Ебанный экз по проге ИВТИ 2 сем

Оглавление

[Рекурсия 1 1](#_Toc138679988)

[Рекурсия 2 2](#_Toc138679989)

[Строки в Delphi 4](#_Toc138679990)

[Строки в Cи 5](#_Toc138679991)

[Типы файлов в Delphi 6](#_Toc138679992)

[ТЕКСТОВЫЙ 6](#_Toc138679993)

[ТИПИЗИРОВАННЫЙ 8](#_Toc138679994)

[БЕСТИПОВЫЙ 10](#_Toc138679995)

[Типы файлов в СИ 12](#_Toc138679996)

[ТЕКСТОВЫЙ 12](#_Toc138679997)

[ДВОИЧНЫЙ/БИНАРНЫЙ 13](#_Toc138679998)

[Указатели в Delphi 14](#_Toc138679999)

[Указатели в Си 16](#_Toc138680000)

[Массивы в Delphi 17](#_Toc138680001)

[Одномерные 17](#_Toc138680002)

[Статистические и динамические 17](#_Toc138680003)

[Динамические одно- двух- мерные массивы 19](#_Toc138680004)

[Массивы в Си статические и динамические одно- двух- мерные массивы 20](#_Toc138680005)

[Односвязный и двусвязный линейные списки 21](#_Toc138680006)

[СТЕК 23](#_Toc138680007)

[Записи в Delphi, структуры и объединения в Си 24](#_Toc138680008)

[Язык Си 26](#_Toc138680009)

[**Процедурный тип** в Delphi и указатель на функцию в Си 31](#_Toc138680010)

# Рекурсия 1

**Рекурсивный алгоритм** - это алгоритм, который вызывает сам себя для решения задачи. При каждом вызове функция создает новый экземпляр самой себя, который работает независимо от остальных экземпляров.

**Дерево рекурсивных вызовов** - это диаграмма, которая показывает все вызовы функции и их зависимости друг от друга. Каждый узел дерева представляет один вызов функции, а дуги между узлами показывают, какие вызовы зависят от других вызовов.

**Глубина вызовов** - это количество уровней в дереве рекурсивных вызовов.

**Объем рекурсии** - это количество вызовов функции, которые происходят во время выполнения программы.

**Линейная рекурсия** - это рекурсия, в которой каждый вызов функции порождает только один новый вызов функции. Примером линейной рекурсии может быть функция вычисления факториала.

#include <stdio.h>

int factorial(int n) {

if (n == 0) {

return 1;

}

else {

return n \* factorial(n - 1);

}

}

**Каскадная рекурсия** - это рекурсия, в которой каждый вызов функции порождает несколько новых вызовов функции. Примером каскадной рекурсии может быть функция вычисления чисел Фибоначчи.

#include <stdio.h>

int fibonacci(int n) {

if (n == 0) {

return 0;

}

else if (n == 1) {

return 1;

}

else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

**Стек рекурсивных вызовов** - это стек, который используется для хранения информации о вызовах функций во время выполнения программы. Каждый раз, когда функция вызывает саму себя, информация о вызове помещается на вершину стека. Когда функция завершается, информация о вызове удаляется из вершины стека.

Переполнение стека рекурсивных вызовов происходит, когда стек заполняется слишком многими вызовами функций, и не хватает места для новых вызовов. Это может произойти, если рекурсия слишком глубокая или если объем рекурсии слишком большой.

Способы предотвращения переполнения стека рекурсивных вызовов включают использование циклов вместо рекурсии, уменьшение глубины рекурсии или увеличение размера стека.

# Рекурсия 2

**Взаимнорекурсивные процедуры/функции** - это процедуры/функции, которые вызывают друг друга взаимно. Такие процедуры/функции могут использоваться для решения задач, которые можно разбить на несколько подзадач, каждая из которых решается отдельной процедурой/функцией.

В **Delphi** для описания взаимнорекурсивных процедур/функций используется ключевое слово **forward**. Оно позволяет объявить процедуру/функцию без ее определения, чтобы можно было вызывать ее из других процедур/функций, которые определены раньше.

Пример описания взаимнорекурсивных процедур в Delphi:

unit MyUnit;

interface

procedure ProcA; forward;

procedure ProcB; forward;

implementation

procedure ProcA;

begin

// some code

ProcB;

// some code

end;

procedure ProcB;

begin

// some code

ProcA;

// some code

end;

end.

В Си для описания взаимнорекурсивных функций используются прототипы. Прототипы позволяют объявить функцию без ее определения, чтобы можно было вызывать ее из других функций, которые определены раньше.

Пример описания взаимнорекурсивных функций в Си:

#include <stdio.h>

void funcA(void);

void funcB(void);

void funcA(void) {

// some code

funcB();

// some code

}

void funcB(void) {

// some code

funcA();

// some code

}

int main() {

funcA();

return 0;

}

# Строки в Delphi

**ShortString** - это строка фиксированной длины, которая может содержать до 255 символов. Она хранится в памяти как массив символов, где первый элемент - это длина строки.

**AnsiString** - это строка переменной длины, которая может содержать до 2^31 символов. Она хранится в памяти как указатель на массив символов.

Для ввода и вывода строк в **Delphi** используются стандартные процедуры **ReadLn** и **WriteLn**. Например:

var

s: string;

begin

WriteLn('Введите строку:');

ReadLn(s);

WriteLn('Вы ввели: ', s);

end.

Нулевой элемент в строках Delphi не используется.

Индекс первого символа строки в Delphi равен 1.

**Статические строки** - это строки, которые определены в коде программы и не могут быть изменены во время выполнения.

**Динамические строки** - это строки, которые создаются во время выполнения программы и могут быть изменены.

Процедуры строковых типов в **Delphi** могут принимать следующие параметры:

- Строка, с которой нужно произвести операцию.

- Индекс символа, с которого нужно начать операцию.

- Длина подстроки, с которой нужно произвести операцию.

Например, процедура Delete удаляет из строки заданное количество символов, начиная с заданного индекса:

var

s: string;

begin

s := 'Hello, world!';

Delete(s, 7, 6); // удаляем подстроку ", world"

WriteLn(s); // выводим "Hello!"

end.

Чтобы подсчитать количество цифр в заданной строке, можно использовать следующий код:

function CountDigits(s: string): Integer;

var

i, count: Integer;

begin

count := 0;

for i := 1 to Length(s) do

if s[i] in ['0'..'9'] then

Inc(count);

Result := count;

end;

var

s: string;

begin

s := 'abc123def456';

WriteLn('Количество цифр в строке "', s, '": ', CountDigits(s));

end.

Эта программа выведет "Количество цифр в строке "abc123def456": 6".

# Строки в Cи

**Строки в Си** - это массивы символов, заканчивающиеся нулевым символом ('\0'). Для ввода и вывода строк используются функции scanf() и printf() со спецификатором %s.

Отличия от строк Delphi заключаются в том, что в Си строки - это массивы символов, а в Delphi - это отдельный тип данных. В Си индекс первого символа строки начинается с 0, а в Delphi - с 1.

**Статические строки** - это строки, определенные в исходном коде программы и выделенные в статической памяти.

**Динамические строки** - это строки, выделенные в динамической памяти с помощью функции malloc() или calloc().

Процедуры строковых типов в Си включают функции для работы со строками, такие как strlen(), strcpy(), strcat(), strcmp() и т.д.

Пример программного кода для подсчета количества цифр в заданной строке:

#include <stdio.h>

#include <ctype.h>

int main() {

char str[100];

int count = 0;

printf("Введите строку: ");

scanf("%s", str);

for (int i = 0; str[i] != '0'; i++) {

if (isdigit(str[i])) {

count++;

}

}

printf("Количество цифр в строке: %d", count);

return 0;

}

В этом примере мы объявляем массив символов str и считываем строку с помощью функции scanf(). Затем мы проходим по каждому символу строки с помощью цикла for и проверяем, является ли символ цифрой с помощью функции isdigit(). Если символ является цифрой, мы увеличиваем счетчик count. В конце мы выводим количество цифр в строке с помощью функции printf().

# Типы файлов в Delphi

# ТЕКСТОВЫЙ

В Delphi существует несколько типов файлов, но одним из наиболее распространенных является текстовый файл. Текстовый файл представляет собой последовательность символов, которые могут быть прочитаны и записаны в файл.

Для работы с текстовым файлом в Delphi используется файловая переменная типа TextFile. Она объявляется следующим образом:

var

MyFile: TextFile;

Для открытия текстового файла используется процедура AssignFile, которая принимает два параметра: файловую переменную и имя файла. Например:

AssignFile(MyFile, 'file1.txt');

Для чтения данных из файла используется процедура ReadLn, которая читает одну строку из файла и перемещает указатель на следующую строку. Например:

var

FirstNumber, LastNumber, CurrentNumber, Sum: Integer;

begin

AssignFile(MyFile, 'file1.txt');

Reset(MyFile);

ReadLn(MyFile, FirstNumber);

while not Eof(MyFile) do

begin

ReadLn(MyFile, CurrentNumber);

LastNumber := CurrentNumber;

end;

Sum := FirstNumber + LastNumber;

CloseFile(MyFile);

Writeln('Sum: ', Sum);

end.

В этом примере мы открываем файл file1.txt, читаем первое число в переменную FirstNumber, затем читаем остальные числа, перемещая указатель на следующую строку, и сохраняем последнее число в переменную LastNumber. Затем мы суммируем первое и последнее число и выводим результат на экран.

Для записи данных в файл используется процедура Write или Writeln. Например:

var

MyFile: TextFile;

begin

AssignFile(MyFile, 'file1.txt');

Rewrite(MyFile);

WriteLn(MyFile, 'Hello, world!');

CloseFile(MyFile);

end.

В этом примере мы открываем файл file1.txt для записи, записываем строку 'Hello, world!' в файл и закрываем файл.

При работе с файлами необходимо учитывать возможные ошибки, такие как отсутствие файла, недостаточные права доступа и т.д. Для проверки ошибок используется функция IOResult, которая возвращает код ошибки после выполнения последней операции с файлом. Например:

var

MyFile: TextFile;

begin

AssignFile(MyFile, 'file1.txt');

Reset(MyFile);

if IOResult <> 0 then

begin

Writeln('Error opening file');

Exit;

end;

// ...

end.

В этом примере мы проверяем результат открытия файла и выводим сообщение об ошибке, если файл не удалось открыть.

Чтобы найти сумму первого и последнего числа в файле file1.txt, можно использовать следующий код:

var

MyFile: TextFile;

FirstNumber, LastNumber, Sum: Integer;

begin

AssignFile(MyFile, 'file1.txt');

Reset(MyFile);

ReadLn(MyFile, FirstNumber);

while not Eof(MyFile) do

begin

ReadLn(MyFile, LastNumber);

end;

Sum := FirstNumber + LastNumber;

CloseFile(MyFile);

Writeln('Sum: ', Sum);

end.

В этом коде мы читаем первое число в переменную FirstNumber, затем читаем остальные числа, перемещая указатель на следующую строку, и сохраняем последнее число в переменную LastNumber. Затем мы суммируем первое и последнее число и выводим результат на экран.

# ТИПИЗИРОВАННЫЙ

В Delphi существует несколько типов файлов, но одним из наиболее удобных является типизированный файл. Типизированный файл - это файл, который содержит данные определенного типа, например, целые числа, строки или записи.

Описание файловой переменной типизированного файла в Delphi выглядит следующим образом:

var

MyFile: file of Integer;

Здесь MyFile - это переменная типа file of Integer, которая будет использоваться для работы с типизированным файлом, содержащим целые числа.

Открытие типизированного файла в Delphi происходит с помощью процедуры Reset или Rewrite. Процедура Reset открывает файл для чтения, а процедура Rewrite - для записи. Например, чтобы открыть файл file1.dat для чтения, необходимо выполнить следующий код:

Reset(MyFile, 'file1.dat');

Чтение данных из типизированного файла в Delphi осуществляется с помощью оператора Read. Например, чтобы прочитать целое число из файла, необходимо выполнить следующий код:

var

MyNumber: Integer;

begin

Read(MyFile, MyNumber);

end;

Запись данных в типизированный файл в Delphi осуществляется с помощью оператора Write. Например, чтобы записать целое число в файл, необходимо выполнить следующий код:

var

MyNumber: Integer;

begin

MyNumber := 42;

Write(MyFile, MyNumber);

end;

Закрытие типизированного файла в Delphi происходит с помощью процедуры Close. Например, чтобы закрыть файл file1.dat, необходимо выполнить следующий код:

Close(MyFile);

Особенностью типизированного файла является то, что он имеет фиксированную структуру. Это означает, что каждый элемент в файле имеет одинаковый размер. Это позволяет быстро перемещаться по файлу и читать или записывать данные в любом месте файла.

Проверка ошибок при работе с типизированным файлом в Delphi осуществляется с помощью функции IOResult. Например, чтобы проверить, была ли ошибка при открытии файла, необходимо выполнить следующий код:

Reset(MyFile, 'file1.dat');

if IOResult <> 0 then

ShowMessage('Ошибка открытия файла');

Ниже приведен код программы на Delphi, которая находит сумму первого и последнего числа в файле file1.dat и меняет их местами:

program TypedFileExample;

{$APPTYPE CONSOLE}

uses

SysUtils;

var

MyFile: file of Integer;

FirstNumber, LastNumber, Temp: Integer;

begin

// Открытие файла для чтения и записи

Reset(MyFile, 'file1.dat');

// Чтение первого числа из файла

Read(MyFile, FirstNumber);

// Поиск последнего числа в файле

while not Eof(MyFile) do

Read(MyFile, LastNumber);

// Перемена местами первого и последнего числа

Temp := FirstNumber;

FirstNumber := LastNumber;

LastNumber := Temp;

// Запись измененных чисел в файл

Reset(MyFile, 'file1.dat');

Write(MyFile, FirstNumber);

Seek(MyFile, FileSize(MyFile) - 1);

Write(MyFile, LastNumber);

// Закрытие файла

Close(MyFile);

// Вывод результата на экран

Writeln('Сумма первого и последнего числа: ', FirstNumber + LastNumber);

end.

В данном примере программа открывает файл file1.dat для чтения и записи, находит первое и последнее число в файле, меняет их местами и записывает измененные числа обратно в файл. Затем программа находит сумму первого и последнего числа и выводит ее на экран.

# БЕСТИПОВЫЙ

В Delphi существует несколько типов файлов, таких как текстовые, бинарные, файлы с фиксированной длиной записей и файлы с переменной длиной записей.

Бестиповый файл - это файл, в котором каждая запись имеет фиксированную длину и не имеет определенного типа данных. Такой файл может использоваться для хранения любых данных, например, чисел, строк, структур и т.д.

Для работы с бестиповым файлом в Delphi используется файловая переменная типа File. Она объявляется следующим образом:

var

MyFile: File;

Для открытия файла используется процедура AssignFile, которая принимает два параметра: файловую переменную и имя файла. Например:

AssignFile(MyFile, 'file1.dat');

Reset(MyFile);

Для чтения данных из файла используется процедура BlockRead, которая принимает три параметра: файловую переменную, переменную, в которую будут записаны данные, и количество байт для чтения. Например:

var

Data: array[1..100] of Byte;

begin

BlockRead(MyFile, Data, SizeOf(Data));

end;

Для записи данных в файл используется процедура BlockWrite, которая принимает три параметра: файловую переменную, переменную, из которой будут записаны данные, и количество байт для записи. Например:

var

Data: array[1..100] of Byte;

begin

BlockWrite(MyFile, Data, SizeOf(Data));

end;

После окончания работы с файлом его необходимо закрыть с помощью процедуры CloseFile:

CloseFile(MyFile);

При работе с файлами необходимо проверять возможные ошибки. Например, при открытии файла может возникнуть ошибка, если файл не существует или недоступен для чтения. Для проверки ошибок используется функция IOResult, которая возвращает код ошибки. Например:

AssignFile(MyFile, 'file1.dat');

Reset(MyFile);

if IOResult <> 0 then

begin

// Обработка ошибки

end;

Чтобы создать копию исходного файла file1.dat, необходимо открыть его для чтения, создать новый файл с помощью процедуры AssignFile и открыть его для записи, а затем последовательно читать данные из исходного файла и записывать их в новый файл. Например:

var

SourceFile, DestFile: File;

Data: array[1..100] of Byte;

begin

AssignFile(SourceFile, 'file1.dat');

Reset(SourceFile);

AssignFile(DestFile, 'file2.dat');

Rewrite(DestFile);

repeat

BlockRead(SourceFile, Data, SizeOf(Data));

BlockWrite(DestFile, Data, SizeOf(Data));

until EOF(SourceFile);

CloseFile(SourceFile);

CloseFile(DestFile);

end;

# Типы файлов в СИ

# ТЕКСТОВЫЙ

В Си файлы могут быть двух типов: текстовые и бинарные. Текстовые файлы содержат данные в виде символов, а бинарные файлы содержат данные в бинарном формате.

Для работы с файлами в Си используется файловая переменная типа FILE. Для открытия файла используется функция fopen(), которая принимает два аргумента: имя файла и режим открытия. Режимы открытия могут быть "r" (чтение), "w" (запись) и "a" (добавление).

Для чтения из файла используется функция fscanf(), которая работает аналогично функции scanf(). Для записи в файл используется функция fprintf(), которая работает аналогично функции printf(). Для закрытия файла используется функция fclose().

Форматированный вывод в файл осуществляется с помощью функции fprintf(). Она принимает первым аргументом файловую переменную, а затем строку формата и аргументы, которые нужно вывести в файл.

Особенностью текстовых файлов является то, что они содержат данные в виде символов, а не бинарных данных. Каждая строка в текстовом файле заканчивается символом перевода строки '\n'. При чтении из текстового файла функция fscanf() автоматически разбивает файл на строки и считывает данные до символа перевода строки.

При открытии файла необходимо проверять наличие ошибок. Если файл не удалось открыть, функция fopen() вернет NULL. В этом случае необходимо вывести сообщение об ошибке и завершить программу.

Пример программного кода для нахождения суммы первого и последнего числа в файле file1.txt:

#include <stdio.h>

int main() {

FILE\* file = fopen("file1.txt", "r");

if (file == NULL) {

printf("Ошибка открытия файлаn");

return 1;

}

int first, last, num;

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

while (fscanf(file, "%d", &num) != EOF) {

last = num;

}

fclose(file);

int sum = first + last;

printf("Сумма первого и последнего числа: %dn", sum);

return 0;

}

В данном примере мы открываем файл file1.txt для чтения. Если файл не удалось открыть, выводим сообщение об ошибке и завершаем программу. Затем мы считываем первое число из файла и сохраняем его в переменную first. Далее мы считываем числа из файла до тех пор, пока не достигнем конца файла. Последнее считанное число сохраняем в переменную last. Закрываем файл и выводим сумму первого и последнего числа.

# ДВОИЧНЫЙ/БИНАРНЫЙ

В Си файлы могут быть открыты в двух режимах: текстовом и двоичном. В текстовом режиме данные записываются в файл в виде символов, а в двоичном режиме - в виде байтов.

Для работы с файлами в Си используется структура FILE, которая содержит информацию о файле, такую как указатель на текущую позицию в файле, режим открытия файла и т.д.

Для открытия файла в Си используется функция fopen(), которая принимает два аргумента: имя файла и режим открытия. Режим открытия файла может быть "r" (чтение), "w" (запись) или "a" (добавление).

Для чтения данных из двоичного файла используется функция fread(), которая принимает четыре аргумента: указатель на буфер, размер каждого элемента, количество элементов и указатель на файл.

Для записи данных в двоичный файл используется функция fwrite(), которая принимает четыре аргумента: указатель на буфер, размер каждого элемента, количество элементов и указатель на файл.

Для закрытия файла используется функция fclose(), которая принимает указатель на файл.

При открытии файла необходимо проверять наличие ошибок. Если файл не удалось открыть, функция fopen() вернет NULL.

Пример программного кода для работы с двоичным файлом:

#include <stdio.h>

int main() {

FILE\* fp;

int nums[100];

int sum, temp;

// открытие файла для чтения и записи

fp = fopen("file1.dat", "rb+");

if (fp == NULL) {

printf("Ошибка при открытии файлаn");

return 1;

}

// чтение чисел из файла

fread(nums, sizeof(int), 100, fp);

// вычисление суммы первого и последнего чисел

sum = nums[0] + nums[99];

// обмен первого и последнего чисел

temp = nums[0];

nums[0] = nums[99];

nums[99] = temp;

// запись чисел в файл

fwrite(nums, sizeof(int), 100, fp);

// закрытие файла

fclose(fp);

printf("Сумма первого и последнего чисел: %dn", sum);

return 0;

}

В данном примере программа открывает файл file1.dat для чтения и записи в двоичном режиме. Затем она читает 100 чисел типа int из файла в массив nums, вычисляет сумму первого и последнего чисел, меняет местами первое и последнее число в массиве и записывает массив обратно в файл. Наконец, программа закрывает файл и выводит на экран сумму первого и последнего чисел.

# Указатели в Delphi

В Delphi указатели могут быть типизированными и бестиповыми. Типизированные указатели имеют тип, который определяет тип данных, на который они указывают. Бестиповые указатели не имеют типа и могут указывать на любой тип данных.

Для создания указателя на запись, состоящую из двух полей: фамилия и возраст, можно использовать следующий код:

type

TPerson = record

LastName: string;

Age: Integer;

end;

var

PersonPtr: ^TPerson;

Person: TPerson;

begin

New(PersonPtr);

PersonPtr^.LastName := 'Иванов';

PersonPtr^.Age := 25;

// Вывод значений полей записи в файл

AssignFile(Output, 'output.txt');

Rewrite(Output);

WriteLn(Output, 'Фамилия: ', PersonPtr^.LastName);

WriteLn(Output, 'Возраст: ', PersonPtr^.Age);

CloseFile(Output);

Dispose(PersonPtr);

end.

В этом коде мы создаем тип записи TPerson, который содержит два поля: LastName и Age. Затем мы создаем указатель на этот тип PersonPtr и выделяем память для него с помощью процедуры New. Затем мы присваиваем значения полям записи через оператор ^. Для вывода значений полей записи мы используем стандартный текстовый файл Output, который мы открываем с помощью процедуры AssignFile, перезаписываем с помощью процедуры Rewrite, выводим значения полей записи с помощью процедуры WriteLn и закрываем файл с помощью процедуры CloseFile. Наконец, мы освобождаем память, выделенную для указателя, с помощью процедуры Dispose.

Указатели могут быть использованы в качестве параметров процедур для передачи значений по ссылке. Например, можно создать процедуру, которая будет изменять значения полей записи через указатель:

procedure ChangePerson(var Person: TPerson);

begin

Person.LastName := 'Петров';

Person.Age := 30;

end;

var

PersonPtr: ^TPerson;

Person: TPerson;

begin

New(PersonPtr);

PersonPtr^.LastName := 'Иванов';

PersonPtr^.Age := 25;

ChangePerson(PersonPtr^);

// Вывод значений полей записи в файл

AssignFile(Output, 'output.txt');

Rewrite(Output);

WriteLn(Output, 'Фамилия: ', PersonPtr^.LastName);

WriteLn(Output, 'Возраст: ', PersonPtr^.Age);

CloseFile(Output);

Dispose(PersonPtr);

end.

В этом коде мы создаем процедуру ChangePerson, которая принимает параметр Person по ссылке и изменяет его значения. Затем мы создаем указатель PersonPtr и выделяем для него память. Мы присваиваем значения полям записи через оператор ^. Затем мы вызываем процедуру ChangePerson, передавая ей значение поля записи через указатель. После этого мы выводим значения полей записи в файл, как и раньше.

# Указатели в Си

Указатели в Си - это переменные, которые хранят адреса в памяти. Они используются для доступа к данным, которые хранятся в других переменных, а также для динамического выделения памяти.

Типизированные указатели - это указатели, которые указывают на определенный тип данных. Например, указатель на целое число будет иметь тип int \*. Бестиповые указатели - это указатели, которые не имеют типа данных. Они используются в основном для работы с памятью и преобразования типов данных.

Операция \* используется для доступа к значению, на которое указывает указатель. Операция & используется для получения адреса переменной.

Выделение памяти в Си происходит с помощью функции malloc(). Она выделяет блок памяти заданного размера и возвращает указатель на начало этого блока. Освобождение памяти происходит с помощью функции free(). Она принимает указатель на начало блока памяти и освобождает его.

Указатели могут использоваться в качестве параметров функции для передачи значений по ссылке. Это позволяет изменять значения переменных внутри функции и сохранять эти изменения после выхода из функции.

Пример программного кода:

#include <stdio.h>

#include <stdlib.h>

struct Person {

char name[50];

int age;

};

int main() {

struct Person\* person\_ptr;

person\_ptr = (struct Person\*)malloc(sizeof(struct Person));

// выделение памяти для структуры Person

if (person\_ptr == NULL) {

printf("Ошибка выделения памяти\n");

exit(1);

}

// присвоение значений полям структуры

strcpy(person\_ptr->name, "Иванов Иван");

person\_ptr->age = 25;

// вывод значений в файл

FILE\* file\_ptr;

file\_ptr = fopen("person.txt", "w");

fprintf(file\_ptr, "Фамилия: %sn", person\_ptr->name);

fprintf(file\_ptr, "Возраст: %dn", person\_ptr->age);

fclose(file\_ptr);

// освобождение памяти

free(person\_ptr);

return 0;

}

В этом примере мы создаем указатель на структуру Person, выделяем для нее память с помощью функции malloc(), присваиваем значения полям структуры, выводим эти значения в файл и освобождаем память с помощью функции free().

# Массивы в Delphi

# Одномерные

# Статистические и динамические

В Delphi есть два типа массивов: статические и динамические.

Статические массивы имеют фиксированный размер, который определяется при объявлении массива. Например, объявление статического массива из 10 элементов типа Integer будет выглядеть так:

var

MyArray: array[0..9] of Integer;

Динамические массивы, напротив, могут менять свой размер во время выполнения программы. Для создания динамического массива используется тип "открытый массив" (Open Array), который является типом с управляемым временем жизни. Например, объявление динамического массива типа Integer будет выглядеть так:

var

MyArray: array of Integer;

Расположение элементов массивов в памяти зависит от типа массива. Для статических массивов элементы располагаются последовательно в памяти, начиная с адреса, который соответствует первому элементу массива. Для динамических массивов элементы также располагаются последовательно в памяти, но адрес начала массива может меняться во время выполнения программы.

Для создания динамического массива с помощью стандартного типа "открытый массив" необходимо использовать функцию SetLength, которая устанавливает размер массива:

var

MyArray: array of Integer;

begin

SetLength(MyArray, 10); // создание массива из 10 элементов

end;

Для создания динамического массива с помощью самостоятельного распределения памяти с использованием процедур GetMem и FreeMem необходимо выполнить следующие шаги:

1. Выделить память под массив с помощью функции GetMem:

var

MyArray: PInteger;

begin

GetMem(MyArray, 10 \* SizeOf(Integer)); // выделение памяти под 10 элементов

end;

2. Использовать указатель MyArray для доступа к элементам массива:

MyArray[0] := 1;

MyArray[1] := 2;

// и т.д.

3. Освободить память, занятую массивом, с помощью процедуры FreeMem:

FreeMem(MyArray);

Важно помнить, что при использовании самостоятельного распределения памяти необходимо следить за тем, чтобы не произошло утечки памяти. Для этого необходимо всегда освобождать память, занятую массивом, после того, как он перестал быть нужным.

# Динамические одно- двух- мерные массивы

В Delphi массивы могут быть статическими и динамическими. Статические массивы имеют фиксированный размер, который определяется при объявлении массива. Динамические массивы, напротив, могут менять свой размер во время выполнения программы.

Для создания динамического одномерного массива в Delphi используется тип данных "открытый массив". Например, следующий код создает динамический массив целых чисел:

var

myArray: array of Integer;

begin

SetLength(myArray, 10); // выделение памяти для 10 элементов

myArray[0] := 1; // присваивание значения первому элементу

// ...

myArray[9] := 10; // присваивание значения последнему элементу

SetLength(myArray, 5); // изменение размера массива на 5 элементов

end;

Для создания двумерного массива в Delphi используется массив массивов. Например, следующий код создает двумерный массив целых чисел:

var

myArray: array of array of Integer;

begin

SetLength(myArray, 3); // выделение памяти для 3 строк

for i := 0 to 2 do

SetLength(myArray[i], 4); // выделение памяти для 4 столбцов в каждой строке

myArray[0][0] := 1; // присваивание значения первому элементу

// ...

myArray[2][3] := 12; // присваивание значения последнему элементу

end;

Расположение элементов массивов в памяти зависит от типа данных и размера массива. В Delphi элементы массива располагаются в памяти последовательно, начиная с первого элемента и заканчивая последним.

Для выделения памяти под динамический массив используется процедура SetLength. Для освобождения памяти, занятой динамическим массивом, используется процедура Finalize.

Двумерные массивы в Delphi могут иметь разное количество элементов в разных строках. Например, следующий код создает двумерный массив с разным количеством элементов в каждой строке:

var

myArray: array of array of Integer;

begin

SetLength(myArray, 3); // выделение памяти для 3 строк

SetLength(myArray[0], 2); // выделение памяти для 2 столбцов в первой строке

SetLength(myArray[1], 3); // выделение памяти для 3 столбцов во второй строке

SetLength(myArray[2], 4); // выделение памяти для 4 столбцов в третьей строке

myArray[0][0] := 1; // присваивание значения первому элементу

// ...

myArray[2][3] := 12; // присваивание значения последнему элементу

end;

Параметры процедур типа массив могут быть переданы по ссылке или по значению. Если параметр передается по ссылке, то изменения, внесенные в массив внутри процедуры, будут видны и за ее пределами. Если параметр передается по значению, то изменения, внесенные в массив внутри процедуры, не будут видны за ее пределами.

# Массивы в Си статические и динамические одно- двух- мерные массивы

В Си массивы могут быть статическими и динамическими. Статические массивы определяются при компиляции программы и имеют фиксированный размер. Динамические массивы создаются во время выполнения программы и их размер может быть изменен в процессе работы программы.

Одномерные массивы в Си объявляются следующим образом:

// Статический массив

int static\_array[10];

// Динамический массив

int\* dynamic\_array = malloc(10 \* sizeof(int));

Для выделения памяти под динамический массив используется функция malloc(), которая выделяет блок памяти заданного размера в байтах. В данном случае мы выделяем память под 10 элементов типа int.

Двумерные массивы в Си могут быть как статическими, так и динамическими. Расположение элементов двумерных массивов в памяти происходит построчно. То есть элементы первой строки идут подряд, затем элементы второй строки и т.д.

Статический двумерный массив объявляется следующим образом:

int static\_2d\_array[3][4];

В данном случае мы объявляем двумерный массив размером 3 на 4 элемента типа int.

Динамический двумерный массив можно создать следующим образом:

int\*\* dynamic\_2d\_array = malloc(3 \* sizeof(int\*));

for (int i = 0; i < 3; i++) {

dynamic\_2d\_array[i] = malloc(4 \* sizeof(int));

}

В данном случае мы выделяем память под 3 строки, каждая из которых содержит 4 элемента типа int.

Двухмерные массивы могут иметь разное количество элементов в разных строках. Например, такой массив:

1 2 3

4 5

6 7 8 9

можно объявить следующим образом:

int jagged\_array[3][];

jagged\_array[0] = (int[]){ 1, 2, 3 };

jagged\_array[1] = (int[]){ 4, 5 };

jagged\_array[2] = (int[]){ 6, 7, 8, 9 };

Описание параметров типа массив у функций выглядит следующим образом:

void my\_function(int my\_array[], int size) {

// ...

}

В данном случае мы объявляем функцию my\_function, которая принимает одномерный массив типа int и его размер. Если мы хотим передать в функцию двумерный массив, то нужно указать количество строк:

void my\_function(int my\_array[][4], int rows) {

// ...

}

В данном случае мы объявляем функцию my\_function, которая принимает двумерный массив типа int с 4 столбцами и количество строк.

Односвязный и двусвязный линейные списки

Список - это структура данных, которая представляет собой последовательность элементов, где каждый элемент содержит ссылку на следующий элемент. Односвязный список имеет только ссылку на следующий элемент, а двусвязный список имеет ссылки на следующий и предыдущий элементы.

Отличия массивов от списков:

- Массивы имеют фиксированный размер, в то время как списки могут динамически изменять свой размер.

- Добавление и удаление элементов в массиве может быть затратным по времени, так как требуется перемещение всех элементов после добавленного или удаленного элемента. В списке добавление и удаление элементов может быть быстрее, так как требуется только изменение ссылок на элементы.

- Доступ к элементам массива может быть быстрее, так как элементы хранятся в памяти последовательно. В списке доступ к элементам может быть медленнее, так как элементы могут храниться в разных местах памяти.

Описание типа "линейный список" на Си:

typedef struct node {

int data;

struct node\* next;

} Node;

typedef struct list {

Node\* head;

int size;

} List;

Здесь Node - это узел списка, который содержит данные и указатель на следующий узел. List - это сам список, который содержит указатель на головной узел и размер списка.

Си:

struct Node {

int data;

struct Node\* next;

};

struct Node\* head = NULL;

// Добавление элемента в начало списка

void addNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = head;

head = newNode;

}

// Удаление всех элементов списка с освобождением памяти

void deleteList() {

struct Node\* currentNode = head;

struct Node\* nextNode;

while (currentNode != NULL) {

nextNode = currentNode->next;

free(currentNode);

currentNode = nextNode;

}

head = NULL;

}

# СТЕК

Описание типа:

typedef struct Node {

int data;

struct Node\* next;

} Node;

typedef struct Stack {

Node\* top;

} Stack;

Допустимые операции:

-push(Stack \* stack, int data) - добавление элемента на вершину стека

- int pop(Stack \* stack) - удаление элемента с вершины стека и возврат его значения

- int peek(Stack \* stack) - возврат значения элемента на вершине стека без удаления

- int isEmpty(Stack \* stack) - проверка, пуст ли стек

Программный код для нахождения суммы всех положительных элементов стека:

int sumPositive(Stack\* stack) {

int sum = 0;

Node\* current = stack->top;

while (current != NULL) {

if (current->data > 0) {

sum += current->data;

}

current = current->next;

}

return sum;

}

Пример использования:

int main() {

Stack stack;

stack.top = NULL;

push(&stack, 5);

push(&stack, -2);

push(&stack, 3);

push(&stack, -7);

push(&stack, 8);

int sum = sumPositive(&stack);

printf("Sum of positive elements: %dn", sum);

return 0;

}

# Записи в Delphi, структуры и объединения в Си

Записи в Delphi и структуры в Си - это конструкции, которые позволяют объединять несколько переменных разных типов в одну структуру данных. Они имеют сходства и различия в синтаксисе описания типов и переменных.

В Delphi запись (record) определяется с помощью ключевого слова "record" и содержит набор полей, каждое из которых имеет свой тип. Например:

type

TPerson = record

Name: string;

Age: Integer;

Address: string;

end;

В этом примере мы определяем тип TPerson, который содержит три поля: Name типа string, Age типа Integer и Address типа string.

В Си структура (struct) определяется с помощью ключевого слова "struct" и содержит набор полей, каждое из которых имеет свой тип. Например:

struct Person {

char name[50];

int age;

char address[100];

};

В этом примере мы определяем тип Person, который содержит три поля: name типа char[50], age типа int и address типа char[100].

Объединения (union) в Си позволяют использовать одну область памяти для нескольких полей разных типов. Например:

union Data {

int i;

float f;

char str[20];

};

В этом примере мы определяем тип Data, который может содержать одно из трех полей: i типа int, f типа float или str типа char[20]. Объединение занимает столько же места в памяти, сколько занимает его самое большое поле.

В Delphi объединения не поддерживаются.

В обоих языках для создания переменной типа запись или структура используется следующий синтаксис:

var

Person1: TPerson;

struct Person person1;

Для доступа к полям записи или структуры используется оператор ".". Например:

Person1.Name := 'John';

strcpy(person1.name, "John");

В обоих языках доступ к полям объединения осуществляется также через оператор ".". Например:

union Data data;

data.i = 10;

В Delphi объединения не поддерживаются, поэтому доступ к полям объединения в Delphi невозможен.

Таким образом, записи в Delphi и структуры в Си имеют сходства в синтаксисе описания типов и переменных, но объединения в Си не имеют аналогов в Delphi.

# Язык Си

В языке Си управляющая структура "ветвление" реализуется с помощью оператора if-else. Синтаксис оператора if-else выглядит следующим образом:

if (условие) {

// блок кода, который выполнится, если условие истинно

} else {

// блок кода, который выполнится, если условие ложно

}

Также в языке Си есть оператор множественного выбора switch-case. Синтаксис оператора switch-case выглядит следующим образом:

switch (выражение) {

case значение1:

// блок кода, который выполнится, если выражение равно значению1

break;

case значение2:

// блок кода, который выполнится, если выражение равно значению2

break;

...

default:

// блок кода, который выполнится, если выражение не равно ни одному из значений

break;

}

Блок-схема оператора if-else:

+-------+

| условие |

+-------+

|

|

+---+---+

| истина |

+-------+

|

|

+-------+

| блок |

| кода |

+-------+

|

|

+---+---+

| ложь |

+-------+

|

|

+-------+

| блок |

| кода |

+-------+

Блок-схема оператора switch-case:

+-----------+

| выражение |

+-----------+

|

|

+---+---+---+---+

| значение1 | значение2 | ... | default |

+---+---+---+---+

| | | |

| | | |

+-------+ +-------+ +-------+ +-------+

| блок | | блок | | блок | | блок |

| кода | | кода | | кода | | кода |

+-------+ +-------+ +-------+ +-------+

В языке Си также есть оператор ?:, который позволяет сократить запись условного оператора if-else. Синтаксис оператора ?: выглядит следующим образом:

условие ? выражение1 : выражение2;

Оператор ?: можно использовать вместо оператора if-else, если нужно выполнить одно действие в зависимости от условия. Например:

int a = 5;

int b = (a > 3) ? 10 : 20; // b будет равно 10, так как a > 3

В языке Delphi управляющая структура "ветвление" реализуется с помощью оператора if-then-else и оператора case. Синтаксис оператора if-then-else выглядит следующим образом:

if условие then

// блок кода, который выполнится, если условие истинно

else

// блок кода, который выполнится, если условие ложно

Синтаксис оператора case выглядит следующим образом:

case выражение of

значение1: // блок кода, который выполнится, если выражение равно значению1

значение2: // блок кода, который выполнится, если выражение равно значению2

...

else // блок кода, который выполнится, если выражение не равно ни одному из значений

end;

Оператор ?: в языке Delphi отсутствует.

Операции отношения (сравнения) в языке Си и Delphi имеют одинаковый синтаксис и набор операторов. Операции отношения включают в себя:

- == (равно)

- != (не равно)

- < (меньше)

- > (больше)

- <= (меньше или равно)

- >= (больше или равно)

Например, в языке Си и Delphi можно написать следующий код:

int a = 5;

if (a > 3) {

// блок кода, который выполнится, если a больше 3

}

if (a != 10) {

// блок кода, который выполнится, если a не равно 10

}

Синтаксис циклов в языке C:

1. Цикл while:

while (условие) {

// тело цикла

}

2. Цикл do-while:

do {

// тело цикла

} while (условие);

3. Цикл for:

for (инициализация; условие; обновление) {

// тело цикла

}

Блок-схема цикла while:

Блок-схема цикла do-while:

Блок-схема цикла for:

В языке Delphi также есть аналоги циклов while, do-while и for.

Пример программного кода циклического алгоритма на языке C с использованием цикла while:

#include <stdio.h>

int main() {

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

int sum = 0, i = 0, n = 5;

float avg;

while (i < n) {

sum += arr[i];

i++;

}

avg = (float)sum / n;

printf("Среднее значение элементов массива: %.2f", avg);

return 0;

}

Пример программного кода циклического алгоритма на языке C с использованием цикла do-while:

#include <stdio.h>

int main() {

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

int sum = 0, i = 0, n = 5;

float avg;

do {

sum += arr[i];

i++;

} while (i < n);

avg = (float)sum / n;

printf("Среднее значение элементов массива: %.2f", avg);

return 0;

}

Пример программного кода циклического алгоритма на языке C с использованием цикла for:

#include <stdio.h>

int main() {

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

int sum = 0, n = 5;

float avg;

for (int i = 0; i < n; i++) {

sum += arr[i];

}

avg = (float)sum / n;

printf("Среднее значение элементов массива: %.2f", avg);

return 0;

}

Функции - это блоки кода, которые могут быть вызваны из других частей программы. Они могут принимать параметры и возвращать значения.

Передача параметров по значению означает, что копия значения параметра передается в функцию, и любые изменения, внесенные в этот параметр внутри функции, не будут влиять на оригинальное значение. Передача параметров по ссылке означает, что функция получает ссылку на оригинальный объект, и любые изменения, внесенные в этот объект внутри функции, будут отражены на оригинальном объекте.

Прототипы функций - это объявления функций, которые сообщают компилятору о том, какие функции будут использоваться в программе. Они содержат имя функции, типы ее параметров и тип возвращаемого значения.

Файлы заголовков - это файлы, которые содержат прототипы функций и объявления структур данных, которые используются в программе. Они обычно имеют расширение .h и включаются в исходный код программы с помощью директивы #include.

Модули - это файлы, которые содержат определения функций и переменных, которые используются в программе. Они могут быть скомпилированы в отдельные объектные файлы и связаны вместе для создания исполняемого файла.

Ниже приведен пример функции на языке C++, которая находит минимальное значение в двумерном массиве и возвращает его значение и индексы:

#include <iostream>

void findMin(int arr[][3], int rows, int cols, int& minValue, int& minRow, int& minCol);

int main() {

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

int minValue, minRow, minCol;

findMin(arr, 2, 3, minValue, minRow, minCol);

std::cout << "Minimum value: " << minValue << std::endl;

std::cout << "Minimum value row index: " << minRow << std::endl;

std::cout << "Minimum value column index: " << minCol << std::endl;

return 0;

}

void findMin(int arr[][3], int rows, int cols, int& minValue, int& minRow, int& minCol) {

minValue = arr[0][0];

minRow = 0;

minCol = 0;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

if (arr[i][j] < minValue) {

minValue = arr[i][j];

minRow = i;

minCol = j;

}

}

}

}

В этом примере функция findMin принимает двумерный массив arr, количество строк rows и количество столбцов cols, а также три ссылки на переменные minValue, minRow и minCol. Внутри функции происходит поиск минимального значения и его индексов, которые сохраняются в переменные, переданные по ссылке. После вызова функции значения этих переменных могут быть использованы в основной программе.

# **Процедурный тип** в Delphi и указатель на функцию в Си

Процедурный тип в Delphi и указатель на функцию в Си - это два разных способа определения типа данных, который представляет функцию.

В Delphi процедурный тип определяется с помощью ключевого слова "procedure" и может быть использован для определения переменных, параметров функций и типов данных записей. Пример определения процедурного типа в Delphi:

type

TMyProcType = procedure(param1: Integer; param2: String);

В Си указатель на функцию определяется с помощью знака "\*" и имени функции. Пример определения указателя на функцию в Си:

void (\*myFuncPtr)(int, char\*);

Использование имени функции в качестве параметра другой функции возможно как в Delphi, так и в Си. Это позволяет передавать функцию в качестве параметра другой функции и вызывать ее внутри этой функции. Пример использования имени функции в качестве параметра другой функции в Delphi:

type

TMyProcType = procedure(param1: Integer; param2: String);

procedure CallMyProc(myProc: TMyProcType);

begin

myProc(42, 'Hello');

end;

procedure MyProc(param1: Integer; param2: String);

begin

// do something

end;

begin

CallMyProc(MyProc);

end;

Пример использования имени функции в качестве параметра другой функции в Си:

void callMyFunc(void (\*myFunc)(int, char\*)) {

myFunc(42, "Hello");

}

void myFunc(int param1, char\* param2) {

// do something

}

int main() {

callMyFunc(myFunc);

return 0;

}