Теоретический раздел

## Раздел 1. Подпрограммы (модуль)

1.1. Общие сведения

В программировании часто надо **повторять отдельные участки вычислений несколько раз** в различных точках программы при разных исходных данных. Чтобы не повторять одни и те же операторы неск-ко раз, используют **подпрограммы.**

**Подпрограмма** – именованная группа операторов языка логически законченная, которую вызывают **по имени** для выполнения любое количество раз из разных мест программы.

**Программный модуль** состоит: заголовок**+** тело модуля. Тело модуля называют **блоком**. А блок состоит из разделов: описаний и операторов.

Блок может содержать в себе другие блоки. Блок, который не входит ни в какой другой блок, называется **глобальным** – это основная программа (программный модуль). Блоки, входящие в состав других блоков – **локальные** или **вложенные** – это **подпрограммы**.

=> все элементы программы (типы, переменные, константы и т.д.) делят на **глобальные** и **локальные**.

Л**окальное** **объявление** для подпрограммы - если оно содержится в данной подпрограмме, а **глобальное –** если оно в объемлющей подпрограмме или модуле.

С каждым объявлением **имени** (- **идентификатора**) объекта программы связана **область** его действия. Вне своей области действия имя не существует.

**Область действия имени** - часть модуля или блока от точки объявления и до конца текущего **уровня вложенности**, включая вложенные блоки, кроме тех вложенных подпрограмм, где есть другие объявления этого же имени. Не допускаются повторные объявления имени на 1ом уровне вложенности.

хороший стиль программирования - **min** глобальных переменных. В идеале программа состоит только из подпрограмм, которые вызываются для выполнения из раздела операторов основной программы.

**2 вида подпрограмм** – **процедуры** и **функции**, которые делят на 2 группы– **встроенные** и **определенные** прогером.

1. **Встроенные (стандартные) процедуры и функции -** часть языка и используются без предварительного описания в разделе объявлений блока.

**2. Процедуры и функции** определенные прогером - локальные блоки. Их предварительное описание обязательно.

1.2. Процедуры

1.2.1. Описание процедур

Процедура должна быть **описана** в разделе объявления процедур и функций программы.

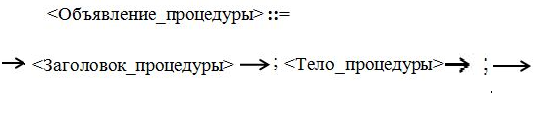
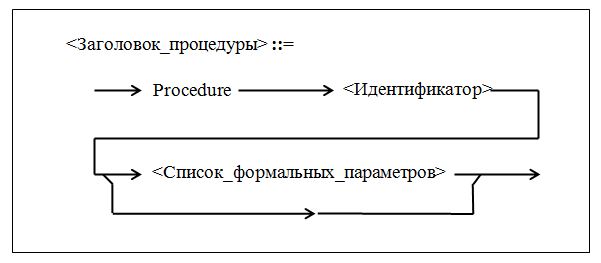


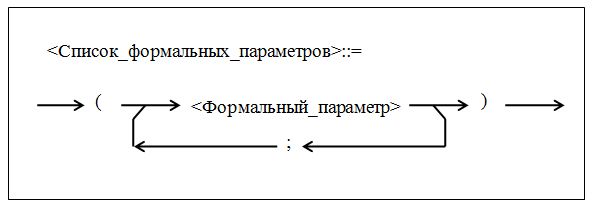
Рис 1.1 – Синтаксическая диаграмма объявления процедуры:

Состоит: **заголовок + тело.**

1) **Заголовок** определяет ее **имя** (идентиф-р)и **список** **форм**альных параметров: 

**Рис 1.2 – Синтаксич** диаграмма объявления заголовка процедуры

Формальные **параметры**: определяют **тип** данных (передаваемых процедуре при ее вызове) и **способ** **передачи** данных. Список формальных параметров (рис 1.3):

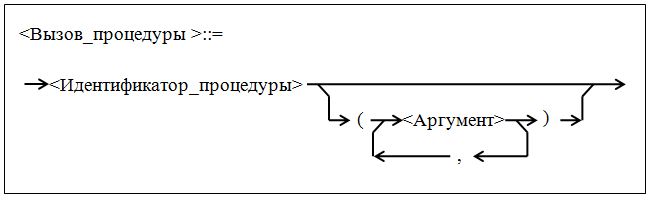


**Рис 1.3 –** **Синтаксич** диаграмма списка формальных параметров

2) **Тело** процедуры состоит: **объявления** идентификаторов (локальных для нее) + **составной оператор begin … end;** (- описывает действия процедуры).

1.2.2. Вызов процедур

– с помощью **оператора** вызова процедуры, имеющего формат (рис 1.4):

 Рис 1.4 – Синтаксич диаграмма оператора вызова процедуры

Оператор вызывает процедуру с указанным именем и передает ей **фактические** параметры (**аргументы**), если они заданы.

Фактические параметры - могут быть использованы выражения типа, совместимого с типом формальных параметров. Количество аргументов д б = кол-ву формальных параметров: первому формальному параметру соответствует1й аргумент, второму –2й и т.д.

Конец выполнения процедуры – достижение конца составного оператора тела процедуры (**End*).*** Чтобы прекратить выполнение процедуры до достижения конца ее тела - встроенная процедура **Exit**.

В обоих случаях – выход из процедуры и возврат к точке ее вызова, т.е. к оператору (в вызывающей программе) **после** оператора вызова процедуры. Такое выполнение процедуры называется **синхронным**.

1.2.3. Процедуры без параметров

Если в операторе вызова процедуры и в заголовке процедуры **нет** параметров, то связь по данным между вызывающей и вызываемой процедурами - через **глобальные** переменные **(одноименные)**: их область действия охватывает обе процедуры.

**Пример 1.1.** Вычислить расстояние между ***N*** парами точек на плоскости. Использовать процедуры **без параметров.**

**Program R1;**

**Var**

**X1, Y1, X2, Y2, D: Real; {Объявление глобальных переменных}**

**N, I: Integer;**

**Procedure Rast1; {=== Процедура без параметров: ========}**

**Begin**

***D:= Sqrt (Sqr (X1 - X2) + Sqr (Y1 - Y2))*** {D– расст м/у точками: X1, Y1, X2, Y2 – координаты 1й и 2й т-к}

**End;**

**Begin {============= Вызывающая программа =========}**

**Writeln (**‘Введите количество пар точек’**); Readln (N);**

**For I:= 1 To N Do**

**Begin**

**Writeln (**‘Вв координаты 1й точки’**); Readln (X1, Y1);**

**Writeln** (‘Вв координаты 2й точки’); **Readln (X2, Y2);**

**Rast1; { - Оператор вызова процедуры Rast1}**

**Writeln (D);**

**End;**

**End.**

Здесь связь между вызывающей программой и процедурой **Past1 -** через **глобальные** переменные ***X1, X2, Y1, Y2, D***.

**«-»процедур без параметров** – жесткая фиксация исходных данных: eсли процедуру без параметров использовать в разных точках вызывающей программы для разных имен переменных, то **перед вызовом процедуры надо переприсвоить имена** – имена переменных должны соответствовать именам в теле процедуры.

Для устранения этого используют процедуры **с** параметрами.

1.2.4. Процедуры с параметрами

Использование параметров – д/удобства использования процедур, чтобы не фиксировать **исходные значения** (-конкретизируют при каждом обращении к процедуре), к которым применяется процедура.

**Формальные параметры** – идентификаторы **переменных**, имена подпрограмм; используют в операторах внутри данной подпрограммы, **заменяются аргументами при ее вызове** на фактические. Задание формальных параметров:

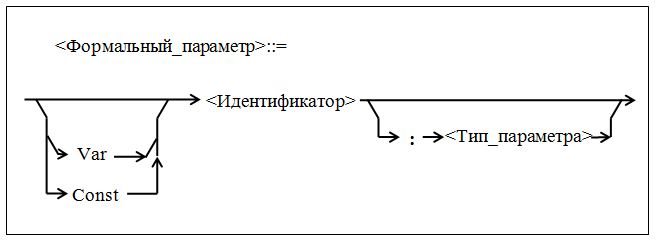


Рис 1.5 − Синтаксическая диаграмма задания формальных параметров

5 **типов параметров**:

1. параметры-значения П-З;
2. параметры-переменные П-П;
3. параметры-константы П-К;
4. параметры без типа б/Т;
5. параметры процедурного типа.

1.2.5 Параметры-значения (П-З)

Procedure Rast2 (Xx1, Xx2, Yy1, Yy2: Real);

– это группа параметров, **перед** ними **нет** ***Var*** или ***Const***, а сразу **тип**, кроме **файловых** типов.

Для каждого П-З при вызове подпрограммы создают **локальные** переменные, которые существуют только во время выполнения подпрограммы, после чего они прекращают существование. При входе в подпрограмму этим переменным (используют при выполнении подпрограммы) присваивают значения соответствующих **фактич**еских параметров вызова подпрограммы.

**Фактич**еский параметр - выражение любого типа, совместимое по присваиванию с типом П-З.

При применении П-З **фактич**еские параметры используют только при входе в подпрограмму д/передачи задаваемых ими значений соответствующим **локальным переменным**. После этого **фактич**еские параметры становятся недоступны из подпрограммы. Поэтому подпрограмма не может изменить значения переменной, являющейся фактическим параметром.

Изменение П-З в теле подпрограммы не влияет на значения соответствующих переменных вызывающей программы. Поэтому с помощью П-З нельзя представлять результаты выполнения подпрограммы, которые должны использоваться вне ее тела (в основной части программы).

**Пример 1.2.** Вычислить расстояние между ***N*** парами точек на плоскости. Использовать процедуры с П-З.

**Var**

**X1, Y1, X2, Y2, D: Real;** {Объявление **глобальных** переменных}

**N, I: Integer;**

**Procedure Rast2 (Xx1, Xx2, Yy1, Yy2: Real);** {Xx1, ..,Yy2–пар-ры-знач}

**Begin**

**D:=Sqrt (Sqr (Xx1-Xx2) + Sqr (Yy1-Yy2))** {D – глобальна перем-ая}

**End;**

**Begin {==== Вызывающая программа ========}**

**Writeln (‘Введите количество пар точек’); Readln (N);**

**For I:= 1 To N Do**

**Begin**

**Readln (X1, X2, Y1, Y2); - фактические пар-ры?**

**Rast2 (X1, X2, Y1, Y2); {Оператор вызова процедуры Rast2}**

**Writeln (D);**

**End;**

**End.**

для вычисления расстояния между точками – процедура **Rast2** с П-З ***Xx1, Xx2, Yy1, Yy2***. Для представления результата работы процедуры - глобальная переменная ***D***, т к данный результат должен быть передан в вызывающую прогу.

1.2.6. Параметры-переменные (П-П) – с var

– группа параметров, перед которой стоит **Var** и за которой следует **тип**. **Фактич**еский параметр - только переменная того же типа (но не константа и не выражение), что и тип **формального** параметра. В подпрограмму передается адрес этой переменной. Поэтому подпрограмма может напрямую использовать и изменять значение этой переменной и передавать в основную программу вырабатываемые ею результаты.

Т о, изменение формального П-П приводит к изменению соответствующего **фактическ**ого параметра.

**Пример 1.3.** Вычислить расстояние между ***N*** парами точек на плоскости. Использовать процедуры с П-П.

Var

X1, X2, Y1, Y2, D: Real; {Объявление глобальных переменных}

I: Interger;

**Const**

**N=10;** {Количество пар точек на плоскости}

**{==== Процедура с П-З** и **П-П =========}**

**Procedure Rast3 (Xx1, Xx2, Yy1, Yy2: Real; Var Dd: Real);** {Xx1,…, Yy2

– **П-З** типа Real**; Dd – П-П**}

**Begin**

**Dd := Sqrt (Sqr (Xx1 - Xx2) + Sqr (Yy1 - Yy2))**

**End;**

**Begin{==== Вызывающая программа ========}**

**For I:=1 To N Do**

**Begin**

**Readln (X1, X2, Y1, Y2);**

**Rast3 (X1, X2, Y1, Y2, D); {Оператор вызова процедуры Rast3}**

**Writeln (D)**

**End;**

**End.**

Отличие этой проги от предыдущей: для представления результата работы процедуры используется П-П ***Dd***.

1.2.7. Параметры-константы (П-К) const

- группа параметров, перед которой стоит зарезервированное слово **Const**и за ним – тип. При их исп в подпрогу передается адрес **фактич**еского пар-ра. Нельзя изменять значения П-К в теле подпрограммы, иначе - ошибка (в отличие от П-З: её изменение в теле подпрограммы возможно, но передать их в вызывающую программу нельзя). Причина: при входе в подпрограмму локальные переменные, соответствующие П-К-ам, не создаются.

Поэтому подпрограммы с П-К выполняются быстрее, чем с П-З. Т о, в тех подпрограммах, где не надо изменять значений параметров, лучше иметь П-К, чем П-З.

Т к в примере 1.3 в теле процедуры значения ее формальных параметров не меняются, то заголовок процедуры может быть описан так:

**Procedure Rast4 (Const Xx1, Xx2, Yy1, Yy2: Real; Var Dd: Real);**

1.2.8. Параметры-переменные без типа (П-П б/Т)

— группа параметров, перед которыми стоит **Var** и за которыми нет тип. **Фактическ**ие параметры здесь – переменные **любого типа**.

Поскольку у **нетипизованных параметров-переменных** нет типа, то изначально они несовместимы с переменными всех типов.

Совместимость с **фактичес**кими параметрами м б 2 *способами*:

**СПОСОБ 1.**

Внутри подпрограммы **объявляется локальная переменная нужного типа*,* налагаемая на переданный фактический параметр*.*** Для описания такой переменной – зарезервиров-ное слово **Absolute**.

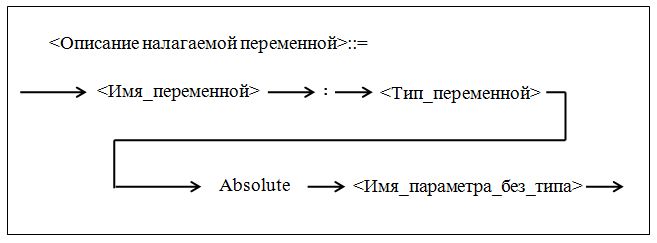


Рис 1.6 – Синтаксич диаграмма описания **налагаемой переменной**

**Пример 1.4.** Использовать процедуры с параметрами без типа (1-й способ). Программа полного и частичного **сравнения значений элементов массивов и записей.**

Стандартная функция **Sizeof** возвращает размер памяти в байтах, занимаемый своим аргументом.

Встроен ф-ция **Inc (n)** увел-ет значение своего аргумента на +1 (n+). Наращивание – в пределах размеров сравниваемых перем-ых в байтах (N < Size) пока n-е байты массивов D и S =ы

**Type**

**Vector = Array [1..6] Of Integer;**

**Point = Record**

**Х, У: Integer**

**End;**

**Var {Объявление глобальных переменных}**

**Vec1, Vес2: Vector; {Vec1, Vec2 — массивы из 6 целых чисел}**

**N, I: Integer;**

**Р: Point; {Р - запись из двух целых полей Х и У}**

**Rеz: Boolean;** {Rez — глобал, куда заносится рез-т вып проц-ы Equal1}

{=сравнивает любые 2 переменные Source и Dest любого размера Size }

**Procedure Equal1** (Vаr **Source, Dest**; Size: Word; Var Rez: Boolean); {**Sourсе, Dest** – сравниваемые переменные, **параметры-переменные б/типа;** Size – размер полей, занимаемых сравниваемыми переменными в памяти машины; **Rez** – результат выполнения процедуры Equal1}

**Var**

**N: Integer; {N** — локальная переменная**}**

**S: Array [0 .. Maxint] Of Byte Absolute Source;** {На переданный

параметр Source **наложен массив байтов S**}

**D: Array [0 .. Maxint] Of Bуtе Аbsolute Dest;** {На переданный пар-р Dest **наложен** **массив байтов D**}

**Begin**

**N := 0;**

**While (N < Size) And (S [N] = D [N]) Do Inc (N); {**Встроен ф-ция Inc (N) увел-ет знач аргумента на +1. Наращивание – в пределах размеров сравниваемых перем-ых в байтах (N < Size) пока N-е байты массивов D и S =ы}

**Rez := N = Size; {Rez = True, если знач-ия D и S в пределах**

**длины Size =ы** (N достигло значения Size), **иначе Rеz = False}**

**End;** {Конец процедуры Equal1}

**Begin {========= Вызывающая программа ==============}**

**Writeln (‘**Вв значения переменных в последовательности: **’);**

**Writeln** (‘Vес1 [1] Vec1 [2] ... Vес1 [6]’**);**

**Writeln** (‘Vec2 [1] Vec2[2] ... Vec2 [6]’**);**

**Fог I:= 1 Tо 6 Do Read (Vec1 [I]); {Ввод массива Vec1}**

**Read1n;**

**For I:=1 Tо 6 Dо Rеаd (Vес2 [I]); {Ввод массива Vеc2}**

**Readln;**

**Equal1 (Vec1, Vес2, Sizeof (Vector), Rez); {1ый оператор вызова процедуры: выполняется полное сравнение массивов Vec1 и Vес2 по всей длине Sizeof (Vector); стандартная функция Sizeof возвращает размер памяти в байтах, занимаемый своим аргументом}**

**Writeln (**‘Рез-т полного сравнения массивов = ‘, Rez**);**

**Writeln** (‘Вв-те количество **N** сравниваемых элементов массивов’);

**Rеаdln (N);**

**Equal1 (Vес1, Vес2, Sizeof( Integer) \* N, Rez); {2ой оператор вызова процедуры — побайтное сравнение первых N элементов типа Integer Vеc1 и Vеc2}**

**Writeln** (‘Результат сравн-ия‘, N, ‘ элементов = ‘, Rez);

**Equal1 (Vec1, Vес2 [3], Sizeof (Integer) \* 2, Rez); {3й оператор вызова процедуры — побайтное сравнение первых 2х элементов типа Integer массива Vec1 с 3-м и 4-м элементами массива Vec2}**

**Writeln** (‘Рез-т сравнен 1-го и 2-ro эл-ов Vec1 с 3-м и 4-м ’ + ‘элементами Vес2 =’, Rez);

**Writeln (**'Введите значения полей Х и У записи Р в виде: Х У’**);**

**Readln (P.Х, Р.У);**

**Equal1 (Vес1 [5], Р, 4, Rez); {4й оператор вызова процедуры — выполняется побайтное сравнение 5-ro и 6-го элементов массива Vес1 с полями записи Р.Х и Р.У}**

**Writeln** (‘Результат сравнения 5-го и 6-ro элементов массива Vec1 с ’ + ‘полями записи Р.Х и Р.У ’**, Rez);**

**End.**

**=================Кратко код:===============**

**type**

Vector = **array**[1..6] **of** integer;

Point = **record**

x, y: integer

**end**;

**var**

Vec1, Vec2:Vector;

n, i: integer;

P: Point;

Rez: boolean;

**procedure** Equal1 (**var** Source, Dest; size: word; **var** Rez: boolean);

**var**

n: integer;

S: **array**[0..Maxint] **of** byte absolute Source;

D: **array**[0.. Maxint] **of** byte absolute Dest;

**begin**

n:= 0;

**while** (n < size) **and** (S[n] = D[n]) **do**

inc(n);

Rez:= n = size;

**end**;

**begin**

writeln('вв знач перемен в послед-ти');

writeln('vec[1] ... vec1[6]');

writeln('vec2[1] ... vec2[6]');

**for** i := 1 **to** 6 **do** read(vec1[i]);

readln;

**for** i:= 1 **to** 6 **do** read(vec2[i]);

readln;

Equal(vec1, vec2, **sizeof**(Vector), Rez);// 1

writeln('рез-т полн сравнен массивов = ', Rez);

writeln('вв к-во n сравниваемых эл-тов мас-ва');

readln(n);

Equal1(vec1, vec2, **sizeof**(integer) \* n, Rez); //2

writeln('рез-т полн сравнен', n, ' эл-тов = ', Rez);

Equal1(vec1, vec2[3], **sizeof**(integer) \* 2, Rez); //3

writeln('рез-т сравнен 1го и 2го эл-тов с 3м и 4м'+'эл-ми vec2 = ', Rez);

Writeln ('Введите значения полей Х и У записи Р в виде: Х У’);

readln(P.x, P.y);

Equal1(vec1[5], P, 4, Rez); //4

Writeln ('Рез-т сравн 5го и 6ro эл-ов массива Vec1 с ' +

'полями записи Р.Х и Р.У ', Rez);

**end**.

Эта прога иллюстрирует возможности сравнения значений **произвольных типов** на примере сравнения **массивов Vec1 и Vес2** (состоят из 6 целых чисел) между собой и с записью ***Р*** (состоит из 2х полей целого типа ***Х*** и ***Y)***. Массивы **Vес1** и **Vес2** сравниваются как в полном объеме, так и по отдельным частям.

Для сравнения переменных любого типа – процедура **Equal1** с 4-мя параметрами — **Source, Dest, Size, Rez*.***

Процедура выполняет сравнение переменных **Source**и**Dest*.*** Размер памяти этих переменных определяется параметром-значением (П-З) **Size** типа **Word.** Логический результат работы процедуры присваивается П-П-ой **Rez** типа **Boolean.**

Переменные ***Source*** и ***Dеst*** оформлены в виде П-П б/типа. Для обеспечения их совместимости с **фактичес**кими параметрами любого типа в теле **процедуры *Equal1*** введены **локальные переменные *S*** и ***D*** типа массив (состоит из ***Maxint+1*** байт), наложенные на параметры **Source** и **Dest** соответственно (***Maxint*** — это встроенная в язык константа, = **макс**имальному числу типа **Integer** в компьютере). При вызове процедуры переменные ***S*** и ***D*** накладываются на конкретную область памяти, занимаемую **фактич**ескими переменными, передаваемыми в качестве значений ***Source* и *Dest*.** Т о, **фактич**еские переменные, независимо от их действительного типа, будут рассматриваться как **массивы байтов**. Конкретное количество байтов в массиве при работе процедуры ограничено величиной ***Size***, передаваемой на вход процедуры как П-Знач.

**!**: в данном примере использованы **2 разные** переменные с одним и тем же именем ***N***. Глобальная ***N*** типа ***Integer*** введена в разделе ***Var*** вызывающей программы как кол-во элементов входных массивов ***Vес1, Vec2***, в пределах которых надо сравнить данные массивы. Её область действия (***N)*** — все тело программы, начиная от точки ее определения, за исключением тела процедуры ***Equal1:*** здесь введена локальная переменная типа ***Integer*** под тем жe именем ***N***. Область её действия (локальной ***N) -*** тело процедуры ***Equal1*.** При выходе из процедуры локальная ***N*** прекращает свое действие (ее значение теряется).

В теле процедуры **Equal1** последовательно сравнивают на равенство соответствующие элементы массивов **S** и **D**: фактически сравнивают соответствующие байты **Source и Dest** (параметров без типа)**.** Если они =ы, то значение локальной **N** увеличивается на +1.

Если при последовательном сравнении элементов массивов ***S*** и ***D*** в пределах поля ***Size***  будет ***S = D***, П-П-ой **Rez** б := знач **True**, иначе - ***False*.**

В вызывающей программе выполнено 4 вызова процедуры ***Equal1*.**

1ый вызов ***Equal1 (Vес1, Vес2, Sizeof (Vесtоr), Rez)*** *-*  сравнение массивов ***Vec1*, *Vec2 -*** в пределах размера памяти, определяемого значением встроенной функции ***Sizeof*** (для аргумента ***Vector*** значение функции ***Sizeof =*** **12 байт** — напомним, что переменная типа **Integer** занимает поле памяти в **2 байта**). Т о, выполняется полное сравнение массивов ***Vec1*** и ***Vес2*.**

2й вызов ***Equal1 (Vес1, Vec2, Sizeof (Integer) \** N, Rez)** - сравнение первых ***N*** элементов типа *I****nteger*** массивов *Vес1* и *Vec2 (Vес1[1]*, *Vec2[1])*.

3й вызов ***Equal1 (Vec1, Vec2 [3], Sizeof (Integer) \*2, Rez)*** *-* сравнивают первые 2 элемента массива *Vес([1],[2])* с 3м и 4ым элементами массива *Vес2*.

4й вызов ***Equal1 (Vec1[5], P, 4, Rez)***– размер сравниваемых полей памяти ограничен **4я байтами**. Поэтому сравнивают 5й и 6й эл-ты массива ***Vес1*** с полями записи ***Р.Х*** и ***Р.У*** соответственно.

Кроме 2х переменных с 1 именем ***N*** здесь есть 2 переменные с именем ***Rеz*:** 1). переменная ***Rez* –** как **фактич**еский параметр в операторах вызова ***Equal1* –** глобальная переменная, объявлена в разделе объявления глобальных переменных вызывающей программы и нужна для занесения результата работы ***Equal1*.** Область её действия – вся программа от точки ее объявления кроме ***Equal1*,** где такое же имя использовано как П-П – 2). Область её действия (**параметра *Rez****) –* тело процедуры.

**СПОСОБ 2.**

Для обеспечения совместимости с фактическими параметрами внутри процедуры вводится **нужный тип.** Данный тип ставитсяв **соответствие П-П-ой б/типа с помощью присваивания типа переменной –** способ применения **нетипизованных П-П-ых.**

**Пример 1.5.** Использовать процедуру с параметрами без типа (2-й способ). Программа полного и частичного сравнения значений элементов массивов и записей.

**Type**

**Vector = Array [1 .. 6] Of Integer;**

**Point = Record**

**Х, У: Integer;**

**End;**

**Var** {====== Объявление глобальных переменных ==========}

**Vеc1, Vec2: Vector**; {Vес1, Vec2 – массивы из 6 целых чисел}

**N, I: Integer;**

**Р: Point; {Р – запись из двух целых полей Х и Y}**

**Rez: Boolean; {Rez – глобальная переменная, куда заносится результат выполнения Equal}**

**{Процедура Equal сравнивает любые 2 переменные Source и Dest любого размера Size}**

**Procedure Equal (Var Source, Dest; Size: Word; Var Rez: Boolean); {Source, Dest – сравниваемые переменные (П-П-ые б/типа); Size – размер полей, занимаемых сравниваемыми переменными в памяти машины; Rez – результат выполнении Equal}**

**Type**

**Bytes = Аггау[0 .. Maxint] Of Byte; {Тип массив байтов вводится для последующего присваивания параметрам без типа Source и Dest}**

**Var**

**N: Integer; {N – локальная }**

**Begin**

**N:= 0;**

**While (N< Size) And (Bytes (Dest) [N] = Bytes (Source)[N]) Do**

**Inc (N); {Наращивание N – в пределах размеров сравниваемых переменных в байтах (N<Size) пока N-е байты перем-ых Dest и Source виде массива байтов, =ы}**

**Rеz := N = Size; {Rez = True, если значения Dest и Source =ы** в пределах длины Size – **N достигло значения Size, иначе Rez = Fаlse}**

**End;**

**Begin {=== Вызывающая программа =====}**

**…**

Это фрагмент программы выполняет те же действия, что и в предыдущем примере. Процедура **Equal** по своему назначению аналогична ***Equal1*.** **Но** совместимость параметров без типа ***Source* и *Dest*** с фактическими параметрами достигнута **иначе**. В теле процедуры введен тип ***Bytes*** *-* массив из ***Maxint+1*** байта. Данный тип присваивается параметрам без типа ***Source* и *Dest*** с помощью присваивания типа: Bytes (Source) Bytes(Dеst).

Вызывающая прога здесь – как в пр-ре 1.4.

1.2.9. Параметры процедурного типа (ППТ)

**-** это группа параметров, перед которыми **нет** **Var**/**Const**и за которыми идет **процедур**ный тип.

Паскаль позволяет организовать подпрограммы, где параметры - другие процедуры или функции. Д/этого применяют **процедурные типы**.

Описание процедурного типа - в разделе ***Туре*.** Синтаксис описания совпадает с заголовком подпрограммы (процедуры или функции), но **имя** подпрограммы в описании опускается. Например:

**Pr = Procedure (Х: Integer; Var Y: Real);**

**Pr1 = Procedure (А, В: String; Р: Pr);**

Здесь объявлены 2 **процедурных типа** — ***Pr, Pr1***. В типе *Pr1* как параметр **– переменная *Р* процедурного типа *Pr*.**

**Имена** параметров в описании ППТ – чисто **иллюстративная роль**. Никакого влияния на это описание данные имена не имеют.

ППТ м использовать в любом контексте (могут быть введены переменные этого типа).

Например, с учетом вышеприведенных описаний типа вводятся 2 переменные ***P*** и ***P1*** процедурного типа:

***Var***

***Р: Pr;***

***P1: Pr1;***

Процедурной переменной м присвоить значение другой процедурной переменной, имя процедуры или ф-ции: переменная слева и значение справа от := д б совместимы по присваиванию. Для обеспечения совместимости подпрограмма должна отвечать требованиям (если ее имя нужно присвоить процедурной переменной):

1. компилироваться в состоянии **{$f+}** (**дальний вызов, переход между сегментами памяти**); т е в исходном тексте программы перед описанием такой подпрограммы должна быть размещена **директива компилятора {$f+}**, а после окончания описания – **{$f-}*;***

2. она **не** должна быть: а). стандартной процедурой или функцией; б). вложенной; в). подпрограммой типа Inline; г). подпрограммой прерывания (Interrupt);

3. ее заголовок должен соответствовать заголовку процедурной переменной.

На физическом уровне при присваивании процедурной переменной имени подпрограммы в данную переменную заносится **адрес подпрограммы.** Поэтому **процедурная переменная аналогична указателям**, т. к. она содержит ссылку на процедуру (или функцию).

Процедурная переменная занимает **4 байта памяти**.

Наличие процедурных типов позволяет описывать подпрограммы, которые воспринимают процедуры или функции как **параметры**.

Если подпрограмма должна передаваться как фактический параметр, она должна удовлетворять правилам совместимости типа при присваивании.

**Пример 1.6.** Применение ППТ. Программа вычисляет значения натуральных логарифмов (***Ln***) или экспонент (*Ехр*) элементов 1мерного массива.

**Процедура** **Log** вычисляет логарифмы элементов массива.

Процедура **Step *–*** экспоненты элементов массива.

Процедура **Vizov** вызывает нужную процедуру **Log** или **Step** и выводит результаты их выполнения.

**{$f+} –** Директива компилятора **вкл**ючения дальнего вызова в памяти

**{$f-} –** Директива компилятора **выкл**ючения дальнего вызова в памяти

Program Proced;

**Const**

**Razmer = 4;**

**Type**

**Mas = Array [1 .. Razmer] Of Real;**

**Proc = Procedure (Х: Mas; Var Х1: Mas);** {Объявл процедур-го типа}

**Var**

**Vec1: Mаs;** {Vec1 - массив из 4ех вещественных чисел}

**Priznak, N, I: Integer;** {Priznak — признак выбора нужной процедуры Log или Sтер}

**{$f+}** {Директива компилятора включения дальнего вызова в памяти}

{=== Процедура вычисления **Ln** элементов одномерного массива ====}

**Procedure Log (Y: Mas; Var Y1: Mas); {** ***Y*** – параметр-значениe,

параметр-переменная ***Y1*** **}**

**Var**

**J: Integer;**  {J — локальная }

**Begin**

**Writeln (**' Вызвана процедура Log. Значения Ln (Vec1[J]): '**);**

**For J := 1 To Razmer Dо**

**Y1[J]:= Ln (Y[J]);**

**End;**

**{=== Процедура вычисл Ехр элементов одномерного массива===}**

**Procedure Step (У: Mas; Var У1: Mas);**

**Var**

**N: Integer; {N** — локальная **}**

**Begin**

**P**

**For N := 1 To Razmer Do**

**У1[N] := Ехр (Y[N]);**

**End;**

**{$f-} {**Директива компилятора выключения дальнего вызова в памяти}

{== Проц-ра вызова вычисляющих процедур и выв-а рез-ов их работы}

**Procedure Vizov (G: Mas; Proced: Рrос); {Proced — ППТ}**

**Var**

**N: 1 .. Razmer;**  {N, Rez — локальные }

**Rez: Mas;**

**Begin**

**Proced (G , Rez);** {Вызов вычисляющей процедуры}

**For N := 1 To Razmer Do**

**` Write (Rez [N]);** {Вывод рез процедуры}

**Writeln;**

**End;**

**{=========== Вызывающая программа ===============}**

**Begin**

**Writeln(**'Введите знач-ия элементов массива в последовательности:**');**

**Writeln (**'Vес1[1] Vecl[2] ... Vec1[', Razmer, ']'**);**

**For I := 1 To Razmer Dо**

**Read (Vec1[I]);** {Ввод массива Vec1}

**Readln;**

**Writeln** ( 'Введите признак вида вычислений: '**);**

**Writeln (**' 0 — д/вычислен Ln, 1 — д/вычисления Exp'**);**

**Rеаdln (Priznak);**  {Ввод признака вида вычислений}

**If Priznak = 0 Then**

**Vizov (Vес1, Log)** {Вызов Vizov. **Фактич** пар-ры - массив Vec1 +

**имя** процедуры Log}

**Else**

**If Priznak = 1 Then**

**Vizov (Vес1, Step)**{Фактич пар-ры – мас Vec1 + имя проц-ры Step}

**Else**

**Writeln (**'Ошибка признака'**);** {Неверно указано знач-е призн-ка}

**End.**

=========КРАТКО=============

Эта прога выполняет обработку элементов 1мерного массива **Vec1** типа ***Mas*** из 4х вещественных (real) чисел. В зависимости от значения переменной ***Priznak*** (0/1) вычисляют натуральные логарифмы или экспоненты элементов массива.

Программа имеет 3 процедуры - **Log, Step, Vizov***.*

*1)* Процедура ***Log*** вычисляет значения натур логарифмов элементов массива ***Vec1***, передаваемого на вход процедуры с помощью параметра-значения (П-З) ***Y*** типа ***Mas*.** Результаты работы процедуры присваиваются параметру-переменной (П-П) ***Y1*** типа ***Mas*.**

2) Процедура ***Step*** – значение экспоненты эл-ов массива ***Vec1*.** Для передачи исходного массива на вход процедуры также используют П-З ***Y*** типа ***Mas*,** а для формирования и хранения ее результатов - параметр-переменная (П-П) ***Y1*** того же типа.

3) Процедура **Vizov *-*** для вызова необходимой вычисляющей процедуры и вывода результатов вычислений. Параметры процедуры ***Vizov*** – П-З ***G*** типа ***Mas*** и ***Proced –*** параметр процедурного типа (ППТ) **Ргос**:

**Proc = Procedure (Х: Mas; Var Х1: Mas);**

В вызывающей программе вводят значения 4х элементов массива ***Vec1*** и значение **Priznak** признака вида вычислений (lg / exp = 0/1).

Затем вызывается процедура **Vizov.** За 1й фактический параметр используется ***Vec1***. 2й фактический параметр в зависимости от значения переменной **Priznak** *-* **имя** процедуры ***Log*** или ***Step***.

Так как процедуры ***Log, Step*** передаются как фактический параметр, то они должны компилироваться с директивой компилятора ***{$f+}*** (записана перед процедурой ***Log*).** Для отмены данной директивы после процедуры ***Step*** (второй) помещена директива компилятора ***{$f-}.***

Таким образом, при вызове процедуры **Vizov** используется оператор вызова вычисляющей процедуры **Proced (G, Rez);**

В зависимости от конкретного значения процедурной переменной **Proced** будет вызвана к исполнению либо процедура ***Log*,** либо ***Step*.** Результаты работы вызванной процедуры помещают в массив **Rez** - локальная переменная процедуры **Vizov**.

1.2.10. Использование производных типов   
в качестве параметров подпрограмм

Производные типы могут применяться как П-П, П-З или П-К, где **фактич**еские параметры – только переменные соответствующего типа, но не выражения (исключение: **строковые константы** string).

В заголовке подпрограмм для указания **типов** формальных параметров используют только **имена типов**, но не их задание.

Например, нельзя писать:

**Procedure X (Y: Array [1..10] Of Integer; I: 1..100);**

Надо **предварит**ельно определить типы:

**Type**

**Mas = Array [1..10] Of Integer;**

**Diap = 1..10;**

**Procedure X (Y: Mas; I: Diap);**

При использовании массива как **П-З** при входе в подпрограмму образуется локальная переменная-массив, куда заносится значение фактического параметра-массива, то есть идет **копирование** массива. На это надо дополнительные время + память. Поэтому лучше не исп массив как П-З.

Передача массива как П-З оправдана, когда подпрограмма изменяет значения параметра-массива, а в вызывающей программе надо сохранить значения массива-аргумента.

Аналогично надо передавать и переменные других произво**дн**ых типов, используемые как параметры подпрограмм.

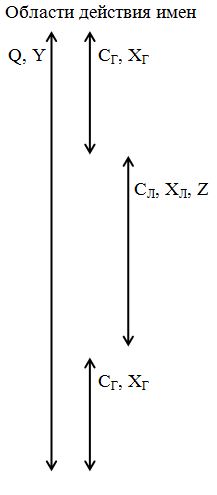
1.2.11. Принцип локализации имен

**Принцип локализации –** вводимые имена (в употребление) в подпрограмме, имеют силу только в ней.

Трудоемкость разработки многих программ - десятки лет + куча программистов, каждый разрабатывает свой фрагмент программы (подпрограмму и не зависят друг от друга при выборе **идентификаторов**: в любой подпрограмме вводят любые типы значений и программные объекты.

Если такое же имя было описано вне тела подпрограммы, то её область действия на процедуру **не идет**.

**Пример 1.7.** Использование одинаковых имен в разных областях действия.



**Var**

**Y: Real;**

**X: Char;**

**Const**

**C=10;**

**Procedure Al (X, Z: Real);**

**Var**

**C: Real;**

**Begin**

**C := X + Z;**

**X := 2 \* X;**

**Y := 1;**

**Writeln** (‘C = ‘, C, ‘ X = ‘, X, ‘Y = ‘, Y**)**

**End;**

**Begin**

**X := ’A’;**

**Y := 0.5;**

**Al (Y, 0.1);**

**Writeln** (‘C = ‘, C, ‘ X = ‘, X, ‘Y = ‘, Y**) End.**

Здесь **идентификаторы** ***C, X*** – и локальные, и глобальные переменные. Через ***CГ, XГ***  - глобальные ***C, X***, ***CЛ, XЛ***– локальные.

В программе - 2 оператора вывода ***Writeln***. Первым будет выполнен оператор в подпрограмме ***Al*** и выведены значения ***С, X, Y,*** =е:

**c = 0.6 x = 1 y = 1**

**c = 10 x = A y = 1**

**Решение: Y := 0.5 – это Х в процедуре, => c = 0.6**

1.3. Функции

1.3.1. Описание функций

Объявление функции – в разделе процедур и функций того блока, в котором эта функция вводится в употребление.

Объявление функции - как объявление **процедуры**, но есть ***отличия:***

**1 отличие:** Заголовок функции имеет вид:

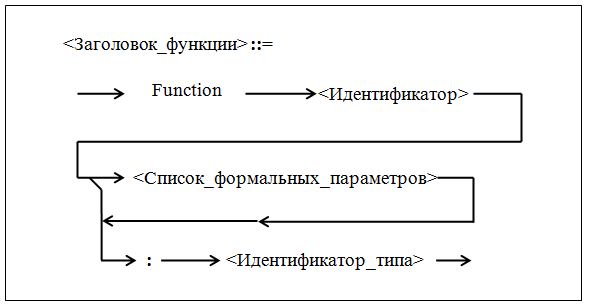


Рис 1.7 – Синтаксическая диаграмма заголовка функции

**Заголовок** определяет 1.**имя** функции**,** 2.список **формаль**ных параметров и 3.**тип** возвращаемого значения. Заголовок начинается служебным **Function.**

<**Список\_формальных\_параметров**> - определяется так же, как процедуры. Формальные параметры м б: П-З-ия, П-П, П-К, параметры б/типа и ППТ (5 шт).

Завершение заголовка функции - тип возвращаемого значения функции (<**Идентификатор\_типа**>). При его описании используют **только имя типа, а не его задание**. Поэтому тип возвращаемого значения – стандартный или предварительно описанный.

**2 отличие:** Как и **процедуры**, функции могут изменять значения глобальных переменных, П-П и параметров б/типа. + каждая функция вычисляет **возвращаемое значение функции**: передается в точку вызова функции.

Чтобы установить возвращаемое знач-е, надо присвоить его идентификатору функции. Поэтому в теле ф-ции обязательно д б хотя бы 1 оператор присваивания: в **левой** части **имя** функции **б/**параметров:

**<Идентификатор\_функции> := <Выражение>**

Хотя бы 1 оператор такого вида должен быть выполнен. Паскаль допускает функции с возвращаемыми значениями любого скалярного типа, типа **String** и типа указатель (3шт).

­­­­­­­­­­­­­­­­ **Пример 1.8. Объявление** ф-ции. Функция **Simvol** определяет наличие искомого символа в анализируемой строке.

**Function Simvol (Const S: String; C: Char): Boolean**;- отличие от проц

**Var I: 1 .. 255;**

**Begin**

**For I := 1 To Length (S) Do**

**If S[I] = C Then**

**Begin**

**Simvol := True; Exit; {Возвращаемое значение}**

**End;**

**Simvol := False; {Возвращаемое значение}**

**End;**

Данная функция посимвольно анализирует значение строки, поступающей на вход ее формального параметра ***S***. Если очередной символ строки = искомому, то возвращаемое значение функции = **True**. Иначе возвращаемое значение функции устанавливается в **False.**

1.3.2. Вызов функции

– для обращения к функции в вызывающей программе.

– это **имя** функции + Списки **фактич**еских параметров в скобках:

**<Вызов\_функции> [(Список\_фактических\_параметров)].**

Количество **фактических** параметров д б = кол-ву **формальных** **в описании ф-ции**, и их типы д соответств-ть.

Напр, для примера 1.8 вызов ф-ции в вызывающей проге м б записан:

**Simvol (X, Y)** (X, Y – фактические параметры)

**Function Simvol (Const S: String; C: Char): Boolean;**

где **фактический** параметр ***X*** д им тип **String,** а **фактический** пар-р ***Y*** – **Char**

Т к функция задает **правило** вычисления возвращаемого значения, то **вызов функции – данное значение**, которое используется как **операнд** какой-либо операции. Т о, вызов функции м б записан везде, где допустимо использование выражений по синтаксису (напр-р, в правой ч оператора :=).

Д/примера 1.8 вызов функции в вызывающей программе м б такой:

**Z := Simvol (X, Y) Or Simvol (A, B);**

Здесь – **2 вызова** функции **Simvol.** При его вычислении в точки вызова передаются возвращаемые значения функции и далее они участвуют в вычислении выражения.

**Пример 1.9.** Вычислить **расстояния** между ***N*** парами точек на плоскости. Использовать функции с параметрами-значениями (П-З).

**Var**

**X1, Y1, X2, Y2: Real; {**Объявление глобальных переменных}

**I: Integer**

**Const**

**N = 10;**

{======= Функция с П-З и возвращаемым значением типа Real ===}

**Function Rast5 (Xx1, Xx2, Yy1, Yy2: Real): Real;**

**Begin**

**Rast5:= Sqrt(Sqr(Xx1-Xx2)+Sqr(Yy1-Yy2))** {Возвращаемое знач-е}

**End;**

**Begin**{========Вызывающая программа==========}

**For I:=1 To N Do**

**Begin**

**Readln (X1, X2, Y1, Y2);**

**{\*} Writeln (Rast5 (X1, X2, Y1, Y2))** {-Точка вызова **}**

**End**

**End.**

Вычисленное функцией расстояние передается в **точку вызова** с помощью возвращаемого значения. Точка вызова – список **фактич**еских параметров оператора вызова процедуры вывода **Writeln**: в Паскале (по синтаксису) в списке вывода м использовать значения выражений.

Вместо оператора **{\*}** в тексте программы м б, например, так:

**X1 := Rast5 (X1, X2, Y1, Y2);**

**Writeln (Х1)***;*

1.4. Рекурсивные подпрограммы

**Рекурсия** – определение какого-либо понятия ч-з само это понятие. Особенность рекурсивного описания подпрограммы – в ее теле есть обращение к самой себе.

Классический пример рекурсивной функции – факториал ***n!***:

****** ***n!*** опред-ся через ***(n-1)!***.

**Пример 1.10.** Рекурсивная функция. Вычислить ***n!***

**Function Fakt (N: Integer): Longint;**

**Begin**

**If N = 0 Then** {Ветвь, определяющая конец рекурсии}

**Fakt := 1 {Возвращаемое значение функции}**

**Else**

**{\*} Fakt := N \* Fakt (N - 1) {Возвращаемое значение функции}**

**End;**

Запись имени функции **Fakt** в левой части оператора присваивания **{\*}** показывает, что это – возвращаемое значение функции. Вызов функции **Fakt** в правой части оператора {\*} – обращении к этой же функции, т.е. есть **рекурсивность.**

2 вида рекурсии **- явная** и **неявная (взаимная**).

**Явная рекурсия** – обращение к подпроге – в теле самой подпрограммы.

**Неявная (взаимная) рекурсия** – обращение к подпрограмме – в теле другой подпрограммы (к ней идет обращение из данной подпрограммы).

Рекурсивное обращение обеспечено **специальными средствами,** которые создают **новый экземпляр ее памяти** при каждом повторном (рекурсивном) вызове подпрограммы, делая временно недоступным прежний экземпляр памяти (т.е. значения локальных переменных и **факти**ческих параметров, существующих на момент рекурсивного вызова). Все экземпляры памяти рекурсивной подпрограммы помешаются в **стек.** Всегда доступен только **текущий (**последний) экземпляр памяти, который при каждом возврате из рекурсивной подпрограммы **удаляется из стека** и становится доступным предыдущий экземпляр.

В **не**рекурсивной подпроге механизма выделения памяти в стеке **нет.**

**Рекурсивность** – свойство описания подпроги, а не свойство подпрограммы. Ту же подпрограмму можно организовать и б/рекурсии. Например, функцию из примера 1.10 можно реализовать без рекурсии так:

**Пример 1.11. Нерекурс**ивная функция. Вычислить ***n!*** по формуле:

**.**

**Function Fakt (N: Integer): Longint;**

**Var**

**K, I: Integer;**

**Begin**

**K:=1;** - первая цифра в формуле n!

**For I := 1 To N Do -** наращивание

**K := K \* I;**

**Fakt := K; {Возвращаемое значение функции}**

**End;**

«+» рекурсив записи – запись проги короче и нагляднее, чем нерекурсивная.

«-» – надо больше памяти и машинного времени (за счет повторных обращений к функции), чем при нерекурсивной записи.

1.5. Директивы

В предыдущих разделах рассмотрены подпрограммы, где **тело** содержит только разделы 1.локальных описаний + 2.операторов = блок. Кроме **блока** тело подпрограммы м иметь **директивы.**

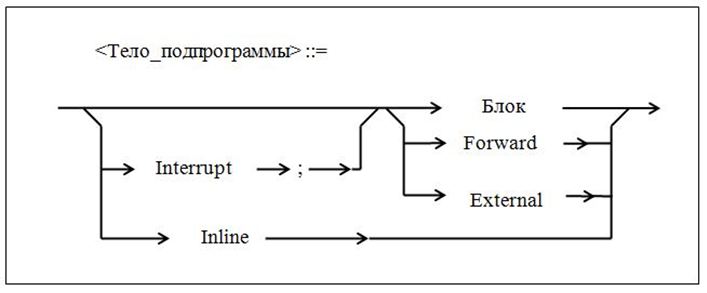


Рис 1.8 – Синтаксическая диаграмма тела подпрограммы

1. прерывание **Interrupt -** подпрограмма **прерывания**
2. **опережающее** описание **Forward**
3. **внешнее** описание **Eхternal**
4. **внутреннее** описание **Inline**

В описании подпрограммы перед блоком м указать директиву прерывания **Interrupt**: тогда это станет подпрограмма **прерывания.**

Вместо блока в описании подпрограммы м б записано описание: 1.**опережающее** (директива **Forward**), 2.**внешнее** (директива **Eхternal**) или 3.**внутреннее** (директива **Inline**).

2.Внешними подпрограммами используются при объединении большого количества объектных модулей. Описания **External** подключает отдельно скомпилированные подпрограммы на языке ассемблера. Команда **"L Имя\_файла***"* для внешней **подпрог**раммы установливает связи с программой или модулем.

3.Директивы **Inline:** записывает вместо раздела операторов инструкции в **машинном** коде.

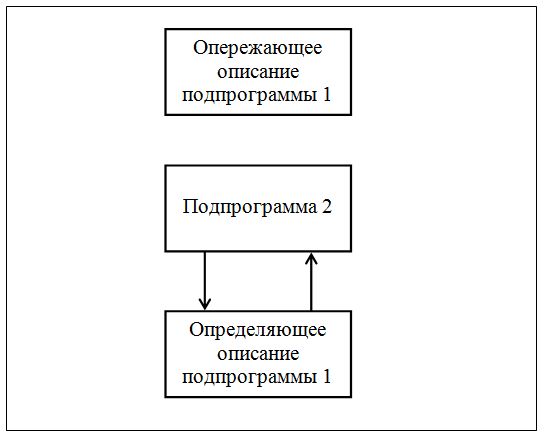
Описание подпрограмм типа **Interrupt, External, Inline -** не рассматриваем.

**Опережающее описание** **Forward** – описание подпрограммы, содержащее вместо блока **директиву** **Forward**. После этого описания подпрограмма должна определяться с помощью **определяющего описания** - это описание, использующее тот же заголовок подпрограммы, что и в опережающем описании, но **без списка формальных параметров**, и в которое **включен блок** (разделы 1.локальных описаний и 2.операторов).

ОпереЖающее + опредеЛяющее описания = **полное описание** **подпр**ограммы. Подпрограмма считается **описанной** с помощью опере**ж**ающего описания.

**Определ**яющее описание м б **внешним** описанием, но не может быть внутренним или другим **опереЖ**ающим описанием. Опред**еля**ющее описание не содержит директиву **Interrupt**.

Между опере**Жа**ющим и опре**деЛ**яющим описаниями могут описываться другие подпрограммы, которые могут обращаться к подпрограмме с о**пережа**ющим описанием (рис 1.9). Так м реализовать **взаимную (неявную) рекурсию**.

 Рис 1.9 – Организация взаимной рекурсии

**Пример 1.12.** Использование процедур со **взаимной рекурсией** при опере**Ж**ающем описании одной из процедур. Применение директивы **Forward.**

Программа вычисления ***N***-ых членов последовательностей, заданных **рекуррентными** формулами:

 Х1 = 1,5; Р1 = -1

**Рекуррентная формула – формула вида** an = f(n, an-1, an-2, …, an-p), выр-щая каждый член послед-ти ч/з предыдущих членов и возможно номер члена послед-ти.

Общая проблематика [вычислений](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) с использованием рекуррентных формул является предметом теории [рекурсивных функций](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F).

**Рекуррентным уравнением** называется уравнение, связывающее несколько подряд идущих членов некоторой числовой последовательности. Последовательность, удовлетворяющая такому уравнению, называется **рекуррентной последовательностью**

**Var**

**N: Integer; {N, Xi, Pi - Глобальные }**

**Xi, Pi: Real;**

{====== Процедура Form\_Xi вычисляет значение X =========}

**{\*} Procedure Form\_Xi (I:** Integer; **Var Xi:** Real)**;** {**Опережающее**

описание процедуры Form\_Xi}

**Forward;**

{========== Процедура Form\_Pi вычисляет значение Р =======}

**Procedure Form\_Pi (I:** Integer; **Var Pi:** Real**);** {Проц с **саморекурс +**

**взаимн** рекурс (**вызыв-т** проц Form\_Xi)}

**Var Xi: Real;**  {Описание локальной переменной Xi}

**Begin**

**If I = 0 Then Pi := 2** {Начальное значение Pi}

**Else**

**Begin**

**Form\_Xi (I - 1, Xi);** {Вызов процедуры Form\_Xi }

**Form\_Pi (I - 1, Pi);** {Рекурсивный вызов процедуры Form\_Pi}

**Pi := Xi – Pi** {Вычисл-ие i-го члена послед-ти Р}

**End**

**End; //---**Конец процедуры Form\_**Pi**

**Procedure Form\_Xi;** {**Определяющее** опис-е процедуры Form\_Xi}

**Var Pi: Real;** {Описание локальной пер-й Pi}

**Begin**

**If I = 0 Then Xi := 1** { Начальное значение Xi}

**Else**

**Begin**

**Form\_Xi (I - 1, Xi);** {Рекурсивный вызов проц-ры Form\_Xi}

**Form\_Pi (I - 1, Pi);** {Вызов процедуры Form\_Pi}

**Xi := 0.5 \* (Xi+Pi)** {Выч-ие I-го члена последовательности Х}

**End**

**End; //---**Конец процедуры **Form\_Xi**

**Begin** {===== Вызывающая программа ============}

**Write ('Вв N:'); Read (N);** {Чтение № N вычисляемых членов послед-ей}

**Form\_Xi (N, Xi);** {Вызов процедуры Form\_Xi}

**Form\_Pi (N, Pi);** {Вызов процедуры Form\_**Pi}**

**Writeln ('X', N, ' = ', Xi);** {Печать рез-ов знач глобальн перем Xi и Pi}

**Writeln ('P', N, ' = ', Pi)**

**End.**

Здесь использованы 2 процедуры - ***Form\_Xi, Form\_Pi*.** В теле обеих процедур содержится как вызов их самих (явная рекурсия), так и взаимный вызов друг друга (**неявная** рекурсия). В связи с последним одна из процедур (***Fоrm\_Xi***) д б определена с помощью опере**ж**ающего описания ({\*} в программе).

Опре**дел**яющее описание процедуры ***Form\_Xi*** содержит разделы 1.объявлений локальных переменных и 2.операторов. В отличие от **обычного** описания в заголовке процедуры ***Form\_Xi*** при определяющем описании **нет** списка формальных параметров.

Между определяющим и опережающим описаниями ***Form\_Xi*** описана процедура ***Form\_Pi*** обычным образом, без директив.

В первый раз проц ***Form\_Xi*** и ***Form\_Pi*** вызываются из головной программы. Затем идут последовательные вызовы процедурами самих себя и друг друга.

1.6. Библиотечные модули пользователя

1.6.1. Общие сведения

**3типа файлов**:

а) файлы, содержащие **исходные тексты** программ (типа **pas)**;

б) **готовые к выполнению** **exe-**файлы – файлы в результате компиляции исходного текста программы;

в) **tpu-**файлы – в результате компиляции библиотечных модулей (***TPU – Turbo Pascal Unit***).

**Unit** – «**модуль**, блок, узел, единичный» (юная модель)

В Турбо Паскале есть стандартные библ-ые модули д/поддержки работы программных модулей: **System, Graph, Dos, Crt, Printer, Overlay, Strings, Turbo3, Graph3**.

Часто используемые процедуры и функции удобно хранить в библиотеках подпрограмм пользователя (вызывать в разных программах). Средство создания библиотек подпрограмм – библиотечные **модули** – это результат трансляции в режиме **Compile** специальной структурной единицы ***Unit*** с установленной директивой **Destination = Disk.** В результате такой компиляции создается **tpu**-файл.

Ввод **tpu**-файлов позволяет создать проги (**exe**-файлы) > 64К байтов. Но **каждый** из **tpu-**файл не может превышать 64К байта. Сумма объемов модулей программы ограничена объемом оперативной памяти компа.

В отличие от программы, модуль нельзя запустить на выполнение самостоятельно. Модули компилир-ся независимо от использующей проги.

Чтобы подключить модуль к программе (или другому модулю), надо указать его имя в директиве **Uses** проги / модуля.

1.6.2. Структура модуля Unit

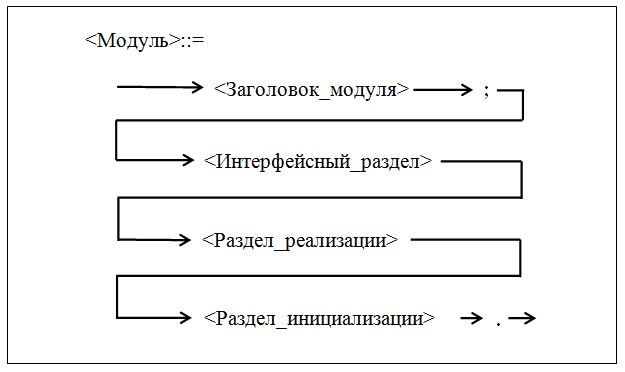


Рис 1.10 – Синтаксическая диаграмма **модуля Unit**

**Структура модуля** также может быть представлена так:

**Unit <Имя\_модуля>;**  {Раздел (секция) Unit (**заголовок** (имя) модуля)}

**Interface** {Интерфейсная секция: Содержаниие – действующие лица}

**Uses <Список\_использ-ых\_Unit>**

{Описание **глобальных** эл-ов проги:}

**Type …**

**Const …**

**Var …**

{Заголовки процедур и функций с указанием параметров}

**Procedure …**

**Function …**

**Implementation** {Секция реализации: описание кто что делает}

<Описание **локальных** 1.типов, 2.констант, 3.переменных, 4.процедур и ф-ций, описанных в секции Interface + 5.внутренних процедур и функций>

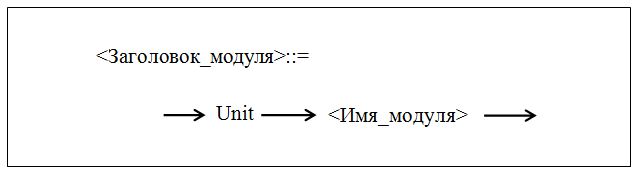
**Inicialization**

**Begin**  {Секция **инициализации: вызывающая прога** - основная}

**<Операторы>**

**End. Assign = “назначать”**

Т о, модуль **Unit** состоит из разделов, называемых также **секции.**

В ***разделе (секции)* Unit** (заголовок модуля) – **имя** библиотечного модуля. Оно д **совпадать** с именем файла c исходным текстом модуля. Например, если у исходного файла имя **Arifm.pas**, то модуль - **Unit Arifm**.

В **секции Interface** **(интерфейсном разделе*)*** описывают **внешние элементы** – **глобальные** типы, константы, переменные, процедуры и функции: элементы основной программы/ другому модулю. Описания - в любом порядке (как и в программном Паскаль-модуле).

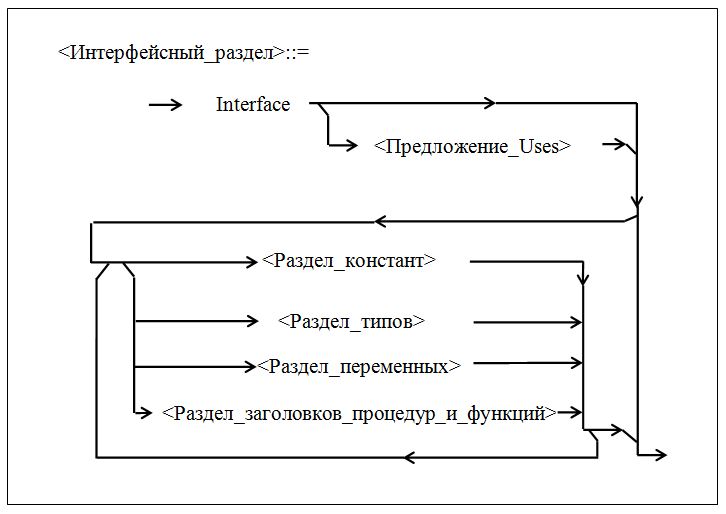


Рис 1.12 – Синтаксическая диаграмма интер**фейс**ного раздела

Описания процедур и функций – их заголовки, с указанием параметров и возвращаемых значений функций, но их **Тел** здесь **нет.** Основная программа и другие модули имеют доступ к элементам интерфейсной секции, как если бы они были описаны в них самих.

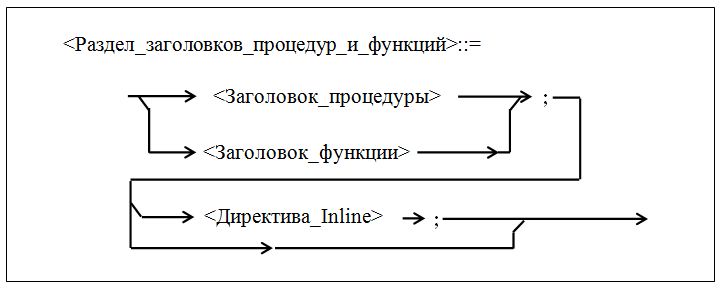


Рис 1.13 – Синтакс-кая диаграмма раздела заголовков процедур и функций

В интерфейсной секции недопустимы заголовки подпрограмм с директивами **Interrupt** и**Forward**. Описание заголовков процедур и функций в интерфейсной секции - аналог опере**ж**ающего описания (аналог директивы **Forward*)***.

Если при объявлении типов, данных или подпрограмм используют типы, введенные в других модулях, то эти модули должны быть перечислены в директиве **Uses** сразу после слова **Interface*.*** В директиве **Uses** данной секции указывать только модули, необходимые в этой секции.

В **секции Implementation** **(разделе реализации*)*** описывают **локальные элементы** (типы, константы, переменные, метки). + тела процедур и функций – их заголовки описаны в секции **Interface**.

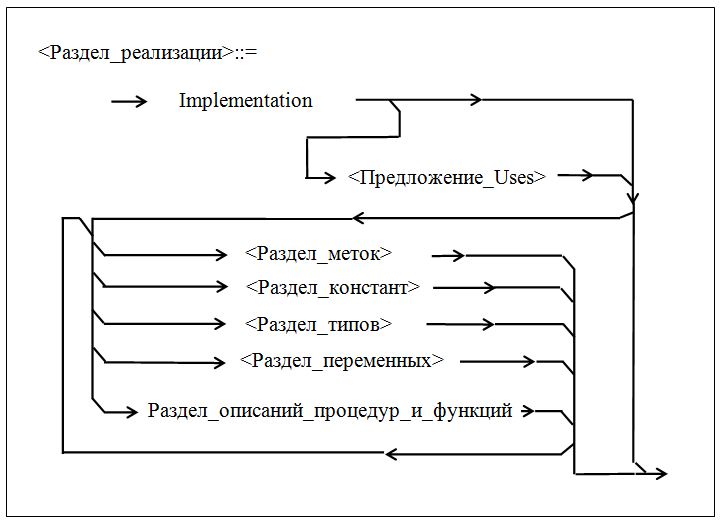


Рисунок 1.14 - Синтаксическая диаграмма раздела **реализации** + здесь полное описание внутренних процедур и функций модуля, т е вычисления процедур и функций (описанны в интерфейсной секции). Внутренние подпрограммы **недоступны** вызывающей программе или другим модулям –**например???.**

Все элементы интерфейсной секции доступны в секции реализации. Все, что описано в секции реализации, **недоступно** программе или **другим модулям.** Это – скрытая часть модуля.

Описанные в **Implementation** типы, константы, переменные – глобальные в отношении к подпрогам этого раздела + операторам раздела инициализации.

Если в **Implementation** в телах подпрограмм или при объявлении типов, переменных используются имена, объявленные в других модулях, и этих модулей нет в **Uses** **Interface*,*** то их перечисляют в **Uses** после слова **Implementation*.***

Если в модуле есть обращения к внешним процедурам /ф-циям на **Assembler,** то после заголовка такой процед-ы/ф-ции вместо тела (в **Implementation)** записывают директиву **External** – говорит, что тело процедуры / ф-ции описано в другом модуле (на *Assembler*, файл с расширением **obj**).

В **секции инициализации -** операторы, которые выполняются до операторов из тела программы (установки стартовых значений).

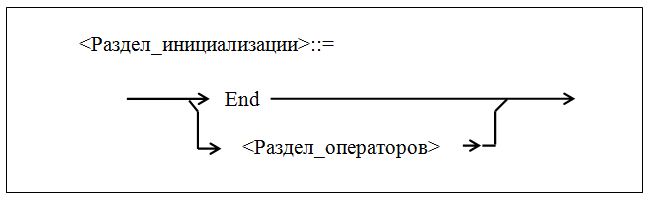


Рис 1.15 - Синтаксическая диаграмма раздела **инициализации**

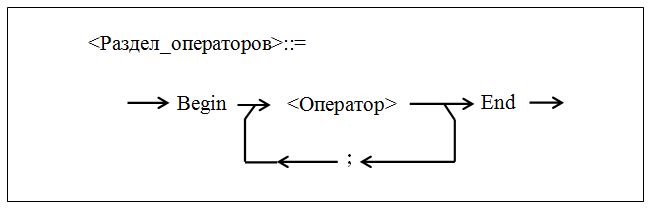


Рис 1.16 - Синтаксическая диаграмма раздела **операторов**

Если программный модуль использет несколько модулей ***Unit*,** то их части инициализации будут выполняться в порядке их перечисления в предложении **Uses** программы до операторов тела программы. Если операторной части в секции инициализации **нет**, то секция состоит из слова ***End***

Все разделы модуля - необязательные, но **служебные слова**, начинающие разделы в ***Unit*** **- обязательны.**

**Пример 1.13.** Стр-ра **пустого модуля**: только служебные слова.

**Unit Hаllo; {Пустой модуль}**

**Interface**

**Implementation**

**End.**

**Пример 1.14.** Абстрактная программа, отражает правила оформления 1.модуля ***Unit*** и 2.программного модуля **( - главная программа)**. Подчеркнутые служебн. слова – д б в ***Unit***.

**Unit U1; {Раздел Unit модуля U1}**

**Interface {1. Интерфейсный раздел}**

**Uses Dos;** {**Dos** – стандартый модуль в библиотеке}

**Const**

**MyValue = 724;**

**Type**

**Ned = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);**

**Var**

**D: Ned;**

**Procedure SetNed (Var Den: Ned);**

**Function WeekEnd: Boolean;**

**Implementation {2. Раздел реализации}**

**Uses U2;**

**Var**

**D1, D2, D3: Ned;**

**Procedure SetNed;**

**Begin**

**<Тело процедуры SetNed>**

**End;**

**Function WeekEnd;**

**Begin**

**<Тело функции WeekEnd>**

**End;**

**Inicialization**

**Begin {3. Раздел инициализации}**

**D := Pn**

**End. {========= Конец модуля U1 =================}**

**Program Main; {**Программный модуль **(главная программа)}**

**Uses U1**; {Предложение использования в программном модуле}

**Var**

**X: Ned;**

**B: Boolean;**

**I: Integer;**

**...**

**Begin**

**...**

**SetNed (X); {Вызов процедуры, описанной в модуле U1}**

**...**

**B := WeekEnd; {Вызов функции, описанной в модуле U1}**

**...**

**I := MyValue; {Обращение к константе в модуле U1}**

**...**

модуль *U1* д б помещен в файл **U1.pas**, затем откомпилирован с директивой

**Destination = Disk.** Результат компиляции – файл **U1.tpu.**

-------------

Чтобы использовать модуль в программе, его имя надо записать в предложении **Uses вызывающего** программного модуля (**Program Main**).

Чтобы найти модуль, указанный в предложении **Uses** программы, компилятор сначала просматривает библиотеку **Turbo.tpl** ( - библиотека стандартных модулей *–* **Dos, Crt** и т.д.). Если нужного модуля здесь нет, то он ищется в текущем каталоге (в том, где есть программный модуль), а затем – в каталоге модулей, заданном командой **Options / Directories / Unit Directories**интегрированной среды Паскаль. Поэтому желательно файл **U1.tpu**  ложить или в текущий каталог, или в каталог модулей (его можно задать в окне ввода **Options / Directories / Unit Directories**).

Если файл модуля находится в другом месте или имя файла не совпадает с именем модуля (например, модуль ***Unit U1*,** а имя файла, где он лежит – ***Unit1.pas***), то компилятору надо передать нужное имя файла (если он в другом месте – то с путем к нему) с помощью директивы компилятора

**{$U <Имя\_файла>}**

Этой директиве передается **имя** файла с расширением **pas**. Директиву помещают перед именем модуля в предложении **Uses:**

**Uses Dos, Crt, {$U Unit1.pas} U1;**

1.6.3. Особенности работы с модулями

**1.** Подключение модулей – в порядке их перечисления в предложении **Uses** слева направо. В этом же порядке срабатывают разделы инициализации модулей. Инициализация происходит только при работе программы. При подключении модуля к модулю инициализ-ии **нет.**

**2. Порядок** подключения влияет на доступность библиотечных типов, данных, процедур и функций.

Пусть есть 2 модуля ***U1*** и ***U2***. В каждом из них в интерфейсной секции описаны одноименные тип ***Ned*,** переменная ***D***, процедура ***SetNed*.** Но они реализованы по-разному. Если в программе записано предложение использования **Uses U1, U2;**

то обращения к элементам **Ned, D, SetNed(Х)** б равносильны обращениям к модулю **U2**. Если в главной проге **тоже** объявляют эти же имена, то они заменяют собой имена, относящиеся к модулю U2/U1. Тогда, д/доступа к содержимому нужного модуля, используют **составные имена**: в первой части – имя модуля:

**U1.Ned, U1.D, U1.SetNed(X)**

Т о, если вводят одинаковые идентификаторы, то надо учитывать порядок их подключения или гарантировать корректность обращения, указывая **явно** принадлежность библиотечных подпрограмм, данных или типов к конкретному модулю.

**3**. Решение проблемы **закольцованности** (циклических ссылок модулей друг на друга, взаимн рекурс.). Например, модуль ***U1*** использует элементы из модуля ***U2*,** а модуль ***U2*** – элементы из ***U1*.** Решение проблемы зависит в каком разделе возникла закольцованность.

Если оба модуля подключают друг друга **в разделе реализации:**

**Unit U1;**

**Interface**

**Implementation**

**Uses U2;**

**…**

**Unit U2;**

**Interface**

**Implementation**

**Uses U1;**

***…***

то закольцованность автоматически разрешается **компилятором**, т к Паскаль для обоих модулей выпол-ет полную компиляцию интерфейсных секций (а описания в интерфейсных секциях аналогичны опережающему описанию процедур и функций с помощью **директивы** **Forward).**

Но если хотя бы 1 из модулей подключает другой в разделе **Interface,** то проблема закольцованности решается только **программным путем**. Тогда используют **3й модуль**. В него помещают все те типы/ переменные/ подпрограммы, которые ссылаются друг на друга в первых 2х модулях. Затем они удаляются из первых 2х модулей, и к этим модулям подключается 3й модуль (независит от своего предложения ***Uses)***

**Пример 1.15.** Пусть **Unit U1** из примера 1.14 в **Implementation** подключает модуль ***U2*.** Пусть **модуль *U2*** имеет **интерфейс**ный раздел:

Unit U1;

**Interface**

……..Ned;

Implementation

Uses U2;

…

**Unit U2;**

**Interface**

**Uses U1;**

**Procedure Dni (Var Dd: Ned);**

**…**

Тип **Ned** объявлен в модуле ***U1*.** Т к на него есть ссылка в интерфейсной секции ***U2*,** то в директиве ***Uses*** модуля ***U2*** надо подключить модуль ***U1*.**

Транслятор такую закольцованность не пропустит. Решение – создать 3й модуль ***U3* -** туда вынесут **объявление типа Ned**:

**Unit U3;**

**Interface**

**Type**

**Ned = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);**

**Implementation**

**End.**

Из модуля *U1* (см. пример 1.14) надо убрать объявление **Ned.** Из предложения **Uses** U2 – убрать подключение модуля ***U1***. В модулях *U1* и *U2* в интерфейсных разделах надо подключить модуль ***U3***. В результате предложения ***Uses*** в интерфейсных разделах модулей *U1* и *U2* примут вид:

***Uses Dos, U3****;* {В модуле U1}

***Uses U3;*** {В модуле U2}

*«+»* *использования модулей* ***Unit*:**

1. Наличие модулей позволяет использовать модульное программирование, то есть представлять программу в виде модулей и при необходимости корректировать отдельные модули, а не всю программу.
2. Модули компилируются независимо друг от друга; при подключении модуля к другой программе он не компилируется заново, что сокращает время компиляции больших программ. Аналогично и при корректировке отдельных модулей – заново компилируются только зависящие от них модули.
3. Модули позволяют создавать большие программы с суммарным размером исполнимого кода ( > 64К).

1.6.4. Подключение к программе внешнего файла

**2й способ подключения внешних** процедур и функций к программе - использовать директивы компилятора **«Включение в программу внешнего текстового файла» -  *{$I <Имя\_файла>} (И-и-и-и раз, И-и-и-и два)***

Она сообщает компилятору: включить в компиляцию названный файл – отлаженные законченные блоки программы м записать в отдельные файлы и заменить в программе на директивы их включения. Это замедляет компиляцию, но экономит место в программе (она превращается в цепочку подключаемых файлов) и упрощает отладку программ. Примеры включения исходных файлов:

**{$I F1.pas}** или эквивалентно **{$I F1}, {$I C:\Dir1\Proc1.ini}**

По умолчанию расширение <Имени\_файла> -- **pas.**

**Включаемый файл** должен отвечать **условиям:**

1. при его включении на место директивы **{$I …}** он должен вписаться без ошибок в структуру и смысл программы;
2. он должен содержать законченный смысловой фрагмент: блок от **Begin** до **End** должен храниться в 1ом файле целиком (например, тело процедуры);
3. включаемый файл нельзя указывать в середине раздела операторов.

Включаемые файлы сами могут содержать директивы **{$I …}.** Максимальный уровень такой вложенности = **8**.

«-» такого подключения к программе внешнего файла по сравнению с использованием библиотечных модулей: а) подключаемые файлы каждый раз компилируются заново. Это увеличивает время компиляции; б) размер программы не может превышать 64К.

## Раздел 2. Простейший ввод-вывод

2.1. Процедуры ввода из стандартного текстового файла Input (read / readln)

По умолчанию устройство ввода данных – клавиатура, связано со стандартным текстовым файлом **Input,** э**кран** – устройство вывода данных, – **Output**.

Операторы вызова процедур ввода из стандартного текстового файла I**nput** записывают так:

**Read (X1, X2, …, Xn); Readln (X1, X2, …, Xn);**

**X1, X2, .. , Xn –** переменные, элементы списка ввода **(фактич**еские параметры процедур ввода).

Процедуры читают символьные данные из файла **Input** и присваивают их переменным **Xi,** при этом символьные данные преобразуют к типу переменных ***X***.

**Tипы переменных X** (5шт):

* целочисленные или их диапазоны;
* вещественные; real
* *Char* или его диапазон;
* String;
* Array Of Char.

***Процедура Readln*** после чтения элементов списка ввода переходит к следующей строке файла ***Input*.** Поэтому по следующей процедуре ***Read*** или ***Readln*** будут читаться данные из следующей строки входного файла. Для процедуры **Readln** список ввода м опустить.

**Пример 2.1.** Пусть есть фрагмент программы:

**Var**

**X, Y, Z: Integer;**

**...**

**{\*} Readln (X, Y);**

**{\*\*} Readln (Z);**

***...***

Пусть во входном файле ***Input*** есть значения:

**248 −15 4 70** – значения 1-й строки (X, Y);

**11** – значения 2-й строки (Z)

Тогда после выполнения процедур ввода {\*} и {\*\*} переменные *X, Y, Z* примут значения:

X = 248 Y = -15 Z = 11

**Процедура Read** после чтения элементов списка ввода - переход на число прочитанных символов данной строки.

Для примера 2.1 выполнение процедур **Read (X, Y) и Read (Z)**вместо**{\*}** и **{\*\*}**даст результат X = 248 Y = -15 **Z = 4.**

**Особенности ввода:**

1) **Тип** вводимого данного из входного файла должен соответствовать типу переменной ***Xi*** из списка **фактич**еских пар-ов процедур ***Read, Readln*.**

2) Если в процедуре ввода ***Read*** в списке несколько переменных, то во входном файле они должны отделятся друг от друга **разделителями** (д/числовых значений – это пробелы), а д/переменных типа ***Char*** – д б **б/разд**елителей и б/окаймляющих **апострофов**:

**Пример 2.2.** Ввод переменных типа ***Char***.

**Var**

**A, B, C: Char;**

-------------------------

**Read (A, B, C);**

-------------------*------*

Пусть во входном файле **Input** записано значение «**End»**. Тогда после выполнения процедуры ***Read(A,B,C)*** переменные примут значения:

A = ’E’ B = ’n’ C = ’d’

3) Для переменных ***Xi*** типа **String** чтение символов из входного файла продолжается, пока не будет достигнута максимальная длина **Xi**, или до конца строки файла. Переход на новую строку процедура **Read** не дает. Поэтому надо использовать **Readln.**

4) Для переменных типа **Array Of Char** каждому элементу **массива** присваивается очередной прочитанный **символ.**

2.2. Процедуры вывода в стандартный текстовый файл Output (write / writeln)

– имеют вид:

**Write (E1, E2, …, En); Writeln (E1, E2, …, En);**

**Ei** – эл-ты списка вывода: выражения, строковые константы /переменные. Для **Writeln** список ***E*** может быть **опущен**.

**Типы** выражения ***Ei*** (6 шт):

1. Целочисленные ( или их диапазоны);
2. Вещественные **real;**
3. **Char** (или его диапазон);
4. **Boolean;**
5. **String;**
6. **Array Of Char.**

Данные типы при передаче в выходной файл преобразуются в **символьный вид** и разбиваются на **строки** фиксированной длины: вывод на экран – **80 позиций**, на печать – **128**.

Для вывода используется **буфер,** в котором предварительно формируется строка символов д/вывода.

Если использовать процедуру **Write,** то выходная строка выводится только после заполнения буфера. Переход на следующую выходную строку - только после заполнения текущей строки.

Для**Writeln** переход на следующую строку - после вывода всех элементов из списка ее фактических параметров.

Процедура **Writeln** б/параметров – просто переход на новую строку.

Количество позиций поля (в строке выходного файла) для всех типов выходных данных (кроме вещественных) = минимально необходимому.

Для вывода вещественных типов – фиксированное = **17** позиций в виде мантиссы и порядка.

Если тип переменной вывода ***Еi*** *–* **Boolean,** то выводится **True / False**

***«-»*** процедур вывода с элементами списка вывода вида ***Еi*** – жесткая фиксация выводимых значений по позициям строки.

Д/управления выводом по позициям строки исп-ют элементы вывода вида

**Е: L1** или **E: L1: L2**

**L1, L2** – выражения **целого** типа.

**L1** – длина поля д/элемента ***Е***.

Если значение **L1 *>*** к-ва позиций, то данные выравниваются по **правой** границе поля (т.е. поле дополняется пробелами слева).

Если поле ***L1*** < нужной длины, оно автоматически расширяется до минимально необходимого.

***L2*** используют если **Еi** – выражение вещественного типа. Если в этом случае **L2**нет, то вещественное число будет выведено в виде мантиссы и порядка (рис 2.1: ***R*** – разрядность дробной части мантиссы). Если ***L1*** больше необходимой длины, выводимое вещественное число ***Real*** занимает фиксированное число позиций по умолчанию = 17 и поле дополняется пробелами **слева**. При 17 ≥ **L1** *≥* 8 ***R = L1 – 7.***

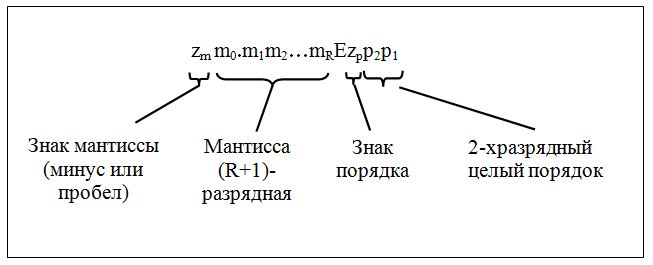


Рис 2.1 – Вид выводимого вещественного числа типа **Real** в форме с **плавающей точкой**

If **L1 < 8** поле вывода числа **Real** расширят до **8** позиций (**L1=8**).

Если **L2** указано, то вещественное число представляют в форме с **фиксиров**анной точкой, а значение **L2** определяет количество цифр в дробной части числа.

Например**,** имеется оператор (пробел обозначен символом ):

**Write (1.2546: 10: 2);**

Данный оператор в выходной файл поместит такое значение:

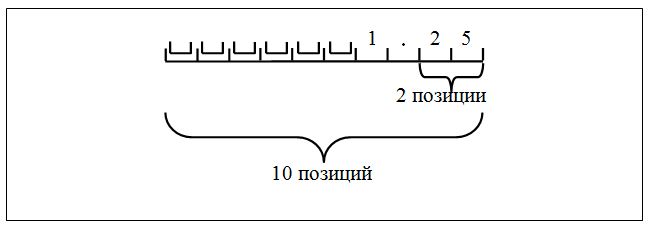


Рис 2.2 – Значение, выв-ое опером выходной файл **Write (1.2546: 10: 2)**.

**Пример 2.3.** Пусть *Y* и *Z -* **массивы.** Значения ***Y(X)*** и ***Z(X)*** заранее подсчитаны. Значение ***Х*** изменяется от 1 до 40. Вывести на экран таблицу знач-ий (рис 2.3), отводя под столбцы и значения заданное к-во позиций (поз.) вывода.

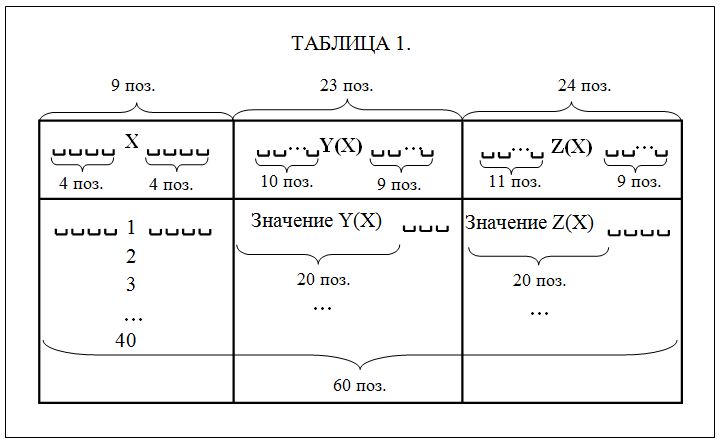


Рис 2.3 – Вид выводимой таблицы (так захотели)

Фрагмент программы, выполняющий формирование таблицы:

--------------------------------------------

***Writeln (‘******ТАБЛИЦА******1.’);***

***For I := 1 To 60 Do Write (‘\_’);*** {Верхняя строка отчеркив-ия шапки}

***Writeln;***{Переход на новую строку}

***Writeln (‘|’, ’X’: 5, ’|’: 5, ’Y(X)’: 14, ’|’: 10, ’Z(X)’: 15, ’|’: 10);*** {Вывод

шапки таблицы}

***For I := 1 To 60 Do Write (‘\_’);*** {Нижн строка отчеркив-ия шапки}

***Writeln;*** {Переход на новую строку}

***For X := 1 To 40 Do***

***Writeln (‘|’, X: 5, ’|’: 5, Y(X): 20: 6, ’|’: 4, Z(X): 20: 6, ’|’:5);***

{Вывод содержимого таблицы}

***For I:=1 To 60 Do Write (‘\_’);***{Строка подчеркивания таблицы}

***Writeln;*** {Переход на новую строку}

## Раздел 3. Записи = Record

3.1. Структура записи

**Записи ==** **комбинированный тип данных** – **тип Record**.

**Запись** – структура данных, состоит из **разнород**ных компонентов (иерархически **упорядоче**нных). Отличие от массивa - компоненты записей могут иметь разные типы, + доступ к ним - по **именам** (не по индексам).

**Поля =** компоненты записей**.** Нет ограничений на **тип** поля записи, поэтому компонентой записи может быть тоже запись = **иерархическая запись**. Уровень иерархии (вложенности) **<= 9.**

Пример простой (не иерархической) записи – комплексноe числo ***a + b\*i***:

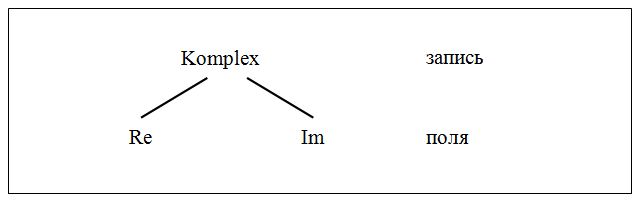


Рис 3.1 – Пример неиерархической записи

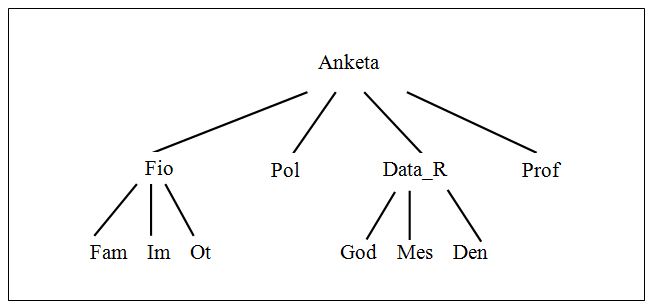


Рис 3.2 - Пример **иерархич**еской записи Anketa

Cинтаксис задания **структуры** данных типа запись имеет вид (рис 3.3):

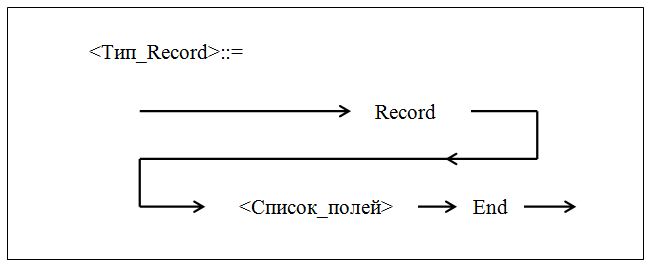


Рис 3.3 – Cинтаксическая диаграмма **задания данных** типа запись

Список полей определяют с помощ синтаксической диаграммы:

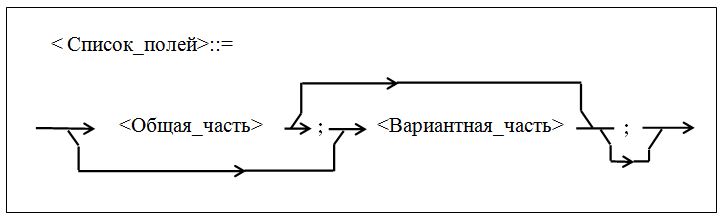


Рисунок 3.4 – Синтаксическая диаграмма **списка полей**

Запись может состоять из частей: **1.общей + 2.вариантной.** Любая из них- необязательная.

**Поле** записи обозначается идентификатором. К полю обращаются **по имени**. Область действия полей записи - сама запись. Имя каждого поля внутри записи должно быть **уникальным.**

3.2. Записи без вариантной части (неиерархической записи)

- только общая часть. Обычная запись без гемора вложений.

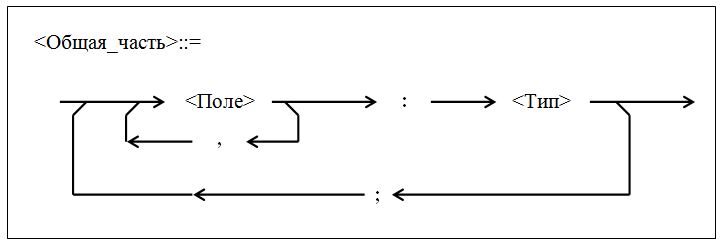


Рис 3.5 – Синтаксическая диаграмма задания **общей** части записи

Каждому **полю** записи дают свое имя и задают тип значения каждого поля

**Пример 3.1.** Объявление записи (её структура – на рис 3.1)

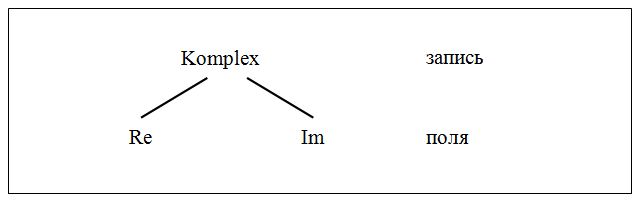


Рис 3.1 – Пр-р **не**иерархической записи

**Type**

**Komplex = Record**

**Re: Real;**

**Im: Real**

**End;**

или эквивалентно:

**Type**

**Komplex = Record**

**Re, Im: Real;**

**End;**

В разделе **Var** ввести переменные типа **Komplex**:

**Var**

**X, Y: Komplex;**

**2 способа** определения типа поля записи:

1). задать прямо в описании записи,

2). описать предварительно – здесь указывается имя типа.

**1). Пример 3.2.** Объявление записи, её структура:

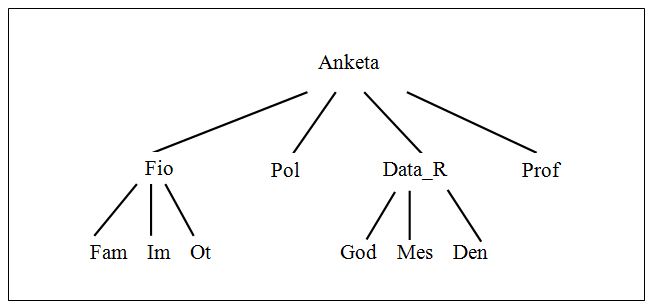


Рис3.2 – **Иерар**хическая запись

**Type**

**Data = Record**

**God: 1900..2000;**

**Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);**

**Den: 1..31**

**End;**

**Anketa = Record**

**Fio: Record**

**Fam: String [20];**

**Im: String [10];**

**Ot: String [20]**

**End;**

**Pol: (Man, Woman);**

**Data\_R: Data;**

**Prof: String [20]**

**End;**

2). В разделе **Var –** ввести переменную типа **Anketa:**

**Var**

**An1, An2: Anketa; - An1, An2,** – полные переменные (полные имена)

**D1, D2: Data;**

Объем памяти для записи = ∑ длин полей нижнего уровня.

**Полная переменная -** имеющая тип записи **верхнего уровня**.

Обращение к **значению поля** - с помощью идентификатора полной переменной, идентификаторов всех полей (с учетом их иерархии), в состав которых входит поле, и имени данного поля, разделенных точкой. Такое имя называется **составным именем**.

Например, д/примеров 3.1, 3.2, **An1, An2, X, Y** – **полные** переменные (полные имена). **Составные** имена записывают: **X.Re, X.Im, An1.Pol, An1.Fio.Fam, An1.Data\_R.God** и т.д.

Для **полных** (толстых) переменных 1 и того же комбинированного типа есть только 1 операция – **операц. присваивания** (в правой части оператора := м использовать только переменную того же типа запись). Для примеров 3.1, 3.2 можно записать:

**X := Y; An1 := An2;**

К этому моменту записи ***Y, An2*** уже д б определены.

Составное имя м использовать везде, где допустимо применение типа поля. Для присваивания полям значений используют оператор **:=** или ввода.

**Пример 3.3.** Присвоение значений **полям** переменной **D1** (к пр 3.2).

**D1.God := 1970;**

**D1.Mes := Jn;**

**D1.Den := 15;**

Для определения значения **полной** переменной надо :=ть значения всем полям (образующим это значение).

**Составные имена** используют в операторах ввода-вывода. Напр-р:

**Read (D1.God, D1.Den); Write (A1.Fio.Fam);**

Удобно пользоваться **массивами** из записей. Например, д/примера 3.2 м объявить массив в разделе var:

**Var**

**Spisok: Array [1..100] Of Anketa; 100 -** человек

Так м хранить анкетные данные на 100 человек, выделять и обрабатывать из них нужные. Напр-р, вывести знач-ия полей записи:

**For I:=1 To 100 Do**

**Writeln (Spisok [I].Fio.Fam, Spisok [I].Fio.Im, Spisok [I].Fio.Ot)**

3.3. Записи с вариантами

Записи, содержащие только общую часть, имеют строго определенную структуру.

Записи **с вариантами** объединяют похожие описания записей, но не идентичные по форме.

Они состоят из частей: 1.необязательной **общей** + 2.**вариантной**.

1). Общая (необязательная) часть описывает поля, которые есть в каждой переменной определяемого типа.

2). Вариантная часть начинается зарезервированным словом **Case** и описывает несколько вариантов структуры записи. Её синтаксис:

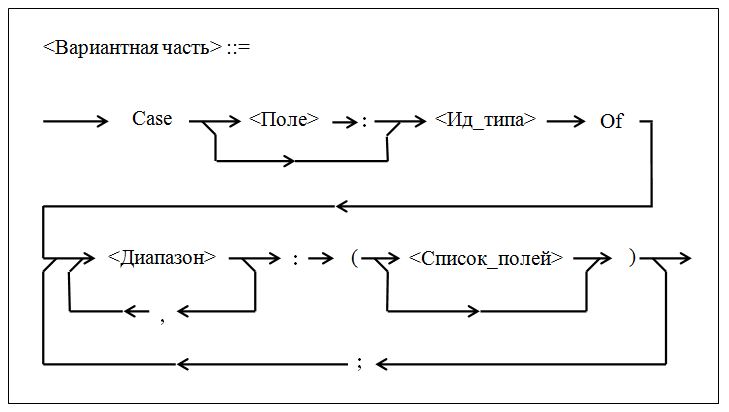


Рис 3.6 – Синтаксическая диаграмма вариантной части записи

Диапазон (см. рис 3.6) имеет формат (рис 3.7):

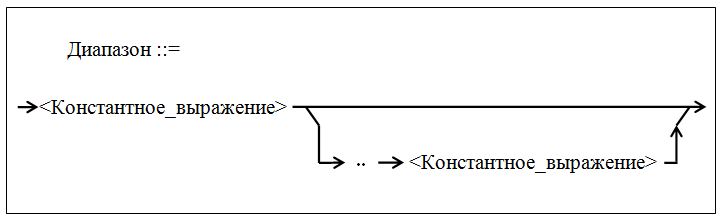


Рис 3.7 – Синтаксическая диаграмма диапазона

В каждый момент выполнения программы 1 из вариантов структуры – активный в зависимости от значения признака варианта.

**Признак варианта** описывается сразу после **Case.**

**Признак** –самостоятельное поле общей части записи.

Например **вриантная часть записи** м им вид:

а) **Case I: Integer Of ...**

Здесь идентификатор **I** определяет (обзывает) поле признака, которое является **самосто**ятельным полем общей части записи.

б) Поля признака может не быть (**I**). Тогда в **Case** указывают только **идентификатор типа** (**Integer Of**), и прогер должен сам следить какой вариант записи – активный.

**Case Integer Of ...**

Здесь **поля** признака нет. Варианты (1,2,..) определяют целыми значениями в списке вариантов. Список полей каждого варианта – в скобки.

**Пример 3.4.** Запись с вариантами **без поля признака**.

**Var**

**Z: Record**

**Case Integer Of**

**1: (I1: 1..10);**

**2: (J1: Char);**

**3: (K1: Boolean)**

**End;**

Здесь **вариантная часть** содержит **3** варианта. Будет активен вариант, соответствующий имени поля. Например, если в программе – обращение к полю **Z.I1**, то вариантная часть воспримется как поле типа **1..10.** При обращении к вариантной части по имени **Z.J1** она воспримется как поле типа **Char.**

Слово *End* заканчивает **всю** конструкцию записи с вариантами.

**Пример 3.5.** Объявление записи с вариантами **с полем** признака.

**Type**

**Anketa1 = Record** {===== Общая часть ==============}

**Fio: Record**

**Fam: String[20];**

**Im: String[10];**

**Ot: String[20];**

**End;**

**Case Pol: (Men, Women) Of** {= Вариантная часть ===}

**Men: (Vozr1: 20..30);**

**Women: (Vozr2: 18..25)**

**End;**

В вариантной части все имена полей должны быть **уникальны**, даже если они есть в разных вариантах.

Запись м иметь только 1 вариантную часть, она д б в конце записи.

Вариантная часть м б **вложенной** в другую вариантную часть.

Если вариантная часть пустая (соответствующая какому-то значению признака), она записывается так:

**<Диапазон>: ()**

Например, если в пр 3.5. нет поля при значении признака, = Women, то этот вариант запишется так:  **Women: ()**

Объем **памяти** для записи с вариантами = ∑ объемы полей **общей** части + ∑ длин полей **вариантной** части максимальная по объему.

Поля записи размещают в памяти последовательно, в соответствии с объявлением.

3.4. Оператор присоединения With

При работе с полями в их составном имени надо писать **путь** к полю через все уровни иерархии, начиная от полного имени записи. Т к уровней иерархии **< 9**, **составное имя** поля м иметь 9 компонент. Работать с ним неудобно, а программа громоздкая.

Для сокращения составного имени поля м использовать оператор присоединения ***With***, имеет формат:

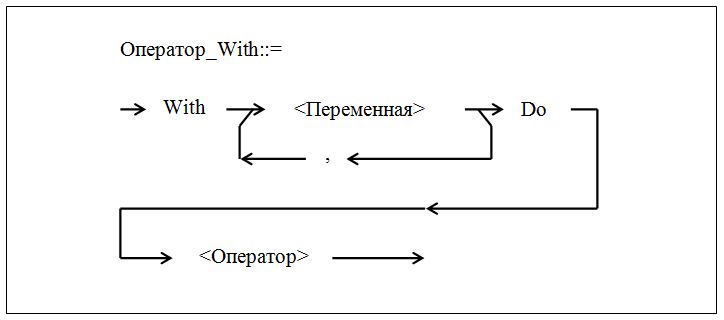


Рис 3.8 – Синтаксическая диаграмма оператора присоединения

Здесь <Оператор> - это любой оператор языка, допустимый для соответствующих типов полей.

В операторе ***With*** указывается список переменных типа ***Record*.**

Оператор ***With*** облегчает доступ к полям этих записей и минимизирует повторные адресные вычисления. К полям записей внутри <**Оператор**> (в ***With)*** можно обращаться как к простым переменным.

**Пример 3.6.** В пр 3.2 объявлена перем ***D1: Data*.** С пом ***With*,** вместо пр 3.3 м:

**With D1 Do**

**Begin**

**God:=1970;**

**Mes:=Yan;**

**Den:=15**

**End;**

Пример 3.3.Присвоение знач-й **полям** переменной **D1** (к прим-у 3.2)

D1.God := 1970;

D1.Mes := Jn;

D1.Den := 15;

**Пример 3.7.** С пом ***With***, к **полям** записи ***An1*** (см. пр 3.2) м обратиться так:

Type

Data = Record

God: 1900..2000;

Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);

Den: 1..31

End;

**Anketa** = Record

Fio: Record

Fam: String [20];

Im: String [10];

Ot: String [20]

End;

Pol: (Man, Woman);

Data\_R: Data;

Prof: String [20]

End;

Var

**An1,** An2: **Anketa;,**

D1, D2: Data;

**With An1 Do**

***Begin***

**Fio.Fam := ’Иванов’;**

**Fio.Im := ’Петр’;**

**Fio.Ot := ’Степанович’;**

**Pol := Man;**

**Data\_R.God := 1970;**

**Data\_R.Mes := Yan;**

**Data\_R.Den := 25;**

**Prof := ’Студент’**

***End;***

Без оператора ***With*** перед всеми **именами полей** надо было бы писать имя записи ***An1***.

Адрес переменной типа ***Record*** вычисляется до выполнения оператора ***With*.** До завершения оператора ***With*** любые изменения переменных, влияющие на вычисленное значение адреса, не отражаются на значении этого адреса.

**Сокращенная форма** оператора **With** – список переменных типа запись в операторе ***With***:

**With** Z1, Z2, Z3, …, Zn **Do <Оператор>**

=сильна **полной форме** оператора **With**:

**With Z1 Do**

**With Z2 Do**

**With Z3 Do**

**. . .**

**With Zn Do <Оператор>**

Из полной формы оператора ***With*** видно, что идентификатор поля **Zn** в **<Операторе>** обозначает компонент записи из ближайшего объемлющего оператора ***With*,** в котором указана переменная с таким полем.

**Пример 3.8.** Д/примера 3.2 для полного удаления в составных именах - полей (см. пр 3.7) м исп **сокра**щенную форму ***With*:**

**With An1, Fio, Data\_R Do**

*Begin*

**Fam := ’Иванов’;**

**Im := ’Петр’;**

**Ot := ’Степанович’;**

**Pol := Man;**

**God := 1970;**

**Mes := Yan;**

**Den := 25;**

**Prof := ’Студент’;**

*End;*

Если у 2х переменных из списка записей ***With –*** поля с 1 и тем же **идентиф**икатором, то внутри ***With*** он обозначает поле **последней** указаной переменной.

**Имена** отдельных полей записи в программе могут совпадать с **именами** переменных. Имя переменной в ***With* недоступ**но.

**Пример 3.9.** Использование **совпадающих** идентификаторов.

**Var V: Record {============ Запись =======}**

**V2: Integer;**

**V1: Record**

**A: Real**

**End;**

**A: Integer**

**End;**

**A: Char;** **{==== Переменная =====}**

**. . .**

**With V, V1 Do** (по цепочке спускаемся вниз)

***Begin***

**V2 := 1;** {значение поля **V.V2 = 1**}

**A := 1.0;**  {значение поля **V.V1.A = 1.0**}

**V.A := 1;**  {значение поля **V.A = 1**}

***End;***

**A := ’A’;**

Здесь 3 элемента с именем ***А*** – **поля** ***V.V1.A*, *V.A*** и переменная ***А*** типа **Char**. Указание имени поля ***А*** в **<Операторе>** оператора ***With*** приведет к обращению к полю ***V.V1.A*,**  т к поле ***А*** есть в записи ***V1***, находящейся в списке записей сокращенной формы оператора ***With*** после записи ***V***. Чтобы обратиться к полю **V.A,** нужно использовать его составное имя или вынести оператор (работающий с данным полем) за пределы ***With*,** где поля записи становятся недоступными, а переменная ***А*** типа ***Char –*** доступной.

3.5. Константа-запись (К-З)

- один из видов структурных типи**зов**анных констант.

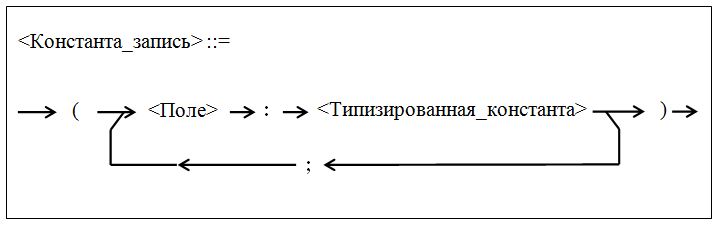


Рис 3.9 – Синтаксическая диаграмма задания **константы-записи**

Как и остальные типизованные константы, К-З м использ-ть как **инициированные** переменные типа запись – переменные, которым при запуске программы присваивается **начальное** значение.

**Пример 3.10. Объявление** константы-записи ***D***.

**Type**

**Fam = (Ivanov, Petrov);**

**Data = *Record***

**God: 1900..2000;**

**Mes: (Jn, Fb, Mr, Ap, Ma, Jn, Jl, Ag, Sp, Oc, Nv, Dc);**

**Den: 1..31**

***End;***

**Ank = Array[Fam] Of Data;**

**Const**

**D: Data = (God: 1950; Mes: Jn; Den: 3);**

**A: Ank = ((God:1970; Mes:Dc; Den:7), (God:1945; Mes:Ma; Den:15));**

***D*** – **типизованная К-З**, **А** – массив из 2х типизованных К-Зей **А[Ivanov] + А[Petrov].** В константе ***А*** внутренние скобки относятся к К-Зсям, внешние – к типизованным **константам-массивам**.

В типизованных К-Зсях поля надо указывать в том же порядке, как они идут в объявлении типа запись.

Если запись содержит вариант (**с Case**), то м указать только поля выбранного варианта. Если в варианте – поле признака, то его знач-ие д б определено.

Использовать компонент **файлового** типа в структурных К-З запрещено!

**Задание для самостоятельной подготовки**. Константы-записи с вариантами (примеры объявления и использования).

## Раздел 4. Множества (Мнж) Set

4.1. Общие сведения o Set

Множественный тип **Set** соответствует понятию множества в математике.

В Паскале допускают только конечные множества. Максимальное количество элементов здесь – **256.** Все элементы Мнж д б значениями 1 типа. Б**азовый тип Мнж** *-* тип элементов Мнж: любой скалярный, кроме вещественных. Целочисленный тип как базовый м использовать только в диапазоне **0 .. 255** или его поддиапазонов.

Данные типа ***Set*** хранят в памяти в **унитарном коде,** занимают 1…32 байта. Объем памяти 1го элемента **Set** = 1б**и**ту. Каждому элементу в **Set** жестко соответствует свой № бита.

**Пример 4.1.** Пусть Мнж имеет набор элементов **0 ...7**, а его конкретное значение = множеству элементов **0, 3, 4**. В памяти компа оно б представлено:

Таблица 4.1 – Представление множества в памяти компьютера

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0-й бит** | 1-й бит | 2-й бит | **3-й бит** | **4-й бит** | 5-й бит | 6-й бит | 7-й бит |
| **1** | 0 | 0 | 1 | 1 | 0 | 0 | 0 |

Т о, элементу 0 соответствует 0-й бит поля памяти, занимаемого мнж-ом, элементу 1 – 1-й бит и т.д. Если в значении множества элемент присутствует, то соответствующий бит устанавливается в 1, иначе - 0.

4.2. Конструктор множества

**Множество** – значение **переменной** множественного типа. Конкретные значения множественного типа задают с помощью **конструктора множества**:

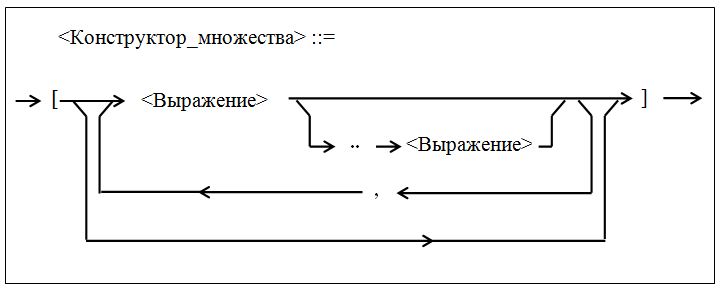


Рис 4.1 – Синтаксическая диаграмма конструктора множества

**Конструктор множества** – это список выражений / диапазонов, разделенных запятой и заключенный в квадратные скобки [1, 2, 3, 8]. При указании диапазона все значения из него входят в результирующее множество. Все значения из списка должны быть 1го типа (любого скалярного, кроме вещественных). Содержит <= 256 элементов.

**Пример 4.2.** Примеры множеств.

*[ ]* – пустое множество (не содержит элементов);

[2, 3, 5, 7, 11] – множество простых чисел от 2 до 11.

*[‘A’* ***..*** *‘Z’]* – множество латинских букв.

*[1****..****10, 100****..****110]*  – множество целых чисел от 1 до 10 и 100 ÷ 110.

[Pn, Vt, Sr] – множество из 3-х элементов перечислимого типа.

*[X* ***..*** *5\*X]* – множ-во целых чисел от текущего знач ***X*** до выр-ия *5\*X*.

Конструктор множества – **множественная константа.**

**Порядок** перечисления **не в**ажен. Каждый элемент учитывают 1 раз.

Например, множества [1, 2, 3] и [1, 3, 2] – равные множества.

4.3. Задание множественного типа

При задании множественного типа надо задать **базовый тип** – ∑ть значений всех возможных множеств (комбинации значений базового типа).

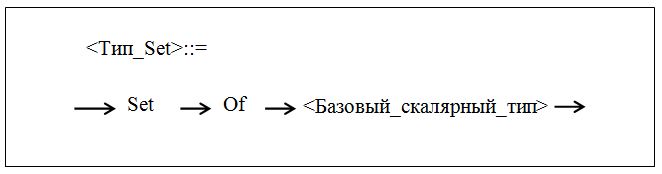


Рис 4.2 – Синтакс диаграмма задания множественного типа

<**Базовый\_скалярный\_тип**> - любой перенумер-ый тип в диапазоне **0 .. 255**

Например: **Set Of 1..3** – значения этого множеств-ого типа м б множества ***[ ], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3].***

**Set Of Boolean** – значения: **[True], [False], [True, False], [ ].**

За **базовый** тип м использовать **имя** или **задание типа**.

Задание типа ***Set*** м делатькак представление множества с помощью массива:

**Array [<Базовый\_скалярный\_тип>] Of Boolean.**

Обработка этих массивов неэффективна, поэтому используют тип **Set**

**Пример 4.3.** Объявление множественного типа.

**Type**

**Ned = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);**

**Denned = Set Of Ned;**  {Используют **имя** базового типа}

**Log = Set Of Boolean;** {Используют **имя** базового типа}

**Var**

**Den: Denned;**

**L1, L: Log;**

**I1, I2, I: Set Of 1..10;** {Исп. **задание** базового типа}

**B: Boolean;**

Den, L1, L, I1, I2, I – **множественные перем-ые** = **перем-ые-множества**

Для присваивания значений множественным переменным используют **множественное выражение**– это выражение, значение которого - **множество**. Частный случай - множественные переменные и конструкторы множества (-[1, 2, 3, 8]).

**Пример 4.4.** Использование множественных выражений. Для переменных, объявленных в примере 4.3, можно записать:

**L1 := [True];**

**L := L1;** {Значение L1 к этому моменту д б определено}

**I := [1, 3, 5];**

**Den := [Sub, Vos];**

4.4. Операции над множествами

Таблица 4.2 – Операции над множествами

| Операция | Описание операции | Тип рез-та |
| --- | --- | --- |
| **=** | Равно | **Boolean** |
| **<>** | Не равно |
| **<=** | Результат операции = **True,** если **левое** множество - подмножество правого |
| **>=** | Результат операции = **True,** если **правое** множество - подмножество левого |
| **In** | Рез-т = **True,** если нек-рое скалярн знач-е (**левый** операнд) – элемент множества (**правый** операнд) |
| Not **\*)** | Дополнение множества (одноместная операция) | **Set** |
| **+** | **Объединен**ие множеств |
| **\*** | **Пересечен**ие множеств |
| **–** | **Разность** множеств **A – B = A \* Not B** |
| Xor **\*)** | Исключающее объединение множеств  **A Xor B = A + B – A \* B** |

Здесь \*) – операции, определенные не во всех версиях языка Паскаль.

Над множественными переменными определена 1 встроенная **функция** – **Sizeof(X**): указывает количество байт для представления значения ***X*** множественной переменной.

***Операция In -*** операция **проверки** вхождения элемента в множество. **Левый** операнд должен принадлежать базовому типу, а **правый** – множественному типу (построен на основе этого базового типа).

**Пример 4.5.** Операц над множествами. Пусть есть объявы – пр 4.3

Type

Ned = (Pn, Vt, Sr, Ch, Pt, Sb, Vs);

Denned = Set Of Ned; {Используют имя базового типа}

Log = Set Of Boolean; {Используют имя базового типа}

Var

Den: Denned;

L1, L: Log;

I1, I2, **I**: Set Of 1..10; {Исп. задание базового типа}

**B: Boolean;**

**begin**

**I := [1, 3, 5];**

**B := 2 In I** {в B значение **False** (нет 2)}

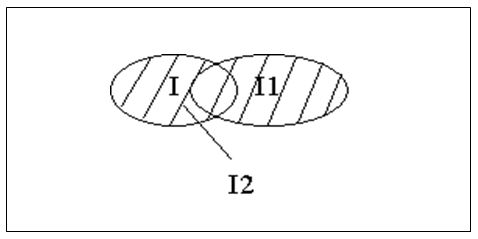
**B := [3, 5]<=I** {в B значение **True**}

**B := [4, 5]<=I** {в B значение **False**}

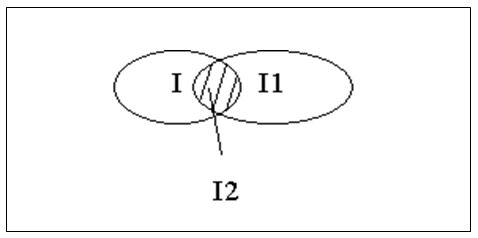
**I := Not I**  {в I значение [2, 4, 6**..**10] – **дополнение** множеств

В 2хместных операциях оба операнда должны принадлежать одному и тому же множественному **типу** значений.

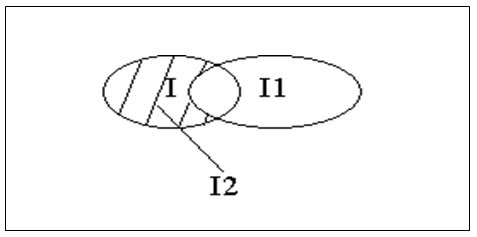
**Объединение множеств** **(I2 := I + I1)** – это множество, состоящее из элементов, входящих **хотя бы в 1** из исходных множеств ***I*** и ***I1***:

 Рис 4.3 – Объединение множеств

**Пересечение множеств** **(I2 := I \* I1)** – это множество, состоящее из элементов, входящих **в оба** исходных множества ***I*** и ***I1***:

 Рис 4.4 – Пересеч множеств

**Разность множеств** **(I2 := I – I1)** – это множество, состоящее из элементов множества ***I***, не входящих в множество ***I1***:

 Рис 4.5 – Разность множеств

**Пример 4.6.** Операц над множествами. Есть объявления в пр-ре 4.3

**I1 := [1, 2, 3];**

**I := [1, 3, 5];**

**I2 := I1 + I;** {в I2 значение [1, 2, 3, 5]}

I2 := I1\*I; {в I2 значение [1, 3]}

I2 := I – I1; {в I2 значение [**5**]}

Выражения в примере 4.6 - **множественные выражения**.

**Старшинство операций** в множественных выражениях – как в арифметических: вначале выражения в скобках ( ), затем \*, потом + и –, слева направо.

Например, результат вычисления множественного выражения

**[1, 2, 5, 6, 7] \* [2 .. 6] + [3, 9]** будет множество = **[2, 3, 5, 6, 9]**

**Пример 4.7.** Использ множественного типа. Посчитать **общее количество букв *X, Y, Z*** в исходном тексте, занчивающемся точкой.

**Program Mno;**

**Var**

**Kol: Integer;**

**B: Char;**

**Begin**

**Kol := 0;**

**Read (B);** {Чтение первой буквы текста}

**While B <> ’.’ Do**  //пока не получили точку, успеваем (осторожность)

**Begin**

**If B In [‘X’, ‘Y’, ‘Z’] Then** {if знач буквы вх в мнж-во[‘X,Y,Z’]}

**Kol := Kol + 1; //**inc(kol);

**Read (B);** {Чтение очередной буквы текста}

**End;**

**Writeln (Kol);**

**End.**

4.5. Ввод / вывод знач-ия множественной переменной

- используется *операция* **объединения (\*) множеств*.***

**Пример 4.8. Ввод** значения переменной типа множества больших латинских букв из входного **файла**. Признак окончания ввода множества – точка.

**Var**

**B: Char;**

**Mn: Set Of ‘A’..‘Z’;** {Тип элементов вводимого множества}

**Begin**

**Mn := [ ];** {**Начальное** значение множества – пустое множество}

**Repeat**

**Read (B);** **{Чтение очередного элемента множества}**

**Mn := Mn + [B]**  {Объединение множеств}

**Until B = ’.’;** {‘**.**’ – признак конца текста}

**...**

Во входном файле надо записать все значения подряд, составляющие значение этой множественной переменной. Если очередное значение **B** не относится к базовому типу множества, то его **не** заносят в множество **Mn**.

Для **вывода** значения множественной переменной – операция **проверки вхождения** элемента в множество:

**Пример 4.9.** Продолжение пр-ра 4.8.Вывод знач-ия переменной типа множества латинских букв.

**For B := ’A’ To ‘Z’ Do**  {Организац цикла по **базовому типу** множ-ва}

**If B In Mn Then** {вхожд очередн знач базовтипа в знач-ие множ-й перем}

**Write (B: 5);**

**End.**

4.6. Типизованные константы-множества (К-Мнж)

- **конструктор** множества: **значение множественной величины** (с точки зрения синтаксиса):

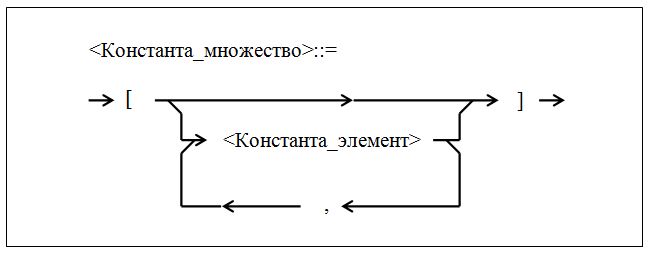


Рис 4.6 – Синтаксическая диаграмма константы-множества

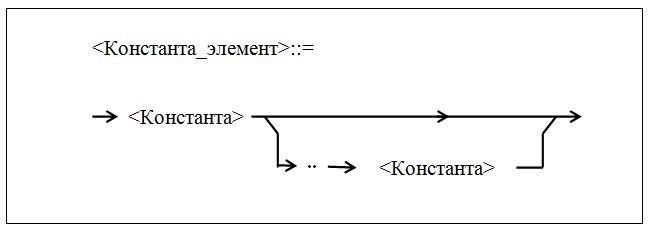


Рис 4.7 – Синтаксическая диаграмма константы-элемента

**<Константа\_элемент>** - значения / диапазоны значений базового типа множества.

К-Мнж м использ-ть как инициированную переменную типа множество.

**Пример 4.10.** Объявление **типизованной константы-множества**.

**Const**

**Dig: Set Of 0..9 = [1, 3, 5];**

**Dig1: Set Of 0..9 = [];**

**Ch: Set Of ‘A’..‘Z’ = [‘A’..‘E’, ‘I’, ‘P’, ‘T’];**

## 

## Раздел 5. Файлы

5.1. Общие сведения

Для всех рассмотренных ранее производных типов было характерно заранее заданное число компонент.

**Файловый тип** – это произвольная последовательность элементов, длина которой заранее **не определена**, а конкретизируется в процессе выполнения программы. Это – **логический файл**, т.е. который используется в программе (файл с точки зрения прогера).

**Физический файл** (набор данных) – именованная область памяти на внешнем носителе, где хранится некоторая информация (файл с точки зрения пользователя). Элементы файла записаны в нем последовательно:

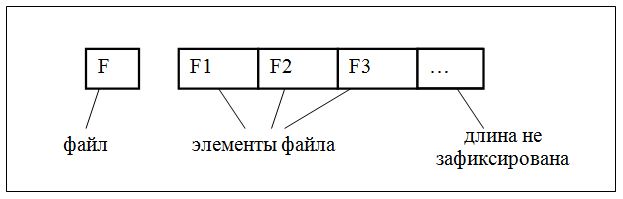


Рис 5.1 – Структура файла

**2** способа обращения к элементам файла: 1). последовательный доступ и 2). прямой доступ.

1). П**оследовательный доступ:** по файлу можно двигаться только последовательно, начиная с 1го элемента. Доступен лишь очередной его элемент. Чтобы добраться до n-го элемента файла, надо начать с 1го элемента и пройти через (n – 1) элементов.

2). П**рямой доступ –** обратиться напрямую к элементу файла с № **n,** без просмотра (n – 1) элементов файла.

Файловый тип – это (!) тип значений, с помощью которого данные получают извне (обрабатываемые программой), а результаты – переданы человеку. И (!) тип значений, который связывает программу с периферийными устройствами компьютера.

**3 типа файлов** (3 вида переменных файлового типа = файловых перем-ых):

1. **текстовые** файлы text;
2. файлы **с типом;**
3. файлы **без типа**.

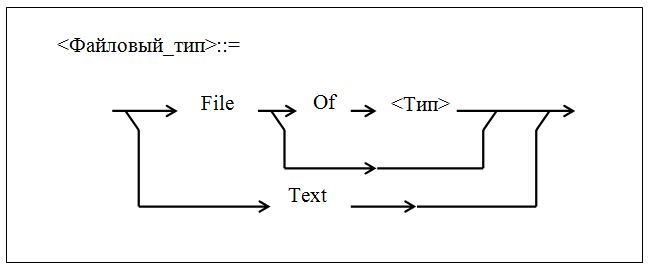


Рис 5.2 – Синтаксическая диаграмма задания фалового типа

Над значениями файлового типа **нет** никакихопераций (даже присваивание файловой переменной значения другой файловой переменной). Все операции – с **элементами** файлов, которые определяются **типом** элементов.

Для доступа к отдельным элементам файлов – стандартные процедуры и функции **ввода-вывода**. Обращение к ним - обычным образом. Для удобства описания действия этих процедур используется понятие **«окно файла»** - текущая позиция файла, указатель файла, – определяет **позицию доступа**, т.е. тот элемент файла, который **доступен** для чтения / записи.

Следующая за последним элементом позиция файла (или первая позиция пустого файла) помечается **маркером конца файла** (**Ctrl–Z**при создании текстового файла с клавиатуры), с его помощью определяется конец файла.

5.2. Процедура Assign

Любым процедурам **ввода-вывода** предшествует процедура **Assign.** Ее нельзя применять к уже открытому файлу. Формат ее заголовка:

**Assign (Var** F**; Name: String) (**assign(f, 'input.txt); **)**

***F*** – **имя** файловой переменной (любого типа); **Name** – выражение строкового типа.

Назначение процедуры **Assign** – организует **связь** между конкретным физическим файлом на внешнем устройстве (конкретным набором данных) и файловой переменной программы (**логическим** файлом) **F**.

**Имя** конкретного набора данных определяется переменной **Name** – это полное имя физического файла. имеет вид:

**<Диск>:\<Имя\_каталога>\…**\<Имя\_каталога>\<**Имя\_файла>**

**D:\MyFiles\First.txt, A:\Katalog1\Katalog2\Rez.pas**

**<Диск>** задается символом от ***A*** до ***Z*** (символ логического устройства (C, D, F, E)). Если он опущен, то – принятое по умолчанию.

**\<Имя\_каталога>\**…\<Имя\_каталога>\ - путь через подкаталоги к фактическому имени файла. Если они опущены, то файл – в текущем каталоге (папке?).

**<Имя\_файла>** - фактическое имя файла, max **8 символов** + тип файла – max **3 символа**, отделенное точкой от имени. Помогает работать с файл-и. Например, имена файлов Rez: Rez**.pas,** Rez**.exe,** Rez**.txt (.dat)**

Max длина полного имени файла – **79 символов*.***

Чтобы связать файловую переменную (логический файл) ***F*** с физическим файлом **Rez.dat** надо записать процедуру **Assign** с параметрами:

Assign (F, 'Rez.dat');

или организовать связь в диалоге:

**Var**

St: String; // - имя физического файла

...

**Begin** ...

Writeln ('Введите имя файла'); Readln (**St**);

Assign (F, **St**);

Вместо имени физического файла **St** в качестве параметра **Name** в списке параметров **Assign** м использовать любое устройство **ввода-вывода** (клавиатура, печать, дисплей и т. п). Тогда **Name** – символическое (логическое) имя устройства **ввода-вывода: символические имена устройств**:

**1. Con** – устройство **кон**соли (при вводе – это **клава**, выв – **экран**).

Например, процедура: **Assign (F, 'Con');**

означает ввод **в** переменную ***F*** с клавиатуры или вывод **из** ***F*** на экран.

По умолчанию стандартные текстовые файлы **Input** и **Output** связаны с консолью, что соответствует фрагменту программы:

**Assign (Input, 'Con'); Assign (Output, 'Con');**

**2. Lpt1, Lpt2, Lpt3** – устройства печати. Если подключено 1 устройство печати, то используется либо имя **Lpt1***либо***Prn (принт):**

**Assign (F, 'Prn'); Assign (Output, 'Prn');**

С этими логическими устройствами м использовать только **имя выходного** файла **Output**.

**3. Com1, Com2** – устойства последовательного **ввода-вывода** д/обмена данными между компами. Вместо **Com1** м б имя ‘**Aux**’.

**4. Nul** – 0-ое устройство. Для него при **выводе** нет никаких действий. При попытке чтения возникает ситуация конца файла.

**5. Crt** – устройство **текст**ового вв-вывода. Похоже на устройство **Con +** д/п функции управления экраном (например, установка цветов, указание места на экране для вывода и т.п.). **Crt****не** поддерживается операционной сист-ой ОС.

**6. ‘’** – использование пустой строки вместо имени **Name**: файловая переменная ***F*** связывается с **Con** (по аналогии с пунктом 1)). Например,

**Assign (F, ‘’);**

5.3. Файлы с типом (типизир Фа)

- состоят из **1-отипных** компонент, их тип указан при обьявлении.

**Пример 5.1.** Пример объявления файлов с типом.

**Type**

**Zap = Record**

**I: Integer;**

**R: Real**

**End;**

**Var**

**F1: File Of Real;**

**F2: File Of Char;**

**F3: File Of String[50];**

**F4: File Of Zap;**

**F5: File Of Integer;**

**F1 - F5** – файловые переменные.

Для работы с типизованными файлами - **процедуры и функции** вв-выв:

**1) Процедура Assign** – связывает файловую переменную с внешним файлом на диске. Описана в п. 5.2 выше.

**2) Процедура Rewrite (F)** – **создает** и открывает новый файл ***F***.

**Окно (- позиция доступа**) устанавливается на I позу файла.

Если файл уже открыт, то его надо закрыть и повторно открыть.

Если в файл ранее записали какие-то элементы, то они становятся **недоступны**. Результат выполнения процедуры **Rewrite(F)*:***

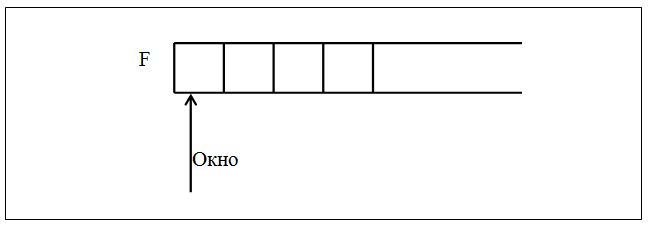


Рис 5.3 – Результат выполнения процедуры **Rewrite(F)**

Перед исп-ием **Rewrite** файл ***F*** должен быть связан процедурой **Assign** с внешним файлом**.**

Таким образом, процедура **Rewrite** создает новый внешний файл (с именем), которому процедура **Assign** присвоила переменную ***F*.**

**3) Процедура Write (F, V1 [, V2, …, Vn])** – записывает **очередной** элемент файла (= значению переменной **Vi**) в ту позицию файла, на которую указывает **окно (- позиция доступа**)**.** После записи переменной **Vi** это окно сдвигается на следующую позицию файла. Если окно дошло до конца файла, файл **расширяется.** Результат выполнения процедуры **Write (F, V1, V2):**

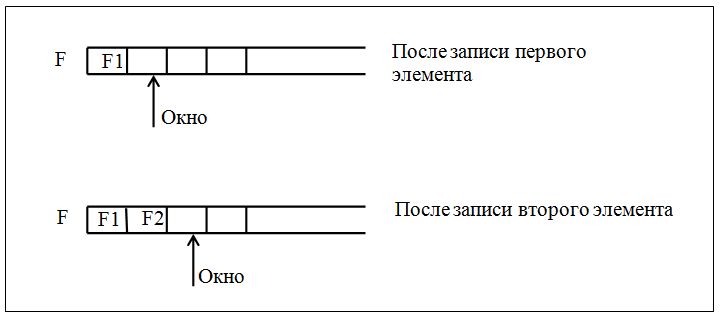


Рис 5.4 – Результат выполнения процедуры **Write (F, V1, V2)**

Перед выполнением процедуры **Write** (в программе) файл д б **открыт.**

**Пример 5.2.** Пример создания файла.

**var**

f1: **file of** char; // f1 – Фа переменная

x: char; //x - переменная-символ

i: integer;

**begin**

assign(f1,'newFile'); // newFile – создаем?

rewrite(f1); //открываем для записи

**for** i:= 1 **to** 100 **do**

**begin**

<операторы>

write(f1, x) // x - пер-ая, получ-ая при вып проги

**end**;

4)  **Процедура Reset (F)** – открывает существующий файл ***F***. может быть применена к файлу любое количество раз. (‘остаток’, повторн устан-ка)

открывается внешний файл с именем, присвоенным переменной ***F*** процедурой **Assign,** содержимое файла не изменяется**.** Если файла с таким именем нет - сообщение об ошибке.

Окно файла устанавливается на первую позицию файла. Если файл был уже открыт, то он закрывается и повторно открывается. Результат:

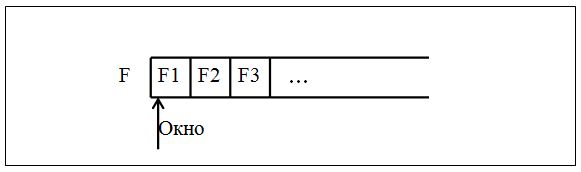


Рис 5.5 – Результат выполнения процедуры **Reset(F)**

5) **Процедура Read (F, [V1, V2, …, VN])** – читает текущий элемент файла ***F*** (- на который указывает окно) в переменную ***Vi*** и передвигает окно на следующую позицию файла. м б только для уже **открытого** файла.

**Файлы с типом** всегда допускают как **чтение,** так и **запись**, независимо были они открыты с помощью процедуры **Reset или Rewrite**.

Таким образом, доступна только та компонента файла, которая определена положением окна. Если нужна компонента, предшествующая окну, то при последовательном доступе надо установить окно на начало файла с помощью **Reset*,*** а затем, последовательно перебирая компоненты с помощью **Read,** дойти до нужной компоненты. Результат:

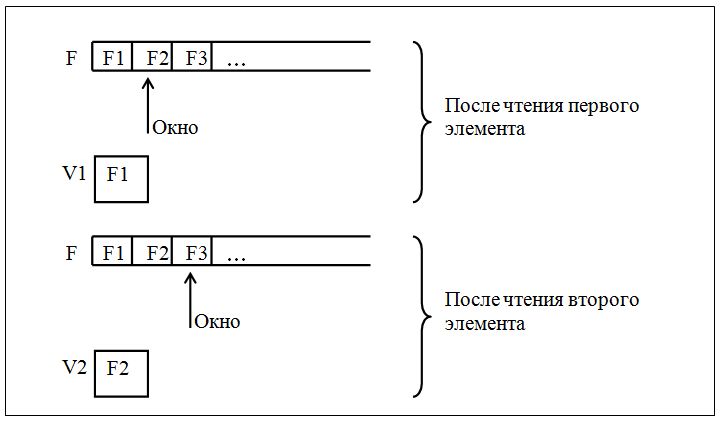


Рис 5.6 – Результат выполнения процедуры Read (F, V1, V2)

1. **Функция Eof(F)** (**E**nd **O**f **F**ile) – для определения факта выхода за пределы файла при чтении.

Функция **Eof**возвращает значение признака конца файла. Если пришел конец файла ***F*** (окно указывает на **маркер конца файла** – позицию, следующую за последней компонентой файла), или если файл пустой, то значение функции **Eof *=* True***,* иначе **False**.

Если **Eof = True**, то нельзя использовать процедуру **Read.**

Если в заголовке функции **Eof** опущено имя файла, то предполагается файл **Input*.*** Например,

**L := Eof;**

***L*** – переменная типа **Boolean***.*

обработка компонент файла в программе организуется с помощью оператора **цикла с предусловием**. В качестве предусловия - логическое выражение **Not Eof(F),** в теле цикла - обработка компонент файла ***F***.

**Пример 5.3.** Чтение из файла. Пусть имеется набор данных Old.

**Var**

**F2: File Of Char;**

**X: Char;**

**Begin**

**Assign (F2, ‘Old’);**

**Reset (F2);**

**While Not Eof (F2) Do**

**Begin**

**Read (F2, X);**

**<Операторы обработки переменной Х>**

**End**

**. . .**

1. **Процедура Seek (F, N)** – делает прямой доступ к элементам файла ***F***. Файл должен быть **открыт.**

***N*** – **№** элемента файла (нумеруются последовательно с **0**), положительное целое выражение типа **Longint*.***

**Seek** позиционирует файл на указанный № элемента: устанавливает окно файла на указанный элемент.

**Seek** **не** производит операций вв-вывода. Следующий за **Seek**вызов **Read */* Write** оперирует со следующими по порядку элементами файла.

**Seekне** обрабатывают текстовые файлы.

**Пример 5.4.** Запись в файл **F2** элементов файла **F1,** начиная с № 100 (фактически это **101**-ый элемент файла F1). Из файла в файл?

**Var**

C: Char;

F1, F2: **File Of** Char;

**Begin**

Assign (F1, ‘Old’);

Assign (F2, ‘New’);

**Reset** (F1);

Rewrite (F2);

**Seek** (F1, 100);

**While Not** Eof (F1) **Do**

**Begin**

Read (F1, C); // перекладыв с 1 файла в другой

Write (F2, C);

**End**;

…

1. **Функция Filepos (F)** – возвращает № текущей позиции (окна) файла ***F***. Если окно установлено на начало файла, то функция возвращает значение 0. Если окно установлено на конец файла (**Eof(F) = True**), то функция = размеру файла = результату функции **Filesize (F**)*).*

Тип результата функции **Filepos – Longint***.* **не** д/текстовых файлов. Файл д б **открыт.**

1. **Функция Filesize(F)** – возвращает текущий размер файла (число элементов в файле). Тип результата *–* **Longint*.***Для пустого файла возвращает значение **0**. Файл д б **открыт**. Не д/текстовых файлов.

**Пример 5.5.** Расширение файла ***F*** – «+» элементов в конец файла.

**. . .**

**Seek (F, Filesize (F));** //ставит окно Фа за посл эл-ом

**Write (F, C);**

**. . .**

процедура **Seek** помещает окно файла **за** последним элементом файла (функция **Filesize**возвращает к-во эл-ов, но т к они нумеруются с 0, то знач-ие **Filesize**совпадает с № следующ компоненты после последней).

1. **Процедура Close (F)** –закрытие открытого файла ***F***.

Для внешнего файла, связанного с файловой переменной ***F***, полностью выполняется его обновление (остатки данных из буфера вв-выв заносят в файл). Затем файл закрывается и м повторно использовать.

**Пример 5.6.** Определение размера файла + **Закрытие** файла.

**…**

**Assign (F, ‘Old’);**

**Reset (F);**

**Write** (‘Размер файла Old: ’**, Filesize (F));** {Выв на экран р-ра файла}

**Close (F);**

**…**

5.4. Текстовые файлы (#13 #10)

- последовательность символов. Но он **не** эквивалентен файлу типа

**T = File Of Char.**

Файл типа ***T*** – это **единая** последовательность символов.

Особенность текстовых файлов – символы разбивают на **строки.** Строки – различной длины (и пустые). В конце каждой строки помещают специальный управляющий символ – **маркер конца строки** - 2 управляющих символа: 1.возрат каретки (**#13** или **^M** – международное обозначение **CR**) и 2.перевод строки (**#10** или **^J** – междун-ое обозначение **LF**).

С этим маркером связана логическая функция **Eoln(E**nd **O**f **L**ine)

**Функция Eoln(F)** = **True*,*** если окно (- текущая позиция доступа) установлено на маркер конца строки, иначе - **False.**

**Текстовые файлы** – это переменные типа **Text**(- относится к предопределенным структурным типам), jописывают указанием имени типа:

**Var**

**X, Y: Text;**

Стандартные текстовые файлы **Input** и **Output** в программе **не** описывают. В начале выполнения любой программы (если не используется модуль **Crt)** автоматически выполняются процедуры:

**Assign (Input, ’’);**

**Assign (Output, ’’);**

**Reset (Input, ’’);**

**Rewrite (Output, ’’);**

**Процедуры и функции вв-выв** для работы с текстовыми файлами:

**1. Процедура Assign (F, Name)** – связывает файловую переменную с внешним файлом на диске. Описана в подразд. 5.2.

**2. Процедура AssignСrt (F)** – связывает текстовые файлы с **Crt**(с дисплеем). Аналогична проц-ре **Assign, н***о* 2го параметр **Name** в ней нет.

Определена в модуле **Crt*,*** который реализует более быстрый ввод-вывод, чем стандартный ввод-вывод и более функциональные возможности.

**Пример 5.7.** Вывод текстового файла ***F*** на принтер (или экран).

**Uses Crt;**

**Var**

**F*:* Text;**

**K*:* (P, C);** *{К –признак вывода, P –* принтер*, С – Crt}*

**Begin**

***…***

**If K = P Then**

**Assign (F, ’Prn’)**

**Else**

**Assigncrt (F);**

***…***

**3. Процедура Append (F)** – открывает существующий текстовый файл для добавления. Предварительно файл ***F*** должен быть связан с внешним файлом процедурой **Assign.** Определена только для текстовых файлов.

Если нет внешнего файла с указанным именем, то при выполнении **Append -** сообщение об ошибке ввода-вывода. Если файл ***F*** уже был открыт, его вначале надо закрыть.

В результате выполнения **Append** текущая позиция (окно) файла устанавливается в конец файла, на место маркера конца файла (**#26** – код ASCII 26 или Ctrl-Z).

После вызова процедуры **Append (F)** файл ***F*** становится доступным **только для записи**, а значение функции **Eof (F)** всегда **= True.**

**4. Процедура Rewrite (F)** – открытие нового файла F, описана выше для файлов с типом (см. подразд. 5.3).

**Отличие для текстовых файлов**: файл ***F*** открывается только для записи (из него читать нельзя).

После вызова **Rewrite (F)** значение функции **Eof (F)** всегда **= True.**

**5. Процедура Reset (F)** – открытие существующего файла F, описана выше для файлов с типом (см. подразд. 5.3). **Отличие** для текстовых файлов: файл открывается **только для чтения** (из него можно только читать, писать в него нельзя).

**6. Процедура Read ([F,] V1 [, V2, …, VN])** – считывает 1 или несколько значений из текстового файла ***F*** в 1 или несколько переменных **Vi.** описана выше для файлов с типом (см. подразд. 5.3). **Отличия** работы с текстовыми файлами (Read):

**а).** файл должен быть открыт для чтения (процедурой **Reset*);***

**б).** первый параметр м опустить (например, **Read(X, Y)*)***, тогда подразумевается стандартный входной текстовый файл **Input*;***

**в).** при выполнении процедуры **Read** - преобразование очередного элемента файла из символьного представления к типу переменной **Vi**.

**Vi** может иметь: 1.символьный, 2.целочисленные, 3.вещественный, 4.строковый тип, 5.тип массива символов или 6.тип диапазона данных типов.

**г).** Если переменная **V** – типа **Char**, то из файла ***F*** в ***V*** считывается очередной символ, включая символы-разделители «Конец файла» (значение Chr(26) **– #26**), и «Конец строки» (Chr(13) – **#13**, Chr(10) – **#10**). Следующая процедура **Read**начинается со следующего символа в файле.

**д).** Если переменная **V** имеет арифметический тип, то пропускаются пробелы, символы табуляции, маркеры конца строки до появления цифровой комбинации: т е считывание прекращается при достижении пробела, табуляции, маркера конца строки / файла, а с ледующее чтение – цифр. Если в цифровой комбинации встретились запрещенные символы (например, **буква** вместо цифры), то - сообщение об ошибке ввода-вывода.

**е).** Если переменная **V** – типа **String**, то в нее передается к-во символов, = длине переменной **V** при объявлении. Маркер в строку не заносится.

если 1) длина **V** < длины текущей строки файла: по следующей операции чтения в переменную ***V*** будет передана очередная последовательность символов из первой строки файла в соответствии с длиной переменной ***V***.

2) строка файла до маркера конца строки или маркера конца файла (если длина ***V >*** длины строки файла): следующая операция считывания начинается маркером конца строки, завершающим предыдущую строку. Но процедура **Read**не делает переход на следующую строку файла после чтения.

Т о, нельзя использовать последовательные вызовы процедуры **Read** для чтения последовательных строк: никогда нет перехода из первой строки во 2ю. 1й вызов **Read** считает 1й элемент, последующие вызовы будут возвращать строку длины = 0. Поэтому при чтении из текстового файла в переменные типа **String** надо использовать **Readln*.***

**7. Процедура Readln [([F] [,] [V1, V2, …, VN])]** – выполняет процедуру **Read и** переходит в начало следующей строки файла. Т о, после прочтения в переменные **Vi** данных из файла пропускаются все оставшиеся символы в текущей строке и маркер конца строки.

Если при вызове процедуры нет переменных **V** **(Readln(F)),** то - переход к следующей строке (если она есть) или к концу файла ***F***.

Если имя файла в процедуре **Readln**нет, то подразумевают стандартный входной текстовый файл **Input*.*** Например,

**Readln (X, Y, Z);**

**Readln;**

Процедура **Readln (F, V)** определена только для текстовых файлов.

**8. Процедура Write ([F,] E1 [, E2, …, EN])** – записывает 1 или несколько значений **Ei** в файл ***F***. В общем случае ***E*** м б выражением. Процедура описана для файлов с типом (см. подразд. 5.3). **Отличия** для текстовых файлов:

1. Файл д б открыт процедурой **Rewrite***/* **Append*.***
2. Если первый параметр (***F***) опущен, то подразумевают стандартный выходной текстовый файл **Output.** Например, **Write (A, B)*.***
3. При выполнении процедуры **Write** идет преобразование выводимого значения из типа выражения ***E*** к символьному типу. Возможны ***типы Ei:*** символьный, арифметические (целочисленные и вещественные), строковый, массив смволов, логический и их диапазоны (char, int+real, string, array, boolean+их диапазон).
4. Для данных типа **Char *и* String** выводится само их значение.
5. Если **Ei** - арифметическое значение, то перед выводом в текстовый файл оно предварительно преобразуется из внутреннего значения в 10чную систему счисления, а затем представляется в коде ASCII (по 1 байту на десятичную цифру).
6. Если **Ei –** тип **Boolean**, то выводится **True (False)***.*
7. Если не указан формат **выв**ода, то под вывод всех типов **Ei** отводится минимум символов: д/вещественных типов – 24 позиции (веществ-ое число будет выведено в виде мантиссы и порядка).
8. Значения **Ei**могут сопровождаться указанием формата вывода:

**E: L1[: L2]**

что позволяет управлять расположением данных по позициям строки. Подробно – в [подразделе. 2.2](file:///D:\БГУИР\ОАиПc\2%20ТЕОРИЯ%20ОАиП%20часть2%20ПОИТ2.doc#_2.2._Процедуры_вывода_в стандартный).

1. **Процедура Writeln ([F,][E1,E2, …,En])** – выполняет процедуру **Write*,*** и затем записывает **маркер** конца строки в файл ***F***.

Если нет списка выражений **Ei*,*** то записывают лишь маркер конца строки.

Если опущено имя файла **F**, то подразумевают файл Output:

**Writeln (F);**

**Writeln (A, B, C);**

**Writeln;**

Процедура определена только для текстовых файлов.

1. **Процедура Close(F)** – закрывает открытый файл ***F***. Рассмотрена ранее (5.3). Особенностей для текстовых файлов нет.

Пример 5.8. Использов процедур при работе с текстовым файлом

**Var**

F: Text;

a, b, c: char;

**Begin**

Assign (F, 'Data.txt');

Rewrite(F); {Открытие файла F только д/записи}

…

Write(F, A); {Запись в Фа F знач-я перем-ой А после вычисл-ий}

…

Reset(F); {Открытие фа F только д/чтен. Перед этим Фа не обязат

д б **за**крыт: выполн-ся автом-ки проц-ой **Reset**}

Read(F, B); {Чтение из Фа F очередного знач в переменную В}

…

Close (F); {Обязат явное закрытие Фа F перед использ-ем проц-ры

Append. И связь с внешн Фа сохраняется, => повторное

использование Assign не нужно}

…

Append (F); {Открытие файла F д/**дополнения**}

…

Write (F, C); {Запись в конец F значения перем-ой С из вычисл-ий}

…

Close (F); {Закрытие файла F в конце работы с ним}

**End**.

Кратко:

**var**

f: text;

a, b, c: char;

**begin**

assign(f, 'data.txt');

rewrite(f);

write(f, a);

reset(f);

read(f, b);

close(f);

append(f);

write(f, c);

close(f);

**end**.

1. **Проц-ра SetTextBuf(F, Buf [, Size])**– опр-ет **буфер** д/текст-го фа.

Процедуру вызывают после **Assign** *и* до других процедур вв-выв (т е м/у проц-ми). ***F*** – имя текст фа, **Buf**– любая переменная (за формальный параметр используют параметр-переменную **б/типа (ППТ)**), **Size** – **необязат**ельное выражение типа **Word*.***

Обмен информацией между программой и внешним набором данных (Фа) - через **буфер вв-вывода** – участок оперативной памяти, его размер по умолчанию – 128 байт. Каждому открытому файлу назначается свой буфер. Если в проге много операций вв-вывода, то используют буфер размера >128, чтоб ум-ть время обращения к внешним наборам данных.

Процедуры **Write***и***Writeln** записывают очередные элементы файла последовательно в буфер (укладывают штабелями). Когда буфер будет полностью заполнен, произойдет физическая запись содержимого буфера во внешний файл. Потом буфер **освобожд**ается д/следующей порции информации.

Аналогично при чтении: из внешнего файла одновременно считывается количество элементов, помещающееся в буфер. Процедуры **Read***и***Readln** читают элементы последовательно (циплят считают) из буфера.

**Буфер** вв-вывода существенно повышает скорость обмена инфой с внешними файлами (например, за счет ум-ия времени перемещения магнитных головок в дисководах).

Операции обмена данными через буфер ввода-вывода делает специальный обработчик файлов (для каждого файла - свой, он назначается при открытии файла).

Процедура **SetTextBuf**назначает текстовому файлу ***F*** свой буфер ввода-вывода, определяемый параметром **Buf*.***Размер буфера в байтах определяется параметром **Size.**

Если параметра **Size**нет, то по умолчанию размер буфера *=* **Sizeof (Buf)*,*** т е вся область памяти, занимаемая параметром **Buf,** используется как буфер. Если параметр **Size** есть, он д б < = переменной **Buf*.***

Процедуру **SetTextBuf**нельзя применять к открытому файлу (она применяется м/у **Assign***и* **Reset*,* Rewrite /****Append**). Определена **только для текстовых** файлов.

**Пример 5.9.** Назначение **буфера** вв-вывода текстовому файлу ***F***. Передача данных из фа ***F*** в стандартный текстовый файл **Output.**

**Var**

**F: Text;**

**C: Char;**

**Buf: Array [1..10240] Of Char;** {**Буфер** размером **10 килобайт,**1=1024}

**Begin**

**Assign (F,** 'A:\bin\Metod.txt'**);**

**SetTextBuf (F, Buf);** {Назначение **буфера** вв-выв Buf текстовому Фа F принудительно. Р-р буфера = р-ру переменной Buf – 10240 байт}

**Reset (F);**

**While Not Eof (F) Do** {Цикл чтения из файла F и записи в фай Output}

**Begin**

**Read (F, C);**

**Writeln (C);** // печать на экран?

**End; …**

**var**

f: text;

c: char;

buf: **array**[1..10240] **of** char;

**begin**

assign(f, 'metod.txt');

settextbuf(f, buf);

reset(f);

**while not** Eof(f) **do**

**begin**

read(f, c);

writeln(c); // печать на экран?

**end**;

**end**.

1. **Процедура Flush (F):** используется **редко** в прикладных программах – для очень важных результатов **– очищает буфер** текстового файла, открытого для вывода процедурой **Rewrite /****Append***.* По этой процедуре информация из буфера записывается во внешний файл (независимо от степени его заполнения (полностью / частично)). Определена **только для текстовых** файлов.
2. **Функция Eof (F)** – конец файла. (см. подразд. 5.3).
3. **Функция Eoln (F)** – конец строки. См начало подраздела.
4. **Функция Seekeof (F)** – устанавливает файл ***F*** в состояние **“конец Фа”.** Аналогична функции **Eof*,*** **но** пропускает все пробелы, табуляцию и маркеры конца строки при считывании **числ**овых значений из текстового файла. Определена только д/текстовых файлов. Файл д б **открыт.**
5. **Функция Seekeoln (F)** – устанавливает файл ***F*** в состояние “**конец строки**”. Аналогична функции **Eoln*,*** но пропускает все пробелы и табуляцию. Используется д/считывания существующего текстового файла числовых данных. только д/текстовых файлов. Файл д б **открыт.**

5.5. Сравнительная характер-ка представления инфы в файлах с типом и текстовых файлах

1у и ту же инфу м записать как в текстовый файл, так и в файл с типом. **Внутренняя структура** представления инфы в этих файлах:

**I. Представление числовой информации (int, real)**

**а) В текстовом (ASCII)**

Пусть в текстовом файле записана последовательность чисел

**8192, 2048, …**

Внутреннее представление этой последоват-ти в текстовом файле (рис 5.7): числа представлены в коде обмена информацией **ASCII** (в распакованном формате в коде 8421). Под каждую цифру 10чного кода числа отводят 1 байт. Числа отделяют др от др min 1 пробелом.

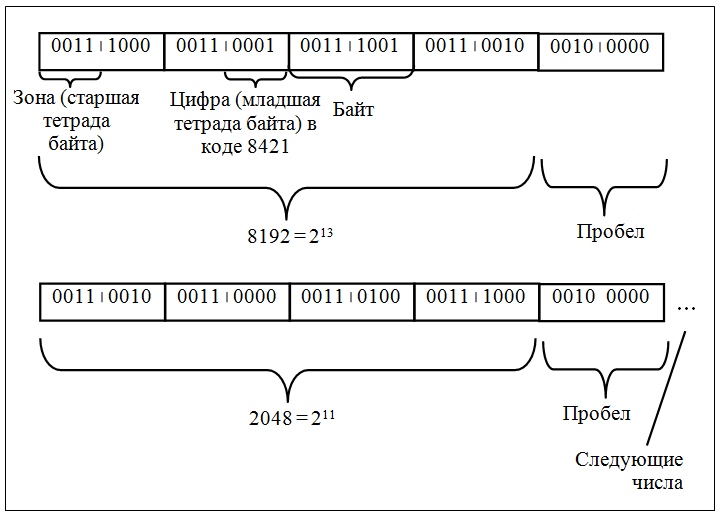


Рис 5.7 – Представление числовой информации в текстовом файле

Т о, для хранения последовательности из 2 первых чисел 8192, 2048 надо не менее **10 байтов**.

**б) В типизованном**

Та же последовательность чисел: **8192, 2048, …**

в файле типа **File Of Integer** имеет внутреннее представление (рис 5.8):

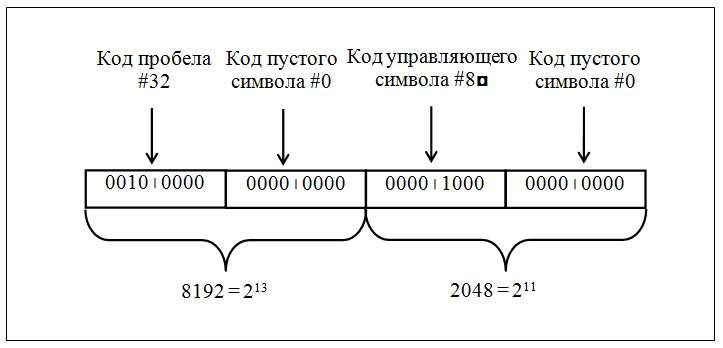


Рис 5.8 – Представление числовой инфы в файле типа **File Of Integer**

здесь числа - в двоичном коде в формате **Integer*.*** Под каждое число отводится **2 байта**. Разделителей между числами **нет**. Т е, для хранения послед-ти из 2х чисел 8192, 2048 надо **4** байта.

При попытке вывода данных байтов файла на экран выведется **пробел** (не изображается), **пустой** символ (не изображается), символ кода ASCII с номером 8 (изображается в виде ◘), **пустой** символ.

Т о, **при работе с числовой инфой, если ее не надо выводить на экран / печать (**это делают лишь **текстовые файлы), эффективнее файлы с типом*:***

1. работа с ними - быстрее за счет факторов:

а) нет преобразования информации, она в файле представлена так же, как в памяти;

б) при работе с файлами с типом м б режим прямого доступа;

в) меньше операций физичес-го вв-выв за счет меньшего р-ра файла;

1. они занимают меньше места (в текстовом файле каждая цифра числа занимает 1 байт, разделители – не менее 1 пробела между числами, маркер конца строки –2 управляющих символа **#13#10**; в файле типа **File Of Integer**все число занимает 2 байта, разделители м/у числами не нужны)

**II. Представление текстовой информации.**

**а) В текстовом**

Пусть в текстовом файле записан текст «ВАШ ОТВЕТ НЕВЕРЕН» и разбит на строки (рис 5.9):

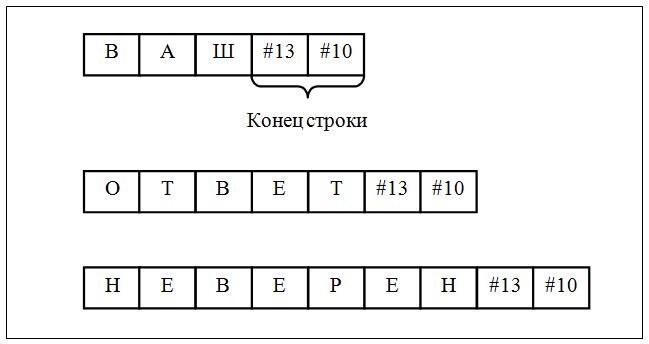


Рис 5.9 – Представление текстовой информации в текстовом файле

Каждая строка - 1 слово, имеет текущую длину и заканчивается маркером конца строки. Каждый символ текста - в коде ASCII и занимает 1 байт. Т о, д/представления информации в текстовом файле надо **21 байт**.

**б) В файле of string [7] – тип строка (типизированный?).**

Этот текст в файле типа **File Of String[7]** имеет внутреннее предст-ие:

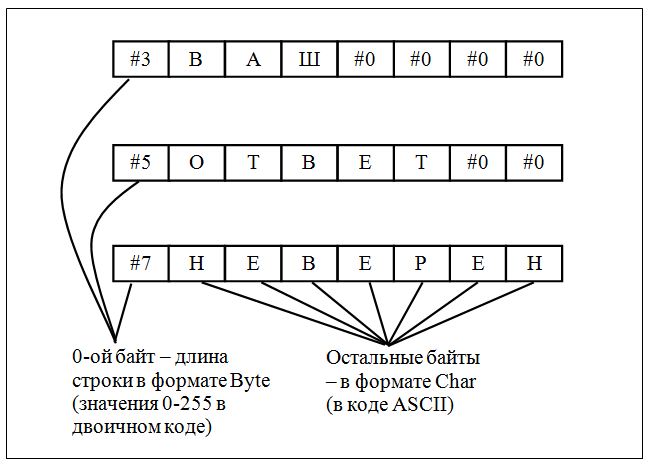


Рис 5.10 – Представление текстовой инфы в файле типа File Of String [7]

В файле с **типом** строки имеют постоянную (max) длину. Пустой символ (символ #0 в кодовой таблице) дополняет строки текущей длины до max длины. =>, для представления этой инфы в файле **типа** **File Of String [7]** надо **24 байта.**

**Сравнение** файлов типа **Text** и **File Of String** показывает:

а) **меньше места** занимает **текстовый** файл (в нем используют строки текущей длины, а не max);

б) большe **скорость** - у **File Of String:** для **типизованных** файлов возможность работы в режиме прямого доступа, работы одновременно в режиме записи-чтения, не надо отслеживать управляющие символы **#13#10**.

Т о, для **хранения** текстовой инфы выбирать файл типа **Text***или***File Of String**в каждом конкретном случае по особенностям задачи.

5.6. Файлы без типа (Фб/Т) (не типизир-ый Фа)

- состоит из компонент **одинакового размера**, их cтруктура (компонент) неизвестна / не имеет значения.

Такое представление стирает все различия между файлами любых типов. Любой файл можно открыть и начать работу с ним, как с файлом без типа, независимо как он был подготовлен (текстовый фа/ фа с типом). Объявление файла без типа: ***Var***

**F: File;**

Назначение – максимально повысить скорость обмена информацией с внешними наборами данных за счет **факторов**:

1. В Фб/Т **нет** преобразования типа компонент;
2. Нет поиска управляющих символов (типа конец строки);
3. В Фб/Т, как и в файлах с типом, м организовать метод прямого доступа: одновременное использование операций чтения и записи независимо какой процедурой (**Reset / Rewrite**) они были открыты.
4. Обмен инфой с внешними наборами данных м б большими блоками – **основной фактор** повышения скорости обмена.

Для Фб/Т определены те же процедуры и функции как для файлов с типом (кроме **Read***и***Write)**:процедуры**Assign, Rewrite, Reset, Seek, Close,** функции **Eof, Filesize, Filepos.** Процедуры **Reset, Rewrite** - имеют особен-ти: при их вызове миспользовать **2** параметра, который имеет вид:

**Rewrite** (F [,Recsize]);

**Reset** (F [,Recsize]);

**Recsize**– это **необязат**ельное выражение типа **Word**, определяющее размер записи (в байтах), используемый при передаче данных. Например,

**Rewrite (F, 1);**

**Reset (F, 1);**

– 2й параметр определяет длину записи в 1 байт.

Если параметра **Recsize**нет, то длина записи = **128** байт – минимально возможный объем инфы для обмена.

1. **Процедура Blockread**

Вместо процедур **Read***и***Write** в Фб/Т – процедуры **Blockread** и **Blockwrite*.***

Процедура **Blockread** имеет формат вызова:

**Blockread (F, Buf, Count [, Result])**; (4 шт)

***F*** – имя файловой переменной б/Т,

**Buf** – любая переменная (как формальный параметр – параметр-переменная б/типа ППТ),

**Count**– выражение типа **Word*,*** определяет к-во считываемых записей.

**Blockread** считывает блок информации длиной в **Count** или меньше записей в область памяти, занимаемую переменной **Buf** (начиная с ее первого байта). Действительное число считанных **полных** записей (<= значения **Count*)*** заносится в параметр **Result**(если он есть).

If **Result *<* Count** - конец файла дошел до полного окончания передачи. Тогда, if параметра **Result** нет - сообщение об ошибке вв-выв (поэтому лучше **Result**использовать).

В результате выполнения процедуры **Blockread** “окно” файла (текущая позиция файла) передвинется на число записей, = значению **Result*.***

Объем блока информации, считываемый **Blockread** в переменную **Buf =** **Result \* Recsize** байт. Здесь **Recsize** – размер записи, определенный при открытии файла. Размер блока инфы <= 64К байт ( - размер сегмента данных), иначе - ошибка вв-выв.

1. **Процедура Blockwrite** имеет формат вызова:

**Blockwrite (F, Buf, Count [, Result]**)

Назначение параметров – как в предыдущей процедуре **Blockread**. Процедура записывает 1 или несколько записей из области памяти, занимаемой переменной **Buf**(начиная с ее первого байта), в файл ***F***.

Параметр **Result** возвращает количество **полных** записанных записей. Если **Result < Count** - диск переполнился до завершения пересылки данных. => если **Result** нет – ошибка вв-выв.

В остальном описание **Blockwrite =Blockread***.*

**Пример 5.10.** Программа быстрого копирования из файла **F1** в **F2**

**Var**

F1, F2: **File**;

Buf: **Array** [1..2048] **Of** Char; //20 кБайт

NumR, NumW: Word;

**Begin**

Assign(F1, 'D1');

Assign(F2, 'D2');

Reset(F1, 1); {Р-р записи при передаче данных =1 байту}

Rewrite (F2, 1); // открыв д/записи

**Repeat**

Blockread (F1, Buf, **Sizeof** (Buf), NumR);

Blockwrite (F2, Buf, NumR, NumW);

**Until** (NumR = 0) **Or** (NumrR <> Num**W**); {Закончился файл F1 или

переполнился диск при создании файла F2}

Close (F1);

Close (F2);

**End**.

**5.7. Проверка операций ввода-вывода**

По умолчанию при выполнении операций вв-выв идет их стандартная проверка системными средствами. Для управления проверкой вв-выв - опция **компилятора** **{*$I*}**, по умолчанию включена **({$I+}).** При этом, если есть ошибка вв-выв – прерывается выполнение программы и сообщение о типе **ошибки.**

Если прогу прервать нельзя по ошибке вв-выв, т е **мы сами** хотим обрабатывать данные ошибки, надо **отключить** стандартную проверку в тексте программы **перед операциями вв-выв** опцией **{$I-}**, а после этих операций **снова включить** ее опцией **{$I+}.**

+д/ контроля операций вв-выв в состоянии **{*$I*-} –** функция **IOResult**(без параметров): возвращает целочисленное значение типа **Word –** состояние последней **выполненной** операции вв-выв. Если нет ошибки вв-выв, то функция возвращает **0,** иначе – **код ошибки**.

Если в состояние **{*$I*-}** и при некоторой операции вв-выв произошла ошибка, то последующие операции вв-выв б игнорированы, (используемые до вызова функции **IOResult)***.*

Т о, не надо использовать 1 вызов функции **IOResult** на несколько операций вв-вывода, т к неясно, в какой из них была ошибка.

Вызов функции **IOResult**очищает свой внутренний флаг ошибки, (возвращаемое значение устанавливается в **0**). Поэтому нельзя **повторно** считать 1о и то же значение **IOResult*.***

**Пример 5.11.** Использ-ие ф-ции **IOResult** д/проверки наличия внешнего Фа.

**Var**

F: **File Of** Char;

**Begin**

Assign (F, 'PRIM');

{$I-}

Reset(F);

{$I+}

**If** IOResult = 0 **Then**

Writeln ('Размер файла: ', Filesize (F), ' байт')

**Else** Writeln ('Файл не обнаружен');

**End**.

## Раздел 6. Ссылочный тип (тип указатель)

6.1. Общие сведения

В предыдущих разделах рассматривались **автоматические переменные** – переменные, порождаемые непосредственно перед выполнением программы (подпрограммы) и существующие в течение всего времени ее выполнения. Объем машинной памяти не меняется по ходу выполнения программы для их размещения. Они порождаются с помощью соответствующих им описаний в разделе объявлений. Например, ***Var X: Integer;***

***X*** – автоматическая переменная.

Место в памяти для хранения их значений выделяется на этапе трансляции. Обращение к ним в программе - указание их имен.

**Динамические переменные** – это переменные, которые порождаются и уничтожаются в процессе выполнения программы, размер значений которых (область отводимой памяти) определяется при выполнении программы и может изменяться.

Для работы с динамическими переменными есть **ссылочный тип (тип указатель*)***. Значение этого типа - ссылка на какой-либо элемент программы. По данной ссылке идет прямой доступ к (телу) этому элементу. Как ссылку используют **адрес** соответствующего элемента в памяти машины.

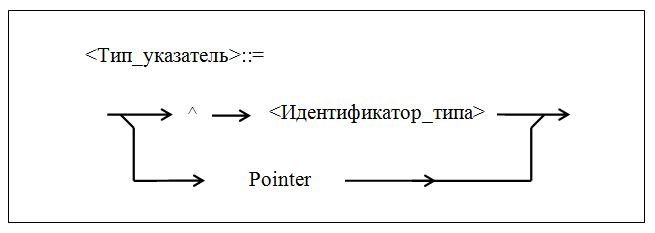


Рис 6.1 – Синтаксическая диаграмма задания типа указатель

1й вид Ука – **<Идентификатор\_типа>** - это **тип** динамически размещаемой переменной. В качестве **типа** – **имя** стандартного или описанного отдельно типа.

2 **вида указателей** ***–*** *1.*указатель на **элемент некоторого типа** (верхнее ребро синтаксической диаграммы) и 2.указатель, **не связанный с типом *(*Pointer*).***

**Пример 6.1.** Использ-ие указателей при объявлении программных эл-ов.

**Type**

***Mas = Array [1*..10] Of Integer;**

**Admas = ^Mas;**  {**Admas** – тип указателя на массив типа Mas}

**P1 = ^Integer; {P1 –** тип указателя на тип **Integer}**

**Var**

**P: ^Integer;** {в Ука P б храниться адрес динамич-ой перем целог типа}

**Q: ^Char;** {в указат Q б храниться адрес динамич перем типа Char}

**Pp: P1;** {в указателе Pр – **адрес** динамич перем-ой целого типа}

**Pt: Pointer;** {Ука Pt **м** хранить адрес динам перемен произвольн типа}

**Adrm: Admas;** {в указат Adrm – адрес **динам массива** типа Mas}

**Указатели** – автоматические переменные ссылочного типа.

Динамические переменные **не** объявляют в программе. Поэтому (!) средство доступа к ним – **указатели.**

Связь указателя ***Р*** с динамической переменной – на рис 6.2:

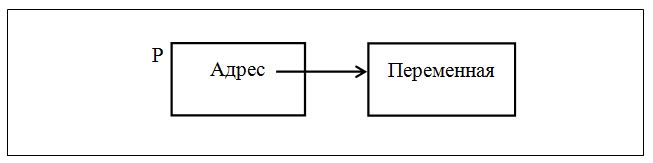


Рис 6.2 - Связь указателя *Р* с динамической переменной

Указатель занимает ***4* байта памяти*.***

Иногда надо в качестве значения указателя принять **пустую ссылку** – ссылку, которая **не** связывает с данным указателем никакой динамической переменной. Для этого служит **предопределенная** константа **Nil-** совместимая по типу с любым типом указателя.

Например, для указателей в примере 6.1, можно записать:

**Pt := Nil;**

**P := Nil;**

**Q := Nil;**

**Adrm := Nil;**

Над значениями ссылочного типа (**ЗСТ**) **нет** операций, которые давали бы результат этого же типа.

Над **ЗСТ** определены только **присваивание** (:=) и **2** операции **сравнения**: **=** и **<>**.

2 **ЗСТ** =ы, если они оба **Nil /** указывают на 1у и ту же переменную. В остальных случаях - неравенство.

В правой и левой частях от оператора **:=** д б выражения 1го типа: справа - значение ссылочной **перем**енной(ЗСП), слева - тип переменной.

**Ссылочным выражением** м б:

1. **пустая** ссылка **Nil*;***
2. ссылочная **перем**енная;
3. ссылочная **функция** – её значение – ссылка.

Например, встроенная ф-ция **Addr(X)** – возвращает адрес эл-та ***X***. ***X*** – любая переменная, процедура, функция. Результат функции **Addr(X)** совместим (по присваиванию) со всеми типами “указатель”. Поэтому можно написать такие операторы присваивания (для переменных типа указатель в прим. 6.1):

**Pt := Addr (А);** {А может иметь любой тип}

**P := Addr (В);** {В – только тип **Integer**}

**Pt: Pointer;**

**P: ^Integer;** {в Ука P б храниться адрес динамич-ой перем целог типа}

Объявление ссылочной переменной в разделе **Var** **не** порождает самой динамич переменной: по этому объявлению транслятор отводит место в памяти для размещения ссылки (**адреса** динамич переменной).

6.2. Методы работы с динамич переменными (3)

с помощью **процедур:**

1. **New***и***Dispose***;*
2. **Getmem** *и***Freemem;**
3. **Mark***и***Release**.

6.2.1. Процедуры New и Dispose

**стандарт**ная процедура **New (P)** – для порождения динамической переменной. ***P*** – ссылочная переменная любого типа.

Данная процедура размещает динамическую переменную типа указателя: выделяют память для хранения наиболее длинного значения типа указателя(ЗТУка). **Адрес начала** этого места памяти присваивают заданной ссылочной переменной ***P***.

**Пример 6.2.** Использование процедуры **New.**

**Var**

**P*: ^Integer;*** {Р – **указатель** на динамич перем-ую типа Integer}

**Begin**

**New (P); {-** порождает **динамич переменную** типа ***Integer***,

адрес этой переменной присваивается **указателю *P}***

**...**

Процедура **New** играет ту же роль для динамич переменной, что раздел описаний **Var** для автоматической переменной. Никакого значения процедура **New** динамической переменной **не** присваивает.

Для обращения к **динам**ич переменной – **переменная с указателем**, её синтаксис:

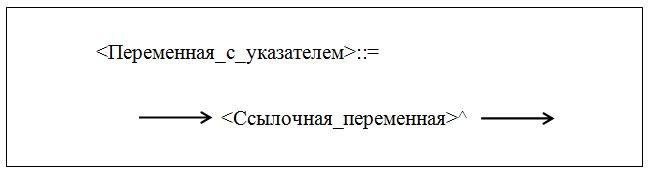


Рис 6.3 – Синтаксическая диаграмма переменной с указателем

Символ **^** называется **знаком карата**.

В пр. 6.2 обращение к **динамич** перемен (связанной с указателем ***Р)*** идет с пом. **переменной с указателем**:

**P^ := 15;** {**Р^ - переменная с указателем**; динам переменой типа Integer (связан с указател P) присвоено значение 15}

Над **переменными с указателями** определены **любые** **операц**ии, что и д/типа **динамической переменной**.

**Пример 6.3.** есть объявы - прим 6.1, 2. Операции над динамич переменными.

**Type**

Mas = **Array** [1..10] **Of** Integer;

Admas = ^Mas; {**– тип** указателя на массив **типа** Mas}

P1 = ^Integer; {– **тип Ука** на **тип** Integer, м указ-ть только на Инты}

**Var**

P: ^Integer; {в P б хран адрес(дом, ул) динам перем целого типа}

Q: ^Char; {в указат Q б храниться адрес динамич перем типа Char}

Pp: P1; {в указателе Pр – адрес динамич перем-ой целого типа}

Pt: Pointer; {Ука Pt м хранить адрес динам пер произв типа}

Adrm: Admas; {в указат Adrm – адрес динам массива типа Mas}

**X: Integer;**

**Begin**

**New (P);**

**P^ := 15;**

**...**

**X := X + P^;**

**P^ := P^ Mod 5;**

**Adrm^[P^ + 2] := 14;**

**Пример 6.4.** Организация массивов указателей. есть объявы пр 6.1:

**Type**

Mas = **Array** [1..10] **Of** Integer; Admas = ^Mas;

P1 = ^Integer;

**Var ...**

**А: Array[1..20] Of P1;** {–мас-армия ссылок на **динамич** перем типа Inte}

**I:** Integer;

**Begin**

***...***

**For I:= 1 To 20 Do**

**Begin**

**New (A[I]);** {Порождение динамич перем, связ-ой с Ука A[I]}

**A[I]^ := I;** {Обращение к динамической переменной}

**End**

***...***

Здесь каждая из **ссылок** **A[I]** обеспечивает **доступ** к соответствующей переменной с указателем **A[I]^.**

Идентификатор типа в определении ссылочного типа м объявить и **до его использования, и после** (в отличие от других типов идентификаторов).

Например, в **примере 6.1** идентификатор **Mas**был объявлен до его использования в типе указателя **Admas*.*** М и обратное объявление:

**Type**

**Admas = ^Mas;**

**Mas = Array [1..10] Of Integer;**

**Пример 6.5.** Работа с указателями + с динамич переменными.

***Var***

**X, Y: ^Integer;** {X, Y – указатели на динамич перем-ые типа Integer}

**Begin**

**New (X);** {Порождение динам перем-ой типа Integer, её адрес - в Х}

**New (Y);**{ Порождение динамич перем типа Integer, её адрес - в Y}

**X^ := 5;** { В динам перем, связ-ую с указат X, всунули знач 5}

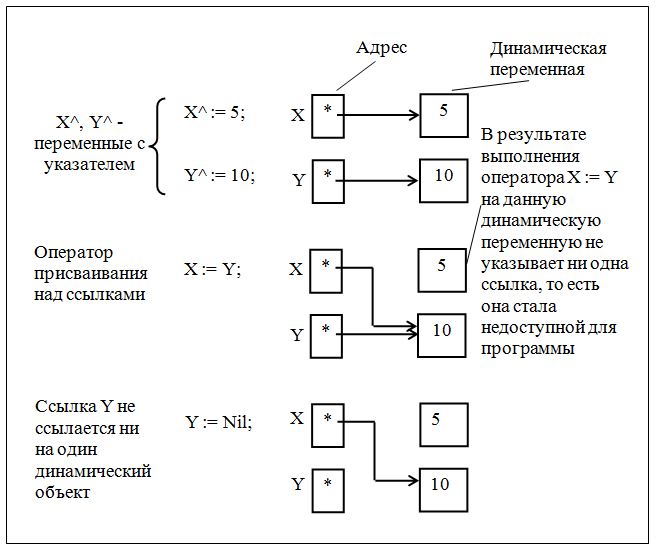
**Y^ := 10;** {В динам перем, связ-ую с указат Y, занесено знач 10}

**X := Y;** {Указателю Х присвоено значение указателя Y}

**Y := Nil;** {Указателю Y := значение **пустой ссылки**}

**Dispose (P)**;

Схематично действия операторов программы:

Рис 6.4 – Действия операторов программы **по примеру 6.5**

Если вместо оператора присваивания **X := Y** (рис 6.4) будет оператор **X^:=Y^,** то его результат б выглядеть так:

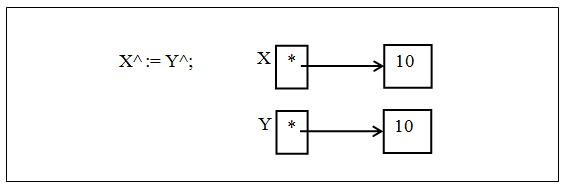


Рис 6.5 – Результат выполнения оператора **X^ := Y^**

Переменная типа указатель м б параметром функции **Ord.**

**(ф-ция** **ord** в **Pascal** возвращает число, соответствующее символу**)**. Результат этой функции - целое значение, = адресу динамич переменной, на которую ссылается данный указатель. Обратное преобразование (Integer 🡪 указатель не допускается). В примере 6.5 после выполнения оператора

**X := Y**

динамическая переменная со значением **5** стала недоступна программе, но продолжает занимать отведенное ей место в памяти и его нельзя использовать для хранения других динамич переменных.

Для **уничтожения** динамич переменных (**освобождения** памяти) – стандартная процедура **Dispose (P)**: динамич переменная удаляется, (связанная с указателем ***P)*** и занимаемая ею память свободна, а значение ***P*** **неопределено*****=* Nil***.*

Проц-ра **Dispose**уничтожает только динам перемен, но не **указатель** на нее.

**Пример 6.6.** Освобождение памяти, занятой динам перемен X^ (модифицированный фрагмент программы из прим 6.5).

…..

New (X); New (Y);

**X**^ := 5; Y^ := 10;

**Dispose *(X);*** {Освобож-ие обл памяти, занятая динамич пер-ой X^}

X := Y;

Y := Nil;

...

**Пример 6.7.** В задан тексте (предл?) найти слово **максим**-ой длины. Раздел-ся 1м пробелом. Признак конца текста – точка. Макс длина слова – 50 симв.

**type**

Mas = **array**[1..50] **of** char;

Ykaz = ^Mas;{Тип указателя на массив}

**var**

Vspom, Rez, Tek: Ykaz; {Переменная типа указателя на масив}

Max, i: integer;

Byk: char;

**begin**

Max := 0;

i := 0;

**New**(Tek); {Порождение динам перемен д/хранен

текущ слова текста}

{1}**New**(Rez); {Порождение динам перемен д/хранен максим-го

из уже прочитанных слов}

**repeat**

read(Byk);

**if** (Byk <> ' ') **and** (Byk <> '.')

**then**

**begin**

i := i + 1; {счетчик длины текущ слова}

Tek^[i] := Byk

**end**

**else**

**if** i > Max **then**

**begin**

Max := i;

{2} Vspom := Tek;

{3} Tek := Vspom;

{4} i := 0;

**end**

**until** Byk = '.';

Dispose(Vspom);

Dispose(Tek);

writeln;

**for** i := 1 **to** Max **do**

write(Rez^[i]);

dispose(Rez)

**end**.

Здесь выполнение оператора {**3}** не требует пересылки компонент массива типа **Mas** из динам-й перем-й **Tek^** в динам-ую перем-ую **Rez^*.*** В указатель **Rez** идет адрес динам-го массива **Tek^.** Таким образом, указатель **Rez** “перебрасывается” на другой массив. В указатель **Tek** с помощью операторов {2}, {4} заносится **адрес предыдущего** динам-го массива максимальной длины, массивы как бы меняются местами, но физически этого нет.

На следующем цикле чтения в массив **Tek^** будет занесено очередное текущее **слово**. Аж слово?

Время выполнения этой программы сильно уменьшается по сравнению с использующей автоматические переменные: нет физической пересылки массивов. Пояснения к этой проге схематически:

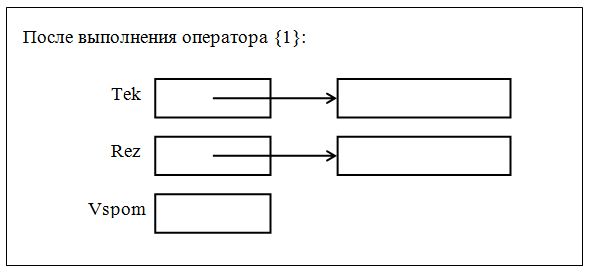


Рис 6.6 – Результат выполнения оператора **New (Rez)**

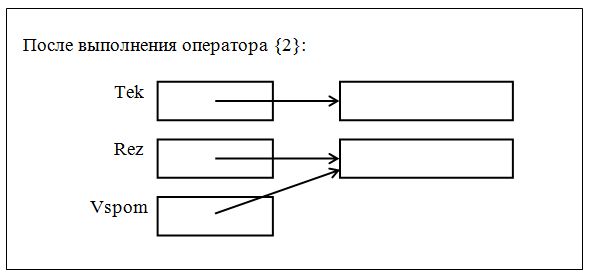


Рис 6.7 – Результат выполнения оператора **Vspom := Rez**

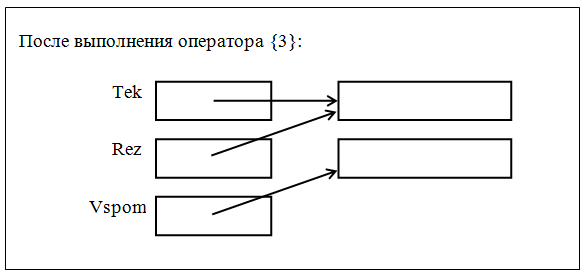


Рис 6.8 – Результат выполнения оператора **Rez := Tek**

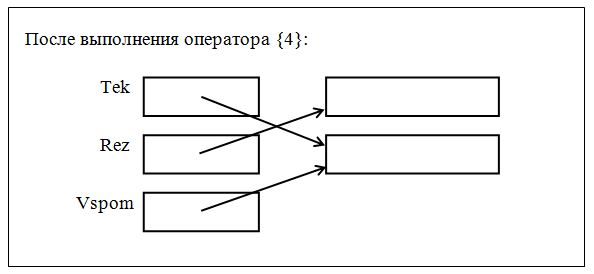


Рис 6.9 – Результат выполнения оператора **Tek := Vspom**

6.2.2. Процедуры Getmem и Freemem(-ory)

– **2й** метод работы с **динамич переменными**. Вызов процедуры **Getmem**:

**Getmem (P, Size);** (Get – полчить)

***P*** – **указатель^** любого типа, **Size***–* выражение типа**Word*.***

Процедура **Getmem** **выделяет** из **Heap-области** блок памяти заданного размера **Size** (порождает динамическую переменную размера **Size)** и адрес этого блока присваивает ссылочной переменной **P**. Максимальный размер блока – 65521байт (**64К байта** – **$F** (**смещение**)).

**Heap*-*область –** 1. **обл динамической памяти**, 2.обл **кучи**, 3. хип-обл.

Процедура **Freemem(P, Size)** – д/**освобождения** блока памяти, занимаемого динамической переменной. **Freemem**у**ничтожает** динам перемен **P^,** занимаемая ею память свободна, значение ***P*** – **неопред**елено.

***P*** – указатель произвольного типа, которому заранее было присвоено **значение** процедурой **Getmem** или оператором присваивания;

значение **Size** (должно точно соответствовать размеру переменной, размещенной до этого) д б таким же как было выделено проц-ой **Getmem.**

Т о, действие процедур **Getmem/Freemem** == **New/Dispose** == **порождают/уничтожают** динамич переменную.

**Пример 6.8.** Эквивалентные фрагменты программ, использующие процедуры **Getmem/Freemem и New/Dispose**.

**Type**

**T = <Тип>; //integer, char, …**

**Var**

**P: ^T;**

**…**

**Begin**

****

**Отличие:** процед-ры **Getmem/Freemem** дают большую свободу действий, чем **New/Dispose**(б/привязки к конкретному типу переменных. А пишут не **Size**, а **Sizeof** в параметрах пр-ра 6.8.

**Пример 6.9.** Использование процедур **Getmem/Freemem**. Разная интерпретация выделенной области памяти.

**Type**

Zap = **Record**

X, Y: Integer;

**End**;

St = String [20];

**Var**

P: Pointer;

**Begin**

Getmem (P, 100); {Выделен блок памяти = 100 байт, адрес эт блока

присвоен ссылочной перем-ой Р}

Zap(P^).X := 124; {Интерпретац выд-ной обл как записи: блоку памяти

(j связан с указателем Р) присвоили тип Zap}

Zap(P^).Y := 47; {Интерпретац выдел-ой обл как записи}

Real(P^) := 0.213; {Интерпр-ция выдел-ой области как Real}

St(P^) := 'ПРОГА'; {Интерпр-ия выделенной области как строки}

Freemem (P, 100); {Освободили блока памяти в 100 байт}

6.2.3. Процедуры Mark (пометить) и Release (освободить)

- повышают эффективность работы с динамической памятью. Обращение к процедуре **Mark** имеет вид: **Mark (P1);**

**P1** – переменная типа **указатель**, ссылается на переменную любого типа.

Процедура **Mark** присваивает своему параметру **P1** адрес начала свободной области динамической памяти.

Помеченная область с помощью процедур **New***/***Getmem** – д/размещения отдельных динамических переменных.

Когда динамические переменные будут ненужны, занимаемую ими память освободит процедура **Release** с параметром Р1: **Release (P1); -**  освобождает всю память, начиная с адреса, полученного при выполнении последней процедуры **Mark*.*** После **Release** значение указателя: **P1=****Nil*.***

Схематически представление работы с динамической памятью с и без использования процедур **Mark/Release:**

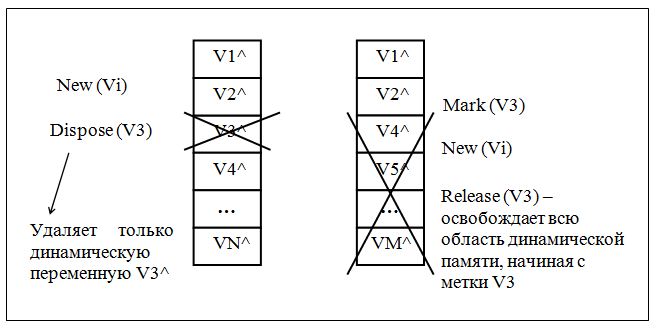
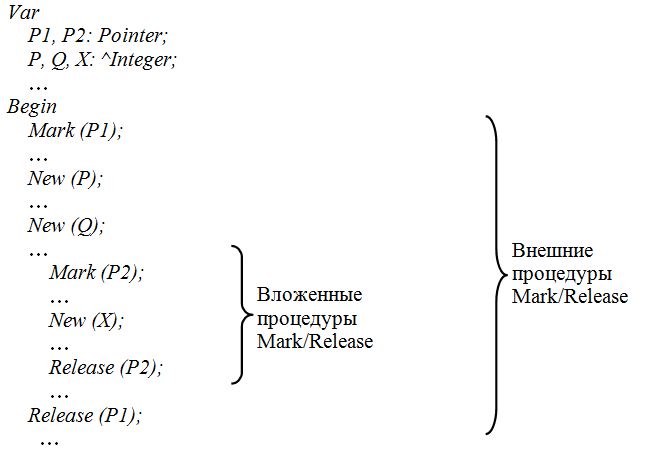


Рис 6.10 – Схематич представление работы с динамической памятью   
 с и без процедур **Mark/Release**

Параметр проц-ы **Mark** нельзя использовать как параметр процедур **New/Getmem** + нельзя изменять (до использования в соответствующей процедуре ***Release*)**.

Возможна **вложенность** процедур **Mark.** Тогда каждой процедуре ***Mark*** соответствует своя ***Release*.**

**Пример 6.10.** Использование **вложенных** процедур ***Mark/Release*.**



В 1 проге **не** используют одновременно процедуры **Dispose**/**Freemem** и **Mark/Release** – рез-ты непредсказуемы (механизмы управлен свободной памятью для *Dispose/Freemem* и *Mark/Release* ***–*** отличаются).

Вместо любой **автом**атической переменной м использовать **динамич**ескую переменную.

Основной «+» использования динам переменных– **экономия памяти**.

«-»: 1. длинее текст программы из-за использования **New/Dispose, Mark/Release, Getmem/Freemem;**

**2.** снижается наглядность программы: переменная с указателем (динамической переменной);

3. снижается **быстродействие** программы: во время ее выполнения надо размещать порожденную динамич переменную в памяти машины, за счет формирования значения соответствующего указателя и за счет усложнения доступа к значению динамической переменной.

## Раздел 7. Динамические структуры данных

7.1. Динамические цепочки 

7.1.1. Структура динамической цепочки

**Динамические цепочки** - аналоги строк текущей длины. В строке каждый следующий элемент занимает ячейку памяти со следующим **адресом** по порядку.

Элементы строки в памяти размещают произвольно, если каждому элементу дать явн указание места: где находится следующий за ним элемент. Тогда каждый элемент строки д им **2 поля**: в 1ом (**Element)** – символ строки– **тело звена**, во 2м (**Adrcled**) – ссылка на следующий элемент строки (адрес следующего элемента) – **справочная часть.**

Каждая такая пара = **звено**. Ссылки сцепляют звенья в 1 **цепочку**, а такой способ последовательности эл-ов - **сцепление**. Сцепление применяется для представления сложных динамических структур данных – строк, списков, деревьев и т д.

**Пример 7.1.** Объявление звена цепочки.

**Type**

**Adr = ^Zveno; // Adr -** ссылки на программные элементы типа **Zveno**

**Zveno = Record**

**Element: Char;**

**Adrcled: Adr**

**End;**

**Adr** *-* ссылки на программные элементы типа ***Zveno***. Имя типа ***Zveno*** используется до его описания (при описании типа ***Adr*** (ссылоч-го типа)). **Ссылочный тип –** (!) тип, где м использовать имя до его описания. Обратную последовательность объявлений (вначале описать тип ***Zveno***, а потом – тип ***Adr***) в прим 7.1 нельзя использовать: в типе **Zveno** использ-ся неописанный тип **Adr**, а **Zveno** не является ссылочным.

Последнее звено цепочки д б со ссылкой **Nil –** признак **конца** цепочки.

Для работы с цепочкой надо использовать **2 указателя**: **ссылку** на ее I звено (**Adr1**) и на текущее (**Adrzv**). Ссылки д иметь тип **Adr:**

**Var**

**Adr1, Adrzv: Adr;**

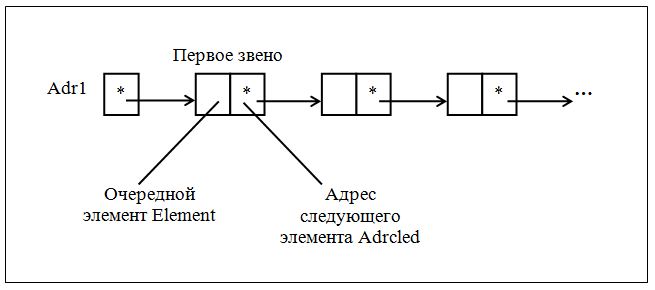


Рис 7.1 – Схематическое представление **цепочки**

Для удобства работы с цепочкой в нее включается заглавное «**0**-ое» звено (это **не** обязательно). В поле **Adrcled** данного звена содержится ссылка на I звено цепочки (его адрес). Поле **Element** - для хранения инфы **о строке.**

7.1.2. Формирование цепочки.

Пусть есть объява звена цепоч по прим 7.1.

Type

Adr = ^Zveno; // Adr - ссылки на программные элементы типа Zveno

Zveno = Record

Element: Char;

Adrcled: Adr

End;

**Алгоритмформирования цепочки*:***

Предварительно перед выполнением этапов 1 – 6 надо сформир-ть **заглавное** «**0**-ое» звено – голову?. При формиров цепочки текущее звено –всегда последнее, поэтому адрес следующего звена **– Nil**

1. Отвести область памяти для следующего звена. Его адрес занести в поле **Adrcled**текущего звена. Это всё происходит в 0-ом звене?
2. Новое звено (I) сделать текущим (занести его адрес в указатель текущего (предыдущ) звена **Adrzv**).
3. В поле **Element** текущего звена занести очередной символ.
4. В поле **Adrcled**текущего звена занести **Nil*.***
5. Прочитать следующий символ исходного текста.
6. **Повторить** этапы алгоритма 1-5.

Схема к алгоритму формирования цепочки с заглавным звеном - рис 7.2. Номера в фигурных скобках - **действия соответств-их этапов алгоритма.**

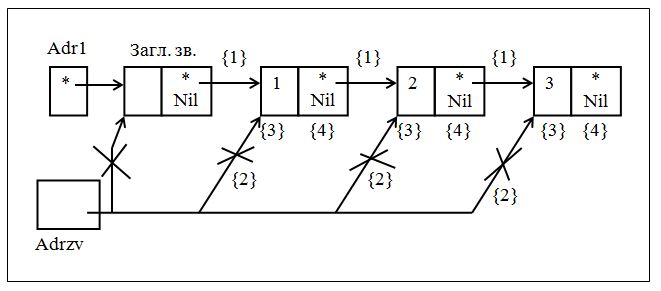


Рис 7.2 – Схематич пояснения к алгоритму формирования цепочки

**Пример 7.2. Формирован** цепочки: **ввод** исходной строки и представление ее в виде цепочки. Признак конца строки – **точка**. Раздел типов = пр-ру 7.1

**Type**

Adr = ^Zveno;

Zveno = **Record**

Element: Char;

Adrcled: Adr

**End**;

**Var**

Adr1, Adrzv: Adr; {Adr1 – адр заглавн звена, Adrzv – адр текущ зв}

Simv: Char;

{======== Формирование заглавного звена цепочки – головы ========}

**Begin**

(1) **New**(Adr1); {Отводят место в памяти д/заглавного звена цепочки}

(2) Adrzv:= Adr1; {Формируется **адрес** текущего звена}/голову сдел-и текущ?

Adrzv^.Adrcled := Nil; {При формиров цепочки текущее звено – всегда

последнее! поэтому адрес следующег звена – Nil}/прощупали – там пусто

//---конец формир головы

**Read (Simv);** {Читается **первая** буква исходного текста}

**While** Simv <> '.' **Do** {Формирование текущих звеньев цепочки:}

**Begin**

{1} **New** (Adrzv^.Adrcled); //Дать обл памяти д/следующ звена, где б **I** буква?

//Его адрес занести в поле Adrcled текущего звена.

{2} Adrzv:= Adrzv^.Adrcled; //Новое звено сделать текущим

// (его адрес занести в указатель текущего звена Adrzv).

{3} Adrzv^.Element:= Simv;//В поле Element текущ звен занест очередн симв

{4} Adrzv^.Adrcled:= Nil; //В поле Adrcled текущего звена внести Nil

{5}**Read (Simv);** //Прочитать следующий символ исходного текста.

**End**

№№{1} – {5} = №№ этапов алгоритма формирования цепочки.

**type**

**Адр** = \*звено

звено = **record**

Элемент: символ

**След-адрес**: **Адр**

**end**

**var**

**Голова**, **Текущий**: **Адр**

**Симв**: символ

**begin**

**new**(**Голова**){место в **памяти** д/головы цепочки}

**Текущий**:= **Голова** {Формируется **адрес** текущего звена}

**Текущий**\*.**След-адрес**:= **0**{адрес следующего звена – Nil}

**read**(**Симв**) {Читается первая буква исходного текста}

**while** **Симв**<>"." **do**{Формир-ие текущих звеньев цепочки:}

**begin**

**new**(**Текущий**\*.**След-адрес**){обл памяти д/следующего звена.

Его адрес занести в поле След-адрес текущего звена}

**Текущий**:= **Текущий**\*.**След-адрес**{Новое звено сделать

текущим: внести его адрес в указатель текущ-о звена}

**Текущий**\*.Элемент:= **Симв**{В поле Element текущего

звена занести очередной символ}

**Текущий**\*.**След-адрес**:= **0**{В поле Adrcled текущего

звена занести Nil}

**read**(**Симв**){Прочитать следующий символ исходного текста}

**end**

...

3 **операции** над динамическими цепочками :

1. **поиск**; 7.1.3.
2. вставка; 7.1.5.
3. удаление. 7.1.4.

7.1.3. Поиск элемента в цепочке (1.)

При поиске элемента в цепочке надо **последовательно** перебрать **все** звенья. Для перехода от звена к звену –выполнять оператор **присваивания в цикле**

**Adrzv := Adrzv^.Adrcled**

Т е присваивать указателю текущего звена **ссылку** на следующее в качестве нового значения. Данный оператор присваивания – аналог оператора  **I := I + 1** для получения № ***I*** следующего элемента при векторном представлении строки типа **Array Of Char***/***String*.***

**Пример 7.3.** Поиск эл-нта в строке-цепочке. Посчитать к-во его вхождений в строку:

..........

**Var**

K: Integer; {Счетчик числа вхождений искомого элемента}

Bykva: Char; {- буква, которую надо найти – ее №}

**Begin**

…

Adrzv := Adr1; {Адресу текущ зв присваив-ся знач-ие адреса нулевого зв

(заглавного)}//голова станов-ся текущим?

K := 0;

Read(Bykva); {Чтение искомой буквы}

**While** Adrzv^.Adrcled **<> Nil** **Do** {**Пока не конец строки**}

**Begin**

Adrzv:= Adrzv^.Adrcled; {Указат-ю присваив знач адреса следующ зв}

**If** Adrzv^.Element = Bykva **Then** {Сравнив содержимое поля эл-та

текущего звена с буквой}

K := K+1;

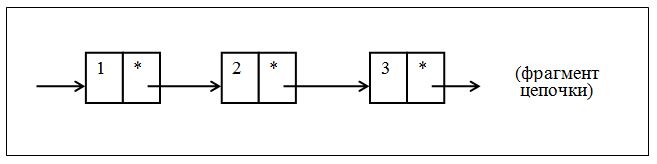
**End**;

**…**

7.1.4. Удаление элемента из цепочки (3)

- надо изменить **ссылку** у **предыдущего** элемента. If какое-то звено есть, но на него нет ссылки из другого звена, то оно недоступно при последов-ом переборе звеньев цепочки. => это звено в цепочку не входит. За новую ссылку принять ссылку в ~~исключаемом~~ элементе (звено 3). М б в СЛЕДУЮЩЕМ? Эл-те?

Например, в исходной цепочке (рис 7.3) надо удалить **элемент 2:** надо, чтобы **звено 1 ссылалось на звено 3** (рис 7.4).



**Рис 7.3** – Фрагмент **исходной** цепи

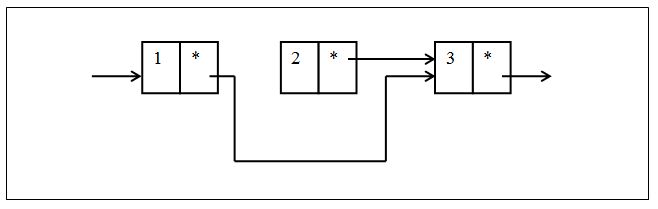


Рис 7.4 – Результат **удаления** элемента 2

**Пример 7.4.** Удаление из цепочки 2го элемента (см прим. 7.1, 2, 3, рис 7.3, 7.4). Для удаления надо использовать оператор

**Adrzv^.Adrcled := Adrzv^.Adrcled^.Adrcled**

Поле адреса эл-та 1 Поле адреса эл-та ~~2~~ (исключаемого)

~~В указателе текущего звена~~ **~~Adrzv~~**~~д б адрес эл-та~~ **~~1~~** ~~(предыдущего исключаемому).~~ – это же не 2связан список!! Что за бред! Какой Предыдущ?

В указателе зв 1 д б адрес звена, следующ за удаляемым – это ПРАВДА!

**Пример 7.5.** Процедура удаления эл-та из цепочки (см пр 7.1–7.3)

**Procedure Udal (Zv: Adr);**

**Var** A: Adr; {A – вспомогательная переменная}

**Begin //** Zv - где объява? Шо цэ?

A := Zv^.Adrcled; {В А положили **адрес** удаляемого звена (2)}

Zv^.Adrcled := Zv^.Adrcled^.Adrcled; {Удален звена из цепочки}

{или равносильно Zv^.Adrcled := А^.Adrcled;}

Dispose (A) {Освобожден памяти, занимаемой удаленным звеном}

**End**;

За параметр **Zv** процедуры д передаваться адрес звена~~,~~ **~~предыдущего~~****~~удаляемому~~**. -- Опять маразм?! – СЛЕДУЮЩЕГО ЗА удаляемым.

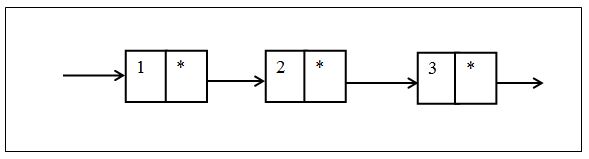
В примере 7.4 удаление элемента из цепочки выполняется быстрее, но память расходуется **неэфф**ективно (нет процедуры **Dispose**).

В примере 7.5 удаление - медленнее, но память расходуется эффективнее. Выбор зависит от хар-ра пославленной задачи.

7.1.5. Вставка элемента в цепочку

- объединение отдельных звеньев в 1 цепь.

Пусть в цепочку (рис 7.5) после элемента 1 надо вставить **элемент 4**.



**Рис 7.5** – Фрагмент исходной цепочки

После вставки цепочка схематически будет выглядеть так:

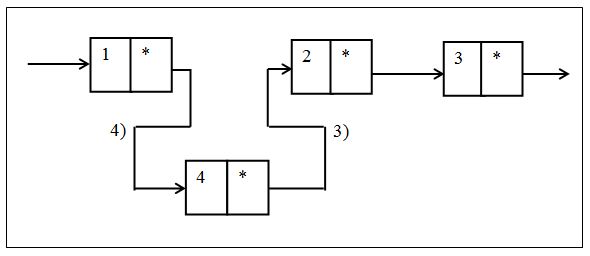


Рис 7.6 – Результат вставки **элемента 4**

**Алгоритм** вставки нового звена (после заданного):

1. Создать **новую** динамическую переменную (запись типа **Zveno**), которой будет представленно **вставляемое** звено.

2. В поле **Element** этой пременной занести вставляемый элемент (символ).

3. В поле **Adrcled** этой переменной занести ссылку **из** поля **Adrcled** предыдущего звена.

1. В поле **Adrcled** предыдущего звена занести ссылку на это новое звено.

№№ 3), 4) (рис 7.6) – действия, соответств-ие этапам 3 и 4 алгоритма.

**Пример 7.6.** Процедура вставки элемента в цепочку.

**Procedure** Vstav (Zv: Adr; El: Char); {**Zv** – адрес предыдущего звена}

**Var**

Q: Adr;

**Begin**

{1} **New** (Q); {Создать новую динам перем (запись типа Zveno), которой б

представленно вставляемое звено.}

{2} Q^.Element := El; {В поле Element этой премен занести вставляемый эл-т}

{3} Q^.Adrcled := Zv^.Adrcled; {В поле Adrcled этой перемен занести ссылку

из поля Adrcled предыдущего звена.}

{4} Zv^.Adrcled := Q {В поле Adrcled предыдущ зв занести ссылку на это зв}

**End**;

№№ операторов – соответствуют №№ алгоритма вставки 1-4.

7.1.6. Линейный однонаправленный список

Динам строка-цепочка - частный случай **линейного однонаправленного списка**. В случае цепочки информационные элементы списка – символы типа **Char**, а в общем случае информ элементы – значения любого типа: числа, массивы, записи, ….

Принцип организации **информац**ионных эл-ов в список **тот же**: этот элемент очередного звена снабжается ссылкой на следующее звено.

**Пример 7.7.** Определение **структуры** звена 1направл-ого списка.

**Type**

**Adres1 = ^Zveno1;**

**Zveno1 = Record**

**Element: <Тип\_элемента\_списка>**

**Adrcled: Adres1;**

**End;**

**Элементы** списка – значения 1го и того же типа.

*«-»* 1направленного списка – по нему можно двигаться только в 1ом направл-ии: от заглавного звена списка к последнему: замедляет работу.

7.2. Двунаправленные списки

- от каждого звена по списку двигаться в любом направлении. Каждое звено 2направленного списка содержит **2 поля** ссылочного типа. Значение 1о поля - ссылка на **следующее** звено списка, значение 2го поля - ссылка на **предыдущее (**звено списка).

**Структур**а звена 2направлен-го списка опред-ся описанием **типа** пр 7.8:

**Пример 7.8.** Описание **типа** звена 2направленного списка.

**Type**

**Adr2 = ^Zveno2;**

**Zveno2 = Record**

**Element: <Тип элемента-списка>;**

**Adrcled: Adr2;**

**Adrpred: Adr2;**

**End;**

Схематично двунаправленный **список с заглавным** звеном (рис 7.7):

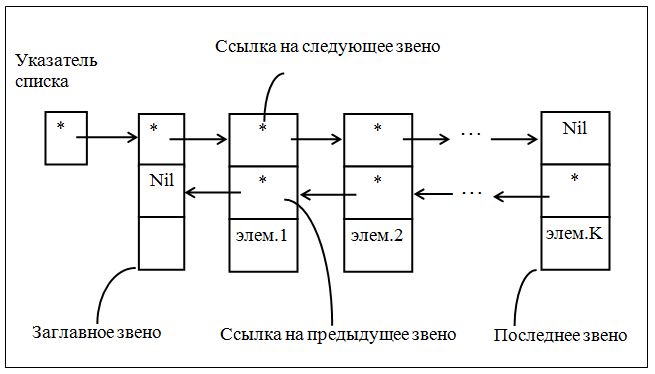


Рис 7.7 – Представление 2направлен-го списка с заглавным звеном

«**элем. *i***» (эл-т ***К***) - **информац**ионная часть ***i-***ого звена.

У заглавного звена списка **нет** предыдущего элемента. У последнего звена списка **нет** следующего элемента, => в поле **Adrcled** последнего и в поле **Adrpred** заглавного звеньев – пустая ссылка **Nil.**

На основе 2направленного списка м организовать 2**направленные кольцевые списки**: значение поля **Adrcled последнего** звена – ссылка на **заглавное** звено, а знач-е поля **Adrpred** заглавного – ссылка напоследнее:

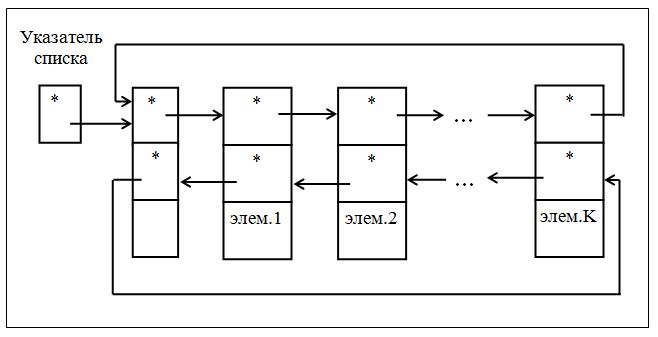


Рис 7.8 – 2направлен кольцевойсписок с заглавным звеном (**1й способ** организации кольца**:** заглавное звено списка включено в кольцо.)

При 2**ом** способе организации кольцевого списка заглавное звено списка **не вкл**ючено в кольцо:

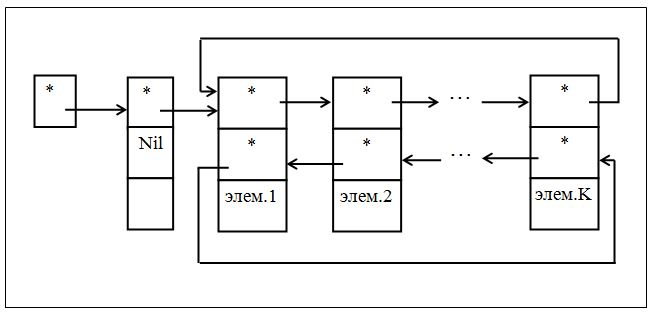


Рис 7.9 –2направлен кольцев спис с заглавным зв (2й способ организ кольца)

«+»1го способа – простореализуется вставка нового звена как в начало списка, так и в конец.

*«-»* – при **циклической** обработке элементов списка надо проверять, не является ли очередное звено заглавным (звеном списка).

У 2го способа данного недостатока **нет**, но труднее реализовать добавление звена в конец списка.

Те же 3 **операции** над 2направл-ми списками, что и над строкой-цепочкой:

1. **поиск** (элемента в списке);
2. **вставка** (элемента в указанное место списка);
3. **удаление** (из списка заданного элемента).

Для работы с 2направленными списками использовать 2 указателя – на **заглавное** (голову) и на **текущее** звено.

Ниже - реализация д/**1го** способа организ **кольцевого** списк (заглавн зв вкл).

7.2.1. Вставка элемента

**Пример 7.8.** Описание типа звена 2направленного списка.

**Type**

**Adr2 = ^Zveno2;**

**Zveno2 = Record**

**Adrcled: Adr2;**

**Adrpred: Adr2;**

**Element: <Тип** элемента-списка**>**

**End;**

**Алгоритм** **вставки** элемента в 2направленном кольцевом списке:

1. По**рождение** **нового** звена – поржденное звено Q.
2. Занесение **вставляемого элемента** **3** в информационное поле порожденного звена.
3. Заносим **в** поле **Adrcled** рожденного звена **3 ссылку** на следующ эл-т из предыдущего звена **1** **Q^.Adrcled:= Predzv^.Adrcled;**
4. вносим в поле **Adrpred** рожденного зв ссылку на предыдущ эл из следующего звена **3**. **Q^.Adrpred := Predzv;**
5. Занесение в поле «**Adrpred»** звена **2 (**следующего за вставляемым) ссылки на вставляемое (Q) звенo. **Predzv^.Adrcled^.Adrpred := Q;**
6. Занесение в поле «**Adrcled»** предшествующего звена 1 ссылки на вставляемое (**Q**) звено. **Predzv^.Adrcled := Q;**

{3} Q^.Adrcled := Predzv^.Adrcled; //из зв1 ссыл на поле зв 2

{4} Q^.Adrpred := Predzv; //из зв 2 ссыл на поле предыд зв 1 + ссылка на

предыдущ зв в проц-ру **Vstav**перед-ся как **параметр**

{5} Predzv^.Adrcled^.Adrpred := Q;

{6} Predzv^.Adrcled := Q; //заплн пустоту зв 1 поле след

Схематично вставка элемента в 2направл кольцевой список с заглавным звеном – на рис 7.10 – рис 7.11.

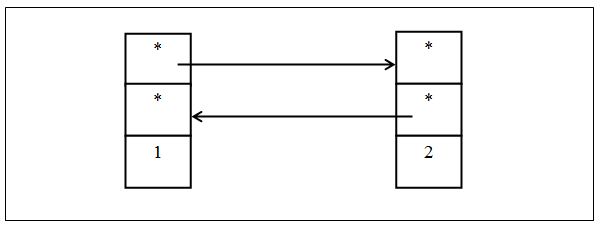


Рис 7.10 – исходноe двунаправленного списка

В исходный список после элемента **1** вставляется элемент **3**.

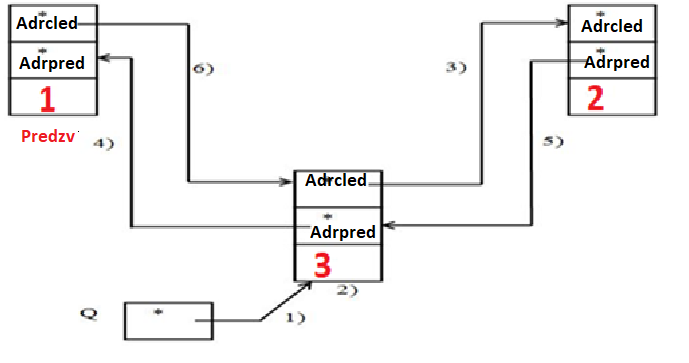


Рис 7.11–результир 2направл спис. 1)–6) =№ этапов алгоритма. Откуда берем

**Пример 7.9.** Процедура вставки.

**Procedure Vstav** (Elem: <Тип\_эл-та >; **Predzv: Adr2**); {Elem – вставляем эл-т,

Predzv – ссыл на предыдущий эл-нт, передается как **параметр**}

**Var** Q: Adr2; //тип запись

**Begin**

{1} **New** (Q); //Порождение нового звена – поржденное звено (см рис)

{2} Q^.Element := Elem; //Занесен вставляемого эл-та в инфо поле порожд зв

{3} Q^.Adrcled := Predzv^.Adrcled; //вносим из предыдущ звена в поле

//Adrcled звена ссылку на следующ эл-нт. (см рис)

{4} Q^.Adrpred := Predzv; //Занесение в поле Adrpred порожденного звена

//ссылки на предыдущий элемент из следующего звена.

{5} Predzv^.Adrcled^.Adrpred := Q; //Занесение в поле Adrpred следующего за вставляемым звена (2) ссылки на вставляемое звено (3).

{6} Predzv^.Adrcled := Q; //Занесение в поле Adrcled предшествующего звена ссылки на вставляемое (порожденное) звено.

**End**;

№№ операторов {1-6} == №№ этапов алгоритма вставки. Реализация **{4}** в примере **отличается** от этапа 4 алгоритма, т. к. ссылка на предыдущее звено в процедуру **Vstav**передается как **параметр**.

7.2.2. Создание 2направленного кольцевого списка с заглавным звеном

- на примере: Для создания **кольцевого списка** можно использовать процедуру **вставки** – см пр 7.9; на каждом шаге создан спис д б з**акольцо**ван

**Пример 7.10. Ввод** исходного **текста** и его представление как 2направленный кольцевой список **с заглавным** звеном. Признак окончания текста – точка. Раздел типов == прим. 7.8.

Type

Adr2 = ^Zveno2;

Zveno2 = Record

Adrcled: Adr2;

Adrpred: Adr2;

Element: <Тип элемента>

End;

Procedure Vstav (Elem: <Тип\_эл-та\_списка>; Predzv: Adr2);

Var

Q: Adr2;

Begin

{1} New (Q); //По**рожд** **нового** звена – поржденное звено (см рис)

{2} Q^.Element := Elem;

{3} Q^.Adrcled := Predzv^.Adrcled; {внос **ссылки** на след эл-т **из пред**ыд}

{4} Q^.Adrpred := Predzv; {внос ссылки на предыд эл-т **из следующего** зв}

{5} Predzv^.Adrcled^.Adrpred := Q; {внос в зв 3 след ссыл на вставляем зв}

{6} Predzv^.Adrcled := Q; {Занес ссылки на вставляемое звено}

End;

**VAR**

**Ring:** Adr2**;**{**Адрес** заглавного звена (Головы) кольца}

**Bykva:** Char**;**{**Символ** исходного текста}

**BEGIN**

{7} **New (Ring);**{- место в памяти д/**заглавн**ого звена}

{8} **Ring^. Adrcled := Ring;** {За**кольцовывается** заглавн зв, т к на кажд

промежуточном этапе список д б кольцом. Само на себя указыв?

{9} **Ring^. Adrpred := Ring;**

**Read (Bykva)**  {Чтение первой буквы текста}

**While Bykva <> ’.’ Do**

**Begin**

**Vstav (Bykva, Ring^. Adrpred);** {**Вызов** проц-ры вставки. Адрес предыдущ созданного звена (последнего звена кольца) **всегда** хранится в заглавном звене в поле **Adrpred**}

**Read (Bykva);**  {Чтение очередной **буквы текста**}

**End**

**END.**

Схематические пояснения к данной программе (рис 7.12): ничего не яснно

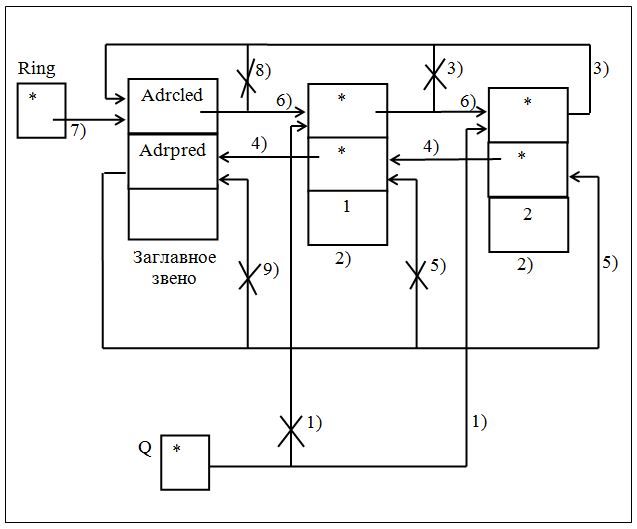


Рис 7.12 – Схематич пояснения к проге прим 7.10. цифры 1) – 6) №№ этапов алгоритма **вставки,** 7) – 9) – №№ соотв-их **операторов**.

Напомним: список за**кольцо**вывается после «+» каждого нового звена.

7.2.3. Удаление элемента

Пусть в проге – описание типа как в примере 7.8. **Алгоритм** удаления элемента – в 2 раза < операц, чем при вставке зв:

1) внесен ссылк в поле ***Adr*pred** зв (3) **из** поля **Adrpred** зв(2) на зв (1)

**Udzv^.Adrcled^. *Adr*pred := Udzv^. Adrpred;**

2) Занесен в поле *Adr***cled** зв (1) (предыдущего удаляемому) ссылки на зв(3) (следующ за удаляемым) из поля **Adrcled** удаляемого зв 2

**Udzv^.Adrpred^.Adrcled := Udzv^. Adrcled;**

3) **Уничтожение** удаляемого звена (2).

На рисунках - схематические пояснения: удаление элемента из 2направленного списка. В исходном списке удаляют звено 2:

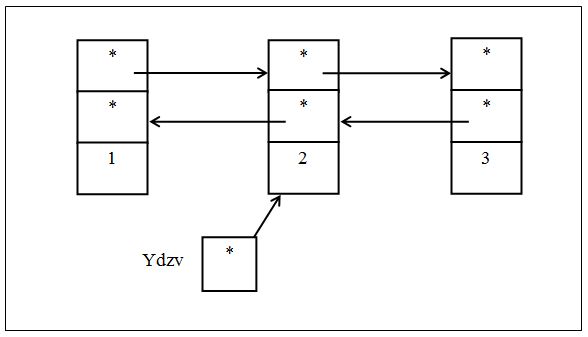


Рис 7.13 – Фрагмент исходн 2направл списка №№ связей = №№ этап алгор-а

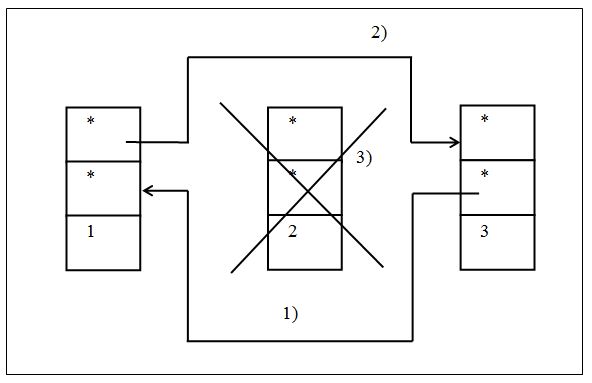


Рис 7.14 – Результат удаления звена 2

**Пример 7.11.** Процедура **удаления** звена из 2направленного списка. №№ операторов == №№ этапов алгоритма удаления звена.

Type

Adr2 = ^Zveno2;

Zveno2 = Record

Adrcled: Adr2;

Adrpred: Adr2;

Element: <Тип элемента >

End;

**Procedure Udalen (Udzv: Adr2); {Udzv** – ссылка на удаляемое зв}

**Begin**

{1} **Udzv^.Adrcled^.Adrpred := Udzv^.Adrpred; //**внесена ссылка в поле

***Adr*pred** зв (**3**) **из** поля ***Adr*pred** зв (2) на зв (1) ;

{2} **Udzv^.Adrpred^.Adrcled := Udzv^.Adrcled;** //внесена ссылка в поле

*Adr***cled** зв (**1**) **из** поля *Adr***cled** зв (2) на зв (3) ;

{3} **Dispose (Udzv); Уничтожение** удаляемого звена (2).

**End;**

**7.2.4. Поиск элемента** **в 2направл-ом кольцевом списке**

- как поиск элемента **в цепочке** (см. п. 7.1.3).

Особенность: в кольцевом списке **нет** последнего элемента формально - каждый элемент имеет ссылку на следующий - учитывать при цикле поиска!

**Пример 7.12.** Ф-ция поиска элемента в 2напр-ом **кольц**ев списке **с** за**главн**ым звеном. Если искомый эл-т есть, возвращаемое знач ф-ции = ***True***, а пар-ру **Iskadr :=** ссылка на звено с данным эл-ом (адрес эл-та)

**Function** Poisk(Adr: Adr2; Elem: <Тип\_эл-та>; **Var** IskАdr: Adr2): Boolean; {Adr – ссыл на заглавн зв; Elem – иском эл-т; IskАdr – адрес искомого эл-та}

**Var**

P, Q: Adr2; {Р - б хранить адрес за**главн** зв Adr, Q – текущего}

B: Boolean; {B – логич-ая перем-ая (наличие искомого эл-та/нет)}

**Begin**

B := False; // - допустим звена нет

P := Adr; {В P – адрес заглавного зв (Головы)}

IskАdr := Nil;

Q := P^.Adrcled; {В Q – адрес первого зв (текущего)}

**While** (P <> Q) **And Not** B {Поиск, пока не дошли до Головы и не

нашли искомый элемент}

**Begin**

**If** Q^.Element = Elem **Then** {Найден! искомый элемент. Ура!}

**Begin**

B := True;

IskАdr := Q {Адрес искомого элемента}

**End**;

Q := Q^.Adrcled {Переход к следующему звену}

**End**;

Poisk := B {Возвращаемое значение}

**End**;

7.3. Очереди и стеки

Понятие очереди в программировании – как в реальной очереди в жизни.

**Очередь** - динамическая структура. Длина очереди и набор образующих ее элементов меняется с **течением времени.**

**2 операции** над очередью:

1. **занесение** элемента в очередь (заказ на обслуживание);
2. **выбор** элемента из очереди (д/обслуживания); выбранный элемент исключают из очереди.

В очереди - **2 позиции**: начало (из этой позиции выбирается элемент из очереди) и конец (в эту позицию помещается элемент).

2вида очереди (отличаются по дисциплине (способу) обслуживания в них элементов):

**1).** ***FIFO*** (*First In – First Out* – первый в очередь – первый из очереди) - поступивший в очередь первым заказ, первым выбирается для обслуживания и удаляется из очереди – сквозная труба.

2). ***LIFO*** (*Last In – First Out* – последний в очередь – первый из очереди) - заказ, поступивший в очередь последним, выбирается первым для обслуживания (и удаляется из очереди) – наз-ся **стек.** Чаще использ.

7.3.1. Очередь LIFO – стеки

Доступна **(!)** позиция – **вершина стека** – позиция, в которой находится последний по времени поступления в стек элемент.

Более быстрое выполнение операций над стеком дает его представление как динамической цепочки звеньев (1-онаправленный список). Вершина стека - первое звено цепочки стека, где заглавное звено не нужно, т к доступна только его вершина.

При использовании структуры 1-направленного списка стек задают описанием типа как в примере 7.7, + д/п-о м ввести тип указателя, чтобы представить стек как единую стр-ру: ссылка на вершину стека ***Stek*:**

Type

Adres1 = ^Zveno1;

Zveno1 = Record

Adrcled: Adres1;

Element: <Тип\_элемента\_списка>

End;

**Stek = Adres1;**

Реальный стек вводится с помощью описания переменной: **Var**  **St: Stek;**

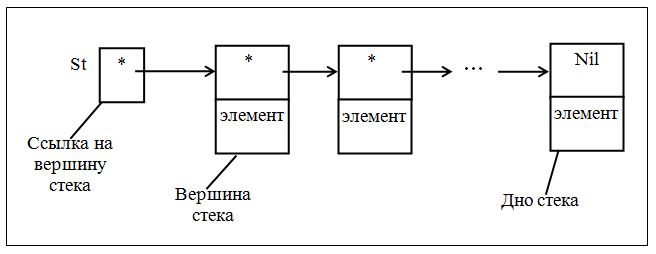


Рис 7.15 – Схематичное представление стека - аналог цепочки

К началу использования стека его надо сделать **пустым**:  **St := Nil;**

**А. Занесение элемента в стек**

**Алгоритм** занесения элемента **в** стек:

1. Создание нового звена.
2. Занесение элемента в новое звено.
3. Занесение **в новое** звено адреса предыдущей вершины стека.
4. Созданное звено сделать **вершиной** стека.

**Пример 7.13. Процедура занесения элемента в стек**. 1й параметр задает нужный стек (если их несколько), 2й – заносимое (в него) значение. **О**писание типа - см пример 7.7.

**Type**

Adres1 = ^Zveno1;

Zveno1 = **Record**

Adrcled: Adres1;

Element: <Тип\_элемента\_списка>

End;

**Stek =** Adres1;

**Procedure Zanes (Var St : Stek; El : <Тип\_**элемента\_стека**>);**

**Var**

**Q: Adres1;**

**Begin**

{1} **New (Q); //** Создание **нового звена**.

{2} **Q^.Element := El;** //Занесение в новое звено **элемента**.

{3} **Q^.Adrcled := St;** Занесен в новое зв адреса предыд верш-ы стека

{4} **St := Q** //Созданное звено сделать **вершиной** стека.

**End;**

**Б. Выбор элемента из стека (**описание типа - из пр-ра 7.7)

**Алгоритм** **выбора** элемента **из** стека (д/обработки):

1. **Прочитать** значение из вершины стека. **A := St^.Element;**
2. Запомнить ссылку на старую вершину. – для чего??? **Q := St;**
3. Исключить первое звено из стека.
4. Уничтожить (стереть) первое звено.

**Пример 7.14.** Процедура выбора элемента **из** стека. 1й параметр задает нужный **стек** (если их неск-ко), во 2й передается значение из вершины стека.

**Type**

Adres1 = ^Zveno1;

Zveno1 = **Record**

Adrcled: Adres1;

Element: <Тип\_элемента\_списка>

End;

**Stek =** Adres1;

**Procedure Vibor (Var St: Stek; Var A: <Тип\_**элементов\_стека**>);**

**Var**

**Q:** Adres1**;**

**Begin**

{1} **A := St^.Element;** //Прочитать значение из вершины стека.

{2} **Q := St;** //Запомнить ссылку на старую вершину.

{3} **St := St^.Adrcled;** //**Исключить** первое звено из стека.

{4} **Dispose (Q)** //Уничтожить первое звено.

**End;**

Если надо ускорить процедуру выбора – операторы **Q:= St** и **Dispose(Q)** не применять. Но это - неэффективное использов-ие памяти.

**В. Создание стека (**описание типа - из пр-ра 7.7)

Д/создания стека м использовать **процедуру** **Zanes** из примера **7.13** (п А).

Procedure Zanes (Var St: Stek; El:<Тип\_элемента\_стека>);

Var Q: Adres1;

Begin

{1} New (Q); // Создание нового звена.

{2} Q^.Element := El; //Занесение в новое звено элемента.

{3} Q^.Adrcled := St; //Занесен в новое зв адреса предыд верш-ы стека

{4} St := Q //Созданное звено сделать вершиной стека.

End;

**Пример 7.15. Создан стека (в**в текста в стек) Признак end текста – точка

**Var**

St: Adres1;

Bykva: Char;

**Begin**

St := Nil; //обнулили, ничего пока нет

Read (Bykva);

**While** Bykva <> '.' **Do**

**Begin**

Zanes (St, Bykva); // Procedure Zanes()

Read (Bykva);

**End;**

**End**.

7.3.2. Очередь FIFO (сквозная труба)

Для её организации используют **2 ссылочные переменные** типа **Adres1** (см. пр 7.7)**:** **Left** – **начало** и **Right** – конец очереди. Слева направо

«+»е эл-та в очередь - в соотв со значен-eм **Right.** Затем значение **Right** изменяется и указывает на **последний** занесенный элемент

**Выбор**ка из очереди элементов - в соотв со значением **Left.** Затем **Left** изменяется и указывает на **следующий** элемент очереди.

Если в очереди 1 элемент - значения **Right =** **Left.** Такое равенство м использовать как признак **конца** очереди при последовательном выборе элементов.

**А. Занесение элемента в очередь FIFO**

**Алгоритм** занесения элемента **в** очередь:

1. Cоздание нового звена.
2. Занесение в последнее звено адреса нового звена.
3. Занесение ***Nil*** в поле **Adrcled** нового звена.
4. Занесение **элемента** (инфы) в информационное поле нового звена.
5. Созданное звено сделать **концом очереди**.

**Пример 7.16. Процедура** занесения элемента в очередь FIFO. I пар-р – **Right** – **адрес последнего** эл-та, 2й – заносимое **значение**. описан типа - прим 7.7.

Type

Adres1 = ^Zveno1;

Zveno1 = Record

Adrcled: Adres1;

Element: <Тип\_элемента\_списка>

End;

**Procedure** Dobavl (**Var** Right: Adres1; El: <Тип\_эл-ов\_очереди>);

**Var**

Q: Adres1;

**Begin**

{1} **New** (Q); //Cоздание нового звена.

{2} Right^.Adrcled := Q; // Занесен в последн звен адреса новог звена

{3} Q^.Adrcled := Nil; // Занесение Nil в поле Adrcled нового звена

{4} Q^.Element := El; // Занесен эле-та в информ-ое поле нов звена

{5} Right := Q // Созданное звено сделать концом очереди.

**End**;

**Б. Выбор элемента из очереди**

**Алгоритм** выбора элемента из очереди:

1. **Чтение** значения из начала очереди.
2. Запоминание **ссылки** на начало очереди. Зачем?
3. Исключение **первого** звена из начала очереди.
4. Уничтожение **первого** звена.

**Пример 7.17.** Проц-ра **выбора эл-та из** очереди. Параметр **Left *–*** д/передачи **адреса** начала очереди. В параметр **Elem** передается знач из нач очереди. описание типа - см пример 7.7.

**Procedure** Udal (**Var** Left: Adres1; **Var** Elem: <Тип\_эл-ов\_очереди>);

**Var**

Q: Adres1;

**Begin**

{1} Elem := Left^.Element; //Чтение значения из начала очереди.

{2} Q := Left; //Запоминание ссылки на начало очереди.

{3} Left := Left^.Adrcled; //Исключен 1-го звена из начала очереди

{4} Dispose(Q) // Уничтожение первого звена.

**End**;

**В. Организация очереди**

описание типа - см пример 7.7.

**Основа** организации очереди – операция занесенияэлем-та в очередь (A)

**Пример 7.18.** Организация очереди. Ввод исходного текста в очередь. Признак конца текста – точка**.** Использ-ть процедуру **Dobavl** (см. пр-p 7.16).

**var**

right, left: adres1;

b: char;

**begin**

read(b);

{1} **new**(left); //{1}-{4} - порождение нового звена

{2} left^.adrsled:= nil;

{3} left^.element:= b;

{4} right:= left;

read(b);

**while** b <> '.' **do**

**begin**

dobavl(right, b);

read(b);

**end**;

**end**;

{i}- действия над очередью FIFOкак №№ операторов в примерах 7.16 – 7.18.

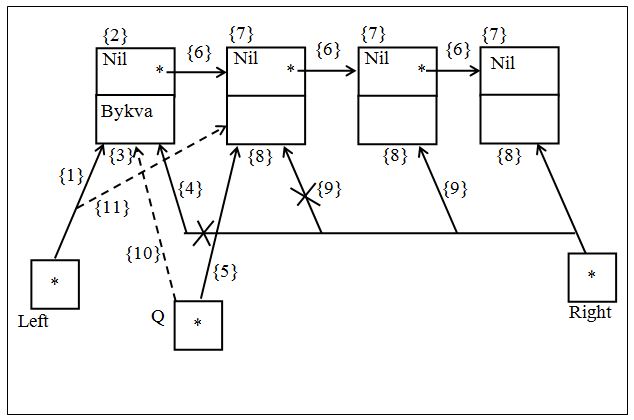


Рис 7.16 – Схематические пояснения к примерам 7.16 – 7.18

7.4. Таблицы. АИС (т☺)

7.4.1. Общие сведения

**Авто**матизиованные **инфо**рмационные **системы** **(АИС).** Их назначение: 1.хранить большое число данных, 2.принимать новые данные и 3.выдавать хранимые данные по запросам. **Данные** предоставляются **записями.**

**Основная задача** **АИС** – организация выдачи по запросу любой записи, независимо какая запись выдавалась перед ней.

Для организации **АИС** используют **таблицы –** структуры данных, где у каждой записи – свое **имя**. Таким образом, **таблица** – это набор именованных записей. **Ключ** записи – **Имя** записи**.** Ключ – целые положительные числа или строки символов одинаковой длины.

Для организации эффективного поиска записи надо, чтобы с ключами м реализовать операции сравнения **(=, <, >**), => лучше, чтобы ключи были у**порядо**чены.

**Операции** над таблицей:

1. **поиск** записи в таблице с заданным ключом;
2. **включение** (**добавл**) в табл записи с заданным ключом (если в табл уже есть запись с этим ключом, то старую запись меняют на новую);
3. **исключение (удален)** из таблицы записи с заданным ключом.

7.4.2. Способы организации таблиц

1. **Однонаправленный список**

**Таблица** – в виде 1направленного списка. Каждое звено списка должно содержать: 1.**ключ** записи, её 2.**текст** и 3.**ссылку** на следующее звено.

*«*+*» способа*:

а) эффективно использ-ся память машины (д/п информация в звене – только ссылка на следующее звено);

б) простой алгоритм перебора записей для поиска нужной записи;

в) простота включения в таблицу новой записи – как новое звено в **конец** списка.

«**-**» – большое **время поиска** нужной записи из-за:

а) если в таблице **N** записей, то д/поиска нужной надо просмотреть в среднем **N/2** элементов списка;

б) если в таблице нет записи с нужным ключом, то надо просмотреть все **N** записей.

1. **1направленный список с упорядоченными записями**

Записи в списке – по **возрастан**ию их ключей.

«+» *способа*: меньшее вр поиска записи по сравнению со способом (1). Поиск записи требует в среднем просмотра **N/2** записей, независимо есть эта запись в таблице или нет.

«-»: усложняется процедура включения новой записи в таблицу.

1. **1направленны список с отдельным хранением текста записи**

- для **ускорения** поиска записи. **Тексты** записей хранятся отдельно от **ключей: п**ри ключе хранится только **ссылка** на текст записи.

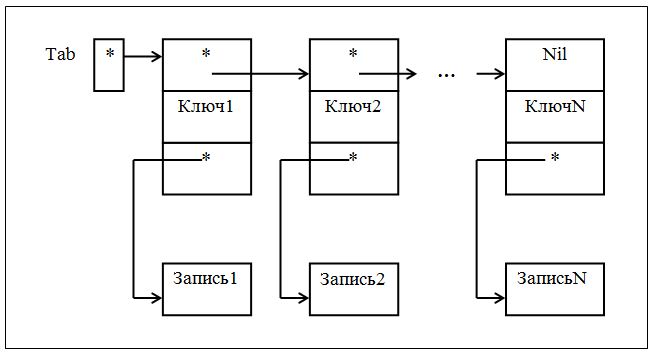


Рис 7.17 – Схема: табл в виде списка с отдельным хранен Текста записи

1. **Представление в виде Массива**

Каждый элемент таблицы – запись с 2я полями – **ключ +** **ссылка** на текст записи. Такие элементы обьединяются в 1мерный массив (а **не** в список).

**Пример 7.19.** Объявление «Т**ипа** таблица» при использовании **массива** записей с 2я полями **– ключ** записи + **ссылка** на текст записи.

**Type**

**Index = 1..N;**

**Text = <Тип\_**текста\_записи**>;**  {напр-р, String}

**Adr = ^Text;**

**Element = Record**

**Kl: Integer**; {Ключ} +

**Adrzap: Adr { ссылка** на текст записи **^Text}**

**End;**

**Mas = Array [Index] Of Element;**

**Var**

**Tabl: Mas;**

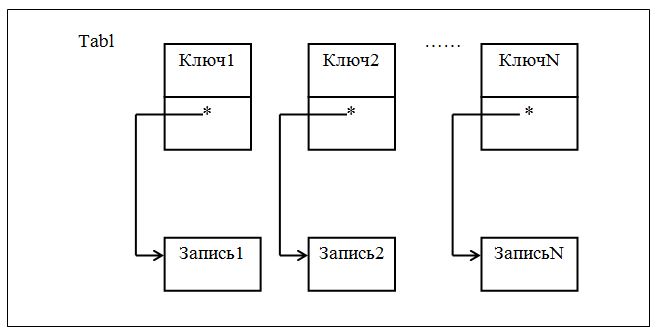


Рис 7.18 – Схематическое представление таблицы в виде массива

Для организации эффект-го поиска в таблице этим способом, надо, чтобы элементы **массива** были упорядочены по возрастанию **ключей**.

Задача поиска записи с заданным ключом = задачa нахождения элемента в массиве **Tabl** сэтим ключом: if определен **I** **индекс** этого элемента, то искомая запись и есть значение переменной **Tabl[I].Adrzap^**

Для поиска элемента с заданным ключом используют **метод половинного деления (дихотомический поиск)**. Его суть:

Берется средний элемент массива с индексом (№) **N/2.** Если искомый ключ **К <** ключа в элементе с номером **N/2**, то требуемый элемент находится в первой половине массива, иначе – во 2ой. На следующем этапе нужная половина массива опять делится 1/2 и определяется, в какой из половин находится соответствующий элемент и так далее. Процесс закончен, когда требуемая запись найдена или зона поиска – пустая (записи с нужным ключом в таблице нет).

Так как на каждом шаге зона поиска уменьшается в 2 раза, то для завершения поиска надо не более ** шагов. Т о, эффект способа по сравнению со способами 1 и 2 быстро возрастает с ростом ***N***.

**Пример 7.20.** Логическая ф-ция **Poisk** – поиск нужного ключа в таблице дихотомическим методом. Ф-ция ***Poisk*** = **True,** if есть нужн ключ, else - False

Есть 2 параметра: 1) **К** – искомый ключ, 2) **Nomer** – **№** **эл**-та массива ***Tabl*** с этим ключом **К**. Т е в параметр-перемен **Nomer** ф-ция устанавлив значен индекса (=№ эл-та) массива **Tabl с** нужным ключом. Объявл –пр 7.19.

**type**

index = 1..n;

text = string; //<Тип\_текста\_записи>;

adr = ^text;

element = **record**

kl: integer; {Ключ}

adrzap: adr; { ссылка на текст записи }

**end**;

Mas = **array**[index] **of** element;

**Var**

Tabl: Mas;

**function** Poisk(k: integer; **var** nomer: index):boolean;

**var**

lev, prav: integer; {Левая и правая границы поиска}

B: boolean;

i: index;

**begin**

lev:= 1;

prav:= n; {N – размер таблицы Tabl}

B:= false;

Nomer:= 0; {if в Nomer остает знач 0,то искомого ключа в Tabl нет}

**repeat**

i:= (lev + prav) **div** 2; {Nomer – № эл-та}

**if** K = Tabl[i].Kl **then** { К – искомый ключ }//if = - т е сразу же нашли!

**begin**

B:= true;

Nomer:= i;

**end**

**else**

**if** K < Tabl[i].Kl **then**//if < - ищем в левой ½

Prav:= i - 1

**else**

lev:= i + 1; //if > - ищем в правой ½

**until** B **or** (lev > prav); {Иском ключ найд/пустая зона(нет нужн ключа)}

Poisk:= B; {Возвращаемое значение функции}

**end**;

«-» *способа (массив)* – плохо приспособлен для реализации добавления и удаления записей, т к надо сдвигать все следующие после вставляемого или исключаемого элементы массива в ту или другую сторону для поддержания упорядоченности элементов. Поэтому способ 4 удобен, если таблица **изменяется редко.**

1. **Двоичное дерево** (это всё Способы организации таблиц)

Реализация таблицы в виде двоичного дерева позволяет эффективно выполнять все три операции над таблицей. См. следующий подраздел.

7.5. Двоичные деревья

7.5.1. Структура двоичного дерева (2ич Дер)

**Схематично двоичное дерево** – набор вершин, соединенных стрелками (ветвями, рис 7.19). Из каждой вершины выходит до 2 ветвей, направленных влево и вправо -вниз. В каждую вершину (кроме 1й) входит 1 стрелка. **Корень** - **вершина**, в которую не входит ни 1 стрелка. Вершины, из которых не выходит ни одна стрелка – **листья**.

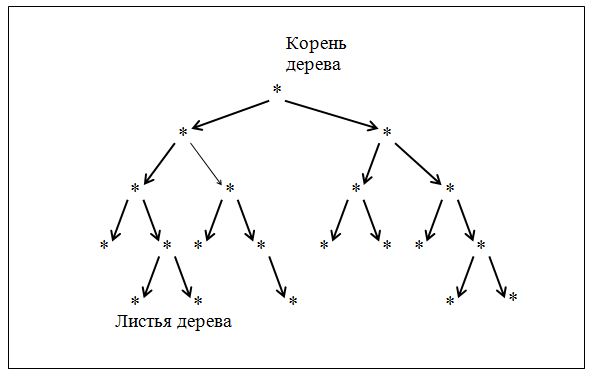


Рис 7.19 – Схематическое представление двоичного дерева

В **таблице** в виде дерева – **тексты** записей хранят **отдельно.**

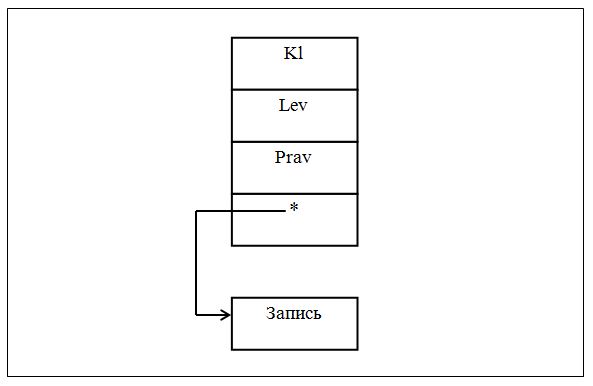
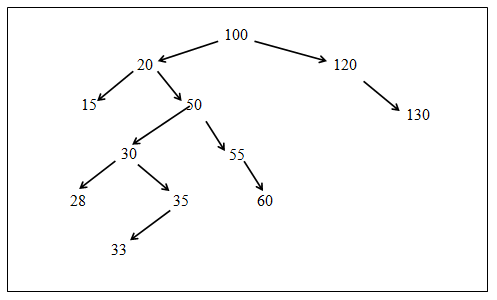
Каждая вершина дерева (**звено**) – запись из **4**х полей: 1)**ключ** записи **(Kl)** **,** 2)**ссылка** на вершину **влево**-вниз (**Lev**), 3)**ссылка** на вершину **вправо**-вниз (**Prav**) и 4)**ссылка на текст** записи (\*).

Рис 7.20 – Стр-ра звена 2ич Дер

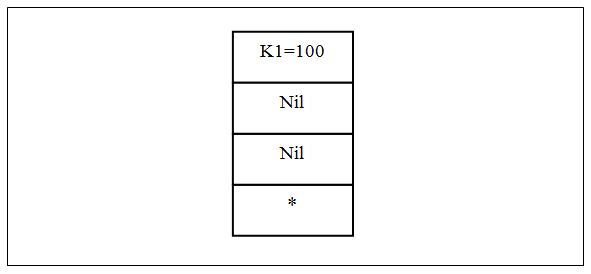
7.5.2. Построение дерева. Принцип

Первая запись – **корень** дерева (вершина дерева). Если **ключ** следующей записи (N/2) < ключа корня – она будет **левая** вершина, иначе – **правая**. Ключ (**К)** каждой следующей записи сравнивается последовательно: с ключом корня, затем с ключами ниже (тех записей, которые находятся на соответствующей ветви дерева). По результату сравнения ключа ***К*** с ключом в очередной вершине идет переход влево или вправо от нее пока не найдется вершина чтобы присоединить новую с ключом ***К***. В зависимости от результата сравнения ключа в этой вершине с поступившим ключом ***К*** вновь сформированная вершина становится левой или правой для найденной вершины.

**Пример 7.21.** Построение дерева. В первоначально пустую таблицу заносят поступающ записи с ключами 100, 20, 120, 50, 15, 130, 55, 30, 35, 60, 33, 28:  Рис 7.21 – результир Дер

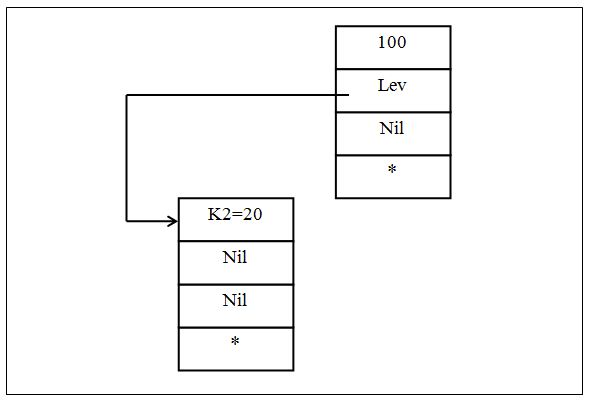
С учетом структуры этого дерева послед-сть его построения здесь такая:

После поступления первой записи с ключом **К1 = 100** дерево имеет вид:

 Рис 7.22 – Вид дерева после поступления 1й записи

Т к у этой вершины пока нет листьев справа-внизу и слева-внизу, то в соответ-ие поля (**ссылки** на левые и правые вершины) – пустые ссылки (**Nil**).

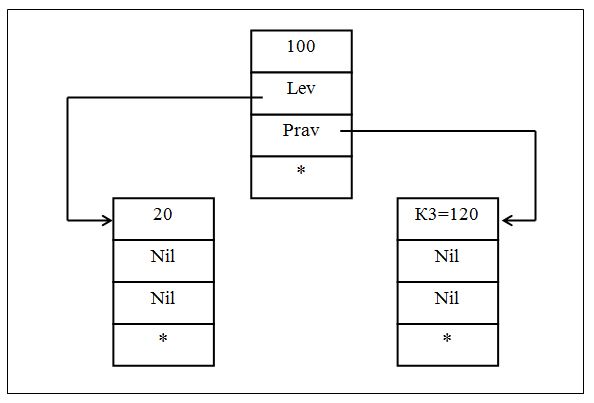
После поступления 2й записи с ключом **К2 = 20** дерево имеет вид:

Рис 7.23 – Вид дер после 2й записс

Т к **К2 < К1***,* то новая вершина – левая по отношению к 1й (корню). Д/этого в поле **Lev**(адрес левой вершины) в вершину 1 заносится **адрес** созданной вершины 2.

После 3й записи с ключом **К3 = 120** дерево имеет вид (рис 7.24): т к **К3 > К1**, то вершина 120 становится правой и в поле ***Prav*** вершины 1 заносится **адрес** вершины 3.

Аналогичные для подключения всех последующих вершин дерева.

Рис7.24 – дерево с 3й запи

Т о, подходящей вершиной, к которой можно подсоединить новую вершину - вершина соответствующей ветви дерева с полем **Lev / Prav** (в зависимости от соотношения ключей ее и новой вершины) = **Nil.**

Двоичное дерево м вводить в употребление с помощ описания прим 7.22:

**Пример 7.22. Описание** двоичного дерева.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

**Var**

DvDer: **AdrZv**;

Для двоичного дерева те же **операции,** как д/таблиц (3шт):

1. **поиск** записи с заданным ключом;
2. **включение** записи с заданным ключом (если уже есть запись с таким ключом, то старая запись меняется на новую);
3. **исключение** записи с заданным ключом.

7.5.3. Поиск записи в двоичном дереве

Рассмотрим его реализацию на примере. Объявление по пр-ру 7.22.

**Пример 7.23.** Логич ф-ция ищет **вершину** дерева с задан ключом. Парам-ры:

**К** - заданный ключ,

**D** - ссылка на **корень** дерева (где б поиск)

**Rez** - переменная, которой := ссылка на найденное звено (если она есть) или ссылка на вершину, после обработки которой поиск прекращен (если её нет (закончилась ветвь)).

……………………………………………………….

**Function** Poisk(K: Integer; **Var** D, Rez: **AdrZv**): Boolean;

**Var**

P, Q: **AdrZv**; //Q – адрес обработанной вершины, Р – адрес корня дерева

B: Boolean; {В – признак что ключ найден! Poisk:= B;}

**Begin**

B := False;

P := D; {В Р заносят адрес корня дерева. Потом в Р б храниться

адрес вершины, подлежащей обработке}

Q := Nil; {Q – адрес обработанной вершины}

**If** D <> Nil **Then** {если дерево не пустое}

**Repeat // D** - ссылка на **корень** дерева

Q := P; {Q – адрес обработанной вершины}

**If** P^.Kl = K **Then** {в корне Р конкретное звено}//сразу нашли её

B:= True {Найдена! нужная вершина}

**Else** //смотрим: < или > искомая вершина корня дерева

**If** K < P^.Kl **Then** {К - заданный ключ}//вершина меньше корня

P := P^.**Lev**

**Else**

P := P^.**Prav**

**Until** B **Or** (P = Nil); {Поиск, пока не найден ключ (В=True) или

пока не закончилась соответств-ая ветвь}

Poisk := B; {Возвращаемое значение}

Rez := Q; {Q – адрес звена с нужным ключом / адрес конца ветви}

**End**;

Скорость поиска в двоичном дереве = скорости дихотомического поиска (см. пример 7.20).

7.5.4. Включение записи в дерево

- надо найти вершину, куда м присоединить новую вершину, соответствующую включаемой записи.

Алгоритм поиска **вершины** = алгоритму поиска вершины с заданным ключом (см. пример 7.23). Вершина найдена, если ***В =* False**. Тогда в **Rez** – адрес вершины, к которой можно подсоединить включаемую вершину. Вершина найдена, если за очередной ссылкой (определяет ветвь продолжения поиска) б ссылка Nil.

Пусть в таблице **нет** записи с тем же ключом, что и у включаемой записи. Объявление – по примеру 7.22.

**Пример 7.24.** Процедура **включения** записи в дерево. Параметры процед: ***К*** – ключ, ***D*** – адрес корня Дер, Zap – текст вставляемой записи.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

**Var**

DvDer: **AdrZv**;

**Procedure** Vkl(K: Integer; **Var** D: AdrZv; **Zap**: Tekst);

**Var**

Q, S: AdrZv; //S – вершина, Q-адрес вершины, к j присоед нов верш

T: AdrТ;

**Begin**

**If Not** Poisk(K, D, Q) **Then** // Poisk – процед-ра из прим 7.23

**Begin**

**New** (T); {Создано звено д/занесения текста записи}

T^ := **Zap**; {Занесен в таблицу текст записи}

**New** (**S**); {Создана просто новая вершина в дереве}

S^.Kl := K; {Занесен ключ К в поле Kl новой вершины}

S^.Adr := T; {Занесен адрес текста записи в поле Adr нов верш}

S^.Lev := Nil;

S^.Prav := Nil; {Созданную вершину сделали листом дерева}

**If** D = Nil **Then** {if дерево еще пустоe, то созданное звено– корень дер}

D := **S**

**Else**

**If** K < Q^.Kl **Then** {В Q-адрес вершины, к которой

присоединяется новая вершина}

Q^.Lev := **S**

**Else**

Q^.Prav := **S**

**End**

**End**;

7.5.5. Удаление записи из дерева

Если некая вершина - конечная (“лист”) или из нее выходит только 1 ветвь, то для удаления записи у вершины-предшественника надо скорректировать ссылку (рис 7.25):

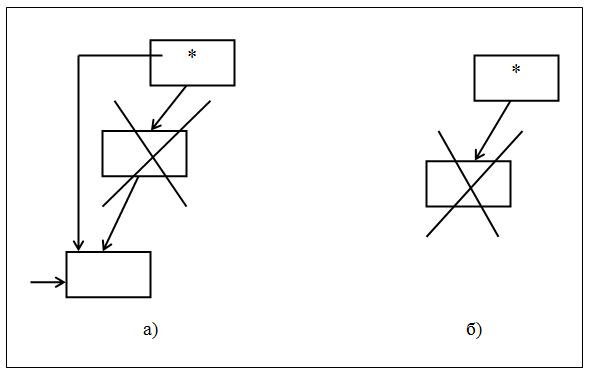
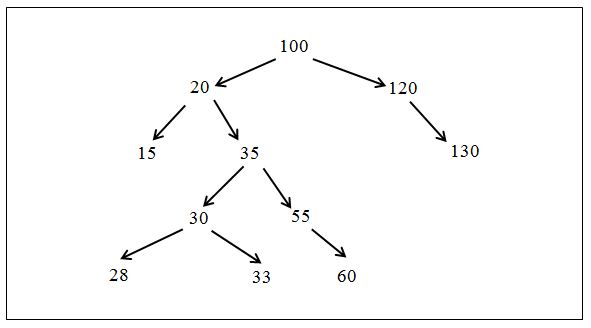


Рисунок 7.25 – Удаление записи из дерева. 2 вар-та:а) из удаляемой записи выходит 1 ветвь и **б)** удаляемая запись – «**лист**» Дер

В а) и б) в соответствующее поле **Lev /** **Prav** вершины-предшественника надо занести содержимое **Lev /** **Prav** **удаляемой** вершины. If удаляем “лист” (б) – б ссылка **Nil***.*

в) у удаляемой вершины – 2 ветви: надо найти звено, которое м вставить на место удаляемого. Это м б 2 вар-а:

**1) Самый правый элемент левого поддерева (**самый большой в левом поддереве от удаляемой вершины). Д/этого надо перейти в следующую вершину по левой ветви от удаляемой, потом –по правой пока очередная правая ссылка не будет = **Nil.**

**Пример 7.25.** Исключение из дерева (пр 7.21) звена с ключом **50**, используя **самый правый** эл-нт левого поддерева для замещения:

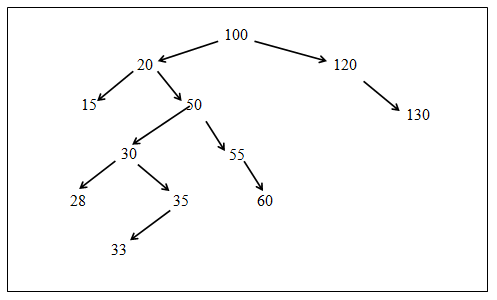


Рис 7.26 – Результат исключения звена с **ключом 50** (вариант 1)

2) **самый левый элемент правого поддерева (**самый маленький в правом от удаляемой вершины поддереве): надо перейти в следующую от удаляемой вершину по правой ветви, потом по левой ветви пока очередная левая ссылка не будет = ***Nil*** (т е наоборот (1)).

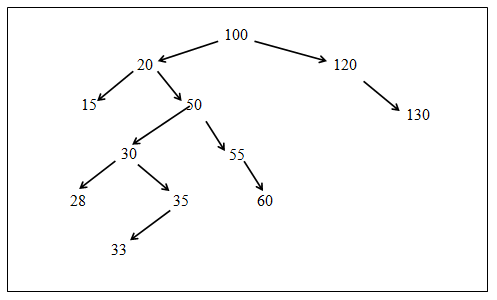
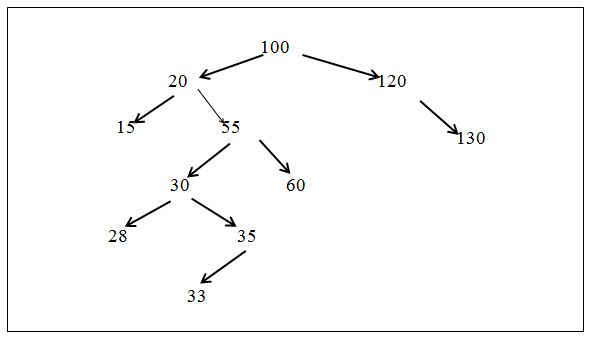
**Пример 7.26.** Исключение из дерева (пр 7.21) звена с ключом **50:** д/замещения использ самый левый эл-т правого поддерева. Рез-т исключ:  

Рис 7.27 – Результат исключения звена с ключом 50 (вариант 2)

Т о, при исключении из 2ичн Дер вершины с задан ключом есть 4 варианта:

1. звена с заданным ключом в дереве **нет**;
2. звено – «лист»;
3. звено с заданным ключом имеет **1 ветвь**;
4. звено с заданным ключом имеет **2 ветви**.

**Пример 7.27. Рекурсивная** процедура **Udder**исключения вершины с заданным ключом из дер. Процедура реализует вариант **1)** исключения в соответствии с Пр-ром 7.25: для замещения – **самый правый** эл-нт левого поддер (1). Параметры: ***K*** – искомый ключ, ***D*** – адрес **корня** дер. Объявление – по Пр-ру 11.2.

**Type**

Tekst = String; //<Тип\_значения\_записи>;

AdrТ = ^Tekst;

**AdrZv** = ^Zveno;

Zveno = **Record**

Kl: Integer; //1

Lev, Prav: AdrZv; //2, 3

Adr: AdrT; //4

**End**;

//--------- процедура Udder------------------------

**Procedure** Udder (**Var** D: **AdrZv**; K: Integer); //D – **адрес** корня дер

**Var**

Q: **AdrZv**; // в Q – адрес удаляемого звена

//----- процедура Ud - переход в **левое** д/удаляемой вершины поддер ------

**Procedure** Ud (**Var** R: **AdrZv**); {При первом вызове за фактич передается

**адрес левой** вершины после удаляемой}

**Begin**

**If** R^.Prav = Nil **Then** {Анализир-ая верш-а - самая правая в лев поддер 35}

**Begin**

{6} Q^.Kl := R^.Kl; {Занесен в поле Кl удаляемого звена ключ из

замещающего звена (в Q – **адрес** удаляемого звена)}

{7} Q^.Adr := R^.Adr; {Занесение в поле Adr удаляемого звена ссылки на

**текст** записи из замещающего звена}

{8} Q := R; {Занесение в Q адреса замещающего звена}

{9} R := Q^.Lev {Занесен в поле Prav звена- предшественника

замещающему содержимого поля Lev из замещающего звена}

**End**;

**Else** {Анализир-ая верш – не самая правая в левом поддер – *вар 2)?*}

{11} Ud (R^.Prav) {Рекурсивный вызов проц-ры Ud д/перехода к следующ

**правой** вершине **левого** поддер}

**End**; //==Конец процедуры Ud ====

**Begin** {========= Тело процедуры Udder ==========}

**If** D = Nil **Then** {Первый случай удаления}//D – **адрес** корня дер

Writeln ('Нет звена с заданным ключом в дер')

**Else // *K*** – искомый ключ

**If K** < D^.Kl **Then** {Ключ в анализир-ой вершине > заданного}

{2} Udder (D^.**Lev**, K) {Рекурсивн вызов проц-ры Udder д/перехода к

вершине **слева**-внизу}

**Else** {Ключ в анализируемой вершине <= заданному}

**If** K > D^.Kl **Then** {Ключ в анализир-ой вершине < заданного}

{3} Udder (D^.**Prav**, K) {Рекурсивн вызов проц-ры Udder д/перехода к

вершине **справа**-внизу}

**Else** {Звено с заданным ключом найдено!}

**Begin** // в Q – адрес удаляемого звена

{1} Q := D; {Занесен в Q адреса удаляемого звена (он взят

из поля Lev или Prav звена-предшественника)}

**If** Q^.Prav = Nil **Then** {2й случай удаления}

{4} D := Q^.Lev {Занесен в поле Lev/ Prav звена-предшеств-ка ***20***

удаляемому ссылки на зв, следующее за удаляемым ***55***}

**Else**

**If** Q^.Lev = Nil **Then** {2й случай удаления – *справа-налево?*}

{5} D:= Q^.Prav {Занесен в поле Lev/ Prav звена-предшест-ка ***20***

удаляемому ссылки на зв, следующее за удаляемым ***55***}

{10}**Else** {3й случай удаления}

Ud(Q^.Lev) {Вызов процедуры **Ud**; переход в **левое** д/удаляемой

вершины поддер}

**End**;

**End**; //=== Конец процедуры Udder ====

Параметр **D** оформлен как параметр-переменная процедуры **Udder**. При рекурсивном вызове процедуры **Udder** как соответствующий **D** фактический параметр используется поле **Lev** (см. опер {2}) или **Prav** ({3}) того звена, от которого перешли к текущему звену (т е звена-предшественника). Поэтому опер-ы {4} и {5} обеспечивают занесение в поле **Lev /Prav** звена-предшественника **ссылки** на звено, следующее за удаляемым, из удаляемого звена.

Аналогичны пояснения и для переменной ***R*** – пар-р-перемен процедуры ***Ud*.**

Процедура ***Ud*** - для поиска подходящей замещающей вершины, для занесения значений из полей замещающего звена в поля удаляемого звена и для удаления замещающего звена со своего предыдущего места.

Удаляется звено с ключом 50. Замещающее звено – 35.

№№ в {} == №№ операторов. Исходное значение ***D*** – адрес корня дерева. По оператору {2} идет переход к вершине 20, по {3} – к 50, по {10} – к 30, по {11} – к 35.

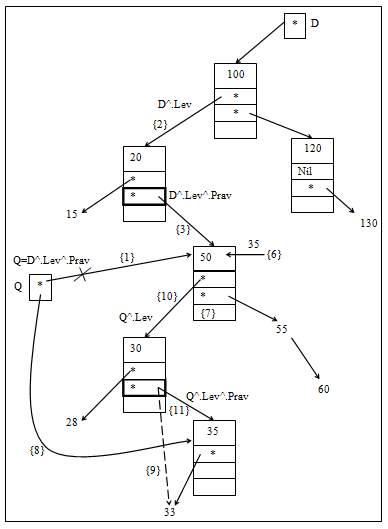


Рис 7.28 - Схематические пояснения к примеру 7.27

**D^.Lev, D^.Lev^.Prav** соответствуют значению **фактич**еского параметра, передаваемому на вход формального параметра **D** на текущем шаге выполнения программы; **Q^.Lev, Q^.Lev^.Prav** – значению фактического параметра, передаваемому на вход формального параметра ***R*** на текущем шаге выполнения программы.

Выделенные поля звеньев дерева - значения фактического параметра (**D^.Lev^.Prav*),*** соответствующего формальному параметру ***D*** для удаляемого звена, и фактического параметра (**Q^.Lev^.Prav***),* соответствующего формальному параметру ***R*** для замещающего звена.

Для физического удаления удаляемого звена надо вместо оператора {4}**(D := Q^.Lev)** в процедуре использовать **составной оператор**

**Begin**

**D := Q^.Lev;**

**Dispose (Q)**

**End;**

а вместо оператора {5} (**D := Q^.Prav)** – оператор

**Begin**

**D := Q^.Prav;**

**Dispose(Q)**

**End;**

Для физического удаления замещающего звена после оператора {9} в процедуре нужно добавить оператор **Dispose (Q*).***

## Раздел 8. Оверлеи

8.1. Общие сведения

После компиляции программа может оказаться слишком большая: при запуске программы выдается сообщение типа «Не хватает памяти». 1 из выходов – оверлейная технология составления программ.

**Модуль Overlay** – средство для **разбиения проги** на отдельные части. Суммарный объем этих частей может значительно превышать объем доступной памяти компьютера.

Оверлейная программа состоит из **2 частей:** файл с расширением 1). **ехе** и 2). **ovr**. **Ехе**-файл содержит постоянную часть программы. **Оvr**-файл хранит части программы, которые подгружаются в память по мере (по морю ☺) необходимости. При этом в памяти находятся лишь необходимые в данный момент процедуры и функции. Затем они перекрываются другими оверлейными процедурами и функциями. Оверлейные части программы по очереди используют 1 и ту же **область памяти** - **оверлейный буфер.**

Действия по загрузке / выгрузке оверлеев из памяти выполняются автоматически **администратором оверлеев***.*

8.2. Правила оформления оверлейных программ

В Паскаль оверлеи м создавать только на уровне **модулей**, т о, **миним**-ая оверлейная единица - модуль **Unit**. Прога д отвечать **треб-ям**:

1. Все оверлейн модули д б **от**компилированы в состоянии ***{$O+}*. (О-О-О-о-о-оverlay) Опция *{$O+}*** – это **глоб**альная опция «Генерация оверлейного кода» компилятора. По умолчанию она **выкл**ючена – ***{$O-}****.* Использование опции эквивалентно установке пункта меню

**Options / Compiler / Overlays Allowed**.

Откомпилированные модули в сост ***{$O+}*,** м б оверлейными, но не обязат-о.

1. Все подпроги (вызывающие оверлейные подпроги прямо / косвенно), д б **от**компилированы в состоянии **{$F+}.**

**Опция *{$F+}*** – это **локальн**ая опция компилятора «Дальний (межсегментный) вызов процедур и функций (***Far***)». По умолчанию данная опция **выкл**ючена – ***{$F-}****,* что соответствует ее значению «Ближний (внутрисегментный) вызов процедур и функций (***Near***)».

Для удовлетворения данных требований в начале каждого оверлейного модуля устанавливаются опции **{$F+, $O+},** а в начало всех других модулей и главной программы помещается опция **{$F+}.**

+ в основной программе в предложении использования **Uses**надо подключить стандартный модуль **Overlay**и все используемые оверлейные модули. Модуль **Overlay**д б записан перед именами оверлейных модулей.

Затем в основной программе надо указать, какие модули из перечисленных в предложении **Uses** будут оверлейными. Для этого используется локальная директива компилятора ***{$O <Имя\_модуля>}*** – «**Имя оверлейного модуля**». Она опред-ет при компиляции программы, какой из модулей помещается в файл с расширением **ovr**, а не **ехе.**

Отличие **директивы** компилятора от **опции** компилятора: имеет **параметр –** <**Имя**\_файла>, <**Имя**\_модуля>, размеры (стека, области динамической памяти и т.п.). В **опциях** используют только ключ (+ / –).

Директиву ***{$O <Имя\_модуля>}*** в основной программе помещают сразу после предложения использования ***Uses*.**

**Пример 8.1.** Размещение в основной программе директив и опций.

**Program Main;**

**{$F+}**

**Uses Overlay, Unit1, Unit2;**

**{$O Unit1}**

**{$O Unit2}**

**...**

**<Текст\_программы>**

При компиляции этой программы (пример 8.1) модули ***Unit1, Unit2*** помещаются в файл ***<Имя>.*ovr**, все остальные части программы – в файл ***<Имя>.*ехе***.*  <Имя> у обоих файлов – одно и то же.

Программы с оверлеями не могут компилироваться в памяти и всегда должны создаваться на **диске**.

**В разделе инициализации** (if он есть) оверлейного модуля запрещено:

1. включать администратор оверлеев.
2. производить действия до включения администратора оверлеев.

Так как разделы инициализации выполняют до выполнения основной программы, то реализация этих ограничений – за счет введения д/п-го неоверлейного модуля. В его разделе инициализации надо включить **администратор оверлеев**. Данный модуль надо подключить в предложении **Uses**основной проги перед оверлейными модулями.

Нельзя организовывать как оверлейные стандартные модули (**Dos, Crt** и т.д.) и модули, содержащие обработку прерываний.

8.3. Инициализация работы оверлеев

В модуле **Overlay** определены процедуры (5): 1.**OvrInit, 2.OvrInitEMS**, 3.Ovr**SetBuf,** 4.Ovr**ClearBuf**, 5.Ovr**SetRetry** и функции (2):1.Ovr**GetBuf** и 2.Ovr**GetRetry.**

8.3.1. Включение администратора оверлеев (админа О) OvrInit

Перед первым обращением к подпрограмме (хранится в ***Ovr***-файле) в программе надо включить (- инициализировать) администратор оверлеев 1 раз процедурой **OvrInit.** Заголовок процедуры имеет вид:

**OvrInit (OvrFileName: String)**

Данная процедура определена в модуле **Overlay.** Процедура производит **инициализацию (- включение)** администратора оверлеев и открывает оверлейный файл (его имя определяет параметр **OvrFileName** и оно д бтаким же, как у файла, содержащего основную часть программы, но с расширением **ovr**). Вызова процедуры **OvrInit** достаточно, чтобы работать с оверлеями (если не оптимизировать работу администратора и не обрабатывать его ошибки).

8.3.2. Обработка ошибок администратора

В модуле **Overlay** определена переменная **OvrResult**типа ***Integer*.** В ней сохраняются коды завершения процедур и функций модуля ***Overlay*.** Определено **7** значений **OvrResult.** Для каждого из них предопределена константа (таблица ошибка!).

Таблица **Ошибка! Текст указанного стиля в документе отсутствует.**.1 – Коды завершения процедур и функций модуля Overlay

|  |  |  |
| --- | --- | --- |
| Константа | Знач-ие | Описание |
| **OvrOk** | **0** | **Нормально**е завершение |
| **OvrError** | **-1** | **Ошибка** управления Overlay– попытка инициализации неоверлейного файла. |
| **OvrNotFound** | **-2** | Файл **.ovr** **не найден.** неправильное размещение оверлейного файла на диске. |
| **OvrNoMemory** | **-3** | Не хватает **памяти** для буфера оверлеев или крупный оверлейный модуль (- определяет размер оверлейного буфера) надо разбить на несколько независимых модулей. |
| **OvrIOError** | **-4** | Ошибка **ввода-вывода** оверлейного файла или его повреждение |
| **OvrNoEMSDriver** | **-5** | **Драйвер** EMS не установлен |
| **OvrNoEMSMemory** | **-6** | Недостаточно EMS памяти |

Если в фактическом параметре **OvrFileName (**проц-ры **OvrInit)** не указан полный путь, то файл ищется в текущем каталоге, затем в каталоге с **ехе**-файлом и в каталогах, указанных в системной переменной MS-DOS Path. Если в этих каталогах файл не найден, генерируется ошибка OvrNotFound и останов программы (ошибка выполнения 208 «Администратор оверлеев не установлен»).

Все эти ошибки м б при инициализации (включении) администратора.

Ошибки **OvrIOError, OvrMemory** м б при работе **процедуры** инициализации **OvrInitEMS.**

Переменная **OvrResult ==** стандартной функции **IOResult** кроме того, что **OvrResult** при обращении к ней не устанавливается в **0**.

Константы ошибок м использовать для анализа ошибок после инициализации администратора; например, с помощью операторов вида:

**If OvrResult = OvrNotFound *…***

или

**If OvrResult <> OvrOk Then**

**Case OvrResult of**

**OvrIOError:…**

**OvrNoEMSDriver:…**

и т.д.

**Пример 8.2.** Исходный текст проги хранится в файле **OvrDemo*.*pas***.*

**Program OvrDemo;** {Главная программа}

**{$F+}**

**Uses Overlay, Crt, Unit1, Unit2;**

**{$O Unit1}**

**{$O Unit2}**

**Begin**

**OvrInit (‘OvrDemo.ovr’);**

**If OvrResult <> 0 Then** {или OvrResult <> OvrOk}

**Begin**

**Writeln (‘**Ошибка инициализации:**’, OvrResult);**

**<Обработка ошибки инициализации>** {напр-р, **Halt** (1); где 1 – код выхода из проги (необязательный параметр, по умолчанию Halt (0))}

**End;**

**Repeat**

**Write1;**

**Write2;**

**Until KeyРressed**; {Ф-ция возвращает **True**, если была нажата

клавиша клавиатуры. Определена в CRT}

**End.** {=== Конец главной программы =======}

**{$O+, F+}**  {1-ый оверлейный сегмент **Unit1}**

**Unit Unit1;**

**Interface**

**Procedure Write1;**

**Implementation**

**Procedure Write1;**

**Begin**

**Writeln (‘**1-ый оверлей’**);**

**End;**

**End.** {============== Конец Unit1 ===============}

**{$O+, F+}** {2-ой оверлейный сегмент Unit2}

**Unit Unit2;**

**Interface**

**Procedure Write2;**

**Implementation**

**Uses Unit1;**

**Procedure Write2;**

**Begin**

**Writeln (‘**2-ой оверлей’**);**

**Writeln (**‘А теперь снова’**);**

**Write1;**

**Writeln (**‘Конец 2го оверлея’**);**

**End;**

**End. {===== Конец Unit2 ===}**

8.3.3. Размещение оверлейного файла в EMS-памяти

**EMS-память** – расширенная память.

Для размещения оверлейного файла в EMS-памяти используют процедуру **OvrInitEMS** (без параметров). Она проверяет, есть ли в компьютере **расширенная память** и установлена ли **программа-драйвер** этой памяти. Если да, то **ovr-**файл целиком загрузится в **EMS-**память и программа будет работать быстрее, чем хранение **ovr-**файла **на диске**.

Вызов процедуры **OvrInitEMS** не обязателен. При ее использовании заранее надо использовать процедуру **OvrInit*.*** Использование **OvrInitEMS**не отменяет необходимости в **буфере** оверлеев.

8.3.4. Управление оверлейным буфером

Размер оверлейного буфера назначается **автоматически (**размера, необходимого для загрузки наибольшего оверлейного модуля**).** Управляет размером функция **OvrGetBuf: LongInt, +** процедуры **OvrSetBuf (Size: LongInt), OvrClearBuf.**

Функция **OvrGetBuf** возвращает размер текущего буфера в байтах.

Если оверлейный модуль вызывает другой оверлейный модуль, то они оба одновременно активны, то есть должны находиться в буфере. Тогда размер буфера д б = сумме размеров этих оверлейных модулей.

Процедура **OvrSetBuf (Size: Longint)** регулирует размер буфера. Она должна вызываться после процедур **OvrInit и OvrInitEMS**. Параметр **Size** задает требуемую величину буфера в байтах.

Дополнительная память для буфера выделяется из динамической памяти. Процедура **OvrSetBuf** требует **пустую** динамическую память. Если же динамические переменные уже были распределены с помощью процедур **New** или **GetMem**, то будет сообщение об ошибке (возможные ошибки **OvrNoMemory** и **OvrError**). Тогда надо определить min-ый размер динамической памяти директивой **{$M…}.**

Если **OvrSetBuf** приводит к ошибке, программа не останавливается, а размер буфера не изменяется.

Удобно менять значение буфера относит-но текущего значения. Напр-р:

**OvrSetBuf (OvrGetBuf + 2048);**

Здесь размер буфера увеличился на 2Кб.

**OvrClearBuf** - принудительная очистка буфера. Это может быть полезно, например, если выполняется фрагмент программы, не использующий оверлейных подпрограмм. Тогда этот участок памяти можно использовать для размещения динамических переменных.

**ФСЁ!!!**