**+МДК 05.02 Разработка кода информационных систем**

**Вопросы по теории (обязательно с примерами!):**

1. Рекурсия и рекурсивные алгоритмы Прямая рекурсия, косвенная рекурсия. Рекурсивный спуск и рекурсивный подъем. Рекурсивная триада. Глубина рекурсии, объем рекурсии. Сравнение рекурсии и итерации.

**Рекурсия (в широком смысле)** – это определение объекта посредством ссылки на себя.

**Рекурсия** – это свойство алгоритмической системы на промежуточных этапах своего функционирования создавать другие системы, включая идентичные себе самой, и использовать результаты их функционирования в дальнейшей работе

**Рекурсивный алгоритм** – это алгоритм, в определении которого содержится прямой или косвенный вызов этого же алгоритма.

**Прямая рекурсия** – это обращение функции к самой себе предполагает, что в теле функции содержится вызов этой же функции, но с другим набором фактических параметров

**Косвенная рекурсия** – это функция содержит вызовы других функций из своего тела. При этом одна или несколько из вызываемых функций на определенном этапе обращаются к исходной функции с измененным набором входных параметров.

**Рекурсивный спуск** – это процесс, когда рекурсивная функция последовательно вызывает саму себя с изменяющимися параметрами, пока не достигнет базового случая.

**Рекурсивный подъем** – это процесс, когда рекурсивная функция возвращает значения на каждом уровне рекурсии до тех пор, пока не завершит все рекурсивные вызовы и не вернет финальный результат

**Рекурсивная триада** включает в себя:

1) параметризация – выделяют параметры, которые используются для описания условия задачи, а затем в решении;

2) база рекурсии – определяют тривиальный случай, при котором решение очевидно, то есть не требуется обращение функции к себе;

3) декомпозиция – выражают общий случай через более простые подзадачи с измененными параметрами

**Глубина рекурсивных вызовов** – наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений.

**Объем рекурсии** – количество вершин полного рекурсивного дерева без единицы



1. Фракталы. Определение, свойства. Применение. Примеры построения.

**Фрактал** — это сложная, бесконечно самоподобная геометрическая фигура, каждый фрагмент которой повторяется при уменьшении масштаба.

Фракталы обладают тремя ключевыми свойствами: самоподобием, дробной размерностью и бесконечной сложностью.

Фракталы применяются в разных областях науки, техники и искусства.

**Применение в науке**

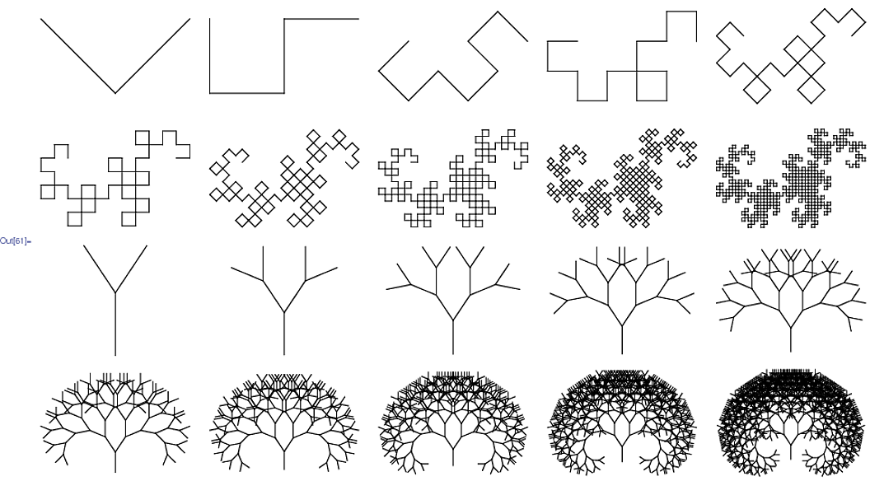
* Моделирование сложных систем (формы гор, облаков или деревьев)
* Изучение турбулентности и хаоса.
* Визуализация данных.

**Применение в технике**

* Разработка антенн.
* Фрактальное сжатие изображений.

**Применение в искусстве**

* Цифровое искусство.
* Архитектура.
* Музыка.



1. Записи. Описание записи. Обращение к полям записи. Записи при работе с файлами.

**Записи – это структурированный комбинированный тип данных.** Запись состоит из определенного числа компонент,**называемых полями, которые могут быть разного типа.**

Общий вид описания типа record в Паскаль:

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** <имя\_записи> = **record**  <имя\_поля1>:<тип\_поля1>;  <имя\_поля2>:<тип\_поля2>;  ...  **end**; |

Обращение к полям записи:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **type** mydate = **record**  month: 1..12;  day: 1..31;  year: **integer**  **end**;  **var** d: mydate;  **begin**  d.day:=1;  d.month:=12;  readln(d.year);  writeln(d.day,'/',d.month,'/',d.year);  **end**. |

Записи при работе с файлами:

var f: Text;

begin

Assign(f, 'file.txt');

Rewrite(f); // Создание файла для записи

Writeln(f, 'Строка'); // Запись строки в файл

Close(f);

Reset(f); // Открытие для чтения

while not Eof(f) do // Пока не конец файла

begin

Readln(f, str); // Чтение строки из файла

Writeln(str); // Вывод на экран

end;

Close(f);

end;

1. Множества. Действия над множествами.

Множества и операция IN в Паскале

Операция in необходима для поиска определенного элемента в величине типа set, т.е. в множестве.

Так, если x есть элемент множества a, то (x in a) дает true.

**Пример:** создать перечисляемый тип из дней недели и множество из рабочих дней недели (**базовым типом которого является перечисляемый тип** дней недели). Определить, является ли понедельник рабочим днем.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **type**  week\_days=(Mon, Tue, Wed, Thu, Fri);  **var**  work\_days: **set** **of** week\_days;  **begin**  work\_days:=[Mon, Wed];  **if** Mon **in** work\_days **then**  writeln ('понедельник - рабочий день')  **else**  writeln ('понедельник - не рабочий день')  **end**. |

**Пример:** Тот же пример решить с использованием типа byte в качестве базового типа.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **var**  work\_days,days\_off: **set** **of** **byte**;  **begin**  work\_days:=[1, 2];  days\_off:=[6, 7];  **if** 1 **in** work\_days **then**  writeln ('понедельник - рабочий день')  **else**  writeln ('понедельник - не рабочий день');  **if** 6 **in** days\_off **then**  writeln ('суббота - выходной день')  **else**  writeln ('суббота - не выходной день');  **end**. |

5 Динамическая память. Динамическое распределение памяти. Размещение объектов в куче. Освобождение памяти.

**О динамических данных**

размер динамических данных заранее неизвестен;

память под них выделяется во время исполнения программы;

динамические данные, как правило, не имеют идентификаторов (имени в стандартном понимании статических данных), но имеют адрес в памяти;

обращение к динамическим данным осуществляется по их адресу в памяти.

**О памяти**

Если говорить об оперативной памяти в программировании, то вводится понятие статической и динамической памяти. Оперативная память неодинакова, она имеет различные области: для статической памяти, для динамической памяти.

Обращение к участку динамической памяти происходит с помощью специальной ссылочной переменной, которая называется указателем или ссылкой.

Указатели в Паскале необходимы при работе с динамической памятью.

Переменная типа «указатель» в качестве своего значения содержит адрес участка динамической памяти, с которой связан этот указатель.

**Размещение объектов в куче (heap)**

Куча (heap) — область оперативной памяти, из которой выделяется динамическая память

Каждое динамически созданное значение размещается в куче

Для работы с такими данными обычно используют указатели

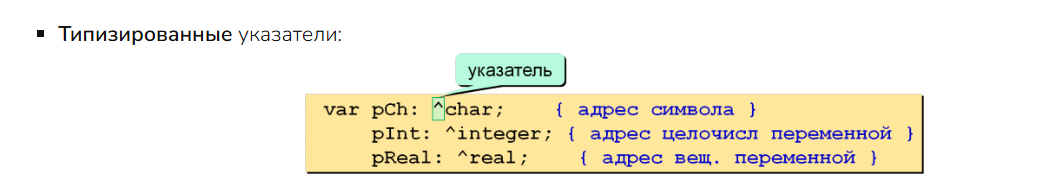
В языке программирования Паскаль управление памятью, в частности освобождение памяти, осуществляется с помощью динамического выделения и освобождения памяти. Динамическое выделение позволяет создавать переменные во время выполнения программы, что особенно полезно при работе с массивами или структурами переменной длины.

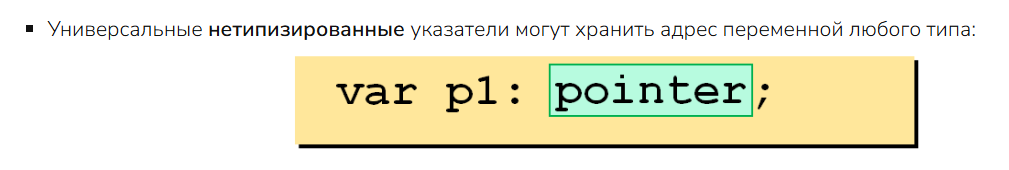
Для освобождения динамически выделенной памяти используется оператор Dispose. Это важно делать, чтобы избежать утечек памяти.

6. Понятие указатель. Использование типизированных и не типизированных указателей. Проблемы использования указателей. Утечка памяти.

Указатели в Паскале необходимы при работе с динамической памятью.

Переменная типа «указатель» в качестве своего значения содержит адрес участка динамической памяти, с которой связан этот указатель.





**Проблемы:**

Неверные указатели (Dangling Pointers): Это указатели, которые ссылаются на память, которая была освобождена или невалидна. Например, если вы освобождаете память, но не обнуляете указатель, он все еще будет указывать на старое место в памяти, что может привести к неопределенному поведению при попытке доступа к этой памяти.

Сложность отладки: Программы с указателями могут быть сложнее для отладки, поскольку ошибки, связанные с памятью, часто приводят к неопределенному поведению, которое может проявляться не сразу.

Неинициализированные указатели: Если указатель не инициализирован перед использованием, он будет указывать на случайное место в памяти. Это может привести к неожиданным сбоям программы или повреждению данных.

**Утечка памяти** — это ситуация, когда динамически выделенная память не освобождается, даже если она больше не используется.

В результате программа постепенно потребляет всё больше ОЗУ, что может привести к замедлению, сбоям или аварийному завершению.

~~ну а вообще, самая большая проблема — это некомпетентные программисты (мы)~~

7. Организация элементарных структур на основе динамической памяти. Очередь, Стек, Дек, Списки (односвязные, двусвязные, линейные, кольцевые).

**Очередь** — динамическая структура данных, у которой в каждый момент времени доступны только два элемента: первый и последний. Добавление элементов возможно только с одного конца (конца очереди), а удаление элементов – только с другого конца (начала очереди).

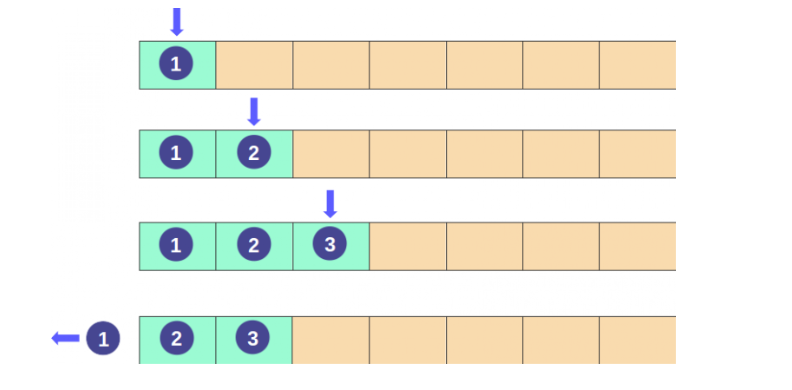
Существует сокращение для очереди: FIFO = First In – First Out, с английского — «Кто первым вошел, тот первым вышел».

Для очереди доступны следующие операции:

* добавить элемент в конец очереди (PushTail);
* удалить элемент с начала очереди (Pop).

Работа с очередью обычным массивом:

Это достаточно простой способ, который подразумевает два неблагоприятных момента: заблаговременное выделение массива, сдвиг элементов при удалении из очереди.

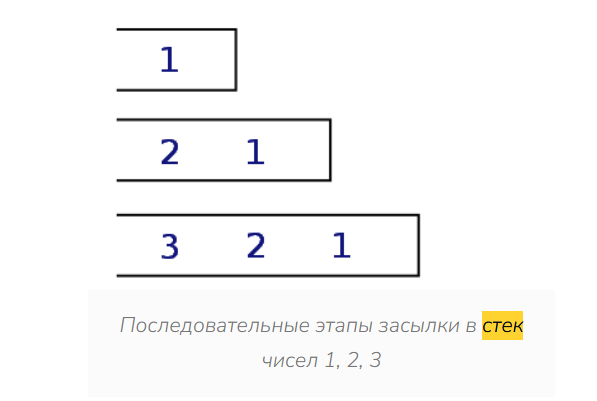


**Стек** — динамическая структура данных, в которой добавление и удаление элементов доступно только с одного конца (с верхнего (последнего) элемента).

Существуют такие сокращения:

LIFO = Last In -> First Out — с английского языка «Кто последним вошел, тот первым вышел»

FILO = First In -> Last Out — «первым вошел, последним вышел»



добавление в стек нового элемента;

add(<нач\_стека>,<новый\_элемент>):<нач\_стека>

определение пуст ли стек;

empty(<нач\_стека>):boolean

доступ к последнему включенному элементу, вершине стека;

take(<нач\_стека>):<тип\_элементов\_стека>

исключение из стека последнего включенного элемента.

del(<нач\_стека>):<нач\_стека>

**Дек** — англ. double ended queue, т.е. очередь с двумя концами – это динамическая структура данных, добавлять и удалять элементы в которой можно с обоих концов.

Основные особенности:

Комбинирует свойства стека (LIFO) и очереди (FIFO)

Быстрое добавление/удаление: O(1) по времени с обеих сторон

**Список** состоит из конечного множества динамических элементов, размещающихся в разных областях памяти. Благодаря указателям элементы списка объединены в логически упорядоченную последовательность.

Каждый элемент списка организован следующим образом:

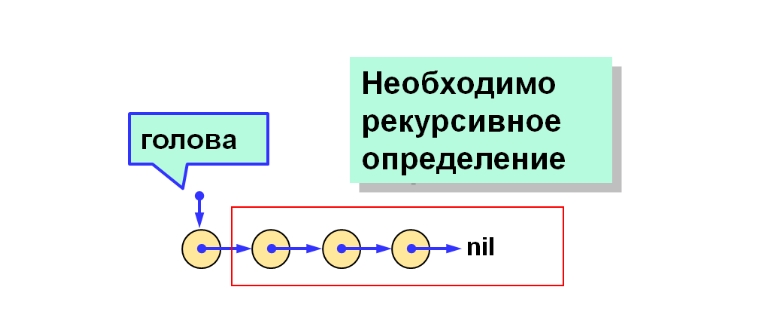
* содержит поля с полезной информацией;
* содержит специальное поле (или несколько полей), которое хранит адрес другого элемента списка.

**Односвязный список**

Список состоит из начального узла — головы — и связанного с ним списка.

Каждый элемент списка содержит информационную и ссылочную части. Т.е. каждый элемент имеет информационное поле (поля) — полезная информация — и ссылку (ссылки), то есть адрес на другой элемент списка.

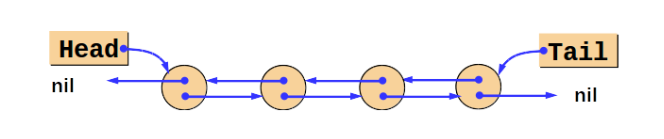
Односвязный (линейный) список: структура, каждый элемент которой «знает» адрес только следующего за ним элемента.



**Двусвязные списки (двунаправленные)**

Двусвязный список позволяет двигаться в обоих направлениях.

При работе с двусвязными списками создаются два указателя.



Линейный список — это абстрактная структура данных, в которой:

* Элементы расположены в строгом порядке
* У каждого элемента есть предшественник и/или последователь
* Поддерживаются операции вставки, удаления, доступа по позиции

линейный список — это общее понятие, а односвязный список — частный случай линейного.

**Кольцевой список** — это разновидность связного списка (односвязного или двусвязного), в котором последний элемент указывает на первый, образуя цикл.

Виды кольцевых списков:

Односвязный кольцевой список

* Каждый узел содержит ссылку только на следующий узел
* Последний узел ссылается на первый (голову)

Двусвязный кольцевой список

* Каждый узел содержит ссылки на следующий и предыдущий узел
* Первый и последний узлы связаны между собой в обе стороны
  1. Организация элементарных структур на основе динамической памяти. Деревья.

Динамическая память позволяет создавать сложные структуры данных, размер которых может изменяться во время выполнения программы. Одной из ключевых структур, эффективно использующих динамическую память, являются **деревья**. **Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.**

**Корень – это начальный узел дерева.**

**Лист – это узел, из которого не выходит ни одной дуги.**

**Двоичное (бинарное) дерево – это дерево, в котором каждый узел имеет не более двух сыновей.**

**Применение:**

* 1. **поиск данных в специально построенных деревьях  
     (базы данных);**
  2. **сортировка данных;**
  3. **вычисление арифметических выражений;**
  4. **кодирование (метод Хаффмана).**
  5. Парадигмы программирования. Языки парадигм программирования.

**Парадигма программирования** — это определённый стиль или подход к написанию программ, который задаёт, как разработчики описывают алгоритмы и структуры данных. В основе парадигм лежат концепции, которые определяют способы мышления о программировании, подходы к решению задач и построению программного обеспечения.

**Виды парадигм**

Некоторые виды парадигм программирования:

* **Императивное программирование**. Программа описывает последовательность инструкций, которые компьютер должен выполнить для достижения цели.
* **Декларативное программирование**. Акцент делается на том, что нужно достичь, а не на том, как это сделать. Программист описывает желаемое состояние или результат, а детали реализации скрыты.
* **Объектно-ориентированное программирование (ООП)**. Программа строится вокруг объектов, которые являются экземплярами классов. Каждый объект объединяет данные и поведение, с ними связанное.
* **Функциональное программирование**. Фокусируется на вычислениях как на применении функций, избегая изменения состояния программы и побочных эффектов.
* **Логическое программирование**. Программа описывается в виде логических утверждений, а вычисления происходят через логический вывод.
* **Реактивное программирование**. Программы реагируют на изменения данных или событий. Применяется для разработки приложений с асинхронной обработкой данных и взаимодействием с внешними источниками. Примеры использования

Каждая парадигма подходит для определённых задач. Например:

* **Императивное программирование** используют для работы с анимацией, графическими интерфейсами, скриптами, играми.
* **Декларативное программирование** подходит для вычислений и работы с данными.
* **ООП** помогает структурировать код, делает его более понятным и поддерживаемым.
* **Функциональное программирование** способствует созданию предсказуемого и тестируемого кода.
  1. Интегрированная среда разработки приложений Lazarus. Структура проекта Lazarus, элементы языка.

Lazarus – это среда объектно- Ориентированного программирования (аналог Borland Delphi)

Основные составные части Lazarus

1 Дизайнер Форм (Form Designer)

2 Окно Редактора Исходного Текста (Editor Window)

3 Палитра Компонент (Component Palette)

4 Инспектор Объектов (Object Inspector)

5 Справочник (On-line help)

Лазарус по своей сути полностью похож на паскаль, и содержит те же эелементы языка.

11.Модули в Lazarus. Структуры и принципы организации программных модулей.

**Виды модулей в Lazarus**

1. **Модули форм** (связаны с .lfm-файлами) — содержат код формы и её компонентов.
2. **Библиотечные модули** — общие функции (например, для работы с базами данных).
3. **Модули классов** — реализация ООП-логики

Модули — это фундаментальные строительные блоки программ на **Object Pascal** в среде **Lazarus**. Они позволяют разбивать код на логические части, упрощая разработку и поддержку приложений. Для того чтобы использовать встроенный модуль, то достаточно просто объявить его. Чтобы использовать модуль, созданный вами, добавьте его в одну папку с текущей программой и также обьявите его. Убедитесь, что обьявленный модуль и название самого модуля совпадают.

12. Событийно-ориентированное программирование. Основные компоненты в Lazarus и их характеристики.

1. Форма (TForm) – основа GUI-приложения, контейнер для других компонентов.
2. Меню (TMainMenu) – позволяет вам поместить главное меню в программу (как File, Edit).
3. Меню (TPopupMenu) – позволяет создавать всплывающие меню.
4. Метка (TLabel) – служит для отображения текста на экране.
5. Поле ввода (TEdit) – стандартный управляющий элемент Windows для ввода. Он может быть использован для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы.
6. Поле ввода (TMemo) – иная форма TEdit. Подразумевает работу с большими текстами.
7. Кнопка (TButton) – позволяет выполнить какие-либо действия при нажатии кнопки во время выполнения программы (OnClick).
8. Список (TListBox, TComboBox) – отображает список элементов.
9. Чекбокс (TCheckBox) – отображает строку текста с маленьким окошком рядом. В окошке можно поставить отметку, которая означает, что что-то выбрано.
10. Радиокнопка (TRadioButton) позволяет выбрать только одну опцию из нескольких.
11. Панель (TPanel) – управляющий элемент, используется в декоративных целях.
12. Кнопка (TBitBtn) – кнопка вроде TButton, однако на ней можно разместить картинку (glyph). TBitBtn имеет несколько предопределенных типов (bkClose, bkOK и др), при выборе которых кнопка принимает соответствующий вид.
13. Изображение (TImage) – отображает графическое изображение на форме. Воспринимает форматы BMP, ICO, WMF. Если картинку подключить во время дизайна программы, то она прикомпилируется к EXE файлу.

ОСНОВНЫЕ ОБЩИЕ СВОЙСТВА:

* Height, Width - высота и ширина компонента в пикселях;
* Align - положение компонента относительно формы: alTop (сверху), alBotton (снизу), alLeft (слева), alRight (справа), alNone (определяется свойствами Left и Top);
* Color – цвет фона компонента;
* Font - внешний вид (размер, цвет и т.д.) шрифта для надписей;
* Caption - связывает текстовую строку с управляющим компонентом: надпись на кнопке, текст метки, заголовок окна для формы;
* Техт - содержимое поля для текстового окна, которое будет видно при загрузке формы;
* Visible - если это свойство имеет значение True, то компонент будет видимым на форме, а если – False, то он будет невидимым;
* Enabled - если это свойство имеет значение False , то компонент недоступен, т.е. щелчок мыши по этому элементу не даст эффекта, при этом текст, размещенный на нем, будет серым. Если это свойство имеет значение True , то компонент можно будет использовать в процессе работы программы;
* AutoSize - если это свойство имеет значение True , то размеры оконного компонента будут автоматически меняться при изменении размера шрифта, или добавления в него неоконных компонентов;
* Hint ShowHint - если свойство ShowHint имеет значение True , то всплывает подсказка – текст, содержащийся в свойстве Hint;
* Focused - если это свойство имеет значение True , то данный компонент получил фокус;
* TabStop - если это свойство имеет значение True , то данный компонент получает фокус с помощью клавиши Tab, т.е. определяет, сможет ли пользователь переходить с помощью клавиши Tab к данному оконному элементу управления;
* TabOrder - порядковый номер выбора компонента клавишей Tab.

Основные события:

* OnClick - щелчок левой кнопкой мыши на объекте;
* OnKeyPress - нажимается и отпускается алфавитно-цифровая клавиша при условии, что объект находится в фокусе;
* OnCreate - вызывается при создании (загрузке) формы;
* OnClose - вызывается при закрытии формы;
* OnEnter – компонент получил фокус ввода;
* OnExit - компонент потерял фокус ввода;
* OnActivate – при активации формы (получении фокуса);
* OnShow – при отображении формы;
* OnChange – при изменении текста;
* OnTimer – срабатывает через заданный интервал;
* OnDblClick – при двойном клике.

13. Подпрограммы в Lazarus. Основные способы передачи параметров в подпрограмму, их сравнение.

Способы передачи параметров в подпрограмму:

1. По значению (параметр копируется в подпрограмму, изменения внутри подпрограммы не влияют на исходную переменную).

procedure MyProc(x: Integer); // По умолчанию передача по значению

1. По ссылке (работает с оригиналом переменной, изменения параметра сохраняются, переменная должна быть инициализирована перед вызовом)

procedure ProcName(var Param: Integer);

1. По константе (запрещает изменения параметра, гарантирует неизменность данных)

procedure Print(const Text: String);

1. Передача для вывода (параметр предназначен для возврата значения, начальное значение игнорируется, переменная может не инициализирована)

procedure ProcName(out Param: Integer);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Критерий | по значению | var | const | out |
| Изменчивость | локальная | да | нет | да |
| Влияние на оригинал | нет | да | нет | да |
| Инициализация | любая (может быть инициализирована или нет) | требуется | любая | не требуется |
| Эффективность | низкая(значительная разница для крупных объектов) | высокая | высокая | высокая |
| Типы данных | простые | любые | сложные | любые |
| Безопасность | высокая | низкая | высокая | средняя |

Таким образом, выбор способа передачи зависит от конкретных требований вашей программы.

14. Подпрограммы в Lazarus. Область видимости. Локальные и глобальные идентификаторы.

Область видимости (scope) определяет область программы, в которой определённая переменная доступна для использования. Правильное использование областей видимости помогает предотвратить конфликты именования, улучшает читаемость кода и облегчает его поддержку.

Идентификатор — это уникальное имя, используемое для обозначения объектов в программе.

Основные области видимости:

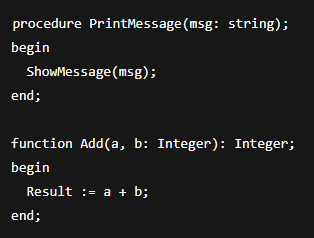
1. Глобальная. Идентификатор объявлен вне всех подпрограмм, обычно в разделе var основной программы (program) или в interface-секции модуля (unit). Виден во всей программе.
2. Локальная. Идентификатор объявлен внутри конкретной подпрограммы (в ее разделе var или const). Виден только внутри этой подпрограммы.
3. Область видимости модуля. Идентификатор объявлен в interface-секции модуля. Виден всем модулям и основной программе, которые подключают данный модуль через uses. Идентификатор объявленный в implementation-секции модуля виден только внутри этого модуля.

15. Подпрограммы в Lazarus. Фактические и формальные параметры

**Подпрограммы** — это процедурные или функциональные блоки кода, которые можно многократно использовать. В Lazarus (на языке Pascal) есть два основных типа подпрограмм:

* **Процедуры** (procedure) — выполняют действия, но не возвращают значение.
* **Функции** (function) — выполняют действия и возвращают результат.

**Пример:**

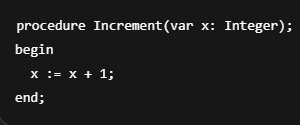


**Формальные параметры** — это переменные, определенные в заголовке подпрограммы (при объявлении).  
**Фактические параметры** — это значения или переменные, передаваемые при вызове подпрограммы.

**Виды передачи параметров:**

* by value (по значению) — копия значения.
* var (по ссылке) — передается ссылка на переменную.
* const — передача неизменяемого значения (оптимизация).

**Пример:**

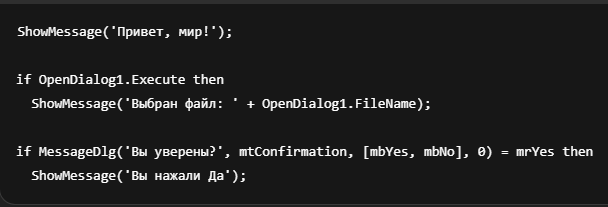


16. Возможности интегрированной среды разработки Lazarus по организации и использованию диалогов, диалоговых окон

Lazarus предоставляет визуальные средства для работы с **диалогами** (диалоговыми окнами), такими как:

* **MessageDlg** — стандартный диалог с кнопками.
* **ShowMessage** — простое информационное окно.
* **OpenDialog** — выбор файла.
* **SaveDialog** — сохранение файла.
* **ColorDialog** — выбор цвета.
* **FontDialog** — выбор шрифта.

**Примеры:**

**Добавление диалогов:**

1. Добавьте нужный компонент из палитры (например, TOpenDialog).
2. Настройте свойства в инспекторе объектов.
3. Используйте метод Execute для вызова.

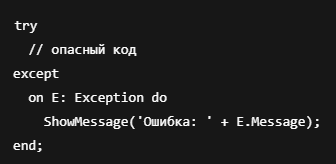
17. Технология отладки приложений в интегрированной среде разработки Lazarus. Исключительные ситуации. Структурная обработка исключений

**Отладка в Lazarus**:

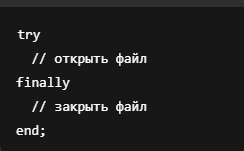
* Использование **точек останова** (breakpoints).
* **Пошаговое выполнение** (Step Into, Step Over).
* Просмотр **переменных** в отладчике.
* **Отладочные сообщения** через writeln или DebugLn.

**Исключительные ситуации (exceptions)** — ошибки времени выполнения, например, деление на ноль, выход за границы массива и т.д.

**Обработка исключений** осуществляется через конструкции:



**Конструкция try...finally** используется для гарантированного выполнения блока:



1. Организация работы с базами данных. Понятие БД, СУБД, виды СУБД. Модели данных. Типы связей. Ссылочная целостность. Механизм доступа к БД в Lazarus.

База данных (БД) — это специальным образом организованная совокупность данных, хранящаяся в электронном виде в компьютерной системе. Основная цель базы данных — хранение информации и её предоставление по требованию.

Система управления базами данных (СУБД) — это совокупность программ и языковых средств, предназначенных для управления данными в базе данных, ведения базы данных и обеспечения её взаимодействия с прикладными программами.

Виды СУБД:

Иерархическая. Данные объединены в древовидную структуру, в корневом элементе которой есть ссылки на узлы, расположенные на втором уровне иерархии, а в узлах второго — на узлы третьего, и так далее.

Сетевая. По способу записи элементов похожа на иерархическую, но связи можно строить между любыми узлами графа, а не только «сверху-вниз».

Реляционная. Данные представлены в виде таблиц, связанных между собой отношениями.

Объектно-ориентированная. Главное достоинство такой СУБД — «бесшовная» интеграция с современными подходами к написанию программ. Главный недостаток — низкая производительность.

Гибридная. Внутри такая база содержит обычные файлы реляционной базы данных, обёрнутые в специальную оболочку, предоставляющую объектный интерфейс.

Ссылочная целостность — это свойство базы данных, означающее, что она содержит полную и непротиворечивую информацию, адекватно отражающую предметную область. Целостность базы данных обеспечивается ограничениями на значения элементов данных, поддержкой заданных принципов взаимосвязи данных (ссылочная целостность), управлением транзакциями.

Механизм доступа к БД в Lazarus предполагает создание интерфейса для работы с базой данных типа SQLite, например, в среде разработки Lazarus можно создать интерфейс для работы с базой данных «Музыкальный магазин», состоящей из взаимосвязанных таблиц.

1. Организация работы с файлами. Характеристики файла. Классификация файлов. Алгоритм работы с файлами.

Организация работы с файлами в Lazarus включает несколько этапов:

Объявление файловой переменной. Например, через примеры: var F1 : file of Integer; (файл целых чисел), var F2 : file of TSeasons; (файл с данными пользовательского типа).

Связывание файловой переменной с файлом на диске. Для этого используется функция Assign, через первый параметр которой передаётся файловая переменная, а через второй — путь к файлу.

Открытие файла. Либо для записи (функция Rewrite), либо для чтения (функция Reset).

Запись или чтение файла. Для этого используются процедуры WriteLn для записи в файл и Read/ReadLn для чтения.

Закрытие файла. После завершения работы с файлом его нужно закрыть.

Классификация файлов в Lazarus включает три типа:

Текстовые файлы. Предназначены для работы с текстом.

Типизированные файлы. Могут хранить массив записей какого-то одного, указанного типа.

Нетипизированные файлы. Это файлы для побайтовой работы с данными любого типа. Могут использоваться для копирования или переноса файлов.

Алгоритм работы с файлами в Lazarus также включает использование функций Bof и Eof: первая возвращает истину, если указатель находится в начале файла, вторая — если в конце.

1. Организация работы с файлами. Основные операции для работы с последовательными файлами. Пример работы.

Организация работы с файлами в Lazarus включает следующие этапы:

Объявление файловой переменной. Например, F1 : file of Integer; (файл целых чисел) или F2 : file of TSeasons; (файл с данными пользовательского типа).

Связь файловой переменной с файлом на диске. Для этого используется функция Assign. Через первый параметр передаётся файловая переменная, а через второй — путь к файлу.

Открытие файла. Для записи используется функция Rewrite, а для чтения — Reset. При этом указатель устанавливается в начало файла, в нулевую позицию.

Запись или чтение данных. Для записи можно использовать процедуру WriteLn, которая записывает данные в стандартное устройство вывода (по умолчанию экран, а может быть и файл). Для чтения — процедуры Read или ReadLn.

Закрытие файла. Для этого используется процедура CloseFile, в которую передаётся имя файловой переменной.

Пример работы с файлами в Lazarus можно рассмотреть на задаче создания типизированного файла и записи туда n вещественных чисел. Алгоритм решения:

Открыть файл для записи с помощью оператора Rewrite.

Ввести значение n.

В цикле (при i, меняющемся от 1 до n) ввести очередное вещественное число a, которое сразу же записать в файл с помощью процедуры Write.

Закрыть файл с помощью процедуры CloseFile.

1. Организация работы с файлами. Основные операции для работы с файлами прямого доступа. Пример работы.

Основные операции для работы с файлами прямого доступа

Открытие/создание файла

AssignFile - связывает файловую переменную с именем файла

Rewrite - создает новый файл (если существует - перезаписывает)

Reset - открывает существующий файл для чтения/записи

Закрытие файла

CloseFile - закрывает файл

Операции записи

Write - записывает данные в текущую позицию

Операции чтения

Read - читает данные из текущей позиции

Дополнительные операции

Eof - проверка конца файла

Пример работы с файлами в Lazarus:

Объявить файловую переменную, например iFile : file of integer.

Связать её с файлом на диске с помощью функции Assign: Assign(iFile, 'ifile.txt');. Через первый параметр функции передаётся файловая переменная, а через второй — путь к файлу.

Открыть файл для записи с помощью функции Rewrite: Rewrite(iFile);.

Записать в файл число 100 с помощью функции Write: Write(iFile, 100);.

Закрыть файл с помощью функции Close: Close(iFile);.

1. Объектно-ориентированное программирование. Классы и объекты. Обращение к членам класса.

Основным понятием ООП является **класс**. **Класс** представляет собой структуру, состоящую из описания полей данных, функций и процедур, обрабатывающих поля данных и свойств. Функции и процедуры класса принято называть методами. Поля, методы и свойства называются элементами или членами класса.

Поле - это обычная переменная языка Паскаль. Допускаются любые разрешенные в языке типы переменных, в том числе поле может иметь тип класса. С помощью полей описываются состояние объекта. С помощью методов – поведение объекта. Свойство - это специфический способ для организации связи с данными класса.

Таким образом, класс описывает состояние и поведение **объекта**. Под **объектом** понимается конкретный экземпляр (сущность) класса. В программе **объект** описывается как переменная типа **класс**.

Объявление класса в простейшем случае производится следующим образом:

|  |
| --- |
| type  <Имя класса> = class  <объявление полей класса>  <объявление методов класса>  end; |

Обращение производится аналогично обращению к полям записи, т.е. через точку, например (Person — название переменной тип класс; name, fam — поля класса; GetData — метод класса):

|  |
| --- |
| Person.name:= 'Виталий ';  Person.fam:= 'Петров';  fname:= Person.GetData; |

или с помощью оператора with:

|  |
| --- |
| with Person do begin  name:= 'Алексей';  fam:= 'Морозов';  fname:= GetData;  end; |

1. Объектно-ориентированное программирование. Инкапсуляция. Спецификаторы доступа.

**Инкапсуляция** — объединение данных и обрабатывающих их функций и процедур в виде отдельных объектов.

Инкапсуляция подразумевает сокрытие внутренней структуры и ограничение доступа. Связь с "внешним миром" осуществляется через так называемый интерфейс класса, т.е. набором правил, регламентирующих доступ к членам класса. Известно лишь то, что делает, что умеет делать класс и как этим классом пользоваться (какой у этого класса интерфейс), т.е. какие у этого класса имеются доступные свойства и методы. Остаётся неизвестно как он делает то, что умеет.

Принцип сокрытия информации – один из важнейших принципов ООП. Скрывая внутреннее устройство класса, мы абстрагируемся от ненужных деталей. Кроме того, это позволяет защитить члены класса от несанкционированного доступа извне.

Для реализации инкапсуляции имеются следующие **спецификаторы** (директивы), управляющие видимостью (доступностью) членов класса:

* private (частный, говорят еще приватный) – поля и методы класса недоступны из других модулей. Это позволяет полностью скрыть всю "кухню" реализации класса. Однако они доступны в пределах того модуля, где описан данный класс. Более того, если в одном модуле определены несколько классов, то они "видят" приватные разделы друг друга. Это сделано для удобства разработчика данного класса (классов) в этом модуле. Согласитесь, глупо ограничивать в доступе к "внутренностям" телевизора самого изготовителя.
* protected (защищенный) – поля и методы класса имеют ограниченную видимость. Они видны в самом классе, во всех классах наследниках этого класса (том числе и в других модулях) и в программном коде, расположенном в том же модуле, что и данный класс.
* public (публичный) – свободно доступны из любого места программы, в том числе и из других модулей.

Как правило, поля класса объявляются как private, а методы – public. Хотя те методы, которые нужны только для внутреннего использования, вполне можно поместить в раздел private или protected.

|  |
| --- |
| program project1; {$mode objfpc}{$H+}  uses  CRT, FileUtil;  type  THuman = class  private  name: string;  fam: string;  public  function GetData: string;  end;  function THuman.GetData: string;  begin  Result:= name + ' ' + fam;  end;  var  Person: THuman;  fname: string;  begin  Person:= THuman.Create;  Person.name:= 'Виталий';  Person.fam:= 'Петров';  fname:= Person.GetData;  writeln(UTF8ToConsole('Это: ' + fname));  writeln(UTF8ToConsole('Нажмите любую клавишу'));  readkey;  Person.Free;  end. |

1. Объектно-ориентированное программирование. Наследование. Полиморфизм.

**Наследование** — создание класса на основе существующего. Новый класс называется наследником или потомком или дочерним классом. А старый класс называется родительским классом или предком или базовым классом.

Наследование одно из важнейших свойств ООП. Создание новых объектов путем использования уже существующих дает программисту ряд преимуществ:

* Не нужно повторно разрабатывать код. Весь код для существующих объектов автоматически может быть использован для новых объектов;
* Вероятность ошибок резко снижается. Если код для уже существующих объектов был уже отлажен и проверен, то любые возникшие ошибки следует искать в кодах, которые были добавлены для новых объектов и, наоборот, если найдены и исправлены ошибки в кодах для исходных объектов, то они будут автоматически исправлены и в кодах для новых объектов, созданных путем наследования;
* Код программы становится более понятным. Для того чтобы понять, как работают объекты, созданные путем наследования достаточно понять, как работают добавленные данные и функции

Объявляется класс наследник следующим образом:

|  |
| --- |
| type  <Имя класса наследника> = class(Имя класса родителя)  <Описание полей, методов и свойств, характерных только для класса наследника>  end;  <Реализация методов> |

Если имя класса родителя не указано, то по умолчанию класс наследуется от самого общего класса TObject.

Класс наследник получает (наследует) все поля, методы и свойства класса родителя. В то же время созданный класс, в свою очередь может выступать в качестве родителя. В Free Pascal существует так называемая иерархия классов, на вершине которого и находится класс TObject. Таким образом, класс TObject является прародителем всех остальных классов.

|  |
| --- |
| program project1;  {$mode objfpc}{$H+}  uses  CRT, FileUtil;  type  THuman = class // объявление базового класса  private  name: string;  fam: string;  public  function GetData: string;  end;  function THuman.GetData: string;  begin  Result:= name + ' ' + fam;  end;  type  TStudent = class(THuman) // объявление класса – наследника  private group: string;  end;  var Student: TStudent;  fname: string;  begin  Student:= TStudent.Create;  Student.name:= 'Виталий';  Student.fam:= 'Петров';  Student.group:= 'ПОВТАС-1/09';  fname:= Student.GetData;  writeln(UTF8ToConsole('Это: ' + fname));  writeln(UTF8ToConsole('Нажмите любую клавишу'));  readkey;  Student.Free;  end. |

Если требуется использовать только объект класса наследника, то создавать экземпляр родительского класса вовсе не обязательно. Более того, можно создать класс наследник в другом модуле, нежели где описан базовый класс

**Полиморфизм —** явление, когда методы класса родителя и методы дочерних классов имеют одинаковые имена, но различные реализации. Например, пусть у вас имеется объект "Геометрическая фигура" и пусть у нее имеется функция вычисления площади. Если необходимо вычислить площадь прямоугольника, то функция будет вычислять ее по одному алгоритму, а если это треугольник, то ее площадь функция будет вычислять по совсем другому алгоритму.

|  |
| --- |
| function TStudent.GetData: string;  begin  Result:= name + ' ' + fam + ', группа ' + group;  end;  function TProfessor.GetData: string;  begin  Result:= name + ' ' + fam + ', кафедра ' + kafedra;  end; |

1. Объектно-ориентированное программирование. Конструкторы и деструкторы.

**Конструктор** — специальный метод класса, который предназначен для выделения памяти и размещения в памяти экземпляра класса. Кроме того, в задачу конструктора входит инициализация значений полей экземпляра класса. Стандартный конструктор устанавливает все данные нового экземпляра класса в ноль. В результате все числовые поля и поля порядкового типа приобретают нулевые значения, строковые поля становятся пустыми, а поля, содержащие указатели и объекты получают значение nil. Под стандартным конструктором имеется в виду конструктор, унаследованный классом от класса TObject.

Если нужно инициализировать данные экземпляра класса определенными значениями, то необходимо написать свой собственный конструктор. Допускается использование нескольких конструкторов. Желательно, чтобы все конструкторы имели стандартное имя Create. Это позволяет переопределять конструкторы, включая и стандартный конструктор, для выполнения каких-либо полезных действий. Описание конструктора производится с помощью ключевого слова constructor.

|  |
| --- |
| program ConstructorExample;  type  TMyClass = class  private  FValue: Integer;  FName: String;  public  **constructor Create(AValue: Integer; AName: String);** // конструктор  procedure Display;  end;  **constructor TMyClass.Create(AValue: Integer; AName: String); // реализация конструктора**  **begin**  **FValue := AValue;**  **FName := AName;**  **Writeln('Объект ', FName, ' создан со значением ', FValue);**  **end;**  procedure TMyClass.Display;  begin  Writeln('Имя: ', FName, ', Значение: ', FValue);  end;  var  MyObj: TMyClass;  begin  MyObj := TMyClass.Create(10, 'Тестовый объект');  MyObj.Display;  MyObj.Free;  end. |