**4. Перечень основных учебных модулей образовательной программы или их разделов и вопросов, выносимых для проверки на государственном экзамене**

1. **АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ**
   1. **Программирование на С++**
      1. Язык С. Общая характеристика языка. Структура программы на С и типы данных. Язык С++ - расширение языка С, дополнительные возможности.

Си — компилируемый статически типизированный высокоуровневый язык программирования общего назначения.

Основные характеристики языка C:

1. Простота: C имеет небольшое число ключевых слов, лаконичный синтаксис и минималистический набор основных операторов, что делает его относительно простым для изучения и использования.
2. Эффективность: Язык C предлагает низкоуровневые операции, позволяющие разработчикам полностью контролировать процесс выполнения программы. Он обеспечивает прямой доступ к памяти и поддерживает указатели, что позволяет эффективно работать с данными.
3. Переносимость: Код на C можно компилировать и запускать на различных платформах без необходимости внесения значительных изменений. Это делает язык C очень переносимым и универсальным.

Структура программы на языке C включает в себя:

1. Директивы препроцессора: Эти директивы начинаются с символа решетки (#) и используются для включения файлов заголовков, определения макросов и других операций, выполняемых до фактической компиляции кода.
2. Функция main(): Вся программа на C должна содержать функцию main(), которая является точкой входа в программу. Она выполняется первой при запуске программы и определяет начало выполнения.
3. Объявления переменных: В начале программы могут быть объявлены переменные, которые будут использоваться в программе. Каждая переменная должна быть объявлена с указанием ее типа.
4. Определение функций: Пользователь может определить свои собственные функции, которые выполняют определенные задачи в программе. Определение функции включает ее имя, тип возвращаемого значения, список параметров и блок кода, который будет выполнен при вызове функции.
5. Операторы и выражения: Программа на C состоит из операторов и выражений, которые выполняют различные операции, такие как присваивание, арифметические вычисления, условные операторы и циклы.

#include <stdio.h>

int func()

int main(void)

{

printf("Hello METANIT.COM! \n");\\ комментарий

return 0;

}

|  |  |  |
| --- | --- | --- |
|  | C | C++ |
| Появление | Разработал [Деннис Ричи](https://ru.wikipedia.org/wiki/%D0%A0%D0%B8%D1%82%D1%87%D0%B8,_%D0%94%D0%B5%D0%BD%D0%BD%D0%B8%D1%81) в 1969г. в [Лабаратории Белла](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D0%B1%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8%D0%B8_%D0%91%D0%B5%D0%BB%D0%BB%D0%B0). | Разработал [Бьярне Страуструп](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D1%83%D1%81%D1%82%D1%80%D1%83%D0%BF,_%D0%91%D1%8C%D1%91%D1%80%D0%BD) в 1979г. |
| Класс языка | [Процедурный](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). | [Объектно-ориентированный](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), [мультипарадигмальный](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), [процедурный](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), [функциональный](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), [обобщённый](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). |
| ООП | Поскольку C не поддерживает концепцию ООП, то он не поддерживает [полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC), [инкапсуляцию](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). | C++ поддерживает [полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC), [инкапсуляцию](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [наследование](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), т. к. является объектно-ориентированным языком. |
| Безопасность данных | Поскольку C не поддерживает инкапсуляцию, то данные могут редактироваться внешним кодом. | В случае с C++ инкапсуляция скрывает данные, чтобы гарантировать, что структуры данных и операторы используются по назначению. |
| Особенности | Не поддерживает перегрузку функций и операторов. | C++ поддерживает как и перегрузку функций, так и перегрузку операторов, а также [пространства имен](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D0%B8%D0%BC%D1%91%D0%BD_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и ссылки, обработка исключений, богатая библиотека. |
| [Ключевые слова](https://www.geeksforgeeks.org/cc-tokens/) | 32 ключевых слова | 52 ключевых слова |
| Функции | Не поддерживаются в структуре, не могут быть "друзьями" и виртуальными | Поддерживаются в структуре, могут быть "друзьями" и виртуальными |
| Память | Функции malloc() и calloc() для динамического выделения памяти, а также free() для освобождения. | Для этих же операций используются операторы new и delete. |
| Вход/Выход | Используются scanf() и printf(). | Используются cout и cin. |

* + 1. Язык С. Основные типы данных. Базовый набор операций. Операторы разветвления и циклов.

Язык C предлагает ряд основных типов данных, которые могут использоваться для объявления переменных и хранения значений. Вот некоторые из основных типов данных в C:

Целочисленные типы данных:

* int: целые числа со знаком.
* char: символы или целые числа без знака, представляющие отдельные символы в кодировке ASCII.

Вещественные типы данных:

* float: одинарная точность с плавающей запятой.
* double: двойная точность с плавающей запятой.

Логический тип данных:

* bool: логическое значение true или false.

Составные типы данных:

* массивы: упорядоченные наборы элементов одного типа.
* структуры: пользовательские типы данных, позволяющие объединять различные типы данных под одной сущностью.
* указатели: переменные, содержащие адреса памяти других переменных.

Базовый набор операций в языке C включает:

Арифметические операции:

* Сложение (+), вычитание (-), умножение (\*), деление (/).
* Модуль (%) - возвращает остаток от деления.

Операции присваивания:

* Присваивание (=) - присваивает значение одной переменной другой.
* Комбинированные операции присваивания (+=, -=, \*=, /=) - выполняют операцию и присваивают результат переменной.

Операции сравнения:

* Равно (==), не равно (!=), больше (>), меньше (<), больше или равно (>=), меньше или равно (<=).

Логические операции:

* Логическое И (&&), логическое ИЛИ (||), логическое НЕ (!).

Операторы разветвления и циклов используются для управления потоком выполнения программы. Некоторые из операторов разветвления и циклов в языке C:

Операторы разветвления:

* if: выполняет определенный блок кода, если указанное условие истинно.
* if-else: выполняет один блок кода, если условие истинно, и другой блок кода, если условие ложно.
* switch: выбирает один из нескольких вариантов выполнения кода на основе значения выражения.

Циклы:

* for: выполняет блок кода заданное количество раз.
* while: выполняет блок кода, пока условие истинно.
* do-while: выполняет блок кода, затем проверяет условие, и если оно истинно, повторяет выполнение блока кода.

Операторы разветвления и циклов позволяют программе принимать решения и выполнять повторяющиеся операции в зависимости от условий, что делает язык C мощным инструментом для управления логикой программ.

* + 1. Язык С. Производные типы данных: массивы, указатели, структура массивы структур. Статические и динамические массивы.

В языке C существуют производные типы данных, которые позволяют более гибко работать с данными. Вот некоторые из них:

Массивы: Массив представляет собой упорядоченный набор элементов одного типа. Он позволяет хранить несколько значений одного типа под одним именем. Например, вы можете объявить массив целых чисел следующим образом:

int numbers[5]; // объявление массива с 5 элементами типа int

Указатели: Указатель представляет собой переменную, которая содержит адрес памяти другой переменной. Он позволяет получить доступ к значению переменной по ее адресу. Указатели могут использоваться для работы с массивами, динамической памятью и функциями. Пример объявления указателя:

int \*ptr; // объявление указателя на int

Структуры: Структура позволяет объединять различные типы данных в одном пользовательском типе. Она может содержать переменные разных типов, называемых полями структуры. Пример объявления структуры:

struct Person {

char name[20];

int age; };

Массивы структур: Вы можете создавать массивы, элементами которых являются структуры. Это позволяет хранить несколько структур под одним именем и обращаться к их элементам. Пример объявления массива структур:

struct Person people[5]; // объявление массива структур Person с 5 элементами

Статические и динамические массивы: В C вы можете создавать статические массивы, размер которых определяется на этапе компиляции и остается неизменным во время выполнения программы. Статический массив объявляется с фиксированным размером:

int numbers[5]; // статический массив с 5 элементами

Для работы с динамическими массивами, размер которых может быть изменен во время выполнения программы, используется динамическое выделение памяти с помощью функции malloc() или оператора new (в C++). Динамический массив объявляется указателем:

int \*dynamicArray = malloc(5 \* sizeof(int)); // динамический массив с 5 элементами

После использования динамического массива, его необходимо освободить с помощью функции free() (в C) или оператора delete (в C++):

free(dynamicArray); // освобождение памяти, выделенной для динамического массива

Использование статических и динамических массивов зависит от требований программы и контекста использования.

* + 1. Списковые структуры. Функции динамического выделения памяти. Разновидности списков: стэк, очередь, кольцевой список. Основные операции со списками.

В C++ также существуют списковые структуры данных и функции динамического выделения памяти. Давайте рассмотрим основные концепции и операции со списками в C++.

Стек (stack):

Стек представляет собой структуру данных типа "Last-In-First-Out" (LIFO), где добавление и удаление элементов происходит только с одного конца списка, называемого вершиной стека.

#include <iostream>

#include <cstdlib>

using namespace std;

// Определяем емкость stack по умолчанию

#define SIZE 10

// Класс для представления stack

class Stack

{

    int \*arr;

    int top;

    int capacity;

public:

    Stack(int size = SIZE);         // конструктор

    ~Stack();                       // деструктор

    void push(int);

    int pop();

    int peek();

    int size();

    bool isEmpty();

    bool isFull();

};

// Конструктор для инициализации stack

Stack::Stack(int size)

{

    arr = new int[size];

    capacity = size;

    top = -1;

}

// Деструктор для освобождения памяти, выделенной для stack

Stack::~Stack() {

    delete[] arr;

}

// Вспомогательная функция для добавления элемента `x` в stack

void Stack::push(int x)

{

    if (isFull())

    {

        cout << "Overflow\nProgram Terminated\n";

        exit(EXIT\_FAILURE);

    }

    cout << "Inserting " << x << endl;

    arr[++top] = x;

}

// Вспомогательная функция для извлечения верхнего элемента из stack

int Stack::pop()

{

    // проверка на опустошение stack

    if (isEmpty())

    {

        cout << "Underflow\nProgram Terminated\n";

        exit(EXIT\_FAILURE);

    }

    cout << "Removing " << peek() << endl;

    // уменьшаем размер stack на 1 и (необязательно) возвращаем извлеченный элемент

    return arr[top--];

}

// Вспомогательная функция для возврата верхнего элемента stack

int Stack::peek()

{

    if (!isEmpty()) {

        return arr[top];

    }

    else {

        exit(EXIT\_FAILURE);

    }

}

// Вспомогательная функция для возврата размера stack

int Stack::size() {

    return top + 1;

}

// Вспомогательная функция для проверки, пуст stack или нет

bool Stack::isEmpty() {

    return top == -1;               // или return size() == 0;

}

// Вспомогательная функция для проверки, заполнен ли stack или нет

bool Stack::isFull() {

    return top == capacity - 1;     // или return size() == capacity;

}

int main()

{

    Stack pt(3);

    pt.push(1);

    pt.push(2);

    pt.pop();

    pt.pop();

    pt.push(3);

    cout << "The top element is " << pt.peek() << endl;

    cout << "The stack size is " << pt.size() << endl;

    pt.pop();

    if (pt.isEmpty()) {

        cout << "The stack is empty\n";

    }

    else {

        cout << "The stack is not empty\n";

    }

    return 0;

}

В C++, стек можно реализовать с помощью контейнера std::stack из стандартной библиотеки. Основные операции со стеком включают:

* Push: добавление элемента в вершину стека с помощью функции push().
* Pop: удаление элемента из вершины стека с помощью функции pop().
* Top: получение значения вершины стека без удаления с помощью функции top().

Пример использования стека в C++:

#include <stack>

std::stack<int> myStack;

myStack.push(10);

myStack.push(20);

int topElement = myStack.top();

myStack.pop();

Очередь (queue):

Очередь представляет собой структуру данных типа "First-In-First-Out" (FIFO), где добавление происходит в конец списка (хвост), а удаление происходит из начала списка (головы).

#include <iostream>

#include <cstdlib>

using namespace std;

// Определяем емкость queue по умолчанию

#define SIZE 1000

// Класс для хранения queue

class Queue

{

    int \*arr;       // массив для хранения элементов queue

    int capacity;   // максимальная емкость queue

    int front;      // front указывает на передний элемент в queue (если есть)

    int rear;       // задняя часть указывает на последний элемент в queue

    int count;      // текущий размер queue

public:

    Queue(int size = SIZE);     // конструктор

    ~Queue();                   // деструктор

    int dequeue();

    void enqueue(int x);

    int peek();

    int size();

    bool isEmpty();

    bool isFull();

};

// Конструктор для инициализации queue

Queue::Queue(int size)

{

    arr = new int[size];

    capacity = size;

    front = 0;

    rear = -1;

    count = 0;

}

// Деструктор для освобождения памяти, выделенной для queue

Queue::~Queue() {

    delete[] arr;

}

// Вспомогательная функция для удаления переднего элемента из очереди

int Queue::dequeue()

{

    // проверка на опустошение queue

    if (isEmpty())

    {

        cout << "Underflow\nProgram Terminated\n";

        exit(EXIT\_FAILURE);

    }

    int x = arr[front];

    cout << "Removing " << x << endl;

    front = (front + 1) % capacity;

    count--;

    return x;

}

// Вспомогательная функция для добавления элемента в queue

void Queue::enqueue(int item)

{

    // проверка на переполнение queue

    if (isFull())

    {

        cout << "Overflow\nProgram Terminated\n";

        exit(EXIT\_FAILURE);

    }

    cout << "Inserting " << item << endl;

    rear = (rear + 1) % capacity;

    arr[rear] = item;

    count++;

}

// Вспомогательная функция для возврата первого элемента queue

int Queue::peek()

{

    if (isEmpty())

    {

        cout << "Underflow\nProgram Terminated\n";

        exit(EXIT\_FAILURE);

    }

    return arr[front];

}

// Вспомогательная функция для возврата размера queue

int Queue::size() {

    return count;

}

// Вспомогательная функция для проверки, пуста ли queue или нет

bool Queue::isEmpty() {

    return (size() == 0);

}

// Вспомогательная функция для проверки того, заполнена ли queue или нет

bool Queue::isFull() {

    return (size() == capacity);

}

int main()

{

    // создаем queue емкостью 5

    Queue q(5);

    q.enqueue(1);

    q.enqueue(2);

    q.enqueue(3);

    cout << "The front element is " << q.peek() << endl;

    q.dequeue();

    q.enqueue(4);

    cout << "The queue size is " << q.size() << endl;

    q.dequeue();

    q.dequeue();

    q.dequeue();

    if (q.isEmpty()) {

        cout << "The queue is empty\n";

    }

    else {

        cout << "The queue is not empty\n";

    }

    return 0;

}

В C++, очередь можно реализовать с помощью контейнера std::queue из стандартной библиотеки. Основные операции с очередью включают:

* Enqueue: добавление элемента в конец очереди с помощью функции push().
* Dequeue: удаление элемента из начала очереди с помощью функции pop().
* Front: получение значения элемента в начале очереди без удаления с помощью функции front().

Пример использования очереди в C++:

#include <queue>

std::queue<int> myQueue;

myQueue.push(10);

myQueue.push(20);

int frontElement = myQueue.front();

myQueue.pop();

Кольцевой список (circular list):

Кольцевой список представляет собой список, в котором последний элемент связан с первым элементом, создавая замкнутую структуру. Такой список позволяет производить циклические обходы и добавление/удаление элементов.

Кольцевой список можно реализовать самостоятельно с использованием классов и указателей. В этом случае вам придется определить свои собственные операции для вставки, удаления и обхода элементов в списке.

Пример реализации кольцевого списка в C++:

#include <iostream>

class Node {

public:

int data;

Node\* next;

};

class CircularList {

private:

Node\* head;

public:

CircularList() : head(nullptr) {}

void insert(int value) {

Node\* newNode = new Node;

newNode->data = value;

if (head == nullptr) {

head = newNode;

head->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

void display() {

if (head == nullptr) {

std::cout << "List is empty." << std::endl;

return;

}

Node\* temp = head;

do {

std::cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

std::cout << std::endl;

}

};

int main() {

CircularList myList;

myList.insert(10);

myList.insert(20);

myList.insert(30);

myList.display();

return 0;

}

В целом, стандартная библиотека C++ предлагает различные контейнеры и алгоритмы, которые могут быть использованы для работы со списковыми структурами данных. Это упрощает разработку и облегчает использование списков в C++.

* + 1. Структурное программирование и средства его реализации на языке С. Функции. Определение и вызов функций. Механизм параметров. Стандартные библиотеки.

Структурное программирование является методологией разработки программ, основанной на структуре и последовательности операций. Оно стремится создать программы, которые легко читаются, понимаются и поддерживаются. В языке C структурное программирование может быть реализовано с использованием функций.

Функции в языке C представляют собой отдельные блоки кода, которые могут быть вызваны из других частей программы. Функции позволяют разбить программу на более мелкие, самодостаточные модули, что упрощает чтение, понимание и сопровождение кода.

Определение функции в языке C имеет следующий синтаксис:

тип\_возвращаемого\_значения имя\_функции(параметры) {

// Тело функции

// ...

return значение\_результата; }

Где:

* тип\_возвращаемого\_значения - тип данных, которые функция возвращает после выполнения (или void, если функция не возвращает значения).
* имя\_функции - уникальное имя, используемое для вызова функции.
* параметры - список переменных, передаваемых в функцию для использования внутри тела функции.
* Тело функции - блок кода, выполняющий определенные операции.
* return значение\_результата; - оператор, который возвращает значение из функции (необязательно, если функция имеет тип void).

Пример определения и вызова функции:

#include <stdio.h>

// Функция, которая выводит приветствие на экран

void sayHello() {

printf("Привет, мир!\n"); }

int main() {

// Вызов функции

sayHello sayHello();

return 0; }

В этом примере функция sayHello определена без параметров и без возвращаемого значения. Она просто выводит на экран сообщение "Привет, мир!". Затем функция вызывается внутри функции main().

Механизм параметров позволяет передавать значения в функцию для использования внутри неё. Пример с функцией, принимающей параметры:

#include <stdio.h>

// Функция, которая складывает два числа и возвращает сумму

int addNumbers(int a, int b) {

int sum = a + b;

return sum; }

int main() {

int result = addNumbers(5, 7);

printf("Сумма: %d\n", result);

return 0; }

В этом примере функция addNumbers принимает два параметра a и b, складывает их, сохраняет результат в переменную sum и возвращает его с помощью оператора return. Затем функция вызывается внутри функции main() с аргументами 5 и 7. Результат функции сохраняется в переменной result и выводится на экран.

Стандартные библиотеки в языке C предоставляют набор готовых функций и структур данных, которые могут быть использованы в программе без необходимости их повторной реализации. Например, стандартная библиотека <stdio.h> предоставляет функции для работы с вводом/выводом (например, printf() и scanf()), а <stdlib.h> предоставляет функции для работы с памятью (например, malloc() и free()).

Для использования функций из стандартных библиотек в программе, обычно необходимо включить соответствующий заголовочный файл с помощью директивы #include.

Пример использования функции printf() из стандартной библиотеки <stdio.h>:

#include <stdio.h>

int main() {

printf("Hello, World!\n");

return 0; }

Этот пример выводит на экран сообщение "Hello, World!" с помощью функции printf() из стандартной библиотеки <stdio.h>.

* + 1. Препроцессор. Назначение и возможности.

Препроцессор в языке C является одной из фаз компиляции и выполняет обработку исходного кода до фактической компиляции. Препроцессор выполняет определенные действия на основе директив препроцессора, которые начинаются с символа #.

Назначение препроцессора заключается в предварительной обработке исходного кода перед компиляцией и включает в себя следующие возможности:

Макросы (#define): Препроцессор позволяет определить макросы, которые представляют собой замены текста в исходном коде. Макросы могут быть использованы для определения констант, создания функций без аргументов и других текстовых замен.

Пример использования макросов:

#include <stdio.h>

#define PI 3.14159

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {

printf("Значение PI: %f\n", PI);

int x = 5;

int y = 10;

int max = MAX(x, y);

printf("Максимальное значение: %d\n", max);

return 0; }

В этом примере макрос PI заменяется на значение 3.14159, а макрос MAX(a, b) заменяется на максимальное значение из a и b.

Включение файлов (#include): Препроцессор также позволяет включать содержимое других файлов в исходный код с помощью директивы #include. Это полезно для повторного использования кода и включения заголовочных файлов с объявлениями функций и структур данных.

Пример включения файла:

#include <stdio.h>

int main() {

printf("Hello, World!\n");

return 0; }

В этом примере содержимое файла <stdio.h> будет включено в исходный код до компиляции, чтобы функции printf() и другие функции стандартной библиотеки были доступны.

Условная компиляция (#ifdef, #ifndef, #if, #endif): Препроцессор позволяет включать или исключать определенный блок кода на основе условий. Это полезно, когда нужно компилировать различный код для разных платформ или настроек.

Пример условной компиляции:

#include <stdio.h>

#define DEBUG

int main() {

#ifdef DEBUG

printf("Отладочная информация\n");

#else

printf("Производственная версия\n");

#endif

return 0; }

В этом примере блок кода, содержащий printf("Отладочная информация\n"), будет включен только при наличии определения макроса DEBUG.

Препроцессор в языке C обеспечивает мощные возможности для предварительной обработки исходного кода, что позволяет разработчикам более гибко управлять исходным кодом и создавать более гибкие программы.

* + 1. Объектно-ориентированное программирование. Основные принципы: инкапсуляция, наследование, полиморфизм. Классы и объекты. Описание класса. Поля данных и методы-члены класса. Конструкторы и деструкторы. Средства ограничения доступа к членам класса.

Объектно-ориентированное программирование (ООП) - это методология разработки программного обеспечения, основанная на концепции объектов, которые являются экземплярами классов. ООП строит программы вокруг объектов, которые взаимодействуют друг с другом, обладают свойствами (полями данных) и поведением (методами).

Основные принципы ООП включают:

1. Инкапсуляция: Инкапсуляция позволяет объединить данные и методы, работающие с этими данными, в единый объект. Объект скрывает внутреннюю реализацию и предоставляет только интерфейс (публичные методы), через который другие объекты могут взаимодействовать с ним. Это обеспечивает контролируемый доступ к данным и защиту от неправильного использования.
2. Наследование: Наследование позволяет создавать новые классы на основе существующих. Класс-наследник (подкласс) наследует свойства и методы класса-родителя (суперкласса) и может добавлять свои собственные свойства и методы. Наследование позволяет создавать иерархию классов, группируя их по сходству и специализации.
3. Полиморфизм: Полиморфизм позволяет использовать один и тот же интерфейс (метод) для разных типов данных. Это означает, что можно создать общий метод, который может принимать объекты разных классов и вести себя по-разному в зависимости от конкретного типа объекта. Полиморфизм позволяет создавать более гибкий и расширяемый код.

Классы и объекты являются основными концепциями ООП:

* Класс - это шаблон или формальное определение, описывающее состояние (поля данных) и поведение (методы) объектов, которые будут созданы на его основе. Класс определяет структуру и функциональность объектов.
* Объект - это экземпляр класса, созданный в памяти во время выполнения программы. Объект имеет свое собственное состояние (значения полей данных) и может выполнять методы, определенные в классе.

Описание класса включает объявление полей данных и методов-членов класса:

class MyClass {

private: // Модификатор доступа private ограничивает доступ к членам класса

int myField; // Поле данных класса

public: // Модификатор доступа public обеспечивает открытый доступ к членам класса

void myMethod(); // Метод класса };

В этом примере класс MyClass имеет одно поле данных myField и один метод myMethod(). Модификатор доступа private указывает, что поле данных является приватным и доступ к нему ограничен внутри класса. Модификатор доступа public указывает, что метод является публичным и может быть вызван извне класса.

Конструкторы и деструкторы — это специальные методы класса:

* Конструктор — это метод, вызываемый при создании объекта класса. Он инициализирует поля данных объекта и может выполнять другие необходимые операции. Конструктор имеет то же имя, что и класс.
* Деструктор — это метод, вызываемый при уничтожении объекта класса. Он освобождает память и выполняет другие завершающие операции. Деструктор имеет то же имя, что и класс, с префиксом ~.

class MyClass {

public: MyClass(); // Конструктор

~MyClass(); // Деструктор // ... };

Средства ограничения доступа к членам класса (полям данных и методам) включают модификаторы доступа private, public и protected. Модификаторы доступа определяют, кто и в каком контексте имеет доступ к членам класса.

* private: Члены класса, объявленные с модификатором private, являются приватными и доступны только внутри самого класса. Они недоступны извне класса или его производных классов.
* public: Члены класса, объявленные с модификатором public, являются публичными и доступны из любого места программы. Они могут быть вызваны извне класса.
* protected: Члены класса, объявленные с модификатором protected, являются защищенными и доступны внутри самого класса и его производных классов. Они недоступны извне класса.

Пример использования модификаторов доступа:

class MyClass {

private: int privateField; // Приватное поле данных

public: int publicField; // Публичное поле данных

protected: int protectedField; // Защищенное поле данных };

В этом примере privateField является приватным и доступным только внутри класса, publicField является публичным и доступным извне класса, а protectedField является защищенным и доступным внутри класса и его производных классов.

Стандартные библиотеки C++ предоставляют множество классов и функций для работы с объектно-ориентированным программированием, таких как std::string, std::vector, std::iostream и многих других. Эти библиотеки облегчают создание и использование классов в C++.

* + 1. Объектно-ориентированное программирование. Наследование: одиночное и множественное. Контейнерные классы.

В объектно-ориентированном программировании (ООП) наследование — это механизм, который позволяет создавать новые классы на основе существующих классов, называемых базовыми классами или суперклассами. Наследование позволяет наследующему классу получить свойства и методы базового класса и добавить свои собственные.

Одиночное наследование: Одиночное наследование предполагает наследование свойств и методов только от одного базового класса.

Пример одиночного наследования:

class BaseClass {

public:

void baseMethod() {

// реализация метода базового класса }

};

class DerivedClass : public BaseClass {

public:

void derivedMethod() {

// реализация метода производного класса }

};

В этом примере DerivedClass наследует метод baseMethod() от BaseClass и добавляет свой собственный метод derivedMethod().

Множественное наследование: Множественное наследование позволяет классу наследовать свойства и методы от нескольких базовых классов.

Пример множественного наследования:

class BaseClass1 {

public: void baseMethod1() {

// реализация метода базового класса 1 }

};

class BaseClass2 {

public: void baseMethod2() {

// реализация метода базового класса 2 }

};

class DerivedClass : public BaseClass1, public BaseClass2 {

public: void derivedMethod() {

// реализация метода производного класса }

};

В этом примере DerivedClass наследует методы baseMethod1() от BaseClass1 и baseMethod2() от BaseClass2 и добавляет свой собственный метод derivedMethod().

Контейнерные классы: Контейнерные классы в ООП представляют собой классы, которые служат для хранения и управления группой объектов других классов. Они предоставляют удобные методы для добавления, удаления, поиска и манипулирования объектами внутри себя.

Примеры контейнерных классов в C++ включают std::vector, std::list, std::map и другие из стандартной библиотеки C++. Эти классы предоставляют различные способы хранения и управления коллекциями объектов.

Например, std::vector является динамическим массивом, который автоматически увеличивается при добавлении элементов. Он предоставляет методы для добавления, удаления, доступа к элементам и других операций.

#include <vector>

int main() {

std::vector<int> myVector; // Создание экземпляра контейнерного класса std::vector

myVector.push\_back(1); // Добавление элемента в конец вектора

myVector.push\_back(2);

myVector.push\_back(3);

for (int i : myVector) {

// Вывод элементов вектора

std::cout << i << " "; }

return 0; }

В этом примере std::vector используется для хранения целых чисел. Метод push\_back() добавляет элементы в конец вектора, а цикл for используется для вывода элементов вектора на экран.

* + 1. Полиморфизм. Переопределение функций. Перегрузка стандартных операций, операторные функции.

Полиморфизм — это один из ключевых принципов объектно-ориентированного программирования, который позволяет объектам разных классов проявлять различное поведение на основе общего интерфейса. Полиморфизм позволяет использовать объекты производных классов через указатель или ссылку на базовый класс.

Переопределение функций: Переопределение функций — это механизм, который позволяет производному классу предоставить свою собственную реализацию функции, определенной в базовом классе. При вызове функции для объекта производного класса будет использоваться его собственная реализация, а не реализация из базового класса.

Пример переопределения функции:

class BaseClass {

public:

virtual void print() {

std::cout << "BaseClass" << std::endl; }

};

class DerivedClass : public BaseClass {

public:

void print() override {

std::cout << "DerivedClass" << std::endl; }

};

В этом примере метод print() переопределен в классе DerivedClass и выводит сообщение "DerivedClass". При вызове print() для объекта DerivedClass, будет использоваться его собственная реализация.

Перегрузка стандартных операций и операторные функции: Перегрузка стандартных операций позволяет определить новое поведение для операций, таких как арифметические операции (+, -, \*, /), операции присваивания (=) и других. Операторные функции - это функции, которые перегружают операторы для определенных классов.

Пример перегрузки оператора сложения (+):

class MyNumber {

private:

int value;

public:

MyNumber(int val) : value(val) {}

MyNumber operator+(const MyNumber& other) {

return MyNumber(value + other.value); }

int getValue() {

return value; }

};

В этом примере класс MyNumber перегружает оператор сложения (+). Операторная функция operator+ определяет новое поведение для оператора сложения для объектов класса MyNumber. При выполнении операции сложения между двумя объектами MyNumber, будет вызываться данная операторная функция.

int main() {

MyNumber num1(5);

MyNumber num2(10);

MyNumber sum = num1 + num2; // Выполнение операции сложения

std::cout << "Sum: " << sum.getValue() << std::endl;

return 0; }

В этом примере создаются два объекта MyNumber и выполняется операция сложения между ними с использованием перегруженного оператора +. Результат выводится на экран.

Таким образом, переопределение функций и перегрузка стандартных операций позволяют объектам проявлять различное поведение и адаптироваться к различным ситуациям. Это важные инструменты в объектно-ориентированном программировании.

* + 1. Организация ввода/вывода в языках С и С++. Форматированный ввод/вывод. Работа с файлами. Потоки, разновидности потоков, поточные классы.

Организация ввода/вывода в языках C и C++:

В языке C и C++ ввод/вывод осуществляется через стандартные потоки (stdin, stdout и stderr) и библиотеку функций stdio.h (в C) или iostream (в C++). Стандартные потоки представляют стандартные устройства ввода/вывода, такие как клавиатура (stdin) и экран (stdout и stderr).

Форматированный ввод/вывод:

Форматированный ввод/вывод позволяет контролировать формат и структуру данных при вводе/выводе. В языке C используется функциональность из библиотеки stdio.h, в то время как в C++ используется объектно-ориентированный подход с использованием классов std::iostream и связанных с ними классов и функций из библиотеки iostream.

Пример чтения данных с использованием scanf в языке C:

#include <stdio.h>

int main() {

int num;

char name[100];

printf("Enter a number: ");

scanf("%d", &num); // Ввод числа с использованием

scanf printf("Enter your name: ");

scanf(" %[^\n]s", name); // Ввод строки с использованием

scanf printf("Number: %d\n", num);

printf("Name: %s\n", name);

return 0; }

В этом примере пользователю предлагается ввести число и имя. Функция scanf используется для ввода числа и строки. %d используется для чтения целого числа, а %[^\n]s для чтения строки с пробелами. Обратите внимание на использование амперсанда & при передаче адреса переменной в scanf для ввода значения.

После ввода данных, они выводятся на экран с помощью printf.

Пример чтения данных с использованием std::cin в C++:

#include <iostream>

int main() {

int num; std::string name;

std::cout << "Enter a number: ";

std::cin >> num; // Ввод числа с помощью std::cin

std::cout << "Enter your name: ";

std::cin.ignore(); // Очистка буфера перед вводом строки

std::getline(std::cin, name); // Ввод строки с помощью std::getline

std::cout << "Number: " << num << std::endl;

std::cout << "Name: " << name << std::endl; return 0; }

В этом примере пользователю предлагается ввести число и имя. Функция std::cin используется для ввода числа, а функция std::getline используется для ввода строки. Результаты выводятся на экран с помощью std::cout.

Пример форматированного вывода в C:

#include <stdio.h>

int main() {

int num = 42;

float pi = 3.14159;

printf("The number is %d and the value of pi is %.2f\n", num, pi);

return 0; }

В этом примере функция printf() используется для форматированного вывода на экран. Знаки %d и %.2f являются спецификаторами формата, которые заменяются соответствующими значениями переменных num и pi.

Пример форматированного ввода-вывода в C++:

#include <iostream>

#include <iomanip>

int main() {

int num = 42;

double pi = 3.14159;

std::cout << "The number is " << num << " and the value of pi is " << std::fixed << std::setprecision(2) << pi << std::endl;

return 0; }

В этом примере объект std::cout используется для форматированного вывода на экран. Оператор << используется для добавления значений переменных в поток вывода. Функции std::fixed и std::setprecision() задают формат и точность вывода числа pi.

Работа с файлами:

В языке C и C++ также предоставляются средства для работы с файлами. В языке C используются функции из библиотеки stdio.h, такие как fopen(), fread(), fwrite(), fclose() и другие. В C++ для работы с файлами используются классы и функции из библиотеки fstream, такие как ifstream, ofstream и fstream.

Вот примеры основных операций работы с файлами в языке C:

Открытие файла:

#include <stdio.h>

int main() {

FILE\* file = fopen("example.txt", "r");

if (file == NULL) { printf("Failed to open the file.\n");

return 1; } // Работа с файлом

fclose(file);

return 0; }

В этом примере функция fopen() используется для открытия файла "example.txt" в режиме чтения ("r"). Если файл не удалось открыть, возвращается NULL.

Чтение данных из файла:

#include <stdio.h>

int main() {

FILE\* file = fopen("example.txt", "r");

if (file == NULL) {

printf("Failed to open the file.\n");

return 1; }

int number;

fscanf(file, "%d", &number);

printf("Number: %d\n", number);

fclose(file);

return 0; }

В этом примере используется функция fscanf() для чтения целого числа из файла. Затем число выводится на экран.

Запись данных в файл:

#include <stdio.h>

int main() {

FILE\* file = fopen("example.txt", "w");

if (file == NULL) {

printf("Failed to open the file.\n");

return 1; }

int number = 42;

fprintf(file, "Number: %d\n", number);

fclose(file);

return 0; }

В этом примере используется функция fprintf() для записи числа в файл вместе с текстом "Number: ".

Добавление данных в конец файла:

#include <stdio.h>

int main() {

FILE\* file = fopen("example.txt", "a");

if (file == NULL) {

printf("Failed to open the file.\n");

return 1; }

int number = 42;

fprintf(file, "Number: %d\n", number);

fclose(file);

return 0; }

В этом примере используется режим "a" при открытии файла с помощью fopen(), что позволяет добавить данные в конец файла, а не перезаписывать его.

Проверка конца файла:

#include <stdio.h>

int main() {

FILE\* file = fopen("example.txt", "r");

if (file == NULL) {

printf("Failed to open the file.\n");

return 1; }

int number;

while (fscanf(file, "%d", &number) == 1) {

printf("Number: %d\n", number); }

fclose(file);

return 0; }

В этом примере цикл while используется для чтения чисел из файла до тех пор, пока файл не достигнет конца. Функция fscanf() возвращает количество успешно считанных элементов данных, поэтому в условии цикла проверяется, что возвращенное значение равно 1 (успешное чтение одного числа).

Это основные операции работы с файлами в языке C. Библиотека stdio.h предоставляет и другие функции и возможности для работы с файлами, такие как позиционирование в файле, переименование файлов и удаление файлов.

Пример работы с файлами в C++:

#include <iostream>

#include <fstream>

int main() {

std::ofstream file("example.txt");

if (file.is\_open()) {

file << "Hello, World!" << std::endl;

file.close(); }

else {

std::cout << "Failed to open the file." << std::endl; }

return 0; }

В этом примере создается объект std::ofstream, который представляет файл для записи. Метод is\_open() используется для проверки успешного открытия файла. Затем строка "Hello, World!" записывается в файл с помощью оператора <<. По завершении записи файл закрывается с помощью метода close().

Запись данных в файл:

#include <iostream>

#include <fstream>

int main() {

std::ofstream file("example.txt");

if (!file) {

std::cout << "Failed to open the file." << std::endl;

return 1; }

int number = 42;

file << "Number: " << number << std::endl;

file.close();

return 0; }

В этом примере используется класс std::ofstream для открытия файла "example.txt" и записи данных в него. Данные записываются с помощью оператора <<.

Добавление данных в конец файла:

#include <iostream>

#include <fstream>

int main() {

std::ofstream file("example.txt", std::ios::app);

if (!file) {

std::cout << "Failed to open the file." << std::endl;

return 1; }

int number = 42;

file << "Number: " << number << std::endl;

file.close();

return 0; }

В этом примере используется флаг std::ios::app при открытии файла с помощью std::ofstream, что позволяет добавлять данные в конец файла.

Проверка конца файла:

#include <iostream>

#include <fstream>

int main() {

std::ifstream file("example.txt");

if (!file) {

std::cout << "Failed to open the file." << std::endl;

return 1; }

int number; while (file >> number) {

std::cout << "Number: " << number << std::endl; }

file.close();

return 0; }

В этом примере цикл while используется для чтения чисел из файла до тех пор, пока файл не достигнет конца. Оператор >> используется для чтения данных из файла.

Это основные операции работы с файлами в языке C++. Библиотека <fstream> предоставляет и другие классы и функции для работы с файлами, такие как позиционирование в файле, переименование файлов

Потоки, разновидности потоков, поточные классы:

Потоки в языке C++ представляют потоки данных, такие как ввод из стандартного ввода, вывод в стандартный вывод или чтение/запись из/в файлы. В C++ потоки реализованы с помощью классов std::istream (для ввода данных), std::ostream (для вывода данных) и std::iostream (для ввода-вывода данных).

Разновидности потоков включают:

1. std::cin - поток ввода, связанный с стандартным вводом (клавиатурой).
2. std::cout - поток вывода, связанный с стандартным выводом (экраном).
3. std::cerr - поток вывода ошибок, связанный с стандартным выводом ошибок (экраном).
4. std::ifstream - поток ввода из файла.
5. std::ofstream - поток вывода в файл.
6. std::fstream - поток ввода-вывода из/в файл.

Эти потоки позволяют работать с различными источниками данных, такими как пользовательский ввод, файлы и другие.

Пример использования потоков ввода-вывода в C++:

#include <iostream>

#include <fstream>

int main() {

int num; std::string name;

std::cout << "Enter a number: ";

std::cin >> num;

std::cout << "Enter your name: ";

std::cin >> name;

std::ofstream file("output.txt");

file << "Number: " << num << std::endl;

file << "Name: " << name << std::endl;

file.close();

return 0; }

В этом примере пользователю предлагается ввести число и имя. Затем значения сохраняются в переменные num и name с помощью потока ввода std::cin. После этого значения записываются в файл "output.txt" с помощью потока вывода std::ofstream.

Таким образом, в языках C и C++ предоставляются мощные инструменты для организации ввода/вывода, форматированного ввода/вывода, работы с файлами и использования различных типов потоков данных.

* 1. **Программирование в среде Visual Studio Net**
     1. Понятие о в визуальном программировании. Проекты Visual Studio .Net. Работа в среде Visual Studio Net с формами и элементами управления в формах. Основные и дополнительные элементы управления, их свойства и методы.

Визуальное программирование относится к методу создания программ, при котором разработчик использует графический интерфейс (GUI) для создания программного кода. Один из популярных инструментов для визуального программирования - это среда разработки Visual Studio .NET, разработанная компанией Microsoft.

Visual Studio .NET предоставляет разработчикам мощный инструментарий для создания различных типов проектов, включая приложения для настольных компьютеров, веб-приложения, мобильные приложения и многое другое. Среда Visual Studio .NET поддерживает несколько языков программирования, включая C#, Visual Basic .NET, F# и другие.

Работа в среде Visual Studio .NET начинается с создания проекта. После создания проекта вы можете добавить формы в проект. Формы представляют собой графические окна, на которых размещаются элементы управления. С помощью форм вы можете создавать интерфейс пользователя для вашего приложения.

Visual Studio .NET предлагает широкий спектр элементов управления, которые могут быть размещены на формах. Некоторые из основных элементов управления включают:

1. Button (Кнопка): Используется для создания кнопок, при нажатии на которые выполняются определенные действия.
2. TextBox (Текстовое поле): Используется для ввода и отображения текста.
3. Label (Метка): Используется для отображения текстовой информации.
4. ComboBox (Выпадающий список): Используется для выбора одного из нескольких предопределенных значений из списка.
5. ListBox (Список): Используется для выбора одного или нескольких предопределенных значений из списка.

Это только несколько примеров основных элементов управления. В Visual Studio .NET также доступны дополнительные элементы управления, такие как чекбоксы, радиокнопки, изображения, таблицы и многое другое.

Каждый элемент управления имеет свои свойства и методы, которые можно настроить и использовать в программе. Например, свойство Text элемента управления TextBox позволяет получить или задать текст, отображаемый в текстовом поле. Методы элементов управления могут выполнять различные действия, например, метод MessageBox.Show() используется для отображения диалогового окна с сообщением.

Общая идея визуального программирования в Visual Studio .NET заключается в том, чтобы разместить нужные элементы управления на форме, настроить их свойства и методы, а затем написать код для обработки событий, которые происходят при взаимодействии пользователя с элементами управления. Это позволяет создавать интерактивные и пользовательские приложения с графическим интерфейсом.

* + 1. Переменные и массивы в Visual С# .Net и работа с ними.

В Visual C# .NET переменные используются для хранения и обработки данных. Переменная - это именованное место в памяти, которое может содержать значение определенного типа данных. Вот некоторые основные типы данных, которые можно использовать в Visual C# .NET:

Целочисленные типы данных:

* int: целые числа от -2,147,483,648 до 2,147,483,647.
* long: целые числа от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807.
* byte: целые числа от 0 до 255.

Вещественные типы данных:

* float: числа с плавающей точкой одинарной точности.
* double: числа с плавающей точкой двойной точности.

Символьные типы данных:

* char: символ Unicode.

Логический тип данных:

* bool: значение true или false.

Для объявления переменной в C# используется следующий синтаксис:

типДанных имяПеременной;

Например, чтобы объявить целочисленную переменную с именем "число", вы можете написать:

int число;

Массивы позволяют хранить наборы элементов одного типа данных. Каждый элемент в массиве имеет свой индекс, который позволяет получить доступ к этому элементу. Для объявления массива в C# используется следующий синтаксис:

типДанных[] имяМассива;

Например, чтобы объявить массив целых чисел с именем "массивЧисел", состоящий из 5 элементов, вы можете написать:

int[] массивЧисел = new int[5];

Вы можете получить доступ к элементам массива, используя индексы. Индексы начинаются с 0. Например, чтобы получить доступ к первому элементу массива, вы можете написать:

int первыйЭлемент = массивЧисел[0];

Массивы также могут быть инициализированы сразу с набором значений:

int[] массивЧисел = { 1, 2, 3, 4, 5 };

Это создаст массив "массивЧисел" с пятью элементами, содержащими значения от 1 до 5.

Вы также можете использовать различные методы и свойства для работы с массивами, такие как Length (длина массива) и методы для поиска, сортировки и изменения элементов массива.

Таким образом, переменные и массивы в Visual C# .NET позволяют хранить и оперировать данными в вашей программе, обеспечивая удобный способ управления информацией.

* + 1. Создание главного и контекстного меню в Visual С# .Net

В Visual C# .NET вы можете создавать главное меню (Main Menu) и контекстное меню (Context Menu) для добавления функциональности и управления вашим приложением. Главное меню обычно размещается в верхней части окна приложения и содержит основные команды, а контекстное меню появляется при щелчке правой кнопкой мыши и предлагает контекстуальные команды, связанные с выбранным элементом интерфейса.

Вот как создать главное и контекстное меню в Visual C# .NET:

Создание главного меню:

* В среде разработки Visual Studio .NET откройте форму, на которой вы хотите разместить главное меню.
* Перейдите на панель инструментов и найдите компонент "MenuStrip". Перетащите его на форму.
* На "MenuStrip" добавьте элементы меню, используя контрол "ToolStripMenuItem". Настройте свойства каждого элемента меню, такие как текст, иконка и обработчик событий при выборе элемента меню.
* Можете использовать вложенные меню, добавляя элементы "ToolStripMenuItem" как дочерние элементы других элементов меню.
* Для связывания главного меню с формой вы можете установить свойство "MainMenuStrip" формы на созданный "MenuStrip" компонент.

Создание контекстного меню:

* В среде разработки Visual Studio .NET откройте форму, на которой вы хотите добавить контекстное меню.
* Перейдите на панель инструментов и найдите компонент "ContextMenuStrip". Перетащите его на форму.
* На "ContextMenuStrip" добавьте элементы контекстного меню, используя контрол "ToolStripMenuItem". Настройте свойства каждого элемента меню, такие как текст, иконка и обработчик событий при выборе элемента меню.
* Можете использовать вложенные меню, добавляя элементы "ToolStripMenuItem" как дочерние элементы других элементов меню.
* Для привязки контекстного меню к элементу интерфейса (например, кнопке или текстовому полю), выделите элемент и в свойствах установите свойство "ContextMenuStrip" на созданный "ContextMenuStrip" компонент.

После создания и настройки меню вы можете определить обработчики событий для выбора элементов меню. Это позволяет выполнять определенные действия при выборе пользователем команд из меню.

Например, чтобы определить обработчик события для выбора элемента меню "Выход", вы можете использовать следующий код:

private void выходToolStripMenuItem\_Click(object sender, EventArgs e) {

// Код, выполняемый при выборе элемента меню "Выход"

// Например, закрытие приложения

this.Close(); }

Таким образом, создание главного и контекстного меню в Visual C# .NET позволяет вам добавлять удобный интерфейс управления вашим приложением и реализовывать функциональность, соответствующую потребностям пользователя.

* + 1. Основные операторы языка Visual С# .Net

В языке Visual C# .NET доступно множество операторов для выполнения различных действий, таких как присваивание значений переменным, выполнение математических операций, управление потоком выполнения и другие. Вот некоторые основные операторы языка Visual C# .NET:

Операторы присваивания:

* "=": присваивает значение справа от оператора переменной слева от оператора.
* "+=": прибавляет значение справа к переменной слева и присваивает результат переменной слева.

Арифметические операторы:

* "+": сложение.
* "-": вычитание.
* "\*": умножение.
* "/": деление.
* "%": остаток от деления.

Операторы сравнения:

* "==": равно.
* "!=": не равно.
* "<": меньше.
* ">": больше.
* "<=": меньше или равно.
* ">=": больше или равно.

Логические операторы:

* "&&": логическое И (AND).
* "||": логическое ИЛИ (OR).
* "!": логическое отрицание (NOT).

Операторы управления потоком выполнения:

* "if-else": условный оператор, выполняющий код в зависимости от условия.
* "switch": оператор выбора, позволяющий выполнить различный код в зависимости от значения переменной.
* "for": цикл, выполняющий повторяющиеся действия определенное количество раз.
* "while": цикл, выполняющий повторяющиеся действия, пока условие истинно.
* "do-while": цикл, выполняющий повторяющиеся действия, пока условие истинно, с проверкой условия в конце цикла.

Операторы приведения типов:

* "(тип)": приведение значения к указанному типу.
* "as": безопасное приведение типа, возвращает null, если приведение невозможно.

Операторы строки:

* "+": конкатенация строк.
* "string.Format": форматирование строк.

Операторы работы с массивами и коллекциями:

* "[]": доступ к элементам массива по индексу.
* "foreach": цикл, перебирающий элементы в коллекции.

Операторы ввода-вывода:

* "Console.ReadLine()": чтение строки с консоли.
* "Console.WriteLine()": вывод строки на консоль.

Это лишь некоторые основные операторы в языке Visual C# .NET. Существуют и другие операторы и конструкции, которые позволяют вам контролировать выполнение кода и выполнять различные операции.

* + 1. Создание многооконных приложений в Visual С# .Net

В Visual C# .NET вы можете создавать многооконные приложения, которые позволяют отображать и управлять несколькими окнами одновременно. Это полезно, когда вам нужно разделить функциональность приложения на разные части или позволить пользователям одновременно работать с разными видами данных. Вот некоторые шаги по созданию многооконных приложений в Visual C# .NET:

Создание главной формы:

* Создайте новый проект Windows Forms Application в среде разработки Visual Studio .NET.
* В Solution Explorer выберите проект и выберите "Add" > "Windows Form" для создания новой формы.
* Разместите элементы управления и настройте интерфейс вашей главной формы, включая меню, кнопки и другие элементы, если это необходимо.

Создание дополнительных форм:

* В Solution Explorer выберите проект и выберите "Add" > "Windows Form" для создания новой формы.
* Настройте интерфейс дополнительной формы, добавляя элементы управления и определяя их свойства и обработчики событий.
* Повторите этот шаг для каждой дополнительной формы, которую вы хотите создать.

Открытие дополнительных форм из главной формы:

* В главной форме определите обработчик события, например, при нажатии кнопки или выборе элемента меню, который будет открывать дополнительную форму.
* В обработчике события создайте экземпляр дополнительной формы и вызовите ее метод Show() или ShowDialog() для отображения формы.
* Вы можете передавать данные или параметры в конструктор дополнительной формы, чтобы передать информацию между формами.

Управление между формами:

* В каждой дополнительной форме добавьте кнопки или элементы управления, чтобы пользователь мог переключаться между формами.
* Определите обработчики событий для этих элементов управления, которые будут открывать или закрывать соответствующие формы.

Передача данных между формами:

* Для передачи данных между формами можно использовать свойства или методы ваших форм.
* Вы можете передавать данные из главной формы в дополнительную форму при ее создании, используя параметры конструктора или публичные свойства.
* Для передачи данных обратно из дополнительной формы в главную форму вы можете использовать обработчики событий или методы обновления данных.

Управление жизненным циклом форм:

* При работе с многооконными приложениями обратите внимание на жизненный цикл форм.
* Используйте методы, такие как Form\_Load, Form\_Activated и Form\_Closed, для выполнения действий при открытии, активации или закрытии формы.

Это лишь общий обзор процесса создания многооконных приложений в Visual C# .NET. В действительности, более сложные сценарии многооконных приложений могут потребовать более продвинутых подходов, таких как использование паттерна MVP (Model-View-Presenter) или MVVM (Model-View-ViewModel).

* + 1. Работа с файлами данных в Visual С# .Net

Работа с файлами данных в Visual C# .NET позволяет вам сохранять и загружать данные из файлов на диске. Это может быть полезно для хранения пользовательской информации, настроек приложения, данных приложения и других типов данных. Вот некоторые основные шаги по работе с файлами данных в Visual C# .NET:

Запись данных в файл:

* Используйте классы из пространства имен System.IO для работы с файлами. Например, классы FileStream, StreamWriter или BinaryWriter.
* Откройте файл для записи с помощью методов классов FileStream или StreamWriter. Вы можете указать путь к файлу и режим открытия (например, FileMode.Create, FileMode.Append).
* Используйте методы записи классов FileStream, StreamWriter или BinaryWriter для записи данных в файл. Например, Write, WriteLine, WriteBytes.
* Не забудьте закрыть файл после записи, вызвав методы Close или Dispose для объектов FileStream или StreamWriter.

Пример записи строки в текстовый файл:

using System.IO;

string filePath = "path/to/file.txt"; // Создание или перезапись файла

using (StreamWriter writer = new StreamWriter(filePath)) {

writer.WriteLine("Hello, World!");

writer.WriteLine("This is a line of text."); // Добавьте свой код записи данных }

Чтение данных из файла:

* Используйте классы из пространства имен System.IO для работы с файлами. Например, классы FileStream, StreamReader или BinaryReader.
* Откройте файл для чтения с помощью методов классов FileStream или StreamReader. Вы можете указать путь к файлу и режим открытия (например, FileMode.Open).
* Используйте методы чтения классов FileStream, StreamReader или BinaryReader для чтения данных из файла. Например, Read, ReadLine, ReadBytes.
* Не забудьте закрыть файл после чтения, вызвав методы Close или Dispose для объектов FileStream или StreamReader.

Пример чтения данных из текстового файла:

using System.IO;

string filePath = "path/to/file.txt";

if (File.Exists(filePath)) {

using (StreamReader reader = new StreamReader(filePath)) {

string line; while ((line = reader.ReadLine()) != null) {

// Обработка прочитанной строки

Console.WriteLine(line); }

// Добавьте свой код чтения данных } }

Работа с различными типами файлов:

* В зависимости от типа данных, с которыми вы работаете, могут потребоваться различные классы для чтения и записи данных. Например, для работы с бинарными данными вы можете использовать классы BinaryReader и BinaryWriter, а для работы с XML-документами вы можете использовать классы XmlReader и XmlWriter.

Обработка исключений:

* При работе с файлами всегда следует учитывать возможность возникновения ошибок, таких как отсутствие файла, недоступность файла или ошибки ввода-вывода.
* Рекомендуется использовать конструкцию try-catch-finally для обработки исключений и корректного закрытия файлов.

Это лишь общий обзор работы с файлами данных в Visual C# .NET. В зависимости от ваших потребностей и типа данных могут потребоваться более специфические подходы и классы для работы с файлами.

* 1. **Web программирование**
     1. Архитектура ADO.NET. Провайдеры баз данных и компоненты организации доступа к базам данных.

ADO.NET (ActiveX Data Objects for .NET) — это набор технологий, предоставляющих доступ к базам данных из приложений .NET. Он предоставляет программистам инструменты и компоненты для работы с различными источниками данных, включая реляционные базы данных, XML-документы и службы веб-серверов.

Архитектура ADO.NET включает следующие основные компоненты:

1. Connection (Подключение): Этот компонент предоставляет средства для установления соединения с базой данных. Он представлен классом SqlConnection и обеспечивает возможности для открытия и закрытия подключения, выполнения команд и управления транзакциями.
2. Command (Команда): Компонент Command предоставляет средства для выполнения SQL-запросов и вызовов хранимых процедур в базе данных. Он представлен классом SqlCommand и позволяет задавать SQL-запросы, передавать параметры и получать результаты запроса.
3. DataReader (Читатель данных): DataReader позволяет последовательно читать данные из базы данных. Он представлен классом SqlDataReader и предоставляет возможность эффективного чтения больших объемов данных в режиме только для чтения.
4. DataAdapter (Адаптер данных): DataAdapter обеспечивает мост между DataSet (набор данных) и источником данных. Он предоставляет методы для извлечения данных из базы данных и заполнения DataSet. DataAdapter также может выполнять обновление изменений из DataSet обратно в базу данных.
5. DataSet (Набор данных): DataSet представляет в памяти кэш данных, который можно использовать для хранения и обработки данных независимо от базы данных. Он содержит таблицы, столбцы и отношения между таблицами. DataSet позволяет выполнять операции фильтрации, сортировки и изменения данных в памяти.

Провайдеры баз данных в ADO.NET представляют собой наборы классов, специфичных для конкретной базы данных, которые позволяют взаимодействовать с этой базой данных. Примерами провайдеров являются:

* System.Data.SqlClient для работы с Microsoft SQL Server.
* System.Data.OleDb для работы с базами данных, доступными через OLE DB.
* System.Data.Odbc для работы с базами данных, доступными через ODBC.
* System.Data.OracleClient для работы с базами данных Oracle.
* System.Data.SQLite для работы с базами данных SQLite.

Каждый провайдер предоставляет собственные классы, такие как Connection, Command, DataReader и DataAdapter, специфические для работы с конкретной базой данных. Эти классы предоставляют функциональность, соответствующую возможностям и особенностям соответствующей базы данных.

С использованием ADO.NET и провайдеров баз данных, разработчики могут создавать мощные приложения .NET, взаимодействующие с различными источниками данных и обеспечивающие эффективное управление данными.

* + 1. Visual Studio Net. Связывание визуальных элементов WINDOWS-форм с таблицами баз данных. Добавление и удаление записей в таблицах баз данных с помощью Visual Studio Net. Обработка исключительных ситуаций при работе с базами данных.

Visual Studio .NET предоставляет различные инструменты и функциональности для связывания визуальных элементов Windows Forms с таблицами баз данных, добавления и удаления записей, а также обработки исключительных ситуаций при работе с базами данных. Ниже приведены основные подходы и компоненты, используемые в Visual Studio .NET для этих целей:

Связывание визуальных элементов Windows Forms с таблицами баз данных:

* DataSet Designer: Visual Studio .NET предоставляет инструмент DataSet Designer, который позволяет создавать наборы данных (DataSets) и связывать их с таблицами баз данных. DataSet Designer позволяет визуально настроить структуру набора данных, добавить таблицы, определить отношения между ними и настроить свойства столбцов.
* Data Source Window: Data Source Window позволяет выбирать и просматривать доступные источники данных, включая базы данных. Выбрав таблицу базы данных, можно перетащить ее на форму Windows Forms, чтобы автоматически создать связанные с ней визуальные элементы, такие как DataGridView (таблица данных) или TextBox (поля данных).
* Data Binding контролы: Visual Studio .NET предоставляет различные контролы, такие как DataGridView, ListBox, ComboBox и другие, которые могут быть связаны с таблицами баз данных. С помощью свойств и настроек контролов можно определить источник данных и поля для отображения.

Добавление и удаление записей в таблицах баз данных:

* DataAdapter: DataAdapter предоставляет функциональность для извлечения данных из базы данных и обновления их обратно в базу данных. Он автоматически генерирует команды (вставка, обновление, удаление) на основе изменений в DataSet. С помощью методов DataAdapter, таких как Fill (заполнение) и Update (обновление), можно добавлять и удалять записи в таблицах баз данных.
* SqlCommand: Класс SqlCommand позволяет выполнить SQL-запросы для добавления и удаления записей в таблицах баз данных. С помощью методов SqlCommand, таких как ExecuteNonQuery, можно выполнить команды SQL, включая вставку и удаление записей.

Обработка исключительных ситуаций при работе с базами данных:

* Try-Catch блоки: Visual Studio .NET позволяет использовать конструкцию Try-Catch для перехвата и обработки исключений при работе с базами данных. Внутри блока Try можно выполнить операции с базой данных, а в блоке Catch можно обработать возможные исключительные ситуации, например, отсутствие подключения к базе данных или ошибки выполнения SQL-запроса.
* Exception Handling: Visual Studio .NET предоставляет возможности для генерации и обработки исключений при работе с базами данных. Исключительные ситуации могут быть обработаны с помощью механизма исключений в C# или VB.NET. Вы можете использовать try-catch блоки или обработчики исключений для выполнения соответствующих действий при возникновении ошибок, связанных с базой данных.

С помощью этих инструментов и подходов в Visual Studio .NET разработчики могут легко связывать визуальные элементы Windows Forms с таблицами баз данных, добавлять и удалять записи и обрабатывать исключительные ситуации, обеспечивая надежную работу приложений, взаимодействующих с базами данных.

* + 1. Использование статических и динамических запросов в ADO.NET и Visual Studio Net.

В ADO.NET и Visual Studio .NET можно использовать как статические, так и динамические запросы для работы с базами данных.

Статические запросы:

Статические запросы представляют собой SQL-запросы, определенные заранее в коде программы или во внешних файловых ресурсах. Они создаются как строки, содержащие SQL-код, и передаются соответствующим объектам ADO.NET для выполнения. Некоторые способы использования статических запросов в ADO.NET и Visual Studio .NET включают:

* Создание и использование объекта SqlCommand: Можно создать экземпляр класса SqlCommand, установить свойство CommandText в SQL-запрос и затем выполнить его с помощью метода ExecuteNonQuery() (для изменяющих операций) или ExecuteReader() (для операций выборки).

Пример использования статического запроса с SqlCommand:

string queryString = "SELECT \* FROM Customers WHERE Country = @Country";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(queryString, connection);

command.Parameters.AddWithValue("@Country", "USA");

connection.Open();

SqlDataReader reader = command.ExecuteReader();

// Чтение данных из SqlDataReader

}

* Использование DataAdapter с предварительно заданными командами: DataAdapter позволяет определить заранее созданные команды (вставка, обновление, удаление и выборка) и использовать их для взаимодействия с базой данных.

Пример использования статического запроса с DataAdapter:

string queryString = "SELECT \* FROM Customers";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlDataAdapter adapter = new SqlDataAdapter();

adapter.SelectCommand = new SqlCommand(queryString, connection);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, "Customers");

// Работа с данными из DataSet

}

Динамические запросы:

Динамические запросы формируются и выполняются во время выполнения программы, основываясь на динамически изменяемых параметрах или условиях. Они могут быть полезными в случаях, когда необходимо генерировать SQL-запросы на основе пользовательского ввода или динамических данных. В ADO.NET и Visual Studio .NET можно использовать следующие подходы для работы с динамическими запросами:

* Использование параметризованных запросов: Параметризованные запросы позволяют вставлять переменные или параметры в SQL-запросы, чтобы избежать проблем с безопасностью и предотвратить атаки SQL-инъекций.

Пример использования параметризованного запроса с SqlCommand:

string queryString = "SELECT \* FROM Customers WHERE Country = @Country";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(queryString, connection);

command.Parameters.AddWithValue("@Country", userInput);

connection.Open(); SqlDataReader reader = command.ExecuteReader();

// Чтение данных из SqlDataReader

}

* Использование строковых операций и метода ExecuteNonQuery(): Можно динамически формировать SQL-запросы, используя строковые операции, и выполнить их с помощью метода ExecuteNonQuery().

Пример использования динамического запроса с SqlCommand и ExecuteNonQuery():

string dynamicQuery = "INSERT INTO Customers (Name, Email) VALUES ('" + name + "', '" + email + "')";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(dynamicQuery, connection);

connection.Open();

int rowsAffected = command.ExecuteNonQuery();

}

Обратите внимание, что при использовании динамических запросов необходимо быть осторожными и следить за безопасностью данных, чтобы избежать возможных атак или ошибок в запросах. Рекомендуется использовать параметризованные запросы или другие механизмы защиты данных при работе с динамическими запросами.

* + 1. Создание отчётов с помощью программы Crystal Reports в Visual Studio Net.

Crystal Reports - это инструмент, входящий в состав Visual Studio .NET, который позволяет разработчикам создавать отчеты на основе данных из различных источников данных. Crystal Reports обеспечивает широкие возможности для создания и настройки профессиональных отчетов с различными элементами дизайна, такими как таблицы, графики, подробности, заголовки, футеры и многое другое. Ниже приведены основные шаги для создания отчетов с помощью Crystal Reports в Visual Studio .NET:

1. Установка Crystal Reports: Убедитесь, что в вашей установке Visual Studio .NET присутствует Crystal Reports. Если его нет, вы можете установить его с помощью инсталляционного пакета Crystal Reports для Visual Studio.
2. Создание проекта отчета: В Visual Studio .NET создайте новый проект отчета Crystal Reports. Для этого выберите соответствующий шаблон проекта в разделе Business Intelligence или Reporting.
3. Указание источника данных: Выберите источник данных для отчета. Crystal Reports поддерживает различные источники данных, включая базы данных, XML-файлы, объекты данных и другие. Вы можете указать соединение с базой данных или другой источник данных, используя Crystal Reports Designer.
4. Создание макета отчета: В Crystal Reports Designer вы можете создавать макет отчета, добавлять и настраивать элементы дизайна, такие как таблицы, поля данных, заголовки, футеры, графики и т.д. Можно настроить форматирование, сортировку, фильтрацию и группировку данных в отчете.
5. Привязка данных: Привяжите данные из выбранного источника данных к элементам отчета. Например, вы можете связать таблицу данных с таблицей в отчете, указав соответствующие поля данных.
6. Настройка параметров отчета: При необходимости можно добавить параметры к отчету, чтобы позволить пользователям вводить значения при запуске отчета. Параметры можно использовать для фильтрации данных или управления содержимым отчета.
7. Генерация отчета: После завершения разработки отчета можно его сгенерировать и проверить. В Visual Studio .NET вы можете просмотреть отчет, чтобы убедиться, что он отображает данные и элементы дизайна корректно.
8. Интеграция отчета в приложение: После создания отчета можно интегрировать его в ваше приложение .NET. Crystal Reports предоставляет API и компоненты для загрузки, отображения и печати отчетов в вашем приложении.

Crystal Reports в Visual Studio .NET предоставляет мощные инструменты для создания профессиональных отчетов с настраиваемым дизайном и возможностями работы с различными источниками данных.

* + 1. Работа с базой данных SQL Server с помощью утилиты Server Explorer из состава Visual Studio Net: создание, модификация, удаление таблиц, представлений и хранимых процедур.

Утилита Server Explorer в Visual Studio .NET предоставляет удобный интерфейс для работы с базой данных SQL Server, включая создание, модификацию и удаление таблиц, представлений и хранимых процедур. Ниже описаны основные шаги для работы с базой данных SQL Server с помощью Server Explorer:

1. Открытие Server Explorer: В Visual Studio .NET выберите в меню "View" (Вид) пункт "Server Explorer" (Обозреватель серверов) или используйте сочетание клавиш "Ctrl+Alt+S". Откроется панель Server Explorer.
2. Добавление подключения к базе данных: В панели Server Explorer щелкните правой кнопкой мыши на "Data Connections" (Подключения к данным) и выберите "Add Connection" (Добавить подключение). В появившемся диалоговом окне выберите "Microsoft SQL Server" (Microsoft SQL Server) в качестве провайдера данных и укажите параметры подключения к базе данных SQL Server.
3. Просмотр исходной структуры базы данных: После успешного подключения к базе данных SQL Server в Server Explorer отобразятся дерево объектов базы данных, такие как таблицы, представления, хранимые процедуры и другие. Вы можете раскрывать соответствующие разделы, чтобы просмотреть объекты базы данных.
4. Создание таблицы: Щелкните правой кнопкой мыши на разделе "Tables" (Таблицы) и выберите "Add New Table" (Добавить новую таблицу). В открывшемся окне дизайнера таблицы вы можете определить столбцы, их типы данных, ограничения и другие свойства таблицы.
5. Модификация таблицы: Чтобы изменить существующую таблицу, щелкните правой кнопкой мыши на таблице в Server Explorer и выберите "Open Table Definition" (Открыть определение таблицы). В окне дизайнера таблицы вы можете добавить, изменить или удалить столбцы, а также внести другие изменения в структуру таблицы.
6. Удаление таблицы: Щелкните правой кнопкой мыши на таблице, которую вы хотите удалить, и выберите "Delete" (Удалить). Появится запрос для подтверждения удаления таблицы. Обратите внимание, что удаление таблицы приведет к потере всех данных, хранящихся в ней.
7. Создание представления: Щелкните правой кнопкой мыши на разделе "Views" (Представления) и выберите "Add New View" (Добавить новое представление). В окне дизайнера представления вы можете определить запрос SQL, который будет использоваться для создания представления.
8. Создание хранимой процедуры: Щелкните правой кнопкой мыши на разделе "Stored Procedures" (Хранимые процедуры) и выберите "Add New Stored Procedure" (Добавить новую хранимую процедуру). В окне редактора хранимой процедуры вы можете определить код SQL для логики хранимой процедуры.

Обратите внимание, что Server Explorer предоставляет простой способ работы с базой данных SQL Server изнутри Visual Studio .NET, однако для более сложных операций с базой данных может потребоваться использование более мощных инструментов, таких как SQL Server Management Studio.

* + 1. Работа с представлениями в клиентских приложениях Visual Studio Net.Работа с хранимыми процедурами в клиентских приложениях Visual Studio Net.

Работа с представлениями в клиентских приложениях Visual Studio .NET позволяет использовать данные, полученные из представлений базы данных, для отображения информации и выполнения различных операций. Вот основные шаги для работы с представлениями в клиентских приложениях:

1. Подключение к базе данных: В вашем клиентском приложении Visual Studio .NET создайте соединение с базой данных, используя соответствующий провайдер базы данных (например, ADO.NET).
2. Запрос данных из представления: С использованием соединения с базой данных и SQL-запросов вы можете выполнить запрос данных из представления. Например, вы можете использовать SqlDataAdapter для выполнения запроса SELECT из представления и получения набора данных.

Пример работы с представлением с использованием SqlDataAdapter:

string queryString = "SELECT \* FROM YourView";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, "YourViewData");

// Используйте полученные данные в вашем приложении

}

1. Отображение данных представления: Полученные данные из представления можно использовать для отображения информации в пользовательском интерфейсе вашего приложения. Например, вы можете привязать данные к элементам управления, таким как DataGridView, ListBox или ListView, для их отображения.

Пример привязки данных к DataGridView:

dataGridView.DataSource = dataSet.Tables["YourViewData"];

1. Выполнение операций на основе данных представления: При необходимости вы можете позволить пользователям выполнять операции, такие как добавление, изменение или удаление записей, на основе данных из представления. Для этого вам может понадобиться дополнительная логика и элементы управления в вашем приложении.

Например, для добавления записи в представление вы можете использовать SqlCommand и выполнить операцию INSERT:

string insertQuery = "INSERT INTO YourView (Column1, Column2) VALUES (@Value1, @Value2)";

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand(insertQuery, connection);

command.Parameters.AddWithValue("@Value1", value1);

command.Parameters.AddWithValue("@Value2", value2); connection.Open();

int rowsAffected = command.ExecuteNonQuery();

// Обработка результата операции

}

Аналогично можно выполнять операции изменения и удаления записей в представлении, используя соответствующие SQL-запросы.

Что касается работы с хранимыми процедурами в клиентских приложениях Visual Studio .NET, вы можете использовать тот же подход, что и для представлений, но вместо SQL-запросов выполнять вызовы хранимых процедур базы данных с использованием SqlCommand. Вы можете передавать параметры хранимой процедуре и получать результаты ее выполнения.

Пример вызова хранимой процедуры с использованием SqlCommand:

using (SqlConnection connection = new SqlConnection(connectionString))

{

SqlCommand command = new SqlCommand("YourStoredProcedure", connection);

command.CommandType = CommandType.StoredProcedure;

command.Parameters.AddWithValue("@Parameter1", value1);

command.Parameters.AddWithValue("@Parameter2", value2);

connection.Open();

int rowsAffected = command.ExecuteNonQuery();

// Обработка результата операции

}

Таким образом, вы можете взаимодействовать с представлениями и хранимыми процедурами базы данных в вашем клиентском приложении Visual Studio .NET, используя соответствующие инструменты ADO.NET и SQL-запросы или вызовы хранимых процедур.

* + 1. Применение Internet технологий. Клиентские сценарии. Примеры.

Клиентские сценарии - это взаимодействие пользователей с веб-приложениями или веб-сайтами, которое происходит на стороне клиента, то есть на устройстве пользователя. Вот несколько примеров клиентских сценариев, которые используют интернет-технологии:

1. Онлайн-покупки: Пользователь посещает интернет-магазин, просматривает каталог товаров, выбирает нужный товар, добавляет его в корзину, оформляет заказ и производит оплату онлайн. Это включает в себя взаимодействие с интерфейсом веб-страницы, отправку данных на сервер, обработку платежа и получение подтверждения заказа.
2. Социальные сети: Пользователь входит в свою учетную запись в социальной сети, просматривает новости и обновления своих друзей, комментирует их посты, загружает фотографии, отправляет личные сообщения и участвует в групповых дискуссиях. Это включает в себя взаимодействие с интерфейсом социальной сети, отправку данных на сервер и получение обновлений контента.
3. Онлайн-банкинг: Пользователь входит в интернет-банк, просматривает баланс своего счета, совершает переводы между счетами, оплачивает счета, устанавливает уведомления и запрашивает выписки. Это включает в себя взаимодействие с интерфейсом банковского веб-приложения, отправку данных на сервер и получение информации о банковских операциях.
4. Поиск и резервирование билетов: Пользователь ищет и бронирует билеты на самолет, поезд, автобус или другие виды транспорта онлайн. Он вводит данные о поездке, выбирает оптимальные варианты, оплачивает билеты и получает подтверждение бронирования. Это включает в себя взаимодействие с интерфейсом поискового и бронировочного сервиса, отправку данных на сервер и получение билетов.
5. Видео- и аудио-стриминг: Пользователь просматривает видео или слушает аудио-материалы онлайн. Например, он может просматривать фильмы или сериалы на платформе видео-стриминга, слушать музыку на стриминговом сервисе или проводить видеоконференции через приложения для видеосвязи. Это включает в себя загрузку и воспроизведение мультимедийного контента через интерфейс веб-приложения или приложения.
6. Онлайн-игры: Пользователь играет в онлайн-игры, взаимодействует с другими игроками, выполняет задания, получает достижения и обменивается сообщениями внутри игрового мира. Это включает в себя взаимодействие с интерфейсом игры через интернет, отправку и получение данных, синхронизацию действий с другими игроками и обновление игрового состояния.

Это лишь некоторые примеры клиентских сценариев, которые используют интернет-технологии. Все они основаны на взаимодействии пользователя с веб-приложениями или веб-сайтами через интерфейс браузера или специальные клиентские приложения.

* + 1. Применение Internet технологий. Серверные сценарии. Примеры.

Применение интернет-технологий включает не только клиентские сценарии, но и серверные сценарии, которые выполняются на сервере и обрабатывают запросы от клиентов. Вот несколько примеров серверных сценариев:

1. Обработка форм и отправка данных: Серверные сценарии могут обрабатывать данные, отправленные клиентами через формы на веб-страницах. Например, сервер может принимать данные, введенные пользователем в форму заказа товара, проводить проверку и обрабатывать эти данные, сохраняя заказ в базе данных или отправляя уведомление.
2. Аутентификация и авторизация: Серверные сценарии позволяют проверять подлинность пользователей и предоставлять им доступ к определенным ресурсам в зависимости от их прав доступа. Например, сервер может проверять учетные данные пользователя при попытке входа в систему и предоставлять доступ к защищенным страницам или функциям только для аутентифицированных пользователей.
3. Обработка запросов API: Серверные сценарии могут обрабатывать запросы API от клиентских приложений или веб-служб. Например, сервер может предоставлять API для получения данных о товарах, обработки платежей, отправки уведомлений и т.д. Клиенты могут отправлять запросы к этому API, и сервер будет обрабатывать эти запросы, возвращая соответствующие данные или выполняя необходимые операции.
4. Генерация динамического контента: Серверные сценарии позволяют генерировать динамический контент для клиентов. Например, сервер может получать данные из базы данных, обрабатывать их и генерировать веб-страницу с динамически подставляемыми данными, такими как новости, список товаров или результаты поиска.
5. Обработка загрузки файлов: Серверные сценарии могут обрабатывать загрузку файлов от клиентов. Например, сервер может принимать файлы, загруженные пользователями через веб-интерфейс, сохранять их на сервере и выполнять дополнительные операции, такие как обработка изображений или сохранение информации о файлах в базе данных.
6. Планирование и автоматизация задач: Серверные сценарии могут выполнять планирование и автоматизацию задач на сервере. Например, сервер может выполнять регулярные резервные копии базы данных, обновление данных из внешних источников, отправку писем или выполнение других задач по расписанию.

Это лишь некоторые примеры серверных сценариев, которые могут быть реализованы с использованием интернет-технологий. Они помогают серверам обрабатывать запросы от клиентов, предоставлять нужные данные и функциональность, а также выполнять автоматизированные задачи для обеспечения бесперебойной работы системы.

* + 1. Применение Internet технологий. Работа с базами данных. Примеры.

Применение интернет-технологий в работе с базами данных предоставляет возможности для хранения, управления и обработки данных с использованием сети Интернет. Вот несколько примеров применения интернет-технологий при работе с базами данных:

1. Веб-приложения с базами данных: Создание и развертывание веб-приложений, которые взаимодействуют с базами данных. Например, онлайн-магазины, социальные сети, блоги и форумы, которые хранят информацию о пользователях, продуктах, сообщениях и других данных в базе данных.
2. Облачные базы данных: Использование облачных сервисов для хранения и обработки данных. Облачные базы данных позволяют разработчикам и организациям масштабировать и управлять базами данных через Интернет без необходимости устанавливать и поддерживать собственную инфраструктуру.
3. API баз данных: Создание API (интерфейсов программного обеспечения) для взаимодействия с базами данных через сеть. API баз данных позволяют разработчикам создавать приложения, которые могут получать, изменять и удалять данные из базы данных с использованием запросов API через Интернет.
4. Репликация и синхронизация данных: Синхронизация данных между разными базами данных и устройствами через Интернет. Например, мобильные приложения могут синхронизировать данные с центральной базой данных, чтобы пользователь имел доступ к актуальным данным на разных устройствах.
5. Аналитика и отчетность: Использование интернет-технологий для анализа данных и создания отчетов. Можно использовать специализированные инструменты и платформы для сбора, обработки и визуализации данных из баз данных, а также для создания интерактивных отчетов и дашбордов, доступных через Интернет.
6. Интеграция с внешними сервисами: Взаимодействие с внешними сервисами и API для обогащения данных из базы данных. Например, приложение может использовать API геолокации для добавления информации о местоположении к данным из базы данных или интегрироваться с платежными системами для обработки платежей.

Это лишь некоторые примеры применения интернет-технологий при работе с базами данных. Использование интернет-технологий расширяет возможности для доступа, обработки и управления данными через Интернет, что позволяет создавать более гибкие и масштабируемые приложения.

* + 1. Применение Internet технологий. Разработка и инсталляция Web приложения. Работа с БД.

Применение интернет-технологий при разработке и установке веб-приложений с базами данных предоставляет возможности для создания функциональных и динамических веб-сайтов, которые взаимодействуют с данными. Вот некоторые этапы и примеры работы с базами данных при разработке и установке веб-приложений:

1. Проектирование базы данных: Определите структуру базы данных, включая таблицы, связи между ними и атрибуты данных. Определите, какие данные будут храниться и как они будут организованы.
2. Создание базы данных: Используйте подходящую систему управления базами данных (СУБД), такую как MySQL, PostgreSQL, Microsoft SQL Server или другую, для создания базы данных. Создайте таблицы, определите поля и связи между таблицами.
3. Подключение к базе данных: В веб-приложении установите соединение с базой данных, используя соответствующий провайдер баз данных и настройки подключения. Это обычно включает указание хоста, порта, имени пользователя, пароля и имени базы данных.
4. Выполнение запросов к базе данных: Используйте язык запросов, такой как SQL (Structured Query Language), для выполнения операций с базой данных. Это может быть выборка данных, добавление новых записей, обновление существующих записей или удаление записей.
5. Обработка данных на сервере: Веб-приложение может выполнять различные операции с данными на сервере, например, проводить валидацию данных, фильтровать и сортировать результаты запросов, вычислять агрегированные значения или преобразовывать данные в нужный формат.
6. Представление данных на клиенте: Веб-приложение может отображать данные из базы данных на клиентской стороне, используя язык разметки, такой как HTML, и язык стилей, такой как CSS. Данные могут быть представлены в виде таблиц, списков, графиков или других элементов интерфейса.
7. Обработка запросов от клиента: Веб-приложение должно обрабатывать запросы от клиента, например, получение данных от пользователя через формы, обработка их на сервере и внесение изменений в базу данных на основе этих запросов.
8. Обеспечение безопасности данных: Обеспечьте защиту данных, используя соответствующие меры безопасности, такие как проверка прав доступа, шифрование данных и предотвращение атак на базу данных.
9. Тестирование и отладка: Проведите тестирование веб-приложения, включая проверку правильности работы с базой данных. Отладьте и исправьте возможные ошибки или проблемы, связанные с базой данных.
10. Установка веб-приложения: Разверните веб-приложение на сервере, настроив соединение с базой данных и предоставив необходимые ресурсы (файлы, конфигурацию и т. д.) для его работоспособности.

Примеры применения интернет-технологий при работе с базами данных в веб-приложениях включают онлайн-магазины, социальные сети, системы управления контентом, блоги, форумы и многое другое. Эти приложения используют базу данных для хранения и обработки информации, обеспечивая взаимодействие пользователя с данными через интерфейс веб-сайта или приложения.

* + 1. Применение Internet технологий. Гипертекст, язык HTML, протокол HTTP.

Применение интернет-технологий включает использование гипертекста, языка HTML (HyperText Markup Language) и протокола HTTP (Hypertext Transfer Protocol). Вот краткое описание каждого из них:

1. Гипертекст: Гипертекст - это метод организации информации, позволяющий создавать связи между различными элементами текста. Он позволяет пользователям переходить с одной части текста на другую посредством гиперссылок. Гипертекст используется для создания веб-страниц, которые содержат гиперссылки на другие страницы, изображения, аудио, видео и другой мультимедийный контент.
2. HTML: HTML является языком разметки, используемым для создания веб-страниц. Он определяет структуру и содержимое веб-документа с помощью тегов и атрибутов. HTML-документы состоят из различных элементов, таких как заголовки, параграфы, списки, таблицы, изображения и формы. С помощью HTML можно создавать интерактивные веб-страницы, отображать текст, изображения, видео, аудио и другой контент.
3. HTTP: HTTP - это протокол передачи данных, который используется для обмена информацией между клиентом (обычно веб-браузером) и сервером. HTTP определяет правила и соглашения для запросов клиента и ответов сервера. Клиент отправляет HTTP-запросы на сервер, например, для получения веб-страницы или отправки данных формы. Сервер обрабатывает запросы и отправляет обратно HTTP-ответы с запрошенными данными или статусом выполнения запроса.

Основные характеристики HTTP:

1. Клиент-серверная модель: Протокол HTTP работает в клиент-серверной архитектуре, где клиент (например, веб-браузер) отправляет запросы на сервер, а сервер обрабатывает запросы и отправляет обратно ответы с запрошенными данными.
2. Без состояния (stateless): HTTP является протоколом без состояния, что означает, что каждый запрос клиента рассматривается сервером независимо от предыдущих запросов. Сервер не сохраняет информацию о состоянии клиента между запросами.
3. Запросы и ответы: Протокол HTTP определяет структуру запросов и ответов. Запрос состоит из метода (GET, POST, PUT, DELETE и т. д.), URL (Uniform Resource Locator) для указания ресурса, заголовков, содержащих метаданные запроса, и тела (для некоторых типов запросов). Ответ содержит статус ответа (например, 200 OK, 404 Not Found), заголовки с метаданными ответа и тело с данными или ресурсом, запрошенным клиентом.
4. Stateless Cookies: Хотя протокол HTTP без состояния, веб-приложения могут использовать механизмы, такие как cookies, для сохранения информации о состоянии клиента между запросами. Cookies сохраняются на клиентской стороне и передаются в заголовках запросов и ответов.
5. Шифрование: Протокол HTTP может использовать протоколы шифрования, такие как HTTPS (HTTP Secure), чтобы обеспечить безопасную передачу данных между клиентом и сервером. HTTPS использует SSL (Secure Sockets Layer) или его преемника, протокол TLS (Transport Layer Security), для шифрования данных и обеспечения конфиденциальности и целостности.

Применение гипертекста, языка HTML и протокола HTTP позволяет создавать и взаимодействовать с веб-страницами и веб-приложениями. Гипертекст обеспечивает связи между страницами, HTML определяет структуру и содержимое страницы, а HTTP обеспечивает коммуникацию между клиентом и сервером для передачи данных и запросов. Эти технологии являются основными строительными блоками интернета и обеспечивают функционирование веб-сайтов и приложений.

* + 1. Применение Internet технологий. Технология ASP.Net для работы с базами данных. Примеры применения объектов ADO и ADO .Net.

Применение интернет-технологий включает использование технологии ASP.NET для работы с базами данных. ASP.NET - это платформа разработки веб-приложений, которая предоставляет средства и функциональность для создания динамических веб-сайтов и взаимодействия с базами данных.

Одним из основных компонентов ASP.NET для работы с базами данных являются объекты ADO.NET (ActiveX Data Objects for .NET). ADO.NET предоставляет набор классов и компонентов для управления подключением к базам данных, выполнения запросов, извлечения данных и обработки результатов.

Примеры применения объектов ADO.NET включают:

1. Подключение к базе данных: С помощью объекта Connection можно установить соединение с базой данных, указав соответствующие параметры подключения, такие как сервер базы данных, имя пользователя, пароль и т. д.
2. Выполнение запросов: Объекты Command используются для выполнения SQL-запросов к базе данных. Можно создать команду, указав текст запроса и параметры, и затем выполнить команду для извлечения данных или изменения состояния базы данных.
3. Извлечение данных: Объекты DataReader предоставляют возможность последовательного чтения данных из результирующего набора запроса. Можно использовать DataReader для получения данных и их обработки построчно.
4. Работа с наборами данных: Объекты DataSet и DataTable предоставляют возможность хранения и манипулирования данными в памяти. Можно заполнить DataSet или DataTable результатами запроса и затем выполнять операции, такие как фильтрация, сортировка и обновление данных.
5. Управление транзакциями: ADO.NET предоставляет объекты Transaction для управления транзакциями базы данных. Можно начать транзакцию, выполнить несколько операций базы данных и затем зафиксировать или откатить транзакцию.

Применение объектов ADO.NET позволяет разработчикам создавать мощные и эффективные веб-приложения, которые взаимодействуют с базами данных для хранения и обработки данных. Они широко используются в различных сценариях, таких как онлайн-магазины, системы управления контентом, веб-порталы и другие приложения, требующие доступа к данным из базы данных.

* 1. **Базы данных и программирование на SQL / PL SQL**
     1. Защита баз данных Access: файлы рабочей группы, права доступа пользователей.

Защита баз данных Access может быть обеспечена с помощью файлов рабочей группы и установки прав доступа для пользователей. Вот некоторые основные концепции и методы защиты баз данных Access:

1. Файлы рабочей группы: Файл рабочей группы (.mdw) - это файл базы данных Access, который содержит информацию о пользователях, группах пользователей и их правах доступа к базе данных. Файл рабочей группы определяет структуру безопасности базы данных, включая пользователей, пароли, разрешения и другие настройки. Его можно создать с помощью инструмента "Мастер безопасности" в Access.
2. Пользователи и группы: В файле рабочей группы можно создавать пользователей и группы пользователей. Пользователи представляют отдельных пользователей, которым назначаются уникальные идентификаторы и пароли. Группы объединяют пользователей с общими правами доступа. Например, можно создать группу "Администраторы" с полными правами доступа и группу "Пользователи" с ограниченными правами.
3. Права доступа: После создания пользователей и групп можно установить права доступа для объектов базы данных, таких как таблицы, формы, отчеты и запросы. Права доступа могут быть установлены на уровне группы или индивидуального пользователя. Некоторые типичные права доступа включают чтение, запись, выполнение и удаление данных.
4. Аутентификация и авторизация: При попытке входа в базу данных Access пользователю будет предложено предоставить идентификационные данные (имя пользователя и пароль). Файл рабочей группы проверяет эти данные для аутентификации пользователя и определяет его права доступа (авторизация) в соответствии с настройками безопасности.
5. Шифрование паролей: Access предоставляет возможность шифрования паролей пользователей в файле рабочей группы. Это обеспечивает дополнительный уровень защиты от несанкционированного доступа к паролям пользователей.
6. Администрирование безопасности: В файле рабочей группы можно выполнять административные задачи, такие как создание и удаление пользователей, изменение прав доступа и настройка параметров безопасности. Также доступны инструменты для мониторинга и аудита действий пользователей в базе данных.

Важно отметить, что защита базы данных Access через файлы рабочей группы имеет свои ограничения и не является самым современным или мощным способом обеспечения безопасности. Если требуется более сложная система безопасности или использование современных баз данных, рекомендуется рассмотреть альтернативные решения, такие как SQL Server и другие СУБД с расширенными функциями безопасности.

* + 1. Совместное использование баз данных Access: доступ к другим базам данных и таблицам других баз данных.

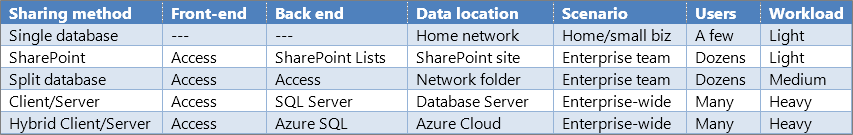
В Access есть несколько способов совместного использования баз данных, включая доступ к другим базам данных и таблицам из других баз данных. Вот некоторые методы, которые можно использовать:

1. Ссылки на таблицы: В Access можно создать ссылки на таблицы из других баз данных. Это позволяет вам получить доступ к данным, которые хранятся в других базах данных, но использовать их в текущей базе данных. Ссылки на таблицы позволяют вам выполнять запросы и взаимодействовать с данными из внешних баз данных так же, как с данными в текущей базе данных.
2. Импорт и экспорт данных: Access позволяет импортировать данные из других баз данных или таблиц в текущую базу данных и экспортировать данные из текущей базы данных в другие форматы или базы данных. Это позволяет обмениваться данными между различными базами данных и использовать их для различных целей.
3. Использование связей: В Access можно создавать связи между таблицами из разных баз данных. Это позволяет вам создавать отношения между данными из разных источников и выполнять запросы, используя данные из связанных таблиц.
4. Ссылки на базы данных: В Access можно создавать ссылки на другие базы данных. Это позволяет вам обращаться к объектам других баз данных, таким как таблицы, формы, запросы и отчеты, и использовать их в текущей базе данных. Ссылки на базы данных обеспечивают более широкий доступ к данным и функциональности, чем простые ссылки на таблицы.
5. Использование SQL: Access поддерживает использование SQL (Structured Query Language) для работы с базами данных. С помощью SQL можно выполнять запросы к данным из разных баз данных и таблиц, объединять данные из разных источников, выполнять агрегатные функции и другие операции.

Эти методы позволяют совместно использовать базы данных Access, получать доступ к данным из разных источников и выполнять операции с данными из разных таблиц и баз данных. Это может быть полезно, когда требуется объединить данные из разных источников или вести работу с данными, которые хранятся в разных базах данных.

*Обзор способов общего доступа к данным Access*

Приложения баз данных изменяются и меняются со временем. Многие факторы, влияющие на потребности и производительность, в том числе количество пользователей одновременно, сетевую среду, пропускную способность, задержку, размер базы данных, пиковое время использования и ожидаемые показатели роста. Если же решение базы данных успешно, скорее всего, оно должно развиваться. К счастью, в Access существует эволюционный путь от простого к более сложным, который может занять некоторое время, чтобы эффективно масштабировать решение. В таблице ниже общались сценарии и рабочие нагрузки Access, которые помогут вам выбрать этот путь.



*Совместное доступ к одной базе данных*

Это самый простой вариант с минимальными требованиями, но он обеспечивает наименьшую функциональность. При этом методе файл базы данных хранится на общем сетевом диске, и все пользователи одновременно его используют. Поскольку все объекты базы данных используются одновременно, несколько пользователей могут одновременно изменять данные, что ограничивает надежность и доступность. Может также снижаться производительность, поскольку все объекты базы данных пересылаются по сети.

Этот вариант подходит в том случае, если базу данных одновременно будут использовать несколько человек и пользователям не потребуется изменять структуру базы данных. Однако этот способ менее безопасен, чем другие способы совместного использования базы данных, так как у каждого пользователя есть полная копия файла базы данных, что увеличивает риск несанкционированного доступа.

Чтобы поделиться базой данных с помощью общей папки:

1. В среде домашнего или малого бизнеса поделитесь папкой с определенными людьми. Дополнительные сведения см. в разделе "Общий доступ к файлам по сети в Windows 10".
2. Приложение Access должно быть настроено для открытия в режиме совместного доступа на компьютерах всех пользователей. Данный режим используется по умолчанию, однако это необходимо проверить:  если пользователь откроет базу данных в монопольном режиме, другие пользователи не смогут работать с данными.
   1. Запустите Access и на вкладке Файл выберите пункт Параметры.
   2. В окне Параметры Access выберите пункт Параметры клиента.
   3. В разделе "Дополнительные настройки" в режимеоткрытия по умолчанию выберите "Общие", нажмите кнопку "ОК"и закроем Access.
3. Скопируйте файл базы данных в общую папку. Затем настройте атрибуты файла таким образом, чтобы разрешить доступ к файлу базы данных для чтения и записи. Для использования базы данных необходим доступ к ней с правами на чтение и запись.
4. На компьютере каждого пользователя создайте ярлык для файла базы данных.

*Совместное использование разделенной базы данных*

Этот способ целесообразен при отсутствии сайта SharePoint или сервера базы данных. Разделенную базу данных можно делиться по локальной сети. При разделении базы данных она реорганизуется в два файла: серверную базу данных, которая содержит таблицы данных, и клиентскую базу данных, в которой содержатся все остальные объекты базы данных (например, запросы, формы, отчеты). Каждый пользователь взаимодействует с данными с помощью локальной копии внешней базы данных.

Преимущества разделения базы данных включают в себя следующие преимущества:

* Улучшенная производительность    В сети общий доступ имеется только к данным: таблицам, запросам, формам, отчетам, макросам и модулям.
* Большая доступность    Транзакции базы данных, такие как изменение записей, завершаются быстрее.
* <c0>Улучшенная безопасность</c0>.    Пользователи могут получать доступ к задней базе данных через связанные таблицы. менее вероятно, что злоумышленники смогут получить несанкционированный доступ к данным через фронтнюю базу данных.
* Улучшенная надежность    Если у пользователя возникла проблема и база данных неожиданно закрывается, чаще всего файл базы данных повреждается только из-за копии интерфейсной базы данных, открытой пользователем.
* Гибкая среда разработки    Каждый пользователь может независимо разрабатывать запросы, формы, отчеты и другие объекты базы данных, не затрагивая других пользователей. Вы также можете разрабатывать и распространять новую версию передней базы данных, не нарушая доступ к данным, хранимым во ее базе данных.

*Совместное передачу данных на сайте SharePoint*

Поделиться данными Access на сайте SharePoint можно несколькими способами.

* Связывание    Процесс связывания подключается к данным в другой программе, так что вы можете просматривать и редактировать последние данные как в SharePoint, так и в Access, не создавая и не сохраняя их копию в Access. Если вы не хотите копировать список SharePoint в базу данных Access, а хотите создавать запросы и создавать отчеты на основе содержимого этого списка, вы можете создать связь с данными.
* Перемещение    При этом на сайте SharePoint создаются списки, связанные с таблицами базы данных. Мастер экспорта таблиц в SharePoint позволяет одновременно перемещать данные из всех таблиц и поддерживать их отношения.

Предупреждение    Несмотря на то что файл базы данных Access можно сохранить в OneDrive или библиотеке документов SharePoint, рекомендуется не открывать базу данных Access из этих мест. Файл можно скачать локально для редактирования, а затем снова отправить, как только вы сохраните изменения в SharePoint. Если базу данных Access из SharePoint открывает несколько человек, может быть создано несколько копий базы данных и могут возникать непредвиденные действия. Эта рекомендация относится ко всем типам файлов Access, включая одну базу данных, разделенную базу данных, а также форматы файлов ACCDB, ACCDC, ACCDE и ACCDR.

*Совместное использование данных с помощью сервера базы данных*

Совместное использование базы данных можно организовать с помощью приложения Access и сервера баз данных (например, сервера SQL Server). Этот способ обеспечивает много преимуществ, но для него требуется дополнительное программное обеспечение — сервер баз данных.

Этот способ напоминает разделение баз данных, поскольку таблицы хранятся в сети, а у каждого пользователя есть локальная копия файла базы данных Microsoft Access, содержащая ссылки на таблицы, запросы, формы, отчеты и другие объекты базы данных. Преимущества этого метода зависят от используемого программного обеспечения сервера баз данных, но в общем случае они включают наличие учетных записей пользователей и избирательный доступ к данным, отличную доступность данных и удобные встроенные средства управления данными. Более того, большинство серверных приложений для работы с базами данных нормально работают с более ранними версиями Access, поэтому не требуется, чтобы все пользователи работали с одной и той же версией. Совместно используются только таблицы. Дополнительные сведения см. в том, как перенести базу данных Access в [SQL Server,](https://support.microsoft.com/ru-ru/office/%D0%BC%D0%B8%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F-%D0%B1%D0%B0%D0%B7%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-access-%D0%BD%D0%B0-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80-sql-server-7bac0438-498a-4f53-b17b-cc22fc42c979)импортировать данные в [базе](https://support.microsoft.com/ru-ru/office/%D0%B8%D0%BC%D0%BF%D0%BE%D1%80%D1%82-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-%D0%B8%D0%B7-%D0%B1%D0%B0%D0%B7%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-sql-server-%D0%B8%D0%BB%D0%B8-%D1%81%D0%B2%D1%8F%D0%B7%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%81-%D0%BD%D0%B8%D0%BC%D0%B8-a5a3b4eb-57b9-45a0-b732-77bc6089b84e)данных SQL Server или связать их с данными, а также связать или импортировать данные из базы данных [Azure SQL Server.](https://support.microsoft.com/ru-ru/office/%D1%81%D0%B2%D1%8F%D0%B7%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%B8%D0%BB%D0%B8-%D0%B8%D0%BC%D0%BF%D0%BE%D1%80%D1%82-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-%D0%B8%D0%B7-%D0%B1%D0%B0%D0%B7%D1%8B-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-azure-sql-server-88c0cc2c-21dd-46f5-b74a-0db3613f5166)

Преимущества совместного использования базы данных с помощью сервера баз данных

* Высокая производительность и масштабируемость    Во многих случаях сервер базы данных обеспечивает лучшую производительность, чем только файл базы данных Access. Многие серверные продукты баз данных также поддерживают очень большие базы данных размером 1 ТБ, примерно в 500 раз больше текущего предела для файла базы данных Access (два гигабайта). Серверы баз данных обычно работают очень эффективно, обрабатывая запросы параллельно (используя несколько нитей в одном процессе для обработки запросов пользователей) и свести к минимуму дополнительные требования к памяти при добавлении новых пользователей.
* Повышенная доступность    В большинстве серверов баз данных можно сделать ее базой данных во время ее использования. Поэтому вам не нужно принудительно выходить из базы данных для хранения данных. Кроме того, серверы баз данных обычно очень эффективно обрабатывают одновременное редактирование и блокировку записей.
* Улучшенная безопасность    Ни одна база данных не может быть полностью защищена. Однако серверы баз данных обеспечивают надежную защиту, которая поможет защитить ваши данные от несанкционированного использования. В большинстве серверов баз данных предлагаются средства безопасности на основе учетных записей, позволяющие указать, кто может видеть таблицы. Даже в случае неправильного получения доступа к переднему нему access несанкционированное использование данных будет предотвращено средствами безопасности, основанными на учетной записи.
* Автоматическое восстановление    В случае сбоя системы (например, сбоя операционной системы или простоя питания) некоторые серверы баз данных имеют механизмы автоматического восстановления базы данных до последнего согласованного состояния в течение нескольких минут без вмешательства администратора базы данных.
* Серверная обработка    Использование Access в конфигурации клиента или сервера помогает сократить сетевой трафик за счет обработки запросов к базе данных на сервере перед отправкой результатов клиенту. Обычно сервер обрабатывает данные эффективнее, особенно при работе с большими наборами данных.
* Azure SQL Server    В дополнение к преимуществам SQL Server динамической масштабируемости без простоев, интеллектуальной оптимизации, глобальной масштабируемости и доступности, отказом от затрат на оборудование и сокращением администрирования.
  + 1. Базы данных Access: связи с внешними данными (импорт, экспорт, присоединение, связи с MS Office).

В базах данных Access есть несколько способов устанавливать связи с внешними данными, включая импорт, экспорт, присоединение и связь с продуктами MS Office. Ниже представлены примеры этих методов:

1. Импорт данных: Access позволяет импортировать данные из внешних источников в базу данных. Это может быть выполнено путем импорта таблиц, файлов Excel, CSV-файлов или данных из других баз данных, таких как SQL Server или Oracle. Импортированные данные становятся доступными в Access для дальнейшего использования и анализа.
2. Экспорт данных: Вы также можете экспортировать данные из базы данных Access во внешние источники. Это может быть выполнено путем экспорта таблиц, запросов или результатов запросов в файлы Excel, CSV-файлы, текстовые файлы или другие базы данных. Экспорт позволяет передавать данные из Access в другие приложения или использовать их в других системах.
3. Присоединение к внешним таблицам: В Access можно установить присоединение к внешним таблицам, которые хранятся в других базах данных или источниках данных. Например, вы можете создать присоединение к таблицам SQL Server или Oracle, и эти таблицы будут доступны в базе данных Access для запросов, отчетов и других операций. Присоединенные таблицы можно использовать так же, как и локальные таблицы в Access.
4. Связь с MS Office: Access обладает возможностью взаимодействовать с другими продуктами MS Office, такими как Excel, Word и PowerPoint. Вы можете создавать связи с данными в этих продуктах и использовать их в базе данных Access. Например, вы можете импортировать данные из Excel в Access, связать данные между Excel и Access, или экспортировать данные из Access в Word для создания отчетов или документации.

Эти методы позволяют устанавливать связи с внешними данными в базах данных Access, импортировать и экспортировать данные, присоединяться к внешним таблицам и использовать данные из других продуктов MS Office. Это позволяет вам интегрировать данные из различных источников и использовать их в своих приложениях Access.

* + 1. Репликация баз данных.

Репликация баз данных - это процесс создания и поддержания копий данных из одной базы данных в другой. Целью репликации является обеспечение доступности данных и повышение производительности системы.

В процессе репликации одна база данных называется исходной (мастер-базой данных), а другая - целевой (репликой). Исходная база данных содержит основные данные, которые необходимо реплицировать, а целевая база данных получает копии этих данных и поддерживает их в актуальном состоянии.

Репликация может быть настроена для различных целей. Некоторые из основных преимуществ репликации баз данных включают:

1. Высокая доступность: Если исходная база данных становится недоступной из-за сбоя или обслуживания, целевая база данных может продолжать обслуживание запросов, так как она содержит актуальные данные.
2. Локальный доступ к данным: Репликация позволяет иметь локальные копии данных в разных местоположениях или на разных серверах. Это может быть полезно для распределенных систем или ситуаций, когда требуется быстрый доступ к данным без задержек сети.
3. Увеличение производительности: Распределение нагрузки на несколько реплик может повысить производительность системы, поскольку каждая реплика может обрабатывать запросы независимо. Это особенно полезно при выполнении запросов на чтение, так как они могут быть распределены между несколькими репликами.
4. Обеспечение отказоустойчивости: Если исходная база данных сталкивается с сбоем или потерей данных, реплика может использоваться для восстановления системы и восстановления данных.

Существуют различные подходы к репликации баз данных, включая мастер-мастер репликацию, мастер-реплика репликацию и каскадную репликацию. Каждый подход имеет свои особенности и может быть выбран в зависимости от требований системы.

Важно отметить, что репликация баз данных не является полным резервным копированием данных. В случае ошибок или удаления данных в исходной базе данных, эти изменения будут распространяться на все реплики. Поэтому для обеспечения полной защиты данных рекомендуется использовать резервное копирование в сочетании с репликацией баз данных.

* + 1. Возможности использования полей со списком (для обеспечения целостности данных, для эффективного доступа к данным) в приложениях баз данных.

Использование полей со списком в приложениях баз данных предоставляет несколько возможностей для обеспечения целостности данных и эффективного доступа к данным. Ниже приведены некоторые из них:

1. Хранение связанных данных: Поля со списком позволяют хранить связанные данные в структурированном формате. Например, вместо создания отдельной таблицы для связанных элементов, можно использовать поле со списком для хранения списка идентификаторов связанных записей. Это упрощает и ускоряет доступ к связанным данным.
2. Управление множественными значениями: Поля со списком могут использоваться для хранения множественных значений, таких как теги, категории или список участников. Это обеспечивает гибкость и удобство при работе с данными, позволяя добавлять, удалять и искать значения в списке.
3. Обеспечение целостности данных: Поля со списком могут использоваться для обеспечения целостности данных, например, путем определения ограничений на список допустимых значений или проверки уникальности элементов в списке. Это помогает предотвратить ошибки и некорректные данные.
4. Запросы и фильтрация данных: Поля со списком обеспечивают эффективный доступ к данным путем использования запросов и фильтрации на основе элементов в списке. Например, можно легко выполнить запрос на поиск всех записей, содержащих определенное значение в поле со списком.
5. Агрегация данных: Поля со списком также могут использоваться для агрегации данных. Например, можно использовать поле со списком для хранения списка товаров в заказе, и затем легко вычислить общую стоимость заказа или количество товаров.
6. Использование индексов: Если поле со списком часто используется для поиска или фильтрации данных, можно создать индекс на это поле. Это повышает производительность запросов, ускоряя доступ к данным и сокращая время выполнения запросов.

Однако стоит учитывать, что использование полей со списком имеет свои ограничения. Например, сложные запросы или операции обновления данных могут быть более затруднительными с использованием полей со списком. Также может возникнуть проблема масштабируемости, если список значений в поле становится очень большим.

При проектировании базы данных и выборе использования полей со списком следует тщательно оценить требования приложения и обратить внимание на потенциальные преимущества и ограничения такого подхода.

* + 1. Управление базами данных сервера Microsoft SQL Server. Физическая архитектура баз данных.

Управление базами данных в сервере Microsoft SQL Server включает в себя административные задачи по созданию, настройке, мониторингу, обслуживанию и оптимизации баз данных. Вот некоторые основные аспекты управления базами данных в SQL Server:

1. Создание баз данных: SQL Server предоставляет возможность создания новых баз данных с помощью SQL Management Studio (SSMS) или с использованием скриптов T-SQL. При создании базы данных можно указать различные параметры, такие как размер, файлы данных и журнала, параметры роста и настройки безопасности.
2. Настройка баз данных: После создания базы данных можно настроить различные аспекты, такие как параметры совместимости, уровень изоляции транзакций, параметры резервного копирования и восстановления, параметры авто-создания статистики и другие.
3. Мониторинг баз данных: SQL Server предоставляет инструменты для мониторинга состояния и производительности баз данных. Это включает в себя мониторинг использования ресурсов, обнаружение блокировок и конфликтов, анализ выполнения запросов, мониторинг журналов транзакций и т.д. Для этого можно использовать SSMS, SQL Server Profiler, SQL Server Management Data Warehouse и другие инструменты.
4. Обслуживание баз данных: Обслуживание баз данных включает выполнение регулярных задач, таких как резервное копирование и восстановление баз данных, проверка целостности данных, реорганизация индексов, обновление статистики и т.д. SQL Server предоставляет планировщик задач (SQL Server Agent) для автоматизации этих задач.
5. Оптимизация баз данных: Оптимизация баз данных включает анализ и улучшение производительности запросов и структуры базы данных. Для этого можно использовать инструменты, такие как SQL Server Query Store, Database Engine Tuning Advisor, индексы, представления, хранимые процедуры и другие оптимизационные техники.

Теперь касательно физической архитектуры баз данных в SQL Server. В SQL Server физическая архитектура базы данных состоит из нескольких основных компонентов:

1. Файлы данных (Data Files): Файлы данных представляют собой физические файлы, в которых хранятся данные таблиц, индексов и других объектов базы данных. Они имеют расширение .mdf и содержат данные, индексы и структуру таблиц.
2. Журнал транзакций (Transaction Log): Журнал транзакций представляет собой файл, который записывает все изменения данных, производимые в базе данных. Он имеет расширение .ldf и используется для обеспечения целостности транзакций, восстановления базы данных и репликации.
3. Файлы резервной копии (Backup Files): Файлы резервной копии содержат полные, дифференциальные или инкрементальные копии базы данных. Они используются для восстановления данных в случае сбоев или потери данных.
4. Файлы журнала резервной копии (Backup Log Files): Файлы журнала резервной копии содержат информацию о резервных копиях, выполняемых в базе данных. Они используются для управления и восстановления резервных копий.

Кроме того, SQL Server использует различные системные таблицы и представления для хранения метаданных о базах данных, таблицах, индексах, представлениях, хранимых процедурах и других объектах.

Физическая архитектура баз данных в SQL Server может быть дополнительно настроена с помощью параметров конфигурации, таких как размер страницы, настройки хранения данных и другие параметры, которые влияют на производительность и оптимизацию базы данных.

* + 1. Управление учетными записями сервера Microsoft SQL Server. Серверные роли.

Управление учетными записями в сервере Microsoft SQL Server включает создание, настройку и управление учетными записями пользователей и ролей, которые имеют доступ к базам данных и выполняют различные операции. Вот некоторые основные аспекты управления учетными записями в SQL Server:

1. Создание учетных записей пользователей: Чтобы пользователь мог получить доступ к базам данных, необходимо создать учетную запись пользователя в SQL Server. Это можно сделать с помощью SQL Management Studio (SSMS) или с использованием скриптов T-SQL. При создании учетной записи можно указать имя пользователя, пароль и другие параметры безопасности.
2. Назначение ролей пользователю: Роли в SQL Server представляют собой набор разрешений, определяющих права доступа к базам данных и объектам в них. Существуют предопределенные роли, такие как sysadmin, db\_owner, db\_datareader и другие. Пользователю можно назначить одну или несколько ролей для определения его прав доступа.
3. Создание пользовательских ролей: Помимо предопределенных ролей, можно создавать пользовательские роли в SQL Server. Пользовательская роль позволяет определить собственные наборы разрешений и назначать их пользователям. Создание пользовательских ролей обеспечивает более гибкое управление доступом к данным.
4. Назначение разрешений на объекты: Разрешения в SQL Server определяют, какие операции разрешены или запрещены для конкретных пользователей или ролей на объекты базы данных, такие как таблицы, представления, процедуры и другие. Можно назначать разрешения непосредственно на объекты или использовать наследование разрешений через роли.
5. Аудит доступа и мониторинг: SQL Server предоставляет инструменты для аудита доступа и мониторинга действий пользователей в базах данных. Можно настроить аудит операций, чтобы записывать информацию о запросах, изменениях данных и других действиях пользователей. Также можно использовать мониторинговые инструменты для отслеживания активности и обнаружения нарушений безопасности.
6. Управление паролями и политиками безопасности: SQL Server позволяет устанавливать сложные пароли для учетных записей пользователей и определять политики безопасности для паролей, такие как требование длины, использование специальных символов, периодическое изменение пароля и т.д. Это помогает обеспечить безопасность доступа к данным.

Серверные роли в SQL Server представляют собой набор предопределенных ролей, которые имеют определенные привилегии и разрешения на уровне всего сервера. Вот некоторые из наиболее распространенных серверных ролей:

1. sysadmin: Эта роль является наивысшей и обладает полными административными привилегиями на сервере SQL. У пользователей с ролью sysadmin есть полный контроль над всеми базами данных и объектами на сервере.
2. serveradmin: Роль serveradmin обладает привилегиями на уровне сервера, такими как управление конфигурацией сервера, запуск и остановка служб SQL Server, создание и изменение планировщиков задач и другие административные задачи.
3. securityadmin: Роль securityadmin отвечает за управление безопасностью сервера и баз данных. Пользователи с этой ролью могут создавать и управлять учетными записями пользователей, ролями безопасности, сертификатами, ключами и другими аспектами безопасности.
4. processadmin: Роль processadmin позволяет пользователям управлять процессами SQL Server, включая управление выполнением запросов, отмену операций и управление ресурсами.
5. setupadmin: Роль setupadmin дает права на выполнение задач, связанных с установкой и настройкой SQL Server, таких как создание и изменение экземпляров SQL Server, установка функциональности и параметров сервера.
6. bulkadmin: Роль bulkadmin предоставляет возможность пользователю выполнять операции загрузки массовых данных (Bulk operations), таких как BULK INSERT и OPENROWSET BULK.

Управление серверными ролями позволяет администраторам наделить пользователей соответствующими привилегиями и разрешениями на уровне всего сервера, обеспечивая эффективное управление безопасностью и доступом к данным.

* + 1. Управление учетными записями пользователя базы данных Microsoft SQL Server. Пользовательские роли. Схемы данных.

Управление учетными записями пользователей базы данных в Microsoft SQL Server включает создание, настройку и управление учетными записями, которые имеют доступ к конкретным базам данных и выполняют операции с их объектами. Вот некоторые основные аспекты управления учетными записями пользователей в SQL Server:

1. Создание учетных записей пользователей базы данных: Чтобы пользователь мог получить доступ к конкретной базе данных, необходимо создать учетную запись пользователя в этой базе данных. Создание учетной записи можно выполнить с помощью SQL Management Studio (SSMS) или с использованием скриптов T-SQL. При создании учетной записи можно указать имя пользователя, пароль и другие параметры безопасности.
2. Назначение ролей пользователю базы данных: Роли пользователей в SQL Server базе данных представляют собой набор разрешений и прав доступа к объектам внутри базы данных. SQL Server предоставляет предопределенные роли, такие как db\_owner, db\_datareader, db\_datawriter и другие. Пользователю можно назначить одну или несколько ролей для определения его прав доступа.
3. Создание пользовательских ролей: Помимо предопределенных ролей, можно создавать пользовательские роли в SQL Server базе данных. Пользовательская роль позволяет определить собственные наборы разрешений и назначать их пользователям. Создание пользовательских ролей обеспечивает более гибкое управление доступом к данным внутри базы данных.
4. Назначение разрешений на объекты: Разрешения в SQL Server базе данных определяют, какие операции разрешены или запрещены для конкретных пользователей или ролей на объекты базы данных, такие как таблицы, представления, хранимые процедуры и другие. Можно назначать разрешения непосредственно на объекты или использовать наследование разрешений через роли.
5. Схемы данных (Data Schemas): Схемы данных в SQL Server базе данных представляют собой логические контейнеры для группировки и организации объектов базы данных, таких как таблицы, представления, функции и другие. Схемы позволяют логически разделять объекты базы данных и управлять их доступом. Пользователям можно назначать разрешения на схемы данных, что дает им доступ только к определенным объектам.

Управление учетными записями пользователей базы данных позволяет администраторам гибко управлять доступом к данным и ограничивать права пользователей на конкретные объекты базы данных. Использование пользовательских ролей и схем данных помогает организовать структуру доступа и обеспечить безопасность данных в базе данных Microsoft SQL Server.

* + 1. Назначение разрешений доступа сервера Microsoft SQL Server. Уровни разрешений доступа.

В Microsoft SQL Server разрешения доступа определяют, какие операции и действия могут выполнять пользователи и роли на уровне сервера. Назначение разрешений доступа в SQL Server осуществляется на разных уровнях, включая уровень сервера, базы данных и объектов базы данных. Вот некоторые основные уровни разрешений доступа в SQL Server:

1. Уровень сервера (Server-Level): На уровне сервера разрешения определяются для контроля операций и действий, которые могут выполняться на самом сервере SQL. Некоторые примеры разрешений на уровне сервера включают возможность создания баз данных, управление безопасностью, запуск и остановку служб SQL Server и другие административные операции. Разрешения на уровне сервера могут быть назначены пользователям, ролям и встроенным ролям, таким как sysadmin.
2. Уровень базы данных (Database-Level): На уровне базы данных разрешения определяются для контроля операций и действий внутри конкретной базы данных. Некоторые примеры разрешений на уровне базы данных включают возможность создания таблиц, выполнения запросов, изменения данных, управления схемами данных и другие операции. Разрешения на уровне базы данных могут быть назначены пользователям, ролям и встроенным ролям, таким как db\_owner, db\_datareader и db\_datawriter.
3. Уровень объектов базы данных (Object-Level): На уровне объектов базы данных, таких как таблицы, представления, процедуры и функции, разрешения определяются для контроля операций и действий над конкретными объектами. Некоторые примеры разрешений на уровне объектов включают возможность чтения, записи, изменения схемы, выполнения процедур и другие операции, связанные с конкретным объектом базы данных. Разрешения на уровне объектов могут быть назначены пользователям, ролям и встроенным ролям.

SQL Server предоставляет широкий набор встроенных разрешений, которые можно назначать на разных уровнях для достижения требуемого уровня доступа и безопасности данных. Администратор базы данных должен тщательно управлять разрешениями, чтобы обеспечить только необходимые права доступа для пользователей и ролей и минимизировать риски безопасности.

* + 1. Программирование в SQL Server. Управление потоком команд. Использование переменных. Хранимые процедуры. Управление видами.

Программирование в SQL Server позволяет создавать более сложные и мощные запросы, управлять потоком команд, использовать переменные, создавать хранимые процедуры и управлять видами данных. Вот некоторые основные аспекты программирования в SQL Server:

1. Управление потоком команд: В SQL Server можно использовать различные конструкции для управления потоком выполнения команд. Например, операторы условного выполнения, такие как IF...ELSE и CASE, позволяют выполнять команды в зависимости от определенных условий. Операторы цикла, такие как WHILE и CURSOR, позволяют выполнять команды многократно до выполнения определенного условия.
2. Использование переменных: В SQL Server можно использовать переменные для хранения временных значений и передачи данных между различными частями запроса. Переменные в SQL Server объявляются с помощью ключевого слова DECLARE и могут хранить значения различных типов данных, таких как целые числа, строки, даты и другие. Переменные могут использоваться в выражениях, условиях, циклах и других конструкциях запросов.
3. Хранимые процедуры: Хранимая процедура в SQL Server представляет собой предопределенный набор команд, который сохраняется в базе данных и может быть вызван повторно. Хранимые процедуры позволяют группировать и структурировать логику выполнения запросов, упрощать обслуживание и повторное использование кода. Они могут принимать параметры, включать операторы условного выполнения и циклы, использовать переменные и возвращать результаты выполнения.
4. Управление видами (Views): Вид в SQL Server представляет собой виртуальную таблицу, которая основывается на данных из одной или нескольких таблиц. Вид позволяет абстрагировать сложные запросы и предоставлять удобный интерфейс для доступа к данным. Виды можно использовать для упрощения запросов, скрытия сложной логики или ограничения доступа к определенным столбцам или строкам данных.

Программирование в SQL Server предоставляет множество возможностей для более гибкой обработки данных и управления запросами. Оно позволяет создавать сложные логические конструкции, работать с переменными, использовать хранимые процедуры и управлять видами данных для более эффективной работы с базой данных.

* + 1. Инфологическое моделирование баз данных.

Инфологическое моделирование баз данных - это процесс создания абстрактной модели данных, которая описывает основные сущности, атрибуты и связи между ними в информационной системе. Эта модель является независимой от конкретной технологии базы данных и фокусируется на логической структуре данных, а не на физической реализации.

Основная цель инфологического моделирования баз данных состоит в том, чтобы создать ясное и точное представление о данных, которые будут храниться и использоваться в системе. Это позволяет разработчикам и аналитикам лучше понимать предметную область и требования к базе данных, а также облегчает коммуникацию между членами команды разработки.

Основные инструменты инфологического моделирования баз данных включают в себя:

1. Сущность-связьная модель (Entity-Relationship Model, ER-модель): Это графический подход к моделированию данных, который использует сущности (объекты предметной области), атрибуты (характеристики сущностей) и связи (отношения между сущностями) для описания данных.
2. Диаграммы классов (Class Diagrams): Это модель данных, основанная на объектно-ориентированном подходе, где классы представляют сущности, атрибуты и методы классов описывают их характеристики и поведение. Диаграммы классов часто используются при разработке баз данных для объектно-ориентированных приложений.
3. Нормализация данных: Это процесс анализа и преобразования структуры данных с целью устранения избыточности и зависимостей, а также повышения эффективности хранения и обработки данных.
4. Диаграммы отношений (Relationship Diagrams): Это дополнительные графические инструменты, которые используются для визуализации связей между таблицами или сущностями в базе данных.

Инфологическое моделирование баз данных является важным этапом в жизненном цикле разработки баз данных и служит основой для физического моделирования, которое определяет способ физической реализации базы данных на конкретной платформе или системе управления базами данных (СУБД).

* + 1. Проектирование реляционных баз данных с использованием принципов нормализации.

Проектирование реляционных баз данных с использованием принципов нормализации - это процесс разработки структуры базы данных, в котором применяются принципы нормализации для устранения избыточности и зависимостей между данными. Нормализация помогает создать эффективную и гибкую базу данных, минимизируя дублирование информации и обеспечивая логическую целостность данных.

Принципы нормализации основаны на наборе нормальных форм, которые определяют требования к структуре данных. Вот основные нормальные формы, используемые в проектировании реляционных баз данных:

1. Первая нормальная форма (1НФ): Здесь данные организованы в отдельные таблицы, и каждый атрибут содержит только одно значение. Это устраняет повторяющиеся группы и структурирует данные.
2. Вторая нормальная форма (2НФ): Для достижения 2НФ необходимо, чтобы каждый неключевой атрибут зависел только от полного первичного ключа, а не от его частей. Это устраняет проблемы с частичными зависимостями.
3. Третья нормальная форма (3НФ): Здесь каждый неключевой атрибут должен зависеть только от первичного ключа и не должен иметь транзитивных зависимостей от других атрибутов. Это помогает устранить транзитивные зависимости и сократить избыточность данных.
4. Четвертая нормальная форма (4НФ): В 4НФ запрещено иметь многозначные зависимости между неключевыми атрибутами. Это позволяет устранить многозначные зависимости и предотвратить избыточность данных.
5. Пятая нормальная форма (5НФ): Здесь устраняются зависимости между многозначными фактами в отношениях путем создания отдельных таблиц для каждого многозначного факта. Это обеспечивает разделение связанных данных и предотвращает дублирование информации.

Проектирование реляционных баз данных с использованием принципов нормализации помогает создать структурированную и эффективную базу данных, способную обеспечивать логическую целостность и удовлетворять требованиям приложения. Однако следует помнить, что слишком высокая нормализация может привести к сложности запросов и производительности, поэтому важно найти баланс между нормализацией и денормализацией в зависимости от конкретных потребностей системы.

* + 1. Этапы жизненного цикла разработки пакетов прикладных программ. Типы жизненного цикла.

Жизненный цикл разработки пакетов прикладных программ (Application Lifecycle) включает несколько этапов, которые охватывают процесс создания, развертывания, поддержки и улучшения программного обеспечения. Этапы могут немного различаться в различных методологиях разработки и организациях, но общие этапы включают:

1. Инициация (Initiation): На этом этапе определяются цели и требования к пакету прикладных программ (ППП). Производится предварительная оценка, планирование, анализ бизнес-процессов и определение основных параметров проекта.
2. Анализ (Analysis): В этой фазе проводится более детальное исследование требований пользователей и бизнес-процессов, а также анализ возможных решений. Создается функциональная спецификация, описывающая функции, данные, интерфейсы и другие характеристики будущего ППП.
3. Проектирование (Design): На этом этапе разрабатывается архитектура и детальный дизайн ППП. Определяются структура базы данных, интерфейсы пользователя, логика приложения, а также проводится оценка технической реализуемости.
4. Реализация (Implementation): В этой фазе разработчики создают и кодируют ППП на основе разработанных спецификаций и дизайна. Осуществляется программирование, тестирование и отладка кода.
5. Тестирование (Testing): На этом этапе производится проверка и проверка работоспособности ППП. Включает модульное тестирование, интеграционное тестирование, системное тестирование и приемочное тестирование для уверенности в правильности функционирования ППП.
6. Развертывание (Deployment): На этом этапе ППП развертывается в реальной среде и готовится к использованию. Это может включать установку на серверы, конфигурацию, импорт данных и другие операции, необходимые для запуска ППП в рабочей среде.
7. Эксплуатация (Operation): В этом этапе ППП используется в производственной среде. Он поддерживается, обновляется, мониторится и управляется для обеспечения его надлежащей работы и соответствия требованиям пользователей.
8. Поддержка и улучшение (Maintenance and Enhancement): В течение жизненного цикла ППП могут потребоваться изменения, исправления ошибок или добавление новых функций. Этот этап включает поддержку и улучшение ППП на основе обратной связи от пользователей и изменяющихся требований.

Типы жизненного цикла программного обеспечения могут быть различными в зависимости от методологии разработки и специфических требований проекта. Некоторые распространенные типы включают:

1. Каскадная модель (Waterfall Model): Линейный подход, где каждый этап выполняется последовательно и переход к следующему этапу происходит только после завершения предыдущего.
2. Итеративные модели (Iterative Models): Включают такие методологии, как Agile (гибкий подход), Scrum и Spiral. Они позволяют более гибко вносить изменения и улучшения в процессе разработки, разбивая его на итерации.
3. Прототипирование (Prototyping): Включает создание прототипов ППП для получения обратной связи от пользователей и уточнения требований перед полноценной разработкой.
4. Развитие по компонентам (Component-Based Development): Разработка программного обеспечения путем создания и интеграции готовых компонентов или модулей.
5. DevOps-цикл (DevOps Cycle): Ориентирован на непрерывную поставку и обеспечение качества программного обеспечения с использованием автоматизации и совместной работы команд разработчиков и операционных специалистов.

Конкретный выбор типа жизненного цикла зависит от требований проекта, временных ограничений, команды разработки и других факторов.

* + 1. Архитектура СУБД ORACLE. Компоненты экземпляра - системные процессы (DBWR, LGWR, SMON, PMON, CKPT).

Архитектура СУБД Oracle имеет клиент-серверную структуру, где клиентские приложения обращаются к серверу Oracle для доступа к данным. Сервер Oracle состоит из экземпляра и базы данных. Вот основные компоненты экземпляра Oracle:

Системный процесс управления базами данных (Database Writer, DBWR): Отвечает за запись данных из кэша БД (буферного кэша) на диски. DBWR периодически выполняет операцию записи, чтобы обновить изменения, хранящиеся в кэше, на дисках.

1. Системный процесс записи журнала (Log Writer, LGWR): Записывает изменения данных в журнале операций (redo log) на дисках. LGWR обеспечивает долговременное хранение операций, чтобы обеспечить целостность данных и возможность восстановления после сбоев.
2. Системный процесс мониторинга экземпляра (System Monitor, SMON): Отвечает за выполнение различных задач, связанных с управлением базой данных. SMON выполняет операции восстановления после сбоев, освобождает ресурсы после отката транзакций, управляет сегментами пространства таблиц и другими административными задачами.
3. Системный процесс мониторинга процессов (Process Monitor, PMON): Отвечает за мониторинг клиентских сеансов и управление подключениями к базе данных. PMON обнаруживает сбои клиентских сеансов и освобождает связанные с ними ресурсы, чтобы предотвратить блокировку или другие проблемы.
4. Системный процесс контрольной точки (Checkpoint Process, CKPT): Запускает операции контрольной точки, которые записывают информацию о состоянии базы данных в файлы контрольной точки (checkpoint files). CKPT помогает согласовать данные на диске и в кэше, что важно для обеспечения целостности и восстановления базы данных.

Кроме перечисленных системных процессов, в архитектуре Oracle также присутствуют другие компоненты, такие как сервер позволяющий подключение клиентов (Listener), фоновые процессы для управления памятью (Memory Manager), выполнения запросов (Query Coordinator) и другие.

Компоненты экземпляра Oracle работают вместе для обеспечения надежной и эффективной работы базы данных, контроля доступа, обработки запросов и обеспечения целостности данных.

* + 1. Архитектура СУБД ORACLE. Компоненты экземпляра – структуры памяти.

Архитектура СУБД Oracle включает различные компоненты памяти, которые играют важную роль в обработке данных и выполнении запросов. Вот основные структуры памяти в экземпляре Oracle:

1. Буферный кэш (Buffer Cache): Это область памяти, в которой хранятся скопированные блоки данных из файлов данных. Буферный кэш улучшает производительность, позволяя избежать повторного чтения данных с диска при повторных запросах.
2. Кэш результатов SQL (SQL Result Cache): Это механизм кэширования результатов выполнения SQL-запросов. Когда SQL-запрос выполняется, его результаты сохраняются в кэше, чтобы в следующий раз запрос с теми же параметрами мог быть выполнен из кэша, без фактического выполнения запроса.
3. Кэш разделяемых пулов (Shared Pool): Это область памяти, в которой хранятся общие структуры данных и объекты для всех пользователей. Кэш разделяемых пулов содержит различные компоненты, такие как планы выполнения запросов, результаты предыдущих запросов, словари данных и другие общие ресурсы.
4. Кэш библиотек (Library Cache): Это структура памяти, в которой хранятся предварительно скомпилированные SQL-инструкции и другие программные объекты, такие как процедуры, функции и пакеты. Кэш библиотек ускоряет выполнение SQL-запросов, так как повторное использование предварительно скомпилированных объектов избавляет от необходимости повторной компиляции.
5. Кэш журналов операций (Redo Log Cache): Это область памяти, в которой хранятся изменения данных до их записи на диски. Кэш журналов операций позволяет быстро записывать операции изменения данных и обеспечивает надежность и восстановление базы данных в случае сбоев.
6. Кэш PGA (Program Global Area): Это область памяти, выделенная для каждой сессии или процесса в Oracle. Кэш PGA содержит данные и структуры для выполнения запросов и обработки данных в рамках конкретной сессии или процесса.

Структуры памяти в экземпляре Oracle служат для ускорения доступа к данным, оптимизации выполнения запросов и обеспечения эффективного использования ресурсов системы. Конфигурация и размер этих структур памяти могут быть настроены в зависимости от требований и ресурсов базы данных.

* + 1. Архитектура СУБД ORACLE. Файловая структура БД.

Архитектура СУБД Oracle включает различные типы файлов, которые используются для хранения данных и других объектов базы данных. Вот основные файлы, которые составляют файловую структуру базы данных Oracle:

1. Файлы данных (Data Files): Файлы данных используются для хранения фактических данных, таких как таблицы, индексы, представления и другие объекты базы данных. Каждый файл данных имеет свое имя и путь в файловой системе. Файлы данных имеют расширение ".dbf".
2. Файлы журналов операций (Redo Log Files): Файлы журналов операций используются для записи изменений данных в базе данных. Они содержат информацию, необходимую для восстановления базы данных после сбоев. Файлы журналов операций имеют расширение ".log".
3. Файлы контрольной точки (Checkpoint Files): Файлы контрольной точки содержат информацию о состоянии базы данных в определенный момент времени. Они помогают восстановить базу данных после сбоев и поддерживают целостность данных. Файлы контрольной точки имеют расширение ".ckpt".
4. Файлы параметров (Parameter Files): Файлы параметров содержат конфигурационные параметры для экземпляра базы данных Oracle. Они определяют настройки, такие как размеры памяти, пути к файлам данных и другие параметры системы. Файлы параметров имеют имена "spfile<номер>.ora" или "init<номер>.ora".
5. Файлы сохраненных точек (Savepoint Files): Файлы сохраненных точек используются для сохранения промежуточных результатов транзакции. Они позволяют откатывать транзакции к определенной точке сохранения, если это необходимо. Файлы сохраненных точек имеют расширение ".svf".

Каждая база данных Oracle может иметь несколько файлов данных, файлов журналов операций и других файлов, которые составляют ее файловую структуру. Эти файлы располагаются на файловой системе сервера базы данных и управляются Oracle для обеспечения доступа к данным и обработки операций.

* + 1. Архитектура СУБД ORACLE. Файловая структура клиент-серверной технологии.

Архитектура СУБД Oracle в клиент-серверной технологии включает следующие основные компоненты:

1. Клиенты (Clients): Клиенты представляют собой приложения или системы, которые обращаются к серверу Oracle для доступа к базе данных. Клиенты могут быть разработаны на различных платформах, таких как Windows, Linux, macOS, и могут использовать различные языки программирования, такие как Java, C++, .NET и другие.
2. Сервер Oracle (Oracle Server): Сервер Oracle представляет собой центральный компонент, который управляет базой данных и обрабатывает запросы от клиентов. Сервер Oracle включает экземпляр базы данных и базу данных, которая содержит фактические данные.
3. Протоколы связи (Communication Protocols): Протоколы связи определяют стандартные методы обмена данными между клиентами и сервером Oracle. Некоторые из распространенных протоколов, используемых в архитектуре Oracle, включают TCP/IP, HTTP, Net8/Net9, SQL\*Net и другие. Эти протоколы обеспечивают безопасную и надежную передачу данных между клиентами и сервером.
4. Драйверы (Drivers): Драйверы являются программными компонентами, которые обеспечивают взаимодействие между клиентами и сервером Oracle. Они переводят запросы клиентов в формат, понятный серверу, и обеспечивают обратную связь с результатами запросов. Драйверы могут быть специфичными для каждой платформы и языка программирования.
5. Сеть (Network): Сеть играет важную роль в клиент-серверной архитектуре Oracle, поскольку она обеспечивает физическое соединение между клиентами и сервером. Сеть может быть локальной (LAN) или глобальной (WAN, интернет) в зависимости от расположения клиентов и сервера. Надежность, пропускная способность и безопасность сети имеют большое значение для эффективного функционирования клиент-серверной архитектуры.

Архитектура клиент-серверной технологии Oracle позволяет множеству клиентов обращаться к одному или нескольким серверам для доступа к базе данных и выполнения операций. Клиенты и серверы взаимодействуют посредством протоколов связи и драйверов, обеспечивая эффективное управление данными и обработку запросов.

* + 1. Табличные пространства БД ORACLE.

В базе данных Oracle табличные пространства (Tablespaces) представляют собой логические контейнеры, в которых хранятся данные таблиц, индексов, кластеров и других объектов базы данных. Табличные пространства определяются для организации и управления физическим расположением данных на диске. Вот некоторые основные характеристики и особенности табличных пространств в Oracle:

1. Роль табличных пространств: Табличные пространства играют важную роль в управлении данными в Oracle. Они позволяют группировать объекты базы данных по логическим категориям и контролировать их размещение и распределение на физических устройствах.
2. Создание и управление табличными пространствами: Табличные пространства создаются с помощью команды CREATE TABLESPACE в Oracle. При создании можно указать различные параметры, такие как размер, местоположение файлов данных, типы хранения и другие параметры, которые определяют характеристики табличного пространства.
3. Атрибуты табличных пространств: Табличные пространства имеют различные атрибуты, которые определяют их поведение и свойства. Некоторые из атрибутов включают размер (например, исходный размер, автономный рост), типы хранения (например, локальное хранение, хранение сжатия), параметры доступа и другие характеристики.
4. Организация сегментов: Табличные пространства содержат сегменты, которые представляют собой логические единицы хранения данных. Сегменты могут быть различных типов, таких как сегменты таблиц, индексов, кластеров и других объектов базы данных. Они распределяются и управляются внутри табличного пространства.
5. Механизмы управления данными: Табличные пространства предоставляют механизмы управления данными, такие как управление пространством (extents), аллокацией блоков данных, управление свободными и занятыми областями, управление автономным расширением и сжатием данных.
6. Мульти-тенантные табличные пространства: В версии Oracle 12c и выше появилась функциональность мульти-тенантных баз данных, где можно создавать табличные пространства как общие для нескольких баз данных (поддерживаются также и классические табличные пространства для отдельных баз данных).

Табличные пространства в Oracle предоставляют гибкость и управляемость для организации и хранения данных в базе данных. Они позволяют логически разделить данные на различные категории и управлять их физическим расположением на диске, что способствует эффективности и производительности базы данных.

* + 1. Режимы работы БД ORACLE.

База данных Oracle имеет несколько режимов работы, которые определяют доступность и функциональность базы данных. Вот некоторые основные режимы работы базы данных Oracle:

1. Режим нормальной работы (Normal Mode): Это режим работы базы данных, когда она полностью функционирует и доступна для пользователей. В этом режиме база данных принимает запросы, выполняет транзакции и обеспечивает доступность данных.
2. Режим администрирования (Administrative Mode): Это режим работы, в котором база данных переводится для выполнения административных задач, таких как выполнение резервного копирования, восстановление данных, изменение структуры базы данных и других административных операций. В режиме администрирования доступ пользователей к базе данных может быть ограничен или полностью недоступен.
3. Режим восстановления (Recovery Mode): Режим восстановления активируется в случае сбоя базы данных или потери данных. В этом режиме выполняются операции по восстановлению базы данных до последней точки сохранения или до определенного момента в прошлом. Режим восстановления позволяет вернуть базу данных в консистентное состояние после сбоев.
4. Режим монтирования (Mount Mode): Режим монтирования активируется при запуске базы данных или при восстановлении базы данных после сбоя. В этом режиме выполняются операции по подключению файлов базы данных и проверке их целостности. В режиме монтирования база данных еще не доступна для пользователей и не выполняет транзакции.
5. Режим открытия (Open Mode): Режим открытия базы данных активируется после успешного монтирования и восстановления базы данных. В этом режиме база данных становится полностью доступной для пользователей, и она начинает выполнять транзакции и обрабатывать запросы.

Эти режимы работы базы данных Oracle предоставляют различные уровни доступности и функциональности в зависимости от ситуации и операций, выполняемых в базе данных.

* + 1. Программирование объектов базы с помощью PL/SQL.

PL/SQL (Procedural Language/Structured Query Language) - это язык программирования, разработанный специально для работы с базами данных Oracle. Он предоставляет возможности для создания, модификации и управления объектами базы данных, такими как таблицы, представления, процедуры, функции и триггеры. Вот некоторые основные аспекты программирования объектов базы данных с использованием PL/SQL:

1. Синтаксис и структура: PL/SQL сочетает в себе язык программирования и SQL. Он имеет структурированный блочный формат, состоящий из объявления секции, исполняющей секции и исключительной секции. PL/SQL использует ключевые слова, операторы, переменные, константы, типы данных и другие элементы для создания программных блоков.
2. Объекты базы данных: С помощью PL/SQL можно создавать и модифицировать объекты базы данных, такие как таблицы, представления, индексы, секвенсы и другие. PL/SQL предоставляет языковые конструкции для создания и управления объектами базы данных, включая операции SELECT, INSERT, UPDATE и DELETE.
3. Процедуры и функции: PL/SQL позволяет создавать процедуры и функции, которые являются наборами инструкций, выполняющих определенные задачи. Процедуры не возвращают значений, в то время как функции возвращают значения. Процедуры и функции могут принимать параметры, иметь локальные переменные и выполнять операции на базе данных.
4. Триггеры: PL/SQL позволяет создавать триггеры, которые реагируют на определенные события, происходящие в базе данных. Триггеры могут быть запущены перед или после операций вставки, обновления или удаления данных и могут выполнять определенные действия или проверки при возникновении этих событий.
5. Исключения: PL/SQL предоставляет механизм обработки исключений для управления ошибками и исключительными ситуациями в программе. С помощью блоков исключений можно перехватывать и обрабатывать ошибки, выполнять определенные действия при возникновении исключительной ситуации и обеспечивать целостность данных.
6. Доступ к данным: PL/SQL позволяет выполнять операции доступа к данным с использованием языка SQL. Это включает выборку данных из таблиц, изменение данных, выполнение агрегатных функций, управление транзакциями и другие операции SQL.

PL/SQL является мощным инструментом для программирования объектов базы данных в Oracle. Он предоставляет возможности для создания сложной бизнес-логики, обеспечения целостности данных, управления транзакциями и выполнения других задач, связанных с базами данных.

* + 1. Создания форм и отчетов в ORACLE APEX. Возможности.

Oracle Application Express (APEX) - это инструмент для разработки веб-приложений, который интегрирован с базой данных Oracle. Он предоставляет возможности создания форм и отчетов на основе данных из базы данных. Вот некоторые возможности создания форм и отчетов в Oracle APEX:

1. Создание форм (Forms): В Oracle APEX можно легко создавать формы для добавления, редактирования и удаления данных в базе данных. Формы автоматически связываются с таблицами или представлениями базы данных и предоставляют пользовательский интерфейс для ввода данных. Они могут содержать различные элементы управления, такие как текстовые поля, выпадающие списки, флажки и кнопки.
2. Создание отчетов (Reports): Oracle APEX предоставляет возможности создания разнообразных отчетов на основе данных из базы данных. Отчеты могут быть простыми табличными отчетами, сводными таблицами, диаграммами, календарями и другими форматами представления данных. Они могут содержать фильтры, сортировку, группировку и другие функции для управления данными в отчетах.
3. Гибкость макета и дизайна: Oracle APEX предлагает различные варианты макетов и тем оформления для создания пользовательского интерфейса форм и отчетов. Разработчики могут выбирать из готовых шаблонов или создавать собственные макеты, настраивать цвета, шрифты и стили в соответствии с требованиями проекта.
4. Интеграция с базой данных Oracle: Oracle APEX плотно интегрирован с базой данных Oracle, что обеспечивает простое создание форм и отчетов на основе существующих таблиц и представлений. Он автоматически создает SQL-запросы для доступа к данным и обеспечивает взаимодействие с базой данных без необходимости вручную писать сложные запросы.
5. Быстрый прототипирование и развертывание: Oracle APEX позволяет быстро создавать прототипы форм и отчетов, что помогает разработчикам и заказчикам быстро визуализировать и оценить функциональность приложения. Кроме того, разработанные приложения могут быть легко развернуты на сервере Oracle APEX и использованы в рабочей среде.

Oracle APEX предоставляет разнообразные возможности для создания форм и отчетов на основе данных из базы данных Oracle. Это позволяет разработчикам быстро создавать функциональные веб-приложения и обеспечивать удобный интерфейс для работы с данными.

* + 1. Триггеры, их назначение, создание и использование.

Триггеры в базах данных представляют собой специальные типы объектов, которые реагируют на определенные события, происходящие в базе данных, и выполняют предопределенные действия. Ниже приведена информация о назначении, создании и использовании триггеров:

Назначение триггеров:

* Автоматизация задач: Триггеры позволяют автоматически выполнять задачи или операции при возникновении определенных событий в базе данных. Например, можно создать триггер, который автоматически обновляет данные в одной таблице при изменении данных в другой таблице.
* Поддержка целостности данных: Триггеры могут использоваться для обеспечения целостности данных в базе данных. Например, можно создать триггер, который проверяет определенные условия перед вставкой, обновлением или удалением данных, и отклоняет операцию, если условия не выполняются.
* Аудит и мониторинг: Триггеры могут использоваться для регистрации и отслеживания изменений данных в базе данных. Например, можно создать триггер, который записывает информацию о каждом изменении данных в специальную аудиторскую таблицу.

Создание триггеров: Триггеры создаются с помощью языка SQL и специальных команд. Обычно они создаются внутри определенной таблицы и связаны с определенными событиями, такими как вставка, обновление или удаление данных в этой таблице. При создании триггера необходимо указать его имя, таблицу, событие, на которое он должен реагировать, и код, который должен быть выполнен при возникновении события.

Использование триггеров:

* Реакция на события: Триггеры выполняют определенные действия при возникновении событий, таких как вставка, обновление или удаление данных. Это может включать выполнение дополнительных операций, проверку условий, запись данных в другие таблицы и т. д.
* Управление целостностью данных: Триггеры могут использоваться для проверки и обеспечения целостности данных в базе данных. Они могут выполнять проверки на основе определенных условий и принимать решение о допустимости операции вставки, обновления или удаления данных.
* Аудит и мониторинг: Триггеры могут использоваться для регистрации и отслеживания изменений данных в базе данных. Они могут записывать информацию о событиях, изменениях данных, пользователях и других сущностях, связанных с изменениями данных.

Триггеры предоставляют мощные возможности для автоматизации задач, поддержки целостности данных и аудита в базах данных. Однако их использование требует внимательного проектирования и тестирования, чтобы избежать нежелательных побочных эффектов и обеспечить правильное функционирование базы данных.

* + 1. Архитектура СУБД PostgreSQL.

Архитектура СУБД PostgreSQL (PostgreSQL Database Management System) представляет собой многоуровневую архитектуру, которая включает различные компоненты для обработки и управления данными. Вот основные компоненты архитектуры PostgreSQL:

1. Клиент-серверная модель: PostgreSQL работает на основе клиент-серверной модели, где клиентские приложения взаимодействуют с сервером PostgreSQL для выполнения операций с данными. Клиентские приложения отправляют SQL-запросы на сервер, который обрабатывает запросы и возвращает результаты обратно клиенту.
2. Серверный процесс: Сервер PostgreSQL работает как отдельный процесс, который прослушивает определенный порт и принимает подключения от клиентских приложений. Когда клиент подключается, сервер создает новый процесс для обслуживания этого клиента.
3. Модуль запросов: Модуль запросов (Query Processing) отвечает за обработку SQL-запросов, полученных от клиентских приложений. Он выполняет разбор запроса, оптимизацию, планирование выполнения и фактическое выполнение запроса на доступных ресурсах.
4. Модуль хранения данных: Модуль хранения данных (Storage Manager) управляет физическим размещением данных на диске. Он обеспечивает чтение и запись данных на дисковое хранилище и управляет кэшированием данных в оперативной памяти для оптимизации производительности.
5. Модуль транзакций: Модуль транзакций (Transaction Manager) обеспечивает управление транзакциями в PostgreSQL. Он поддерживает ACID-свойства транзакций (атомарность, согласованность, изолированность и долговечность) и управляет блокировками и контролем конкурентного доступа к данным.
6. Модуль управления метаданными: Модуль управления метаданными (Metadata Manager) отвечает за хранение и управление метаданными базы данных PostgreSQL. Это включает информацию о таблицах, представлениях, индексах, схемах и других объектах базы данных.
7. Модуль безопасности: Модуль безопасности (Security Manager) обеспечивает механизмы аутентификации и авторизации пользователей. Он контролирует доступ к базе данных и ее объектам, определяет права доступа и обеспечивает защиту данных от несанкционированного доступа.
8. Модуль резервного копирования и восстановления: Модуль резервного копирования и восстановления (Backup and Recovery) отвечает за создание резервных копий данных и их восстановление в случае сбоев или потери данных. Он обеспечивает надежность и целостность данных путем создания резервных копий и журналов транзакций.

Архитектура PostgreSQL обеспечивает гибкость, масштабируемость и надежность работы с данными. Она позволяет эффективно обрабатывать SQL-запросы, управлять транзакциями, обеспечивать безопасность данных и обеспечивать возможность восстановления данных в случае сбоев.

* + 1. Функции SQL

Функции в языке SQL (Structured Query Language) представляют собой предопределенные или пользовательские операции, которые выполняются над данными или значениями в базе данных. Они предназначены для обработки данных, выполнения вычислений и возврата результирующих значений. Вот некоторые типы функций, используемых в SQL:

1. Агрегатные функции: Агрегатные функции выполняют вычисления над набором значений и возвращают единственное значение. Некоторые примеры агрегатных функций включают функции SUM (сумма), AVG (среднее значение), COUNT (количество), MAX (максимум) и MIN (минимум). Эти функции могут применяться к столбцам таблицы или результатам других запросов.
2. Скалярные функции: Скалярные функции принимают одно или несколько значений и возвращают единственное значение. Они могут использоваться для выполнения вычислений или преобразования данных. Например, функции CONCAT (объединение строк), UPPER (преобразование в верхний регистр), LOWER (преобразование в нижний регистр), SUBSTRING (подстрока) и другие.
3. Функции даты и времени: Функции даты и времени позволяют выполнять операции и вычисления, связанные с датами и временем. Например, функции DATEADD (добавление интервала к дате), DATEDIFF (разница между двумя датами), GETDATE (текущая дата и время) и другие.
4. Функции преобразования данных: Функции преобразования данных позволяют изменять типы данных или форматирование значений. Примеры таких функций включают функции CAST (преобразование типа данных), CONVERT (преобразование типа данных с указанием формата), TO\_CHAR (преобразование в строку с указанием формата) и другие.
5. Функции условного оператора: Функции условного оператора выполняют условные проверки и возвращают соответствующие значения в зависимости от результата проверки. Примеры таких функций включают функции CASE (условный оператор), COALESCE (возврат первого непустого значения), NULLIF (возврат NULL, если значения равны) и другие.

Это лишь некоторые примеры функций, используемых в SQL. СУБД могут предоставлять различные дополнительные функции, и также возможно создание пользовательских функций для выполнения специфических операций или вычислений в соответствии с требованиями приложения.

* + 1. Триггеры PostgreSQL

В PostgreSQL триггеры (triggers) представляют собой специальные объекты базы данных, которые реагируют на определенные события, происходящие в базе данных. Триггеры могут быть назначены на таблицы и выполнять определенные действия, как автоматически реагируя на изменения данных (вставка, обновление, удаление), так и выполняя пользовательские операции.

Основные аспекты триггеров в PostgreSQL:

1. Триггерные функции: Триггеры в PostgreSQL связаны с триггерными функциями (trigger functions), которые являются обработчиками событий. Триггерные функции определяют логику выполнения действий при срабатывании триггера. Они могут быть написаны на языке PL/pgSQL, PL/Python, PL/Perl и других языках, поддерживаемых PostgreSQL.
2. Типы триггеров: В PostgreSQL поддерживаются несколько типов триггеров, включая:
   1. Триггеры BEFORE: Срабатывают перед выполнением операции (вставка, обновление или удаление). Позволяют изменять данные перед тем, как они будут сохранены в базе данных.
   2. Триггеры AFTER: Срабатывают после выполнения операции. Позволяют выполнять дополнительные действия после изменения данных.
   3. Триггеры INSTEAD OF: Заменяют стандартное выполнение операции и предоставляют возможность определить пользовательскую логику обработки операции.
3. События триггеров: Триггеры в PostgreSQL могут быть назначены на различные события, включая вставку (INSERT), обновление (UPDATE) и удаление (DELETE) данных в таблице. Также возможно определение триггеров на уровне столбцов таблицы или на уровне базы данных.
4. Контекст выполнения триггера: Триггеры имеют доступ к данным, которые инициировали срабатывание триггера. В триггерных функциях можно использовать специальные переменные, такие как NEW и OLD, для доступа к новым и старым значениям данных, а также для выполнения дополнительных операций на основе этих значений.
5. Управление выполнением триггеров: PostgreSQL предоставляет возможности для управления выполнением триггеров, такие как порядок срабатывания триггеров, отключение и активация триггеров для конкретных операций или таблиц.

Триггеры в PostgreSQL предоставляют мощный механизм для автоматизации задач и поддержки бизнес-логики в базе данных. Они позволяют реагировать на изменения данных и выполнять дополнительные операции, такие как проверка целостности, аудит, ведение журнала изменений и другие действия, специфичные для приложения.

* + 1. Функции PL/pgSQL

Функции PL/pgSQL (PL/pgSQL functions) представляют собой пользовательские функции, написанные на языке PL/pgSQL, который является процедурным языком программирования, специфичным для PostgreSQL. PL/pgSQL предоставляет расширенные возможности для разработки хранимых процедур, триггеров и функций в PostgreSQL.

Вот некоторые основные аспекты функций PL/pgSQL:

1. Синтаксис: Функции PL/pgSQL определяются с использованием блока кода, который начинается с ключевого слова CREATE FUNCTION и заканчивается ключевым словом END. Внутри блока кода могут быть определены переменные, операторы, условия, циклы и другие конструкции языка PL/pgSQL.
2. Переменные: Функции PL/pgSQL могут использовать переменные для хранения и манипулирования данными. Переменные объявляются с использованием ключевого слова DECLARE и могут иметь различные типы данных, такие как целые числа, строки, даты и другие.
3. Операторы: Функции PL/pgSQL могут содержать различные операторы для выполнения операций над данными. Некоторые операторы включают операторы присваивания (:=), операторы условий (IF, CASE), операторы циклов (LOOP, WHILE, FOR) и другие.
4. Условные выражения: Функции PL/pgSQL могут содержать условные выражения для выполнения различных действий в зависимости от условий. Это позволяет реализовать логику ветвления в функции. Примеры условных выражений включают операторы IF, CASE и COALESCE.
5. Обработка исключений: Функции PL/pgSQL могут содержать блоки для обработки исключений, что позволяет обрабатывать ошибки и выполнить соответствующие действия. В блоке EXCEPTION можно определить, какие действия следует выполнить при возникновении определенного исключения.
6. Возврат значений: Функции PL/pgSQL могут возвращать значения, используя ключевое слово RETURN. Значение может быть любым допустимым типом данных PostgreSQL.
7. Вложенные вызовы функций: Функции PL/pgSQL могут включать вызовы других функций, включая встроенные функции PostgreSQL или пользовательские функции.

Функции PL/pgSQL позволяют разработчикам создавать более сложные и гибкие логические блоки кода в PostgreSQL. Они широко используются для создания хранимых процедур, бизнес-логики, обработки данных и других задач, требующих процедурного программирования в PostgreSQL.

1. **ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ**
   1. **Моделирование информационных систем**
      1. Процессный подход к управлению предприятием, реижиниринг.

Процессный подход к управлению предприятием и реинжиниринг являются ключевыми концепциями в области управления организациями. Давайте разберем каждый из них отдельно:

Процессный подход к управлению предприятием: Процессный подход представляет собой методологию управления, основанную на идее того, что организация может быть рассмотрена как совокупность взаимосвязанных и взаимодействующих процессов, которые определяют достижение ее целей и выполнение ее задач. Вместо традиционной функциональной организации, где акцент делается на отдельных отделах или функциях, процессный подход смотрит на предприятие как на совокупность процессов, которые взаимодействуют и создают ценность для клиентов и заинтересованных сторон.

Процессный подход включает в себя следующие основные принципы:

* Ориентация на клиента: фокус на предоставлении ценности клиентам и удовлетворении их потребностей.
* Комплексность и системность: понимание и оптимизация взаимосвязей и взаимодействий между процессами в организации.
* Управление результатами: оценка и управление результатами процессов для достижения целей организации.
* Постоянное улучшение: использование непрерывного цикла улучшения (например, ПДСА - планирование, выполнение, проверка, корректировка) для повышения эффективности и эффективности процессов.

Реинжиниринг: Реинжиниринг - это радикальная переработка и трансформация бизнес-процессов с целью достижения существенного улучшения в производительности, качестве, стоимости или предоставлении услуг. Он предполагает кардинальное переосмысление и перестройку процессов, а не только инкрементальное улучшение.

Основные шаги реинжиниринга включают:

* Идентификацию ключевых бизнес-процессов, которые нуждаются в изменениях.
* Переосмысление целей и задач процессов с учетом стратегии организации и потребностей клиентов.
* Разработку нового дизайна процессов, которые реализуют заданные цели и принципы.
* Внедрение и оперативное управление новыми процессами.
* Измерение и оценку результатов реинжиниринга для дальнейшего улучшения.

Реинжиниринг обычно связан с использованием информационных технологий для автоматизации и оптимизации переработанных процессов.

Комбинация процессного подхода к управлению и реинжиниринга позволяет организации переосмыслить свою деятельность, сосредоточиться на потребностях клиентов и достичь существенного улучшения в эффективности и результативности работы.

* + 1. Методология IDEF0 описания бизнес процессов на уровне функций.

Методология IDEF0 (Integration Definition for Function Modeling) представляет собой структурированный подход к описанию и моделированию бизнес-процессов на уровне функций. Она разработана для того, чтобы помочь организациям понять, анализировать и улучшать свои бизнес-процессы.

IDEF0 использует графическую нотацию для представления функций, связей между функциями и потоков данных внутри процесса. В основе методологии лежит иерархический подход, где более высокоуровневые функции разбиваются на более низкоуровневые функции до достижения достаточно детального описания процесса.

Основные элементы, используемые в IDEF0, включают:

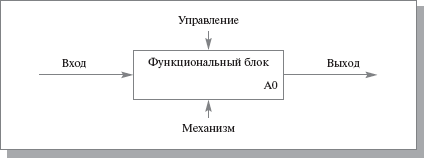
1. Функциональные блоки: Представляют основные функции, выполняемые в процессе. Каждый блок описывает, что происходит внутри функции, какие данные входят и выходят, а также какие управляющие данные могут влиять на выполнение функции.
2. Стрелки и потоки данных: Используются для представления потоков данных между функциональными блоками. Они указывают направление передачи данных и отражают взаимосвязи между функциями.
3. Управляющие данные: Показывают входные данные, которые могут влиять на выполнение функции. Они могут быть представлены в виде дополнительных входных стрелок или атрибутов функциональных блоков.
4. Механизмы управления: Описывают, как управляются функции внутри процесса, включая принятие решений, контроль и синхронизацию.

IDEF0 позволяет создавать подробные диаграммы бизнес-процессов, которые помогают визуализировать последовательность функций, потоки данных и взаимосвязи между ними. Это упрощает анализ и понимание процесса, а также позволяет выявить узкие места и возможности для оптимизации и улучшения производительности.

Методология IDEF0 широко используется для моделирования бизнес-процессов в различных отраслях, включая производство, информационные технологии, финансы и др. Она помогает компаниям оптимизировать свои операции, улучшить качество и доставку продуктов или услуг, а также повысить эффективность работы организации.

На диаграмме функциональный блок изображается прямоугольником (рис.). Каждая из четырех сторон функционального блока имеет свое определенное значение (роль), при этом:

* верхняя сторона имеет значение «Управление» (Control);
* левая сторона имеет значение «Вход» (Input);
* правая сторона имеет значение «Выход» (Output);
* нижняя сторона имеет значение «Механизм» (Mechanism).



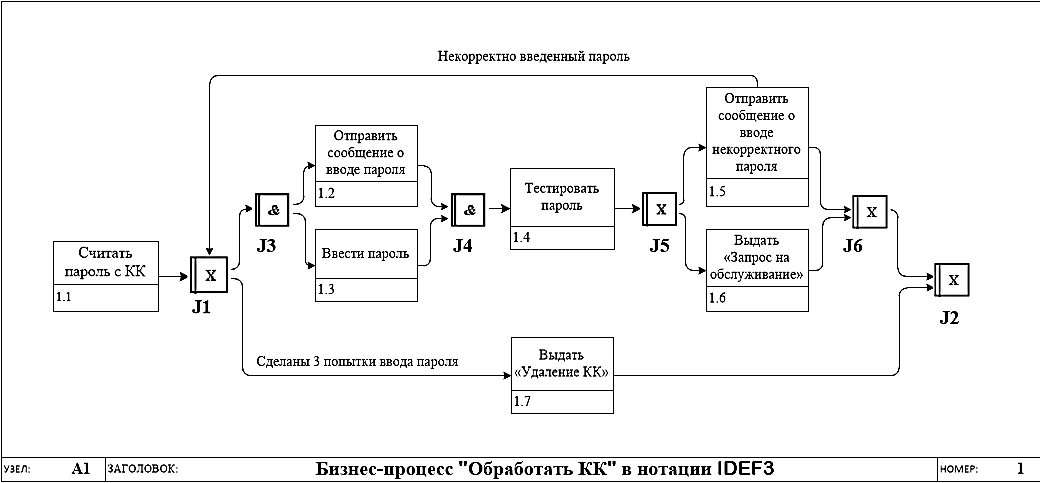
Интерфейсная дуга/стрелка (Arrow) отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, представленную данным функциональным блоком. Интерфейсные дуги часто называют потоками или стрелками.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон функционального блока подходит данная интерфейсная дуга, она носит название «входящей», «исходящей» или «управляющей».

* + 1. Методология описания бизнес - процессов IDEF3.

Этот метод предназначен для моделирования **последовательности выполнения действий** и взаимозависимости между ними в рамках процессов. Модели IDEF3 могут использоваться для детализации функциональных блоков IDEF0, не имеющих диаграмм декомпозиции.

Диаграммы IDEF3 отображают **действие** в виде прямоугольника. Действия именуются с использованием глаголов или отглагольных существительных, каждому из действий присваивается уникальный идентификационный номер. Все **связи** в IDEF3 являются однонаправленными и организуются слева направо. 

Типы связей IDEF3:

- *Временное предшествование* (Temporal precedence), простая стрелка. Исходное действие должно завершиться, прежде чем конечное действие сможет начаться.

- *Объектный поток* (Object flow), стрелка с двойным наконечником. Выход исходного действия является входом конечного действия. Исходное действие должно завершиться, прежде чем конечное действие сможет начаться. Наименования потоковых связей должны чётко идентифицировать объект, который передается с их помощью.

- *Нечеткое отношение* (Relationship), пунктирная стрелка.

Завершение одного действия может инициировать начало выполнения сразу нескольких других действий, или наоборот, определенное действие может требовать завершения нескольких других действий до начала своего выполнения (ветвление процесса). Ветвление процесса отражается с помощью специальных блоков:

- «И», блок со знаком **&**.

- «Исключающее ИЛИ» («одно из»), блок со знаком **Х**.

- «ИЛИ», блок со знаком **О**.

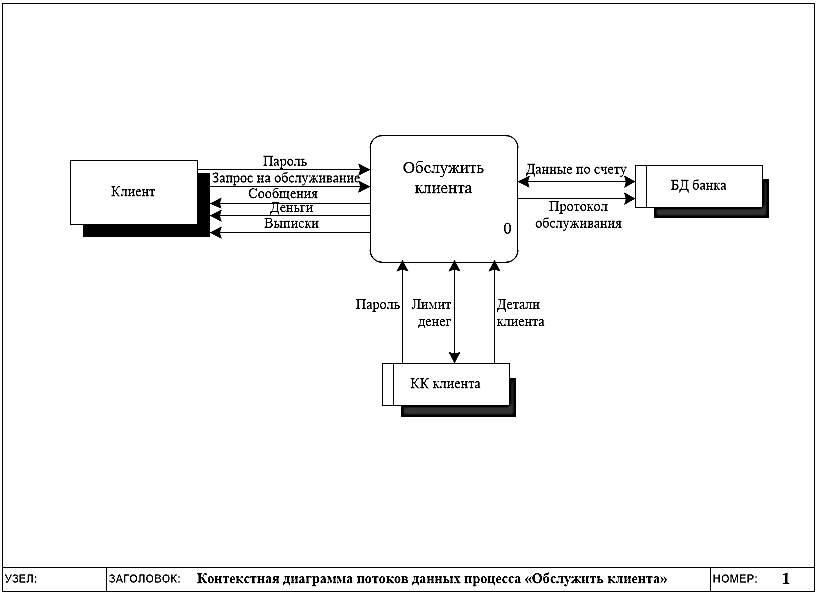
Если действия «И», «ИЛИ» должны выполняться синхронно, это обозначается двумя двойными вертикальными линиями внутри блока, асинхронно - одной.

Метод IDEF3 позволяет декомпозировать действие *несколько раз*, что обеспечивает документирование *альтернативных* потоков процесса в одной модели.

* + 1. Методология DFD описания бизнес процессов на уровне данных.

Методология DFD (Data Flow Diagram) представляет собой графический метод описания бизнес-процессов на уровне данных. Она широко используется для моделирования, анализа и документирования бизнес-процессов.

DFD-диаграммы состоят из нескольких элементов:

1. Процессы (Process): Они представляют собой действия или операции, выполняемые в рамках бизнес-процесса. Процессы обрабатывают данные, преобразуя их из одного состояния в другое.
2. Потоки данных (Data Flow): Потоки данных представляют передачу информации между процессами, внешними сущностями и хранилищами данных. Они указывают направление потока данных от источника к приемнику.
3. Внешние сущности (External Entity): Это внешние компоненты или системы, взаимодействующие с бизнес-процессом. Они могут быть представлены в виде внешних организаций, пользователей, внешних систем и т. д.
4. Хранилища данных (Data Store): Хранилища данных представляют собой места, где данные могут быть сохранены и извлечены. Это могут быть базы данных, файлы, архивы и т. д. 

DFD-диаграммы строятся с помощью стандартных символов и нотаций, которые обозначают каждый из этих элементов. Они позволяют визуализировать поток данных в бизнес-процессе, идентифицировать его источники и потребителей, а также определить связи между ними.

DFD-диаграммы могут быть разделены на уровни детализации, начиная с общего представления высокоуровневого процесса и последовательно уточняя его до более подробных уровней. Это позволяет получить более полное представление о потоках данных и их обработке внутри процесса.

DFD-диаграммы полезны для анализа и оптимизации бизнес-процессов, выявления узких мест, идентификации потребностей в информационных системах и взаимодействии с внешними сущностями. Они помогают улучшить понимание бизнес-процессов и обеспечить их эффективное функционирование.

* + 1. Описание бизнес-процессов при помощи блок-схем.

Описание бизнес-процессов с использованием блок-схем (или блок-диаграмм) является популярным и распространенным методом визуализации и анализа бизнес-процессов. Блок-схемы представляют собой графические диаграммы, в которых используются различные блоки и стрелки для представления шагов и связей в процессе.

Основные элементы, используемые в блок-схемах, включают:

1. Прямоугольники (блоки): Прямоугольники представляют шаги или операции, выполняемые в рамках бизнес-процесса. Каждый блок содержит описание действия, которое выполняется на этом шаге.
2. Ромбы (условия): Ромбы используются для представления условий или принятия решений в процессе. Они указывают на возможные ветвления и переходы в процессе в зависимости от определенных условий или критериев.
3. Стрелки (потоки управления): Стрелки представляют потоки управления или последовательность выполнения шагов в процессе. Они указывают направление движения и связи между блоками.
4. Ромбы с текстом (ввод/вывод данных): Ромбы с текстом используются для представления ввода или вывода данных в процессе. Они указывают на обмен информацией между процессом и внешними системами или пользователями.
5. Стрелки соединения: Стрелки соединения используются для связи различных элементов блок-схемы и указывают на последовательность выполнения шагов.

Блок-схемы позволяют визуализировать последовательность шагов и связи между ними в бизнес-процессе. Они позволяют легко понять поток работы, идентифицировать ветвления, принятие решений и взаимодействие с данными. Блок-схемы также могут использоваться для выявления узких мест, анализа производительности и оптимизации бизнес-процессов.

При создании блок-схем важно обеспечить понятность и четкость диаграммы, использовать понятные обозначения и правильно организовать последовательность шагов. Также рекомендуется документировать блок-схемы и обновлять их при изменении процесса для обеспечения актуальности и понимания бизнес-процессов.

* + 1. Моделирование данных, методология описания данных: сущность – связь. Реляционная модель данных.

Моделирование данных является важной составляющей процесса разработки информационных систем. Оно позволяет описать структуру данных, их связи и атрибуты, что облегчает понимание и проектирование баз данных.

Методология описания данных "сущность-связь" (Entity-Relationship, ER) широко используется для моделирования данных. В рамках этой методологии данные представляются в виде сущностей (entities), связей (relationships) между сущностями и их атрибутов.

Основные элементы, используемые в методологии "сущность-связь", включают:

1. Сущности (Entities): Сущности представляют различные объекты или понятия, о которых хранится информация в базе данных. Каждая сущность имеет свой уникальный идентификатор (первичный ключ) и атрибуты, описывающие ее характеристики.
2. Связи (Relationships): Связи определяют отношения между сущностями. Они могут быть однонаправленными или двунаправленными и могут иметь различные степени связности, такие как один к одному, один ко многим или многие ко многим.
3. Атрибуты (Attributes): Атрибуты определяют характеристики сущностей и связей. Они описывают данные, которые связаны с каждой сущностью или связью. Атрибуты могут быть простыми (неделимыми) или составными (состоящими из нескольких податрибутов).

ER-диаграммы используются для визуализации модели данных, где сущности представляются в виде прямоугольников, связи - в виде линий с указанием типа связи, атрибуты - в виде овалов или прямоугольников, связанных с соответствующей сущностью или связью.

Реляционная модель данных является одной из самых распространенных моделей данных. Она основана на концепции таблиц, называемых реляционными таблицами. Каждая таблица представляет собой сущность, а столбцы таблицы - атрибуты сущности. Связи между сущностями представляются с помощью ключей, связывающих записи в разных таблицах.

Реляционная модель данных обеспечивает структурированное хранение и организацию данных, а также возможности для запросов, анализа и манипулирования данными. Она позволяет строить эффективные базы данных, обеспечивая целостность, нормализацию и доступность данных.

Для создания реляционной модели данных на основе ER-диаграммы требуется преобразование сущностей, связей и атрибутов в соответствующие таблицы, столбцы и ключи. Также необходимо определить связи между таблицами с использованием внешних ключей.

Реляционная модель данных широко применяется в разработке баз данных для различных приложений и систем. Она обладает гибкостью, масштабируемостью и поддержкой стандартных языков запросов, таких как SQL (Structured Query Language).

* + 1. Имитационное моделирование на GPSS разомкнутых и замкнутых СеМО. Расчёт задержек.

Имитационное моделирование на GPSS (General Purpose Simulation System) позволяет анализировать и оценивать производительность систем в реальном времени. GPSS поддерживает моделирование как разомкнутых систем с множеством входов и выходов (СеМО - система с экзогенными воздействиями и множеством выходов), так и замкнутых систем с циклическими взаимодействиями (СеМО - система с эндогенными воздействиями и одним выходом).

В разомкнутых системах на GPSS входные потоки (экзогенные воздействия) могут представлять поступление заявок, сообщений, клиентов и т.д. Эти входные потоки могут быть заданы с помощью статистических распределений или конкретных временных интервалов. Выходные потоки могут представлять завершение обработки заявок или другие события, которые моделируются в системе.

В замкнутых системах на GPSS взаимодействия происходят циклически, где выход одной единицы является входом для другой. Такие системы могут быть использованы для моделирования процессов с обратной связью, например, циклической обработки или циклического обновления данных.

Расчет задержек в имитационном моделировании на GPSS может осуществляться путем измерения времени, затраченного на обработку заявок или прохождение через систему. GPSS предоставляет возможности для установки меток времени в модели, чтобы фиксировать начало и окончание обработки заявок. Задержка рассчитывается как разница между временем окончания и временем поступления заявки.

GPSS также позволяет собирать статистические данные о времени обработки, задержках, пропускной способности и других параметрах производительности системы. Эти данные могут быть использованы для анализа и оптимизации процессов, определения узких мест, оценки эффективности системы и принятия решений для улучшения производительности.

В целом, имитационное моделирование на GPSS предоставляет инструменты для анализа и оценки производительности систем в различных условиях, позволяя оптимизировать и принимать обоснованные решения для улучшения работы системы.

Для расчета разомкнутых и замкнутых СеМО (систем с экзогенными и эндогенными воздействиями) в имитационном моделировании на GPSS, необходимо учесть следующие шаги:

1. Определение модели системы: Определите структуру системы, включая входы, выходы, компоненты и связи между ними. Это может включать в себя определение их характеристик, таких как распределение времени обработки, количество ресурсов и т.д.
2. Определение входных потоков (разомкнутая СеМО): Определите входные потоки заявок, которые поступают в систему. Это может включать в себя определение интервалов между поступлениями заявок или использование статистических распределений для определения времени поступления заявок.
3. Определение обработки заявок: Определите процессы обработки заявок в системе. Укажите время, необходимое для выполнения каждой операции или процесса, а также использование ресурсов (если применимо).
4. Определение выходных потоков (разомкнутая СеМО): Определите выходные потоки, представляющие завершение обработки заявок или другие события, которые происходят в системе.
5. Установка параметров модели: Задайте параметры моделирования, такие как время моделирования, количество повторений или статистические параметры для сбора данных.
6. Запуск имитации: Запустите имитацию модели в GPSS с заданными параметрами. GPSS будет моделировать процессы в системе и собирать статистические данные во время выполнения.
7. Анализ результатов: Анализируйте полученные результаты, включая время обработки заявок, задержки, пропускную способность и другие параметры производительности системы. Можно провести сравнительный анализ различных сценариев или вариантов модели для определения оптимальной конфигурации или улучшения производительности.

Важно отметить, что точный расчет разомкнутых и замкнутых СеМО может быть сложным и требовать учета различных факторов, таких как взаимодействие компонентов, временные ограничения и прочее. В реальных системах может потребоваться более сложное моделирование и дополнительные инструменты для точного анализа и расчета производительности.

* + 1. Проведение экспериментов на имитационных моделях. Оценка точности результатов моделирования. Адекватность моделей.

Проведение экспериментов на имитационных моделях является важной частью процесса имитационного моделирования. Эксперименты позволяют оценить производительность системы, провести анализ различных сценариев и принять обоснованные решения на основе полученных результатов.

Оценка точности результатов моделирования:

1. Верификация модели: Перед проведением экспериментов необходимо убедиться в правильности модели. Это включает проверку логики модели, соответствие модели реальной системе, правильность параметров и алгоритмов моделирования. Верификация помогает убедиться, что модель работает правильно и даёт корректные результаты.
2. Валидация модели: Валидация модели заключается в сравнении результатов моделирования с реальными данными или наблюдениями. Сравнение позволяет определить, насколько точно модель отражает поведение системы. Если результаты моделирования близки к реальным данным, можно считать модель валидной.
3. Чувствительностный анализ: Чувствительностный анализ позволяет оценить влияние изменения параметров модели на ее результаты. Изменение параметров и последующее анализирование результатов помогает определить, насколько точно модель реагирует на изменения и позволяет оценить ее надежность.
4. Статистический анализ: При проведении экспериментов на имитационной модели можно применять статистический анализ для оценки результатов. Это может включать расчет средних значений, доверительных интервалов, анализ вариации и других статистических метрик. Статистический анализ позволяет получить количественные оценки и понять уровень неопределенности результатов моделирования.

Адекватность моделей: Адекватность моделей отражает степень соответствия модели реальной системе, ее способность воспроизводить основные характеристики системы и предсказывать поведение. Адекватность модели может быть оценена субъективно или с использованием количественных метрик, таких как сравнение результатов моделирования с реальными данными, сравнение с экспертными оценками или сравнение с другими независимыми моделями.

Оценка адекватности модели может включать сравнение статистических метрик, анализ чувствительности, анализ поведения модели в различных сценариях и другие методы. Важно помнить, что адекватность модели может изменяться в зависимости от конкретных целей моделирования и контекста применения.

Общая оценка точности и адекватности результатов моделирования требует комплексного подхода, включающего проверку и верификацию модели, валидацию с использованием реальных данных, статистический анализ и анализ поведения модели. Это позволяет получить доверительность в результаты моделирования и сделать обоснованные выводы и рекомендации на основе модели.

* + 1. Финансовое моделирование бизнес – процессов.

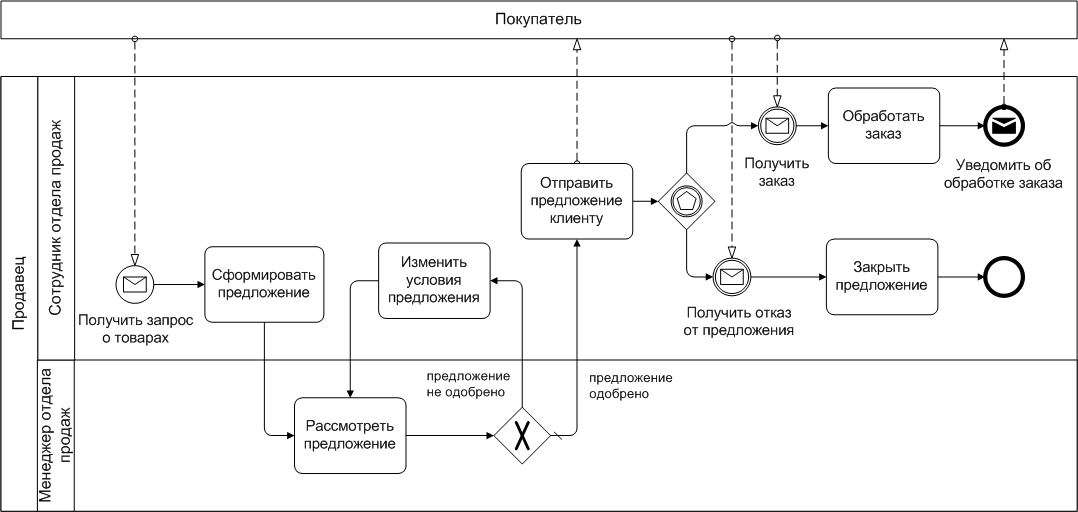
Финансовое моделирование бизнес-процессов - это процесс создания математических моделей и анализа финансовых аспектов бизнес-процессов с целью прогнозирования результатов, оценки финансовой устойчивости, принятия решений и оптимизации финансовых показателей. Финансовое моделирование может быть применено к различным бизнес-процессам, включая планирование бюджета, оценку инвестиций, анализ стоимости, управление рисками и т.д.

Вот некоторые основные аспекты финансового моделирования бизнес-процессов:

1. Прогнозирование финансовых показателей: Финансовое моделирование позволяет прогнозировать доходы, расходы, прибыль, денежный поток и другие финансовые показатели на основе предполагаемых входных данных и параметров. Модель может учитывать различные факторы, такие как объем продаж, цены, издержки производства, налоги и другие факторы, влияющие на финансовые результаты.
2. Оценка инвестиций: Финансовое моделирование может использоваться для оценки финансовой целесообразности инвестиций и проектов. Это может включать расчет показателей окупаемости, внутренней нормы доходности (IRR), чистой приведенной стоимости (NPV) и других финансовых метрик. Моделирование позволяет провести сценарный анализ и оценить влияние различных факторов на финансовые результаты проекта.
3. Управление рисками: Финансовое моделирование может быть использовано для оценки и управления финансовыми рисками. Это может включать моделирование вероятностей различных событий, оценку влияния рисков на финансовые результаты и разработку стратегий снижения рисков. Моделирование рисков позволяет принять информированные решения и оптимизировать финансовую устойчивость бизнес-процессов.
4. Планирование бюджета: Финансовое моделирование может быть использовано для планирования бюджета и определения оптимального распределения ресурсов. Моделирование позволяет оценить финансовые потребности и возможности, учесть различные переменные, провести сценарный анализ и разработать бюджетные планы.
5. Анализ стоимости: Финансовое моделирование может быть применено для анализа стоимости бизнес-процессов и принятия решений по их оптимизации. Моделирование стоимости позволяет оценить эффективность использования ресурсов, определить факторы, влияющие на стоимость, и разработать стратегии снижения издержек.

Финансовое моделирование бизнес-процессов может выполняться с использованием различных инструментов и методов, таких как электронные таблицы (например, Microsoft Excel), специализированные программы для финансового моделирования, статистические методы и другие аналитические подходы.

* + 1. Моделирование бизнес-процессов в нотации BPMN.

Моделирование бизнес-процессов с использованием нотации BPMN (Business Process Model and Notation) является широко распространенным подходом. BPMN предоставляет стандартную и наглядную графическую нотацию для моделирования бизнес-процессов, которая позволяет легко визуализировать и анализировать потоки работы и взаимодействия между различными участниками процесса. 

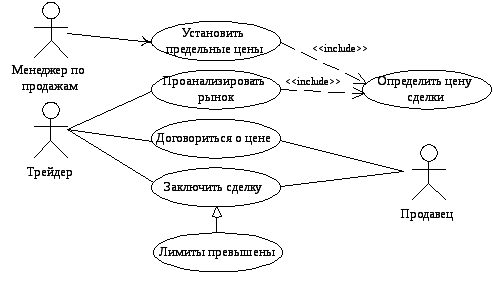
Вот некоторые ключевые элементы и концепции, используемые в нотации BPMN:

1. Задачи (Tasks): Задачи представляют основные шаги или действия, которые выполняются в рамках бизнес-процесса. Они обозначаются прямоугольниками с закругленными углами и содержат описание выполняемой работы.
2. События (Events): События представляют важные моменты или изменения состояния в бизнес-процессе. Они могут быть начальными событиями (начало процесса), промежуточными событиями (события, которые происходят внутри процесса) или конечными событиями (завершение процесса). События обозначаются кругами.
3. Шлюзы (Gateways): Шлюзы определяют разветвления и объединения в потоках работы. Они могут быть исключительными (Exclusive), параллельными (Parallel), включающими (Inclusive) и другими типами шлюзов. Шлюзы обозначаются ромбами.
4. Потоки (Flows): Потоки представляют направление передачи информации, управления или выполнения между элементами процесса. Потоки обозначаются стрелками и указывают на направление потока работы.
5. Пулы и дорожки (Pools and Lanes): Пулы представляют основные участники или организации, участвующие в бизнес-процессе, а дорожки (лейны) разделяют пулы на различные роли или подразделения. Пулы и дорожки обозначаются прямоугольниками, разделенными вертикальными линиями.

С помощью элементов и концепций BPMN можно создавать диаграммы бизнес-процессов, которые наглядно отображают последовательность шагов, потоки работы, принимаемые решения, связи между участниками и другую информацию, необходимую для анализа и оптимизации бизнес-процессов.

* + 1. UML-моделирование. Диаграмма прецедентов.

UML (Unified Modeling Language) - это стандартный язык моделирования, который широко используется для визуализации, проектирования и документирования различных аспектов системы. UML предоставляет набор графических диаграмм, которые помогают описывать структуру, функциональность и взаимодействие компонентов системы.

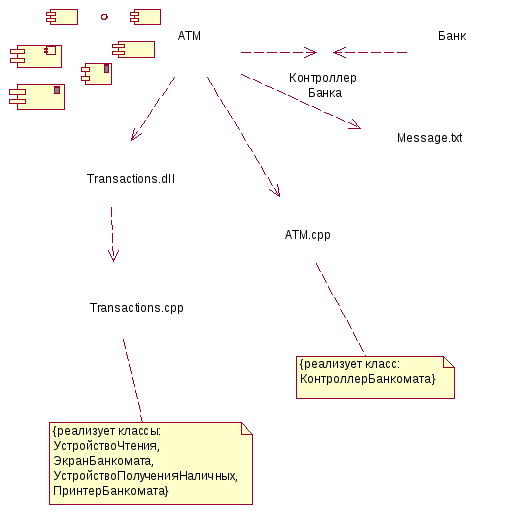
Одной из наиболее распространенных диаграмм UML является диаграмма прецедентов (Use Case Diagram). Диаграмма прецедентов помогает моделировать функциональные требования системы, иллюстрируя взаимодействие акторов (пользователей или внешних систем) с системой и их сценарии использования. 

Вот основные элементы, используемые на диаграмме прецедентов:

1. Акторы (Actors): Акторы представляют внешние сущности, которые взаимодействуют с системой. Акторы могут быть пользователями, ролями, другими системами или внешними компонентами. Они представляются в виде стикеров с именем актора.
2. Прецеденты (Use Cases): Прецеденты представляют конкретные функциональные возможности системы или действия, которые могут быть выполнены акторами. Они описываются в виде эллипсов с названием прецедента.
3. Отношения: Диаграмма прецедентов использует различные отношения для описания взаимодействий между акторами и прецедентами. Наиболее распространенные отношения включают включение (include), расширение (extend) и обобщение (generalization).
   1. Отношение включения (include) указывает, что один прецедент включает в себя функциональность другого прецедента. Это означает, что выполнение включаемого прецедента является обязательным для выполнения включающего прецедента.
   2. Отношение расширения (extend) указывает, что некоторая функциональность может быть расширена в определенных сценариях. Расширяемый прецедент может быть выполнен по умолчанию, но в некоторых случаях может происходить расширение функциональности.
   3. Отношение обобщения (generalization) используется для установления иерархии между прецедентами. Это означает, что один прецедент обобщает или является более общим, а другой прецедент специализирует или наследует эту общую функциональность.

Диаграмма прецедентов помогает визуализировать взаимодействия между акторами и прецедентами, а также идентифицировать основные функциональные требования системы. Это полезный инструмент для обмена информацией между командой разработки и заказчиком, а также для уточнения и документирования требований к системе.

* + 1. UML-моделирование. Диаграмма компонентов.

Диаграмма компонентов (Component Diagram) в UML (Unified Modeling Language) используется для визуализации и описания архитектуры системы, выделения ее основных компонентов и их взаимосвязей. Диаграмма компонентов помогает понять структуру системы и организацию компонентов, а также связи и зависимости между ними. 

Вот основные элементы, используемые на диаграмме компонентов:

1. Компоненты (Components): Компоненты представляют физические или логические единицы функциональности системы. Компоненты могут быть программными модулями, библиотеками, пакетами, фреймворками или даже физическими устройствами. Компоненты обозначаются прямоугольниками с названием компонента.
2. Интерфейсы (Interfaces): Интерфейсы определяют способ взаимодействия между компонентами или между компонентами и внешними сущностями. Интерфейсы определяются с помощью сигнатур методов и атрибутов. Интерфейсы обозначаются прямоугольниками с названием интерфейса.
3. Зависимости (Dependencies): Зависимости представляют отношения между компонентами, показывающие, что один компонент зависит от другого. Зависимости могут быть направленными и показывают, что компонент использует функциональность или интерфейс другого компонента. Зависимости обозначаются стрелками, указывающими на направление зависимости.
4. Размещение (Deployment): Размещение показывает физическое размещение компонентов и аппаратного обеспечения системы. Это может включать серверы, компьютеры, устройства хранения данных и другое оборудование. Размещение обозначается с помощью узлов (Nodes) и связей между компонентами и узлами.
5. Связи (Connections): Связи представляют взаимодействия и обмен данными между компонентами или между компонентами и интерфейсами. Связи могут быть направленными и множественными, показывая типы коммуникации и передаваемую информацию.

Диаграмма компонентов позволяет легко понять структуру системы, выделить ее ключевые компоненты, показать зависимости и взаимодействия между ними, а также распределение компонентов по физическим узлам. Это полезный инструмент при проектировании и документировании архитектуры системы.

* + 1. UML-моделирование. Диаграмма последовательности.

Диаграмма последовательности (Sequence Diagram) в UML (Unified Modeling Language) используется для визуализации и описания взаимодействия между объектами или компонентами в рамках определенного сценария выполнения. Диаграмма последовательности позволяет представить последовательность сообщений, передаваемых между объектами, и порядок их выполнения во времени.

Вот основные элементы, используемые на диаграмме последовательности:

1. Объекты (Objects): Объекты представляют конкретные экземпляры классов или компонентов, участвующие во взаимодействии. Каждый объект обозначается вертикальным прямоугольником с названием объекта или именем класса.
2. Жизненный цикл объектов (Lifelines): Жизненный цикл объекта показывает его существование и продолжительность во времени. Он представляется вертикальной линией, пересекающей объект. Жизненный цикл может быть активным, когда объект выполняет действия, или пассивным, когда объект ожидает сообщений.
3. Сообщения (Messages): Сообщения представляют передачу информации или вызов методов между объектами. Сообщения указываются стрелками, которые показывают направление передачи сообщения. Сообщения могут быть синхронными (подчеркнутые стрелки), асинхронными (неподчеркнутые стрелки), ответными (стрелки с возвратом) и др.
4. Фреймы (Frames): Фреймы используются для организации сообщений, отображая контекст выполнения в рамках определенного временного интервала. Фреймы помогают визуально группировать связанные сообщения и показывать их последовательность.
5. Операции (Operations): Операции представляют методы или функции, которые вызываются объектами в процессе взаимодействия. Операции обозначаются над сообщениями и содержат имя операции и аргументы.

Диаграмма последовательности помогает визуализировать порядок и взаимодействие объектов в рамках конкретного сценария выполнения. Она особенно полезна для моделирования взаимодействия между объектами в рамках определенного метода или функциональности системы, что позволяет лучше понять и анализировать поведение системы на уровне объектов.



* + 1. UML-моделирование. Диаграмма классов.

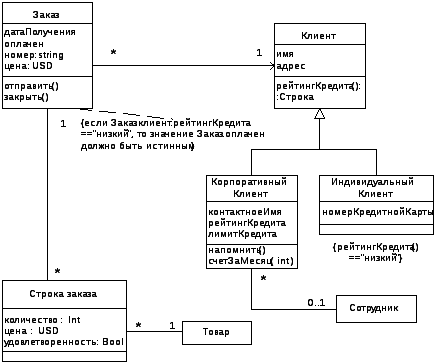


Диаграмма классов (Class Diagram) в UML (Unified Modeling Language) используется для визуализации структуры и отношений между классами в системе. Диаграмма классов позволяет описать атрибуты классов, методы, связи и наследование между классами.

Вот основные элементы, используемые на диаграмме классов:

1. Классы (Classes): Классы представляют абстрактные модели объектов или компонентов системы. Каждый класс обозначается прямоугольником, в котором указывается имя класса. Внутри прямоугольника указываются атрибуты класса (поля) и методы (операции).
2. Атрибуты (Attributes): Атрибуты представляют данные или свойства, принадлежащие классу. Они указываются внутри прямоугольника класса и имеют имя и тип данных. Атрибуты могут быть публичными (+), приватными (-) или защищенными (#), что определяет доступность к атрибутам из других классов.
3. Методы (Methods): Методы представляют операции или функции, которые могут быть выполнены классом. Они также указываются внутри прямоугольника класса и имеют имя, параметры и возвращаемый тип данных. Методы могут быть публичными (+), приватными (-) или защищенными (#), что определяет доступность к методам из других классов.
4. Отношения (Relationships): Отношения показывают связи и взаимодействия между классами. Наиболее распространенные отношения включают ассоциацию, наследование, реализацию и агрегацию.
   1. Ассоциация (Association) показывает связь между двумя классами и может быть однонаправленной или двунаправленной. Она представляет, что один класс имеет ссылку на другой класс.
   2. Наследование (Inheritance) показывает отношение "является" между классами, где один класс наследует свойства и методы другого класса. Наследующий класс называется подклассом, а класс, от которого он наследует, называется суперклассом или базовым классом.
   3. Реализация (Implementation) показывает, что класс реализует интерфейс или абстрактный класс. Реализующий класс обязан реализовать методы, определенные в интерфейсе или абстрактном классе.
   4. Агрегация (Aggregation) показывает отношение "имеет" или "содержит" между классами. Она представляет, что один класс содержит другой класс в качестве своего члена или частичного компонента.

Диаграмма классов позволяет описать структуру системы, иллюстрируя классы, их атрибуты, методы и взаимосвязи между ними. Она является важным инструментом для проектирования и анализа архитектуры системы, а также для обмена информацией между членами команды разработки и другими заинтересованными сторонами.

* 1. **Проектирование информационных систем**
     1. Определение информационной системы. Классификация информационных систем. Архитектура информационной системы.

Информационная система (Information System) - это совокупность взаимосвязанных компонентов, которые собирают, хранят, обрабатывают и предоставляют информацию для поддержки принятия решений, управления и выполнения бизнес-процессов в организации.

Основные компоненты информационной системы включают:

1. Аппаратное обеспечение (Hardware): Компьютеры, серверы, сетевое оборудование и другие физические устройства, необходимые для функционирования информационной системы.
2. Программное обеспечение (Software): Операционные системы, прикладные программы и базы данных, которые выполняют различные функции обработки информации в информационной системе.
3. Данные (Data): Информация, которая хранится и обрабатывается в информационной системе. Они могут включать текст, числа, изображения, видео и другие форматы.
4. Процессы (Processes): Логика и алгоритмы, по которым информация обрабатывается и передается в информационной системе. Это включает в себя бизнес-процессы, правила обработки данных, алгоритмы вычислений и другие операционные процедуры.
5. Люди (People): Пользователи информационной системы, которые взаимодействуют с ней, используют ее функциональность и принимают решения на основе предоставленной информации.

Классификация информационных систем может быть основана на различных критериях, включая функциональность, охват деятельности и уровень организации. Вот некоторые распространенные классификации:

1. По функциональности:
   1. Операционные информационные системы (Operational Information Systems): Поддерживают ежедневные операции и транзакции в организации, такие как системы учета и управления складом.
   2. Управленческие информационные системы (Management Information Systems): Предоставляют информацию для принятия решений на уровне руководства и планирования, такие как системы управленческой отчетности и аналитики.
   3. Стратегические информационные системы (Strategic Information Systems): Ориентированы на поддержку стратегического планирования и принятия решений высшего уровня, такие как системы бизнес-интеллекта и прогнозирования.
2. По охвату деятельности:
   1. Внутриорганизационные информационные системы: Ориентированы на внутренние потребности организации и обеспечивают взаимодействие между различными подразделениями и функциональными областями.
   2. Межорганизационные информационные системы: Обеспечивают взаимодействие между несколькими организациями и позволяют обмен информацией и совместную работу внутри цепочки поставок или бизнес-экосистемы.
3. По уровню организации:
   1. Корпоративные информационные системы: Предоставляют общую информационную инфраструктуру для всей организации, охватывая различные функциональные области и подразделения.
   2. Отделные информационные системы: Направлены на поддержку конкретных функциональных областей или подразделений, таких как финансы, маркетинг или производство.

Архитектура информационной системы определяет структуру, компоненты и взаимосвязи между ними в информационной системе. Она включает описание различных уровней абстракции, таких как бизнес-архитектура, архитектура приложений, архитектура данных и техническая архитектура. Цель архитектуры информационной системы - обеспечить эффективное функционирование и взаимодействие компонентов системы, а также обеспечить масштабируемость, безопасность и поддерживаемость информационной системы в целом.

* + 1. Оценки показателей качества СМО M/M/1, открытой СеМО.

Для оценки показателей качества СМО (Системы массового обслуживания) M/M/1 (M/M/1 - одноканальная СМО с экспоненциальным распределением поступления и обслуживания) следующие основные показатели могут быть рассчитаны:

1. Интенсивность поступления (λ): Это среднее число заявок, поступающих в СМО за единицу времени. В СМО M/M/1 интенсивность поступления обозначается как λ и представляет собой среднее число заявок в единицу времени.
2. Интенсивность обслуживания (μ): Это средняя скорость обработки заявок в СМО. В СМО M/M/1 интенсивность обслуживания обозначается как μ и представляет собой среднее число обслуживаемых заявок в единицу времени.
3. Интенсивность потерь (P₀): Это вероятность отказа в обслуживании заявки в СМО. В СМО M/M/1 интенсивность потерь обозначается как P₀ и представляет собой вероятность того, что заявка будет отклонена из-за отсутствия доступного обслуживающего устройства.
4. Среднее число заявок в системе (L): Это среднее число заявок, находящихся в системе (в очереди и в обслуживании) в любой момент времени.
5. Среднее время пребывания заявки в системе (W): Это среднее время, которое заявка проводит в системе (в очереди и в обслуживании) до ее полного завершения.
6. Среднее число заявок в очереди (Lq): Это среднее число заявок, находящихся в очереди в любой момент времени.
7. Среднее время ожидания заявки в очереди (Wq): Это среднее время, которое заявка проводит в очереди до начала обслуживания.

Расчет этих показателей основан на формулах, связанных с процессом обслуживания и потока заявок в СМО M/M/1. Формулы могут быть достаточно сложными и зависят от специфических параметров системы. Они могут быть найдены в теории СМО и статистическом анализе.

Зная значения интенсивности поступления (λ) и интенсивности обслуживания (μ), можно использовать эти формулы для расчета остальных показателей качества СМО M/M/1.

* + 1. Методология проектирования ИС (выбор объекта и цели проектирования, применение системного подхода в проектировании, внешнее и внутреннее проектирование, выделение подсистем, средства проектирования).

Методология проектирования информационной системы (ИС) включает несколько этапов и подходов, которые помогают разработать эффективную и функциональную ИС. Вот основные аспекты методологии проектирования ИС:

1. Выбор объекта и цели проектирования: Первый шаг в проектировании ИС - определение объекта и цели проектирования. Это может быть новая система, модернизация существующей системы или автоматизация определенных бизнес-процессов. Четкое определение целей помогает сосредоточиться на конкретных требованиях и ожиданиях от ИС.
2. Применение системного подхода в проектировании: Системный подход в проектировании ИС означает рассмотрение системы как целого, а не только ее отдельных компонентов. Это включает анализ взаимосвязей и взаимодействий между компонентами системы, а также учет внешней среды и потребностей пользователей. Системный подход помогает создать сбалансированную и интегрированную ИС.
3. Внешнее и внутреннее проектирование: Внешнее проектирование ИС связано с определением интерфейсов и взаимодействия системы с внешними субъектами, такими как пользователи, другие системы или сторонние компоненты. Внутреннее проектирование, с другой стороны, фокусируется на внутренних компонентах и структуре системы, таких как архитектура, базы данных, алгоритмы и модули.
4. Выделение подсистем: В процессе проектирования ИС может потребоваться выделение подсистем - логических или функциональных блоков системы, выполняющих определенные задачи или функции. Выделение подсистем помогает упростить процесс разработки, позволяет распределить задачи между разработчиками и обеспечить логическую структуру системы.
5. Средства проектирования: Для проектирования ИС используются различные средства и инструменты, которые помогают в создании диаграмм, моделей и спецификаций. Примеры таких средств включают CASE-системы (Computer-Aided Software Engineering), UML-диаграммы (Unified Modeling Language), функциональные блоки и т. д. Выбор средств зависит от конкретных требований и предпочтений разработчиков.

Все эти аспекты методологии проектирования ИС помогают обеспечить системный и структурированный подход к созданию информационных систем, которые отвечают требованиям и целям бизнеса или организации.

* + 1. Математическая постановка задачи проектирования ИС как задачи многокритериальной оптимизации. Методика решения задачи многокритериальной оптимизации.

Математическая постановка задачи проектирования информационной системы (ИС) как задачи многокритериальной оптимизации заключается в определении набора критериев, которые необходимо учесть при выборе оптимального варианта проектирования. Каждый критерий представляет собой определенную характеристику или показатель, который должен быть улучшен или оптимизирован.

Примеры критериев в задаче проектирования ИС могут включать стоимость разработки, время разработки, производительность, надежность, безопасность, удобство использования, масштабируемость и другие.

Методика решения задачи многокритериальной оптимизации включает следующие шаги:

1. Формулировка целевых функций: Для каждого критерия определяются целевые функции, которые описывают желаемое значение этого критерия. Целевые функции могут быть линейными, нелинейными, дискретными или комбинацией различных функций.
2. Формализация ограничений: Ограничения могут быть связаны с ресурсами, бюджетом, временем или другими ограничениями, которые необходимо учесть при проектировании ИС. Ограничения также могут быть представлены в виде равенств или неравенств.
3. Создание множества альтернативных решений: Генерируются различные варианты проектирования ИС, которые отвечают целевым функциям и удовлетворяют ограничениям. Это может включать комбинирование различных компонентов, параметров и конфигураций системы.
4. Оценка альтернатив: Каждая альтернатива оценивается по каждому критерию с использованием соответствующих целевых функций. Оценки могут быть числовыми значениями или качественными оценками.
5. Многокритериальная оптимизация: Применяются методы многокритериальной оптимизации для выбора оптимального варианта проектирования. Это может включать методы взвешивания критериев, методы Парето-оптимальности, методы агрегации критериев и другие.
6. Анализ и выбор решения: Анализируются полученные результаты, проводится сравнение альтернативных решений по критериям и выбирается наилучшее решение с учетом предпочтений и требований.

Методика решения задачи многокритериальной оптимизации может варьироваться в зависимости от конкретной задачи проектирования ИС и используемых методов оптимизации.

* + 1. Постановка задачи линейного программирования. Методы решения.

Постановка задачи линейного программирования (ЛП) заключается в определении оптимального значения линейной целевой функции при заданных линейных ограничениях. В общей форме задача ЛП может быть сформулирована следующим образом:

Максимизировать (или минимизировать) линейную целевую функцию: Z = c₁x₁ + c₂x₂ + ... + cₙxₙ,

при условиях: a₁₁x₁ + a₁₂x₂ + ... + a₁ₙxₙ ≤ b₁, a₂₁x₁ + a₂₂x₂ + ... + a₂ₙxₙ ≤ b₂, ... aₘ₁x₁ + aₘ₂x₂ + ... + aₘₙxₙ ≤ bₘ,

x₁, x₂, ..., xₙ ≥ 0.

Где:

* Z - целевая функция, которую требуется максимизировать или минимизировать.
* c₁, c₂, ..., cₙ - коэффициенты целевой функции.
* x₁, x₂, ..., xₙ - переменные решения.
* a₁₁, a₁₂, ..., aₘₙ - коэффициенты ограничений.
* b₁, b₂, ..., bₘ - правые части ограничений.
* m - количество ограничений.
* n - количество переменных решения.

Методы решения задачи линейного программирования включают:

1. Симплекс-метод: Симплекс-метод является классическим методом решения задач ЛП. Он основан на последовательном переходе от одного допустимого базисного решения к другому с целью нахождения оптимального решения. Симплекс-метод эффективно решает задачи с небольшим количеством переменных и ограничений.
2. Двойственность: Двойственность в линейном программировании связывает исходную задачу с ее двойственной формой, которая представляет собой задачу, обратную к исходной. Решение двойственной задачи позволяет получить информацию о чувствительности исходной задачи к изменениям в ограничениях и коэффициентах.
3. Метод внутренней точки: Метод внутренней точки является итерационным методом, который ищет оптимальное решение внутри допустимой области, приближаясь к оптимуму с помощью последовательности внутренних точек.
4. Методы приближенного решения: Для больших и сложных задач ЛП могут применяться методы приближенного решения, такие как методы генетических алгоритмов, методы метапоиска и другие эвристические подходы.

Выбор конкретного метода решения задачи ЛП зависит от размеров задачи, структуры ограничений и доступных вычислительных ресурсов.

* + 1. Разработка информационного обеспечения ИС (организация ввода, хранения, представления и вывода информации, проектирование выходных документов и требования к их качеству).

Разработка информационного обеспечения информационной системы (ИС) включает организацию ввода, хранения, представления и вывода информации, а также проектирование выходных документов и установление требований к их качеству. Вот основные аспекты каждого из этих элементов:

1. Организация ввода информации:
   1. Определение источников информации и способов ее получения, например, через пользовательский ввод, автоматический сбор данных и т. д.
   2. Разработка интерфейсов ввода данных, включая формы, поля, кнопки и другие элементы пользовательского интерфейса, обеспечивающие удобство и эффективность ввода данных.
   3. Установление проверок и валидации вводимых данных, чтобы гарантировать их корректность и целостность.
2. Организация хранения информации:
   1. Выбор и проектирование базы данных, которая будет использоваться для хранения информации. Это может быть реляционная база данных, иерархическая база данных, документоориентированная база данных и т. д.
   2. Определение структуры данных и таблиц базы данных, включая определение полей, связей и ограничений целостности данных.
   3. Установление процедур резервного копирования и восстановления данных, чтобы обеспечить их сохранность и доступность.
3. Организация представления информации:
   1. Проектирование пользовательского интерфейса, который будет отображать информацию из ИС. Это может быть в виде форм, таблиц, графиков, диаграмм и т. д.
   2. Установление принципов организации и представления данных, чтобы обеспечить удобство использования и понимания информации пользователями.
   3. Использование средств визуализации и графического дизайна для создания привлекательных и интуитивно понятных интерфейсов.
4. Организация вывода информации:
   1. Проектирование выходных документов, отчетов, графиков и других форматов представления информации для пользователей или других систем.
   2. Определение структуры и содержания выходных документов, включая формат, шаблоны, заголовки, таблицы, графики и т. д.
   3. Установление требований к качеству выходных документов, таких как точность, актуальность, ясность, согласованность и т. д.

Важно учитывать потребности и ожидания пользователей ИС, чтобы создать информационное обеспечение, которое соответствует их требованиям и обеспечивает удобство использования и эффективность работы.

* + 1. Разработка программного обеспечения ИС (выбор инструментальных средств программирования, разработка программной документации).

Разработка программного обеспечения информационной системы (ИС) включает выбор инструментальных средств программирования и разработку программной документации. Вот основные этапы этих процессов:

1. Выбор инструментальных средств программирования:
   1. Определение требований к программному обеспечению ИС и анализ возможных вариантов инструментальных средств программирования.
   2. Выбор языка программирования, который наилучшим образом соответствует требованиям проекта. Это может быть язык общего назначения (например, Java, C++, Python), язык специализированного назначения или фреймворк.
   3. Выбор среды разработки, интегрированной среды разработки (IDE) или других инструментов, которые обеспечат эффективность и удобство разработки.
2. Разработка программной документации:
   1. Создание спецификации требований, где формулируются функциональные и нефункциональные требования к программному обеспечению ИС.
   2. Разработка технического проекта, где описывается архитектура системы, используемые компоненты, модули, интерфейсы и другие аспекты разработки.
   3. Создание документации пользователя, которая предоставляет информацию о функциональности системы, инструкции по использованию, справочные материалы и т. д.
   4. Разработка кодовой документации, включая комментарии в исходном коде, описание классов, функций и других элементов программы.

В процессе разработки программного обеспечения ИС также следует придерживаться современных методологий разработки, таких как Agile или Waterfall, и использовать практики управления версиями и контроля изменений.

Важно создавать чистый, модульный и хорошо документированный код, который будет понятен и легко поддерживаться другими разработчиками в будущем.

* + 1. Архитектура ЭВМ. Неймановская архитектурная реализация ЭВМ. Основные характеристики ЭВМ.

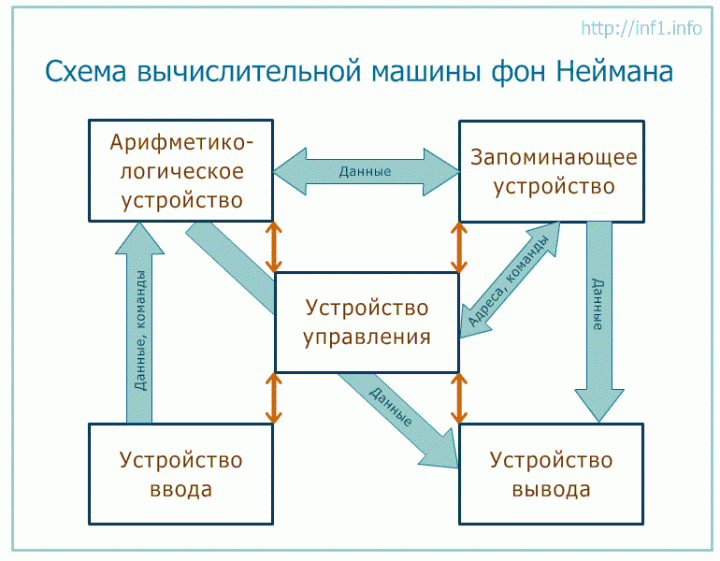
Архитектура ЭВМ (Электронно-вычислительной машины) описывает структуру и организацию компонентов, которые обеспечивают функционирование и выполнение вычислений. Неймановская архитектура является одной из наиболее распространенных архитектурных реализаций ЭВМ. Вот основные характеристики ЭВМ:

Принцип Неймановской архитектуры:

* Центральное процессорное устройство (ЦПУ) выполняет вычисления и управляет операциями в ЭВМ.
* ЦПУ состоит из арифметико-логического устройства (АЛУ), устройства управления и регистров памяти.
* Программа и данные хранятся в одной памяти, из которой они считываются для выполнения.
* Программы представлены в двоичном коде и исполняются последовательно.

Основные компоненты ЭВМ:

* Центральное процессорное устройство (ЦПУ): отвечает за выполнение инструкций и арифметических операций.
* Память: хранит данные и программы, используемые ЭВМ.
* Ввод-вывод (ВВ): обеспечивает обмен информацией с внешними устройствами, такими как клавиатура, монитор, принтер и т. д.
* Шина данных: передает данные между ЦПУ, памятью и внешними устройствами.
* Шина управления: передает сигналы управления и команды между различными компонентами ЭВМ.



Основные характеристики ЭВМ:

* Производительность: измеряется в количестве операций, которые машина может выполнить за определенное время.
* Память: объем памяти, доступный для хранения программ и данных.
* Скорость: время, за которое машина выполняет одну операцию или программу.
* Масштабируемость: возможность расширения и модификации системы, чтобы удовлетворять растущим потребностям пользователей.
* Надежность: способность системы сохранять работоспособность и обеспечивать доступность данных даже при сбоях или ошибках.
* Совместимость: возможность интеграции и взаимодействия с другими системами или устройствами.

Неймановская архитектура является основой для большинства современных компьютеров и обеспечивает эффективную обработку информации и выполнение вычислений.

* + 1. Операционные системы. Виды, классификация. Архитектура. Принцип работы. Установка. Настройка.

Операционная система (ОС) является программным обеспечением, которое управляет ресурсами компьютерной системы и обеспечивает взаимодействие между пользователем и аппаратным обеспечением. ОС выполняет множество функций, таких как управление памятью, управление процессами, управление файлами, ввод-вывод данных и другие.

Вот основные аспекты операционных систем:

Виды и классификация операционных систем:

* ОС для персональных компьютеров (Windows, macOS, Linux).
* Серверные операционные системы (Windows Server, Linux Server, UNIX).
* Мобильные операционные системы (Android, iOS).
* Встроенные операционные системы (RTOS - Real-Time Operating Systems) для устройств встроенных систем.
* Сетевые операционные системы (Novell NetWare, FreeBSD).

Архитектура операционной системы:

* Монолитная архитектура: все компоненты ОС находятся в одной программе, которая работает в привилегированном режиме.
* Микроядерная архитектура: базовые функции ОС реализованы в микроядре, а дополнительные функции вынесены в отдельные модули, работающие в привилегированном или пользовательском режиме.
* Гибридная архитектура: комбинирует элементы монолитной и микроядерной архитектур, чтобы обеспечить баланс производительности и гибкости.

Принцип работы операционной системы:

* ОС загружается при старте компьютера и управляет запуском и выполнением приложений.
* ОС предоставляет интерфейсы для взаимодействия с пользователем, такие как графический интерфейс (GUI) или командная строка.
* ОС управляет ресурсами компьютера, включая процессорное время, память, дисковое пространство и устройства ввода-вывода.
* ОС обеспечивает безопасность, защиту данных и контроль доступа к системе и ее ресурсам.
* ОС управляет процессами, планирует и распределяет ресурсы для выполнения задач.
* ОС обеспечивает управление файлами и директориями, организацию и доступ к данным на диске.

Установка и настройка операционной системы:

* Установка ОС включает выбор соответствующей дистрибуции, загрузочного носителя и следование инструкциям по установке.
* После установки ОС необходимо настроить параметры системы, такие как язык, часовой пояс, сетевые настройки и безопасность.
* Также требуется установка и настройка драйверов для оборудования компьютера.
* Дополнительные настройки могут включать настройку пользовательских учетных записей, программного обеспечения и служб.

Обратите внимание, что процесс установки и настройки операционной системы может различаться в зависимости от конкретной ОС и ее версии.

Виды операционных систем:

* Серверные операционные системы: предназначены для управления серверами и обеспечения работы сетей.
* Операционные системы для настольных компьютеров: предназначены для использования на обычных настольных и переносных компьютерах.
* Встроенные операционные системы: устанавливаются на специализированные устройства, такие как мобильные телефоны, автомобили, бытовая техника и другие электронные устройства.

Классификация операционных систем:

* Однопользовательские операционные системы: позволяют одному пользователю выполнять одну задачу в определенный момент времени.
* Многопользовательские операционные системы: позволяют нескольким пользователям одновременно работать на одном компьютере или через сеть.
* Однопроцессорные и многопроцессорные операционные системы: поддерживают работу на компьютерах с одним или несколькими процессорами соответственно.

Архитектура операционной системы: ОС состоит из нескольких компонентов, включая ядро (kernel), файловую систему, драйверы устройств, пользовательский интерфейс и другие. Ядро является основной частью ОС и отвечает за управление аппаратными ресурсами, планирование задач, управление памятью, взаимодействие с устройствами и обеспечение безопасности.

Принцип работы операционной системы: ОС обеспечивает интерфейс между пользователем и аппаратными ресурсами компьютера. Она управляет выполнением программ, управляет памятью, обеспечивает взаимодействие с устройствами ввода-вывода, обеспечивает защиту данных и обеспечивает множество других функций.

Установка и настройка операционной системы:

* Установка: процесс установки ОС на компьютер. Обычно включает выбор ОС, запуск установочного процесса, выбор раздела для установки, настройку параметров и завершение установки.
* Настройка: процесс настройки ОС для индивидуальных потребностей пользователя. Включает настройку сетевых параметров, настройку безопасности, установку дополнительного программного обеспечения и настройку пользовательских предпочтений.
  + 1. Файловые системы (FAT 32, NTFS).

Файловая система - это метод организации и хранения файлов на компьютере или другом устройстве. Вот две распространенные файловые системы:

FAT32 (File Allocation Table 32):

* FAT32 является старшей версией файловой системы FAT, разработанной компанией Microsoft.
* Она поддерживает максимальный размер раздела до 2 ТБ и максимальный размер файла до 4 ГБ.
* FAT32 обеспечивает хорошую совместимость с различными операционными системами, включая Windows, Mac и Linux.
* Однако у нее есть ограничения, такие как ограничение размера файла и относительно низкая эффективность использования места на диске из-за большого размера кластера.

NTFS (New Technology File System):

* NTFS является более современной файловой системой, разработанной компанией Microsoft.
* Она предоставляет более надежное и эффективное хранение данных, чем FAT32.
* NTFS поддерживает большие размеры файлов (до 16 ТБ) и разделов (в зависимости от операционной системы).
* Она обеспечивает более надежное восстановление данных при сбоях и поддерживает различные функции безопасности, такие как разрешения доступа и шифрование файлов.
* NTFS часто используется в операционных системах Windows, но не всегда совместима с другими операционными системами, такими как macOS и некоторыми дистрибутивами Linux.

Выбор между FAT32 и NTFS зависит от конкретных требований и совместимости с операционными системами. Если нужна более надежная и расширенная функциональность, то рекомендуется использовать NTFS. Если требуется совместимость с различными ОС или работа с устройствами, которые поддерживают только FAT32, то можно выбрать FAT32.

* + 1. Блок схема ЭВМ. Характеристики блоков. Принципы работы блоков. Выполнение программ.

Блок-схема ЭВМ (Электронно-вычислительная машина) является графическим представлением работы компьютерной системы и ее компонентов. Она позволяет визуально представить последовательность операций, логику выполнения программ и поток данных в компьютере. В блок-схеме используются различные блоки и стрелки, которые соединяют блоки, обозначающие последовательность действий.

Характеристики блоков в блок-схеме ЭВМ:

* Прямоугольник (процесс): обозначает операцию или действие, которое должно быть выполнено. Примеры: вычисления, сравнения, ввод-вывод данных и т.д.
* Ромб (условие): используется для представления условного оператора или ветвления в программе. Примеры: проверка условий, циклы и т.д.
* Параллелограмм (ввод/вывод): обозначает операции ввода или вывода данных с устройствами. Примеры: чтение данных с клавиатуры, запись данных на диск и т.д.
* Шестиугольник (функция): представляет функцию или подпрограмму, которая может быть вызвана из основной программы.
* Стрелки: соединяют блоки и указывают направление потока выполнения программы.

Принципы работы блоков в блок-схеме ЭВМ:

* Последовательность (Sequence): блоки выполняются последовательно, один за другим, в порядке их расположения на блок-схеме.
* Ветвление (Decision): блоки условий позволяют определить различные варианты выполнения программы в зависимости от условий.
* Циклы (Loop): блоки циклов позволяют повторять выполнение набора операций несколько раз, пока выполняется определенное условие.

Выполнение программы в блок-схеме ЭВМ:

* Программа начинается с начального блока и последовательно выполняет блоки в порядке, определенном стрелками.
* Если встречается блок условия, программа проверяет условие и выбирает соответствующую ветвь для выполнения.
* После выполнения блока программа продолжает выполнять следующие блоки до завершения или до достижения блока, который требует возврата или перехода к другому месту в программе.
* Циклы могут приводить к многократному выполнению определенных блоков до тех пор, пока условие цикла истинно.

Блок-схемы ЭВМ представляют удобный способ визуализации логики программ и алгоритмов, что помогает разработчикам и аналитикам лучше понять и проанализировать работу программного кода и компьютерных систем.

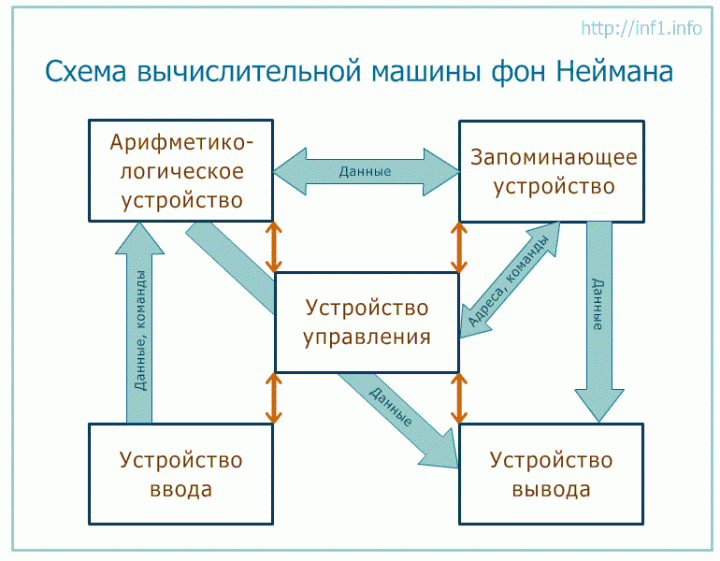
Архитектура ЭВМ (Электронно-вычислительной машины) описывает структуру и организацию компонентов, которые обеспечивают функционирование и выполнение вычислений. Неймановская архитектура является одной из наиболее распространенных архитектурных реализаций ЭВМ. Вот основные характеристики ЭВМ:

Принцип Неймановской архитектуры:

* Центральное процессорное устройство (ЦПУ) выполняет вычисления и управляет операциями в ЭВМ.
* ЦПУ состоит из арифметико-логического устройства (АЛУ), устройства управления и регистров памяти.
* Программа и данные хранятся в одной памяти, из которой они считываются для выполнения.
* Программы представлены в двоичном коде и исполняются последовательно.

Основные компоненты ЭВМ:

* Центральное процессорное устройство (ЦПУ): отвечает за выполнение инструкций и арифметических операций.
* Память: хранит данные и программы, используемые ЭВМ.
* Ввод-вывод (ВВ): обеспечивает обмен информацией с внешними устройствами, такими как клавиатура, монитор, принтер и т. д.
* Шина данных: передает данные между ЦПУ, памятью и внешними устройствами.
* Шина управления: передает сигналы управления и команды между различными компонентами ЭВМ.



Основные характеристики ЭВМ:

* Производительность: измеряется в количестве операций, которые машина может выполнить за определенное время.
* Память: объем памяти, доступный для хранения программ и данных.
* Скорость: время, за которое машина выполняет одну операцию или программу.
* Масштабируемость: возможность расширения и модификации системы, чтобы удовлетворять растущим потребностям пользователей.
* Надежность: способность системы сохранять работоспособность и обеспечивать доступность данных даже при сбоях или ошибках.
* Совместимость: возможность интеграции и взаимодействия с другими системами или устройствами.

Неймановская архитектура является основой для большинства современных компьютеров и обеспечивает эффективную обработку информации и выполнение вычислений.

* + 1. Глобальные вычислительные сети. Архитектура. Принцип работы. Маршрутизация, коммутация, протоколы, адресация. Семиуровневая модель OSI. Техническое обеспечение.

Глобальные вычислительные сети (ГВС) - это распределенные сети компьютеров, которые объединяют компьютеры и другие устройства по всему миру, обеспечивая обмен информацией и ресурсами. ГВС обеспечивают глобальное соединение между компьютерами и поддерживают различные услуги, такие как передача данных, электронная почта, веб-сервисы и другие приложения.

Архитектура ГВС:

* ГВС основаны на клиент-серверной архитектуре, где компьютеры в сети разделяются на клиентские устройства (пользовательские компьютеры) и серверные устройства (серверы), которые предоставляют определенные услуги и ресурсы.
* ГВС включают различные сетевые компоненты, такие как маршрутизаторы, коммутаторы, межсетевые экраны и другие сетевые устройства, которые обеспечивают передачу данных и коммуникацию между устройствами в сети.

Принцип работы ГВС:

* Маршрутизация: процесс определения наилучшего пути передачи данных от отправителя к получателю через сеть. Маршрутизаторы выполняют эту функцию, принимая решения о пересылке данных на основе информации в сетевых пакетах.
* Коммутация: процесс передачи данных от отправителя к получателю внутри сети. Существуют различные методы коммутации, включая пакетную коммутацию (передача данных в виде пакетов), канальную коммутацию (резервирование выделенного канала для передачи данных) и пакетно-канальную коммутацию (сочетание двух предыдущих методов).
* Протоколы: ГВС используют протоколы передачи данных, которые определяют правила и форматы для обмена информацией между устройствами в сети. Примеры протоколов включают TCP/IP (Transmission Control Protocol/Internet Protocol), HTTP (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol) и другие.
* Адресация: каждое устройство в ГВС имеет уникальный сетевой адрес, который позволяет идентифицировать его в сети. В IP-сетях используется IP-адрес для идентификации устройств, а в Ethernet-сетях используется MAC-адрес.

Семиуровневая модель OSI (Open Systems Interconnection):

Семиуровневая модель OSI является стандартной моделью, описывающей взаимодействие компьютерных систем в сетях.

Модель включает семь уровней: физический, канальный, сетевой, транспортный, сеансовый, представительный и прикладной.

Каждый уровень выполняет определенные функции и обеспечивает определенные услуги, такие как управление физическими соединениями, маршрутизацию, обработку ошибок, управление сеансами и т.д.

Семиуровневая модель OSI (Open Systems Interconnection) - это стандартная модель, которая описывает взаимодействие компьютерных систем в сетях. Модель OSI была разработана Международной организацией по стандартизации (ISO) и представляет собой концептуальную рамку, которая позволяет разделить функции сетевых протоколов на семь уровней. Каждый уровень выполняет определенные задачи и предоставляет определенные услуги для обеспечения эффективного обмена данными между компьютерами в сети. Ниже приведены основные уровни модели OSI:

1. Физический уровень (Physical Layer): Этот уровень отвечает за передачу физических сигналов по среде связи, таких как провода или беспроводные волны. Он определяет электрические, механические и функциональные аспекты физического соединения между устройствами.
2. Канальный уровень (Data Link Layer): Канальный уровень обеспечивает надежную передачу данных между соседними узлами в сети. Он отвечает за обнаружение и исправление ошибок, контроль доступа к среде передачи и управление потоком данных.
3. Сетевой уровень (Network Layer): Сетевой уровень отвечает за маршрутизацию пакетов данных в сети. Он определяет адресацию узлов, выбор оптимального пути для доставки данных и управление трафиком.
4. Транспортный уровень (Transport Layer): Транспортный уровень обеспечивает надежную доставку данных от источника к назначению. Он разделяет данные на меньшие блоки, управляет потоком данных и обеспечивает контроль целостности и управление ошибками.
5. Сеансовый уровень (Session Layer): Сеансовый уровень устанавливает, управляет и завершает сеансы связи между приложениями на разных узлах в сети. Он также обеспечивает синхронизацию и восстановление сеансов при сбоях.
6. Представительный уровень (Presentation Layer): Представительный уровень отвечает за перевод, кодирование и сжатие данных, чтобы обеспечить совместимость между различными системами и представление данных в понятной форме.
7. Прикладной уровень (Application Layer): Прикладной уровень предоставляет интерфейс для взаимодействия приложений пользователя с сетевыми службами. На этом уровне работают приложения, такие как веб-браузеры, почтовые клиенты, файловые менеджеры и другие программы.

Модель OSI обеспечивает стандартизацию и взаимодействие между различными типами компьютеров и сетевыми устройствами, позволяя им обмениваться данными в распределенной сетевой среде.

Техническое обеспечение ГВС:

* Для построения и поддержки ГВС необходимо различное техническое оборудование, включая сетевые коммутаторы, маршрутизаторы, межсетевые экраны, серверы, сетевые кабели, сетевые интерфейсы и другие устройства.
* Также важным аспектом является выбор правильного программного обеспечения, включая операционные системы, протоколы, системы безопасности и другие приложения, необходимые для управления и обеспечения работоспособности сети.
  + 1. Локальные вычислительные сети. Архитектуры ЛВС. Методы доступа. Принцип работы. Маршрутизация, коммутация, протоколы, адресация.

Локальные вычислительные сети (ЛВС) - это сети, которые охватывают ограниченную географическую область, такую как здание, офис или кампус. ЛВС предназначены для обеспечения связи между компьютерами, устройствами хранения данных и другими периферийными устройствами внутри ограниченной территории. Вот некоторые аспекты ЛВС:

Архитектуры ЛВС:

* Звездообразная архитектура: В этой архитектуре все устройства подключаются к центральному коммутатору или концентратору. Все данные проходят через центральное устройство, которое координирует передачу информации между устройствами в сети.
* Шинная архитектура: В этой архитектуре все устройства подключаются к одной шине, через которую передаются данные. Каждое устройство прослушивает шину, и только одно устройство может передавать данные в определенный момент времени.
* Кольцевая архитектура: В этой архитектуре устройства подключены в кольцо, и данные передаются от одного устройства к другому по кольцу. Каждое устройство принимает и передает данные до тех пор, пока они не достигнут нужного узла.

Методы доступа:

* CSMA/CD (Carrier Sense Multiple Access with Collision Detection) - метод доступа, применяемый в Ethernet-сетях. Устройства передают данные только после определения, что среда передачи свободна. Если происходит коллизия (две устройства передают данные одновременно), они обнаруживают коллизию и повторно передают данные.
* CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) - метод доступа, используемый в беспроводных сетях, где передача данных может быть подвержена помехам. Передача данных происходит после прослушивания среды и отправки запроса на передачу.

Принцип работы:

* ЛВС обеспечивают связь между устройствами посредством передачи данных в виде пакетов или кадров. Устройства в ЛВС могут передавать данные друг другу, используя сетевые протоколы и методы доступа.

Маршрутизация и коммутация:

* Маршрутизация относится к процессу выбора оптимального пути для передачи данных между сетями или сегментами сети. Маршрутизаторы принимают решения о передаче пакетов данных на основе информации о сетевых адресах.
* Коммутация относится к процессу пересылки данных внутри сети. Коммутаторы являются устройствами, которые обеспечивают коммутацию данных, передавая их только на нужные порты, исходя из адресов устройств.

Протоколы и адресация:

* В ЛВС используются различные протоколы для передачи данных, такие как TCP/IP, Ethernet, Wi-Fi и другие. Эти протоколы определяют правила и форматы данных для передачи в сети.
* Адресация в ЛВС используется для идентификации устройств и места назначения данных. Примеры адресации включают IP-адреса для идентификации компьютеров и MAC-адреса для идентификации сетевых интерфейсов устройств.

Семиуровневая модель OSI, упомянутая ранее, также применима к ЛВС и предоставляет общую структуру для организации протоколов и функций в сети.

* + 1. Семиуровневая модель OSI. Intranet. Техническое обеспечение.

Семиуровневая модель OSI (Open Systems Interconnection) - это стандартная модель, используемая для описания взаимодействия сетевых протоколов и функций в компьютерных сетях. Каждый уровень модели OSI выполняет определенные задачи и обеспечивает определенные сервисы для передачи данных. Ниже приведены уровни модели OSI:

1. Физический уровень (Physical Layer):
   * Определяет методы передачи физических сигналов по среде передачи данных, таких как проводная или беспроводная среда.
   * Отвечает за физическое соединение устройств и передачу битов данных.
2. Канальный уровень (Data Link Layer):
   * Обеспечивает безошибочную передачу данных между устройствами в пределах одной локальной сети (ЛВС).
   * Отвечает за управление доступом к среде передачи данных, обнаружение и исправление ошибок передачи данных.
3. Сетевой уровень (Network Layer):
   * Осуществляет маршрутизацию и пересылку данных между различными сетями.
   * Отвечает за установление путей передачи данных, преобразование логических адресов и управление трафиком в сети.
4. Транспортный уровень (Transport Layer):
   * Обеспечивает надежную и упорядоченную доставку данных между приложениями на конечных узлах.
   * Отвечает за сегментацию и сборку данных, контроль надежности доставки, управление потоком данных.
5. Сеансовый уровень (Session Layer):
   * Устанавливает, поддерживает и завершает соединение между приложениями на разных узлах.
   * Отвечает за управление сеансами связи, синхронизацию и восстановление соединения при сбоях.
6. Представительный уровень (Presentation Layer):
   * Отвечает за представление данных, кодирование, сжатие и шифрование данных для передачи.
   * Обеспечивает согласование синтаксического и семантического понимания данных между приложениями.
7. Прикладной уровень (Application Layer):
   * Предоставляет интерфейсы и протоколы для взаимодействия с приложениями пользователей.
   * Включает в себя различные протоколы, такие как HTTP, FTP, SMTP, DNS, и другие.

Intranet - это частная сеть, использующая технологии и протоколы Интернета для внутреннего использования в организации. Intranet обеспечивает сотрудникам доступ к различным информационным ресурсам, внутренним приложениям, коммуникации и совместной работе. Техническое обеспечение интранета включает в себя серверы, сетевое оборудование, операционные системы, базы данных, программное обеспечение для управления контентом и другие компоненты, необходимые для функционирования и обеспечения безопасности интранета.

Организация интранета и его настройка зависят от потребностей организации. Она включает в себя установку и конфигурирование серверов, настройку сетевого оборудования, создание безопасных соединений, реализацию аутентификации и авторизации пользователей, управление доступом к ресурсам интранета и другие административные задачи, связанные с обеспечением работы и безопасности интранета.

* + 1. Стек протоколов TCP/IP. Назначение основных протоколов.

Стек протоколов TCP/IP (Transmission Control Protocol/Internet Protocol) - это набор сетевых протоколов, которые используются для передачи данных в компьютерных сетях, особенно в Интернете. Включает в себя несколько протоколов, каждый из которых выполняет определенные функции. Ниже приведены основные протоколы стека TCP/IP и их назначение:

1. IP (Internet Protocol):
   * Определяет формат пакетов данных и адресацию для доставки данных в сети.
   * Обеспечивает маршрутизацию и передачу данных между сетями.
2. TCP (Transmission Control Protocol):
   * Обеспечивает надежную и упорядоченную доставку данных между узлами в сети.
   * Отвечает за разделение данных на пакеты, управление потоком данных, контроль надежности передачи и управление соединением между узлами.
3. UDP (User Datagram Protocol):
   * Обеспечивает передачу данных без гарантии доставки и упорядочивания.
   * Используется для передачи данных, где более важна скорость, чем надежность, например, в потоковом видео или VoIP-соединениях.
4. HTTP (Hypertext Transfer Protocol):
   * Используется для передачи веб-страниц, изображений и других ресурсов в Интернете.
   * Определяет формат запросов и ответов между клиентом и сервером.
5. SMTP (Simple Mail Transfer Protocol):
   * Используется для отправки и доставки электронной почты.
   * Отвечает за передачу электронных писем между почтовыми серверами.
6. FTP (File Transfer Protocol):
   * Используется для передачи файлов между клиентом и сервером.
   * Обеспечивает функции загрузки, скачивания, удаления и переименования файлов на удаленном сервере.
7. DNS (Domain Name System):
   * Отвечает за преобразование доменных имен (например, [www.example.com](http://www.example.com/)) в IP-адреса.
   * Обеспечивает разрешение имен в сети для обнаружения ресурсов.
8. ICMP (Internet Control Message Protocol):
   * Используется для передачи сообщений об ошибках и управляющих сообщений между сетевыми устройствами.
   * Позволяет сетевым устройствам обмениваться информацией о доступности и состоянии сети.

Это лишь несколько основных протоколов стека TCP/IP, и в нем также есть другие протоколы, выполняющие различные функции для обеспечения коммуникации и передачи данных в компьютерных сетях.

* + 1. IP-адресация. Протоколы. Виды адресов. Специальные IP-адреса.

IP-адресация - это система назначения и идентификации адресов в компьютерных сетях, использующих протокол IP. IP-адрес является числовым идентификатором, который присваивается каждому устройству в сети для их уникальной идентификации и обеспечения маршрутизации данных.

Протоколы, связанные с IP-адресацией, включают:

1. IPv4 (Internet Protocol version 4) - это наиболее распространенная версия протокола IP, которая использует 32-битные адреса. IPv4-адреса записываются в виде четырех десятичных чисел, разделенных точками (например, 192.168.0.1).
2. IPv6 (Internet Protocol version 6) - это новая версия протокола IP, разработанная для замены IPv4. IPv6 использует 128-битные адреса и записывается в виде восьми групп из четырех шестнадцатеричных символов, разделенных двоеточиями (например, 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

Виды IP-адресов:

1. Общедоступные (публичные) IP-адреса - это адреса, которые уникальны и могут быть использованы для общего доступа к сети Интернет. Они присваиваются провайдерами интернет-услуг (ISP) и организациям для подключения к Интернету.
2. Частные IP-адреса - это адреса, которые используются в локальных сетях (например, домашних сетях или внутренних корпоративных сетях) и не маршрутизируются в Интернете. Частные IP-адреса обеспечивают внутреннюю коммуникацию в пределах локальной сети.

Специальные IP-адреса - это адреса, которые имеют специальное назначение и используются для определенных целей в сетевой коммуникации. Некоторые из наиболее распространенных специальных IP-адресов включают:

Адрес петли обратной связи (Loopback):

* IPv4: 127.0.0.1
* IPv6: ::1 Этот адрес используется для тестирования сетевых приложений на самом устройстве. Он обеспечивает возможность отправки и получения данных, не покидая устройства.

Адрес широковещательной сети (Broadcast):

* IPv4: 255.255.255.255
* IPv6: Не используется широковещательная адресация, вместо этого используются групповые адреса. Этот адрес используется для отправки сообщений, которые должны быть получены всеми устройствами в сети.

Адрес маршрутизатора по умолчанию (Default Gateway):

* Это адрес маршрутизатора, который используется для пересылки данных между локальной сетью и внешней сетью (например, Интернетом). В большинстве случаев адрес маршрутизатора по умолчанию обозначается как 0.0.0.0.

Ограниченные адреса для частных сетей:

IPv4:

* - 10.255.255.255 (10.0.0.0/8)
* 172.16.0.0 - 172.31.255.255 (172.16.0.0/12)
* 192.168.0.0 - 192.168.255.255 (192.168.0.0/16)

IPv6:

* Начинаются с префикса fc00::/7 (Unique Local Addresses) Эти адреса предназначены для использования в частных локальных сетях и не маршрутизируются в Интернете.

Адреса мультикастовой передачи:

* IPv4: Начинаются с префикса 224.0.0.0 и распространяются до 239.255.255.255
* IPv6: Начинаются с префикса ff00::/8 Эти адреса используются для групповой передачи данных, когда пакеты отправляются одновременно нескольким получателям.

Это лишь некоторые примеры специальных IP-адресов. В сетевой коммуникации существуют и другие специальные адреса, которые имеют свои определенные цели и функции.

* + 1. Понятие искусственной нейронной сети. Виды нейронных сетей. Их назначение. Этапы построения модели нейронной сети.

Искусственная нейронная сеть (ИНС) - это математическая модель, вдохновленная работой биологической нервной системы, которая используется для обработки информации и решения задач. Она состоит из соединенных и взаимодействующих искусственных нейронов, которые обрабатывают и передают сигналы друг другу.

Виды нейронных сетей:

1. Прямое распространение (Feedforward) нейронные сети: Это самый распространенный тип нейронных сетей. Сигналы передаются от входного слоя к выходному слою без обратной связи. Включает в себя многослойные перцептроны (Multilayer Perceptrons - MLP) и сверточные нейронные сети (Convolutional Neural Networks - CNN).
2. Рекуррентные нейронные сети (Recurrent Neural Networks - RNN): Используют обратную связь, что позволяет им иметь память и обрабатывать последовательности данных. Рекуррентные нейронные сети особенно хороши для анализа текста и временных рядов.
3. Самоорганизующиеся карты Кохонена (Self-Organizing Maps - SOM): Эти нейронные сети обучаются без учителя и используются для кластеризации и визуализации данных. Они помогают обнаружить скрытые структуры и паттерны в данных.
4. Нейронные сети с долгой краткосрочной памятью (Long Short-Term Memory - LSTM): Это разновидность рекуррентных нейронных сетей, способная хранить информацию в течение длительного времени. Они часто применяются для обработки последовательностей событий и предсказания временных рядов.

Нейронные сети могут быть классифицированы по различным критериям. Вот несколько основных видов нейронных сетей:

Прямое распространение (Feedforward) нейронные сети:

* Многослойные перцептроны (Multilayer Perceptrons - MLP): Самый распространенный тип нейронных сетей, состоящий из одного или нескольких скрытых слоев между входным и выходным слоями.
* Сверточные нейронные сети (Convolutional Neural Networks - CNN): Часто используются для обработки изображений и распознавания образов, обладают сверточными слоями, пулингом и полносвязными слоями.

Рекуррентные нейронные сети (Recurrent Neural Networks - RNN):

* Простые рекуррентные сети (Simple RNN): Имеют обратную связь, которая позволяет использовать информацию о предыдущих входах для обработки последовательных данных.
* LSTM (Long Short-Term Memory): Расширение RNN, позволяющее более эффективно улавливать долгосрочные зависимости в последовательных данных.
* GRU (Gated Recurrent Unit): Еще одно расширение RNN, упрощенная версия LSTM с меньшим количеством внутренних компонентов.

Самоорганизующиеся карты Кохонена (Self-Organizing Maps - SOM):

* Используются для кластеризации и визуализации данных, обладают способностью обнаруживать скрытые структуры и паттерны в данных.

Глубокие нейронные сети (Deep Neural Networks - DNN):

* Имеют несколько скрытых слоев, что позволяет им обрабатывать более сложные задачи и извлекать высокоуровневые признаки из данных.
* Включают в себя такие архитектуры, как глубокие многослойные перцептроны (Deep MLP) и глубокие сверточные нейронные сети (Deep CNN).

Генеративно-состязательные сети (Generative Adversarial Networks - GAN):

* Состоят из двух моделей - генератора и дискриминатора, которые соревнуются друг с другом.
* Используются для генерации новых данных, синтеза изображений, текстов и т.д.

Назначение нейронных сетей:

* Классификация: Используется для определения категории или класса объектов на основе входных данных.
* Регрессия: Прогнозирование непрерывных значений на основе входных данных.
* Кластеризация: Группировка объектов на основе их сходства.
* Обработка естественного языка: Работа с текстовыми данными, анализ и синтез речи.
* Распознавание образов: Идентификация и классификация образов, распознавание лиц и объектов.
* Управление и оптимизация: Принятие решений, управление процессами и оптимизация систем.

Этапы построения модели нейронной сети:

* Определение задачи и выбор типа нейронной сети в зависимости от поставленной задачи.
* Сбор и подготовка данных: Сбор данных, их предобработка, масштабирование, разделение на обучающую, валидационную и тестовую выборки.
* Определение архитектуры сети: Определение количества слоев, количества нейронов в каждом слое и типов активационных функций.
* Обучение сети: Использование алгоритма обратного распространения ошибки для обучения весов и биасов сети на обучающей выборке.
* Оценка и тюнинг модели: Оценка производительности модели на валидационной выборке, внесение изменений в архитектуру или параметры модели для улучшения результатов.
* Тестирование модели: Оценка производительности модели на тестовой выборке для проверки ее обобщающей способности.
* Развертывание и использование модели: Применение модели на новых данных для решения поставленной задачи.
  + 1. Современные методы обучения нейронной сети. Обратное распространение ошибки.

Одним из самых распространенных методов обучения нейронных сетей является метод обратного распространения ошибки (Backpropagation). Этот метод позволяет оптимизировать веса и параметры нейронной сети, минимизируя ошибку между предсказанными и ожидаемыми значениями.

Процесс обратного распространения ошибки включает следующие шаги:

1. Прямое распространение (Forward Pass): Входные данные проходят через нейронную сеть, и каждый нейрон вычисляет свой выход на основе входных данных и текущих весов.
2. Вычисление ошибки: Вычисляется ошибка между предсказанными значениями нейронной сети и ожидаемыми значениями целевой переменной.
3. Обратное распространение ошибки (Backward Pass): Ошибка распространяется назад по сети, начиная с выходного слоя. Каждый нейрон вычисляет свой градиент ошибки, указывающий направление, в котором нужно изменить его веса, чтобы уменьшить ошибку.
4. Обновление весов: Веса нейронов обновляются в соответствии с вычисленными градиентами ошибки. Для этого применяется метод оптимизации, такой как градиентный спуск или его вариации.
5. Повторение: Процесс прямого и обратного распространения ошибки повторяется на каждой эпохе обучения до достижения желаемой точности или сходимости модели.

Вместе с обратным распространением ошибки существуют и другие методы обучения нейронных сетей, такие как стохастический градиентный спуск (Stochastic Gradient Descent), алгоритмы оптимизации с моментом (Momentum), адаптивные методы оптимизации (например, Adam, RMSprop) и другие. Эти методы помогают ускорить обучение и повысить его стабильность, особенно при обучении глубоких нейронных сетей.

* + 1. Современные техники глубокого обучения.

Глубокое обучение представляет собой подраздел машинного обучения, который использует искусственные нейронные сети с несколькими слоями для извлечения высокоуровневых признаков из данных. Вот некоторые современные техники глубокого обучения:

1. Сверточные нейронные сети (Convolutional Neural Networks, CNN): Эти сети обычно применяются для анализа и обработки изображений. Они используют сверточные слои для автоматического извлечения локальных признаков изображения и пулинговые слои для уменьшения размерности данных.
2. Рекуррентные нейронные сети (Recurrent Neural Networks, RNN): Эти сети эффективно работают с последовательными данными, такими как тексты и речь. Они имеют обратные связи между нейронами, что позволяет использовать информацию о предыдущих состояниях для анализа текущего ввода.
3. Генеративные модели: Включают в себя сети глубокого обучения, такие как генеративные состязательные сети (Generative Adversarial Networks, GAN) и вариационные автокодировщики (Variational Autoencoders, VAE), которые способны генерировать новые данные на основе обучающего набора.
4. Трансформеры (Transformers): Это архитектура нейронных сетей, которая показала большой успех в области обработки естественного языка и машинного перевода. Они используют механизм внимания для эффективной обработки последовательностей и моделирования долгосрочных зависимостей.
5. Самообучение (Self-Supervised Learning): Это подход, в котором модель обучается на неразмеченных данных для извлечения полезных представлений. Самообучение стало популярным в последние годы и позволяет обучать модели на больших объемах неразмеченных данных.
6. Трансформации и аугментации данных: Это методы, которые изменяют или расширяют исходные данные для улучшения обобщающей способности модели. Примерами могут быть аугментация изображений, генерация синтетических данных или применение случайных трансформаций к данным.
7. Обучение с подкреплением (Reinforcement Learning): Это метод обучения, в котором модель обучается на основе взаимодействия с окружающей средой и получает обратную связь в виде награды или штрафа. Обучение с подкреплением эффективно применяется в областях, где модель должна принимать последовательные решения, например, в играх или управлении роботами.

Это лишь несколько примеров современных техник глубокого обучения, и исследования в этой области продолжаются, внося новые инновации и методы.

* + 1. Оценка качества моделей нейронных сетей.

Оценка качества моделей нейронных сетей является важным шагом в процессе их разработки. Вот некоторые популярные методы оценки качества моделей нейронных сетей:

1. Потери (Loss): Одним из основных показателей качества моделей нейронных сетей являются потери (loss). Потери измеряют разницу между предсказанными значениями модели и фактическими значениями. Чем меньше потери, тем лучше модель.
2. Точность (Accuracy): Точность (accuracy) показывает, насколько хорошо модель предсказывает правильные классы или значения. Она определяется как отношение числа правильно предсказанных примеров к общему числу примеров. Чем выше точность, тем лучше модель.
3. Полнота (Recall) и точность (Precision): При решении задач классификации или детекции объектов важными метриками являются полнота (recall) и точность (precision). Полнота измеряет, какой процент положительных примеров был правильно идентифицирован моделью, а точность измеряет, какой процент из предсказанных положительных примеров является правильными.
4. F1-мера (F1-Score): F1-мера является гармоническим средним между полнотой и точностью. Она предоставляет единый показатель для оценки модели, учитывающий и полноту, и точность.
5. Кривая ROC (Receiver Operating Characteristic): Кривая ROC используется для оценки качества моделей бинарной классификации. Она показывает зависимость между полнотой и ложно положительной ставкой (false positive rate). По форме кривой можно судить о качестве модели.
6. Средняя абсолютная ошибка (Mean Absolute Error, MAE) и среднеквадратичная ошибка (Mean Squared Error, MSE): Эти метрики используются для оценки качества моделей регрессии. MAE измеряет среднюю абсолютную разницу между предсказанными и фактическими значениями, а MSE измеряет среднеквадратичную разницу.
7. Кросс-валидация (Cross-Validation): Кросс-валидация позволяет оценить качество модели на основе разбиения данных на обучающую и валидационную выборки. Это помогает оценить, насколько модель обобщает свои предсказания на новых данных.

Кроме того, существует множество других метрик, которые могут быть применены в зависимости от конкретной задачи и типа модели. Важно выбирать подходящие метрики, учитывая особенности задачи и ожидания от модели.

* + 1. Сверточные нейронные сети.

Сверточные нейронные сети (Convolutional Neural Networks, CNN) являются одним из самых популярных типов нейронных сетей, применяемых в области компьютерного зрения и обработки изображений. Они эффективно работают с данными, имеющими пространственную структуру, такими как изображения.

Основные особенности сверточных нейронных сетей:

1. Сверточные слои: Сверточные слои выполняют операцию свертки между входными данными и фильтрами (ядрами). Это позволяет нейронной сети выделять локальные пространственные паттерны в данных.
2. Пулинг слои: Пулинг слои используются для уменьшения размерности данных и создания инвариантности к масштабу и небольшим трансляциям. Наиболее распространенным методом пулинга является субдискретизация (subsampling) или операция максимального пулинга (max pooling).
3. Нелинейная активация: Сверточные нейронные сети обычно используют нелинейную активацию, такую как ReLU (Rectified Linear Unit), для введения нелинейности и усиления способности модели к моделированию сложных функций.
4. Архитектура слоев: Сверточные нейронные сети обычно имеют несколько сверточных слоев, а затем один или несколько полносвязных слоев для классификации или регрессии.

Сверточные нейронные сети показывают высокую эффективность в задачах распознавания образов, классификации изображений, сегментации, детекции объектов и других задачах компьютерного зрения. Они способны автоматически извлекать иерархические признаки из входных данных, позволяя модели эффективно работать с изображениями и другими пространственными данными.

* 1. **ЭКОНОМИКА ИНФОРМАЦИОННЫХ СИСТЕМ**
     1. Экономические показатели предприятия: показатели прибыльности. Анализ показателей.

Экономические показатели предприятия играют важную роль в оценке его финансового состояния и результативности. Показатели прибыльности являются одним из основных аспектов анализа эффективности деятельности предприятия. Вот несколько ключевых показателей прибыльности и методы их анализа:

1. Валовая прибыль (Gross Profit): Валовая прибыль рассчитывается путем вычета из выручки от продажи себестоимости производства или закупки товаров. Анализ изменений валовой прибыли может указывать на эффективность процесса производства, управления затратами и контроля над ценами.
2. Операционная прибыль (Operating Profit): Операционная прибыль представляет собой разницу между валовой прибылью и операционными расходами. Анализ операционной прибыли позволяет оценить эффективность операционной деятельности предприятия и его способность генерировать прибыль от основной деятельности.
3. Чистая прибыль (Net Profit): Чистая прибыль является финальной прибылью предприятия после учета всех операционных расходов, налогов и прочих затрат. Анализ чистой прибыли позволяет оценить общую финансовую результативность предприятия.
4. Рентабельность продаж (Profit Margin): Рентабельность продаж показывает, сколько прибыли генерируется от каждого рубля выручки. Этот показатель позволяет сравнивать прибыльность разных предприятий или отделов внутри предприятия.
5. Рентабельность активов (Return on Assets, ROA): ROA показывает, какую прибыль генерирует предприятие от использования своих активов. Анализ ROA помогает оценить эффективность использования активов предприятия.
6. Рентабельность собственного капитала (Return on Equity, ROE): ROE показывает, какую прибыль генерируется от каждого рубля собственного капитала предприятия. Анализ ROE позволяет оценить эффективность использования капитала и привлечения инвестиций.

Анализ показателей прибыльности включает сравнение текущих значений с предыдущими периодами, анализ трендов, сравнение с конкурентами в отрасли и установление целей для улучшения прибыльности. Это помогает предприятию определить слабые места, принять меры для повышения прибыли и обеспечить устойчивое финансовое состояние.

* + 1. Экономические показатели предприятия: показатели ликвидности. Анализ показателей.

Экономические показатели ликвидности предприятия отражают его способность вовремя и полностью выполнять текущие обязательства. Анализ показателей ликвидности помогает оценить финансовую устойчивость предприятия и его способность справиться с возможными финансовыми трудностями. Вот несколько ключевых показателей ликвидности и методы их анализа:

1. Коэффициент текущей ликвидности (Current Ratio): Этот показатель выражает соотношение между текущими активами и текущими обязательствами предприятия. Он позволяет оценить способность предприятия погасить текущие обязательства за счет текущих активов. Чем выше коэффициент текущей ликвидности, тем лучше.
2. Коэффициент быстрой ликвидности (Quick Ratio): Этот показатель учитывает только наиболее ликвидные текущие активы, исключая запасы. Он дает представление о способности предприятия погасить текущие обязательства без необходимости продажи запасов. Высокий коэффициент быстрой ликвидности считается положительным.
3. Коэффициент абсолютной ликвидности (Absolute Liquidity Ratio): Этот показатель выражает соотношение между наиболее ликвидными активами (наличные деньги, краткосрочные финансовые вложения) и текущими обязательствами. Он позволяет оценить способность предприятия погасить текущие обязательства исключительно за счет наличных средств.
4. Коэффициент оборачиваемости дебиторской задолженности (Accounts Receivable Turnover Ratio): Этот показатель отражает, как быстро предприятие собирает средства от своих дебиторов. Высокий коэффициент оборачиваемости дебиторской задолженности указывает на эффективное управление кредитами и взыскание задолженности.
5. Коэффициент оборачиваемости запасов (Inventory Turnover Ratio): Этот показатель показывает, как быстро предприятие продает свои запасы. Высокий коэффициент оборачиваемости запасов указывает на эффективное управление запасами и минимизацию неиспользуемых запасов.

Анализ показателей ликвидности позволяет оценить финансовое положение предприятия, его способность встретить текущие обязательства и избежать финансовых проблем. Сравнение показателей с прошлыми периодами и сравнение с аналогичными предприятиями в отрасли помогают выявить тренды и проблемные области, требующие внимания и корректировки.

* + 1. Экономические показатели предприятия: показатели деловой активности. Анализ показателей.

Экономические показатели деловой активности предприятия отражают его эффективность использования активов и ресурсов для генерации выручки и прибыли. Анализ показателей деловой активности помогает оценить эффективность операционной деятельности предприятия и его способность генерировать доходы. Вот несколько ключевых показателей деловой активности и методы их анализа:

1. Оборачиваемость активов (Asset Turnover): Этот показатель показывает, как эффективно предприятие использует свои активы для генерации выручки. Высокий показатель оборачиваемости активов указывает на эффективное использование активов для создания доходов.
2. Оборачиваемость запасов (Inventory Turnover): Этот показатель отражает, как быстро предприятие продает свои запасы. Высокий показатель оборачиваемости запасов указывает на эффективное управление запасами и минимизацию неиспользуемых запасов.
3. Оборачиваемость дебиторской задолженности (Accounts Receivable Turnover): Этот показатель показывает, как быстро предприятие собирает средства от своих дебиторов. Высокий показатель оборачиваемости дебиторской задолженности указывает на эффективное управление кредитами и взыскание задолженности.
4. Оборачиваемость кредиторской задолженности (Accounts Payable Turnover): Этот показатель показывает, насколько быстро предприятие выплачивает свои кредиторам. Высокий показатель оборачиваемости кредиторской задолженности указывает на эффективное управление платежами и использование отсрочки платежей.
5. Коэффициент использования активов (Asset Utilization Ratio): Этот показатель выражает соотношение между выручкой и средствами, вложенными в активы предприятия. Он позволяет оценить эффективность использования активов для генерации выручки.

Анализ показателей деловой активности позволяет оценить эффективность использования активов и ресурсов предприятия, его способность генерировать доходы и обеспечивать рентабельность. Сравнение показателей с прошлыми периодами и сравнение с аналогичными предприятиями в отрасли помогает выявить тренды и проблемные области, требующие внимания и корректировки.

* + 1. Экономические показатели предприятия: показатели финансовой устойчивости. Анализ показателей.

Показатели финансовой устойчивости предприятия отражают его способность удовлетворить свои финансовые обязательства в долгосрочной перспективе и противостоять финансовым рискам. Анализ показателей финансовой устойчивости позволяет оценить финансовое положение предприятия и его способность справиться с возможными финансовыми трудностями. Вот несколько ключевых показателей финансовой устойчивости и методы их анализа:

1. Коэффициент обеспеченности собственными оборотными средствами (Current Ratio): Этот показатель позволяет оценить способность предприятия покрыть свои текущие обязательства за счет собственных оборотных средств. Высокий коэффициент обеспеченности указывает на хорошую финансовую устойчивость.
2. Коэффициент обеспеченности собственными средствами (Equity Ratio): Этот показатель показывает долю собственных средств в общей стоимости активов предприятия. Высокий коэффициент обеспеченности собственными средствами указывает на высокую финансовую устойчивость и независимость от внешних источников финансирования.
3. Коэффициент финансового плеча (Debt-to-Equity Ratio): Этот показатель показывает соотношение между заемными и собственными средствами предприятия. Низкий коэффициент финансового плеча указывает на низкий уровень долговой нагрузки и более стабильную финансовую устойчивость.
4. Показатель текущей задолженности (Current Debt Ratio): Этот показатель показывает долю текущей задолженности в общей стоимости активов предприятия. Низкий показатель указывает на низкий уровень финансового риска и лучшую финансовую устойчивость.
5. Показатель финансовой устойчивости (Financial Stability Index): Этот показатель объединяет несколько финансовых показателей и предоставляет общую оценку финансовой устойчивости предприятия. Он может быть рассчитан с использованием различных методов, включая показатели ликвидности, деловой активности и финансового плеча.

Анализ показателей финансовой устойчивости позволяет оценить финансовую стабильность предприятия и принять меры по улучшению его финансового положения, если необходимо. Он также помогает инвесторам и кредиторам принять решение о предоставлении финансовой поддержки или инвестиций предприятию.

* + 1. Экономические показатели предприятия: положение на рынке ценных бумаг. Анализ показателей.

Анализ показателей положения предприятия на рынке ценных бумаг является важным аспектом финансового анализа и позволяет оценить финансовую привлекательность и конкурентоспособность предприятия для инвесторов. Вот несколько ключевых показателей, которые используются при анализе положения предприятия на рынке ценных бумаг:

1. Цена акций (Stock Price): Анализ изменения цены акций предприятия на протяжении определенного периода времени позволяет оценить интерес инвесторов к акциям предприятия. Рост цены акций может свидетельствовать о хорошем финансовом положении и перспективах предприятия.
2. Дивидендная доходность (Dividend Yield): Этот показатель отражает соотношение между выплаченными дивидендами и текущей ценой акций. Высокая дивидендная доходность может быть привлекательной для инвесторов, так как указывает на возможность получения стабильного дохода.
3. Коэффициент переоценки (Price/Earnings Ratio): Этот показатель отражает соотношение между рыночной ценой акций и прибылью на акцию. Высокий коэффициент переоценки может указывать на высокие ожидания инвесторов относительно будущих прибылей предприятия.
4. Объем торговли акциями (Trading Volume): Анализ объема торговли акциями предприятия позволяет оценить интерес инвесторов к акциям предприятия. Большой объем торговли может указывать на активную торговлю акциями и интерес к инвестициям.
5. Рыночная капитализация (Market Capitalization): Это общая стоимость предприятия на рынке ценных бумаг и рассчитывается как произведение цены акций на общее количество акций в обращении. Высокая рыночная капитализация может свидетельствовать о доверии инвесторов к предприятию и его перспективам.

Анализ этих и других показателей позволяет оценить финансовое положение и привлекательность предприятия для инвесторов на рынке ценных бумаг. Это может помочь принять решение о покупке или продаже акций предприятия.

* + 1. Показатели эффективности инвестиций: Чистый приведённый эффект. Расчёт.

Чистый приведенный эффект (NPV - Net Present Value) является одним из ключевых показателей эффективности инвестиций. Он позволяет оценить разницу между суммой денежных поступлений и затрат в течение жизненного цикла инвестиционного проекта, приведенную к текущему моменту времени с учетом стоимости денег во времени.

Формула для расчета NPV выглядит следующим образом:

NPV = Σ(CF\_t / (1 + r)^t) - Initial Investment

где:

* CF\_t - денежные поступления в период t,
* r - дисконтированная ставка (ставка дисконта),
* t - период времени,
* Initial Investment - начальные инвестиционные затраты.

Шаги для расчета NPV:

1. Определите денежные поступления (CF\_t): для каждого периода времени в проекте определите ожидаемые денежные поступления после вычета всех затрат.
2. Определите дисконтированную ставку (r): это ставка, которую вы используете для дисконтирования денежных потоков. Она отражает стоимость капитала или ожидаемую доходность вложений. Обычно она выражается в виде процента.
3. Определите начальные инвестиционные затраты (Initial Investment): это сумма денег, которую необходимо вложить в проект в начале его реализации.
4. Примените формулу NPV: для каждого периода времени примените формулу NPV, дисконтируя денежные поступления на соответствующую ставку дисконта и складывая все значения. Затем вычтите начальные инвестиционные затраты.
5. Интерпретация результатов: если NPV положительный, то проект считается прибыльным и эффективным, поскольку он принесет положительную чистую прибыль. Если NPV отрицательный, то проект нецелесообразен, поскольку он приведет к отрицательной чистой прибыли.

Расчет NPV позволяет оценить степень доходности инвестиций и сравнить различные проекты или варианты инвестиций для принятия обоснованного решения.

* + 1. Показатели эффективности инвестиций: Внутренняя норма прибыли инвестиции. Расчёт.

Внутренняя норма прибыли (IRR - Internal Rate of Return) является показателем эффективности инвестиций. Она определяет дисконтированную ставку доходности, при которой чистый приведенный эффект (NPV) инвестиций равен нулю. IRR показывает ожидаемую доходность инвестиций и является дисконтированной ставкой, при которой сумма дисконтированных денежных поступлений равна сумме дисконтированных затрат.

NPV = Σ(CF\_t / (1 + r)^t) - Initial Investment

где:

* CF\_t - денежные поступления в период t,
* r - дисконтированная ставка (ставка дисконта),
* t - период времени,
* Initial Investment - начальные инвестиционные затраты.

Для расчета IRR используется следующий метод:

1. Определите денежные поступления и затраты по проекту на каждый период времени.
2. Предположим, что у вас есть оценка или предположение о дисконтированной ставке (начните с какого-то разумного значения, например, 10%).
3. Примените формулу NPV, описанную ранее, для расчета NPV с использованием предполагаемой ставки дисконта.
4. Если NPV положительное, увеличьте ставку дисконта, чтобы получить новое значение NPV. Если NPV отрицательное, уменьшите ставку дисконта.
5. Повторяйте шаги 3 и 4, изменяя ставку дисконта, пока NPV не станет равным нулю или будет достаточно близким к нулю. Это значение ставки дисконта будет являться внутренней нормой прибыли (IRR).

Результатом расчета IRR будет процентная ставка, которая показывает ожидаемую доходность инвестиций. Если IRR превышает требуемую ставку дисконта или ожидаемую норму доходности, то инвестиции считаются эффективными.

Интерпретация результатов:

* Если IRR больше требуемой ставки дисконта, то инвестиции считаются прибыльными и эффективными.
* Если IRR меньше требуемой ставки дисконта, то инвестиции считаются неэффективными или неприбыльными.

Важно отметить, что расчет IRR может быть сложным и требует использования численных методов или программного обеспечения для достижения точного значения.

* + 1. Показатели эффективности инвестиций: Индекс рентабельности инвестиции. Расчёт.

Индекс рентабельности инвестиции (PI - Profitability Index), также известный как коэффициент рентабельности или коэффициент эффективности, является одним из показателей эффективности инвестиций. Он измеряет отношение приведенной стоимости денежных поступлений (PV - Present Value) к приведенной стоимости затрат (PC - Present Cost) по инвестиционному проекту.

Для расчета индекса рентабельности используется следующая формула:

PI = PV / PC

где:

* PV - приведенная стоимость денежных поступлений (сумма дисконтированных денежных потоков)
* PC - приведенная стоимость затрат (сумма дисконтированных затрат)

Расчет индекса рентабельности предполагает дисконтирование денежных потоков по проекту, то есть приведение их стоимости к текущему моменту времени с использованием определенной ставки дисконта. Дисконтирование позволяет учесть временную стоимость денег и сравнить стоимость затрат с ожидаемыми денежными поступлениями.

Интерпретация результатов:

* Если PI больше 1, то инвестиция считается эффективной и прибыльной. Чем выше значение PI, тем более выгодной является инвестиция.
* Если PI равно 1, то инвестиция не приносит дополнительной прибыли, но также не вызывает убытков. Это может рассматриваться как граница между прибыльными и не прибыльными инвестициями.
* Если PI меньше 1, то инвестиция считается неприбыльной и неэффективной.

Расчет индекса рентабельности позволяет оценить эффективность инвестиционного проекта, учитывая денежные потоки и их приведенные стоимости. Однако, важно также учитывать другие факторы, такие как риски, время окупаемости и стратегическую значимость проекта при принятии решения об инвестировании.

* + 1. Показатели эффективности инвестиций: Срок окупаемости и дисконтированный срок окупаемости. Расчёт.

Срок окупаемости (Payback Period) и дисконтированный срок окупаемости (Discounted Payback Period) являются показателями эффективности инвестиций и используются для определения времени, за которое инвестиция вернет свои затраты.

*Срок окупаемости (Payback Period):*

Срок окупаемости выражает время, необходимое для возврата первоначальных инвестиций. Он рассчитывается как сумма периодов времени, за которые кумулятивные денежные поступления становятся равными или превышают первоначальные затраты.

Расчет срока окупаемости выполняется по следующей формуле: Payback Period = Инвестиции / Годовая чистая прибыль

Годовая чистая прибыль определяется как разница между годовыми денежными поступлениями и годовыми затратами.

*Дисконтированный срок окупаемости (Discounted Payback Period):*

Дисконтированный срок окупаемости учитывает временную стоимость денег и приводит денежные потоки к их приведенным стоимостям, используя определенную ставку дисконта. Этот показатель позволяет более точно оценить эффективность инвестиции, учитывая дисконтирование денежных потоков.

Расчет дисконтированного срока окупаемости выполняется следующим образом: Discounted Payback Period = Накопленные дисконтированные денежные потоки / Дисконтированный годовой денежный поток

Дисконтированный годовой денежный поток определяется как разница между дисконтированными годовыми денежными поступлениями и дисконтированными годовыми затратами.

В обоих случаях чем меньше срок окупаемости, тем более эффективной считается инвестиция. Однако, дисконтированный срок окупаемости учитывает временную стоимость денег и предпочтительнее использовать при оценке проектов с долгим сроком реализации или при сравнении проектов с разными денежными потоками во времени.

Важно отметить, что срок окупаемости и дисконтированный срок окупаемости являются лишь одними из множества показателей эффективности инвестиций, и при принятии решения об инвестировании следует учитывать и другие факторы, такие как риски, стоимость капитала, стратегическая значимость проекта и т. д.

* + 1. Финансовые модели. Примеры. Анализ и управление финансовыми показателями.

Финансовые модели являются инструментами для анализа и прогнозирования финансовых показателей предприятия. Они помогают принимать решения, оптимизировать финансовые процессы и планировать будущие действия. Вот некоторые примеры финансовых моделей:

1. Прогнозирование доходов и расходов: Эта модель используется для прогнозирования будущих доходов и расходов предприятия на основе исторических данных и других факторов. Она позволяет оценить финансовую устойчивость предприятия и определить возможные потенциальные риски.
2. Модель оценки инвестиций: Эта модель помогает оценить финансовую целесообразность инвестиционных проектов. Она учитывает затраты на инвестиции, потенциальные доходы, сроки окупаемости и другие факторы, чтобы принять решение о том, стоит ли инвестировать в проект или нет.
3. Модель управления рабочим капиталом: Эта модель помогает оптимизировать использование рабочего капитала предприятия, включая запасы, дебиторскую и кредиторскую задолженность. Она позволяет определить оптимальные уровни запасов, сроки оплаты поставщикам и условия кредитования для эффективного управления денежными потоками.
4. Модель оценки стоимости акций: Эта модель используется для оценки стоимости акций компании на основе финансовых показателей, таких как прибыль, рост и дивиденды. Она помогает инвесторам принять решение о покупке или продаже акций.

Анализ и управление финансовыми показателями включает в себя мониторинг и оценку ключевых финансовых показателей, таких как прибыльность, ликвидность, финансовая устойчивость и рентабельность инвестиций. Он также включает в себя принятие мер для улучшения финансового положения предприятия, таких как оптимизация расходов, управление дебиторской и кредиторской задолженностью, планирование инвестиций и прочие финансовые стратегии.

Для анализа и управления финансовыми показателями могут использоваться различные методы и инструменты, включая финансовые отчеты, показатели анализа, бюджетирование, сценарный анализ и другие.

* + 1. Финансовые модели. Примеры. Анализ и управление показателями инвестиций (анализ чувствительности).

Анализ чувствительности является важной частью анализа и управления показателями инвестиций. Он позволяет оценить, насколько изменения в различных факторах могут повлиять на результаты инвестиционного проекта. В результате анализа чувствительности можно определить, какие факторы являются наиболее критическими и требуют особого внимания при принятии решений. Вот некоторые методы анализа чувствительности, которые могут применяться в финансовых моделях:

1. Изменение ставок дисконтирования: При анализе чувствительности можно изменять ставку дисконтирования, которая используется для приведения будущих денежных потоков к их текущей стоимости. Изменение ставки дисконтирования может значительно влиять на оценку стоимости инвестиции и ее рентабельность.
2. Изменение объемов продаж: Анализ чувствительности может включать изменение объемов продаж или спроса на продукцию или услугу, которую предполагается производить или предоставлять с помощью инвестиции. Это позволяет оценить, насколько чувствительны результаты проекта к изменениям в объемах продаж.
3. Изменение стоимости сырья или материалов: Если инвестиционный проект связан с использованием определенных сырьевых материалов, анализ чувствительности может включать изменение стоимости этих материалов. Это позволяет оценить влияние колебаний цен на себестоимость продукции и прибыльность проекта.
4. Изменение операционных расходов: Анализ чувствительности может включать изменение операционных расходов, таких как затраты на персонал, аренду помещений, энергозатраты и другие переменные расходы. Это позволяет оценить влияние изменения расходов на финансовые показатели проекта.
5. Изменение налоговой ставки: Если налоговые платежи составляют значительную часть затрат или влияют на доходы проекта, анализ чувствительности может включать изменение налоговой ставки. Это позволяет оценить влияние изменения налогов на финансовую результативность проекта.

Анализ чувствительности может проводиться с помощью финансовых моделей, включая дисконтированный денежный поток, анализ чистой приведенной стоимости, моделирование сценариев и другие. При этом учитываются различные факторы и их влияние на финансовые показатели инвестиций, что помогает принять более информированные решения.

1. **ЗАДАЧИ, ВЫНОСИМЫЕ НА ГОСУДАРСТВЕННЫЙ ЭКЗАМЕН**

|  |  |
| --- | --- |
| **ДАНО:** | Текстовый файл с некоторой информацией. |
| **ТРЕБУЕТСЯ:** | 1. Создать новую пару ключей RSA длиной 512 бит и сохранить в одном файле открытый и секретный ключ, а в другом – только открытый ключ. 2. Создать новый ключ симметричного шифрования по алгоритму DES. 3. Зашифровать с помощью ключа симметричного шифрования текст, содержащийся в файле. 4. Зашифровать симметричный ключ DES с помощью открытого ключа RSA. 5. Сохранить зашифрованный ключ DES и зашифрованную с помощью него информацию в новом бинарном файле. |
| **ОГРАНИЧЕНИЯ:** | 1. Пункт 1 выполнить в отдельном программном модуле. 2. Пункты 2,3,4,5 выполнить вместе в отдельном программном модуле. 3. Язык программирования – Visual Basic .Net или C # (по выбору) |

1 пункт:

using System;

using System.IO;

using System.Security.Cryptography;

class Program

{

static void Main()

{

RSACryptoServiceProvider RSA = new RSACryptoServiceProvider(512);

// Записать информацию об открытом и секретном ключах в файл

FileStream fs = new FileStream("key1.bin", FileMode.OpenOrCreate);

StreamWriter w = new StreamWriter(fs);

w.Write(RSA.ToXmlString(true));

w.Flush();

fs.Close();

// Записать информацию о секретном ключе в другой файл

fs = new FileStream("key2.bin", FileMode.OpenOrCreate);

w = new StreamWriter(fs);

w.Write(RSA.ToXmlString(false));

w.Flush();

fs.Close();

}

}

2, 3, 4, 5 пункт:

using System;

using System.IO;

using System.Security.Cryptography;

class Program

{

static void Main()

{

FileStream fs = new FileStream("Текст.txt", FileMode.Open);

StreamReader r = new StreamReader(fs);

RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();

RSA.FromXmlString(r.ReadToEnd());

fs.Close();

// Открытие шифрованного файла для записи

fs = new FileStream("Шифр.bin", FileMode.Create);

// Создание нового симметричного ключа для шифрования по алгоритму DES

DESCryptoServiceProvider DES = new DESCryptoServiceProvider();

// Шифрование симметричного ключа и вектора инициализации

// с помощью открытого ключа RSA

byte[] EncrypKey = RSA.Encrypt(DES.Key, false);

byte[] EncrypIV = RSA.Encrypt(DES.IV, false);

// Запись асимметрично зашифрованных ключа и вектора инициализации в файл

fs.Write(EncrypKey, 0, EncrypKey.Length);

fs.Write(EncrypIV, 0, EncrypIV.Length);

// Шифрование информации с помощью алгоритма DES и запись её в файл

ICryptoTransform transform = DES.CreateEncryptor();

CryptoStream cs = new CryptoStream(fs, transform, CryptoStreamMode.Write);

StreamWriter w = new StreamWriter(cs);

w.Write(RichTextBox1.Text);

w.Flush();

// Очистка буфера памяти и закрытие файла

cs.FlushFinalBlock();

w.Close();

fs.Close();

}

}

|  |  |
| --- | --- |
| **ДАНО:** | Текстовый файл с некоторой информацией. |
| **ТРЕБУЕТСЯ:** | 1. Создать новую пару ключей RSA длиной 512 бит и сохранить в одном файле открытый и секретный ключ, а в другом – только открытый ключ. 2. Создать цифровую подпись RSA для данного текста и поместить её в отдельный файл. В качестве алгоритма хеширования использовать SHA-1. 3. Проверить истинность цифровой подписи. |
| **ОГРАНИЧЕНИЯ:** | 1. Каждый пункт выполнить в отдельном программном модуле. 2. Язык программирования – Visual Basic .Net или C # (по выбору) |

1:

using System;

using System.IO;

using System.Security.Cryptography;

class Program

{

static void Main()

{

RSACryptoServiceProvider RSA = new RSACryptoServiceProvider(512);

// Записать информацию об открытом и секретном ключах в файл

using (FileStream fs = new FileStream("key1.bin", FileMode.OpenOrCreate))

using (StreamWriter w = new StreamWriter(fs))

{

w.Write(RSA.ToXmlString(true));

// Очистка буфера памяти

w.Flush();

}

// Записать информацию о секретном ключе в другой файл

using (FileStream fs = new FileStream("key2.bin", FileMode.OpenOrCreate))

using (StreamWriter w = new StreamWriter(fs))

{

w.Write(RSA.ToXmlString(false));

// Очистка буфера памяти

w.Flush();

}

}

}

2:

using System;

using System.IO;

using System.Security.Cryptography;

using System.Text;

class Program

{

static void Main()

{

string text;

byte[] signature;

// Открытие исходного файла и чтение информации в переменную text

using (FileStream fs = new FileStream("Пример3.txt", FileMode.Open))

using (StreamReader r = new StreamReader(fs))

{

text = r.ReadToEnd();

}

// Прочитать информацию о секретном ключе

RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();

using (FileStream fs = new FileStream("key2.bin", FileMode.Open))

using (StreamReader r = new StreamReader(fs))

{

RSA.FromXmlString(r.ReadToEnd());

}

// Преобразование данных в поток байт

byte[] dataBytes = Encoding.UTF8.GetBytes(text);

// Подписывание данных

using (SHA1CryptoServiceProvider sha1 = new SHA1CryptoServiceProvider())

{

signature = RSA.SignData(dataBytes, sha1);

}

// Запись подписи в файл

using (FileStream fs = new FileStream("Подпись.bin", FileMode.OpenOrCreate))

{

fs.Write(signature, 0, signature.Length);

}

}

}

3:

using System;

using System.IO;

using System.Security.Cryptography;

using System.Text;

class Program

{

static void Main()

{

string text;

byte[] signature = new byte[128]; // Здесь предполагается, что подпись имеет фиксированный размер 128 байт

// Открытие исходного файла и чтение информации в переменную text

using (FileStream fs = new FileStream("Текст.txt", FileMode.Open))

using (StreamReader r = new StreamReader(fs))

{

text = r.ReadToEnd();

}

// Прочитать информацию об открытом ключе

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

using (FileStream fs = new FileStream("key1.bin", FileMode.Open))

using (StreamReader r = new StreamReader(fs))

{

rsa.FromXmlString(r.ReadToEnd());

}

// Преобразование данных в поток байт

byte[] dataBytes = Encoding.UTF8.GetBytes(text);

// Прочитать информацию о подписи

using (FileStream fs = new FileStream("Подпись.bin", FileMode.Open))

{

fs.Read(signature, 0, signature.Length);

}

// Верификация подписи

using (SHA1CryptoServiceProvider sha1 = new SHA1CryptoServiceProvider())

{

if (rsa.VerifyData(dataBytes, sha1, signature))

{

Console.WriteLine("Подпись верна");

}

else

{

Console.WriteLine("Подпись не верна");

}

}

}

}

|  |  |
| --- | --- |
| **ДАНО:** | Шесть графических файлов формата .JPG, созданных в программе Paint, в каждом из которых нарисован квадрат следующего цвета: белый, синий, желтый, оранжевый, красный, зелёный. |
| **ТРЕБУЕТСЯ:** | 1. Создать программу, рисующую на форме куб с гранями соответствующих цветов, созданных на основе имеющихся графических файлов. 2. Обеспечить поворот данного куба вверх, вниз, вправо и влево с помощью соответствующих кнопок, нанесенных на форму (написать обработчики событий каждой из кнопок). |
| **ОГРАНИЧЕНИЯ:** | 1. Исходное положение куба – фасад – белый, верх – синий, правая сторона – красный, левая сторона – желтый, низ – зеленый, задняя сторона – оранжевый. 2. В исходном положение видны фасад, верх и правая сторона. 3. Обеспечить отображение изображения при возможных манипуляциях с формой (изменение размеров, свертывание и др.) 4. Язык программирования – Visual Basic .Net или C # (по выбору) |

using System;

using System.Drawing;

using System.Drawing.Drawing2D;

using System.Windows.Forms;

public class Form1 : Form

{

public Image img\_b, img\_w, img\_r, img\_o, img\_y, img\_g;

public Graphics g;

public int[] mas = { 1, 2, 3, 4, 5, 6 };

public int[] mas\_t = new int[6];

private void Form1\_Paint(object sender, PaintEventArgs e)

{

Point[] f = { new Point(170, 210), new Point(370, 210), new Point(170, 410) };

Point[] t = { new Point(230, 150), new Point(430, 150), new Point(170, 210) };

Point[] r = { new Point(370, 210), new Point(430, 150), new Point(370, 410) };

img\_b = Image.FromFile("picture\\b.jpg");

img\_w = Image.FromFile("picture\\w.jpg");

img\_r = Image.FromFile("picture\\r.jpg");

img\_o = Image.FromFile("picture\\o.jpg");

img\_y = Image.FromFile("picture\\y.jpg");

img\_g = Image.FromFile("picture\\g.jpg");

g = CreateGraphics();

switch (mas[0])

{

case 1:

g.DrawImage(img\_w, f);

break;

case 2:

g.DrawImage(img\_r, f);

break;

case 3:

g.DrawImage(img\_o, f);

break;

case 4:

g.DrawImage(img\_g, f);

break;

case 5:

g.DrawImage(img\_b, f);

break;

case 6:

g.DrawImage(img\_y, f);

break;

}

switch (mas[1])

{

case 1:

g.DrawImage(img\_w, r);

break;

case 2:

g.DrawImage(img\_r, r);

break;

case 3:

g.DrawImage(img\_o, r);

break;

case 4:

g.DrawImage(img\_g, r);

break;

case 5:

g.DrawImage(img\_b, r);

break;

case 6:

g.DrawImage(img\_y, r);

break;

}

switch (mas[4])

{

case 1:

g.DrawImage(img\_w, t);

break;

case 2:

g.DrawImage(img\_r, t);

break;

case 3:

g.DrawImage(img\_o, t);

break;

case 4:

g.DrawImage(img\_g, t);

break;

case 5:

g.DrawImage(img\_b, t);

break;

case 6:

g.DrawImage(img\_y, t);

break;

}

}

private void Button3\_Click(object sender, EventArgs e)

{

// кнопка вправо

mas\_t[0] = mas[3];

mas\_t[1] = mas[0];

mas\_t[2] = mas[1];

mas\_t[3] = mas[2];

mas\_t[4] = mas[4];

mas\_t[5] = mas[5];

Array.Copy(mas\_t, mas, mas.Length);

Refresh();

}

private void Button2\_Click(object sender, EventArgs e)

{

// кнопка влево

mas\_t[0] = mas[1];

mas\_t[1] = mas[2];

mas\_t[2] = mas[3];

mas\_t[3] = mas[0];

mas\_t[4] = mas[4];

mas\_t[5] = mas[5];

Array.Copy(mas\_t, mas, mas.Length);

Refresh();

}

private void Button1\_Click(object sender, EventArgs e)

{

// кнопка вверх

mas\_t[0] = mas[5];

mas\_t[1] = mas[1];

mas\_t[2] = mas[4];

mas\_t[3] = mas[3];

mas\_t[4] = mas[0];

mas\_t[5] = mas[2];

Array.Copy(mas\_t, mas, mas.Length);

Refresh();

}

private void Button4\_Click(object sender, EventArgs e)

{

// кнопка вниз

mas\_t[0] = mas[4];

mas\_t[1] = mas[1];

mas\_t[2] = mas[5];

mas\_t[3] = mas[3];

mas\_t[4] = mas[2];

mas\_t[5] = mas[0];

Array.Copy(mas\_t, mas, mas.Length);

Refresh();

}

public Form1()

{

InitializeComponent();

}

private void InitializeComponent()

{

this.SuspendLayout();

//

// Form1

//

this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.ClientSize = new System.Drawing.Size(800, 450);

this.Name = "Form1";

this.Text = "Form1";

this.Paint += new System.Windows.Forms.PaintEventHandler(this.Form1\_Paint);

this.ResumeLayout(false);

Button button1 = new Button();

button1.Text = "Вверх";

button1.Click += new EventHandler(Button1\_Click);

button1.Location = new Point(10, 10);

Controls.Add(button1);

Button button2 = new Button();

button2.Text = "Влево";

button2.Click += new EventHandler(Button2\_Click);

button2.Location = new Point(10, 40);

Controls.Add(button2);

Button button3 = new Button();

button3.Text = "Вправо";

button3.Click += new EventHandler(Button3\_Click);

button3.Location = new Point(10, 70);

Controls.Add(button3);

Button button4 = new Button();

button4.Text = "Вниз";

button4.Click += new EventHandler(Button4\_Click);

button4.Location = new Point(10, 100);

Controls.Add(button4);

}

static void Main()

{

Application.Run(new Form1());

}

}

|  |  |
| --- | --- |
| **ТРЕБУЕТСЯ:** | 1. Создать программу, рисующую на форме круг, разделенный на 20 секторов по 30 градусов каждый. Каждый сектор закрасить в свой цвет. 2. Обеспечить динамическую смену цветов (вращение по часовой стрелке) данного круга. |
| **ОГРАНИЧЕНИЯ:** | 1. Обеспечить отображение изображения при возможных манипуляциях с формой (изменение размеров, свертывание и др.) 2. Язык программирования – Visual Basic .Net или C # (по выбору) |

Для добавления элемента Timer на форму и обработки события Timer1\_Tick в коде на C#, вам потребуется выполнить следующие шаги:

Добавьте элемент Timer на вашу форму. Вы можете сделать это в дизайнере форм, перетащив элемент Timer из панели инструментов на форму.

Объявите переменную в классе формы:

public int n\_color = 11;

В обработчике события Timer1\_Tick добавьте следующий код:

private void Timer1\_Tick(object sender, EventArgs e)

{

Brush[] pointBrushes = new Brush[]

{

new SolidBrush(Color.Цвет1),

new SolidBrush(Color.Цвет2),

new SolidBrush(Color.Цвет3),

new SolidBrush(Color.Цвет4),

new SolidBrush(Color.Цвет5),

new SolidBrush(Color.Цвет6),

new SolidBrush(Color.Цвет7),

new SolidBrush(Color.Цвет8),

new SolidBrush(Color.Цвет9),

new SolidBrush(Color.Цвет10),

new SolidBrush(Color.Цвет11),

new SolidBrush(Color.Цвет12)

};

Graphics g = CreateGraphics();

for (int i = 0; i < 12; i++)

{

int k = (i + n\_color) % 12;

g.FillPie(pointBrushes[k], 50, 50, 200, 200, i \* 30, 30);

}

n\_color--;

if (n\_color == -1)

n\_color = 11;

}

В методе InitializeComponent() добавьте следующий код для инициализации Timer и установки интервала времени:

private void InitializeComponent()

{

// ... остальной код ...

Timer timer1 = new Timer();

timer1.Tick += new EventHandler(Timer1\_Tick);

timer1.Interval = 1000; // Установите интервал в миллисекундах, например, 1000 для одной секунды.

timer1.Start();

}

Обратите внимание, что вам может потребоваться настроить интервал Timer (timer1.Interval) в зависимости от вашего требования. В приведенном примере интервал установлен на 1000 миллисекунд (1 секунда).



|  |  |
| --- | --- |
| **ДАНО:** | Набор графических файлов формата .JPG, содержащихся в некоторой папке на компьютере. |
| **ТРЕБУЕТСЯ:** | 1. Создать программу, реализующую функции слайд-шоу из файлов, содержащихся в данной папке c помощью элемента PictureBox. |
| **ОГРАНИЧЕНИЯ:** | 1. Программа должна обеспечивать функции выбора папки, выбора скорости показа с помощью элемента ComboBox, оптимального подбора размеров PictureBox. 2. Программа должна предусматривать наличие кнопок «Старт» и «Стоп», по которым начинается и останавливается показ файлов. Эти кнопки не должны быть доступными одновременно. 3. Программа должна предусматривать наличие кнопок «Вперед» и «Назад», с помощью которых должна быть обеспечена возможность просмотра файлов по одному. В режиме слайд-шоу эти кнопки не доступны. По достижении первого и последнего файла соответствующие кнопки также должны быть недоступны. 4. Язык программирования – Visual Basic .Net или C # (по выбору) |

На форму необходимо добавит следующие элементы:

* PictureBox
* Timer
* FolderBrowserDialog
* Кнопки «Выбор папки», «Начать показ», «Завершить показ», «Вперед», «Назад»
* ComboBox

Обработчик события нажатия кнопки «Выбор папки»:

DialogResult result = folderBrowserDialog1.ShowDialog();

if (result == DialogResult.OK)

{

string[] files = Directory.GetFiles(folderBrowserDialog1.SelectedPath, "\*.jpg");

int imgCount = files.Length;

if (imgCount == 0)

{

MessageBox.Show("В данной папке нет графических файлов", "Предупреждение", MessageBoxButtons.OK);

}

}

Выбор скорости показа с помощью элемента Combobox. Обработчик события SelectedIndexChanged:

string n = comboBox1.Text;

switch (n)

{

case "10":

timer1.Interval = 10;

break;

case "50":

timer1.Interval = 50;

break;

case "150":

timer1.Interval = 150;

break;

case "500":

timer1.Interval = 500;

break;

case "1000":

timer1.Interval = 1000;

break;

}

Обработчик события нажатия кнопки «Начать показ»:

if (Img > Img\_Count)

{

Img = 0;

}

button\_next.Enabled = true; // Кнопка вперед

button\_prev.Enabled = true; // Кнопка назад

timer1.Start();

Обработчик события нажатия кнопки «Завершить показ»:

timer1.Stop();

if (Img < Img\_Count && Img != 0)

{

button4.Enabled = true;

button5.Enabled = true;

}

Обработчик события нажатия кнопки «Вперед»:

if (Img == Img\_Count)

{

button4.Enabled = false;

}

Img = Img + 1;

button5.Enabled = true;

if (Img < Img\_Count)

{

pictureBox1.Image = Image.FromFile(Files[Img]);

}

ResizeImage();

Обработчик события нажатия кнопки «Назад»:

if (Img > Img\_Count)

{

Img = Img - 2;

}

else

{

Img = Img - 1;

}

button4.Enabled = true;

if (Img == 0)

{

button5.Enabled = false;

}

pictureBox1.Image = Image.FromFile(Files[Img]);

ResizeImage();



|  |  |
| --- | --- |
| **ДАНО:** | Два числа типа Single. |
| **ТРЕБУЕТСЯ:** | 1. Создать программу, вычисляющую сумму данных чисел, их разность, произведение и частное от деления одного числа на другое, результат целочисленного деления одного числа на другое и результат деления по модулю одного числа на другое. |
| **ОГРАНИЧЕНИЯ:** | 1. Числа вводятся пользователем в два элемента TextBox. 2. Вид математической операции выбирается пользователем с помощью элемента ComboBox. 3. Обеспечить контроль вводимых данных, позволяя пользователю вводить только цифры и одну десятичную точку. 4. Обеспечить правильность отображения результата в случае ввода некорректных данных (для операций деления, целочисленного деления, деления по модулю). 5. Язык программирования – Visual Basic .Net или C # (по выбору) |

Для вычисления в обработчик события нажатия кнопки «Вычислить» записать:

TextBox3.Enabled = true;

if (ComboBox1.Text == "/" && TextBox2.Text == "0")

{

TextBox3.Text = "На 0 делить нельзя";

}

else if (ComboBox1.Text == "/")

{

TextBox3.Text = (double.Parse(TextBox1.Text) / double.Parse(TextBox2.Text)).ToString();

}

if (ComboBox1.Text == "\\" && TextBox2.Text == "0")

{

TextBox3.Text = "На 0 делить нельзя";

}

else if (ComboBox1.Text == "\\")

{

TextBox3.Text = (int.Parse(TextBox1.Text) / int.Parse(TextBox2.Text)).ToString();

}

if (ComboBox1.Text == "+")

{

TextBox3.Text = (int.Parse(TextBox1.Text) + int.Parse(TextBox2.Text)).ToString();

}

if (ComboBox1.Text == "-")

{

TextBox3.Text = (int.Parse(TextBox1.Text) - int.Parse(TextBox2.Text)).ToString();

}

if (ComboBox1.Text == "\*")

{

TextBox3.Text = (int.Parse(TextBox1.Text) \* int.Parse(TextBox2.Text)).ToString();

}

if (ComboBox1.Text == "mod")

{

TextBox3.Text = (int.Parse(TextBox1.Text) % int.Parse(TextBox2.Text)).ToString();

}

Контроль вводимых данных

char ch = e.KeyChar;

string d = "-0123456789.";

if (char.IsControl(ch))

{

return;

}

if (d.IndexOf(ch) == -1)

{

e.Handled = true;

}

if (TextBox1.Text.Contains(".") && ch == '.')

{

e.Handled = true;

}

Этот код проверяет вводимые символы и обрабатывает событие KeyPress. Он использует переменные ch для хранения вводимого символа и d для допустимых символов. Затем он проверяет, является ли символ управляющим символом с помощью метода char.IsControl. Если символ не является управляющим символом и не содержится в списке допустимых символов d, то e.Handled устанавливается в true, чтобы предотвратить его ввод. Также проверяется наличие точки в тексте TextBox1 и предотвращается ввод дополнительных точек.

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить целостность базы данных (целостность таблиц, целостность столбцов и бизнес-правила). | |
| **ОГРАНИЧЕНИЯ:** | | 1. Бизнес-правила обеспечить с помощью объектов **CONSTRAINT** (предусмотрев, в частности, идентичность кодирования всех данных). 2. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

([KodIzdatelya])

REFERENCES [dbo].[Izdateli] ([KodIzdatelya] )

CONSTRAINT сk\_god **CHECK**(DataVihoda <= ‘sysdate’)

)

CREATE TABLE [dbo].[Izdateli] (

[KodIzdatelya] [bigint] NOT NULL ,

[ImyaIzdatelya] [char] (10) NOT NULL,

[AdreaIzdatelya] [char] (10) NOT NULL,

CONSTRAINT [PK\_Izdately] PRIMARY KEY

(

[KodIzdatelya]

),

CONSTRAINT сk\_adres **CHECK**(AdreaIzdatelya in (‘Moscow’, ‘London’))

)

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить целостность базы данных (целостность таблиц, целостность столбцов и декларативную целостность ссылок). | |
| **ОГРАНИЧЕНИЯ:** | | 1. Декларативную целостность ссылок обеспечить с помощью объектов **CONSTRAINT.** 2. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

***SQL запросы:***

*Издатели:*

CREATE TABLE [dbo].[Izdateli] (

[KodIzdatelya] [bigint] NOT NULL ,

[ImyaIzdatelya] [char] (10) NOT NULL,

[AdreaIzdatelya] [char] (10) NOT NULL,

CONSTRAINT [PK\_Izdately] PRIMARY KEY

(

[KodIzdatelya]

),

CONSTRAINT [u\_Izdateli] UNIQUE

(

[KodIzdatelya]

)

)

*Авторы:*

CREATE TABLE [dbo].[ [Avtory] (

[KodAvtora] [bigint] NOT NULL ,

[Familia] [char] (10) NOT NULL,

[DataRojdenya] [datetime] NOT NULL,

[MastoRojdenya] [char] (10)  ,

CONSTRAINT [PK\_avtory] PRIMARY KEY

(

[KodAvtora]

)

)

*Книги:*

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

(

[KodIzdatelya]

) REFERENCES [dbo.][Izdateli] ([KodIzdatelya] )

)

*Авторкниги:*

CREATE TABLE [dbo].[ [AvtorKnigi] (

[KodAvtora] [bigint] NOT NULL ,

[KodKnigi] [bigint] NOT NULL ,

CONSTRAINT [PK\_AvtorKnigi] PRIMARY KEY

(

[KodAvtora],

[KodKnigi]

)   ,

CONSTRAINT [FK\_AvtorKnigi\_Avtory] FOREIGN KEY

(

[KodAvtora]

) REFERENCES [dbo.][Avtory] ([KodAvtora] ),

CONSTRAINT [FK\_AvtorKnigi\_Knigi] FOREIGN KEY

(

[KodKnigi]

) REFERENCES [dbo.][Knigi] ([KodKnigi] )

)

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить процедурную целостность ссылок. | |
| **ОГРАНИЧЕНИЯ:** | | 1. Целостность ссылок обеспечить только с помощью триггеров. 2. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

(

[KodIzdatelya]

) REFERENCES [dbo.][Izdateli] ([KodIzdatelya] )

)

Create trigger Izd\_Delete

ON Izdateli FOR DELETE

AS BEGIN

IF EXISTS (Select \*From Knigi, Deleted

Where knigi.KodIzdatelya= Deleted.KodIzdatelya)

BEGIN

Rollback transaction

Raiserror 50002 ‘Транзакция не м.б.

Завершена, т.к. у издателя есть книги’

END END

Create trigger Avtor\_Knigi\_Insert

ON AvtorKnigi FOR INSERT

AS BEGIN

IF NOT EXISTS (Select \*From Knigi, Inserted

Where Knigi.KodKnigi= Inserted.KodKnigi)

Rollback transaction

Raiserror 50002 ‘Транзакция не м.б.

Завершена, т.к. код несуществующей книги’

END END

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить безопасность базы данных. | |
| **ОГРАНИЧЕНИЯ:** | | 1. Необходимый уровень безопасности базы данных обеспечить путем создания групп пользователей, которым позволено только просматривать неполную информацию из таблиц базы данных. 2. Заданный уровень безопасности обеспечить с помощью видов и хранимых процедур. 3. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

(

[KodIzdatelya]

) REFERENCES [dbo.][Izdateli] ([KodIzdatelya] )

)

use teis – работа с БД teis

GO

exec sp\_addlogin teis\_read, 12345 –добавляем логин, аргументы: новый логин, пароль

GO

exec sp\_adduser teis\_read, tr – добавляем пользователя, аргументы: логин, имя нового пользователя

GO

exec sp\_addrole reader, dbo – добавляем роль: имя новой роли, владелец новой роли БД

GO

exec sp\_addrolemember reader, tr – добавляем пользователя tr к роли reader

GO

CREATE VIEW avtor\_shared

as

select Familia,DataRojdenya,MastoRojdenya from dbo.Avtory

GO

grant select on dbo.avtor\_shared to reader

GO

deny select,insert,update on dbo.avtory to reader

***Grant****список\_прав\_доступа On имя\_объекта*

*To имя\_пользователя [,…n]*

***Grant****список\_прав\_доступа To имя\_пользователя [,…n]*

***CreateView****имя\_представления [(имя\_столбца,…)]*

***As****команда\_select*

*[With Check Option]*

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить безопасность базы данных. | |
| **ОГРАНИЧЕНИЯ:** | | 1. Необходимый уровень безопасности базы данных обеспечить путем создания групп пользователей, в обязанности которых входит только ввод новых данных в таблицы. 2. Заданный уровень безопасности обеспечить с помощью видов и хранимых процедур. 3. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

(

[KodIzdatelya]

) REFERENCES [dbo.][Izdateli] ([KodIzdatelya] )

)

use teis - работа с БД teis

GO

exec sp\_addlogin teis\_ins, 12345 - добавляем логин, аргументы: новый логин, пароль

GO

exec sp\_adduser teis\_ins, teis\_inserter - добавляем пользователя, аргументы: логин, имя нового пользователя

GO

exec sp\_addrole updater, dbo - добавляем роль: имя новой роли, владелец новой роли БД

GO

exec sp\_addrolemember updater, teis\_inserter - добавляем пользователя tr к роли reader

GO

CREATE PROCedure t\_ins

(@KodAvtora bigint,

@Familia char(10),

@DataRojdenya datetime,

@MastoRojdenya char(10)

)

AS

if user\_name()='teis\_inserter'

Begin

INSERT INTO [Avtory]([KodAvtora], [Familia], [DataRojdenya], [MastoRojdenya])

VALUES(@KodAvtora, @Familia, @DataRojdenya, @MastoRojdenya)

END

ELSE

Print 'No access'

GO

grant execute on dbo.t\_ins to teis\_ins

GO

deny select,insert,update on dbo.avtory to teis\_ins

***Grant****список\_прав\_доступа On имя\_объекта*

*To имя\_пользователя [,…n]*

***Grant****список\_прав\_доступа To имя\_пользователя [,…n]*

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных SQL Server | |
|  | **Таблица** | | **Реквизиты** |
|  | Книги | | Код книги, название, число страниц, код издателя, дата выхода |
|  | Авторы | | Код автора, фамилия, дата рождения, место рождения |
|  | Издатели | | Код издателя, имя издателя, адрес издателя |
|  | Автор книги | | Код автора, код книги |
| **ТРЕБУЕТСЯ:** | | 1. Изобразить схему базы данных. 2. Создать таблицы SQL Server. 3. Обеспечить безопасность базы данных. | |
| **ОГРАНИЧЕНИЯ:** | | 1. Необходимый уровень безопасности базы данных обеспечить путем создания групп пользователей, которые могут просматривать сводку, обобщающую данные из всех таблиц (без отображения кодов). 2. Заданный уровень безопасности обеспечить с помощью видов и хранимых процедур. 3. Запросы на создание объектов базы данных реализовать на Transact-SQL. | |

CREATE TABLE [dbo].[ [Knigi] (

[KodKnigi] [bigint] NOT NULL ,

[Nazvanie] [char] (10) NOT NULL,

[ChisloStranits] [bigint] NOT NULL,

[KodIzdatelya] [bigint] NULL ,

[DataVihoda] [datetime] NULL ,

CONSTRAINT [PK\_Knigi] PRIMARY KEY

(

[KodKnigi]

) ,

CONSTRAINT [FK\_Knigi\_Izdateli] FOREIGN KEY

(

[KodIzdatelya]

) REFERENCES [dbo.][Izdateli] ([KodIzdatelya] )

)

USE Teis                                                          */\* Выбирается база Teis \*/*

GO

exec sp\_addlogin teis\_view, 12345               */\* создаетсялогин teis\_view \*/*

GO

exec sp\_adduser teis\_view, teis\_v                 */\* создаетсяпользователь teis\_v \*/*

GO

exec sp\_addrole viewer, dbo                          */\* создаетсяроль viever, владелец dbo \*/*

GO

exec sp\_addrolemember viewer, teis\_v        */\* пользователю teis\_v добавляетсяроль viewer \*/*

GO

***CreateView****имя\_представления [(имя\_столбца,…)]*

***As****команда\_select*

*[With Check Option]*

CREATE VIEW userview

AS

SELECT

[ImyaIzdatelya],

[AdreaIzdatelya],

[Familia],

[DataRojdenya],

[MastoRojdenya],

[Nazvanie],

[ChisloStranits],

[DataVihoda]

FROM

[Izdateli],

[Avtory],

[Knigi],

[AvtorKnigi]

WHERE

Knigi.KodKnigi=AvtorKnigi.KodKnigi

AND

AvtorKnigi.KodAvtora=Avtory.KodAvtora

AND

Knigi.KodIzdatelya=Izdateli.KodIzdatelya

***Grant****список\_прав\_доступа On имя\_объекта*

*To имя\_пользователя [,…n]*

***Grant****список\_прав\_доступа To имя\_пользователя [,…n]*

GO

grant select on dbo.userview to teis\_v

GO

deny select,insert,update on dbo.avtory to teis\_v

GO

deny select,insert,update on dbo.AvtorKnigi to teis\_v

GO

deny select,insert,update on dbo.Izdateli to teis\_v

GO

deny select,insert,update on dbo.Knigi to teis\_v

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных ORACLE | |
|  | **Таблица** | | **Реквизиты** |
|  | Фильм | | Код фильма, название фильма, режиссер, стоимость, количество экземпляров |
|  | Клиент | | Код клиента, адрес, телефон |
|  | Договор | | Код договора, код фильма, код клиента, дата выдачи, дата возврата |
| **ТРЕБУЕТСЯ:** | | 1. Создать триггер, который поддерживал бы в актуальном состоянии значение в поле «количество экземпляров» таблицы Фильм. 2. В словаре данных проверить наличие и состояние триггера. 3. Проверить работу триггера. | |

Создание триггера:

**SQL> edit trig273**

При этом автоматически создастся файл trig273.sql, и откроется блокнот для его редактирования. В этот текстовый редактор заносим текст программы:

*CREATEORREPLACETRIGGERtrig273*

*AFTER INSERT OR UPDATE*

*ON Dogovor*

*REFERENCING OLD AS old\_row NEW AS new\_row*

*FOR EACH ROW*

*DECLARE*

*CountFilms NUMBER;*

*BEGIN*

*IF (INSERTING) THEN*

*SELECT kol*

**INTO CountFilms**

*FROM film*

*WHERE filmid=:new\_row.filmid;*

*IF(CountFilms = 0) THEN*

*raise\_application\_error(-20001,'Все кассеты с данным фильмом находятся у клиентов!!!Придётся потерпеть!!!');*

*END IF;*

*UPDATE film*

*SET kol=kol-1*

*WHERE filmid=:new\_row.filmid;*

*ELSIF (UPDATING) THEN*

*IF (:new\_row.vozvrat IS NOT NULL) THEN*

*IF (:old\_row.vozvrat IS NULL) THEN*

*UPDATE film*

*SET kol=kol+1*

*WHERE filmid=:new\_row.filmid;*

*END IF;*

*END IF;*

*END IF;*

*END;*

*/*

Запуск триггера на выполнение:

*SQL> @trig273*

*Ввод усечен до 1 символов*

*Триггер создан.*

Проверка в словаре данных:

SELECT TRIGGER\_NAME, TRIGGER\_TYPE, TABLE\_NAME, TRIGGERING\_EVENT

FROM USER\_TRIGGERS

WHERE TRIGGER\_NAME = '*trig273*'

Проверка работы триггера:

Select \* from film;

Insert into dogovor (kod\_f, kod\_kl, data\_vidachi)

values(1,2, ’12.10.09');

select \* from film;

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных ORACLE | |
|  | **Таблица** | | **Реквизиты** |
|  | Фильм | | Код фильма, название фильма, режиссер, стоимость, количество экземпляров |
|  | Клиент | | Код клиента, адрес, телефон |
|  | Договор | | Код договора, код фильма, код клиента, дата выдачи, дата возврата |
| **ТРЕБУЕТСЯ:** | | 1. Создать триггер, который проверяет возможность вставки в таблицу Договор новой записи с существующими кодами клиентов и кодами фильмов в таблицах Фильм и Клиент. 2. В словаре данных проверить наличие и состояние триггера. 3. Проверить работу триггера. | |

Создание триггера:

## SQL> edit trig271

При этом автоматически создастся файл trig271.sql, и откроется блокнот для его редактирования. В этот текстовый редактор заносим текст программы:

*CREATEORREPLACETRIGGERtrig2711*

*BEFORE INSERT*

*ON Dogovor*

*REFERENCING NEW as new\_row*

*FOR EACH ROW*

*DECLARE*

*CountFilm NUMBER;*

*CountClient NUMBER;*

*BEGIN*

*SELECT count(filmid)*

*INTO CountFilm*

*FROM Film*

*WHERE Filmid=:new\_row.filmid;*

*SELECT count(Clientid)*

*INTO CountClient*

*FROM Client*

*WHERE Clientid=:new\_row.Clientid;*

*IF (CountFilm = 0) THEN*

*raise\_application\_error(-20001,'Данногофильманесуществует!!!');*

*ELSIF(CountClient = 0) then*

*raise\_application\_error(-20001,'Данного клиента не существует!!!');*

*ENDIF;*

*END;*

*/*

После сохранения созданного, исполняем этот скрипт из командной строки

Запуск триггера на выполнение:

*SQL> @trig2711*

*Ввод усечен до 1 символов*

*Триггер создан.*

Выполняем проверку работы триггера:

*SQL> insertintodogovor*

*2  values(10,'18.12.2004','',7);*

*insert into dogovor*

*\**

*ошибка в строке 1:*

*ORA-20001: Данного клиента не существует!!!*

*ORA-06512: на  "EI0101.TRIG2711", line 19*

*ORA-04088: ошибка во время выполнения триггера 'EI0101.TRIG2711'*

*SQL> insert into dogovor*

*2  values(12,'18.12.2004','',5);*

*insert into dogovor*

*\**

*ошибкавстроке 1:*

*ORA-20001: Данного фильма не существует!!!*

*ORA-06512: на  "EI0101.TRIG2711", line 17*

*ORA-04088: ошибка во время выполнения триггера 'EI0101.TRIG2711'*

*SQL> insert into dogovor*

*2  values(10,'18.12.2004','',5);*

*1 строка создана.*

Проверка в словаре данных:

SELECT TRIGGER\_NAME, TRIGGER\_TYPE, TABLE\_NAME, TRIGGERING\_EVENT

FROM USER\_TRIGGERS

WHERE TRIGGER\_NAME = '*trig2711*'

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных ORACLE | |
|  | **Таблица** | | **Реквизиты** |
|  | Фильм | | Код фильма, название фильма, режиссер, стоимость, количество экземпляров |
|  | Клиент | | Код клиента, адрес, телефон |
|  | Договор | | Код договора, код фильма, код клиента, дата выдачи, дата возврата |
| **ТРЕБУЕТСЯ:** | | 1. Создать триггер, который позволит удалить записи о фильмах из таблицы Фильм, если их нет в таблице Договор или выдаст сообщение о наличие связанных записях. 2. В словаре данных проверить наличие и состояние триггера. 3. Проверить работу триггера. | |

Создание триггера:

## SQL> edit trig113

При этом автоматически создастся файл trig113.sql, и откроется блокнот для его редактирования. В этот текстовый редактор заносим текст программы:

*CREATEORREPLACETRIGGERtrig113*

*BEFORE DELETE*

*ON FILM*

*REFERENCING OLD as old\_row*

## FOR EACH ROW

*DECLARE*

*CountDogovor number;*

*BEGIN*

*SELECT count(filmid)*

*INTO CountDogovor*

*FROM dogovor*

*WHERE filmid=:old\_row.filmid;*

*if (CountDogovor > 0) THEN*

*raise\_application\_error(-20001,*

*'Необходимо удалить связанные записи из таблицы договор!');*

*ENDIF;*

*END;*

*/*

После сохранения созданного, исполняем этот скрипт из командной строки

Запуск триггера на выполнение:

## SQL>@trig113

*Ввод усечен до 1 символов*

*Триггер создан.*

Выполняем удаление с целью проверки работы триггера:

*SQL> delete from film*

*2  where filmid=9;*

*delete from film*

*\**

*ошибкавстроке 1:*

*ORA-20001: Необходимо удалить связанные записи из таблицы договор!*

*ORA-06512: на  "EI0101.TRIG113", line 11*

*ORA-04088: ошибка во время выполнения триггера 'EI0101.TRIG113*

Выполняем удаление с целью проверки работы триггера:

*SQL> delete from film*

*2  where filmid=6;*

***1 строка удалена.***

Проверка в словаре данных:

SELECT TRIGGER\_NAME, TRIGGER\_TYPE, TABLE\_NAME, TRIGGERING\_EVENT

FROM USER\_TRIGGERS

WHERE TRIGGER\_NAME = '*trig113*'

|  |  |  |  |
| --- | --- | --- | --- |
| **ДАНО:** | | Таблицы реляционной базы данных ORACLE | |
|  | **Таблица** | | **Реквизиты** |
|  | Фильм | | Код фильма, название фильма, режиссер, стоимость, количество экземпляров |
|  | Клиент | | Код клиента, адрес, телефон |
|  | Договор | | Код договора, код фильма, код клиента, дата выдачи, дата возврата |
| **ТРЕБУЕТСЯ:** | | 1. Создать триггер, который облегчает работу заполнения БД при работе с таблицами Договор и Фильм, если фильм с номером вернули, то надо подсчитать, сколько стоит услуга. 2. В словаре данных проверить наличие и состояние триггера. 3. Проверить работу триггера. | |

Создание триггера:

## SQL> edit trig274

При этом автоматически создастся файл trig274.sql, и откроется блокнот для его редактирования. В этот текстовый редактор заносим текст программы:

*CREATEORREPLACETRIGGER trig274*

*AFTER UPDATE*

*ON Dogovor*

*REFERENCING OLD AS old\_row NEW AS new\_row*

*FOR EACH ROW*

*DECLARE*

*Stoim NUMBER;*

*Prok NUMBER;*

*Itog NUMBER;*

*BEGIN*

*Stoim := 0;*

*IF (:new\_row.vozvrat IS NOT NULL) THEN*

*IF (:old\_row.vozvrat IS NULL) THEN*

*SELECT Stoimost\_prok*

*INTO Prok*

*FROM Film*

*WHERE :new\_row.filmid=film.filmid;*

*Stoim:=(:new\_row.vozvrat - :new\_row.vidacha);*

*Itog:=Stoim\*Prok;*

*DBMS\_OUTPUT.PUT\_LINE('-----------------------------------------');*

*DBMS\_OUTPUT.PUT\_LINE('                                         ');*

*DBMS\_OUTPUT.PUT\_LINE('Kolichestvo dnei prokata:');*

*DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(Stoim));*

*DBMS\_OUTPUT.PUT\_LINE('Obshaya summa k oplate:');*

*DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(Itog));*

*DBMS\_OUTPUT.PUT\_LINE('                                         ');*

*DBMS\_OUTPUT.PUT\_LINE('-----------------------------------------');*

*END IF;*

*END IF;*

*END;*

*/*

Запуск триггера на выполнение:

*SQL> @trig274*

## Ввод усечен до 1 символов

*Триггер создан.*

Выполняем проверку работы триггера:

*SQL> update dogovor*

*2  set vozvrat='20.11.2004'*

*3  where filmid=1 and clientid=2;*

*-----------------------------------------*

*Kolichestvo dnei prokata:*

*10*

*Obshaya summa k oplate:*

*1000*

*-----------------------------------------*

Проверка в словаре данных:

SELECT TRIGGER\_NAME, TRIGGER\_TYPE, TABLE\_NAME, TRIGGERING\_EVENT

FROM USER\_TRIGGERS

WHERE TRIGGER\_NAME = '*trig274*'

**ДАНО:** В организацию пришло письмо «Вышлите нам сто тысяч», подписанное цифровой подписью 21. Подпись была сформирована на основе хэш-функции X.509 по упрощенной схеме (в десятичных кодах) с параметрами p=5, q=17 и начальным значением H0 = 2 и алгоритма RSA.

**ТРЕБУЕТСЯ:** Определить, следует ли высылать отправителю сто тысяч?

**РЕШЕНИЕ:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| А – 01 | З – 08 | О - 15 | Х – 22 | Ь – 29 |
| Б – 02 | И – 09 | П – 16 | Ц – 23 | Э – 30 |
| В – 03 | Й – 10 | Р – 17 | Ч - 24 | Ю – 31 |
| Г – 04 | К – 11 | С – 18 | Ш – 25 | Я – 32 |
| Д – 05 | Л – 12 | Т – 19 | Щ – 26 | \_ - 33 |
| Е – 06 | М – 13 | У – 20 | Ъ – 27 |  |
| Ж - 07 | Н – 14 | Ф- 21 | Ы – 28 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 | M10 | M11 | M12 | M13 | M14 | M15 | M16 | M17 | M18 | M19 | M20 | M21 |
| в | ы | ш | л | и | т | е | \_ | н | а | м | \_ | с | т | о | \_ | т | ы | с | я | ч |
| 3 | 28 | 25 | 12 | 9 | 19 | 6 | 33 | 14 | 1 | 13 | 33 | 18 | 19 | 15 | 33 | 19 | 28 | 18 | 32 | 24 |

1. Вычислить произведение двух простых чисел p и q:

n = pq = 5\*17 = 85

2. Для каждого блока вычислить его хэш-код по формуле:

, т. е. каждый из блоков складывается с предыдущим и возводится в квадрат по модулю *n*. Полученное в итоге значение *Hk* и будет хэш-кодом.

H1= (М1 + H0)2 mod85= (3 + 2)2mod85 = 25

H2= (М2 + H1)2 mod85= (28 + 25)2mod85 = 4

H3= (М3 + H2)2 mod85= (25 + 4)2mod85 = 76

H4= (М4 + H3)2 mod85= (12 + 76)2mod85 = 9

H5= (М5 + H4)2 mod85= (9+ 9)2mod85 = 69

H6= (М6 + H5)2 mod85= (19 + 69)2mod85 = 9

H7= (М7+ H6)2 mod85= (6 + 9)2mod85 = 55

H8= (М8 + H7)2 mod85= (33 + 55)2mod85 = 9

H9= (М9 + H8)2 mod85= (14 + 9)2mod85 = 19

H10= (М10 + H8)2 mod85= (1 + 19)2mod85 = 60

H11= (М11 + H10)2 mod85= (13+ 60)2mod85 = 59

H12= (М12 + H11)2 mod85= (33 + 59)2mod85 = 49

H13= (М13 + H12)2 mod85= (18 + 49)2mod85 = 69

H14= (М14 + H13)2 mod85= (19 + 69)2mod85 = 9

H15= (М15 + H14)2 mod85= (15 + 9)2mod85 = 66

H16= (М16 + H15)2 mod85= (33 + 66)2mod85 = 26

H17= (М17 + H16)2 mod85= (19 + 26)2mod85 = 70

H18= (М18 + H17)2 mod85= (28 + 70)2mod85 = 84

H19= (М19 + H18)2 mod85= (18 + 84)2mod85 = 34

H20= (М20 + H19)2 mod85= (32 + 34)2mod85 = 21

H21= (М21 + H20)2 mod85= (24 + 21)2mod85 = 70

Таким образом, хэш-код сообщения *H* = 70. Для преобразования его в электронную цифровую подпись необходимо выполнить процедуру шифрования с помощью алгоритма RSA.

1) s = (p-1)(q-1) = 4\*16 = 64

d = 3 – взаимно простое с s (64 нацело не делится на 3)

2) (e\*d)mods = 1

(e\*3)mod64 = 1

(e\*3) = 64+1 = 65 – нет целочисленного решения (65 не делится нацело на 3)

(e\*3) = 64\*2+1 = 129, есть решение (129/3 = 43), поэтому e = 43

Открытый ключ: (e; pq) = (43; 85)

Секретный ключ: (d; pq) = (3; 85)

Зашифровав свертку с помощью секретного ключа, получим электронную цифровую подпись:

C = Hemodn

**C = 7043mod85 = 80 ()**

***80 ≠ 21, следовательно отправителю не следует высылать сто тысяч!***

**ДАНО:** ЭВМ, имеющая показательное распределение наработки до отказа.

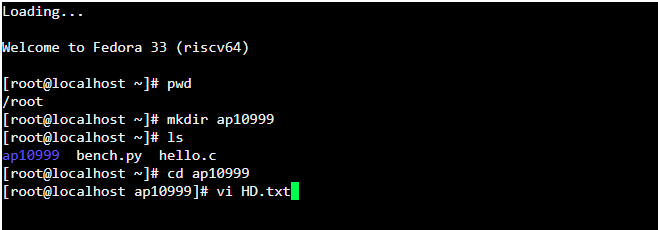
**ТРЕБУЕТСЯ:** Определить, какова должна быть средняя наработка до отказа Т, чтобы вероятность безотказной работы Р(t) была не менее 0,99 в течении наработки t = 300 часов.

**ДАНО:** система обработки данных, имеющая показательное распределение наработки до отказа. Вероятность безотказной работы СОД в течении наработки (0,100) часов равна 0,99.

**ТРЕБУЕТСЯ:** Вычислить без применения таблиц показательной функции exp(x) среднюю наработку до отказа Т.

|  |  |
| --- | --- |
| **ДАНО:** | Командный интерпретатор shell в Linux |
| **ТРЕБУЕТСЯ:** | 1. Создать структуру каталогов, изображенную на рис. 1. 2. Провести ряд экспериментов, иллюстрирующих доступ к файлам по основным именам, по ссылкам и по символическим ссылкам. 3. Провести ряд экспериментов, иллюстрирующих реакцию системы на удаление файла, на который имеются ссылки, и файла, на который имеются символические ссылки.     Рис. 1 |
| **ОГРАНИЧЕНИЯ** | 1. каталоги представлены элементами вида , файлы ‑ элементами вида 2. Жирными линиями представлена вложенность файлов/подкаталогов в каталоги. 3. Один файл создается в текстовом редакторе, остальные файлы создаются копированием файла и с помощью ссылок. 4. Жесткие ссылки представлены черными тонкими линиями. 5. Символические ссылки представлены на рисунке стрелками. Стрелка на линии указывает на целевой файл ссылки. Корнем дерева, должен быть один из подкаталогов в домашнем каталоге. |

Создаем каталог ap10999 и файл HD.txt





Создаем каталоги D1 и D2

В D1 копируем файл /root/ap10999/HD.txt и создаем каталог D3.

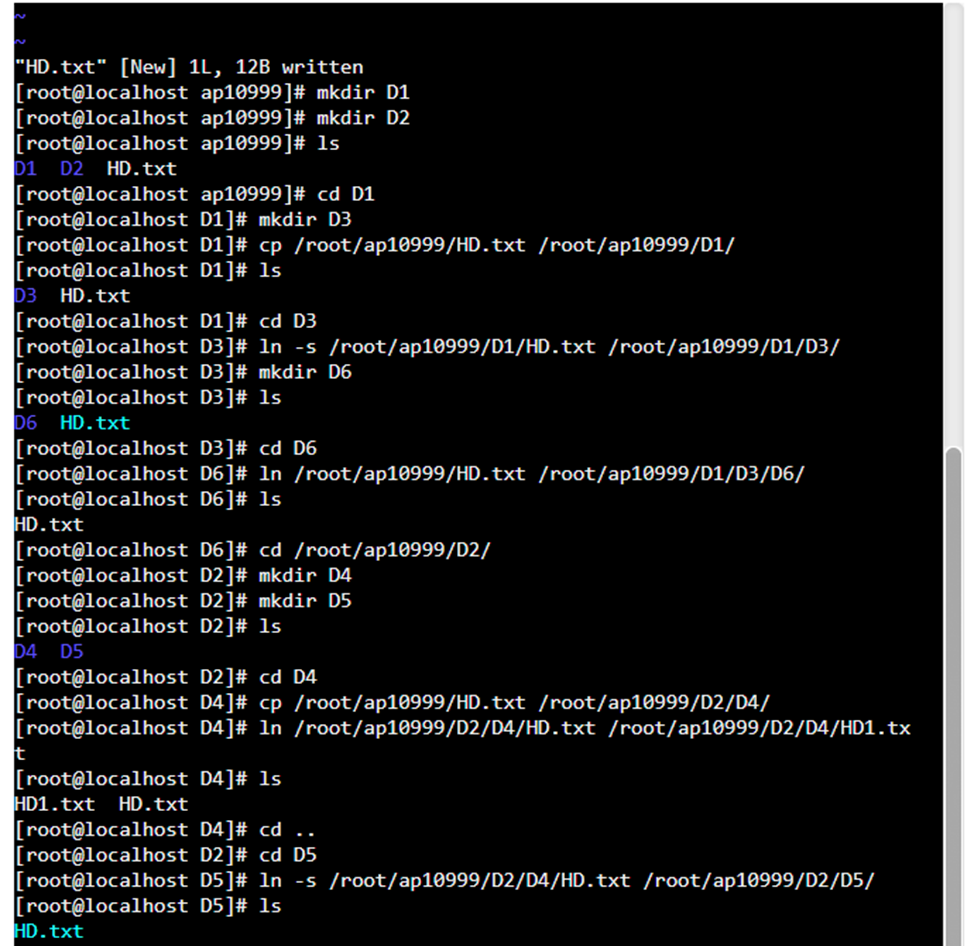
В D3 создаем символическую ссылку на файл /root/ap10999/D1/HD.txt и создаем каталог D6.

В D6 создаем жесткую ссылку на файл /root/ap10999/HD.txt.

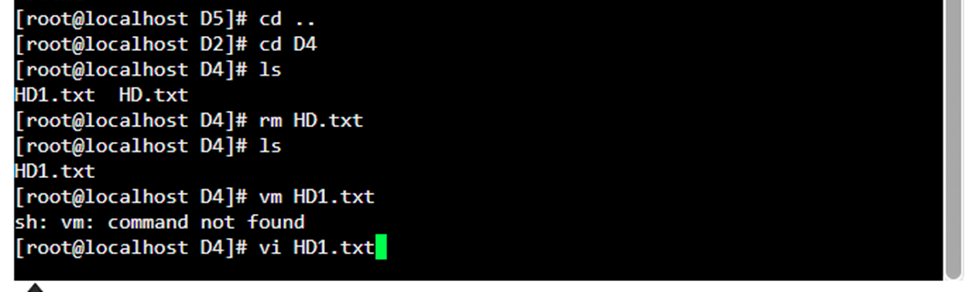
В D2 создаем каталоги D4 и D5.

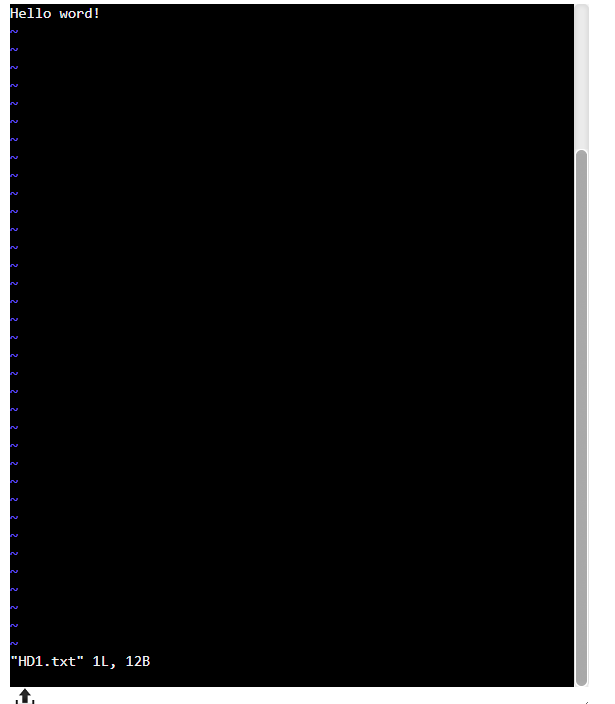
В D4 копируем файл /root/ap10999/HD.txt и создаем на копию жесткую ссылку HD1.txt

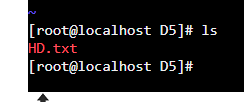
В D5 создаем символическую ссылку на файл /root/ap10999/D2/D4/ HD.txt

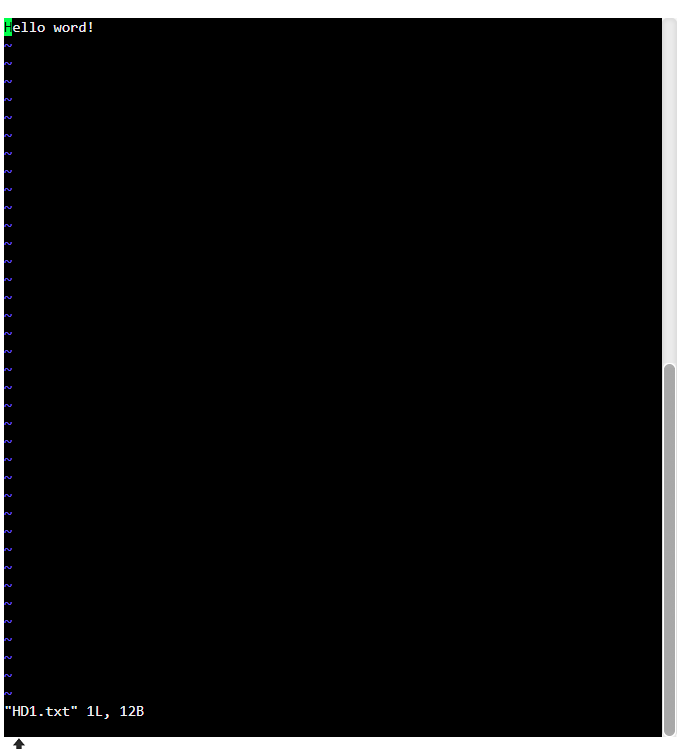


Для того чтобы проверить реакцию системы на удаление файла на который имеется жесткая и символическая ссылки удалим файл /root/ap10999/D2/D4/ HD.txt на который имеется оба вида ссылок.









Ссылки в Linux бывают двух типов: символические и жесткие. Не смотря на то, что оба типа называются ссылками, они имеют существенные отличия друг от друга. Поэтому очень важно понимать, как создавать и использовать тот или иной тип ссылок.

Что такое символические ссылки

Символическая ссылка (symbolic link) — это специальный файл, который является ссылкой на другой файл или каталог (их еще называют целевым файлом, целевым каталогом).

Символические ссылки также называют символьными, мягкими ссылками (soft links) или сим-ссылками (sym-link).

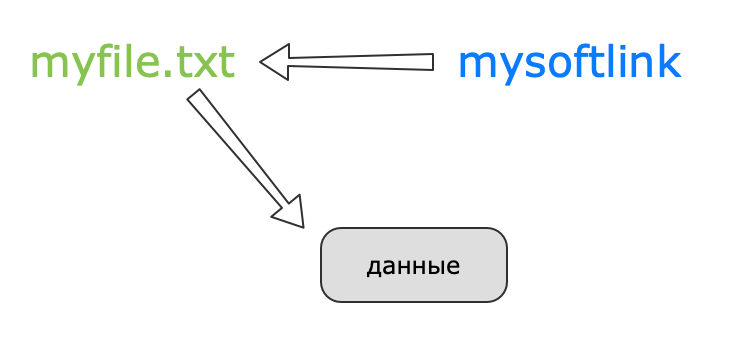
Важно понимать, что символическая ссылка не содержит в себе внутри копии самого файла, на которую она указывает. Она является всего лишь указателем на файл. Не смотря на это, символическая ссылка обладает собственными правами доступа, так как сама является небольшим файлом, который содержит путь до целевого файла.

Возвращаясь к аналогии с ярлыками в Windows, символические ссылки это своего рода ярлыки на файлы. Можно создавать несколько символических ссылок на один файл и эти ссылки могут иметь разные имена.

Связь между символической ссылкой и файлом, на который она указывает, является «мягкой». Если удалить символическую ссылку, то файл, на который она указывает, не удаляется.

Если удалить файл, на который указывает ссылка, то сама ссылка не обновляется и остается на диске. При этом она указывает на уже несуществующий файл. Аналогично, если переименовать или переместить целевой файл, то ссылка не обновляется автоматически.

При создании символических ссылок можно указывать относительный путь до целевого файла. В таком случае ссылка считает, что относительный путь указан относительно каталога, в котором создана сама ссылка (но не относительно каталога, из которого она была создана).

Схематично отношение между файлом, символической ссылкой и данными, которые хранятся в файле, можно показать следующим образом:  


Что такое жесткие ссылки

Жесткая ссылка (hard link) является своего рода синонимом для существующего файла. Когда вы создаете жесткую ссылку, создается дополнительный указатель на существующий файл, но не копия файла.

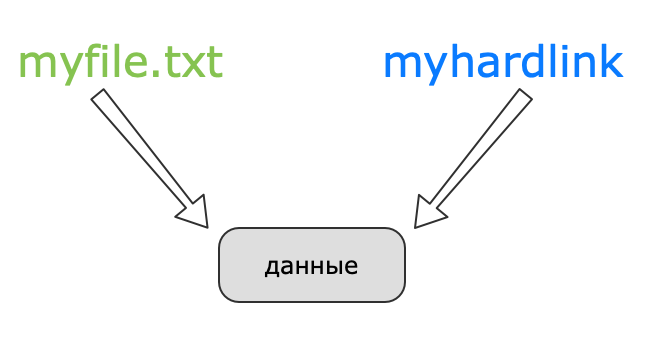
Жесткие ссылки выглядят в файловой структуре как еще один файл. Если вы создаете жесткую ссылку в том же каталоге, где находится целевой файл, то они должны иметь разные имена. Жесткая ссылка на файл должна находится в той же файловой системе, где и другие жесткие ссылки на этот файл.

В Linux каждый файл имеет уникальный идентификатор - индексный дескриптор (inode). Это число, которое однозначно идентифицирует файл в файловой системе. Жесткая ссылка и файл, для которой она создавалась имеют одинаковые inode. Поэтому жесткая ссылка имеет те же права доступа, владельца и время последней модификации, что и целевой файл. Различаются только имена файлов. Фактически жесткая ссылка это еще одно имя для файла.

Жесткие ссылки нельзя создавать для директорий.

Жесткая ссылка не может указывать на несуществующий файл.

Жесткие ссылки появились раньше, чем символические, но сейчас уже устаревают. В повседневной работе жесткие ссылки используются редко.

Схематично отношение между исходным файлом, жесткой ссылкой и данными можно показать следующей схемой:  


Отличия символических ссылок от жестких

Кратко подведем итог, написанного выше.

Символическая ссылка:

* Указывает на целевой файл или каталог. Фактически является небольшим файлом, содержащим путь до целевого файла.
* Не содержит внутри себя содержимого самого файла. Содержит путь к целевому файлу.
* Имеет собственные права доступа, которые не распространяются на целевой файл.
* Удаление / переименование / перемещение целевого файла не обновляет автоматически ссылку. Ссылка начинает указывать на несуществующий файл, становится неработающей.
* Изменение прав доступа у целевого файла не обновляет права доступа у ссылки.
* Может быть создана для директории.
* Ссылка и целевой файл имеют разные файловые индексы (inode) в файловой системе.
* Может указывать на несуществующий файл.
* Символическая ссылка может использовать относительный путь до целевого файла.

Жесткая ссылка:

* Является своего рода еще одним именем на файл.
* Не может указывать на директорию.
* Нельзя создавать жесткие ссылки между файлами разных файловых систем.
* Не может указывать на несуществующий файл.
* Жесткая ссылка и файл, для которого она создавалась, имеют одинаковые индексы (inode) в файловой системе.

**ДАНО:** узел с IP-адресом 57.179.208.27, адрес сети равен 57.179.192.0.

**ТРЕБУЕТСЯ:** определить, каково наибольшее возможное количество единиц в разрядах маски. Определить для этой сети широковещательный адрес.

Заметим, что первые два байта IP-адреса совпадают с адресом сети, следовательно, маска сети для этих двух байт состоит только из единиц. Заметим также, что четвёртый байт IP-адреса отличен от нуля, но при этом четвёртый байт адреса сети равен нулю, значит, нужно положить четвёртый байт маски равным нулю.

Рассмотрим третий байт IP-адреса и адреса сети в двоичной системе счисления:

20810 = 1101 00002

19210 = 1100 00002

Откуда ясно, что два первых слева бита маски − единицы, а третий бит может быть как нулём, так и единицей. Для того, чтобы количество единиц было наибольшим, третий бит должен быть равен единице. Получаем, что третий слева байт маски равен 1110 0000.

Таким образом, наибольшее возможное количество единиц в разрядах маски равно 8 · 2 + 3 = 19.

Широковещательный адрес: 57.179.223.255