTextBuf(Buffer: PChar);

function GetTextBuf(Buffer: PChar; BufSize: Integer): Integers;

function GetTextLen: Integer;

Если эта строка выводится в компоненте, используется шрифт, задаваемый свойством Font:

property Font: TFont;

В этом случае свойство

type TAlignment = (taLeftJustify, taRightJustify, taCenter);

property Alignment: TAlignment;

регулирует расположение текста относительно границ компонента:

taLeftJustify - прижать к левой границе; taRightJustify - прижать к правой границе; taCenter - расположить по центру.

Видимая часть элемента заливается цветом Color:

property Color: TColor; Обычно значение этого свойства выбирается из таблицы стандартных цветов Windows в виде константы clxxxx (перечень этих констант содержит раскрывающийся список свойства). В некоторых случаях может потребоваться залить компонент нестандартным цветом. В этом случае учтите, что свойство Color хранит четырехбайтное значение, каждый байт которого (слева направо, т. е. от старшего к младшему) имеет следующее назначение:

- 1 - указатель формата цвета (см. ниже);
- 2, 3, 4 - интенсивность соответственно синей, зеленой и красной

  составляющих.

Например, значение $00000000 определяет черный цвет, $00ff0000 -чистый синий цвет, $00ffffff - белый цвет и т. д.

Старший байт указывает, как используются остальные байты значения. Если он равен нулю, они определяют RGB-цвет так, как это описано выше. Если старший байт равен 1, три оставшихся байта определяют номер одной из 65536 возможных логических палитр (второй байт в этом случае игнорируется). Наконец, если старший байт равен 2, младшие определяют относительный цвет: в этом случае Windows отыскивает в текущей логической палитре ближайший к указанному цвет и использует его для заливки компонента. Другие значения старшего байта игнорируются (точнее, игнорируются старшие 5 разрядов этого байта; самый старший бит, если он равен 1, определяет черный цвет независимо от значений остальных 31 разрядов).

Компоненты Delphi

Основой проекта являются формы. Проект может не содержать форму (косольное приложение), иметь одну или несколько форм. Форма — окно, которое может содержать (контейнер) другие элементы-компоненты. Окно имеет стандартные свойства окон и может быть разных видов (Разновидности форм определяются значениями их свойств FormStyle, а также разнообразием форм-заготовок, хранящихся в репозитории Delphi. Самая первая подключенная к проекту форма (стандартное имя формы - Form1) становится главным окном программы. Окно этой формы автоматически появляется на экране в момент старта программы. С помощью меню среды в проект можно добавлять или удалять формы. Программист может указать любую форму, окно которой станет главным. Для этого нужно обратиться к опции меню Project | Options и, раскрыв список Main form, выбрать нужную форму.Каждое следующее окно становится видно только после обращения к его методу show или showModal. Чтобы обратиться к этим методам, нужно сослаться на объект-окно, который автоматически объявляется в интерфейсном разделе связанного с окном модуля. Для этого, в свою очередь, главное окно должно знать о существовании другого окна, что достигается ссылкой на модуль окна в предложении uses. Delphi автоматизирует вставку ссылки на модуль в предложение uses. Для этого на этапе конструирования нужно активизировать главное окно, щелкнув по нему мышью, после чего обратиться к опции File | uses unit. В появившемся диалоговом окне нужно выбрать модуль и нажать ок.

Компонентами в Delphi называются потомки класса TComponent. Компоненты располагаются в палитре компонентов на различных вкладках. Фактически это объекты специальных классов, которые можно включить (поместив на форму) в проект. Часть из этих объектов можно использовать для организации интерфейса.

КОМПОНЕНТЫ СТРАНИЦЫ STANDARD

- MainMenu - главное меню формы (программы)

- PopupMenu- вспомогательное (локальное) меню
- Label -метка для отображения текста
- Edit- ввод и отображение строки
- Memo- ввод и отображение текста (список строк)
- Button - кнопка
- CheckBox - независимый переключатель
- RadioButton -зависимые переключатели
- ListBox-список выбора
- ComboBox - раскрывающийся список выбора
- ScrollBar - управление значением величины
- GroupBox - панель группирования
- RadioGroup - группа зависимых переключателей
- Panel -панель

- Некоторые КОМПОНЕНТЫ СТРАНИЦЫ ADDITIONAL
  - BitBtn - кнопка с изображением
  - SpeedButton - кнопка для инструментальных панелей
  - MaskEdit - специальный редактор
  - Image - отображение картинок
  - Shape - стандартная фигура
  - StringGrid - компонент таблицы
- Некоторые КОМПОНЕНТЫ СТРАНИЦЫ WIN 32
  - ImageList - хранилище изображений
  - RichEdit - ввод и отображение RTF-текста
  - TrackBar- регулятор величины
  - ProgressBar - индикатор прогресса
  - UpDown - спаренная кнопка - счетчик(связывается с Edit)
  - DateTimePicker - ввод и отображение даты/времени
  - MonthCalendar - календарь
  - TreeView - дерево иерархии
- Некоторые КОМПОНЕНТЫ СТРАНИЦЫ SYSTEM
  - Timer- таймер
  - PaintBox - окно для рисования
  - MediaPlayer - медиаплейер
- КОМПОНЕНТЫ СТРАНИЦЫ DIALOGS
  - OpenDialog и SaveDialog - диалоги открытия и сохранения файлов
  - OpenPictureDialog и SavePictureDialog - диалоги открытия и сохранения изображений
  - FontDialog -диалог выбора шрифта
  - ColorDialog - диалог выбора цвета
  - PrintDialog - диалог настройки параметров печати
  - PrinterSetupDialog - диалог настройки параметров принтера
  - FindDialog- диалог поиска
  - ReplaceDialog - диалог поиска и замены

Компонент TWebBrowser используется для просмотра HTML и других документов.

Как правило все компоненты имеют свойства и методы, которые можно использовать и нужно настраивать. Есть свойства, которые присущи всем компонентам — например Name — имя компонента (лучше не менять!), Enabled — логическое свойство (включен или выключен компонент), Visible — логическое свойство (виден или не виден компонент). Набор свойств Top,Left, Height, Width и др. отвечают за местоположение и размеры компонента. Другие

свойства являются характерными именно для данного компонента. Среди них можно выделить главное или главные характерные свойства:
Button и Label- Caption, Edit – Text, Memo – список строк Lines, UpDown - свойство связи Associate, StringGrid — свойство Cells (массив клеток таблицы). Некоторые компоненты и свойства редактируются с помощью специальных средств — например компонент меню MainMenu или свойство списка строк Lines.
Компонент-редактор Memo — представляет из себя окно много строчного редактора, в котором можно загружать из файла, редактировать и сохранять в файле текст. Гдавное свойство список строк Lines. К любой строке списка можно обращаться как к элементу массива — например Memo1.Lines[5]:='новый текст'; Некоторые методы Lines : Clear — очистка, удаляются все строки, Add(S) — добавляется строка S в конец, Append — добавляется пустая строка, Delete — удаляется строка, LoadFromFile — загрузить текст из файла, SaveToFile — сохранить текст в файле.
Компоненты диалогов OpenDialog(открыть файл), SaveDialog(сохранить файл) имеют подобные свойства и методы. Свойство FileName – путь+имя файла (полное имя файла), метод и свойство Execute — запуск диалога.

14.Работа с деревьями и списками в Delphi. Компоненты работающие со списками.

Класс TStrings

Компоненты класса TMemo предназначены для ввода, редактирования и/или отображения достаточно длинного текста. Текст хранится в свойстве Lines класса TStrings и, таким образом, представляет собой пронумерованный набор строк (нумерация начинается с нуля). Такой набор элементов называют список. С помощью свойств и методов этого класса (Count, Add, Delete, Clear и т. д.) можно динамически формировать содержимое компонента.

Свойство Lines содержит пронумерованный список строк: первая строка в этом списке имеет индекс 0, вторая - 1, а общее количество строк можно узнать с помощью Lines .count.

Компонент класса TListBox представляет собой стандартный для Windows список выбора, с помощью которого пользователь может выбрать один или несколько элементов выбора. property Items: TStrings;Содержит набор строк, показываемых в компоненте property Itemlndex: Integer; Содержит индекс сфокусированного элемента. Если MultiSelect=False, совпадает с индексом выделенного элемента. property Sorted: Boolean; Разрешает/отменяет сортировку строк в алфавитном порядке. Компонент TRichEdit представляет собой многострочный редактор, работающий с расширенным текстовым форматом RTF. Текст формата RTF хранит дополнительную служебную информацию, управляющую свойствами каждого абзаца и сменой шрифта по ходу текста.

property Lines: TStrings; Содержит набор строк текста. С помощью его методов LoadFromFile и SaveToFile компонент может читать текст из файла или записывать в него текст

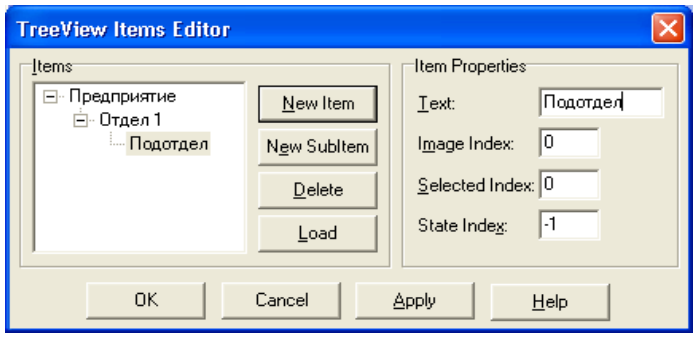
property Paragraph Содержит атрибуты текущего абзаца, т. е. абзаца с выделением или с текстовым курсором. Программа не может изменить свойство paragraph, но может изменить свойства связанного с ним абзаца

Методы класса TStrings

Имя метода	Описание метода
Add (public)	Добавить строку в список.
AddObject (public)	Добавить строку и связанный с ней объект в список
AddStrings (public)	Добавить группу строк в список
Append (public)	Добавить пустую строку в список.
CompareStrings (protected)	Сравнить 2 строки — не доступна.
Equals (public)	Сравнить 2 строки. Доступно для использования.
Exchange (public)	Поменять положение строки в списке.

GetText (public)	Выдать список строк как единый текст.
IndexOf (public)	Возвращает номер строки в списке.
Insert (public)	Добавить строку в нужное место списка.
LoadFromFile (public)	Загрузить список строк из текстового файла.
LoadFromStream (public)	Загрузить список строк из потока
Move (public)	Переместить строку в нужное место списка.
SaveToFile (public)	Сохранить список строк в текстовом файле.
SaveToStream (public)	Сохранить список строк в потоке

Дерево — особый вид списка, который представляется как «нелинейный» список, когда элементы списка могут образовывать разветвления. Компонент для работы с деревьями - TreeView - дерево иерархии, позволяет создавать иерархические деревья. Для создания и редактирования дерева используется специальный редактор. Внешний вид редактора:



Дерево создается как иерархическая структура, состоящая из элементов и подчиненным им подэлементов. На каждом уровне иерархии с элементом связана строка текста. Есть и другие компоненты, которые создают деревья различных объектов — рисунков, компонентов. Их принято называть коллекциями. Особенность использования таких компонентов — возможность программного редактирования коллекции (так же как и для дерева TreeView)

15.Компонент StringGrid. Использование StringGrid в проектах Delphi.

Компонент StringGrid

StringGrid - это компонент таблицы. Ее размеры определяются свойствами (см. Рис. 1) :

ColCount	3
DefaultColWidth	45
DefaultDrawing	True
DefaultRowHeight	15
FixedCols	1
FixedRows	1
GridLineWidth	1
Name	StringGrid1
Options	[goFixedVertLine,goFixe
RowCount	21
ScrollBars	ssBoth
Tag	0

Рисунок 1. Часть свойств StringGrid (вид в Object Inspector), отвечающих за размеры.

Здесь ColCount — число столбцов, RowCount — число строк, FixedCols и FixedRows — число выделенных столбцов и строк, GridLineWidth — толщина разделительных линий, DefaultColWidth и DefaultRowHeight — ширина столбцов и высота строк.

Пример — процедура заполняющая таблицу случайными числами

```
procedure TForm1.Button1Click(Sender: TObject);
var k:integer;
begin
                randomize;
```

```
for k := 1 to 20 do
 begin
   m[k]:=random(90)+1;
form1.StringGrid1.Cells[0,k]:=inttostr(k);
form1.StringGrid1.Cells[1,k]:=inttostr(m[k]);
 end;
```

Здесь таблица используется для вывода элементов массива.

Главное свойство StringGrid — это Cells[A,B], где A — номер столбца, B — номер строки. Обращаясь к конкретной строке и столбцу мы выделяем отдельную клетку, в которой может храниться текстовая строка. Таким образом StringGrid — аналог двумерного массива строк. Будем называть этот компонент таблицей строк. В таблицу можно заносить разные данные, как строки, так и числа. Но при этом нужно помнить, что фактически это все строки. Поэтому нужно использовать функции преобразования типов (inttostr, strtoint, floattostr, strtofloat). Delphi имеет так же компоненты для создания таблиц, содержащих например рисунки, они построены как наследники класса таблицы строк, однако работать с ними существенно сложнее.

16.Компоненты диалогов. Диалоги выбора шрифта, выбора цвета. Диалоги открытия и сохранения файлов.

Компоненты - диалоги

В состав Windows входит ряд типовых диалоговых окон, таких как окно выбора загружаемого файла, окно выбора шрифта, окно для настройки принтера и т. д. В Delphi реализованы классы, объекты которых дают программисту удобные способы создания и использования таких окон. Эти компоненты принято называть диалогами и расположены они на специальной странице Dialogs.

Работа со стандартными диалоговыми окнами осуществляется в три этапа.

Вначале на форму помещается соответствующий компонент и осуществляется настройка его свойств (следует заметить, что собственно компонент-диалог не виден в момент работы программы, видно лишь создаваемое им стандартное окно). Настройка свойств может проходить как на этапе конструирования, так и в ходе прогона программы. Как и для любых других компонентов, программист не должен заботиться о вызове конструктора и деструктора диалога - эти вызовы реализуются автоматически в момент старта и завершения программы.

На втором этапе осуществляется вызов стандартного для диалогов метода Execute, который создает и показывает на экране диалоговое окно. Вызов этого метода обычно располагается внутри обработчика какого-либо события. Например, обработчик выбора опции меню. Открыть файл может вызвать метод Execute Диалога TOpenDialog, обработчик нажатия инструментальной кнопки сохранить может вызвать такой же метод у компонента TSaveDialog и т. д. Только после обращения к Execute на экране появляется соответствующее диалоговое окно. Окно диалога является модальным окном, поэтому сразу после обращения к Execute дальнейшее выполнение программы приостанавливается до тех пор, пока пользователь не закроет окно. Поскольку Execute - логическая функция, она возвращает в программу True, если результат диалога с пользователем был успешным. Проанализировав результат Execute, программа может выполнить третий этап - использование введенных с помощью диалогового окна данных - имени файла, настроек принтера, выбранного шрифта и т. д.

Пример процедуры-обработчика, которая загружает содержимое текстового файла в Memo1:

```
procedure TForm1.BitBtn2Click(Sender: TObject);
begin
form1.OpenDialog1.Execute;
```

```
if form1.OpenDialog1.FileName<>'' then begin
form1.Memo1.Lines.LoadFromFile(form1.OpenDialog1.FileName);
end;
```

КОМПОНЕНТЫ СТРАНИЦЫ DIALOGS

- TOpenPictureDialog и TSavePictureDialog - диалоги открытия и сохранения изображений
- TFontDialog -диалог выбора шрифта
- TColorDialog - диалог выбора цвета
- TPrintDialog - диалог настройки параметров печати
- TPrinterSetupDialog - диалог настройки параметров принтера
- TFindDialog- диалог поиска
- TReplaceDialog - диалог поиска и замены

Варианты диалогов:

- TOpenDialog и TSaveDialog - диалоги открытия и сохранения файлов Свойство FileName: string содержит маршрут поиска и выбранный файл при успешном завершении диалога. Программа может использовать это свойство для доступа к файлу с целью читать из него данные (TOpenDialog) или записывать в него (TSaveDialog). Свойство Filter: string используется для фильтрации (отбора) файлов, показываемых в диалоговом окне. Это свойство можно устанавливать с помощью специального редактора на этапе конструирования формы или программно, как это сделано в предыдущем примере. Для доступа к редактору достаточно щелкнуть по кнопке в строке Filter окна Инспектора объектов. При программном вводе фильтры задаются одной длинной строкой, в которой символы “|” служат для разделения фильтров друг от друга, а также для разделения описания фильтруемых файлов от соответствующей маски выбора. Например, оператор

```
OpenDialogI.Filter := 'Текстовые файлы|*.txt| Файлы Паскаля|*.pas';
```

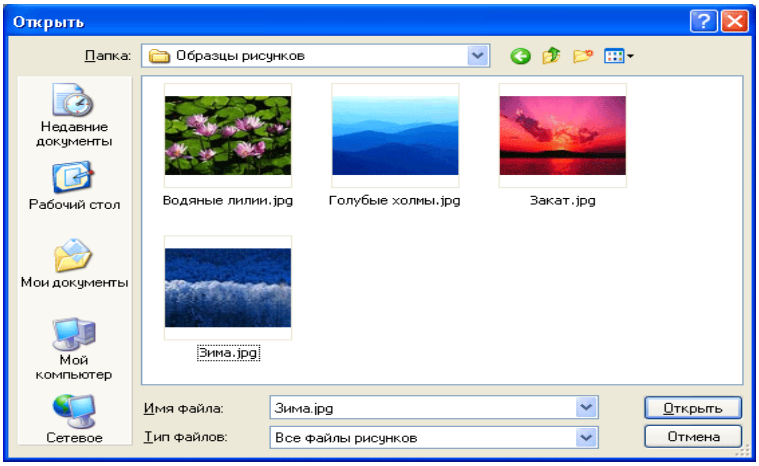
задает две маски - для отбора файлов с расширениями pas и TXT.

Установить начальный каталог позволяет свойство InitialDir string. Например:

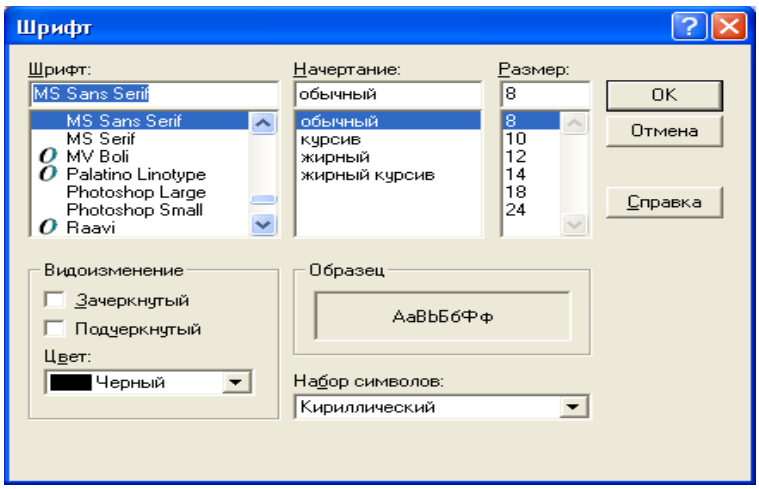
```
OpenDialogI.InitiaiDir := 'c:\program files\borland\delphi5\source';
```

С помощью свойства DefaultExt: String[3] формируется полное имя файла, если при ручном вводе пользователь не указал расширение. В этом случае к имени файла прибавляется разделительная точка и содержимое этого свойства.

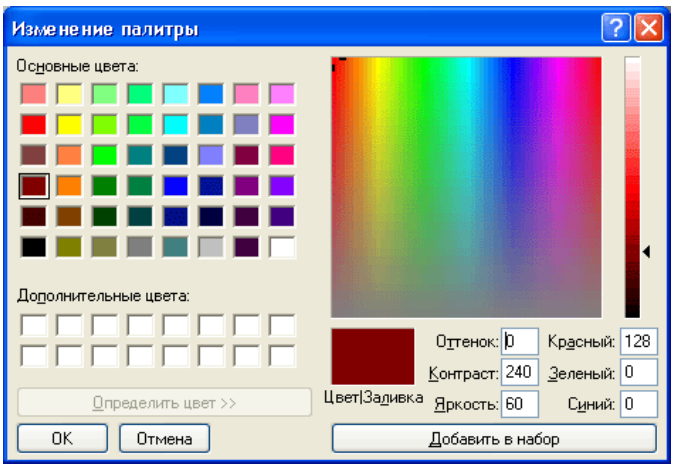
- TOpenPictureDialog и TSavePictureDialog - диалоги открытия и сохранения изображений. Специализированные диалоги для открытия и сохранения графических файлов TOpenPictureDialog И TSavePictureDialog отличаются от TOpenDialog и TSaveDialog двумя обстоятельствами. Во-первых, в них предусмотрены стандартные фильтры для выбора графических файлов (с расширениями bmp, ico, wmf и emf) . Во-вторых, в окна диалога включены панели для предварительного просмотра выбираемого файла. Пример :



- TFontDialog - диалог выбора шрифта. Компонент TFontDialog создает и обслуживает стандартное окно выбора шрифта. Пример:



- TColorDialog - диалог выбора цвета. Компонент создает и обслуживает стандартное диалоговое окно выбора цвета. Свойства компонента Color: TColor — выбранный цвет. ColorDialogOption — вид окна выбора цветов. Пример:



окно диалога загрузки графического рисунка из файла

17.Основы компьютерной графики. Графические построения и моделирование цветов.

Введение в компьютерную графику

Изображение на экране монитора состоит из отдельных точек(пикселей) и называется растровым изображением. Методы и задачи построения, обработки, хранения таких изображений относят к компьютерной графике, а изображения называют графическими. Принято делить графику на растровую и векторную. Существуют специальные программы, работающие исключительно с графическими изображениями (графические редакторы), специальные форматы файлов для хранения графики и наконец специальные алгоритмы, методы, классы и подпрограммы, которые разработаны для работы с графическими изображениями. Любое использование растровой или векторной графики в конечном случае приводит к формированию растрового изображения. Основные задачи, которые возникают при программировании растровых изображений:

7. Преобразование векторного изображения в растровое и наоборот.
8. Быстрый вывод на экран формируемого растрового изображения.
9. Минимизация искажения при формировании растрового изображения.
10. Максимальное ускорение вывода элементов изображений – графических примитивов.

Экран монитора может работать в двух режимах: текстовом и графическом. В текстовом режиме экран делится на строки и столбцы клеткой. Клетка такой сетки называется знакоместо и может содержать один знак кодовой таблицы. У знакоместа два цвета: цвет фона и цвет знака, что дает возможность представить экран в достаточно разнообразном виде.

В графическом режиме экран поделен на точки – пиксели, точки так же располагаются вдоль строк и столбцов, но их гораздо больше, чем знакомест. Количество пикселей по горизонтали и вертикали – это разрешение экрана. В настоящее время, это цифры порядка тысячи. Каждый пиксель может иметь свой цвет, в результате формируется мозаичное изображение, которое называется графическим. Графическая информация хранится в ЭВМ в растровом или векторном виде.

Растровое изображение – изображение в виде мозаики, для его хранения нужен массив содержащий код цвета каждого пикселя (пример формат BMP). Для записи растрового рисунка нужно задать местоположение каждого пикселя и его цвет. Для экономии объема растровый рисунок задается в виде прямоугольника пикселей. Указываются координаты верхнего левого угла ширина и высота. Далее последовательно блок за блоком цвета пикселей.

Векторное изображение – сжатие растровой графической информации с помощью выделения одноцветных элементов (отрезков – векторов, кривых – контуров, алгоритмических или фрактальных фигур). Поэтому существуют варианты векторного, контурного, алгоритмического и фрактального сжатия в векторной графике. Алгоритмическое сжатие по смыслу подобно общему алгоритмическому сжатию информации, когда вместо самой информации записывается алгоритм построения информации (в данном случае изображения). Фрактальное сжатие теоретически наиболее мощное, так как здесь используются специальные самоподобные изображения – фракталы. Такие изображения просто алгоритмизируются, но фактически могут быть очень сложными (напрмер так можно изображать деревья, ландшафт и т.д.). Таким образом, в векторной графике используется специальная упакованная (сжатая) запись изображения. Примеры – векторный формат графического пакета Corel Draw (cdr), файлы чертежных систем AutoCad и ArhiCad.

Простейший векторный формат представляет графическое изображение как набор отрезков прямых одного цвета, т.е. если у нас на одной прямой находятся хотя бы 4 пикселя, то для хранения такого отрезка достаточно знать направление, длину отрезка и один раз цвет. Следовательно, если цвет кодируется 3 байтами, то получаем выигрыш в сжатии информации. Ширина прямой – 1 пиксель.

Для коротких отрезков можно использовать отрезки по осям, в результате векторный формат позволяет сжать графическую информацию.

Контурный формат – одноцветные пиксели формируются в виде кривой. Такое расширение возможности с одной стороны увеличивает сжатие, а с другой стороны усложняет алгоритм.

Особенно удобными такие форматы оказались для представления шрифтов – набора знаков, изображающих текст. Особенностью шрифтов является масштабирование, растровые шрифты не масштабируются. На сегодняшний день существует два основных стандарта Post Script и True

Type.

Рисунок так же можно представить в виде алгоритма воспроизведения этого рисунка, это характерно для графических систем, которые работают с векторным форматом Auto Cad, Corel Draw. Рисунки в этих системах хранятся в виде команд, которые могут воспроизвести этот рисунок. Это достаточно высокая система сжатия. К ним можно отнести формат *.wmf, который содержит команды графического интерфейса Windows, что позволяет автоматически воспроизвести рисунок кнопки, окна и т.д.. Здесь хранят стандартное изображение.

Растровая графика и использование цветов

Растровая графика требует работы с отдельными пикселями или группами пикселей объединенными в виде графического примитива. Для каждого такого объекта требуется указать цвет или цвета (если объект многоцветный). Поэтому алгоритмы растровой графики можно разделить на алгоритмы определения цветов и построения примитивов.

Особенности восприятия цветового изображения человеком

Глаз человека воспринимает мир цветным в результате реакции сетчатки глаза на волны света различной длины волны. При этом происходит смешение цветов, образуются оттенки и некоторое «сглаживание» изображения. Подобное «сглаживание» цветовой картинки объясняется несколькими причинами:

- Зрительное восприятие глаза зависит от чувствительности к падающему цвету колбочек 3-х типов, активизирующихся при поглощении красного, синего и зеленого цвета соответственно. При этом наиболее хорошо воспринимается зеленый цвет, далее красный и наихудшая чувствительность при поглощении синего цвета. Таким образом, реакция на различные цвета несколько не симметрична.
- Как любой оптический прибор, глаз вносит искажения в формирующееся изображение. Помимо геометрических искажений, возможны и искажения цветов, которые определяются эффектами дифракции и интерференции света. В частности резкая граница между цветными областями всегда немного «расплывается», образуется эффекты «ореола», «переливания» цветов и т.д..
- Для фиксации и распознавания изображения человеком главную роль играет даже не глаз, а обработка изображения нервной системой головного мозга. При этом цветовое изображение может значительно модифицироваться, палитра цветов и контрастность уменьшаться.

Практически, учет подобных обстоятельств возможен только в результате тщательных исследований психологов и медиков. Например, эксперименты с восприятием яркости цвета в изображении (которое определяется суммой яркостей 3-х основных цветов, воспринимаемых глазом) дают эмпирическую формулу:

Яркость = 0.59*Зеленый + 0.3*Красный + 0.11*Синий

Эта формула определяется разной восприимчивостью к основным цветам. Таким образом, восприятие яркости оказывается не полностью аддитивным.

Другим интересным следствием, приведенных ранее особенностей восприятия цвета человеком, является известная технология JPEG. Оказывается сглаживание цветов при обработке изображения в нервной системе можно использовать практически для сжатия графической информации. Поскольку «сглаженное» изображение имеет меньшее количество цветов в палитре и существенно меньшую разницу цветов близко расположенных пикселей, то становится возможным предварительная обработка полноцветного изображения (формата BMP) в изображение с меньшим количеством цветов и сглаженным переходом цвета между соседними пикселями (формат JPEG). При этом часть информации о исходном изображении теряется, но для восприятия человека обе картинки практически не различимы.

Моделирование цветового изображения объекта

Таким образом, использование характерных особенностей восприятия цветового изображения приводит к модели сложения цветов (аддитивная модель) Красного (Red), Зеленого (Green), Синего (Blue) - RGB. В этой модели интенсивность каждого цвета является независимой величиной, поэтому часто представляют эту систему как

трехмерную систему координат, где по каждой координате откладывается один из базовых цветов. Интенсивность итогового цвета определяется суммой интенсивностей базовых цветов (например по приведенной ранее формуле), а положение цвета в спектре «радуги» отношением интенсивностей базовых цветов. В стандартной цветовой палитре (TrueColor) каждый базовый цвет определяется 1 байтом, а общий цвет – 3 байтами +1 технический байт. Такая цветовая схема обеспечивает 2^{24} цвета, что обычно достаточно. Палитра – количество цветов, которые одновременно могут быть отображены на экране монитора. В принципе таких цветов может быть меньше чем 2^{24} , что обеспечивает возможность использования других палитр (HighColor - 2^{16} цветов, 256 цветная палитра и т.д.).

Для моделирования цветов на экране монитора трехцветная модель RGB является наиболее употребительной, однако могут использоваться и другие цветовые модели. Например, существует система TLS (тон, яркость и насыщенность). Здесь цветовой тон определяется преобладающей длиной волны в спектре цвета, яркость соответствует световой энергии, насыщенность определяет долю белого цвета.

Для моделирования цветов изображения для печати (например на цветном принтере) используются специальные схемы – например четырехцветная (CMYK). Здесь используется не аддитивная, а субтрактивная модель цвета. Каждый цвет формируется как разность исходного белого цвета и набора интенсивностей базовых цветов. Это вариант при печати удобней аддитивного, так как экран мы воспринимаем как излучение света, а изображение на бумаге как отражение. При этом, отраженный цвет проще представлять как разность исходного белого цвета и поглощенных краской цветов. В четырех цветной палитре так же существует понятие оттенка, что определяет его модель как 4-х параметрическую.

18.Классы графики в Delphi. Класс TCanvas.

Построение графических изображений в Delphi. Класс TCanvas.

Delphi позволяет программисту разрабатывать программы, которые могут выводить графику. Программа выводит графику на поверхность объекта (формы или компонента Image). Поверхности объекта соответствует свойство canvas. Для того чтобы вывести на поверхность объекта графический элемент (прямую линию, окружность, прямоугольник и т. д.), необходимо применить к свойству canvas этого объекта соответствующий метод. Например, инструкция Form1.Canvas.Rectangle (10,10,100,100) вычерчивает в окне программы прямоугольник. поверхности, на которую программа может выводить графику, соответствует свойство Canvas. В свою очередь, свойство canvas — это объект типа TCanvas. Методы этого типа обеспечивают вывод графических примитивов (точек, линий, окружностей, прямоугольников и т. д.), а свойства позволяют задать характеристики выводимых графических примитивов: цвет, толщину и стиль линий; цвет и вид заполнения областей; характеристики шрифта при выводе текстовой информации.

Методы вывода графических примитивов рассматривают свойство Canvas как некоторый абстрактный холст, на котором они могут рисовать (canvas переводится как "поверхность", "холст для рисования"). Холст состоит из отдельных точек — пикселов. Положение пиксела характеризуется его горизонтальной (X) и вертикальной (Y) координатами. Левый верхний пиксел имеет координаты (0, 0). Координаты возрастают сверху вниз и слева направо. Значения координат правой нижней точки холста зависят от размера холста. **Размер холста можно получить, обратившись к свойствам** Height и width **области иллюстрации** (image) или к **свойствам формы:** ClientHeight и Clientwidth.

Классы TPen, TBrash, TPoint

Художник в своей работе использует карандаши и кисти. Методы, обеспечивающие вычерчивание на поверхности холста графических примитивов, тоже используют карандаш и кисть. Карандаш применяется для вычерчивания линий и контуров, а кисть — для закрашивания областей, ограниченных контурами. Карандашу и кисти, используемым для

вывода графики на холсте, соответствуют свойства Pen (карандаш) и Brush (кисть), которые представляют собой объекты классов TPen и TBrush, соответственно. Значения свойств этих объектов определяют вид выводимых графических элементов.
Класс TPoint - класс точки, имеет 2 свойства (поля) X и Y — координаты точки.

Графические примитивы. Методы TCanvas.

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

Вычерчивание прямой линии осуществляет метод LineTo, инструкция вызова которого в общем виде выглядит следующим образом:

Компонент.Canvas.LineTo(x,y)

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

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

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

Метод polyline вычерчивает ломаную линию. В качестве параметра метод получает массив типа TPoint. Каждый элемент массива представляет собой запись, поля x и y которой содержат координаты точки перегиба ломаной. Метод Polyline вычерчивает ломаную линию, последовательно соединяя прямыми точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д. Метод Polyline можно использовать для вычерчивания замкнутых контуров. Для этого надо, чтобы первый и последний элементы массива содержали координаты одной и той же точки.

Прямоугольник вычерчивается методом Rectangle, инструкция вызова которого в общем виде выглядит следующим образом:

Объект.Canvas.Rectangle(x1, y1,x2, y2)

где:

- объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;
- x1, y1 и x2, y2 — координаты левого верхнего и правого нижнего углов прямоугольника.

Метод RoundRec тоже вычерчивает прямоугольник, но со скругленными углами. Инструкция вызова метода RoundRec выглядит так:

Объект.Canvas.RoundRec(x1,y1,x2, y2, x3, y3)

где:

- x1, y1, x2, y2 -- параметры, определяющие положение углов прямоугольника, в который вписывается прямоугольник со скругленными углами;
- x3 и y3 — размер эллипса, одна четверть которого используется для вычерчивания скругленного угла

Вид линии контура (цвет, ширина и стиль) определяется значениями свойства Pen, а цвет и

стиль заливки области внутри прямоугольника — значениями свойства Brush поверхности (canvas), на которой прямоугольник вычерчивается.

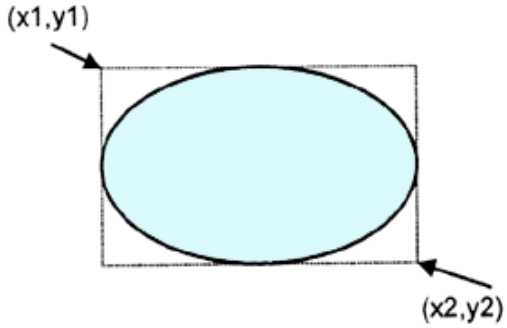
Есть еще два метода, которые вычерчивают прямоугольник, используя в качестве инструмента только кисть (Brush). Метод FillRect вычерчивает закрашенный прямоугольник, а метод FrameRect — только контур. У каждого из этих методов лишь один параметр — структура типа TRect. Поля структуры TRect содержат координаты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

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

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

Для вывода текста на поверхность графического объекта используется метод TextOut. Примитивы типа окружность-дуга окружности, эллипс-дуга эллипса могут строиться при использовании специальных методов Canvas.
Метод Ellipse вычерчивает эллипс или окружность, в зависимости от значений параметров. Инструкция вызова метода в общем виде выглядит следующим образом:
Объект.Canvas.Ellipse(x1,y1, x2,y2]
где:

- объект — имя объекта (компонента), на поверхности которого выполняется вычерчивание;
- x1, y1, x2, y2 — координаты прямоугольника, внутри которого вычерчивается эллипс или, если прямоугольник является квадратом, окружность.

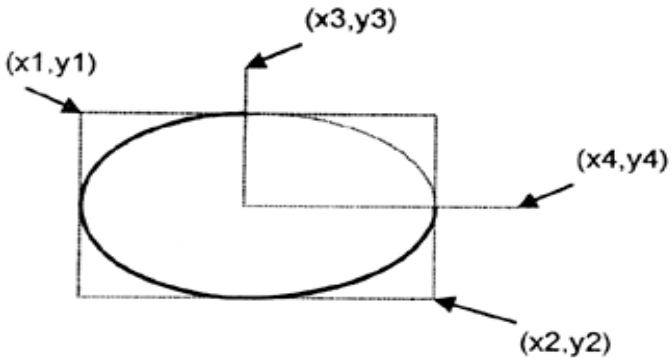


Цвет, толщина и стиль линии эллипса определяются значениями свойства Pen, а цвет и стиль заливки области внутри эллипса — значениями свойства Brush поверхности (canvas), на которую выполняется вывод.
Вычерчивание дуги выполняет метод Arc, инструкция вызова которого в общем виде выглядит следующим образом:
Объект.Canvas.Arc(x1,y1,x2,y2,x3,y3,x4,y4)
где:

- x1, y1, x2, y2 — параметры, определяющие эллипс (окружность), частью которого является вычерчиваемая дуга;
- x3, y3 — параметры, определяющие начальную точку дуги; x4, y4 — параметры, определяющие конечную точку дуги.

Начальная (конечная) точка — это точка пересечения границы эллипса и прямой, проведенной из центра эллипса в точку с координатами x3 и y3 (x4, y4). Дуга вычерчивается против

часовой стрелки от начальной точки к конечной



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

Объект. Canvas.Pie(x1,y1,x2,y2,x3,y3,x4,y4)

где:

- x1, y1, x2, y2 — параметры, определяющие эллипс (окружность), частью которого является сектор;
- x3, y3, x4, y4 — параметры, определяющие координаты конечных точек прямых, являющихся границами сектора.

Начальные точки прямых совпадают с центром эллипса (окружности). Сектор вырезается против часовой стрелки от прямой, заданной точкой с координатами (x3, уз), к прямой, заданной точкой с координатами (x4, y4)

19.Компоненты для графических построений. Компоненты Image, Paintbox, Shape, Timer.

Компоненты Image, Paintbox, Shape, Timer

В среде Delphi есть ряд комонентов, которые дают дополнительные возможности для работы с графическими изображениями:

Компонент Image — помимо доступа к Canvas этот компонент может использоваться для воспроизведения рисунка, который находится в файле с расширением bmp, jpg или ico. Компонент image находится на вкладке **Additional** палитры компонентов. Здесь же располагается и интересный компонент Shape.

Свойства Image

Свойство	Определяет
Picture	Окно диалога выбора файла рисунка
AutoSize	Признак автоматического изменения размера компонента в соответствии с реальным размером иллюстрации. Если размер компонента меньше размера иллюстрации, и значение свойств AutoSize и strech равно False, то отображается часть иллюстрации
Strech	Признак автоматического масштабирования иллюстрации в соответствии с реальным размером компонента. Чтобы было выполнено масштабирование, значение свойства AutoSize должно быть False

Чтобы вывести рисунок в поле компонента image во время работы программы, нужно применить метод LoadFromFile к свойству Picture, указав в качестве параметра имя файла рисунка. Например, инструкция Form1.Image1.Picture.LoadFromFile('e:\temp\bart.bmp') загружает рисунок из файла bart.bmp и выводит ее в поле вывода иллюстрации (imagel).

Компонент Shape - стандартная фигура

Компонент рисует одну из простейших геометрических фигур, определяемых следующим множеством: type TShapeType = (stRectangle, stSquare, stRoundRect, stRoundSquare, stEllipse, stCircle) ; (прямоугольник, квадрат, скругленный прямоугольник, скругленный квадрат, эллипс, окружность). Фигура полностью занимает все пространство компонента. Если задан квадрат или круг, а размеры элемента по горизонтали и вертикали отличаются, фигура чертится с размером меньшего измерения. Помимо стандартных чертежных инструментов Brush и pen (шрифт для компонента не нужен) в компоненте определено свойство shape.' TShapeType, которое и задает вид геометрической фигуры. Изменение этого свойства приводит к немедленной перерисовке изображения.

Компоненты панели System

TPaintBox - окно для рисования

Назначение компонента TPaintBox - дать простое окно с холстом для рисования произвольных изображений. Канва содержится в свойстве Canvas компонента, графические инструменты - в свойствах Font, pen и Brush, а собственно рисование осуществляется в обработчике события OnPaint.

TTimer - таймер

Компонент TTimer (таймер) служит для отсчета интервалов реального времени. Его свойство interval определяет интервал времени в миллисекундах, который должен пройти от включения таймера до наступления события onTimer. Таймер включается при установке значения True в его свойство Enabled. Раз включенный таймер все время будет возбуждать события onTimer до тех пор, пока его свойство Enabled не примет значения False.

20.Классы графики в Delphi. Классы TPen, TBrash, TPoint.

Для того чтобы вывести на поверхность объекта графический элемент (прямую линию, окружность, прямоугольник и т. д.), необходимо применить к свойству canvas этого объекта соответствующий метод. Например, инструкция Form1.Canvas.Rectangle (10,10,100,100) вычерчивает в окне программы прямоугольник. поверхности, на которую программа может выводить графику, соответствует свойство Canvas. В свою очередь, свойство canvas — это объект типа TCanvas. Методы этого типа обеспечивают вывод графических примитивов (точек, линий, окружностей, прямоугольников и т. д.), а свойства позволяют задать характеристики выводимых графических примитивов: цвет, толщину и стиль линий; цвет и вид заполнения областей; характеристики шрифта при выводе текстовой информации.

Классы TPen, TBrash, TPoint

Художник в своей работе использует карандаши и кисти. Методы, обеспечивающие вычерчивание на поверхности холста графических примитивов, тоже используют карандаш и кисть. Карандаш применяется для вычерчивания линий и контуров, а кисть — для закрашивания областей, ограниченных контурами. Карандашу и кисти, используемым для вывода графики на холсте, соответствуют свойства Pen (карандаш) и Brush (кисть), которые представляют собой объекты классов TPen и TBrush, соответственно. Значения свойств этих объектов определяют вид выводимых графических элементов.

Свойства объекта **TPen** (карандаш):

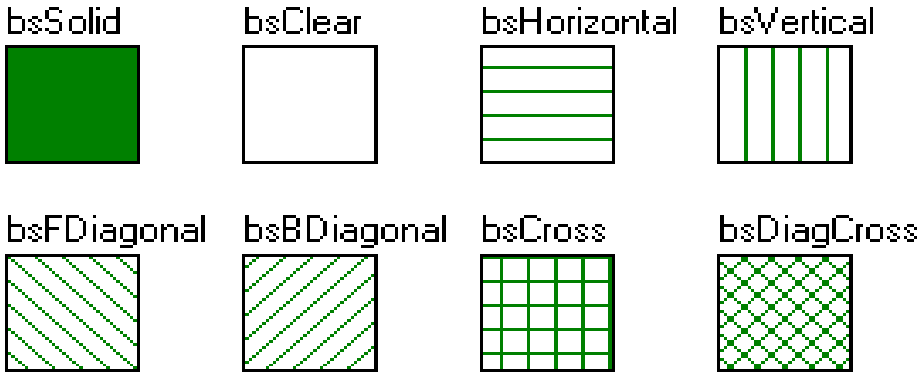
Свойство -Определяет

Color -Цвет линии, Width -Толщину линии, Style - Вид линии, Mode - Режим отображения.

Свойство Color задает цвет линии, вычерчиваемой карандашом. Свойство width задает

толщину линии (в пикселах). Например, инструкция Canvas. Pen. width: =2 устанавливает толщину линии в 2 пиксела. Свойство style определяет вид (стиль) линии, которая может быть непрерывной или прерывистой, состоящей из штрихов различной длины. Есть именованные константы, позволяющие задать стиль линии. Толщина пунктирной линии не может быть больше 1. Если значение свойства Pen.width больше единицы, то пунктирная линия будет выведена как сплошная. Свойство Mode определяет, как будет формироваться цвет точек линии в зависимости от цвета точек холста, через которые эта линия прочерчивается. По умолчанию вся линия вычерчивается цветом, определяемым значением свойства Pen.Color. Однако программист может задать инверсный цвет линии по отношению к цвету фона. Это гарантирует, что независимо от цвета фона все участки линии будут видны, даже в том случае, если цвет линии и цвет фона совпадают.

Кисть (canvas.Brush, класс **TBrush**) используется методами, обеспечивающими вычерчивание замкнутых областей, например геометрических фигур, для заливки (закрашивания) этих областей. Кисть, как объект, обладает двумя свойствами:
Color – Цвет закрашивания замкнутой области
Style - Стиль (тип) заполнения области
Область внутри контура может быть закрашена или заштрихована. В первом случае область полностью перекрывает фон, а во втором — сквозь незаштрихованные участки области будет виден фон.
Константы, позволяющие задать стиль заполнения области, приведены в примере:



Класс TPoint - класс точки, имеет 2 свойства (поля) X и Y — координаты точки.

21.Алгоритмы графических построений. Построение графика функции.

Графическая система координат

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

Form1.Canvas.Pixels[10,10]:=clRed

окрашивает точку поверхности формы в красный цвет.

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

Нужно учитывать, что координаты пикселей по вертикальной оси (ось Y) имеют перевернутую ось – нулевое значение вверху экрана и не имеют отрицательной части оси.

Тогда использование обычных координат приводит к обязательной процедуре преобразования координат обычных в экранные по формуле:

Y' = Ymax – Y , здесь Y' — новая координата, Ymax – максимальное значение по оси Y, а Y — исходная координата точки. Таким образом, нам нужно максимальное значение по оси Y, которое должно быть не более ClientHeight — размера области построения.

Масштабирование

Помимо переворота осей для правильного позиционирования точек изображения на экране монитора необходимо масштабирование (процедура пересчета координат в масштаб размера области построения на экране). Рассмотрим например ось X. Пусть мы отображаем отрезок $[X_1, X_2]$ на область построения с координатами $[0, X_{max}]$, тогда формула преобразования : $X' = X_{max}*(X-X_1)/(X_2-X_1)$. Для оси Y аналогичные переобразования должны так же учитывать переворот оси:

$Y' = Y_{max}-Y_{max}*(Y-Y_1)/(Y_2-Y_1)$.

При этом результат нужно округлять до целого, так как значения координат соответствуют целочисленному числу пикселей.

Алгоритмизация задач графических построений

Любая картинка, чертеж, схема могут рассматриваться как совокупность графических примитивов: точек, линий, окружностей, дуг и др. Таким образом, для того чтобы на экране появилась нужная картинка, программа должна обеспечить вычерчивание (вывод) графических примитивов, составляющих эту картинку.

Существует как минимум несколько вариантов алгоритмизации графического построения:

а) Построение по точкам. Задаются координаты всех точек (пикселей) изображения и затем определяются их цвета. Это метод долгий и не оптимальный.

б) Построение по векторам. Здесь изображение заменяется множеством замкнутых областей одного цвета. Границы областей задаются замкнутой ломанной, а внутри заливаются штриховкой. Часто этот метод используется для рисунков, состоящих только из линий (например диаграммы, графики функций и т. д.). Наиболее сложный процесс здесь — оперирование с массивами точек для ломанных. Эти точки часто приходится вычислять численно, с помощью специальных численных методов (сплайны, полиномы Безье, интерполяция и т. д.).

в) Построение по примитивам. Здесь изображение разбивается на набор графических примитивов (как правило одного цвета) и запоминаются отдельные точки «привязки» примитива к месту в изображении. Примитивы строятся специальными процедурами (например методами Canvas). Это самый быстрый способ построения.

Во всех 3 вариантах приходится проводить расчеты преобразования координат точек в точки системы координат изображения.

Рассмотри некоторые примеры алгоритмов построения изображений:

Построение отрезка прямой

Для построения отрезка прямой необходимо использовать формулы (y и x – координаты концов отрезка):

$$y = k \cdot x + d, \_\_ d = \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1}, \_\_ k = \frac{y_1 - y_2}{x_1 - x_2}$$

Построение кривых линий, дуг

Для построения кривых линий необходимы формулы нелинейных функций – парабол, гипербол, тригонометрических функций, экспоненты и их комбинации. Если они должны проходить через определенные точки, то для однозначности функции этих точек должно быть достаточно много.

Другим важным методом задания примитивов и фигур является использование параметрических функций. Например для задания спирали можно использовать формулу окружности с зависящим от параметра t радиусом:

$$y = \sqrt{R^2 - x^2}, \_\_ R = a * t$$

Примитивы типа окружность-дуга окружности, эллипс-дуга эллипса могут строиться при использовании специальных методов Canvas.

Построение графика функции

Задача построения графика функции разбивается на 2 варианта:

Построение графика одномерной функции. Если задана функция F(x) и необходимо построить ее график на отрезке [a,b], то необходимо выполнить следующие операции:

1. Создать область построения. Выбирается холст (свойство Canvas доступно у формы или компонента Image, который помещается на Panel).
2. Построить с помощью метода Rectangle внешний прямоугольник и линии осей. Оси лучше рисовать прямоугольником, так это сразу дает хорошую толщину и вид линий.
3. Вычислить 2 массива – массив точек x_i и массив значений функции в точках $y_i=F(x_i)$. Далее их нужно будет перевести в целочисленные координаты графического изображения. Результат удобно совмещать в массиве типа TPoint, его поля x и y и будут искомыми значениями. Для перевода нужно воспользоваться приведенными выше формулами преобразования. Здесь нам потребуются граничные значения для масштабирования. Проще всего найти max(F) и min(F) и выбрать эти значения для границ по оси Y. При этом произойдет автоматическое масштабирование графика так, чтобы он полностью занимал всю графическую область. Аналогично a и b можно считать границами по оси X. Основная проблема здесь – нет симметричности графика, в некоторых случаях оси нужно удалять из графика. Для симметрии и нерушимости осей можно в качестве границ выбирать +max(|F|) и -max(|F|), +max(|a|,|b|) и - max(|a|,|b|). Теперь весь график помещается в области построения, а оси пересекаются точно по центру области.
4. Построить ломанную линию, соединяющую все точки нашего массива типа TPoint с помощью метода PolyLine. Если точек не менее 40, то график будет достаточно качественным и «гладким».

Построение графика двумерной функции. Такой график изображается как плоскость в 3-х мерном пространстве. Чтобы его изобразить, нужна двумерная сетка точек (x,y) на которой заданы значения функции Z=F(x,y) (сеточная аппроксимация, точки x_i,y_i).. Эти точки можно перевести в экранные координаты и учесть масштабирование как в одномерном графике, но этого теперь недостаточно. Для изображения трехмерного объекта на плоском экране нужна проекция. Теоретически проекция представляется линейным преобразованием координат точек с помощью матрицы проекции. Практически можно поступить проще – в каждой точке координаты x и y будут изменены прибавлением значения Z (умноженного на некий множитель) и введено искажение прямоугольной сетки в некоторую косоугольную. Например:

```
Xn:=(x-x0)*cos(alf)-(y-y0)*sin(alf);
Yn:=((x-x0)*sin(alf)+(y-y0)*cos(alf))*cos(bet)-(z-z0)*sin(bet);
Zn:=((x-x0)*sin(alf)+(y-y0)*cos(alf))*sin(bet)+(z-z0)*cos(bet);
Xn:=Xn/(Zn/a+1);
Yn:=Yn/(Zn/a+1);
```

С помощью углов поворота alf и bet сетка становится косоугольной, а выражения типа Xn:=Xn/(Zn/a+1) показывают вариант прибавления значения Z к координате X.

22.Алгоритмы графических построений. Понятие о геометрическом моделировании.

Алгоритмизация задач графических построений

Любая картинка, чертеж, схема могут рассматриваться как совокупность графических примитивов: точек, линий, окружностей, дуг и др. Таким образом, для того чтобы на экране появилась нужная картинка, программа должна обеспечить вычерчивание (вывод) графических примитивов, составляющих эту картинку.

Существует как минимум несколько вариантов алгоритмизации графического построения:

а) Построение по точкам. Задаются координаты всех точек (пикселей) изображения и затем определяются их цвета. Это метод долгий и не оптимальный.

б) Построение по векторам. Здесь изображение заменяется множеством замкнутых областей одного цвета. Границы областей задаются замкнутой ломанной, а внутри заливаются штриховкой. Часто этот метод используется для рисунков, состоящих только из линий (например диаграммы, графики функций и т. д.). Наиболее сложный процесс здесь — оперирование с массивами точек для ломанных. Эти точки часто приходится вычислять численно, с помощью специальных численных методов (сплайны, полиномы Безье, интерполяция и т. д.).

в) Построение по примитивам. Здесь изображение разбивается на набор графических примитивов (как правило одного цвета) и запоминаются отдельные точки «привязки» примитива к месту в изображении. Примитивы строятся специальными процедурами (например методами Canvas). Это самый быстрый способ построения.

Во всех 3 вариантах приходится проводить расчеты преобразования координат точек в точки системы координат изображения.

Рассмотри некоторые примеры алгоритмов построения изображений:

Построение отрезка прямой

Для построения отрезка прямой необходимо использовать формулы (y и x – координаты концов отрезка):

$$y = k \cdot x + d, \_\_ d = \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1}, \_\_ k = \frac{y_1 - y_2}{x_1 - x_2}$$

Построение кривых линий, дуг

Для построения кривых линий необходимы формулы нелинейных функций – парабол, гипербол, тригонометрических функций, экспоненты и их комбинации. Если они должны проходить через определенные точки, то для однозначности функции этих точек должно быть достаточно много.

Другим важным методом задания примитивов и фигур является использование параметрических функций. Например для задания спирали можно использовать формулу окружности с зависящим от параметра t радиусом:

$$y = \sqrt{R^2 - x^2}, \_\_ R = a * t$$

Примитивы типа окружность-дуга окружности, эллипс-дуга эллипса могут строиться при использовании специальных методов Canvas.

Геометрическое моделирование

Геометрическим моделированием называются методы моделирования:

- геометрического изображения объекта
- поверхности объекта
- среды объекта (освещение и т.д.)
- моделирование цветов, оттенков

Сущность геометрического моделирования – представить объект геометрически правильно в двумерной или трехмерной системе координат. Фигура может быть представлена в виде набора графических примитивов (отрезков, дуг, окружностей, эллипсов, сплайнов). Так как изображение формируется в системе координат, то задается некий массив точек для привязки примитивов, который хранит расположение в пространстве между ними. Как правило можно выделить отдельно задачи проектирования, движения и обрезания изображений.

Движение изображения — последовательное перемещение его с задержкой для фиксации глазом его нового местоположения. Фактически это вариант преобразования координат в 2d, где есть формулы для поворота, переноса, сжатия и растяжения.

1) Поворот на угол

$$x^* = x\cos\varphi - y\sin\varphi \quad y^* = x\sin\varphi + y\cos\varphi$$

2) Растяжение

$$x^* = ax \quad y^* = by$$

3) Перенос.

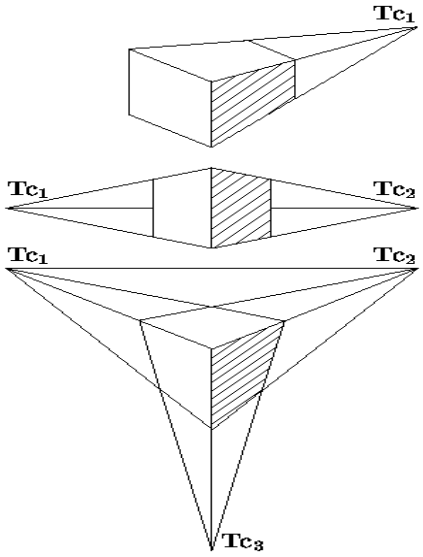
$$x^* = x + dx \quad y^* = y + dy$$

Принято произвольное линейное преобразование описывать матрицей, объединяющей все эти виды переносов, растяжений и поворотов.

Проекции необходимы для представления 3-х мерных изображений на плоскости (существуют 2d и 3d моделирование). Проектирование производится путем пересчета координат точек с помощью специальных матриц проектирования.

Чтобы изобразить трехмерное тело на плоскости (экрана например) необходимы специальные средства искажения изображения, которые принято называть проекциями. Существуют несколько вариантов проекций:

- Параллельная проекция: точки предмета проецируются (продолжаются) лучами параллельно заданному направлению. Таким образом, параллельные линии остаются параллельными, но получают некий наклон. Поскольку человек привык к визуальному эффекту уменьшения размеров при удалении предметов, то эта проекция годится только для показа близких целей.
- Центральная проекция – проектирующие лучи проходят через одну точку. Эта точка называется центром проекции. Существуют варианты 1 ,2 и 3 центров проекции. Фактически центр проекции – бесконечность. Если центр один, то бесконечность видна в одном направлении, если нет, то появляются другие перпендикулярные направлений бесконечности. В 3-х мерном пространстве их может быть не более 3.



Пример построения центральных проекций (3 варианта)

Эти 2 базовых проекции дают целый набор проекций – наследников:

Каждая из описанных проекций имеет свою матрицу проектирования, которая позволяет пересчитать точки трехмерного тела в координаты на плоскости с учетом выбранной проекции.

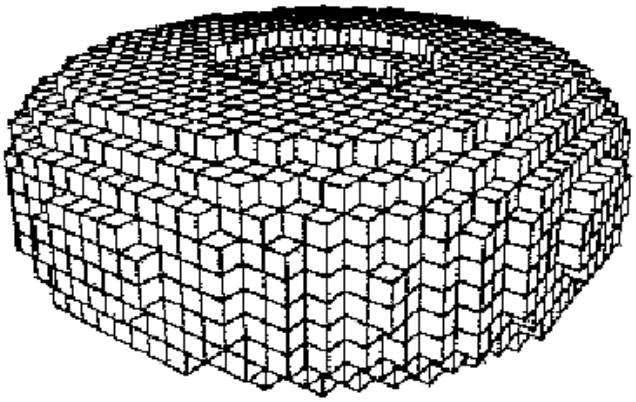
Моделирование объема и поверхности объекта

Для моделирования поверхности используют принцип текстуры, т.е. выбираются значительные части поверхности, которые будут заполнены одинаковой текстурой. Текстура состоит из одинаковых рисунков, которые повторяются. Таким образом, вариант текстуры определяется исходным образцом, который копируется. Этот образец может быть создан отдельно, скопирован из другого изображения, фотографии.

Очевидная сложность пересчета для огромного количества пикселей объемного изображения требует введения каркасной модели. В этой модели хранятся не все пиксели изображения, а только ряд точек. При построении изображения точки соединяют кривыми (сплайнами, кривыми Безье, интерполяционными полиномами). Таким образом, каркасная модель позволяет представить объем ограничивающей его поверхностью, однако это не единственный вариант.

Другой вариант – воксельное представление объема. Воксель – простейший элемент объема, которым заполняют объем. Примеры вокселей – кубы, шары, пирамиды и т.д. При использовании вокселей, хранят только координаты центров этих фигур - если все воксели одинаковы. При этом существует вариант неоднородных вокселей. Если объем достаточно велик, то выгоднее внутри иметь более крупные по размерам воксели, а в близи поверхности размер вокселя уменьшать. Тогда непрерывность объема дает возможность экономить при хранении такой информации. Далее поверхность как бы «натягивают» на воксельный объем. Для этого определяются граничные точки крайних вокселей. Эти точки используются для привязки граничной поверхности. Таким образом, массив точек центров и размеров вокселей не только определяет объем фигуры, но и положение его поверхности.

При необходимости перемещения, поворота, сжатия или растяжения объема достаточно произвести матричные преобразования с вокселями (т.е. координатами центра и дополнительной точки вокселя, определяющей его размер).



Пример заполнения кубическими вокселями объемной фигуры.

23.Классы и объекты в Delphi. Структура класса.

Классический язык Pascal позволяет программисту определять свои собственные сложные типы данных — записи (records). Язык Delphi, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять классы. Класс — это сложная структура, включающая, помимо описания данных, описание процедур и функций, которые могут быть выполнены над представителем класса — объектом.

Вот пример объявления простого класса:

```
TPerson = class
private
fname: string[15]; faddress: string[35];
public
procedure Show;
end;
```

Данные класса называются полями, процедуры и функции — методами.

В Приведенном Примере TPerson — это имя класса, fname и faddress - имена полей, show — имя метода.

Описание класса помещают в программе в раздел описания типов (type).

Дальнейшим развитием объектно-ориентированного подхода стало появление объекта, который реализует принципы инкапсуляции, полиморфизма и наследования. Для объектов в Object Pascal уже определена специальная пользовательская структура данных – класс.

О. Класс – пользовательский тип данных, который по структуре подобен записи, но в качестве полей содержит специальные элементы – поля, методы и свойства.

Структура класса гораздо сложнее записи, поэтому внутри класса выделены специальные области:

```
Type
   Имя класса = class (имя предка)
        Published   {Специальная область элементов, доступных для инспектора объектов}
        …..
        Private   {Область закрытых элементов, доступных только внутри методов данного класса и подпрограмм данного модуля, недоступны потомкам}
        Protected {Область закрытых элементов, доступных методам класса и его потомков}
        Public        {Область элементов, доступных в любом модуле проекта}
        Automated {Секция автоматизации - элементы, необходимые для технологии OLE }
        end;
```

Разрешается сколько угодно раз объявлять любую секцию в произвольном порядке. По умолчанию после объявления класса область считается типа Published, пользователю не рекомендуется редактировать эту область, так это отразится на работе визуальной среды. Если необходимо в класс добавить свои элементы, то можно ввести новую область недоступную инспектору объектов. При описании элементов структуры класса принято в

начале задавать поля, затем методы и свойства.

Замечание. После специального слова class в скобках может быть указан класс-предок, от которого наследуются дополнительные поля, методы и свойства. Для среды Delphi характерно, что здесь по умолчанию все классы считаются наследуемыми от общего для всех класса TObject. Принято имя класса начинать с буквы T.

Инкапсуляция и свойства объекта

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

В языке Delphi ограничение доступа к полям объекта реализуется при помощи свойств объекта. Свойство объекта характеризуется полем, сохраняющим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства. Метод установки значения свойства называется методом записи свойства (write), а метод получения значения свойства — методом чтения свойства (read).

В описании класса перед именем свойства записывают слово property (свойство). После имени свойства указывается его тип, затем — имена методов, обеспечивающих доступ к значению свойства. После слова read указывается имя метода, обеспечивающего чтение свойства, после слова write — имя метода, отвечающего за запись свойства.

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

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

Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта: например, можно разрешить только чтение. Для того чтобы инструкции программы не могли изменить значение свойства, в описании свойства надо указать лишь имя метода чтения. Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку.

24.Свойства и методы класса. Конструктор и деструктор класса.

Элементы структуры класса

Рассмотрим теперь элементы структуры класса подробнее:

1. **Поле.** О. Поле – элемент структуры, обозначающий хранение простых данных. Поля имеют тип, обычно это стандартные типы (число, строка, знак, логический тип, массив). Но можно использовать и сложную структуру – например другой класс. Главная особенность поля – способ доступа к данным. Здесь разрешен прямой доступ к данным. Принцип инкапсуляции требует, чтобы к полям обращались только через свойства и методы, поэтому разумно размещать поля в областях Private и Protected. Класс потомок получает все поля всех своих предков и может дополнять их своими, но не может их переопределять (по сравнению например с методами) или удалять.

Замечание. Так же как и для полей записи обращение к полям возможно либо с помощью указания имени объекта, точки и имени поля (составное имя), либо с помощью структурного оператора with.

Метод. Метод – процедура или функция, с помощью которых реализуются действия объекта. Доступ к методам аналогичен доступу к полям. Однако, если поля не рекомендуется произвольно менять, то ограничений по использованию методов не декларируется. Почему? Здесь, как и в любой методике программирования требования определяются 2-мя главными принципами – упрощение действий программиста и ограничение возможностей выполнения неверных действий. Обращаясь напрямую к полям мы пользуемся наилучшим способом запутывания работы программы, что влечет неоправданное усложнение алгоритма, поиска

ошибок и тестирования. Обращение к полям требует знания их имен, типов, возможных значений – это одновременно и усложнение работы и место появления новых ошибок. Методы и свойства создаются так, чтобы контролировать возможные ошибки и уменьшить количество параметров. Кроме того, передача свойств и методов под управление инспектора объектов, так же облегчает задачу управления объектом. Может показаться, что эти проблемы не очень важны. Однако на практике (за счет наследования) объекты имеют десятки, а то и сотни полей, свойств и методов. В этой ситуации, данные требования существенно экономят время разработки программ. Таким образом, при разработке классов стремятся данные передавать через свойства, а количество параметров у методов уменьшить до минимального числа.

При описании класса методы описываются в 2 этапа:

- Заголовок метода помещается в ту или иную область описания класса.
- Содержимое процедуры или функции метода размещается ниже области Type, где описывается структура класса. При этом указывается имя класса, точка и имя метода. Параметры можно не указывать, они уже описаны в заголовке метода (но при генерации метода описание параметров часто повторяют). Доступны внутри любого метода все поля, методы и свойства данного класса и некоторые элементы других классов.

Обычно пользователь пользуется стандартными методами класса, которые уже полностью описаны в библиотеках Delphi. Программировать же приходится методы обработки событий.

Свойство – поле процедурного типа. Свойства должны регулировать доступ к полям. Для этого у каждого свойства есть поле и 2 метода – чтения (read) и записи (write). Обычно свойство связано с полем и указывает те методы класса, которые обеспечивают чтение значения из поля и запись нового значения в поле. Формат записи заголовка свойства:

Property имя свойства: тип Read метод чтения из поля Write метод записи в поле Default значение по умолчанию;

При работе программы свойство используется как обычное поле, но фактически ввод и чтение данных возможно только через указанные процедуры-методы. Если нет необходимости в специальных методах, то можно указывать вместо процедуры (метода) имя поля. Если поле доступно только для чтения то не указывается метод записи и наоборот. Значение по умолчанию присваивается полю свойства, если оно не было определено (что удобно для инициализации объекта). Принято имя метода чтения начинать с Get, а метода записи с Set. Для того чтобы свойство было доступно инспектору объектов его необходимо поместить в область Published.

Еще один вариант особых методов – конструкторы и деструкторы. Конструктор – метод, который создает конкретный объект - экземпляр класса. Деструктор – метод, который удаляет конкретный объект из памяти. Использование конструктора обязательно, деструктора – по мере необходимости, так как все объекты автоматически уничтожаются после завершения программы. Деструктор важен при работе с динамической памятью. В Object Pascal обычно используется конструктор Create базового класса Tobject. Таким образом, описывать конструктор нет необходимости. Если же такая необходимость появится, то для определения методов конструктора и деструктора вместо стандартного указания Procedure или Function используются специальные операторы Constructor и Destructor.

Примечание

В Delphi объект — это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных.

Выделение памяти осуществляется при помощи специального метода класса — конструктора, которому обычно присваивают имя Create (создать). Для того чтобы подчеркнуть особую роль и поведение конструктора, в описании класса вместо слова procedure используется слово constructor.

Если в программе какой-либо объект больше не используется, то можно освободить память, занимаемую полями данного объекта. Для выполнения этого действия используют метод-де-

структор Free. Например, для того, чтобы освободить память, занимаемую полями объекта professor, достаточно записать

professor.Free;

Выделение памяти для данных объекта происходит путем присваивания значения результата применения метода-конструктора к типу (классу) объекта. Например, после выполнения инструкции

professor := TPerson.Create;

выделяется необходимая память для данных объекта professor.

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

25.Полиморфизм и наследование классов в Delphi.

Примечание

В Delphi объект — это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных. Динамические данные отличаются от обычных (статических) тем, что их структура и содержимое может меняться в процессе работы программы. Среди динамических структур играют крайне важную роль такие структуры – очередь, стек, список, дерево.

Для создания динамических структур необходимо выделение для них специальной динамической памяти.

Динамическая память - это оперативная память ПК, предоставляемая программе при ее работе. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого статическое размещение осуществляется компилятором Object Pascal в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных.

Наследование - это процесс, в результате которого один тип наследует свойства другого типа.

Полиморфизм - это концепция, позволяющая иметь различные реализации для одного и того же метода, которые будут выбираться в зависимости от типа объекта, переданного методу при вызове.

Полиморфизм и виртуальные методы

Полиморфизм — это возможность использовать одинаковые имена для методов, входящих в различные классы. Концепция полиморфизма обеспечивает з случае применения метода к объекту использование именно того метода, который соответствует классу объекта.

Полиморфизм - это свойство классов решать схожие по смыслу проблемы разными способами. В рамках Object Pascal поведенческие свойства класса определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках класса, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо *перекрыть* его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два *одноименных* метода, имеющих разную алгоритмическую основу и, следовательно, придающих объектам разные свойства. Это и называется полиморфизмом объектов.

В *Object Pascal* полиморфизм достигается не только описанным выше механизмом наследования и перекрытия методов родителя, но и их *виртуализацией* (см. пример ниже), позволяющей родительским методам обращаться к методам своих потомков.

Пример:

function info: string; virtual; - директива для виртуального метода

Как уже говорилось, методы класса могут перекрываться в потомках.

Потомки обоих классов могут выполнять сходную по названию процедуру DoWork, но, в общем случае, будут это делать по-разному. Такое замещение методов называется статическим, т. к. реализуется компилятором.

В Object Pascal гораздо чаще используется динамическое замещение методов на этапе прогона программы. Для реализации этого метод, замещаемый в родительском классе, должен объявляться как динамический (с директивой dynamic) или виртуальный (virtual). Встретив такое объявление, компилятор создаст две таблицы -DMT (Dynamic Method Table) и VMT (Virtual Method Table) и поместит в них адреса точек входа соответственно динамических и виртуальных методов. При каждом обращении к замещаемому методу компилятор вставляет код, позволяющий извлечь адрес точки входа в подпрограмму из той или иной таблицы. В классе-потомке замещающий метод объявляется с директивой override *(перекрыть).* Получив это указание, компилятор создаст код, который на этапе прогона программы поместит в родительскую таблицу точку входа метода класса-потомка, что позволит родителю выполнить нужное действие с помощью нового метода.

Для реализации полиморфизма существует специальные формы перекрытия методов – виртуальные, динамические, перезагружаемые методы.

Наследование классов в Delphi

Использование принципа наследования дает новую методологию проектирования сложных программ. Цикл жизни программы можно представить как разработку новых усовершенствованных классов. Причем новая версия программы сразу получает все достижения разработанной ранее. Тот же эффект происходит при наследовании классов, причем разрабатывая класс-потомок, можно ограничиться поверхностным знакомством с классом предка, так как любой метод или свойство можно переделать без удаления старого кода или даже совмещая их работу. Это дает возможность привлечения для программирования большого коллектива, использования старых разработок, широкого обмена классами и компонентами.

Указание на наследование задается при описании класса.

Замечание. После специального слова class в скобках может быть указан класс-предок, от которого наследуются дополнительные поля, методы и свойства. Для среды Delphi характерно, что здесь по умолчанию все классы считаются наследуемыми от общего для всех класса TObject. Принято имя класса начинать с буквы Т.

Иерархия классов Delphi

Для реализации проектов Delphi использует библиотеку классов, которая содержит большое количество разнообразных классов, поддерживающих форму и различные компоненты формы (командные кнопки, поля редактирования и т. д.). Все классы построены по принципу наследования в виде дерева (иерархии) классов. Общий предок — класс TObject. Все производные классы можно разделить на несколько семейств:

1. Технические (промежуточные) классы. Используются для внутренней реализации более сложных классов, часто не описаны и недоступны. Примеры: TPersystent, TComponent, TControl.
2. Классы описывающие объекты, которые не являются компонентами Delphi. Примеры: TPoint, TCanvas, TFont, TPen, TStrings, TBrash, TRect.
3. Классы описывающие объекты, которые являются компонентами, но не отображаются на экране при запуске проекта (не визуальные), однако в режиме проектирования они видны. Примеры: TTimer, TTable, TActionList, TDataModule.
4. Классы описывающие объекты, которые являются компонентами и отображаются на экране при запуске проекта (визуальные). Среди них есть компоненты, являющиеся визуальными условно. Такие компоненты становятся видны в особых случаях. Это например диалоги, которые появляются при выполнении метода Execute или выпадающее подменю при нажатии правой клавиши мышки.
5. Особую роль среди классов компонентов играют контейнеры, которые могут помещать другие компоненты внутри себя (становятся их хозяевами). Примеры: TForm, TPanel, TDataModule. Выделяются так же классы, которые имеют доступный холст для рисования - TImage и TForm.

26.События и их обработка. Обработка исключительных ситуаций. Событийное управление работой проекта.

Событийное управление проектом

- Одновременно с ООП и визуальными средами разработки проектов, появилась методика событийного управления работой проекта, когда объекты программы функционируют автономно, до наступления определенного события (щелчка кнопки мышки, нажатия клавиши и т.д.). Таким образом, проект формируется автоматически как автономно функционирующую систему одного из стандартных классов. Классы таких проектов формируют в Delphi специализированные библиотеки образцов, которые принято называть репозиториями. Выбирая вид проекта разработчик автоматически получает работающий проект, в котором необходимо модернизировать 10-15 % кода для создания своего оригинального проекта. Главным образом, дополнительное кодирование связано с созданием обработчиков стандартных событий прописанных как в самом проекте, так и в объектах классов его составляющих либо подчиненных.

Для большинства видимых компонентов определен набор обработчиков событий, связанных с мышью: Обработчики OnMouseDown и OnMouseUp определяют реакцию программы на соответственно нажатие и отпускание кнопки мыши, оп-MouseMove - на перемещение указателя мыши над компонентом, on-click и OnDblClick - соответственно на щелчок и двойной щелчок левой кнопки. Во всех обработчиках параметр sender содержит ссылку на компонент, над которым произошло событие, а x и y определяют координаты точки чувствительности указателя мыши в момент возникновения события в системе координат клиентской области родительского компонента. Событие OnClick возникает после OnMouseDown, но перед OnMouseUp, а событие OnDblClick возникает после OnMouseUp.

РЕАКЦИЯ НА СОБЫТИЯ ОТ МЫШИ И КЛАВИАТУРЫ

События от мыши

Для большинства видимых компонентов определен набор обработчиков событий, связанных с мышью: Обработчики OnMouseDown и OnMouseUp определяют реакцию программы на соответственно нажатие и отпускание кнопки мыши, оп-MouseMove - на перемещение указателя мыши над компонентом, оп-click и OnDblClick - соответственно на щелчок и двойной щелчок левой кнопки. Во всех обработчиках параметр sender содержит ссылку на компонент, над которым произошло событие, а x и y определяют координаты точки чувствительности указателя мыши в момент возникновения события в системе координат клиентской области родительского компонента. Событие OnClick возникает после OnMouseDown, но перед OnMouseUp, а событие OnDblClick возникает после OnMouseUp.

События от клавиатуры

События от мыши получают любые потомки TControl. В отличие от этого события от клавиатуры получают только некоторые оконные компоненты (потомки TWinControi). Обработка событий связана со следующими свойствами этих компонентов. Параметр Key в обработчиках TKeyEvent содержит виртуальный код клавиши, а в обработчике TKeypressEv-ent - ASCII-символ. Обработчики OnKeyDown и опкеуор перехваты-вают нажатие большинства клавиш клавиатуры, в то время как обработчик QnKeypress - только нажатие алфавитно-цифровых клавиш. Получаемый им символ Key учитывает выбранный язык и нажатую клавишу Shift.

Обработка исключительных ситуаций

Система прерываний компьютера служит для отслеживания и реагирования на события. Это событийное управление системой. Исключительные ситуации — варианты событий, на которые реагирует система. Стандартные классы исключительных ситуаций — классы, созданные для обработки исключительных ситуаций. Есть стандартные события — нажатие клавиш клавиатуры или мыши, изменение размеров, значений некоторых свойств компонентов. Они выделяются на вкладке Events инспектора объектов для каждого

компонента индивидуально. Для обработки событий к ним присоединяют некую процедуру — обработчик событий.
Ошибки, которые могут быть в программе, принято делить на три группы:

- синтаксические;
- ошибки времени выполнения;
- алгоритмические.

Синтаксические ошибки, их также называют ошибками времени компиляции (Compile-time error), наиболее легко устранимы. Их обнаруживает компилятор, а программисту остается только внести изменения в текст программы и выполнить повторную компиляцию.

Ошибки времени выполнения, в Delphi они называются исключениями (exception), тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы и во время тестирования.

При возникновении ошибки в программе, запущенной из Delphi, среда разработки прерывает работу программы, о чем свидетельствует заключенное в скобки слово **Stopped** в заголовке главного окна Delphi, и на экране появляется диалоговое окно, которое содержит сообщение об ошибке и информацию о типе (классе) ошибки.

С алгоритмическими ошибками дело обстоит иначе. Компиляция программы, в которой есть алгоритмическая ошибка, завершается успешно. При пробных запусках программа ведет себя нормально, однако при анализе результата выясняется, что он неверный. Для того чтобы устранить алгоритмическую ошибку," приходится анализировать алгоритм, вручную "прокручивать" его выполнение.

Предотвращение и обработка ошибок

Как было сказано выше, в программе во время ее работы могут возникать ошибки, причиной которых, как правило, являются действия пользователя. Например, пользователь может ввести неверные данные или, что бывает довольно часто, удалить нужный программе файл.

Нарушение в работе программы называется исключением. Обработку исключений (ошибок) берет на себя автоматически добавляемый в выполняемую программу код, который обеспечивает, в том числе, вывод информационного сообщения. Вместе с тем Delphi дает возможность программе самой выполнить обработку исключения.

КЛАСС EXCEPTION - ОБРАБОТКА ИСКЛЮЧЕНИЙ

Класс Exception является прямым потомком базового класса tobject. Вместе со своими потомками он предназначен для обработки исключительных ситуаций (исключений), возникающих при некорректных действиях программы: например, в случае деления на ноль, при попытке открыть несуществующий файл, при выходе за пределы выделенной области динамической памяти и т. п. В этом разделе рассматриваются основные свойства исключений и их использование для повышения надежности программ.

Для обработки исключений в Object Pascal предусмотрен механизм защищенного блока:

except	**finally**
<обработчики исключений>	<операторы>
else	**end;**
<операторы>	

end;

Try	**Try**
<операторы>	<операторы>

Защищенный блок начинается зарезервированным словом try (попытаться [выполнить]) и завершается словом end. Существуют два типа защищенных блоков - except (исключить) и finally (в завершение), отличающихся способом обработки исключения. В блоке except порядок выполнения операторов таков: сначала выполняются операторы секции try... except; если операторы выполнены без возникновения исключительной ситуации, работа защищенного блока на этом прекращается, и управление получает оператор, стоящий за end; если при выполнении части try возникло исключение, управление получает соответствующий обработчик в секции except, а если таковой не найден - первый из операторов, стоящих

За словом else. В блоке finally операторы В секции finally. . .end получают управление всегда, независимо от того, возникло ли исключение в секции try.. .finally или нет. Если исключение возникло, все операторы в секции try.. .finally, стоящие за "виновником" исключения, пропускаются, и управление получает первый оператор секции finally... end. Если исключения не было, этот оператор получает управление после выполнения последнего оператора секции try.. .finally.

Обработчики исключений в блоке except имеют такой синтаксис:

on <класс исключения> **do** <оператор>;

Здесь on, do - зарезервированные слова; <класс исключения> -класс обработки исключения; <оператор> - любой оператор Object Pascal, кроме оператора передачи управления goto на метку вне блока except.

27.Логические основы алгоритмизации. Логические операции и выражения. Логический тип данных языка Паскаль. Логические выражения в Паскале.

Логические основы алгоритмизации. Логические операции и выражения

При записи алгоритмов необходимо использование логических выражений, которые определяют разветвление в алгоритме. Практически логические выражения в алгоритмах представляются в виде логических высказываний, логических формул, предикатов.

Логическое высказывание — это любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

Логическая формула — образуется из нескольких логических высказываний, связанных логическими связками (логическими операциями).

Употребляемые в обычной речи слова и словосочетания ««**не«, «и«, «или«, «если... , то«, «тогда и только тогда«** и другие позволяют из уже заданных высказываний строить новые высказывания. Такие слова и словосочетания называются **логическими связками.**

Каждая логическая связка соответствует операции над логическими высказываниями и имеет свое название и обозначение:

Основные операции:

Операция, выражаемая словом «*не»,* называется **отрицанием** и обозначается чертой над высказыванием ,знаком ¬, конструкцией not.

Операция, выражаемая связкой *«и»,* называется **конъюнкцией** (лат. conjunctio — соединение) или логическим умножением и обозначается точкой « · » , знаками ∧ или **&,** конструкцией and.

Операция, выражаемая связкой *«или»* (в неисключающем смысле этого слова), называется **дизъюнкцией** (лат. disjunctio — разделение) или логическим сложением и обозначается знаком ∨ или плюсом. конструкцией or.

Исключающее или (или одно или только другое) обозначается конструкцией xor.

Используя эти операции и скобки можно строить очень сложные логические формулы.

О. Предикат — логическое выражение, логическое значение которого зависит не только от логических операций и значений высказываний, но и от значений обычных переменных.

Использование предикатов дает возможность строить логические формулы с использованием обычных переменных. Например : (X>5) and (Y<2).

В Object Pascal определены следующие логические операции:

not - логическое НЕ;

and - логическое И;

or - логическое ИЛИ;

xor - исключительное ИЛИ.

Логические операции применимы к операндам целого и логического типов. Если операнды - целые числа, то результат логической операции есть тоже целое число, биты которого (двоичные разряды) формируются из битов операндов по правилам, указанным в табл. 2.

Таблица 2

Логические операции над данными целого типа (поразрядно) 5

Операнд 1	**Операнд 2**	**not**	**and**	**or**	**xor**
1	-	0	-	-	-
0	-	1	-	-	-
0	0	-	0	0	0
0	1	-	0	1	1
1	0	-	0	1	1
1	1	-	1	1	0

К логическим же в Object Pascal обычно относятся и две сдвиговые операции над целыми числами:

i **shl** j - сдвиг содержимого / на *j* разрядов влево; освободившиеся младшие разряды заполняются нулями;

i **shr** j - сдвиг содержимого *i* на *j* разрядов вправо; освободившиеся старшие разряды заполняются нулями.

В этих операциях *i* и у - выражения любого целого типа.

Логические операции над логическими данными дают результат логического типа по правилам, указанным в табл. 3.

Таблица 3

Логические операции над данными логического типа

Операнд 1	Операнд 2	not	and	or	xor
True	-	False	-	-	-
False	-	True	-	-	-
False	False	-	False	False	False
False	True	-	False	True	True
True	False	-	False	True	True
True	True	-	True	True	False

Логический тип данных

В Object Pascal есть несколько логических типов данных. Наиболее популярный — тип boolean, который имеет 2 варианта значений истина и ложь, которые обозначаются как True и False. В других логических типах число вариантов больше 2, но на практике они используются редко. Значение логического типа принимают логические выражения типа 3>5, (x<2) and (y>0). Во многих операторах такие выражения обязательно используются и определяют логику их работы.

28.Эффективность и скорость алгоритмов. Вспомогательные алгоритмы.

Сложность алгоритма, оценка сложности алгоритма. Понятие о полиномиальных и реально выполнимых алгоритмах.

Известно, что правильность — далеко не единственное качество, которым должна обладать хорошая программа. Одним из важнейших является эффективность. Эффективность – сравнительная характеристика алгоритма и определяется неким критерием эффективности. Это может быть время выполнения алгоритма, число операций процессора, размер памяти используемой программы и т.д.. Этот критерий определяется как зависимость или функция от различных входных данных (параметра N). В тоже время можно говорить о эффективности алгоритма и сложности задачи. Можно сравнивать разные алгоритмы для одной задачи, а можно для разных задач. Поэтому принято говорить о сложности или скорости алгоритма и о сложности задачи. Так как эти величины есть функции, то можно их сравнивать по асимптотическим оценкам при стремлении параметра N->∞. Нахождение точной зависимости критерия эффективности для конкретной программы — задача достаточно сложная. По этой причине обычно ограничиваются **асимптотическими оценками** этой функции, то есть описанием ее примерного поведения при больших значениях параметра N. При этом для асимптотических оценок используют традиционное отношение O (читается "O большое") между двумя функциями f(n)=O(n). Асимптотические оценки функции при N->∞ можно представить выражением O(f(N)), которое означает const* f(N). Пусть заданы 2 оценки O(f(N)) и O(g(N)). Будем считать, что:

Функция f(N) растет медленнее g(N), если $\lim_{N\to\infty}$ [O(f(N)) / O(g(N))] = 0.

Функция f(N) растет быстрее g(N), если $\lim_{N\to\infty}$ [O(g(N)) / O(f(N))] = 0.

Функции f(N) и g(N) имеют одинаковую скорость роста, если $\lim_{N\to\infty}$ [O(f(N)) / O(g(N))] = const≠0.

Аналогично по асимптотике роста критерия эффективности можно говорить о скоро-

сти роста алгоритма. Причем, чем больше скорость роста алгоритма, тем он фактически медленнее и наоборот. Соответственно по функции асимптотики можно различать линейные O(N), квадратичные $O(N^2)$, кубические $O(N^3)$ экспоненциальные $O(e^N)$ и др. алгоритмы.

Сложность задачи можно оценить по скорости наиболее эффективного для нее алгоритма. При этом, может оказаться, что наиболее эффективный алгоритм нам пока не известен. Поэтому реальная сложность задачи может оказаться ниже чем представляемая нами, но не может оказаться больше чем текущая.

В качестве примера рассмотрим алгоритм нахождения факториала числа. Легко видеть, что количество операций, которые должны быть выполнены для нахождения факториала N! числа N в первом приближении прямо пропорционально этому числу, ибо количество повторений цикла (итераций) при вычислении по формуле N!=(N-1)!*N программе равно N. В подобной ситуации принято говорить, что алгоритм имеет ***линейную сложность*** (сложность O(N)). Можно ли вычислить факториал быстрее? Оказывается, да. Можно написать такую программу, которая будет давать правильный результат для тех же значений N с логарифмической скоростью. Про алгоритмы, в которых количество операций примерно пропорционально log(N) (в информатике обычно используют для основания двоичный логарифм) говорят, что они имеет ***логарифмическую сложность*** (O(log(N)).

Полиномиальным алгоритмом называют алгоритм, скорость которого может быть представлена полиномом какой-либо конечной степени.

Реально-выполнимым алгоритмом называют алгоритм, у которого время решения задачи возможно для реального использования на практике. Так как время решения зависит не только от алгоритма, но и от задачи (то есть от N), то некоторые задачи могут быть решены при малых N, но не решаются для больших значений N.

Когда начинающие программисты тестируют свои программы, то значения параметров, от которых они зависят, обычно невелики. Поэтому даже если при написании программы был применен неэффективный алгоритм, это может остаться незамеченным. Однако, если подобную программу попытаться применить в реальных условиях, то ее практическая непригодность проявится незамедлительно.

С увеличением быстродействия компьютеров возрастают и значения параметров, для которых работа того или иного алгоритма завершается за приемлемое время. Таким образом, увеличивается среднее значение величины N, и, следовательно, возрастает величина отношения времен выполнения быстрого и медленного алгоритмов. Исходя из практики использования различных алгоритмов для решения задач было принято использовать понятие реально выполнимых алгоритмов только для задач полиномиальной сложности и более простых. Таким образом, класс полиномиальных (P) алгоритмов совпадает с классом реально выполнимых алгоритмов.

Полиномиальные и не полиномиальные алгоритмы. Класс NP – алгоритмов

Задачи, которые решаются различными полиномиальными алгоритмами, очень разнообразны, но для информатики наиболее популярны задачи поиска и сортировки. Простой поиск в массиве реализуется алгоритмом линейной скорости от числа элементов массива O(N). Возможен вариант быстрого бинарного поиска, если массив был заранее отсортирован. Тогда скорость алгоритма выше полиномиальной ($O(\log_2(N))$).

Помимо класса полиномиальных алгоритмов (P – класса), могут существовать задачи не имеющие алгоритма решения полиномиальной скорости. Такие задачи принято называть NP задачами, а алгоритмы их решения NP- алгоритмами.

Доказана теорема о полноте класса NP задач:

Если существует полиномиальное решение хотя бы одной NP-задачи, то на его основе можно построить алгоритмы решения для любой NP – задачи. Таким образом, NP-задачи являются не полиномиальными только в совокупности. Это свидетельствует о полноте класса NP-задач.

Скорость роста NP-алгоритмов представляется либо как $O(e^N)$, $O(K^N)$ или O(N!) (в соответствии с формулой Стирлинга, их скорости близки и выше любой фиксированной

степени).

Примерами NP-задач являются задачи:

- Разложение числа на простые сомножители. Если считать за N – число сомножителей, то скорость роста алгоритма O(N!).
- Задача коммивояжера. В данной задаче заданы N городов и расстояния между ними, необходимо построить замкнутый путь, который удовлетворял условиям – минимальности суммарного расстояния, проходил бы через все города и только по одному разу. Фактически это задача поиска минимального гамильтонова пути в графе с заданными длинами ребер. Скорость роста такого алгоритма выше полиномиальной.

Поскольку к классу реально выполнимых алгоритмов относятся только полиномиальные и более быстрые алгоритмы, то NP-алгоритмы выполнимы только потенциально. Это означает, что при увеличении числа элементов в задаче время выполнения такого алгоритма быстро выходит за пределы возможностей его реализации. Поэтому для NP- задач очень важно правильно оценить требуемое количество операций для ее реального решения.

Вспомогательные алгоритмы

Одной из важнейших сравнительных характеристик алгоритма является рекурсивность — использование повторяющихся частей алгоритма (циклы, подпрограммы, рекурсия). Чем выше рекурсивность тем компактнее (меньше) запись алгоритма. Первоначально алгоритмизация рекурсивность организовывала с помощью операторов переходов, затем появились циклы и наконец попрограммы в виде процедур и функций. Этот подход стали считать наиболее эффективным, что привело к созданию библиотек подпрограмм, модулей, модульному принципу программирования. С появлением ООП появились так же классы и методы, которые так же формировали библиотеки и пакеты классов.

В алгоритмизации подпрограммы принято называть вспомогательными алгоритмами. Вспомогательный алгоритм — часть алгоритма, выделенная в обособленный блок, имеющий собственное имя и набор параметров (и значение для функции) для обмена данными с основным алгоритмом. Блок-схема вспомогательного алгоритма оформляется и записывается отдельно, начинается с блока Начало и завершается блоком Возврат (так как при завершении управление возвращается в место вызова вспомогательного алгоритма). В месте вызова вспомогательного алгоритма в блок-схеме используются специальный блок-прямоугольник с двойными боковыми линиями. В языке блок-схем есть специальный метод записи вспомогательных алгоритмов — с помощью блока возврат как аналог отдельного алгоритма и блока предопределенного процесса (прямоугольник с двойными боковыми линиями).

Модульный принцип структурного программирования требует при алгоритмизации сложных задач проводить модульную декомпозицию — выделение вспомогательных алгоритмов. С появлением ООП появились так же объектная декомпозиция, при которой выделяются объекты и их классы, которые формируют наследственное дерево классов.

29.Языки и средства программирования.

Методы и средства программирования. Классификация языков программирования.

Основы программирования

Программирование — процесс создания структуры программы, который происходит по следующим этапам:

- Постановка задачи (выделяются цель, средства ее достижения и условия решения задачи).
- Создание алгоритма решения задачи.
- Запись алгоритма в виде программы на языке программирования.
- Отладка программы.

- Тестирование программы.

Разработка программы состоит из двух различных действий – создания алгоритма и представления его в виде программы. При этом, создание алгоритма – это, как правило, наиболее сложный этап, т.к. создать алгоритм – значит, найти метод решения задачи. Поэтому, чтобы понять, как создать алгоритм, необходимо понять процесс решения задачи.

Тестирование алгоритма – это проверка правильности или неправильности работы алгоритма на специально заданных *тестах* или тестовых примерах – задачах с известными входными данными и результатами (иногда достаточны их приближения). Тестовый набор должен быть минимальным и полным, то есть обеспечивающим проверку каждого отдельного типа наборов входных данных, особенно исключительных случаев.

Трассировка – это метод пошаговой фиксации динамического состояния алгоритма на некотором *тесте*. *Трассировка* облегчает отладку и понимание алгоритма.

Процесс поиска и исправления (явных или неявных) ошибок в алгоритме называется **отладкой алгоритма**. Как правило отладка происходит совместно с процессом **трансляции** программы (преобразования из кода языка программирования в команды процессора ЭВМ). Трансляция выполняется либо компилятором, либо интерпретатором языка программирования.

Перед программированием задачи может быть составлена графическая **блок-схема алгоритма**, которая составляется по специальным правилам из специальных графических блоков.

По одной из классификаций методы программирования делятся на следующие типы:

- Операциональные
- Процедурные, называемые также директивные (directive) или императивными (imperative),
- Декларативные (declarative) языки,
- Объектно-ориентированные (object-oriented).

Операциональное программирование обычно рассматривается на примере машинно-ориентированного языка Ассемблер (хотя оно присутствовало и в первых языках высокого уровня). **Суть этого метода программирования — представлять программу как последовательный набор операций исполнителя (как правило — процессор ЭВМ).**

Замечание: При создании компьютерных средств программирования исторически первым явилось использование программ написанных в двоичных кодах (программирование в кодах). В дальнейшем были созданы первые ассемблеры, в которых были специальные текстовые конструкции (команды), которые требовалось переводить (транслировать) в двоичные коды. Так появились трансляторы языков программирования (интерпретаторы и компиляторы). Последующее развитие операционального программирование встретило большие трудности, которые были разрешены появлением декларативного и структурного подхода. Теоретическое развитие структурного подхода (Вирт, Якопини, Тьюринг, Бем) позволило создать процедурные зыки (классическим примером является язык Паскаль, созданный Н.Виртом как образец языка структурного/процедурного программирования).

Языки программирования делятся на следующие типы:

- Машинно-ориентированные языки и языки высокого уровня
- Операциональные, Процедурные, называемые также директивные (directive) или императивными (imperative), Декларативные (declarative) языки, Объектно-ориентированные (object-oriented).
- По сферам применения языки общего назначения (C, Pascal, Basic, Fortran, Prolog, Lisp, Java), языки работы с БД (SQL,QBE,FoxPro), языки моделирования (UML,GPSS), языки разработки скриптов (PHP, JavaScript) и другие.

Операциональное программирование обычно рассматривается на примере машинно-ориентированного языка Ассемблер (хотя оно присутствовало и в первых языках высокого уровня). Машинно-ориентированность языка Ассемблер отражалась в учете особенностей архитектуры конкретного процессора, что делало невозможным перенос программа (без корректировки) с одного типа ЭВМ на другой. Поэтому появились операциональные языки высокого уровня, синтаксис которых не зависел от особенностей архитектуры ЭВМ (машинно-неза-

висимые языки).

Замечание: При создании компьютерных средств программирования исторически первым явилось использование программ написанных в двоичных кодах (программирование в кодах). В дальнейшем были созданы первые ассемблеры, в которых были специальные текстовые конструкции (команды), которые требовалось переводить (транслировать) в двоичные коды. Так появились трансляторы языков программирования (интерпретаторы и компиляторы). Последующее развитие операционального программирование встретило большие трудности, которые были разрешены появлением декларативного и структурного подхода.

К процедурным языкам относятся такие классические языки программирования, как Algol, Fortran, Basic, Pascal, C. **Суть этого метода программирования — представлять программу в виде структуры процедур исполнителя (как правило — транслятор языка, который преобразует эти процедуры в последовательность операций ЭВМ).**

Наиболее существенными классами декларативных языков являются **функциональные** (functional), и реляционные - **логические** (logic) языки. К категории функциональных языков относятся, например, Lisp и Haskell. Самым известным языком логического программирования является Prolog (Пролог). **Суть этого метода программирования — представлять программу как описание (декларацию) задачи с указанием требуемой цели (целевое указание). Для достижения этой цели строится автоматически алгоритм с помощью специального ядра системы, освобождая тем самым программиста от рутинных операций.**

Среди объектно-ориентированных языков программирования (языков ООП) отметим C++, Java, Object Pascal, Python. **Суть этого метода программирования — представлять программу как нобор взаимодействующих объектов, обладающих собственными встроенными алгоритмами — методами. Для взаимодействия объектов проектируется некая среда, функционирующая автономно и управляемая с помощью событийного управления (обработки событий в среде).**

Особенность операционального программирования – ориентация на построение инструкции для исполнителя – процессора ЭВМ в терминах его операций. Главный недостаток – сложность программ, злоупотребление оператором перехода GOTO.

Особенность процедурного программирования – переход к более строгому, логически структурному программированию. Основные принципы процедурного программирования – использование стандартных технологий проектирования, структурного и модульного принципов, исключение GOTO, максимальное использование стандартных алгоритмов. Главный недостаток – постепенное загромождение модулей процедурами и функциями, недостаточная читабельность и понимаемость больших программ.

Особенность декларативного программирования – программист только описывает (создает декларацию) задачу, а методы и средства ее решения подбирает специальная программа – ядро языка программирования. Главный недостаток – недостаточное развитие среды и средств программирования для данных языков.

Особенность ООП – программист описывает структуру классов для объектов (создает иерахию классов по технологии наследования) и использует специальную среду проекта (как правило предоставляемую системой визуального программирования), затем описывает методы классов и обработчики событий для среды и классов/объектов. Главный недостаток – необходимость использования специальной технологии объектной декомпозиции, которая требует специальных навыков и знаний от программиста.

30.Особенности языка Паскаль. Структура программы и ее разделы. Ввод и вывод данных.

Появление языка программирования Паскаль. Особенности языка Паскаль

Теоретическое развитие структурного подхода (Вирт, Якопини, Тьюринг, Бем) позволило создать процедурные зыки. Классическим примером структурного языка является процедурный язык Паскаль, созданный Н.Виртом как образец языка структурного/процедурного программирования. Этот язык был разработан специально для «правильного» обучения струк-

турному программированию, что обеспечило его широкое распространение в сфере образования. Таким образом, Basic остался языком упрощенным, обычно реализуемым в простейших системах. Язык С стал языком для профессиональных программистов и энтузиастов программного «творчества», так как он дает программисту гораздо больше свободы, но зато и требует гораздо больше знаний и усилий. Паскаль остался языком для строгого структурного программирования. Сегодня в области образования он часто конкурирует с Python, который тоже ориентирован на обучение. Долгое время Паскаль и С конкурировали на равных, но прорыв С++ в объектно-ориентированную сферу программирования дал ему преимущество. Сегодня в ООП языках общего назначения порядок популярности C#, Java, Object Pascal.

В данном курсе изучается язык Object Pascal, который реализован в ряде сред программирования (Delphi, Kylix, Lazarus). **Object Pascal** («Объектный Паскаль») — язык программирования, разработанный в фирме Apple Computer в 1986 году группой Ларри Теслера, который консультировался с Никлаусом Виртом. Этот вариант практически Паскаля практически полностью поддерживает стандартный Паскаль и расширяет его возможности.

Особенности стандартного Паскаля:

- Строгая последовательность разделов программы и модулей.
- Любой компонент должен быть описан до его использования, неописанные переменные или константы не разрешены.
- Строгая типизация данных.
- Минимальный набор стандартных операций, строгие требования к виду выражений и записи операторов.

В Object Pascal некоторые из этих требований не выполняются.

Структура программы и ее разделы. Ввод и вывод данных

Программа Паскаля разделена на ряд обязательных и не обязательных разделов, которые как правило строго следуют друг за другом:

Program имя программы; (обязательный, задает имя программы и файла)
Uses имена подключаемых модулей; (необязательный, определяет модули)
Label имена меток; (необязательный, сейчас не используется)
Const имена констант; (необязательный, не очень удобный)
Type имена пользовательских типов данных; (необязательный)
Var имена переменных; (необязательный, но всегда есть)
процедуры и функции пользователя
begin (обязательный блок, начало основной программы)
… (необязательные операторы программы)
….(необязательные операторы программы)
end. (обязательный оператор, конец основной программы, должна быть точка)

Стиль структурного программирования требует, чтобы меток не было, основная программа имела минимальный код (текст программы). Все нужно выносить в процедуры и функции пользователя, а лучше во внешние модули. Так в Delphi основная программа генерируется автоматически, а программист работает только с модулями.

Самые необходимые операторы программы — операторы ввода/вывода информации. В Паскале это операторы Write/Writeln и Read/Readln. Однако в Delphi они используются только для файлов и при создании консольного приложения. В остальных случаях ввод/вывод производится с помощью компонентов проекта.

Оператор присваивания и составной оператор

Составной оператор - это последовательность произвольных операторов программы, заключенная в операторные скобки - зарезервированные слова **begin ... end** . Составные операторы - важный инструмент Object Pascal, дающий возможность писать программы по современной технологии структурного программирования (без операторов перехода goto). Object Pascal не накладывает никаких ограничений на характер операторов, входящих в составной оператор. Среди них могут быть и другие составные операторы - язык Object Pascal допускает произвольную глубину их вложенности.

Оператор присваивания := обеспечивает присваивание переменной или выражению конкретное значение. Нужно отличать этот оператор от операции сравнения = которая не меняет значение, а только сравнивает.