22. Указатели и ссылочные переменные

22.1 Основные определения

Как известно, оперативная память представляет собой совокупность элементарных ячеек для хранения информации - байтов, каждый из которых имеет свой номер. Эти номера называются адресами, они позволяют обращаться к любому байту памяти.

Замечание: существует два способа доступа к какому-то значению:

- по имени (имя как синоним разыменованного указателя);
- по адресу.

```
Var
  i,j : integer;
  p : ^integer;
begin
  i := 1;
  j := 2;
  p := @i;
  p^ := 3; ≡ i := 3;
```

То, как задается адрес в программе, зависит от ОС (точнее, от модели памяти, используемой в ОС):

- в старой 16-битной MS DOS в реальном режиме работы процессора адрес ячейки памяти 20-разрядный. Адреса задавались совокупностью двух шестнадцатиричных слов - *сегмента* и *смещения*. ***Сегмент*** - это участок памяти, имеющий максимальную длину *65536* байт (*64* Кбайта) и начинающийся с физического адреса, кратного *16* (т.е. 0, 16, 32, 48 и т.д.). ***Смещение*** указывает, сколько байтов от начала сегмента следует пропустить, чтобы обратиться к нужному адресу. Каждому сегменту соответствует **непрерывная и отдельно адресуемая область памяти**. Сегменты могут следовать в памяти один за другим без промежутков, или с некоторым интервалом, или перекрывать друг друга.

- в 32-битных ОС (в защищенном режиме) реализована так называемая «плоская» модель памяти, где используется лишь 32-битное смещение (64--битовое в режиме x64)/

1

Определение: Указатели – это такие переменные, содержанием которых являются не значения какого либо типа, а **адреса других переменных**. С помощью указателей мы **даем адресам имена** (имена указателей) и можем работать с адресами как с обычными переменными. **Внутреннее представление** указателей (а конкретно - дальних указателей – FAR-указателей) в TP следующее: Каждый указатель обычно занимает 32 бита - два слова (типа word) (IP - 16 бит, EIP - 32 бита , RIP - 64 бита) и хранит двоичное число - адрес.

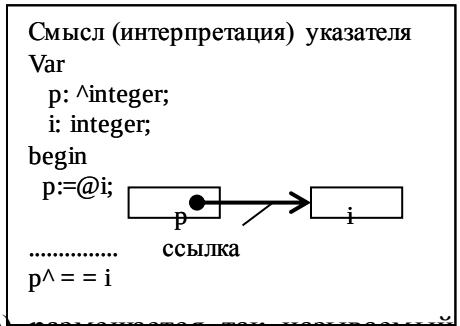
22.2 Описания(объявления) указателей в программе

В TP различают **типизированные** и **нетипизированные** указатели.

ссылочные переменные

(потому что их значением является ссылка, то есть адрес)

```
Var
 P : pointer;  {нетипизированный указатель}
 Z : ^ integer;  {типизированный указатель}
```



Справа от “крышки” (символ над цифрой 6 на клавиатуре) размещается так называемый базовый тип указателя, то есть тип значения в памяти (на тип значения некоей переменной в памяти), на которое (точнее, на адрес начала которого) может указывать (ссылаться) указатель (т.е. хранить ее адрес). Тип может быть стандартный или определяемый программистом (простой или составной). В обоих случаях объявление указателя означает выделение памяти (под него) на стадии компиляции в размере 4-х байт для размещения в этой области некоего адреса (значения указателя).

Значением нетипизированного указателя может быть адрес (ссылка) любого объекта любого типа. Значением типизированного указателя может быть адрес объекта лишь того типа, который был указан после символа ^ при объявлении указателя.

Замечание: До того момента, как переменной – указателю не присвоено конкретное значение, его значение непредсказуемо и нельзя пользоваться этим значением. **Неинициализованные** указатели («**висящие**» указатели) являются одним из наиболее частых источников **зависания** программ (обычно в момент разыменования висящих указателей).

Имеется предопределенная константа *nil* (в Си - NULL), которая используется для представления неопределенного значения указателя (указателя, которому еще не присваивалось никакого значения). Для того чтобы «отлавливать» висящие указатели существует следующий прием:

```
Const
  p1:^real=NIL;
  p:pointer=NIL;
  ……..
```

При таком объявлении неинициализированный указатель имеет значение NIL, что легко проверить:

```
if (p1 = NIL)
  then {ошибка - указатель висит, нельзя разыменовать}
  else {можно p1 разыменовать};
```

Замечания:

1)Нетипизированные указатели в отличии от типизированных указателей **нельзя разыменовывать**. Эта операция (разыменования) заключается в обращении по тому адресу, который является содержимым переменной указателя. Обращение к значению в памяти через разыменование часто называют косвенной адресацией (в отличии от прямой – когда обращается по имени).

2) Нетипизированные указатели используются для выделения памяти под объекты, **размер которых неизвестен во время компиляции.** При этом в каждом конкретном случае использования такого нетипизированного указателя прибегают к помощи операции приведения типа (к тому типу, который нужен в данный момент). Типизированные указатели используются для выделения памяти под объекты фиксированного размера, определяемого их типом (sizeof(тип)).

3) Нетипизированные указатели **совместимы с любым типом указателя** (при использовании их справа в операторе присваивания).

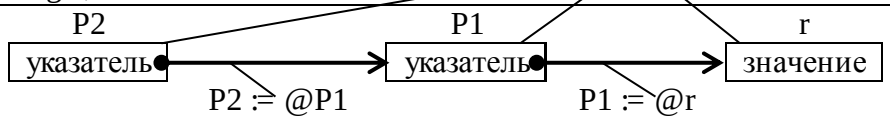
4) В Паскале, к сожалению, с использованием типизированных указателей непросто объявить указатель на указатель.


```
Type
 T = ^integer;
Var
 P1 : T;
 P2 : ^T;
 r : integer;
```

нельзя

```
Var P2: ^^integer;
```

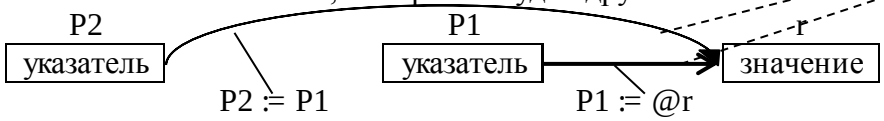
картина в памяти



То же самое в С:

```
int r;
int *p1; p1 = &r;
int **p2; p2 = &p1;
**p2 = 1;  ≡  *r = 1; ≡  *p1 = 1;
```

NB: если написать P2 := P1, то картина будет другая



Теперь P1 и P2 – «двойственные» указатели (есть еще неинициализированные или висящие указатели и потерянные ссылки (если есть некая область памяти, но на нее потеряна (испорчена) ссылка)

NB: можно было в данном случае использовать **нетипизированный** указатель, т.к. ему должно быть не важно на что указывать, но при (для) разыменовании его все равно нужен тип.

В С++ кроме указателей есть т.н. **ссылки**. Можно их сравнить:

Указатели (снижают безопасность кода)

1. Есть в С++, но нет в Java и на код тоже
2. Могут указывать на произвольную ячейку памяти (может быть преобразован в число – физический адрес)
3. Допустима адресная (целочисленная) арифметика над указателями (а в Паскале - нельзя)
4. В Паскале иногда указателями называют нетипизированные указатели, а ссылками (ссылочными переменными) – типизированные указатели. В этом случае адресная арифметика для ссылок задаёт перемещение по элементам типа ссылки, а для указателей (нетипизированных) – перемещение по байтам

```
var   p1, p2: pointer;
begin
  p1 := @r; p2 := @p1;
  integer(p2^^) := 1;
```

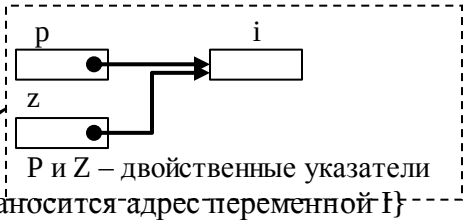
Ссылки (повышают безопасность кода)

1. Есть в Java (и в С++ --- *int& i=j;* *или int& i; i = j;*)
2. Ссылка может указывать только на объект, но не на произвольную ячейку в памяти (не м.б. преобразована в число)
3. Над ссылками (это константы) недопустима целочисленная (адресная) арифметика
4. Ссылку можно рассматривать как константный **разыменованный** указатель (синоним) на объект
5. Под ссылку, в отличие от указателя может не выделяться память (это только синоним)

22.3 Операции с указателями

а) Операция **присваивания** (значения адреса или значения другой переменной-указателя)

```
var
 p : pointer;
 i  : integer;
 z : ^ integer;
begin
  p := @i; {в переменную P заносится адрес переменной I}
        или
  p := ADDR(i);  {в переменную P заносится адрес переменной I}
  z := p; {Теперь Z указывает туда же, что и P     - теперь p и z - это двойственные указатели}
end.
```



Нетипизированному указателю можно присвоить значение как нетипизированного, так и типизированного указателя. Присваивание значения типизированному указателю должно производиться **с учетом его типа,** т.е. типизированному указателю можно присвоить либо значение нетипизированого указателя, либо типизированного того же типа, либо адрес переменной того же типа:

Пример 1

```
Var
  i    : integer;
  r    : real;
  p1 : ^real;
  p2 : ^integer;
begin
  P1:=@r;  так можно всегда
  P2:=@i;  так можно всегда
  P1:=P2;  так нельзя никогда
end.
```

Так **можно**:

```
Var
  p: pointer;
begin
  p := @i; или p := p2;
  p1 := p
```

Пример 2

```
Var
  p1:^string;
  p2:^byte;
  p  :pointer;
  b  :byte;
  s:string;
begin
  p2 := @b;
  p1 := p2;  так нельзя (да и не нужно)
  s := 'abcd';
  b:=2;
  writeln('p2^=',p2^);  - будет выведено 2
  p1:=@s;
  p:=p1;        так можно
  p2:=p;
  writeln('p2^=', p2^); - будет выведено 4, т.к. p2 (указатель на байт) теперь адресует младший байт S (s[0]), где хранится длина строки S
end.
```

Пример 3

```
Var
a : array[1..5] of byte;
p : ^byte;
begin
 p := @a[3]; - значение p будет адрес 3-го элемента массива a
 a[3] := 3;  ≡  p^ := 3;
```

NB:

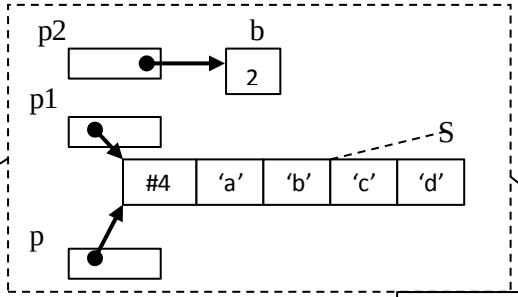
В Паскале нельзя:

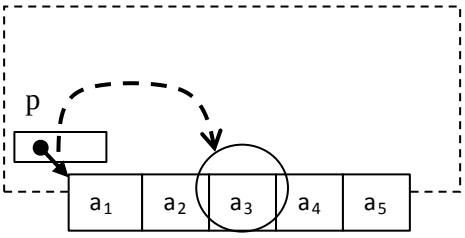
```
(p+2)^ := 3;
```

В Си можно:

```
*(p+2) = 3; ≡ a[3] = 3;
```

***NB**: p* в Си должен быть указателем на массив (а не на байт). Иначе перемещаться будем по байтам, а не эл-там массива





нельзя:
char(p1^) := #3; тип (p1^) - строка
надо:
char(p1^[0]) := #2; //изменили длину строки

можно:
p := p2; ≡ p := &b;
byte(p^) := 2; ≡ b := 3;
p := &s; ≡ p := p1;
char(p^) := #2; //изменили длину строки

б) **Сравнение**:

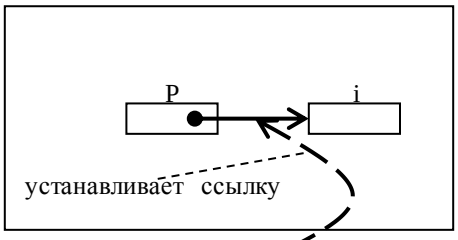
Можно сравнивать два указателя на равенство и неравенство (нельзя сравнивать указатели между собой на предмет того, какой из них больше другого).

в) Операция **разыменования** (разадресации).

С использованием этой операции имеется две возможности для обращения к значению переменной:

1) Использование идентификатора переменной (прямое обращение);
2) Использование адреса переменной и операции разыменования (косвенное обращение).

```
Var
  I:integer;
  P:^integer;
begin
  I := 25;
  P^ := 25;
end.
```



Если перед последней операцией забыть использовать P := @I, то получим трудноуловимую ошибку на стадии выполнения.

Замечания: 1. результат операции разыменования имеет тип, равный базовому типу переменной-указателя.

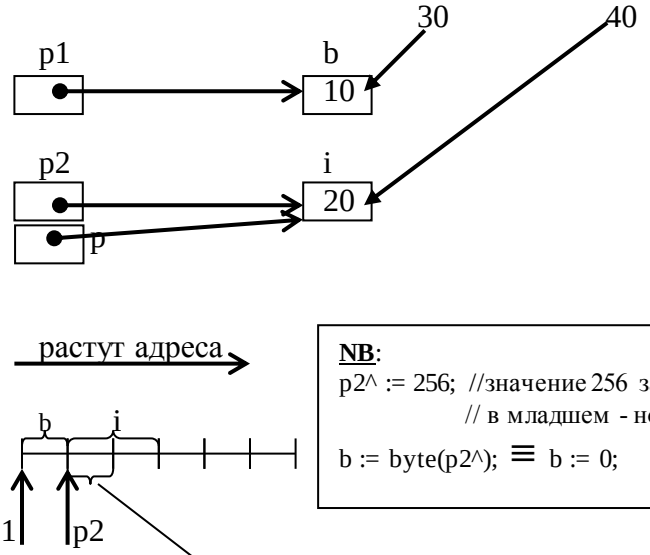
2. операция разыменования может быть использована в любом месте программы, где допускается нахождение простой (обычной) переменной - P^ := P^ + 5

3. перед выполнением разыменования указателю должно быть присвоено конкретное значение адреса.

4. при разыменовании важно кроме адреса еще знать, на какую «ширину» применяется эта операция, то есть сколько байт, находящихся по определенному адресу, надо интерпретировать как значение определенного (базового) типа. Информация о «ширине» значения каждого типа содержится в его типе. Поэтому прямо (без приведения типа) разыменовывать можно лишь типизированные указатели, нетипизированные – только с использованием приведения типа (типа операции разыменования).

Пример:

```
Var
  B : byte;
  I : integer;
  P1 : ^byte;
  P2 : ^integer;
  p : pointer;
begin
  B := 10;
  I := 20;
  p := &I;
  P1 := @B; ≡ addr(B)
  P2 := @I;
  P1^ := 30; ≡ B := 30;
  P2^ := 256; ≡ I := 256;
  Byte(P2^) := 5; ----------------------------------- 5 запишется строго в младший байт переменной I
end.                  ширина 2 байта
```



NB:
p2^ := 256; //значение 256 занимает 2 байта
// в младшем - ноль, в старшем 1
b := byte(p2^); ≡ b := 0;

5. Указание Byte(P2^) заставляет при разыменовании задействовать лишь один байт (поэтому b будет равно 256 + 5).

6. Замечание: если некая переменная P является указателем, то **следует различать записи**

1) P :=25; -----прямая адресация – обращаемся к содержимому P

2) P^:=25; -----косвенная адресация – обращаемся не к содержимому P, а к содержимому той области, адрес которой хранится в P.

3) тип(P^) := 25; --- ширина разыменования P определяется типом <тип>.

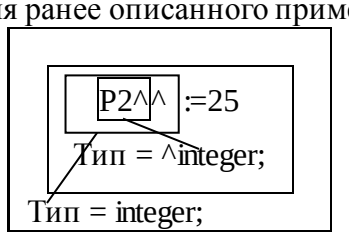
7. Если имеет место следующее объявление:

```
Var
  p: pointer;
```

то присваивание P^ :=25; недопустимо по 2 причинам:

1) нетипизированный указатель разыменовывать нельзя, Однако это возможно с использованием операции приведения: INTEGER(p^) :=25;
2) неизвестно по какому адресу будет записано значение 25, т.к. переменная P еще не получила никакого значения.

8. Если имеется указатель на указатель, то может быть многократное разыменование. Так для ранее описанного примера (в конце п.2.2.2) можно записать:



^^^^^ - можно писать справа от имени указателя, но нельзя слева от типа:

```
Var
  p: ^^^integer; ----- так нельзя
```

NB: при возникновении проблем с разыменованием, часто двойное разыменование записывают в виде (P2^)^.

≡ ^char

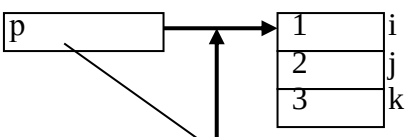
г) Операции **адресной арифметики** (уменьшение/увеличение указателя).

Замечание: В языке Си такие операции разрешены, а в Паскале **запрещены** (но для типа **Pchar разрешены**). Стандартными средствами их делать нельзя.

Пример: имеются переменные i, j, k, и указатель p.

Память под переменные выделяется следующим образом: по младшему адресу - i, за ней - j, за j - k.

```
Var
 I,
 J,
 K : integer;
 P : ^integer;
Begin
 I := 1;
 J := 2;
 k := 3;
 p := @i;
```



Возрастание адресов переменных

Зная адрес i и то, что за i следом в памяти идут j и k, хотелось бы, изменяя его (адрес i), получать доступ к значениям переменных j и k, но это возможно далеко не всегда

В переменной P записан адрес переменной i..

Задача: Зная что переменные находятся подряд и адрес переменной j больше адреса переменной i на 2, как можно через адрес переменной i обратиться к адресу переменной j.

Решение: P := P+2; ----- но прямо так на Паскале сделать нельзя (только для pchar).

Замечание: адресная арифметика имеет смысл только применительно к массивам, т.к. p:=p+2; перемещает указатель не на 2 байта, а на 2 элемента массива.

22.4 Понятие динамической переменной

Запись

```
 Var
    x:^real;
```

означает, что x - это **статическая** переменная-указатель, под которую статически (на этапе компиляции) выделена память (6 байт). Однако именно с помощью статических указателей можно организовывать динамические переменные.

Динамические переменные – это те переменные, место под которые выделяется не в стеке или сегменте данных, а в динамической памяти. Динамическая память - это вся свободная память компьютера за вычетом того, что непосредственно выделено программе (и программой) (под стек, данные, код). Другое название динамической памяти – куча (**heap** в Паскале и С). Вы можете менять в своей программе параметры работы с памятью:

по умолч. – 16 Кб по умолч. – 0 по умолч. - 640Кб (в реальном режиме)

{$m размер стека, минимальный размер кучи, максимальный размер кучи}.

для нашей программы

Необходимость в динамических переменных возникает в следующих 3-х случаях:

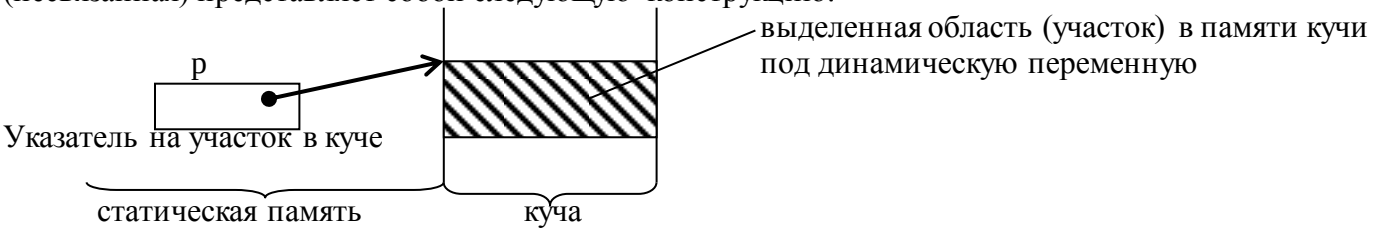
1). В программе должны использоваться переменные достаточно большого размера, но необходимость в их использовании возникает лишь **эпизодически**. Поэтому выделенную под эти переменные память желательно для экономии памяти периодически **освобождать** (отдавать в распоряжении системы).

2). Необходимо работать (выделять под них память) с переменными, **размер которых заранее** (до начала выполнения программы) **неизвестен** (становится известным только при выполнении).

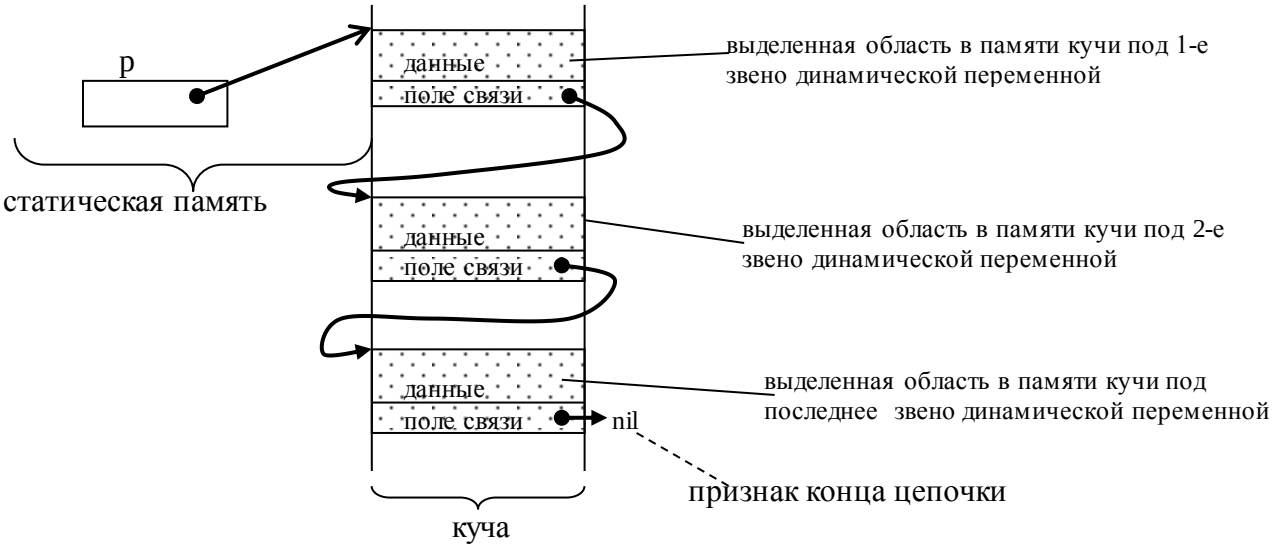
3). В программе необходимо использовать переменные, суммарный размер **которых превышает размер сегмента** (64 Кбайта).

Отличительными **особенностями динамических переменных** являются следующие:

1). Они не имеют имени, а имеют только участок (в куче) и указатель (статический) на этот участок. Начиная с этого адреса вы динамически выделяете и очищаете память. Т.е. динамическая переменная (несвязанная) представляет собой следующую конструкцию:



Связанные динамические переменные имеют другую (более сложную) структуру: в ней несколько выделенных областей (типа Запись) в куче связаны между собой через поле связи:



Связанные динамические переменные делятся:

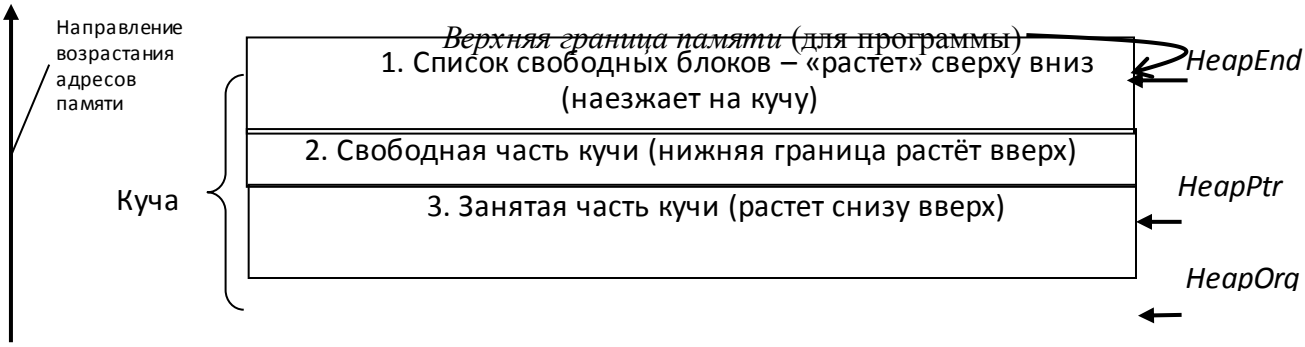
1) по направлению связи между звеньями:
   - от старого звена к новому (список);
   - от нового звена к старому (стек) ;
2) по числу связей звеньев с соседями (по которым можно ходить вперед и назад):
   - односвязные (линейные списки) ;
   - двухсвязные линейные списки;
   - многосвязные нелинейные (графы, деревья) ;
3) по числу соседей в каждую сторону
   - линейные (списки) (только один сосед справа и/или слева) ;
   - нелинейные (графы) ;
4) по характеру считывания информации:
   - с разрущающим считыванием (стек) ;
   - с неразрушающим считыванием (список).

2). Работать с динамической переменной (не зная ее имени, а зная только указатель), можно только с помощью ссылки (на область в куче) и операции разыменования (указателя на область в куче).

Замечание: Остается старое ограничение Паскаля – никакая переменная (в том числе и динамическая) не м.б. больше 64Кбайт.

22.5 Организация памяти кучи в программе на Турбо-Паскале

Замечание: стрелки (справа) отмечают начало каждого из областей.



HeapEnd, HeapOrg, HeapPtr – предопределенные переменные – указатели. Heapend отмечает верхнею границу кучи. Heapptr отмечает начало свободной части кучи (верхнюю границу занятой части кучи). Heaporg – отмечает нижнюю границу кучи (не изменяется при выполнении программы). Обычно, когда программа только начала выполняться, то начальное состояние кучи характеризуется следующим равенством

heapptr = = heaporg .

Heapptr обычно растет в верх при выделении памяти в кучи и соответственно обычно падает вниз при освобождении. Количество байт на которые heapptr совершает «прыжок» (вверх или вниз) определяется (как и в случае с операцией разыменования) размером (точнее, типом) области памяти, которая выделяется в куче или отдается (возвращается) назад в кучу:

```
Var
  p:^real;
begin
  new(p);----------«ширина» базового типа для указателя равна 6 байт, столько байт
                   теоретически и будет выделено
end.
```

Замечание: в силу особенностей работы менеджера памяти Турбо-Паскаля минимальный выделяемый размер памяти = 8 байт. При этом память в куче выделяется порциями, кратными 8 байт.

В той или иной степени вы можете управлять размером свободной области. Для этого можно испьзовать **директиву компилятора {$M p1, p2, p3},** где

p1, p2, p3 – размер в байтах нужных вам областей.

P1 – размер стека (по умолчанию 16 Кбайт).

P2 – минимальный размер кучи (по умолчанию 0).

P3 – макс. размер кучи для программы (по умолчанию 640Кбайт – максимум для реального режима).

P2 – задает размер кучи, меньше которого программа выполняться не может.

P3 – указывает, что куча размером больше чем p3, программе не нужна.

Если p2 =0, то ваша программа будет выполняться при любом размере кучи (или вовсе без нее).

Если p3=0, то ваша программа будет выполняться без кучи. Это бывает нужно, если вы запускаете в процессе вашей программы другие (дочерние) программы (процессы). При запуске из вашей программы другого внешнего процесса, кучу нужно освободить по следующей причине: если вы забудете это сделать, то всё, что выше вашей программы, будет отведено под кучу и не останется место для запуска дочернего процесса. Итак, при необходимости запуска внешнего процесса размер кучи (параметр p3 директивы {$M …}) нужно уменьшить, в крайнем случае, до 0.

22.6 Процедуры и функции выделения памяти.

Замечание 1: порядок работы с динамическими переменными:

А) объявление указателя (статического);

Б) выделение памяти в куче (инициализация указателя);

В) Использование памяти в куче (через разыменование указателя);

Г) освобождение памяти в куче .

Замечание 2: После каждого выделения памяти, неважно каким из возможных способов, в куче создается так называемая динамическая переменная, начало которой отмечено статическим указателем (в Delphi все объекты классов - динамические записи).

Выделение места в памяти обычно возможно двумя способами:

Первый способ реализуются с помощью процедуры new. При этом **явно не указывается**, сколько памяти в куче надо выделить. Этот размер определяется косвенно по базовому типу указателя, который передали процедуре New.

Заголовок (прототип) процедуры new имеет вид:

procedure new(var p:pointer);

1 способ - **Процедура new** имеет один параметр, каковым при вызове должен быть типизированный (или приведенный к нужному типу нетипизированный) указатель. Процедура new выполняет два действия:

- выделяет область памяти в куче размером, соответствующим тому базовому типу, который имеет (на который указывает) ссылочная переменная, переданная в качестве фактического параметра.
- адрес начала выделенной области памяти записывается в ссылочную переменную, переданную в качестве фактического параметра (графически это отображается так, что образуется стрелка от указателя к области памяти в куче).

2 способ – **процедура getmem()**. Ее заголовок имеет вид:

getmem(var p:pointer; size:word);

в С++ есть операция *new*

Отличается от процедуры new тем, что здесь ***явно*** указывается размер выделяемой области памяти. Процедура getmem имеет два параметра: ссылочную переменную и размер в байтах выделяемой области. В качестве указателя в getmem можно использовать как типизированный так и нетипизированный (без приведения типа) указатель.

Есть еще и 3 способ - **Функция new.** Единственным параметром функции является имя типа ссылки (имя типа типизированного указателя - надо в секции type объявить), в соответствии с размером которого и выделяется в куче нужная область памяти. Функция new выполняет два следующих действия:

- выделение в куче области памяти, размером, определяемым типом.
- передача адреса начала этой области памяти в точку вызова.

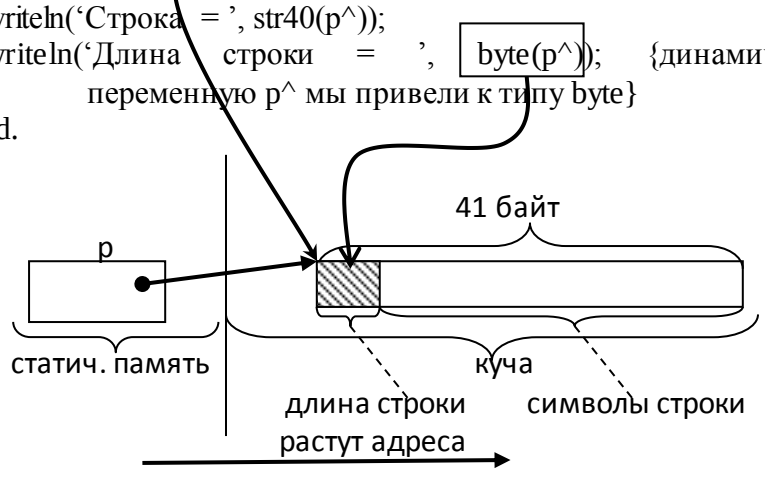
Пример 1

```
Type
  t1 = ^ integer;
Var
  p1 : t1;
  p2 : pointer;
begin
  {использование процедуры new}
  New(p1);
  New(p2); {так нельзя}
  New(t1); {так нельзя}
  {использование функции new}
  p1 := new(t1);
  p2 := new(t1);
  p2 := new(p1); {так нельзя}
  p1 := new(p2);{так нельзя}
  {использование процедуры getmem}
  Getmem(p1, 2);  Getmem(p2, 2);
  {разыменование}
  p1^:=25;
  p2^:=25; {так нельзя}
  {разыменование с приведением типа}
  integer(p2^):=25;
end.
```

Пример 2

```
Type
  str40=string[40];
Var
  p:pointer
begin
  getmem(p, 41); {теперь p указывает на начало блока из 41
                  байта}
  str40(p^):='Привет.'; {динамич. переменную p^ мы
                  привели к типу str40}
  writeln('Строка = ', str40(p^));
  writeln('Длина строки = ', byte(p^)); {динамич.
        переменную p^ мы привели к типу byte}
end.
```

равно sizeof(str40)



*22.7 Процедуры **освобождения памяти***

В Паскале имеются следующие процедуры освобождения памяти:

dispose(var p:pointer);
freemem(var p:pointer; size:word);

Они возвращают память в кучу (освобождают занимаемую динамической переменной память).

В продолжение рассмотренного выше примера1 можно записать:

Dispose(p1); ⟷ new(p1);
Freemem(p2, 2); ⟷ getmem(p2, 2);

*это **должно совпадать** (чтобы не делать ошибки лучше написать не сам размер, а использовать спец. функцию sizeof(тип))*

Двойные стрелки означают, что каждая из процедур, освобождающих память, должна быть парна соответствующей процедуре выделения памяти.

Вызов процедуры dispose имеет вид: dispose(p1); где p1 - указатель на динамическую переменную.

Dispose выполняет следующие два действия:

1) выделенная ранее область памяти (на нее указывает p1) возвращается назад в кучу (размер свободной части кучи увеличивается), а heapptr (см. рис. выше) обычно опускается вниз.

Замечание: Освободившаяся область может попасть или в свободную часть кучи или в т.н. список свободных блоков.

2) состояние указателя (у нас - p1), который отмечал занятую и освобождаемую область памяти, становится неопределенным и пользоваться им для разадресации (разыменования) нельзя.

Вызов другой процедуры освобождения памяти имеет вид:

freemem(p2,2)

где p2 – указатель на динамическую переменную, а второй параметр - размер памяти, который вы хотите освободить.

Требование: размер освободившейся области памяти должен быть равен размеру выделенной ранее области памяти (с помощью getmem).

22.8 Анализ состояния кучи

Этот анализ выполняется с использованием двух функций без аргументов: memavail и maxavail.

1) **memavail** – возвращает в точку вызова общий размер свободной памяти кучи (в байтах).

В общий размер свободной памяти кучи включается сумма размеров всех свободных участков, которые расположены как выше так и ниже heapptr. Пользоваться значением, возвращаемым memavail, надо достаточно осторожно, потому что память из кучи по запросам должна и будет выделяться непрерывными кусками. Т.е. при раздробленной на куски куче может не оказаться непрерывного свободного участка необходимого нам размера (и в результате память по нашему запросу не будет выделена).

Замечание: фрагментация кучи чаще всего происходит в случаях (по причинам): а) память в куче всегда выделяется кусками, кратными 8 байтам; б) если участки памяти кучи освобождаются не в порядке, обратном выделению, например, когда с помощью dispose освобождаются участки памяти, между которыми и heapptr (heapptr будет вверху) остаются занятые участки (получается чересполосица в памяти - см. далее).

2) **maxavail** –возвращает размер наибольшего непрерывного участка в куче в точку вызова. Этот участок может располагаться как ниже, так и выше heapptr. В первом случае информация об этом блоке находиться в списке свободных блоков.

С *помощью анализа значений*, возвращаемых этими двумя функциями, можно сделать заключение о состоянии кучи: есть ли память в куче, насколько она фрагментирована и можно ли рассчитывать на то, что по нашему запросу в куче будет выделена память.

Пример

```
Type
        t = ^ тип;
Var
        p : t;
begin
        if (maxavail >= sizeof(тип))
                then { памяти в куче достаточно}
                   p := new(t);    или  new(p);
                else { памяти почему-то недостаточно и выделить ее нельзя}
                   if (memavail >= sizeof(тип))
                        then { имеем (maxavail < sizeof(тип)) and (memavail >= sizeof(тип))}
                           writeln('Память есть, но выделить ее нельзя, т.к. куча фрагментирована')
                        else { имеем (maxavail < sizeof(тип)) and (memavail < sizeof(тип))}
                           writeln('Память выделить нельзя, т.к. в куче действительно не хватает места');
end.
```

размер выделяемой памяти

Кроме контроля состояния кучи с помощью функции ***memavail*** можно следить за тем, насколько корректно та или иная программа (подпрограмма) использует кучу, например, не забыла ли вызванная на выполнение программа (подпрограмма) освободить (перед завершением) временно выделенную память в куче (нет ли утечек памяти):

```
Var
 old_mem,   new_mem : longint;
begin
 old_mem := memavail;                {свободная память в начале работы}
```

Вызовы подпрограмм, Занимающих и освобождающих память в куче.

```
 new_mem := memavail;                {свободная память после работы  вызванной подпрограммы}
 if (old_mem <> new_mem) //или if (old_mem > new_mem)
  then
    Что-то забыли освободить
```


Замечания:

1) такой приём следует использовать на стадии отладки программы
2) желательно анализируемый участок сократить до одной подпрограммы

Как говорилось выше одна из причин фрагментации памяти кучи в том, что в ходе освобождения памяти в куче между занятыми участками оказываются незанятые (черезполосица).

Пример:

```
Var
 P1,P2,P3,P4,P5 : ^integer;
 p: pointer;
begin
 New(P1); P1^ := 1;  mark(p);
 New(P2); P2^ := 2; { }
 New(P3); P3^ := 3;
 New(P4); P4^ := 4;
 New(P5); P5^ := 5; {release(p);}
```

дырка в куче (свободный участок среди занятых)

После всех new получим:

Рисунок соответствует точке выполнения программы после p5^:=5. Если после этого начинается выполнятся следующая последовательность действий:

```
Dispose(p5);
Dispose(p4);
Dispose(p3); и т.д.,
```

то heapptr шаг за шагом будет опускаться вниз. Это означает, что освобожденные области памяти будут попадать в свободную часть кучи (свободная часть кучи на рисунке выше будет расти вниз).

Если же вместо этого с той же точки начать выполнять:

```
Dispose(p2);
Dispose(p3);
Dispose(p4);
```

то Heapptr своего значения не изменит. При этом свободная память в куче будет разбита на две части:

1) начиная с Heapptr и выше
2) ниже Heapptr (то, что отмечено фигурной скобкой)

Информация о свободных блоках, расположенных ниже Heapptr, хранится в упомянутом выше списке свободных блоков.

22.9 Ошибки при работе с указателями

22.9.1. Висящие указатели

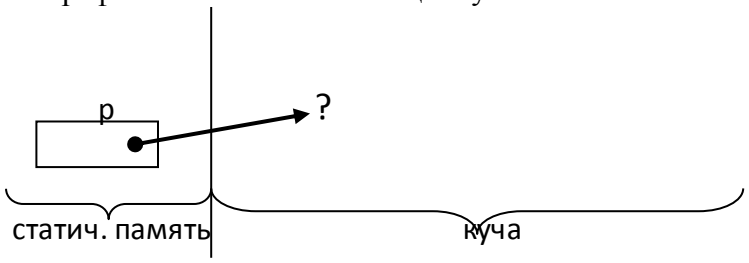
Это указатели которые:

а) ещё не имеют значения (не инициализированы)

б) потеряли значение (например из-за Dispose или FreeMem)

в) ссылаются на объект в статической (автоматической) памяти, который перестал существовать (время жизни закончилось) (не обязательно из-за работы Dispose).
или пропал из области видимости

Графически наличие висящего указателя в памяти можно показать следующим образом:



Примеры появления в программе висящих указателей:

а)

```
Var
  P : ^integer;
begin
…
  P^ := 25;
```

Неизвестно куда значение 25 попадёт, поэтому перед разыменованием надо инициализировать указатель, например, так: New(P);

б)

```
Var
  p : ^integer;
begin
  .....
  New(p);
…
  Dispose(p);
…
  P^ := 25;
```

в)
```
Var
  p : ^integer;
Procedure P1;
  Var
    i: integer;  {память под переменную I выделена только на время выполнения P1}
  begin
    p := @i; //p - глобальная переменная, а  i - локальная
    i := 111;
    Writeln(p^);   ---------- будет выведено 111
  end;
Procedure P2;
  begin
    Writeln(p^);   ---------- указатель P здесь висит (локальная переменная I больше не существует)
  end;
begin
  P1;
  P2;
end.
```

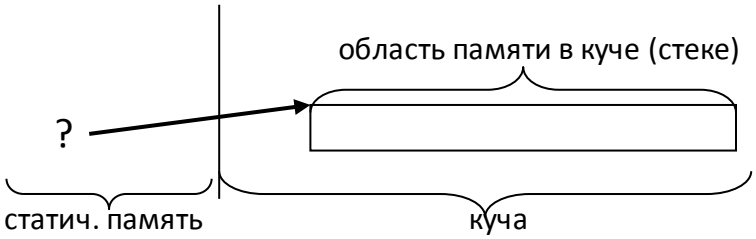
Глобальная переменная-указатель *P* в процедуре P1 установлена на локальную переменную *I* (ей выделено место в стеке). После выхода из процедуры P1 переменная *I* перестаёт существовать (фрейм стека процедуры освободился) и указатель *P*, как говорят, зависает (ссылается на область, которая уже перестала существовать).

22.9.2. Потерянные ссылки

сирота

Они соответствуют ситуации, обратной ранее демонстрируемой, когда существует область памяти в куче (или стеке), но на неё нет указателя (потерян)(orphan).

Графически наличие потерянной ссылки в памяти можно показать следующим образом:



Пример:

```
Type
  t = ^ тип;
Procedure P1;
  Var
    p: t;  //p - локальная переменная
  begin
    new(p);      или  p := new(t);
  end; //перед выходом память не освободили
begin
  writeln('Свободная  память - ' , memavail  , ' байт');
  P1; ---------- вызов процедуры P1
  writeln('Свободная  память - ' , memavail  , ' байт'); ---------- будет выведено другое (меньшее) число
end.
```

Переменная *P*(указатель) является локальной (ей выделено место в стеке). Её значение теряется после выхода из процедуры (фрейм стека для процедуры P1 освобождается), при этом (после выхода из P1) теряется возможность доступа к выделенной в куче области памяти (ее адрес хранился в *P*), хотя сама область остаётся существовать (остается занятой). И мы **не сможем её освободить с помощью Dispose** (в ЯП типа C# и Lava в таком случае *сборщик мусора* автоматически постарается освободить эту память).

22.10 Несвязанные динамические структурированные переменные (массивы и записи)

22.10.1 Динамические записи

До сих пор мы рассматривали простые (неструктурированные) динамические переменные. При записи обращения к ним (с помощью операции разыменования) больших проблем (с форматом этого обращения) не возникало. Для тех динамических переменных, названия которых вынесены в заголовок, форма записи обращения к ним имеет более экзотический вид, чем для ранее рассмотренных. Разберём пример:

```
Type
  t=record
       A:longint;
       B:byte;
       C:Char;
    End;
Var
  V1:t; -----статическая переменная-запись
  V2:^t;-----типизированный указатель
Begin
  V1.a:=1;
  V1.b:=2;
  V1.c:='3';
```

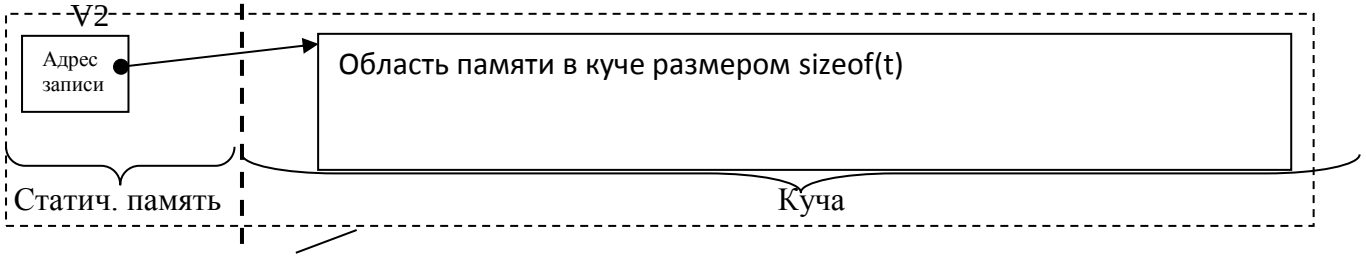
обращение к полям обычной (статической) записи

Вопросы: как создать динамическую запись и как обращаться к ее полям (для заполнения записи)?

1 шаг: объявить соответствующий типизированный указатель (V2).

2 шаг: выделение памяти (в куче)

```
new(v2);
```



динамическая структура (внутри пунктира)

3 шаг: заполнение полей записи (в куче, с помощью разыменования)

Для обращения к полям динамической структуры должна поддерживаться та же форма записи, что и для обычной (статической) структуры, т.е. обращение к полю динамической структуры должно иметь следующий вид:

```
? . имя_поля
```

Слева от точки раньше (в случае статической записи) находилось имя переменной-записи. Однако у динамических переменных, как известно, нет имен. Роль имени у динамических переменных выполняет разыменованный указатель на начало записи (в куче). Здесь (в рамках приведенной выше графической интерпретации динамической записи) для доступа к началу записи надо перейти (от указателя) по стрелке (разыменовать указатель) Поэтому вместо имени будем использовать разыменование указателя с адресом динамической переменной:

```
V2^.a:=1;
V2^.b:=2;
```

Такие составные имена имеют следующий смысл: с помощью операции разыменования мы отыскиваем и обращаемся к началу области памяти, выделенной в куче под динамическую запись; имя поля задает смещение в этой области относительно ее начала (в куче).

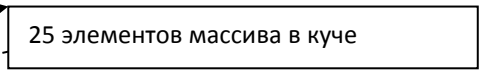
22.10.2 Динамические одномерные массивы (с фиксированными границами)

Замечание 1: Они эквивалентны обыкновенному массиву, образованному (размещенному) динамически в куче (на стадии выполнения).

Замечание 2: В общем случае динамический массив – это массив, границы которого при объявлении не фиксированы.

Динамический массив в Delphi:

```
                                  фактически это указатель на массив из real
                                  фактически это указатель на массив указателей на массивы из longint
  v1: array of real;
  v2: array of array of longint;                    V1
begin
.....указатель      число элементов
  setlength( v1 , 25 ); {выделение памяти}
.....
  v1[20] := 1.2;          {использование} ≡ v1^[20] := 1.2;
.....
  v1 := nil;   или setlength(v1,0); {удаление динамического массива}
```

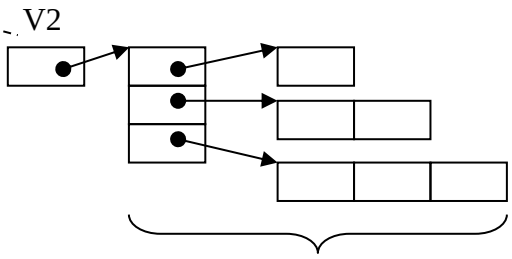



```
число строк                 число колонок
  setlength( v2 , 5 , 10 );   {выделение памяти}
.....
  v2[ 4 , 9 ] := 12.34;             {использование}  ≡ (v2^[4])^[9] := 12.34;
```

индексы начинаются с 0 (поэтому присвоено значение последнему элементу массива)

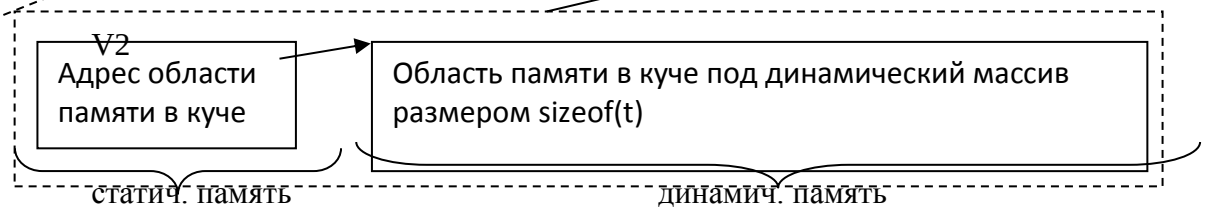
если надо создать непрямоугольный массив:

```
  setlength(v2,3);      {3 строки}
  setlength(v2[0],1);   {1 элемент в 1-ой строке}
  setlength(v2[1],2);   {2 элемента во 2-ой строке }
  setlength(v2[2],3);   {3 элемента в 3-ей строке }
```



Рассмотрим пример:

```
type
  t=array[1..10] of byte;
var
  v1;t;
  v2:^t;                                  разыменованный V2 имеет тип массив
begin                  статический массив
{заполнение статического массива}
  v1[1]:=1;
  v1[2]:=2;
{создание и работа с динамическим массивом}          динамический массив (в пунктире)
  new(v2);
  v2^[1]:=1;
  v2^[2]:=2;
```

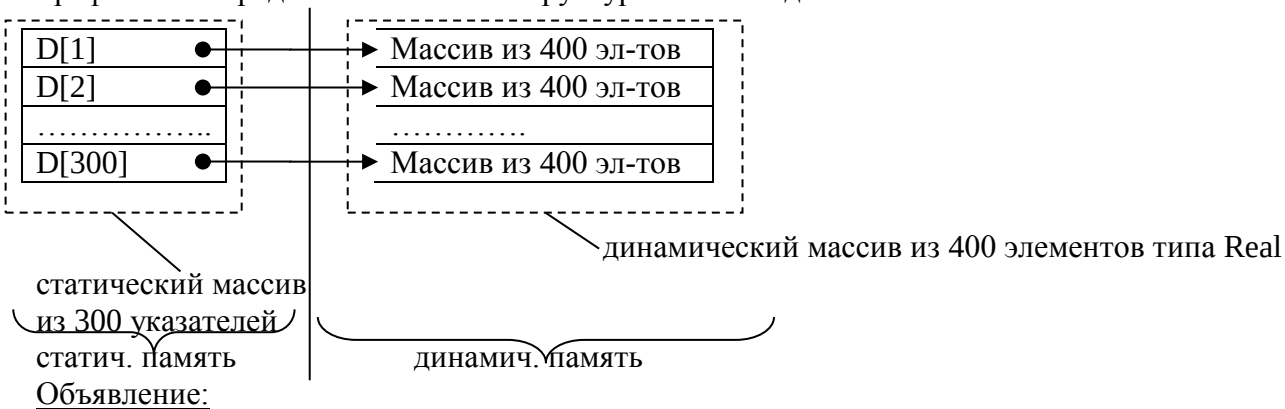



```
  v2^ := v1; //пересылка массива целиком
```

22.10.3 Массивы размером более 64 кбайт (например 300x300 элементов типа Real)

Замечание: в Паскале нельзя создать ни статическую, ни динамическую переменную размером превышающую 64 Кб. Поэтому «длинные» динамические переменные собираются (логически) из отдельных (простых) динамических переменных (<= 64 К байт) так чтобы их суммарный объем достиг нужного размера.

Графическое представление такой структуры имеет вид:



Объявление:

```
type
  D400=Array[1..400] of real; //массив из 400 элементов          м.б. размещен в куче
  D400Ptr=^D400; //указатель на массив из 400 элементов
Var
  D: Array[1..300] of D400Ptr; {Статический массив из 300 указателей на динамический массив из 400
                                                                  элементов}
  i, j : word;  //индексы
begin
{Размещение такой структуры в памяти: }
for i:=1 to 300 do
  New(D[i]);
{Освобождение памяти, занятой такой структурой:}
for i:=1 to 300 do
  Dispose(D[i]);
{Обращение к элементу такого массива (разыменование):}
D[i]^[j]:=……;
```

номер столбца (номер элемента в массиве из 400 элементов типа Real)

номер строки (номер элемента в массиве из 300 указателей)

Недостаток:

Этот прием нельзя использовать для создания одномерных массивов

22.10.4 Динамический массив строк переменной длины

Задача: из текст. файла надо считать все (сколько их будет - неизвестно) непустые строки и разместить их в памяти так, чтобы они заняли столько байт, сколько нужно (сколько занимали в файле) (рваные строки). надо запросить или определить

{Вариант без использования Pchar }

```
uses                          отключаем контроль диапазонов
  crt;
{$R-}                         или array[0..0] of char;
Type
  one_str =string[1];         {тип - строка                          }
  s_ptr    = ^one_str;        {тип - указатель на строку             }
  mas_str  = array[1..4] of s_ptr; {тип - массив указателей на строку  }
Var                                   разыменованный ms имеет тип массив указателей
  i         : word;  {счетчик}        (ms^[i] имеет тип строка)
  stroka  : string;  {сюда читаем строку из файла}
  ms        : ^mas_str;         {указатель на массив указателей на строки}
  str_len   : word;             {длина динамической строки             }
  kol_strok: word;              {число динамических строк              }      длина строки
begin
 clrscr;                                                  (ms^[1])^[0] - 1-й символ 1-й строки
                                          ms       ms^
 {инициализация указателей}                          ms^[1]
  ms := nil;                                         ms^[2]
                                                     ms^[3]
                                                     ……
                                                     ms^[kol_strok]
        или вычисление
 {запрос числа строк}
 write('kol_strok = ?>');
 readln(kol_strok);                                                динамическая строка
                                                                   длиной str_len
                                          массив указателей на динамические строки
                                  статич. память      динамич. память
 {выделение памяти под массив указателей на строки  }
 {в ms будет записан адрес начала массива указателей }
 getmem(ms, sizeof(s_ptr)*kol_strok);

 {выделение памяти под массив строк и заполнение его}      {      В окне Watch:                    }
 for  i:=1 to kol_strok do                                  { ms - адрес массива указателей         }
   begin                                                    { (ms^[i])^ - вся i-ая строка           }
     write('Введите ',i, '-ю строку --> ');                 {a (ms^[i])^[0] - 1-й символ i-й строки}
     readln(stroka);                                        { pchar((ms^[i])) - строка в            }
     getmem(ms^[i], lenght(stroka) + 1);                    {       побайтном изображении,          }
     move( stroka , (ms^[i])^ , length(stroka) + 1 );       {       включая мусор                   }
        откуда   куда   длина
   end;                                  если one_str:array[0..0] of char, то (ms^[i])^ - это
                                         первый символ строки (длина строки)
 {считывание строк из массива строк из динамической памяти в обычную статическую строку}
 for  i:=1 to kol_strok do
 begin
   move( (ms^[i])^, stroka, byte((ms^[i])^[0]) + 1);
   write(i,'-я строка --> ');
   writeln(stroka);              если one_str:array[0..0] of char, то
 end;                            надо     byte((ms^[i])^)
end.
```

22.11 Указатели на подпрограммы (процедуры и функции) - см. 18.12 Процедуры и функции как параметры

На Паскале, к сожалению, нет указателей на подпрограммы (как в языке С).

Нельзя

```
Uses
  crt;
 Type
  t=procedure(s:string);
Var
  p:^t;
Procedure proc1(s:string);
begin; ….end;
Procedure Proc2(s:string);
begin; ……..end;
begin
  clrscr;
  {вызов процедуры 1}
  proc1('Студент');
  p:=@proc1;
  p^('Студент');
……
```

Синтаксических ошибок нет, но под ДОС происходит зависание намертво, а под Windows - - «программа выполнила недопустимую операцию”

Можно

```
Uses
  crt;
 Type
  t = procedure(s:string);
Var
  p: t; ---- скрытый указатель на процедуру
Procedure proc1(s:string); far;
begin
  writeln(s);
end;
Procedure proc2(s:string); far;
begin
  writeln('Ты тоже ' + s);
end;
begin
  clrscr;
  {вызов proc1}
  proc1('Студент');
    или
  p:=proc1;
  p('Студент');
  {вызов proc2}
  proc2('Студент');
      или
  p:=proc2;
  p('Студент');
end.
```

тип указателя на процедуру

процедурный тип

или {$F+}
т.е. процедура должна компилироваться в режиме дальней адресации

скрытый вызов proc1

обращение по адресу, который хранится в *P*, производится без операции разыменования

скрытый вызов proc2

Замечание 1:Если бы подпрограмм было много, то вместо отдельных указателей на функции можно было бы использовать массив указателей на подпрограммы:

```
Const
  P : array[1..N] of t = (proc1, proc2, proc3);
Var
  i: integer;
begin
  for i:=1 to N do
    p[i]('Студент'); //это можно было бы использовать в простейшем меню, где значением i был бы
..........................  //  номер выбранного пункта меню
```

сколько нужно указателей

Замечание 2:

Указателю на подпрограммы можно присвоить: в Free Pascal

- nil;
- значение другого указателя этого же типа;
- адрес (имя) подпрограммы (без использования @ или addr или с их использованием)

Пример меню через указатель на функции (в Си):

```
…
void p(void)
{
  puts(“Вызвана процедура p”);
}
void (*f)(void); //указатель на процедуру
void main(void)
{
  p( ); //вызов p
  f = &p;
  (*p)( ); //тоже вызов p
-------------------------------------
void (*f[10])(void); //массив указателей на процедуру
void main(void)
{
  f[0] = &proc1;
  f[1] = &proc2;
  f[2] = &proc3;        или  void (*f[10])(void) = {proc1, proc2, …};
……….
  f[9] = &proc10;
……
(*(f[1]))( );  //вызов proc2
```

22.12 Динамические связанные структуры данных

Классификация дана в п.22.4.

22.12.1 Понятие связанные динамических структур данных.

Динамические связанные структуры данных строятся из определенного набора специальным образом ***связанных компонентов***. К динамическим связанным структурам данных относятся стеки, списки, деревья, очереди, графы и т.п. Место под каждый компонент такой структуры данных отводится в куче. При этом каждый компонент представляет из себя динамическую запись, у которой нет имени, но есть адрес в области памяти кучи. Каждая такая динамическая запись, состоит как минимум из двух полей:

- **Информационное поле** (для информации, которую необходимо сохранить в данном компоненте). В качестве носителя информации используется: отдельные переменные, массивы, записи и т.д.
- **Поле связи** (одно или несколько), которое используется для организации связи между компонентами. Если поле связи одно, то связь устанавливается с одним из соседних компонентов (или левым или правым). При этом проход по связанной структуре будет возможен лишь в одном направлении (в направлении связей). Если полей связей несколько, то проход возможен в направлении любой из связей. Признаком конца связанной цепочки является значение nil в поле связи. Адрес начала цепочки хранится в статической переменной-указателе.

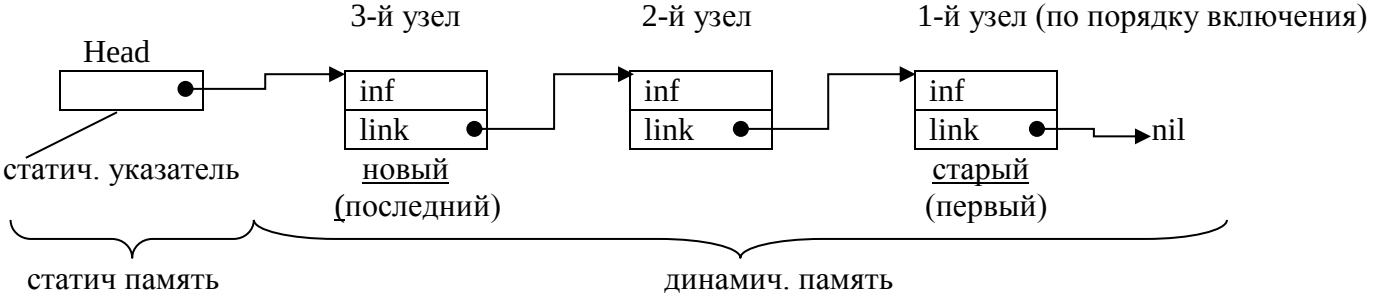
22.12.2 Стеки.

еще есть операция *peek*

22.12.2.1 Понятие стека как структуры данных

Замечание: о стеках речь уже шла ранее. Следует напомнить, что стек является такой структурой данных, у которой 1) доступ (чтение=pop или запись=push) к элементам возможен только **через голову** в соответствии с принципом (дисциплиной) обслуживания LIFO; 2) считывание - разрушающее; 3) направление связей - от нового к старому.

Рассмотрим организацию стека с помощью связанной динамической структуры. Графическое представление такой структуры имеет вид:



где inf – поле информации.

link – поле связи.

Слова «старый» и «новый» надо рассматривать с точки зрения хронологии (порядка помещения информации в стек). Стрелки в этой структуре данных идут в направлении от нового к старому.

Для **представления стека в программе** нам потребуются следующие описания:

```
Type
  stkptr = ^Node;
  node = record
           inf : char;
           link : stkptr;
         end;
Var
  head, oldhead : stkptr;
```

с помощью такого описания можно переходить от звена к звену:

1. head := nil; //стек пуст
2. head^.link^.link^.inf := ‘a”; //3-е звено ≠ head^^^ := ‘a’;

понадобится позже (пока оставить место)

это SP и ESP в процессоре

Замечание: Приведенный выше вариант объявления типа указателя вида ^node формально является

в Паскале ошибкой. Ошибка состоит в том, что заводится указатель на тот тип (node), что описан ниже по тексту программы. На самом деле для таких случаев и в Паскале, и других языках (в Си тоже) сделано ***исключение***: при объявлении связанных структур можно ссылаться на тип, описанный ниже по программе. Но это исключение действует лишь для случая, когда и тип и ссылка на него описываются ***в одной секции type.***

Возможные операции над стеком:

1) Добавление элементов (через голову) - **push**
2) Удаление (извлечение) элемента (через голову) - **pop**
3) Поиск и считывание нужного элемента
4) проверка - какой элемент находится в голове (без извлечение) - **peek**

Замечание: при удалении и считывании нужный элемент отыскивается по его содержимому.

```
в Си:
struct stack
  {
    char inf;
    struct stack *next;
};
```

22.12.2.2 Операции над стеком

а) Добавление элементов в стек

Рассмотрим **процесс создания стека**, начиная с первого звена. Этот процесс будет состоять из многократного повторения **следующих трех элементарных действий**:

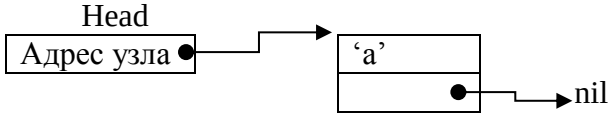
- выделение памяти под новую компоненту (и получение ее адреса);
- с использованием полученного адреса заполнение всех информационных полей новой компоненты (кроме поля связи);
- организация связи между соседними (предыдущими) звеньями стека (через поле связи).

```
{1-е звено}
new(head);
head^.link := nil;
head^.inf := 'a';
.....
```

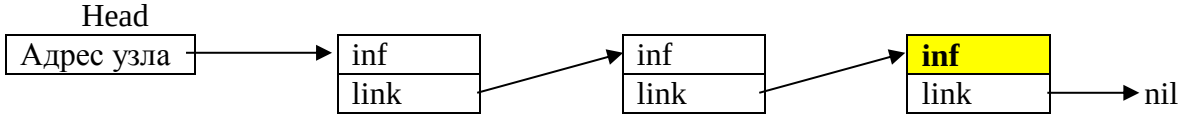
обращение к полям как для динамической переменной

оставить место

Замечание: head^.inf означает доступ к полю inf для динамической структуры, адрес которой хранится в переменной head. То, что эта структура динамическая, означает, что у этой структуры нет имени. С помощью разыменования переменной *head*, хранящей адрес, мы получим доступ к структуре в целом и к отдельному элементу структуры.



Замечание:
Head^ = начало структуры.
Head^.link^.link^.inf – данная конструкция соответствует доступу к полю информации третьего (от головы) элемента стека.

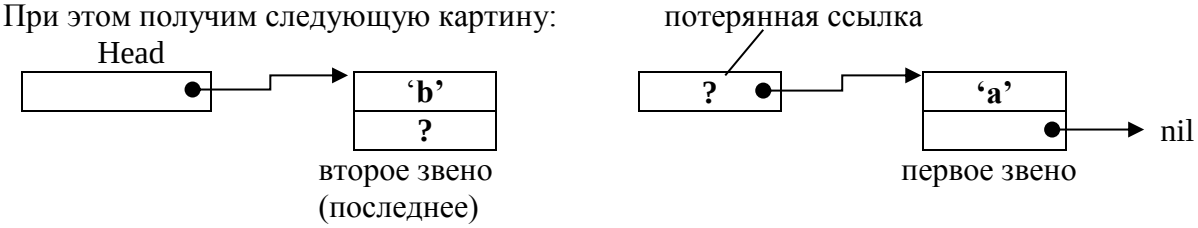


Чтобы не путаться в таких конструкциях, необходимо **помнить о следующем**: каждый элемент link^ означает переход по стрелке на соответствующем графическом изображении связанной структуры.

Попробуем выполнить те же самые действия, которые выполняются для созданного первого звена, **для второго звена** и посмотрим что получиться:

```
{ 2-е звено }
new (head);
head^.inf := 'b';
head^.link := ?;  {надо поставить адрес предыдущего элемента}
```

При этом получим следующую картину:



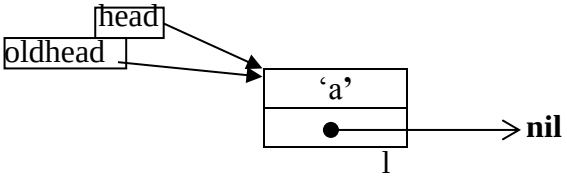
Что имеем:

1) потерялся указатель на первое звено и доступ к нему теперь невозможен
2) 1-е и 2-е звено оказались несвязанными

Для того, чтобы получить картину, соответствующую требуемому связанному изображению стека, необходимо **связать второй и первый узлы**, для чего в поле связи второго элемента нужно записать ссылку на первый. Для того, что бы эту ссылку на первый элемент записать в поле связи второго элемента, ее необходимо сохранить (при создании первого элемента) – в переменной old_head:

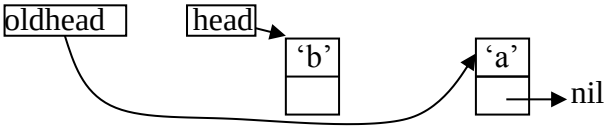
```
{ 1-е звено }
new (head);
head^.link := nil;
head^.inf := 'a';
oldhead := head;
```
---- под это ранее мы оставляли место

После выполнения всех этих действий после создания 1-го звена получается следующая картина:



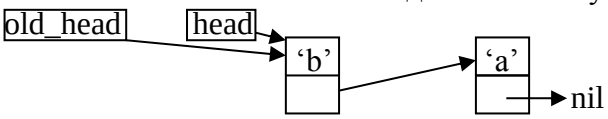

```
{ 2-е звено }
new (head);
head^.inf := 'b';
```

После выполнения всех этих действий после создания 1-го звена получается следующая картина:




```
head^.link := oldhead;
oldhead := head;
```

После выполнения всех этих действий получается следующая картина:



тогда при создании 1-го и последующих звеньев надо писать:

```
new(head);
head^.inf := .....;
head^.link := oldhead;
oldhead := head;
.....................
```

Теперь действия, которые выполняются для второго звена, можно с успехом применять для создания любого n-го (n >= 3) звена. Разница будет только в том, какое значение будет записано в поле информации.

Замечание: легко увидеть, что действия по созданию 1-го и 2-го (3-го, 4-го и т.д.) звена совпадают с точностью до одной строчки:

head^.link := ...

Наличие таких отличий не позволяет включить действия по созданию первого звена в цикл (в тело цикла). Чтобы включить действия по созданию первого звена в цикл, необходимо изменить определение переменной old_head:

```
Const
  oldhead : stk_ptr = nil;
```

Можно определить (пусть имеется):

```
procedure add(var p1:stkptr; var p2:stkptr);
```

где p1 и p2 – head и old_head соответственно

С ее использованием заполнение стека выглядело бы так:

```
for i:= 1 to n do
    add(head, oldhead);
```
можно сделать глобальной переменной

```
head := nil
oldhead := nil;
for i := 1 to n do
 begin
    new(head);
    head^.inf := .....;
    head^.link := oldhead;
    oldhead := head;
 end;
```

б) Удаление элементов из стека

Удаление элементов стека (с головы) может производиться **двумя путями**:
1). Без физического освобождения памяти, занимаемой освободившимися узлами стека.
2). С физическим освобождением памяти, занимаемой освободившимися узлами стека.

1-й путь: //очиска стека - head := nil;

Исключение одного звена (через голову) выполняется одной строчкой типа head := head^.link; для исключения всех звеньев требуется цикл:

небезопасно — пока в стеке есть звенья

```
While head^.link <> nil do
 Head := head^.link ;
```

перемещаем голову на следующий элемент (выкидываем элемент из головы ≡ разрушаем стек)

Будут удалены все звенья, исключая первое

безопасно

```
While head <> nil do
 Head := head^.link ;
```

безопасно только при наличии в стеке элемента с link = nil

Будут удалены все элементы

Здесь используется следующая **идея освобождения памяти** узлами (ячейками) стека:
Поскольку каждый узел стека – это динамическая переменная, то эта переменная существует, пока возможна ссылка на нее (известен ее адрес). Поэтому действие - убрать узел из стека – равнозначно уничтожению ссылки на него для того, чтобы доступ к этому узлу стал невозможен. Формально это означает замену стрелок на графическом изображении стека: той стрелки, которая шла к узлу, на ту, которая шла от него. После замены стрелок к тому узлу, слева у которого стрелку убрали, доступ становится невозможен (потерянная ссылка).

Данный способ освобождения элементов из стека имеет **достоинство**: мы всегда можем вернуть назад прежнее состояние стека. Это возможно потому, что физическая структура стека при освобождении элементов не портилась. Для этого надо выполнить следующее:

Head := old_head;

Второй путь:

```
While head <> nil do
  begin
    oldhead := head^.link; {сохраним ссылку на следующий элемент}
    dispose(head); {уничтожить узел}
    Head := oldhead; {голова перемещается на новое место}
  End;
```

Нельзя было:
head:=head^.link;
(терялась ссылка на 1-ый узел)

Здесь **нельзя сразу применять dispose**, т.к. при выполнении dispose(head) теряется голова стека. Поэтому вначале (перед dispose) нужно сохранять ссылку на следующий элемент.

в) Поиск элемента в стеке (линейный поиск)

Разрушающее считывание

```
Const
 нашли:boolean = false;
…
begin
 …
 Repeat //небезопасно
   If (head^.inf = искомое значение)
    Then нашли := true
    Else Удалить элемент из стека (из головы)
 Until (нашли = true) OR (head = nil) ;
 If (нашли = true)
   Then Действия;
End.
```

можно было бы изменить значение head, например, по рассмотренному выше 1-му пути head := head^.link;

когда дошли до конца стека

Так нельзя
head := head^.link;
так как при этом стек при поиске будет разрушен (и после поиска его надо будет восстанавливать).
Надо вместо head при поиске использовать, например, oldhead:

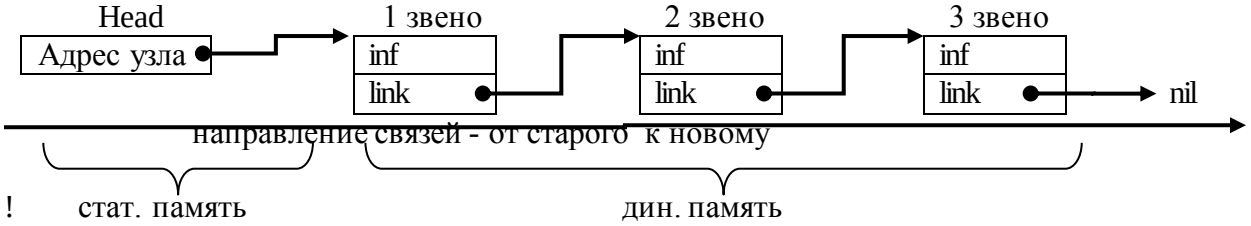
```
.....
oldhead := head;
while (oldhead <> nil) and (нашли <> true) do
  begin
    .....
    oldhead := oldhead^.link;
    .....
 end;
```

head остается неразрушенным

22.12.3 Линейные однонаправленные СПИСКИ

22.12.3.1 Понятие линейного однонаправленного списка

Графическое изображение однонаправленного списка имеет вид:



Имеются следующие 4-е **отличия в организации стеков и списков**:

1. В направлении связи. Если в стеке направление связей (стрелок) шло от нового к старому. То в списке наоборот – от старого к новому.
2. Если в стеке при включении нового элемента голова перемещалась на этот новый элемент, то в списке положение головы фиксируется, и она всегда показывает на самый первый включенный в список элемент.
3. Если в стек включать новый элемент можно было только одним способом – сделать его самым первым, то в списке можно включать элемент в любое место.
4. Если в стеке для того, чтобы получить доступ к нужному элементу, необходимо было вытолкнуть из стека все что находится выше него, то в списке для этого никаких элементов выталкивать не надо. В списке можно обращаться напрямую к любому элементу (неразрушающее считывание).

Для **представления списков в программе** понадобится следующее описание:

```
Type
 Lstptr= ^node;
 Node = record
          Inf : char;
          Link : lstptr
        End;
Var
 head , current : lstptr;
```

изменять эту переменную нельзя, от неё будем перемещаться от элемента к элементу

текущее звено (которое создали или на котором находимся в данный момент при перемещении по списку)

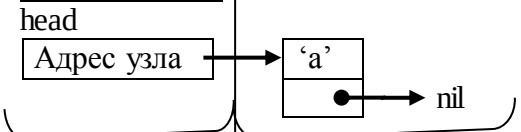
вначале тут оставить место под переменную

Рассмотрим создание списка начиная с **первого звена**:

```
{ 1-е звено }
new (head);
head^.inf := 'a';
head^.link := nil;
.....................;
```

Как для стека

---------- оставить место!

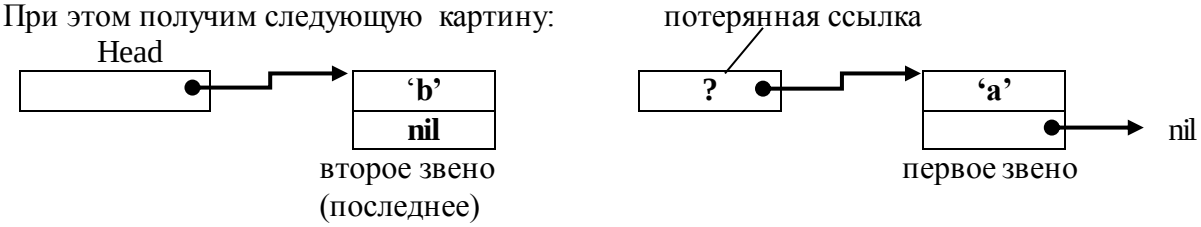


Все ***проблемы*** начинаются со **второго звена**. Возможны по крайней мере два некорректных подхода к созданию 2-го и последующих звеньев:

1 подход: можно попробовать второе звено создавать по образу и подобию первого, но так делать нельзя, т.к. head должен (после создания 2-го звена) указать на первый элемент, а не на второй (новый), как это было в стеке.

```
New(head);
Head^.inf := 'b';
Head^.link := nil;
```

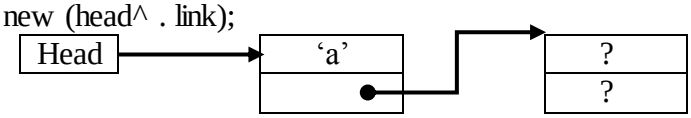
При этом получим следующую картину:



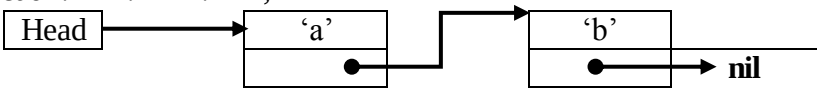
Даже если здесь сохранить ссылку на предыдущее звено (в переменной oldhead - как было у стека), то все равно при той же картине в памяти (как у стека) голова будет указывать на 2-ой элемент, а должна на 1-ый.

2 подход: его использование избавлено от проблем 1-го подхода, но приводит к проблемам с записью последовательности операций

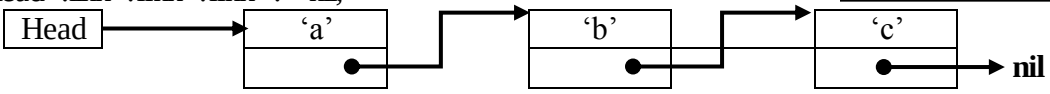
{ 2-е звено }
new (head^ . link);



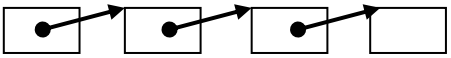
head^.link^.inf := 'b'
head^.link^.link := nil;



{ 3-е звено }
new(head^.link^.link);
head^.link^.link^.inf := 'c';
head^.link^.link^.link := nil;

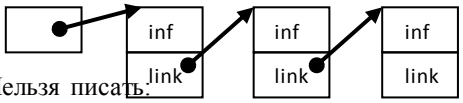


1) p1 p2 p3 p4



Можно писать:
p1^^^ --- попадем на 4-ый элемент

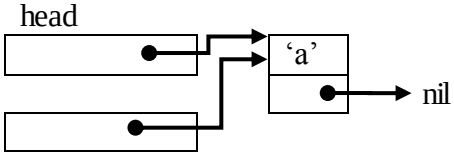
2) p



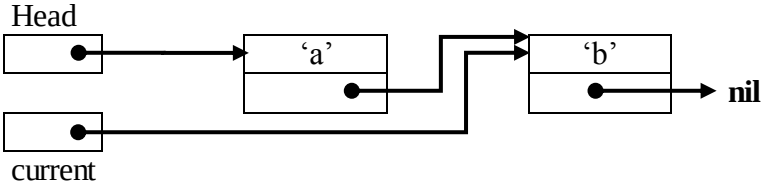
Нельзя писать:
p^^^
Надо писать:
p^.link^.link^ - попадем в 3-е звено (справа крайнее)

Проблема состоит в том, что для этого второго подхода, действия по созданию второго, третьего..седьмого звеньев **нельзя обобщить и поэтому нельзя описать в цикле** (если надо создать список заданного размера). Для того, чтобы включить все действия по созданию списка в цикл (или в подпрограмму), необходимо использовать специальную **переменную current** – она будет использоваться для отметки текущего (создаваемого в данный момент) узла списка. Для того, чтобы ее использовать, при создании первого звена надо заложить еще одно действие. Тогда процесс создания звеньев с первого будет выглядеть следующим образом:

{ 1-е звено }
new(head);
head^.inf := 'a';
head^.link := nil;
current := head; ---------- вставили на оставленное ранее место



{ 2-е звено и последующие - как в рассмотренном выше 2-ом подходе, но не через *head*, а через *current*}
new(current^.link);
{перетаскиваем current на текущий узел}
current := curent^.link;
current^.inf := '…';
current^.link := nil;



Замечание: Из приведенной последовательности действий видно, что формирование первого звена выполняется не так, как остальных. Чтобы такого не было и для удобства работы со списком, часто в список вводят т.н. "заглавное" или **"нулевое звено"**, в поле link которого содержится ссылка на первый (работающий) узел стека, а в поле информации можно помещать идентифицирующую список информацию:

```
Type
 lstptr= ^node;
 node = record
            Inf : char;
            Link : lstptr
          End;
Var
 Head, current : lstptr;
 i: integer;
begin
 { формирование «нулевого» звена }
  new(head);
  current := head;
  current^.inf := …..;
  current^.link := nil;
 { формирование 1-го звена и последующих }
  for i:=1 to <число звеньев> do
    begin
       new(current^.link);
       current := curent^.link;
       current^.inf := '…';
       current^.link := nil;
    end;
end.
```

head указывает на «нулевое» звено, а адрес 1-го звена определяется выражением *head^.link*

первую из операций – создание (добавление в хвост) – мы уже рассмотрели только что выше

22.12.3.2 Основные операции над списками.

а) Поиск заданного элемента в списке

1) **Замечание**: слово «заданный» понимается в том смысле, что мы будем искать тот элемент, в поле информации которого находится значение, удовлетворяющее заданному критерию.
2) **Замечание**: при работе со списком обычно неизвестно, в каком конкретно элементе хранится заданное содержимое, причем нет возможности напрямую обращаться к элементу с заданным значением (как говорят, элементы списков и стеков адресуются по содержимому). Поэтому необходимо организовать перебор (в цикле) всех элементов списка до тех пор, пока в поле информации очередного элемента не найдем то значение, которое ищем.
3) **Замечание**: начало списка отмечено указателем head, а конец списка отмечается значением nil в связи последнего элемента
4) **Замечание**: решать задачу будем в следующей **постановке**: необходимо **найти номер узла,** содержащего заданное значение.
5) **Замечание**: выход из цикла осуществляется если:
   - нашли то, что ищем (в поле информации текущего элемента списка содержится указанное значение);
   - не нашли, но достигнут конец списка (в поле link текущего элемента содержится значение *nil*).

С учетом сделанных замечаний решение задачи будет иметь следующий вид:

```
Var
 I : integer;  {искомый номер узла}
 J : integer;  {счетчик пройденных узлов списка начиная с 1-го}
 нашли : boolean;  {признак нашли/не нашли то, что ищем}
 { Описание списка }
……….
Begin
 { Заполнение списка }
……..
{ Подготовка к поиску }
 i := 0; -------- начальное значение номера искомого узла
 j := 1; --------- начальное значение счетчика пройденных элементов списка
 нашли := false;  ----- пока еще не нашли
 Current := head;
{ Поиск нужного звена }

 Repeat
  if current^.inf = '?'
     then
        Begin
          {запоминаем номер j в переменной i}
          i := j;
          {готовим выход из цикла}
          нашли := true;
        End
     else
         begin
           { Переход к следующему звену списка}
           Current := current^.link;
           j := j+1;
         end;
 Until ( current = nil ) or (нашли = true);
{ Если переменная нашли = true, значит мы нашли то, что искали. Если нашли = false значит не нашли. }
 if (нашли = true)
   then
     Write('нужная информация находится в звене номер ', i : 2)
   Else
      Writeln ('не нашли');
End.
```

небезопасно

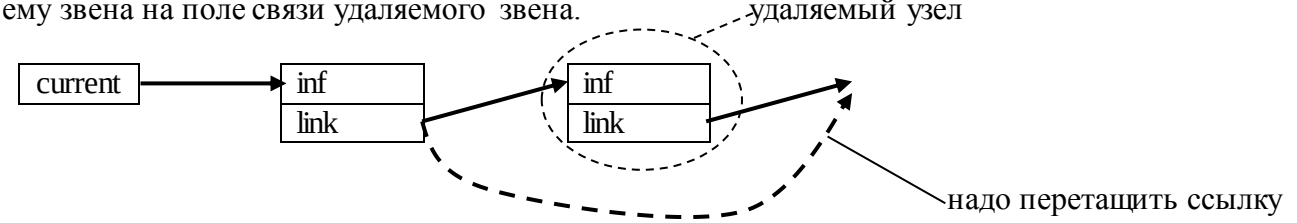
шаблон поиска

в списке больше нет элементов

б) Удаление заданного элемента.

Замечание 1: как и в предыдущей операции (поиск) слово «заданный» понимается в том смысле, что мы будем перед собственно удалением искать (и потом удалять) тот элемент, в поле информации которого находится значение, удовлетворяющее заданному критерию.

Замечание 2: Поиск будет выполняться в цикле. Вспомним, что доступ к элементам списка (список – динамическая структура) осуществляется лишь по ссылкам, содержимся в предшествующих им элементах списка. Если какое-то звено физически существует, но на него нет ссылки из какого-то другого звена, то оно недоступно при последовательном переборе звеньев от начала к концу списка. Поэтому считается, что такое «потерянное» звено не входит в список (исключено из него). Фактически процедура удаления из списка элемента заключается в **выполнении следующих действий:** если нужно удалить некий элемент из списка, то нужно заменить поле связи предыдущего ему звена на поле связи удаляемого звена.



С учетом сказанного в процессе поиска нужного элемента **цикл по перебору элементов будет завершен** или когда мы найдем элемент, предшествующий удаляемому (если в поле inf следующего элемента находится искомое значение, то есть current^.link^.inf = <искомое значение>), или при достижении конца списка (если в поле связи текущего элемента содержится *nil*). После этого будет необходимо поменять поле связи у элемента, предшествующего удаляемому, на поле связи удаляемого элемента. В виде программы это будет выполняться следующим образом:

Описание списка

```
Var
 нашли : boolean; {признак нашли (не нашли) что удалять}
 Buffer : lstptr; {вспомогательный указатель }
                  {будет нужен при переносе ссылки}
Begin
```

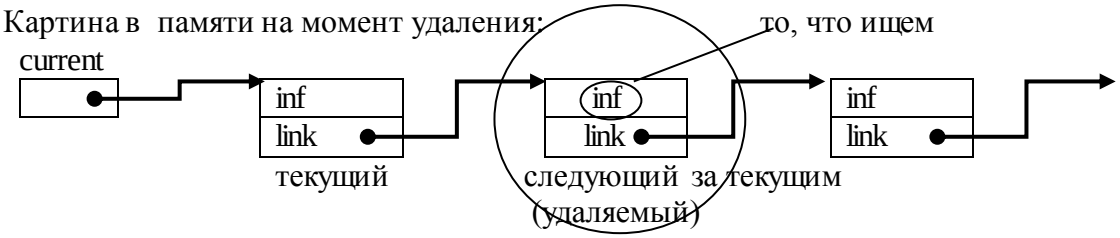
Заполнение списка

```
{Подготовка к поиску элемента, предшествующего заданному}

 нашли := false;
 { Встаем на 1-й элемент}
 Current : head;
 { Поиск узла, предшествующему удаляемому }
 While (current <> nil) and (нашли <> true) do
   Begin
    {Анализ поля информации следующего звена}
    If current^.link^.inf = <то, что ищем>
       Then
         {нашли}
         нашли := true
       Else
         {не нашли и идем к следующему звену }
         Current := current^.link;
   End;
```

```
Встать на 1-й элемент списка
Нашли := ложь
Пока (не нашли) и (в списке есть элементы)
 Нач
   Если поле инф. следующего (за текущим)
    элемента содержит то, что надо
     То
       нашли := истина
     Иначе
       перейти к следующему элементу
 Кон
Если нашли := истина
 То заменить поле связи текущего элемента
```

Картина в памяти на момент удаления:

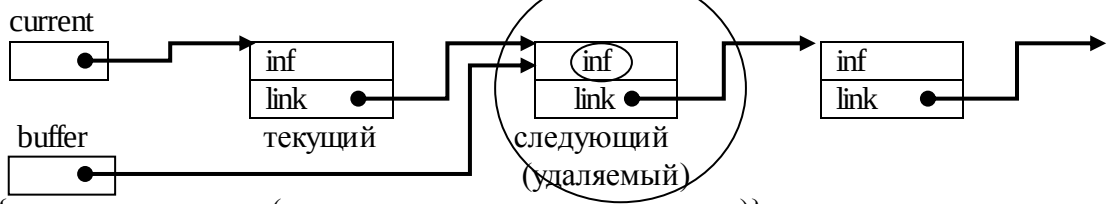


{ Предполагается, что если нашли = true, то current отмечает блок, предшествующий удаляемому}

```
if нашли = true                                   удаление без освобождения памяти
  then {нашли}
    current^.link := current^.link^.link   {меняем поле связи предыдущего звена}
  else writeln('Не нашли');
```

Тот же по смыслу фрагмент, но **с освобождением памяти**:

```
{сохраняем для dispose() ссылку на удаляемый блок}
buffer:= current^.link;
```




```
{меняем поле связи (меняем левую стрелку на правую)}
current^.link:= current^.link^.link;
{освобождаем память}
dispose(buffer);
```

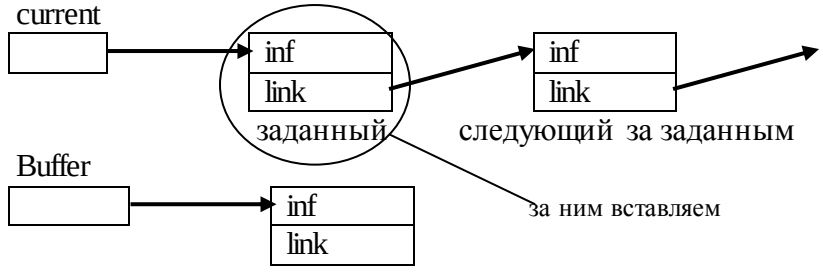
в) Вставка нового элемента после заданного

Замечание: необходимо перед собственно вставкой *найти* «**заданный**» элемент, то есть элемент за которым надо вставить новый. current^.inf = эталон

Когда нашли заданный элемент (на него будет указывать current), то физически процесс вставки элемента после заданного должен происходить следующим образом:

- Динамически выделить память под новый элемент списка (адрес запоминаем в Buffer);
- В поле информации этого нового элемента занести нужную информацию;
- В поле связи **нового** элемента поместить ссылку из поля связи заданного элемента (на него указывает current);
- В поле связи **заданного** элемента занести адрес вставляемого элемента (из Buffer).

Картина памяти на момент вставки:



В виде программы решение выглядит таким образом:

описание списка

```
Var
 нашли : boolean;  {признак нашли (не нашли) элемент, после которого вставлять}
 Buffer  : lstptr;  {вспомогательный указатель}
Begin
```

поиск заданного элемента

Заполнение списка

```
{ Подготовка к поиску элемента, предшествующего заданному}
нашли := false;
{ Встаем на 1-й элемент}                    условие завершения: (current = nil) or (нашли = true)
Current : head;
{Поиск узла предшествующему вставляемому}
While  (current <> nil) and ( нашли <> true ) do
        не коне списка          не нашли
 Begin
   {Анализ поля информации текущего звена}
    if current^.inf = 'то что ищем'; { проверяем текущее звено по шаблону поиска }
       Then {нашли}
         нашли := true
       Else {не нашли и идем к следующему звену}
         Current := current^.link;
 End;
{ Предполагается, что если нашли = true, то current отмечает текущий блок}
if нашли = true
  then  { нашли и вставляем}
   begin
7     new(buffer);  { выделяем память под новый узел }
      buffer^.inf:=  …….; { заполняем поле инф. нового узла }
      buffer^.link:=current^.link;  { заполняем поле связи нового узла }
      current^.link:=buffer;  { заполняем поле связи текущего узла }
   end
  else
    writeln( 'некуда вставлять' );
end.
```

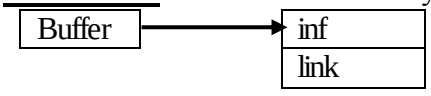
вообще-то так нельзя делать
(голова списка не должна изменяться)

г) Вставка в начало списка

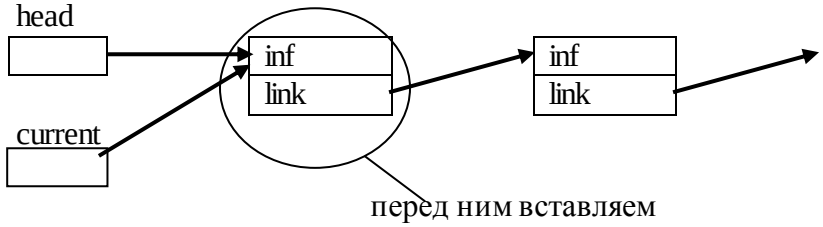
Алгоритм вставки:

1) Выделить динамическую память под новый элемент (адрес поместить в buffer)
2) В поле связи нового элемента поставить значение из head (current)(ссылка на бывший первый элемент)
3) В head записать значение равное адресу нового элемента (из buffer)

Замечание: в момент ставки буфер должен быть не пустой, то есть должна быть такая картина:



вставляемое звено



перед ним вставляем

В виде программы имеем:

```
{ встали на 1-й элемент }
current:=head;
begin
 { заполнение списка }
……..
 { собственно вставка }
 new(buffer);
 buffer^.inf:=……..;
 buffer^.link:=head;
 head:=buffer;
 current := head;
```

В данной операции перед вставкой не надо проверять, куда вставлять - это заранее известно

похожие действия выполнялись при создании 2-го и следующих звеньев списка

д) Вставка в конец списка

Замечание: у него current^.link = nil

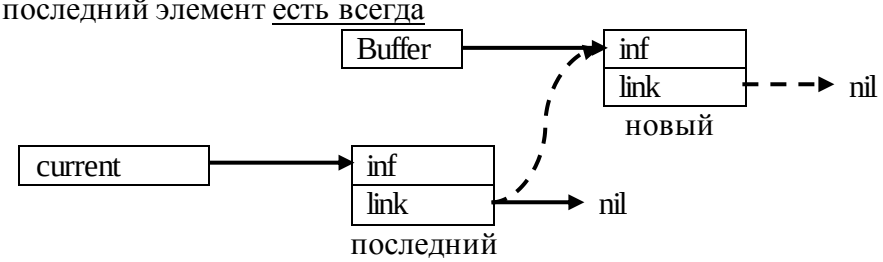
Сначала надо встать на последний элемент (current должен содержать его адрес), а уже потом собственно вставить

Алгоритм собственно вставки:

1) Выделить динамическую память под новый элемент (адрес – в buffer)
2) В поле связи **нового** (вставляемого) элемента (buffer^.link) занести значение nil
3) В поле связи последнего элемента (current^.link) записать значение равное адресу нового элемента (buffer) на него указывает current

```
……
1) { встали на 1-й элемент }
    current:= head;
2) { проход до последнего элемента, т.е. пока в поле связи элемента не появится nil }
    while current^.link <> nil do
      current:=current^.link; { переход к следующему элементу }
3) { собственно вставка }
    new(buffer);
    buffer^.inf:=….;
    buffer^.link:=nil;
    current^.link:=buffer;
……
```

Замечание: перед вставкой не надо проверять, нашли или нет то, что искали, так как в списке последний элемент есть всегда



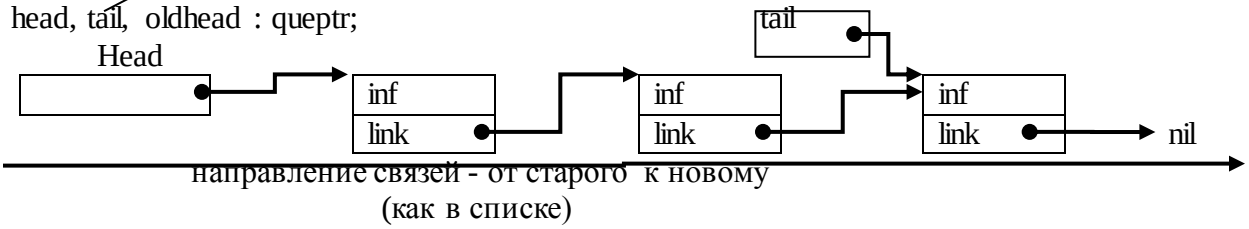
22.12.4 Очередь

Очередь – динамическая структура данных, добавление элементов в которую производится в один конец (хвост), а выборка элементов осуществляется с другого конца (головы).

Над очередью, как и над стеком, определены две операции – добавление (в хвост) и выборка (из головы - в очереди, как и в стеке разрушающая выборка).

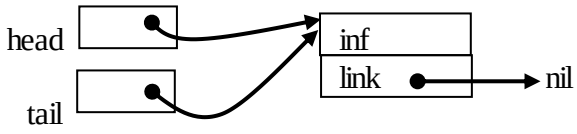
Очередь реализует, в отличие от стека, дисциплину обслуживания FIFO - извлекается элемент, пришедшим в очередь первым (в стеке - LIFO).

```
Type
 queptr= ^node;
 node = record
          inf  : char;
          link  : queptr
        end;
Var        указатель на хвост
 head, tail, oldhead : queptr;
```



Создание

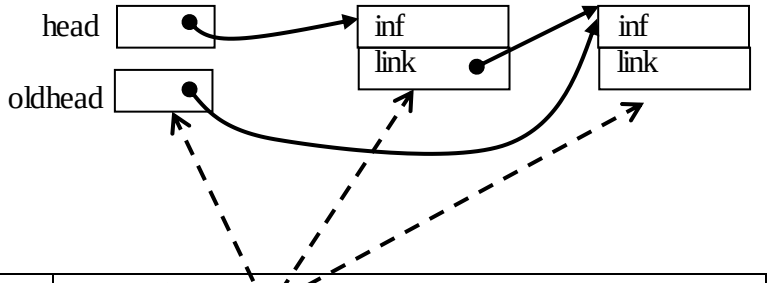
```
{ 1-е звено }
new(head);
head^.inf  := …;
head^.link  :=  nil;
tail := head;
{ 2-е звено и последующие  - добавление в хвост}
new(tail^.link);
{перетаскиваем хвост на добавляемый узел}
tail := tail^.link;
tail^.inf  := '…';
tail^.link  := nil;
```



хвост остается на месте, пока не извлечем последний элемент

Извлечение одного элемента

```
{без освобождения памяти}
head := head^.link;
{проверка на последний элемент}
if  head = nil
  then
     tail := nil;
```



Извлечение всех элементов

Без освобождения памяти

```
While head <> nil do
  begin
  head := head^.link;
  end;
tail := nil;  //при этом head  = nil
```

перемещаем голову на след. элемент

С освобождением памяти

```
While head <> nil do
  begin
    oldhead := head^.link;
    dispose(head);
    head := oldhead;
  end;
tail := nil;  //при этом head  = nil
```

Сохраняем ссылку на элемент, следующий за удаляемым (он станет первым)

22.13 Строки с завершающим нулем

22.13.1 Понятие, определение и объявление в программе

В Turbo-Pascal версии 7.0 для совместимости с другими языками программирования (Си) и ОС Windows (там основной язык программирования = Си) введен еще один (кроме string) вид (тип) строк – строки, оканчивающиеся символом с кодом 0 (#0) - т.н. ASCIIZ-строки или строки с завершающим нулем.

в Delphi это ShortString

Значения строк обычного типа (String) хранятся в памяти как последовательность байт, первым (младшим по адресам) из которых является байт длины строки, за которым следует последовательность символов строки. Максимальная длина строки типа string равна 255 символам. Таким образом, обычная строка занимает до 256 байт памяти.

#6	‘п’	‘р’	‘и’	‘в’	‘е’	‘т’

это просто ноль

В отличие от строк типа string строки с завершающим нулем не содержат байта длины, поэтому для ASCIIZ-строк не накладывается ограничение на их размер (фактически размер может быть до 65535 байтов - в Паскале нельзя определить тип, длиннее чем 64 Кбайт).

‘п’	‘р’	‘и’	‘в’	‘е’	‘т’	#0

Для таких ASCIIZ-строк в Паскале 7.0 имеется стандартный тип PChar, который представляет собой указатель на строку с завершающим нулем. Фактически этот тип является (описан в модуле System) указателем на символ:

в Delphi им соответствуют AnsiString и PChar

```
Type
  PChar = ^Char,
```

(они совместимы, но не равны)

Однако, если разрешен *расширенный синтаксис (директивой {$X+} - она действует по умолчанию)*, то тип PChar совместим с т.н. символьным массивом с нулевой базой (индексы массива начинаются с нуля и длина строки не хранится в строке). Это позволяет использовать переменную типа PChar как строку, эквивалентную массиву символов с нулевой базой типа


```
PChar ≡ array[0..n] of Char,
```

потому что индексация начинается с 0

где n - количество символов в строке, *не считая* завершающего строку символа с кодом 0.

В отличие от типа string символ с индексом 0 здесь является первым значащим символом строки (а не длиной как раньше), а последний символ с индексом n - завершающим символом с кодом 0.

При работе с ASCIIZ-строками целесообразно включать расширенный синтаксис. В этом случае не возникают трудности при использовании различных стандартных подпрограмм, работающих со строками.

22.13.2. Особенности использования строк с завершающим нулем.

22.13.2.1 ***Инициализация*** с помощью присваивания

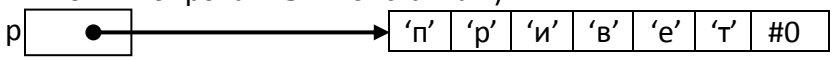
а) присваивание переменной типа PChar значения строкового литерала (константы без имени и типа)

При разрешении расширенного синтаксиса с типом PChar совместим по присваиванию строковый литерал (строковая константа), т.е. переменной типа PChar можно присвоить строковый литерал. Например:

```
Var
  P: PChar;
begin
  P := 'Привет';
  Writeln(‘P = ‘, P) { Выведет P = Привет}
   end;
```

В результате такого присваивания указателю P не присваивается строковый литерал, вместо этого указателю P присваивается адрес области памяти, содержащей строку с завершающим нулем, соответствующей строковому литералу (в сегменте данных). Дело в том, что компилятор строковые

литералы (ASCIIZ –строки, создаваемые статически) размещает (хранит) в сегменте данных (аналогично типизированным константам).

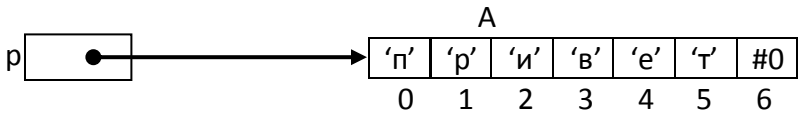


б) присваивание переменной типа PChar имени символьного массива с нулевой базой

Уже говорилось, что, если с помощью директивы {$X+} разрешен расширенный синтаксис, то символьный массив с нулевой базой совместим с типом PChar. Это означает, что везде, где ожидается использование типа данных PChar, может использоваться символьный массив с нулевой базой.

Например:

```
var
  A: array[0..6] of Char;
  P: PChar;
Begin
  A := 'Привет';
  P := A;
  Writeln(‘P = ‘, P) { Выведет P = Привет}
  Writeln(‘A = ‘, A) { Выведет A = Привет}
end;
```

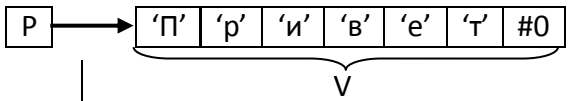


Известно, что компилятор интерпретирует имя любого массива (в нашем случае – символьного) как указатель-константу, значением которого является адрес первого элемента массива. Благодаря оператору присваивания P := A; P теперь указывает на первый элемент массива A.

в) присваивание переменной типа PChar адреса (или имени) типизированной константы, инициализированной литералом + #0

массива с нулевой базой

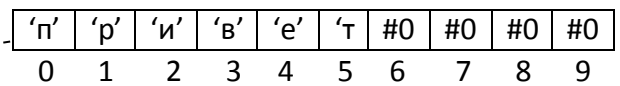
```
const
  V: array[0..6] of Char ='Привет'#0;
var
  P: PChar;
begin
  P := @V; {или P := V;}{или P := ‘Привет’;}
  Writeln(‘P = ‘, P); { Выведет P = Привет}
  Writeln(‘V = ‘,V); { Выведет V = Привет}
end;
```



при присваивании (при выполнении) так делать не надо

Замечание: Типизованную константу, имеющую тип символьного массива с нулевой базой, можно инициализировать с помощью строкового литерала, имеющего только меньшую длину, чем размер массива (если длина литерала будет равна или больше размера массива, то не останется места в строке под #0 и нельзя будет использовать этот массив как ASCIIZ –строку). Если меньше, то оставшиеся символы массива устанавливаются в значение NULL (#0), и массив будет представлять собой строку с завершающим нулем.

```
type
  T = array[0..9] of Char;
const
  F: T =’привет';
```



г) инициализация типизированной костанты типа Pchar значением строкового литерала

```
Const
  P: Pchar = ‘Привет’#0;
```

д) присваивание формальному параметру типа PChar значения строковой константы при передаче фактического значения формальному параметру типа PChar

Когда формальный параметр процедуры или функции имеет тип PChar, то в качестве фактического параметра при вызовах можно использовать строковые литералы. Например, если имеется процедура с описанием:

```
procedure P(Str: PChar);
```

то допустимы следующие вызовы процедуры:

```
P('Строка для проверки');
P(#10#13);
```

В этих вызовах аналогично тому, как это происходило при присваивании значение переменной типа PChar (см. выше), компилятор генерирует строку с завершающим нулем, представляющую собой копию литерала-строки в сегменте данных, и передает указатель на эту область памяти в параметр Str процедуры P.

22.13.2.2 ***Индексирование*** значения указателя типа PChar

Так как символьный массив с нулевой базой совместим с указателем типа PChar, то этот указатель (PChar) можно индексировать аналогично символьному массиву с нулевой базой (если заранее позаботиться об инициализации указателя адресом символьного массива).

```
var
        A: array[0..6] of Char;
        P: PChar;
        Ch: Char;
begin
        A := 'Привет';
        P := A;
        Ch := A[5]; { Индексируем имя массива }
        Ch := P[5]; { Индексируем имя PChar }
end;
```


В данном примере оба последних присваивания присваивают Ch значение, содержащееся в шестом символе массива A.

При индексировании символьного указателя индекс задает беззнаковое значение (смещение), которое добавляется к значению указателя перед его разыменованием. Таким образом

P[0] эквивалентно P^ и задает символ, на который указывает P (первый элемент массива).

P[1] задает символ справа от того, на который указывает P, **≡** (p+1)^

P[2] задает следующий справа символ от P[1],

и т.д.

Замечание: Необходимо учитывать, что компилятор при индексировании указателя PChar не выполняет проверку диапазона (проверку выхода за границы массива), так как у него нет информации о типе, по которой можно определить максимальную длину строки с завершающим нулем, на которую указывает символьный указатель.

считается, что адресная арифметика быстрее, чем индексирование

22.13.2.3 ***Адресная арифметика*** (арифметические операции со значениями указателей PChar)

Расширенный синтаксис Pascal ({$X+}) позволяет использовать для работы с указателями PChar отдельные арифметические операции. Для увеличения или уменьшения смещения в значении указателя PChar можно использовать операции плюс (+) и минус (-) или процедуры inc(p) и dec(p). При этом операцию минус (-) можно использовать для вычисления расстояния (разности смещений) между двумя символьными указателями PChar.

Предположим, что P и Q представляют собой значения тип PChar.

Тогда допустимы следующие конструкции:

P + 2 ---- 2 прибавляется к значению указателя P

P - 2 ---- 2 вычитается из значения указателя P

P - Q ---- значение Q вычитается из значения P

При работе с группой символов в операции P + 2 к адресу, задаваемому P, прибавляется 2. При этом получается указатель, который указывает на 2 символа после P (вправо). В операции P – 2 2 вычитается из адреса, задаваемого P, и получается указатель, указывающий на 2 символа до P (влево).

Операция P - Q вычисляет расстояние между Q (младший адрес) и P (старший адрес). При этом возвращается результат типа Word, показывающий число символов между Q и P. Эта операция предполагает, что P и Q указывают на один и тот же массив символов. Если эти два указателя указывают на разные символьные массивы, то результат непредсказуем.

22.13.2.4 Операции ***сравнения*** >, <, <=, >=, =, <> применительно к указателям типа PChar

Ранее говорилось, что стандартный синтаксис Pascal позволяет при сравнении указателей определять только их равенство или неравенство. Однако, расширенный синтаксис (разрешенный по директиве компилятора {$X+}) позволяет применять операции <, >, <= и <= к значениям PChar. Однако, стоит помнить, что при таких проверках предполагается, что два сравниваемых указателя указывают на один и тот же массив символов. Если два указателя указывают на различные символьные массивы, то результат не определен.

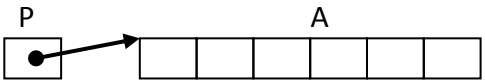
22.13.2.5 ***Посимвольный*** ввод-вывод с использование строк с завершающим нулем

Расширенный синтаксис Pascal позволяет применять к символьным массивам с нулевой базой стандартные процедуры Read, ReadLn и Val, Write, WriteLn, Assign и Rename.

Пример использования функций с завершающим нулем

```
{ Программа, демонстрирующая возможность обращения в Turbo/Borland Pascal 7.0 к постоянной (нетипизированной) строке текста через указатель типа pChar}
program StringTest;
uses
 Strings; {этот модуль нужен для работы с ASCIIZ-строками}
const
 A = 'Строка A постоянного текста';
var
 i: byte;
 P: pchar;
begin
 writeln;
 { Внимание: длину строки можно определить не через length, а через strlen (из модуля Strings)! }
 writeln('Длина строки A - ',length(a),' символов');
```

```
{ Определение указателя P на строку текста }
P:=A;
{ Вывод всей строки текста A через указатель P }
writeln(P);
{ Посимвольный вывод строки через указатель P}
for i:=1 to strlen(P) do ----------
  write((P+i-1)^);
writeln;
{ Посимвольный вывод фрагмента строки через указатель, начиная с 10-го символа}
for i:=10 to strlen(P) do
  write((P+i-1)^);
writeln;
{ Посимвольный вывод строки в обратном порядке }
for i:=strlen(P) downto 1 do
   write((P+i-1)^);
writeln;
{ Посимвольный вывод строки в стиле Си }
i:=0;                    предполагается, что P отмечает цепочку символов
while ord((P+i)^)<>0 do { пока не встретится 0-байт }
   begin
      write((P+i)^);
      inc(i);
   end;
readln
end.
```



Длину строки PChar надо определять не через length, а через strlen (у string, ShortString и AnsiString - через length)

Замечание: Если используется ***ключ {$X-}***, то переменная типа PChar в этом случае рассматривается как указатель на один единственный символ. В этом случае для описания

```
Type
  TSt = array[0..7] of Char;   {массив для строки из 7 символов}
var
   Stroka : PChar;
const
   A: TSt = 'привет!'#0;
begin
  Stroka := 'ПРИВЕТ!';
  WriteLn(Stroka);
   Stroka := @A;
  WriteLn(Stroka);
  WriteLn(Str[l]);
   ReadLn
end.
```

ни один из операторов, за исключением операторов Stroka:=@A; и ReadLn, недопустимы, а оператор WriteLn (Stroka), выполненный после оператора Stroka:=@A, выдаст один символ "п".

Для работы с ASCIIZ-строками в версии 7.0 используются специальные стандартные модули Strings и WinDos.

22.13.3. ***Функции*** для работы со строками с завершающим нулем (из модуля STRINGS)

22.13.3.1 Работа с динамической памятью

function StrNew(Str: PChar): PChar;

Создает (выделяет память) в динамической области копию строки Str. Результат - указатель на новую строку.

procedure StrDispose(Str: PChar);

Удаляет из динамической памяти ранее выделенную память под строку.
Str - удаляемая строка. Если Str = nil, процедура ничего не выполняет.

22.13.3.2 Функции преобразования

function StrPas(Str : PChar) : string;

Преобразует ASCIIZ-строку в строку типа string.
Результат - преобразованная строка. Str – исходная строка.

function StrPCopy (Dest : PChar; Source : string): PChar;

Преобразует строку Source типа string в ASCIIZ-строку Dest. Результат - указатель на преобразованную строку (с завершающим нулем). Размер полученной строки не контролируется.

function StrLen(Str: PChar): Word;

Определяет размер строки без учета заключительного символа с кодом 0 (он не включается в длину).

function StrCat(Dest, Source : PChar) : PChar;

Присоединяет к концу строки Dest строку Source. Результат - указатель на объединенную строку. Размер полученной строки не контролируется.

function StrCopy(Dest, Source : PChar) : PChar;

Копирует строку Source в строку Dest. Возвращает в качестве результата указатель на начало новой строки Dest. Source – копируемая строка; Dest – строка для копирования. Размер полученной строки не контролируется.

function StrECopy(Dest, Source : PChar) : PChar;

Копирует строку Source в строку Dest. Возвращает в качестве результата указатель на последний (нулевой) элемент строки (на конец строки) Dest.

function StrEnd(Str: PChar): PChar;

Получает указатель на конец строки Str (на завершающий нулевой элемент).

function StrComp(Strl, Str2: PChar): Integer;

Сравнивает две строки (символы сравниваются слева направо своими кодами).
Результат
<0, если str1 меньше str2,
=0, если строки равны,
>0, если str1 > Str2;

function StrPos(Str1, Str2: PChar): PChar;

Ищет первое вхождение строки Str2 в строку Str1. Результат - указатель на первое вхождение строки Str2 в строку Str1, либо nil, если такого вхождения нет.

function StrScan(Str : PChar; Chr : Char) : PChar;

Ищет первое вхождение символа Chr в строку Str Результат - указатель на найденный символ либо nil, если такого символа нет.

function StrRScan(Str: PChar; Chr: Char): PChar;

Ищет последнее вхождение символа Chr в строку Str .Результат - указатель на найденный символ либо nil, если такого символа нет.

function StrLower(Str: PChar): PChar;

Преобразует в строке Str прописные латинские буквы в строчные (преобразует строку к нижнему регистру).

function StrUpper(Str : PChar) : PChar;

Преобразует в строке Str строчные латинские буквы в прописные (преобразует строку к верхнему регистру).

---

function StrIComp(Strl, Str2 : PChar) : integer;

Сравнивает строки Str1 и Str2 без различия регистра символов (между прописными и строчными латинскими буквами). Результат такой же, что и у функции StrComp.

function StrLCat(Dest, Source : PChar; MaxLen : Word) : PChar;

Добавляет в конец строки Dest MaxLen символов строки Source. Если размер строки Source меньше MaxLen, копируется фактическое количество символов. В качестве результата возвращает объединенную строку. Dest – первая строка; Source – копируемая строка; MaxLen – число копируемых символов. Размер полученной строки не контролируется.

function StrLComp(Strl, Str2 : PChar; MaxLen : Word) : integer;

Сравнивает MaxLen первых символов строк Str1 и Str2. Если размер строк меньше MaxLen, сравнивается фактическое количество символов. Результат такой же, что и у функции StrComp.
Strl, Str2 – сравниваемые строки; MaxLen – число сравниваемых символов.

function StrLCopy(Dest, Source : PChar; MaxLen : Word) : PChar;

Копирует MaxLen символов строки Source в строку Dest. Если размер строки Source меньше MaxLen, копируется фактическое число символов. Результат – скопированная строка. Dest – строка для копирования; Source – копируемая строка; MaxLen – число копируемых символов.
Размер полученной строки не контролируется.

function StrLIComp(Str1, Str2 : PChar; MaxLen : Word) : PChar;

Сравнивает MaxLen первых символов строк Strl и Str2 без различия между прописными и строчными латинскими буквами. Если размер строк меньше MaxLen, сравнивается фактическое число символов. Результат такой же, что и у функции StrComp. Strl, Str2 – сравниваемые строки; MaxLen – число сравниваемых символов.

function StrMove(Dest, Source : PChar; Count : Word) : PChar;

Копирует (перемещает) Count символов строки Source в строку Dest, даже если это число больше размера строки Source (строки могут перекрываться). Результат – скопированная строка. Dest – первая строка; Source – вторая строка. Размер полученной строки не контролируется.