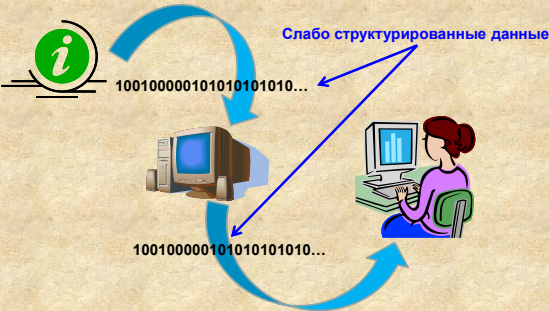


Тема 4

Структуры данных
(часть 1)

Понятие структуры данных



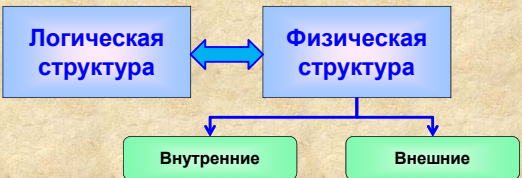
Черников Б.В.

2

Структура данных – множество элементов данных и множество связей между ними

Физическая структура данных отражает способ физического представления данных в памяти машины и называется еще структурой хранения, внутренней структурой или структурой памяти

Структура данных без учета ее представления в машинной памяти – абстрактная или логическая структура



Черников Б.В.

3

Классификация структур данных



Черников Б.В.

4

Элементарные – не могут быть расчленены на составные части, большие, чем биты

Составные – структуры данных, составными частями которых являются другие структуры данных (элементарные или, в свою очередь, составные)

Важный признак составной структуры данных – характер упорядоченности ее частей

- ♦ линейные
- ♦ нелинейные

Изменчивость – изменение числа элементов или связей между составными частями структуры

- ♦ статические
- ♦ динамические

Черников Б.В.

5

Типы данных

Структуры данных

Структура хранения данных указанного типа – выделение памяти, представление данных в ней и метод доступа к данным

Множество допустимых значений, которые может иметь тот или иной объект описываемого типа

Набор допустимых операций, которые применимы к объекту описываемого типа

Черников Б.В.

6

ЭЛЕМЕНТАРНЫЕ ДАННЫЕ

- Представляют собой **единое и неделимое** целое
- В каждый момент времени они могут принимать **только одно** значение

```
var
i, j: integer; x: real; s: char; b: boolean; p: pointer;
```

Целочисленный Вещественный Символьный Логический Указатель

```
i := 46; x := 3.14; s := 'A'; b := true;
```

Черников Б.В.

7

Данные целочисленного типа

Представляется счетное число объектов, являющихся дискретными по своей природе

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать 1, 2 или 4 байта

Тип	Диапазон значений	Машинное представление
shortint	-128 ... 127	8 бит со знаком
integer	-32768 ... 32767	16 бит со знаком
longint	-2147483648 ... 2147483647	32 бита со знаком
byte	0 ... 255	8 бит без знака
word	0 ... 65535	16 бит без знака
comp	$-2^{63}+1$... $2^{63}-1$	64 бита со знаком

Значения целочисленных типов представляются в памяти ЭВМ **абсолютно точно**

Черников Б.В.

8

Значения вещественных типов определяют число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа

Тип	Диапазон значений	Значимые цифры	Размер в байтах
real	$2,9 \cdot 10^{-39} \dots 1,7 \cdot 10^{38}$	11-12	6
single	$1,4 \cdot 10^{-45} \dots 3,4 \cdot 10^{38}$	7-8	4
double	$4,9 \cdot 10^{-324} \dots 1,8 \cdot 10^{308}$	15-16	8
extended	$3,1 \cdot 10^{-4944} \dots 1,2 \cdot 10^{4932}$	19-20	10

Операции над числовыми типами

Основные операции	Арифметические операции
<ul style="list-style-type: none"> создание уничтожение выбор обновление 	<ul style="list-style-type: none"> сложение вычитание умножение деление
операции сравнения $> < \geq \leq <=>$	
Для целочисленных операндов	
<ul style="list-style-type: none"> целочисленное деление div остаток от деления mod 	

Черников Б.В.

9

Данные символьного типа

Тип **char** (1 байт) – символы из некоторого **предопределенного множества** (ASCII – American Standard Code for Information Interchange)

Одна таблица может поддерживать только **один национальный алфавит**

Этот недостаток преодолен во множестве **UNICODE** – каждый символ кодируется двумя байтами, что обеспечивает **более 2^{16}** возможных кодовых комбинаций и дает возможность иметь **единую таблицу кодов**

Операции над символьными типами – **только операции сравнения**

Черников Б.В.

10

Данные логического типа

Значениями логического типа может быть одна из предварительно объявленных констант **false** (ложь) или **true** (истина)

Данные логического типа занимают **один байт** памяти:

- значению **false** соответствует **нулевое** значение байта
- значению **true** – любое **ненулевое** значение байта

Над логическими типами – операции булевой алгебры: **НЕ** (not), **ИЛИ** (or), **И** (and), **ИСКЛЮЧАЮЩЕЕ ИЛИ** (xor)

Результаты логического типа получаются при сравнении данных любых типов

Черников Б.В.

11

Данные типа указатель

Тип указателя представляет собой **адрес ячейки памяти**

В языках программирования высокого уровня определена специальная константа **nil**, которая означает **пустой указатель** или указатель, **не содержащий какой-либо конкретный адрес**

Указатели применяются:

- при необходимости представить **одну и ту же** область памяти (одни и те же физические данные) как **данные разной логической структуры**
- при работе с **динамическими структурами данных**

Могут быть **типируемыми** и **нетипируемыми**

```
var
ipt: ^integer;
cpt: ^char;
```

Черников Б.В.

12

Основные операции:

- ♦ **Присваивание** – двухместная операция, оба операнда – **указатели**
Если оба указателя, участвующие в операции присваивания типизированные, то оба должны указывать на данные одного и того же типа
- ♦ **Получение адреса** – **одноместная**, ее операнд может иметь **любой тип**
Результат – **типизированный указатель** (в соответствии с типом операнда), содержащий адрес операнда
- ♦ **Выборка** – **одноместная**, ее операндом является **типизированный указатель**
Результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется **типом указателя**

Черников Б.В.

13

ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

Цель описания типа данных и определения некоторых переменных, относящихся к статическим типам, состоит в том, чтобы **зафиксировать диапазон значений, присваиваемых этим переменным, и соответственно размер выделяемой для них памяти**

Поэтому такие переменные – **статические**

Черников Б.В.

14

Массив

Массив – **поименованная совокупность однотипных элементов, упорядоченных по индексам, определяющих положение элемента в массиве**

Задание имени массива, типа индекса и типа элементов:

имя: `array[ТипИндекса] of ТипЭлемента;`

Массив бывает **одномерным** (вектор), **двумерным** (матрица), **трехмерным** (куб)...

`var`

`Vector: array [1..100] of integer;`

`Matrix: array [1..100, 1..100] of integer;`

`Cube: array [1..100, 1..100, 1..100] of integer;`

Черников Б.В.

15

Операции:

- ♦ **Сравнение на равенство и неравенство массивов**
- ♦ **Присвоение для массивов с одинаковым типом индексов и одинаковым типом элементов**

Доступ к массивам в этих операциях – через имя массива без указания индексов

Можно выполнять операции над **отдельными элементами** массива
Перечень таких операций определяется **типом элементов**
Доступ к отдельным элементам массива осуществляется **через имя массива и индексы элементов**:

`Cube[0,0,10] := 25;`

`Matrix[10,30] := Cube[0,0,10] + 1;`

Элементы массива обычно располагаются **непрерывно**, в соседних ячейках.

Размер памяти, занимаемой массивом, есть **суммарный размер элементов массива**

Черников Б.В.

16

Строка

Строка – **последовательность символов (элементов символьного типа)**

Количество символов в строке (длина строки) может динамически меняться от 0 до 255:

`var`

`TTxt: string;`

`TWrd: string[10];`

Каждый символ строки имеет свой индекс: от 1 до заданной длины строки

Элемент строки **с индексом 0 недоступен** с использованием индекса и содержит **текущее количество символов** в строке

Черников Б.В.

17

Операции:

- ♦ **Определенные для символьного типа данных**
- ♦ **Присвоение строк в целом**
- ♦ **Операции сравнения строк <, >, ≥, ≤, =, <>**
- ♦ **Операция сцепления (конкатенации) строк +**

Символы строки располагаются **непрерывно**, в соседних ячейках

Размер памяти, занимаемой массивом, есть **суммарный размер элементов массива (включая элемент, содержащий длину строки)**

Черников Б.В.

18

Запись

Запись – агрегат, составляющие которого (поля) имеют имя и могут быть различного типа

type

```
TPerson = record
    Name: string;
    Address: string;
    Index: longint;
end;
```

var Person1: TPerson;

Операция **присваивания** для записей в целом – записи должны быть одного типа)

Можно выполнять операции **над отдельным полем**

```
Person1.Index := 117997;
Person1.Name := 'РЭУ им. Плеханова';
Person1.Address := 'Москва, Стремянный переулок, д.36';
```

Черников Б.В.

19

Множество

Множество – совокупность **однородных** элементов, объединенных **общим признаком** и представляемых как **единое целое**

Операции:

- ♦ Объединение множеств +
- ♦ Пересечение множеств *
- ♦ Разность множеств –
- ♦ Проверка элемента на принадлежность к множеству in

var

RGB, YIQ, CMY: set of char;

CMY := ['M', 'C', 'Y'];

RGB := ['R', 'G', 'B'];

YIQ := ['Y', 'Q', 'I'];

WriteLn('Пересечение цветовых схем RGB и CMY ', RGB*CMY);

WriteLn('Пересечение цветовых схем YIQ и CMY ', YIQ*CMY);

Черников Б.В.

20

Таблица

Таблица – одномерный массив (вектор), элементами которого являются **записи**

Отдельная запись массива – **строка** таблицы

Чаще всего используется **простая запись**, т. е. поля – **элементарные данные**

Совокупность одноименных полей всех строк – **столбец**, а конкретное поле отдельной строки – **ячейка**

type

```
TPerson = record
    Name: string;
    Address: string;
    Index: longint;
```

end;

TTable = array[1..1000] of TPerson;

var PersonList: TTable;

Черников Б.В.

21

Доступ к элементам таблицы производится не по индексу, а по **ключу**, т. е. по значению одного из полей записи

Ключ таблицы (основной, первичный) – поле, значение которого может быть использовано для **однозначной идентификации** каждой записи таблицы

Вторичный ключ – поле таблицы с несколькими ключами, не обеспечивающий (в отличие от первичного ключа) однозначной идентификации записей таблицы

Если последовательность записей упорядочена относительно какого-либо столбца (поля), то такая таблица называется **упорядоченной**, иначе – таблица **неупорядоченная**

Черников Б.В.

22

Основная операция – доступ к записи по ключу

Перечень операций над отдельной ячейкой определяется типом ячейки

```
PersonList[1].Index := 117997;
PersonList[1].Name := 'РЭУ им. Плеханова';
PersonList[1].Address := 'Москва, Стремянный переулок, д.36';
```

Ячейки таблицы обычно располагаются **построчно, непрерывно, в соседних ячейках**
Размер памяти, занимаемой таблицей, есть **суммарный размер ячеек**

Черников Б.В.

23

Линейные списки

Список – структура данных, представляющая собой **логически связанную последовательность** элементов списка

Динамическая структура данных – структура данных, определяющие характеристики которой могут изменяться на протяжении ее существования

Отличие динамических структур от статических:

- ♦ в них нельзя обеспечить хранение в заголовке всей информации о структуре, поскольку каждый элемент должен содержать информацию, логически связывающую его с другими элементами структуры
- ♦ для них зачастую неудобно использовать единый массив смежных элементов памяти, поэтому необходимо предусматривать какую-либо схему динамического управления памятью

Процедура создания
New(Current);

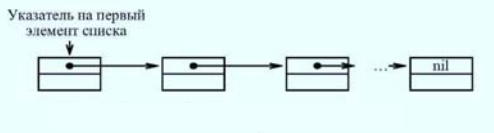
Процедура освобождения
Dispose(Current);

Черников Б.В.

24

Линейный однонаправленный список

В этом списке любой элемент имеет **один указатель**, который указывает на **следующий элемент** в списке или является **пустым указателем** у последнего элемента



При выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя **на первый элемент**

В противном случае часть или весь список будет недоступен

Операции:

- вставка элемента
- просмотр
- поиск
- удаление элемента

type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
Data: TypeData; {поле данных элемента}
Next: PElement; {поле указателя на следующий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}

Процедуры вставки InsFirst_LineSingleList и Ins_LineSingleList	
Входные параметры	Выходные параметры
<ul style="list-style-type: none">данные для заполнения создаваемого элементауказатель на начало спискауказатель на текущий элемент в списке (при необходимости)	<ul style="list-style-type: none">указатель на начало спискауказатель текущего элемента (показывает на вновь созданный элемент, при вставке первого элемента указателем на него будет указатель на заголовок списка)

```
procedure Ins_LineSingleList(DataElem: TypeData;  
var ptrHead, ptrCurrent: PElement);  
{Вставка непервого элемента в линейный однонаправленный список}  
{справа от элемента, на который указывает ptrCurrent}  
var ptrAddition: PElement; {вспомогательный указатель}  
begin  
    New(ptrAddition);  
    ptrAddition^.Data := DataElem;  
    if ptrHead = nil then  
        begin {список пуст, создаем первый элемент списка}  
            ptrAddition^.Next := nil;  
            ptrHead := ptrAddition;  
        end  
    else  
        begin  
            {список не пуст, вставляем элемент списка справа от  
            элемента, на который указывает ptrCurrent}  
            ptrAddition^.Next := ptrCurrent^.Next;  
            ptrCurrent^.Next := ptrAddition;  
        end;  
    ptrCurrent := ptrAddition;  
end;
```

```
procedure InsFirst_LineSingleList(DataElem: TypeData;  
var ptrHead: PElement);  
{Вставка первого элемента в линейный однонаправленный список}  
var ptrAddition: PElement; {вспомогательный указатель}  
begin  
    New(ptrAddition);  
    ptrAddition^.Data := DataElem;  
    if ptrHead = nil then  
        begin {список пуст, создаем первый элемент списка}  
            ptrAddition^.Next := nil;  
        end  
    else  
        begin {список не пуст - вставляем новый элемент слева  
        (перед) первым элементом}  
            ptrAddition^.Next := ptrHead;  
        end;  
    ptrHead := ptrAddition;  
end;
```

При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку

Операция просмотра списка

```
procedure Scan_LineSingleList(ptrHead: PElement);  
{Просмотр линейного однонаправленного списка}  
var ptrAddition: PElement; {вспомогательный указатель}  
begin  
    ptrAddition := ptrHead;  
    while ptrAddition <> nil do  
        begin {пока не конец списка}  
            writeln(ptrAddition^.Data); {Вывод значения элемента}  
            ptrAddition := ptrAddition^.Next;  
        end;  
    end;  
end;
```

Операция поиска элемента в списке

Входные параметры	Выходной параметр
<ul style="list-style-type: none">значение, которое должен содержать искомый элементуказатель на список	<ul style="list-style-type: none">указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет

```
function Find_LineSingleList(DataElem: TypeData;  
var ptrHead, ptrCurrent: PElement): boolean;  
{Поиск элемента в линейном однонаправленном списке}  
var ptrAddition: PElement; {Дополнительный указатель}  
begin  
    if (ptrHead <> nil) then  
        begin {Если существует список}  
            ptrAddition := ptrHead;  
            while (ptrAddition <> nil) and (ptrAddition^.Data <> DataElem) do  
                {пока не конец списка и не найден элемент}  
                ptrAddition := ptrAddition^.Next;  
            {Если элемент найден, то результатом работы функции  
            является true}  
            if ptrAddition <> nil then  
                begin  
                    Find_LineSingleList := true;  
                    ptrCurrent := ptrAddition;  
                end  
            else  
                begin Find_LineSingleList := false; end;  
            end  
        end  
    else  
        begin Find_LineSingleList := false; end;  
    end;  
end;
```

Операция удаления элемента

```

procedure Del_LineSingleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из линейного однонаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
    if ptrCurrent <> nil then
        begin {вх.параметр корректен}
            if ptrCurrent = ptrHead then
                begin {удаляем первый}
                    ptrHead := ptrHead^.Next;
                    dispose(ptrCurrent);
                    ptrCurrent := ptrHead;
                end
            else
                begin {удаляем непервый}
                    {устанавливаем вспомогательный указатель на элемент, предшествующий
                    удаляемому}
                    ptrAddition := ptrHead;
                    while ptrAddition^.Next <> ptrCurrent do
                        ptrAddition := ptrAddition^.Next;
                    {непосредственное удаление элемента}
                    ptrAddition^.Next := ptrCurrent^.Next;
                    dispose(ptrCurrent);
                    ptrCurrent := ptrAddition;
                end
            end;
        end;
    end;
end;

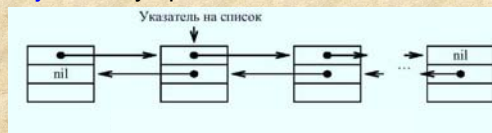
```

Черников Б.В.

31

Линейный двунаправленный список

В этом списке любой элемент имеет **два указателя**, один из которых указывает на **следующий элемент** в списке или является **пустым указателем** у последнего элемента, а второй – на **предыдущий элемент** в списке или является **пустым указателем** у первого элемента



В отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как **благодаря двум указателям** в элементах можно получить **доступ к любому элементу списка из любого другого элемента**, осуществляя переходы в прямом или обратном направлении

Черников Б.В.

32

Операции:

- вставка элемента
- просмотр
- поиск
- удаление элемента

```

type
    PElement = ^TypeElement; {указатель на тип элемента}
    TypeElement = record {тип элемента списка}
        Data: TypeData; {поле данных элемента}
        Next: PElement; {поле указателя на следующий элемент}
        Last: PElement; {поле указателя на предыдущий элемент}
    end;
var
    ptrHead: PElement; {указатель на первый элемент списка}
    ptrCurrent: PElement; {указатель на текущий элемент}

```

Операция вставки реализуется с помощью двух процедур, аналогичных процедурам вставки для линейного однонаправленного списка:

InsFirst_LineDubleList и **Ins_LineDubleList**

Однако при вставке последующего элемента придется учитывать особенности добавления элемента в конец списка

Черников Б.В.

33

```

procedure Ins_LineDubleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);
{Вставка нового элемента в линейный двунаправленный список}
{справа от элемента, на который указывает ptrCurrent}
var ptrAddition: PElement; {вспомогательный указатель}
begin
    New(ptrAddition);
    ptrAddition^.Data := DataElem;
    if ptrHead = nil then
        begin {список пуст - создаем первый элемент списка}
            ptrAddition^.Next := nil;
            ptrAddition^.Last := nil;
            ptrHead := ptrAddition;
        end
    else
        begin
            {список не пуст - вставляем элемент списка справа от элемента,
            {на который указывает ptrCurrent}
            if ptrCurrent^.Next <> nil then {вставляем не последний}
                ptrCurrent^.Next^.Last := ptrAddition;
                ptrAddition^.Next := ptrCurrent^.Next;
                ptrCurrent^.Next := ptrAddition;
                ptrAddition^.Last := ptrCurrent;
            end;
            ptrCurrent := ptrAddition;
        end;
    end;
end;

```

Черников Б.В.

34

```

procedure InsFirst_LineDubleList(DataElem: TypeData; var ptrHead: PElement);
{Вставка первого элемента в линейный двунаправленный список}
var ptrAddition: PElement; {вспомогательный указатель}
begin
    New(ptrAddition);
    ptrAddition^.Data := DataElem;
    ptrAddition^.Last := nil;
    if ptrHead = nil then
        begin {список пуст - создаем первый элемент списка}
            ptrAddition^.Next := nil;
        end
    else
        begin
            {список не пуст - вставляем новый элемент слева (перед) первым
            элементом}
            ptrAddition^.Next := ptrHead;
            ptrHead^.Last := ptrAddition;
        end;
    ptrHead := ptrAddition;
end;

```

Черников Б.В.

35

Операция удаления элемента

```

procedure Del_LineDubleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из линейного двунаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
    if ptrCurrent <> nil then
        begin {входной параметр корректен}
            if ptrCurrent = ptrHead then
                begin {удаляем первый}
                    ptrHead := ptrHead^.Next;
                    dispose(ptrCurrent);
                    ptrHead^.Last := nil;
                    ptrCurrent := ptrHead;
                end
            else
                begin {удаляем непервый}
                    if ptrCurrent^.Next = nil then
                        begin {удаляем последний}
                            ptrCurrent^.Last^.Next := nil;
                            dispose(ptrCurrent);
                            ptrCurrent := ptrHead;
                        end
                    else
                        begin {удаляем не последний и непервый}
                            ptrAddition := ptrCurrent^.Next;
                            ptrCurrent^.Last^.Next := ptrAddition^.Next;
                            ptrAddition^.Last := ptrCurrent^.Last;
                            dispose(ptrCurrent);
                            ptrCurrent := ptrAddition;
                        end
                    end;
                end;
            end;
        end;
    end;
end;

```

Черников Б.В.

36