18. Вспомогательные алгоритмы (подпрограммы).

18.1 Три способа записи повторяющихся команд

Можно выделить 3 способа записи повторяющихся в программе команд: циклы, макро (макросы)(макрокоманды)(открытые подпрограммы) и обычные подпрограммы (закрытые подпрограммы).

Если повторяющиеся команды располагаются в теле программы в одном месте, компактно (одна за другой), то можно (нужно) свернуть повторяющиеся действия в *цикл* (если для повторяющихся действий удалось найти обобщенную запись).

$S := x_1 + x_2 + x_3 + ...$

```
s:=x1;
s:=s+x2;
s:=s+x3;        for i:=1 to n do
.....             s:=s+xi;
```

Если повторяющиеся действия сильно разнесены по телу программы (располагаются в разных местах программы), то для сокращенной записи этих повторяющихся действий принято использовать макро (макросы)(макрокоманды)(открытые подпрограммы) или обычные подпрограммы (закрытые подпрограммы). При этом повторяющиеся действия записываются (оформляются как подпрограмма) один раз, а каждый случай появления повторяющихся действий надо заменить на вызов подпрограммы.

Подпрограмм бывают двух видов:

- закрытые (процедуры и функции);
- открытые (макро).

В случае открытой подпрограммы при ее вызове (макровызове) в точку вызова (вместо вызова) компилятор вставляет текст макро (макроподстановка), причем это делается (например, препроцессором в Си) до начала процесса трансляции программы.

1

Пусть, например, надо сложить в разных местах программы 3 раза по 3 числа.

Можно для сложения каждой тройки чисел сделать простую макрокоманду типа:

имя макро — формальные параметры вызова (то, что позволяет настроить макро на работу с конкретными данными)

```
#define sum(x,y,z)  ((x)+(y)+(z))
```

тело макро (макроопределение)

что меняем — на что меняем

с учетом параметров, указанных при вызове (фактических параметров)

С использованием такого макро сложение в разных местах программы трех троек – (1,2,3), (4,5,6), (7,8,9) – можно было бы оформить следующим способом:

```
.............
...sum(1,2,3)...  -->в точку вызова будет подставлен исходный код , выполняющий 1+2+3
...sum(4,5,6)...  -->в точку вызова будет подставлен исходный код , выполняющий 4+5+6
.............
...sum(7,8,9)...  -->в точку вызова будет подставлен исходный код , выполняющий 7+8+9
```

x y z — фактические параметры вызова (соответствующих формальным x,y,z)

макровызовы

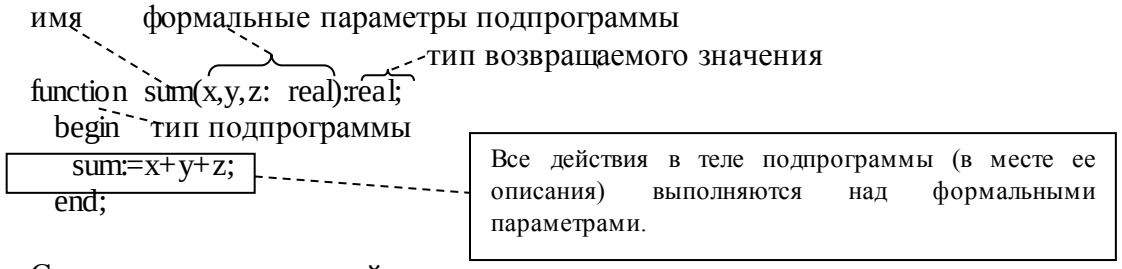
Достоинство использования макро – большая скорость работы (по сравнению с закрытыми)

Недостаток – повторение одного и того же исходного текста (кода программы) после каждой макроподстановки.

В случае закрытой подпрограммы (процедуры или функции) как и в случае открытой надо один раз описать подпрограмму и потом несколько раз вызвать. Только в отличие от открытой подпрограммы при вызове закрытой подпрограммы не происходит копирования в точку вызова

текста макро, а вместо этого из точки вызова передается управление в подпрограмму – подпрограмма выполняется, а после своего завершения возвращает управление в точку программы, следующую за точкой вызова.

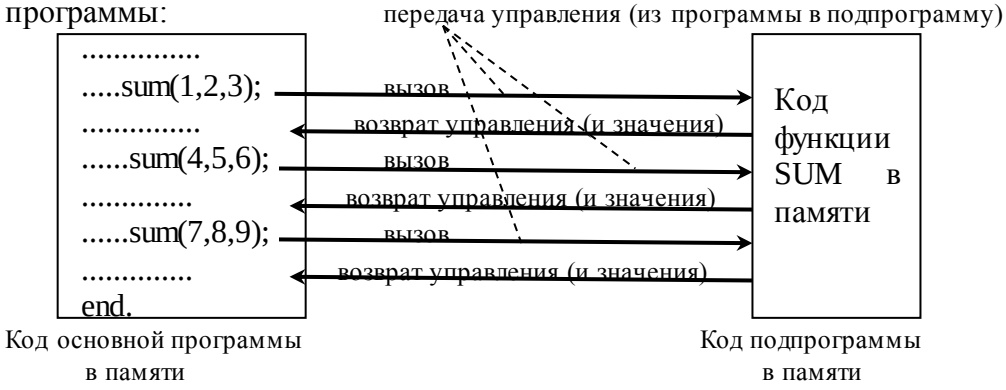
Описание закрытой подпрограммы (функции) = заголовок +блок:



С использованием такой подпрограммы сложение в разных местах программы трех троек – (1,2,3), (4,5,6), (7,8,9) – можно было бы оформить следующим способом:

```
Var
  s: real;
.....
Begin
.....
...s:=sum(1,2,3);.  --> будет вызвана sum и в точку вызова будет подставлено значение 1+2+3
                    фактические параметры вызова
.....
...s:=sum(4,5,6);.  --> будет вызвана sum и в точку вызова будет подставлено значение 4+5+6
.....
...s:=sum(7,8,9);.  --> будет вызвана sum и в точку вызова будет подставлено значение 7+8+9
.....
end.
```


Вот что происходит при описанных вызовах функции (закрытой подпрограммы) в памяти программы:



Недостаток использования закрытых подпрограмм – меньшая скорость работы (по сравнению с макро) – тратится время на передачи управления (сначала в подпрограмму, а потом назад), и на передачу параметров в/из подпрограмму .

Преимущества (достоинства) использования программ в программе:

- экономия места (в программе) - --- описываем лишь 1 раз в программе;
- экономия времени (программиста) --- 1 раз отлаживаем;
- программу легче отлаживать (легче искать, где ошибка) --- если она поделена на части;
- можно использовать для разработки программы нескольких программистов.

18.2 Понятие блока

Закрытая подпрограмма (***пока будем говорить о процедуре***) состоит из *заголовка* и *блока*.

Заголовок содержит в себе описание имени процедуры и формальных параметров (аргументы – те, что заданы - и результаты – те, что требуется получить и передать в точку вызова).

Блок процедуры содержит в себе описание так называемых локальных переменных и действия

подпрограммы (ее тело).

Определение: *Блок* - фрагмент подпрограммы или программы, содержащий в себе действия над некоторыми определенными объектами и описания этих объектов. Действия в блоке заключаются между словами *begin* и *end* (Паскаль). Описания - в секциях *Var* и *Const*.

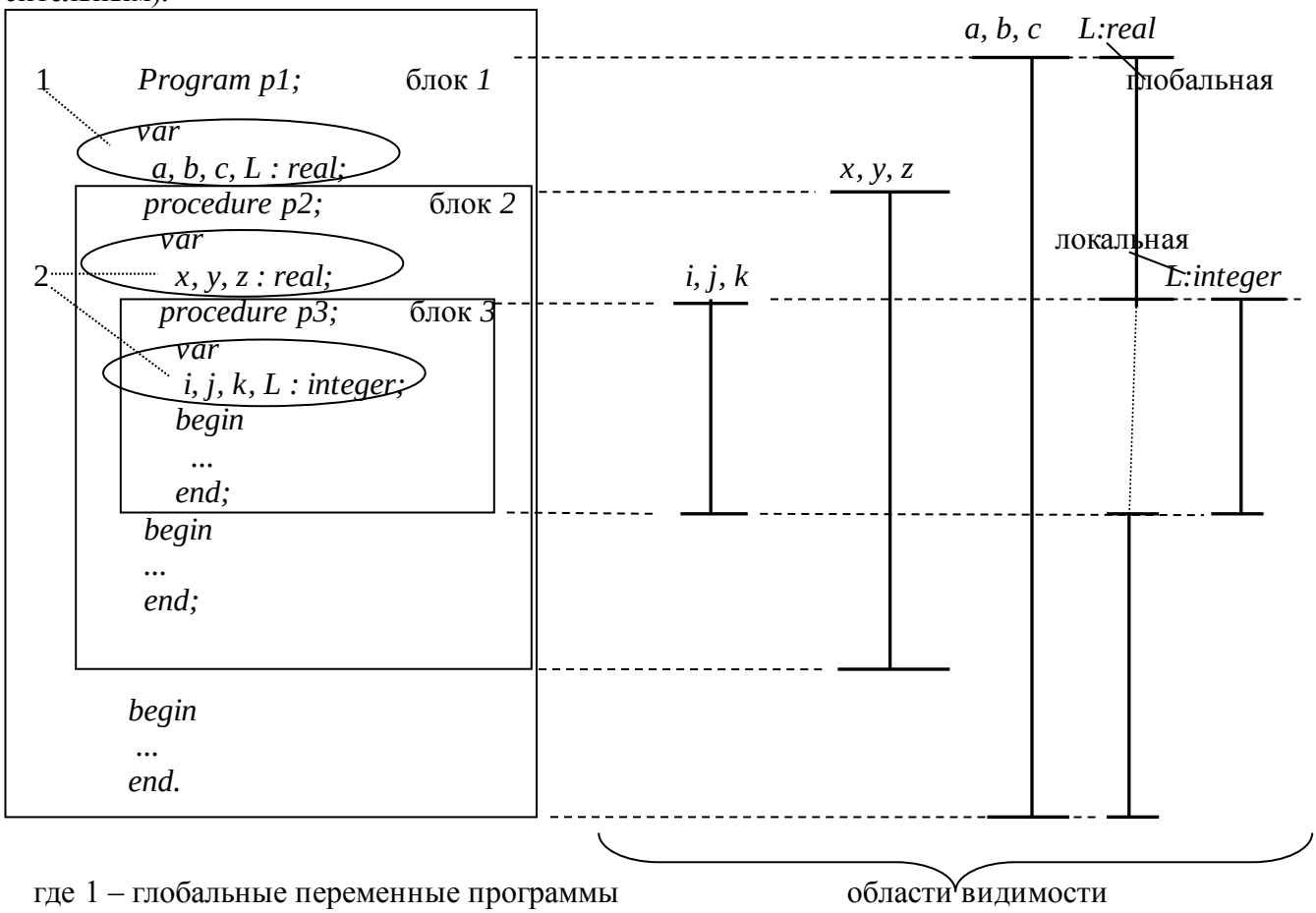
В языке Си конструкция {блок} может использоваться самостоятельно практически везде, где может встречаться обычный оператор:

```
{ ≡ begin
.....
int i;
i=1;
.....
}≡ end
```

В Паскале в отличие от Си блоки самостоятельно не используются, а только в составе подпрограмм и программ (к блоку добавляется заголовок и описания локальных для блока переменных).

Примечание. С блоком не надо путать составной оператор. Составной оператор включает только действия, а блок представляет собой законченную конструкцию, где имеются и описания и действия.

Блоки бывают **внешние** (охватывающие) и **внутренние** (подобное разделение является относительным).



где 1 – глобальные переменные программы

2 – локальные переменные процедур p2 и p3

В рассмотренном примере два внутренних блока и два внешних:

Блоки *2* и *3* являются внутренними для блока *1*.

Блоки *1* и *2* являются внешними для блока *3*.

Приведенная программа имеет, как говорят, блочную структуру.

Видно, что для описания подпрограмм в программе отведено строго определенное место (над основным разделом действий программы).

Видно, что подпрограммы м.б. вложены одна в другую (p3 вложена в p2).

18.3 Объекты подпрограммы (то, над чем выполняются действия).

Объекты подпрограммы делятся на параметры (только локальные - аргументы и результаты) и переменные (локальные и глобальные).

Параметры - это объекты, указанные в заголовке подпрограммы. Они в свою очередь делятся на аргументы (входные параметры, т.е. то, что дано, или, другими словами, то, что при выполнении подпрограммы должно быть известно) и результаты (выходные параметры, т.е. то, что требуется получить).

Переменные - это объекты, которые перечисляются в секции описания переменных:
- или в секции описания переменных в данной подпрограмме;
- или в секции описания переменных во внешнем блоке (по отношению к данному).

Объекты алгоритма делятся на ***глобальные* и *локальные***.

Локальные объекты - это объекты, которые описаны в данном блоке. Это параметры из заголовка данного блока и переменные, описанные в данном блоке.

Внешние объекты - это объекты, которые описаны во внешнем блоке (в охватывающем).

Глобальные объекты - это объекты, которые описаны в самом внешнем блоке (выше всех подпрограмм по тексту программы).

На рисунке из п. 18.2 (см. выше) для блока *3* параметров нет вообще. Все перемененные, описанные в нем самом (в блоке 3), являются локальными для данного блока. Внешними для него являются переменные, которые описаны в охватывающих блоках, т.е. в блоках *2* и *1 (то, что в блоке 1 - глобальные)*.

Для блока *2* локальные объекты – x,y,z, описанные в нем самом, а внешние (глобальные) - те переменные, которые описаны в блоке 1 – это a,b,c,L.

real

Для объектов подпрограммы различают ***область видимости* и *время жизни*.**

Время жизни - это интервал времени выполнения программы, в течение которого объект подпрограммы существует, т.е. сохраняет свое значение (место в памяти), даже, если он временно невидим (т.е. даже если к нему временно нельзя обратиться).

Объект "начинает свою жизнь" обычно с начала выполнения соответствующего блока, т.е. того блока, где он описан (где под него выделена память), и, как правило, завершает свое существование по окончании этого же блока. В зависимости от величины времени жизни различают 3 вида объектов - статические (существуют все время, пока выполняется программа), автоматические (существуют, пока выполняется блок), динамические (существуют, пока под них программистом динамически выделена память).

Есть так называемые статические локальные переменные, у которых время жизни глобальное, в область видимости - локальная. В языке Си за это отвечает класс памяти *static*, а на Паскале им соответствуют *локальные (определенные прямо в подпрограмме) типизированные константы*.

Область видимости (действия) - это часть текста программы, в которой объект, как говорят, "виден", т.е. для него известно имя и тип (в пределах области видимости возможен корректный доступ к памяти с использованием имени объекта). *Видимыми* в данном блоке считаются те объекты, которые описаны в данном блоке или в блоках, его охватывающих. То, что описано ниже данного блока (по уровню вложенности), считается невидимым в данном блоке.

локальная

В рассмотренном примере в блоке *3* видны: $i, j, k, L, x, y, z, a, b, c.$

В блоке *2* видны: x, y, z (в нем описанные), $a, b, c, L,$ (описанные в охватывающем блоке).

глобальная

В блоке *2* будут не видны те переменные, которые описаны в блоке *3*.

Примечание: В блоке *3* переменная L будет видна как *целая*, а в блоке *2* - как *вещественная*.

Область видимости и время жизни *не совпадают* для глобальных переменных: переменная может «жить», но стать временно невидима.

18.4 Свойства локальных и глобальных объектов

Свойства локальных объектов:

1. При входе в соответствующий блок локальные объекты принимают следующие значения:
   a) *формальные параметры*: принимают значения соответствующих *фактических* параметров.
   b) локальные переменные принимают неопределенные значения (не обязательно устанавливаются в ноль).
2. При выходе из блока локальные объекты теряют свое значение и становятся неопределенными.

Их "время жизни" и область видимости ограничивается пределами блока, где они объявлены.

Выводы:

1). Локальным переменным в начале выполнения программы надо присваивать начальные значения.

2). Локальным переменным в *разных блоках* можно присваивать одинаковые имена. При этом никакого влияния друг на друга не возникнет, поскольку "время жизни" таких перемененных ограничено пределами одного алгоритма (и предполагается, что в каждый момент времени активна только одна подпрограмма).

Свойства глобальных объектов:

1). Глобальные объекты живут и могут быть видны (если не перекрыты одноименными локальными) во всех блоках, для которых они являются глобальными, время их жизни – все время выполнение программы.

2). *Значения* глобальных объектов при отсутствии явного присваивания им определенных значений автоматически устанавливаются в ноль перед началом выполнения программы.

3). *Область видимости* глобальной переменной перекрывается (закрывается) областью видимости одноименной локальной переменной. То есть, область видимости глобальной переменной уменьшается на величину области видимости одноименной локальной переменной. Время жизни глобальной переменной равно времени выполнение всей программы в целом.

Примечание (к рисунку).

Из рисунка видно, что переменная L существует в программе в двух вариантах: как вещественная глобальная и как целая локальная в блоке *3,* где она описана как локальная переменная. В соответствии с правилом номер 3 в блоке *3* локальная переменная перекрывает объявление ее как глобальной переменной. Поэтому в блоке *3* переменная *L* будет видна как локальная типа *integer*, а за пределами блока 3 как глобальная типа *real.*

18.5 Выделение памяти под локальные и глобальные переменные

При выделении памяти под объекты программы следует различать три момента:

1). Когда выделяется память (в какой момент)?

2). На какое время (когда освободится)?

3). Где она выделяется (в каком месте)?

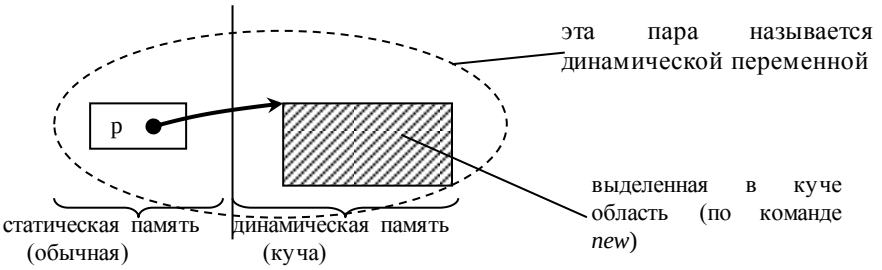
Под *глобальные* переменные память выделяется в начале выполнения программы. Выделение происходит в статической памяти программы (резервируется за глобальными переменными и константами-литералами до конца выполнения программы). Время жизни глобальных переменных - в течение выполнения того охватывающего блока, где они описаны. Освобождение памяти, занятой глобальными переменными, происходит после завершения выполнения программы. Область видимости глобальных переменных может перекрываться областью видимости одноименной локальной переменной.

Под *локальные* объекты (формальные параметры и локальные переменные) память выделяется в начале выполнения текущей (очередной) процедуры или функции в т.н. *стеке* – структура данных, в которой доступ возможен лишь к тем данным, которые находятся в вершине стека (процедура работы со стеком - LIFO) (простейший пример – детская пирамидка). В отличие от выделения памяти в *статической памяти программы* место расположения локальных переменных в стеке резервируется только на время выполнения процедуры или функции. Это место в стеке подлежит

освобождению по завершении процедуры или функции. Поэтому время жизни локального объекта - одна подпрограмма (где он описан, как локальный). Область видимости – данная подпрограмма плюс все вложенные в нее подпрограммы.

Замечание. Существуют еще так называемые динамические переменные. Они отличаются тем, что: а) память под них выделяется в куче (динамической памяти); б) выделение и освобождение памяти происходит по команде программиста (при выполнении программы).

```
Var
  p : ^integer;
begin
  new(p);
.....
end.
```



18.6 Передача параметров в подпрограммы.

В общем случае параметры у процедур и функций могут отсутствовать. В этом случае аргументы (входные параметры) и результаты (выходные параметры) передаются как глобальные переменные (объявленные в охватывающем блоке), общие для всех подпрограмм, но если надо передать значения аргументов непосредственно в подпрограмму (через заголовок – так повышается понятность программы) и получить результат непосредственно в точке вызова подпрограммы (тоже через заголовок), то при описании *процедур* и *функций* необходимо описывать и комментировать все параметры (и входные и выходные) в их заголовках. Такие описания в заголовке подпрограммы - это описания формальных параметров подпрограммы:

```
procedure p(a:byte;        { описание параметра a}
            b: char;       { описание параметра b}
            var c:integer; { описание параметра c});
```

Формальные параметры в программах на Паскале бывают двух типов:

1) . Параметры – значения (обычно это аргументы).
2) . Параметры – переменные (обычно это результаты).

Если в заголовке подпрограммы формальный параметр описан как ***параметр* - значение**, то при ***вызове подпрограммы*** значение соответствующего фактического параметра копируется в область памяти, отведенную (в момент вызова подпрограммы) под него в стеке, исходя из типа соответствующего формального параметра. Все действия в теле подпрограммы всегда записываются над формальными параметрами и глобальными переменными (если они есть), а реально выполняются при вызове над копиями соответствующих фактических параметров (эти копии – в стеке) и над глобальными переменными. Вопрос только в том, что происходит дальше с измененными (если они менялись) значениями формальных параметров.

Особенностью использования *параметров - значений* является то, что все действия (изменения), которым будут подвергнуты в теле вызванной подпрограммы эти формальные параметры-значения подпрограммы, будут относиться лишь к копиям (в стеке) соответствующих объектов вызывающей подпрограммы (указанных как фактические параметры) и при выходе из вызываемой подпрограммы в вызывающую (подпрограмму) эти изменения не отразятся на оригинальных значениях соответствующих объектов (в вызывающей подпрограмме), указанных как фактические параметры (при вызове).

При передаче ***параметра - переменной*** в стек копируется не значение фактического параметра, а ***адрес*** фактического параметра. Соответствие между формальным и фактическим параметром устанавливается таким образом, что формальный параметр становится **синонимом** (они занимают одну и ту же область в памяти) фактического. Это приводит к тому, что изменение формальных параметров – переменных вызывает одновременное изменение и внешних (глобальных) объектов, соответствующих фактическим параметрам. Поэтому по выходу из подпрограммы изменения параметров – переменных потеряны не будут (см. рис. ниже).

На Паскале *параметры - значения* описываются как обычные переменные:

Формальные параметры

procedure Summa (a : byte ; b : char ;)

Параметры - переменные на Паскале описываются после слова *var:*

procedure summa(... var c : integer; ...)

Особенность формальных и файтических параметров на Паскале:

0) в заголовке процедуры или функции описание одного формального параметра отделяется от другого точкой с запятой.
1) Каждый новый тип параметра - переменной должен сопровождаться словом *var.*

... var c : integer ; d : byte; ⟶ *d* - параметр-значение

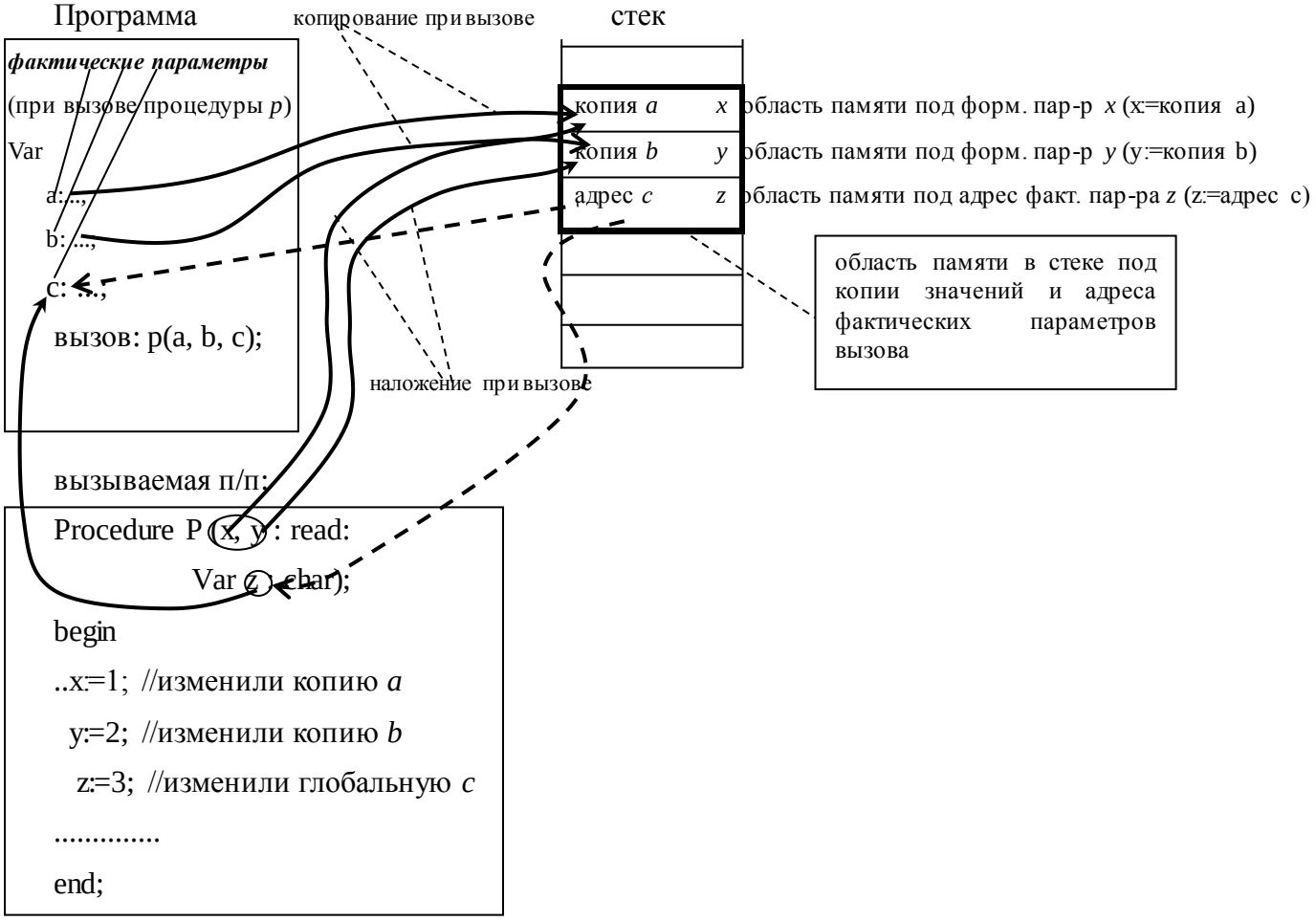
var c : integer ; var d : byte; ⟶ *d* - параметр-переменная

2) Вызов процедуры имеет следующий вид:

Summa (1, 2, l, f);

фактические параметры

3) При вызове: один фактических параметр от другого отделяется запятой (а не ‘;’).
4) При описании подпрограммы после заголовка надо ставить точку с запятой (а в Си - не надо!).
5) Соответствие между формальными и фактическими параметрами устанавливается по порядку их записи слева направо.
6) Количество и тип формальных и фактических параметров должны совпадать.
7) Для формальных параметров – переменных соответствующим фактическим параметром может быть только переменные (не значение).
8) Для формальных параметров – значений соответствующим фактическими параметрами могут быть константы, переменные, вызовы функций, выражения.
9) Имена формальных и фактических параметров обычно не совпадают.
Графически передачу параметров можно изобразить так:



ПРИМЕР. Вычислить с помощью процедуры сумму 3-х чисел:

```
Program P;
Var
 a, b, c, s: real;
```
описание глобальных переменных (фактических параметров вызова)

```
procedure summa(x,  y, z: real; //слагаемые - аргументы
                var g: real); //сумма - результат
begin
 g:=x+y+z;
end
```
описание процедуры

```
begin  //основной раздел действий
  c:=1;
  b:=2;  -------- ввод значений фактических параметров
  a:=3;
  s:=0;
  writeln('s=',s:5:2);  ---------- до вызова
  summa(a,  b, c, s);  ---------- вызов процедуры
  writeln('s=',  s:5:2);  ---------- после вызова
end.
```

В результате выполнения данной программы на экран будет выведено две строки:

s = 0

s = 6

В этом примере *x, y, z* и *g* - в заголовке процедуры - формальные параметры. При вызове процедуры в формальные параметры примут значения соответствующих фактических:

x=a, y=b, z=c, g=s

причем g и s станут синонимами.

Что произойдет, если из заголовка **удалить слово Var**? Переменная *g* становится параметром - значением.

Что будет выведено на экран? В этом случае при вызове процедуры внутри процедуры *g* - примет значение *6*. Это значение будет размещено в стеке и при выходе из процедуры там же и останется. Ввиду того, что не была установлена прямая связь между фактическим и формальным параметром, значение S останется после вызова = 0. На экран будет выведено то значение s , которое она имела до вызова процедуры, т.е. *s=0*.

18.7 Подпрограммы, возвращающие значение (функции)

В том случае, если подпрограмма возвращает одно значение, удобно использовать другую форму подпрограммы - ***функцию***.

Заголовок, тело и форма вызова такой подпрограммы отличаются от заголовка, тела и формы вызова процедуры.

На Паскале отличия проявляются в том, что

1) в заголовке вместо слова *procedure* пишется *function*
2) в теле функции должен быть оператор присваивания типа

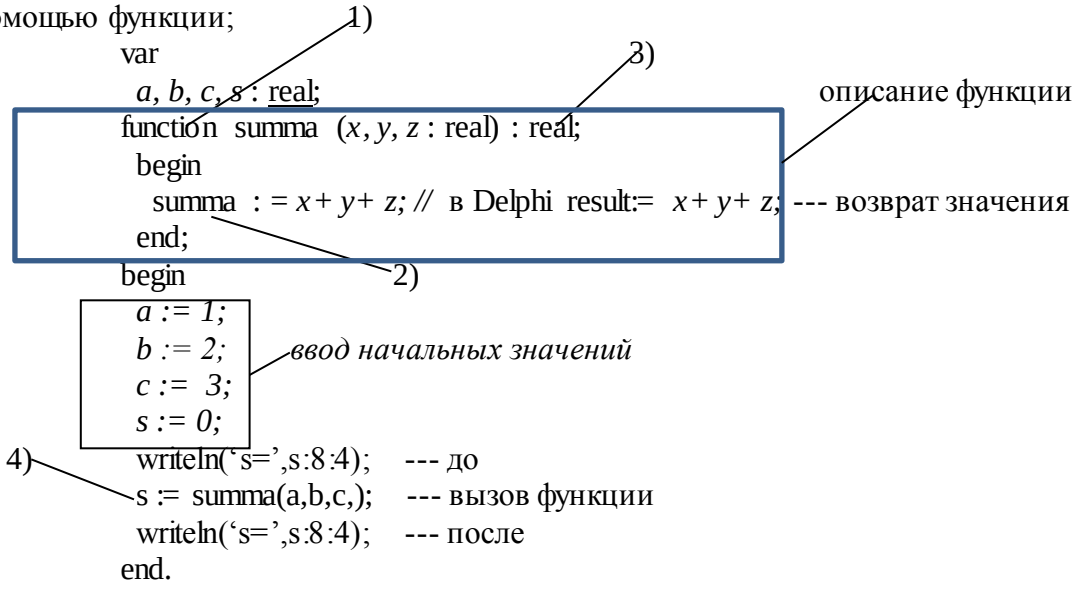
имя_функции := возвращаемое значение.

3) в заголовке справа от круглых скобок через двоеточие указывается тип возвращаемого значения
4) вызов функции возвращает вычисленное возвращаемое значение в точку вызова.

Заголовок функции имеет вид:

function имя(список_формальных_параметров) : тип_возвращаемого_значения;

ПРИМЕР. Рассмотренная задача нахождения суммы трех чисел может быть оформлена с помощью функции;




```
var
  a, b, c, s : real;
function summa (x, y, z : real) : real;
  begin
    summa : = x + y + z; // в Delphi result:= x + y + z; --- возврат значения
  end;
begin
  a := 1;
  b := 2;
  c := 3;
  s := 0;
  writeln('s=',s:8:4);    --- до
  s := summa(a,b,c,);     --- вызов функции
  writeln('s=',s:8:4);    --- после
end.
```


В том числе из примера важно выделить два момента:

1-й момент. ***Форма описания и вызова процедуры и функции отличаются***. В отличие от процедуры функция задает не просто набор действий как этап вычислительного процесса (это было в случае процедуры), а задает **правило вычисления одного значения** (с помощью определенных действий). Поэтому и форма вызова функции другая, чем у процедуры - вызов функции может размещаться или в составе какого-либо выражения или в правой части оператора присваивания (R-value).

В общем случае при **вычислении выражения, содержащего вызов функции**, выполняются следующие действия:

а) вычисляются значения выражений для фактических параметров функции;

б) значения фактических параметров присваиваются соответствующим формальным параметрам ;

в) выполняются действия в теле функции над формальными параметрами (с переданными им значениями фактических параметров) и вычисляется возвращаемое функцией значение;

г) возвращаемое значение подставляется в выражение, из которого (в котором) осуществляется обращения к функции;

д) вычисляется значение выражения с использованием возвращенного функцией значения.

2-ой момент В теле функции обязательно должна быть хотя бы одна строчка в которой **имени функции присваивается возвращаемое значение.** (в Delphi вместо этого д.б. строка вида:

result:= возвращающие значение;)

Обычно делают две следующих ошибки при записи функции:

1). *function f(x : real): real;*
begin
*f(x) := x*3.14*2;*
end;

Должно быть имя := значение

2). *function g(a: byte): real;*
Begin
g:=25;
*g := g * a;*
end;

Нет параметра при вызове функции

В первом случае сделано две ошибки:

1-я синтаксическая: Нарушен синтаксис, т.к. для возврата из функции значения должен быть записан оператор присваивания в форме имя := значение.

2-я семантическая: для f(x) -компилятор посчитает, что вызов функции стоит слева от знака присваивания.

Во втором случае ошибка (синтаксическая) заключается в том, что имя функции используется в выражении в правой части оператора присваивания так же как имя переменной, т.е. без списка параметров, что не совпадает с тем, что записано в заголовке этой функции (там написано, что функция с параметром).

Примечания.

1. *Возвращаемым значением* может быть значение ***простого*** *типа или* ***строкового*** *типа или* ***ссылочного*** *типа.* Если в теле функции **отсутствует** оператор присваивания имени функции какого-либо значения, то возвращаемое значение функции считается **неопределенным**.
2. Часто бывает так, что необходимо возвращать разные значения функции в зависимости от наступления или не наступления различных ситуаций. В этом случае в теле цикла появляется **несколько операторов присваивания имени** функции возвращаемого значения. В таких функциях трудно понять, какой из них на самом деле возвращает в данный момент значение, и поэтому такие функции труднее отлаживать. Целесообразно придерживаться следующего *правила*: если необходимо вычислить возвращаемое значение в нескольких местах тела функции, то нужно ввести специальную вспомогательную переменную для такого присваивания, а затем непосредственно перед завершением тела функции, записать оператор присваивания, где слева - имя функции, а справа - имя вспомогательной переменной:

```
function  fact(n:byte):longint;
var
  i:word;       //параметр (счетчик) цикла for
  f:longint;    //вспомогательная переменная
begin
  f:=1;
  for i:=1  to n do                                  имя функции
    f := f * i; //ошибкой была бы запись вида fact := fact * i
  fact := f;
end;
```

18.8 Особенности использования процедур и функций в Турбо Паскале

1. Если формальным параметром процедуры или функции является *параметр - значение*, то при вызове этой процедуры или функции соответствующим фактическим параметрам может быть любое выражение (частный случай выражения: переменная, константа). Если формальным параметром является *параметр-переменная*, то при вызове процедуры или функции фактическим параметром может быть **только переменная эквивалентного типа**.

2. Типом любого формального параметра может быть либо стандартный тип, либо тип ранее определенный в секции описания типов. (Нельзя конструировать типы в заголовках процедур и функций).

Вывод: Если надо передать нестандартный тип в процедуру или функцию, то этот тип следует до (выше) описания подпрограммы определить (описать в секции определения типов), а в заголовке подпрограммы должна быть только ссылка на этот тип.

хотя можно: *Function summa (a: array of byte) : integer;*

неверная запись — открытый массив

Function summa (a: array [1..10] of byte): integer;

надо

```
Type
  t = array [1..10] of byte;
function summa (a : t) : integer;
```

Требование запрета конструирования типов в заголовке процедуры и функции, введено для обеспечения совместимости типов формальных и фактических параметров-переменных, поскольку по правилам Паскаля то, что описано в разных местах, имеет разные типы.

3. *Тип параметра значения* может быть любым, кроме файлового. *Тип параметра переменной* может быть любым, включая файловый.

3 **Вывод:** Если необходимо в процедуру или функцию передавать файл, то он должен быть передан как параметр переменная.

4. Фактически при передаче параметров в процедуру и функцию *по значению*, значения этих параметров копируются в стек, что приводит к потерям времени и памяти на такое копирование. Если копируемые значения являются сложными (записи, большие массивы и т.д.), то чтобы избежать потерь времени от такого копирования есть две возможности:

1) Можно передавать длинные фактические параметры не как параметры-значения, а как *параметры-переменные* (по адресу). В этом случае в стек копируется не значение, а только адрес начала параметра. Неудобства этого способа состоит в том, что а) во время работы подпрограммы значение фактического параметра может быть заменено, что м.б. нежелательно, б) ухудшается понятность программы (по адресу принято передавать только результаты, а мы передаем входные данные).
2) Для избавления от этого недостатка можно использовать т.н. *параметры-константы*. *Параметры-константы* похожи на параметры-переменные тем, что передаются *по адресу*, т.е. в стек копируется адрес. Отличие состоит в следующем: в заголовке процедуры или функции вместо слова *var* указывается слово *const*

   (*const a : byte*)

   Компилятор при использовании параметра-константы жестко следит за тем, чтобы значение этих параметров не менялось.

5. Можно использовать *параметр - переменную или параметр - константу без типа.*

(...Var a; const b; ...)

Они передаются только по адресу. Их использование позволяет избавиться от несовместимости типов формальных и фактических параметров. В этом случае фактический параметр может быть параметром любого типа. Сложность их использования состоит в том, что в

теле процедуры или функции их нельзя использовать без операции приведения типа.

6. Можно в качестве формального параметра использовать т.н. "*открытые" массивы* - *массивы без границ.* *A : array of char;*

В качестве фактического параметра для формального параметра типа открытого массива может выступить или статический (обычный) массив или динамический массив (в куче).

Наличие таких формальных параметров (открытых массивов) позволяет в качестве фактических параметров использовать в подпрограмме массивы *любого размера* (только тип элементов должен быть таким же). Конкретное число элементов массива определяется при вызове подпрограммы (на основании того, что передается в качестве фактического параметра). Имеется возможность в подпрограмме просмотреть нижнюю и верхнюю границы массива. Для этого используются стандартные функции *low* и *high*. Для открытых массивов *low* равно *0*, *high* возвращает число элементов массива – 1. Рассмотрим пример:

```
Var z:array[1..10]  of integer;   //массив, передаваемый в функцию
Procedure P(x:array  of integer);
Var
  i : byte;
Begin
  for i := low(x)  to high(x)  do
    writeln(x[i]);
 …
end;
begin
 …
 P(z);
 ...
end.
```

вернет low(x)

здесь элементы x[0] ≡ z[1]
x[9] ≡ z[10]

вернет high(x)

7. Можно использовать так называемые *локальные статические переменные*, которые отличаются от обычных локальных переменных глобальным временем жизни.

Обычные локальные переменные имеют *время жизни = времени выполнения* процедуры или функции (где они объявлены), а статические локальные переменные имеют время жизни, равное времени выполнения всей программы (правда у них ограничена область видимости). Это достигается за счет того, что память под статические локальные переменные распределяется не в *стеке,* а в *статической части программы*. Описываются статические локальные переменные в теле процедуры или функции как ***типизированные константы***. Один из возможных вариантов использования статических локальных переменных - ***подсчет числа вызовов процедур или функций***.

```
Function summa (...) : byte;
const
 count : byte = 0;
begin
  count := count + 1;
end;
```

на Си: static unsigned char c = 0;

класс памяти

это действие выполняется лишь один раз за все время выполнения программы. А именно – при первом вызове функции.

8. Есть следующие возможности «внезапного» выхода из процедур или функций (в циклах есть break и continue):

exit; - обеспечивает прекращение выполняемой процедуры или функции и возвращает управление в точку вызова (одноуровневый выход - на один уровень выше по цепочке вызовов).

Halt(n); - процедура используется для преждевременного выхода и из подпрограммы и изо всей программы, на каком бы уровне вложенности подпрограмма бы не вызывалась.

n -код возврата; принято, если *n = 0* то завершение было нормальным; если *n > 0* то программа завершена некорректно. Код возврата анализируется или порождающим процессом с помощью DosExitCode (из модуля Dos), или помощью проверки ERRORLEVEL в командном файле.

18.9. Побочный эффект (side effect)

Под термином ***"главный эффект"***, понимается возврат функцией одного значения или возврат процедурой вычисленных значений (результатов) через параметры-переменные. Обычно предполагается (с точки зрения ясности работы программы), что все действия, которые выполняются в процедурах или функциях, соответствуют главному эффекту, т.е. предполагается, что главный эффект является единственным и за пределами подпрограммы невозможно обнаружить никаких других последствий ее выполнения. Однако по аналогии с фармакологией наряду с главным эффектом часто бывает и побочный.

Побочным эффектом называется такой эффект, который (при вызове процедуры) **проявляется вне тела** процедуры или функции, и заключается в выполнении действий, отличных от тех, которые производят главный эффект.

в C# нет глобальных переменных (но есть статические)

Наиболее часто **побочный эффект проявляется в следующих вариантах** (4 варианта):

1 вариант. **Изменение** в теле процедуры или функции значений глобальных параметров. При этом использование глобальных переменных само по себе не является побочным эффектом. Побочных эффектом является их **изменение** в теле процедуры или функции. При этом предполагается, что главная (вызывающая) программа и подпрограммы обмениваются между собой данными через общие области памяти (глобальные переменные) - как через доску объявлений.

ПРИМЕР.

глобальные переменные

```
var
 x, y : real;
function A (B : real) :real;
 begin
   A:= PI * B;
   x:= 1.5;
 end;
   begin
    x:= 2;
    writeln ('x =', x);
    y:= A(x);
    writeln ('y =', y, ' при x = ', x);
end.
```

стандартная функция

пользовательская (наша) подпрограмма

тело (код) программы (основной раздел действий)

Вызов подпрограммы

Побочный эффект заключается в изменении значения глобального параметра *X*. Вначале выполняется *X := 2*, далее при вызове функции *A X* изменяется на *1.5*. После этого вызова в главный алгоритм будет возвращено новое значение глобального параметра *X = 1.5*. Это значение и будет выведено на экран последним оператором. Главное, глядя только на главную программу невозможно понять, почему значение X изменилось.

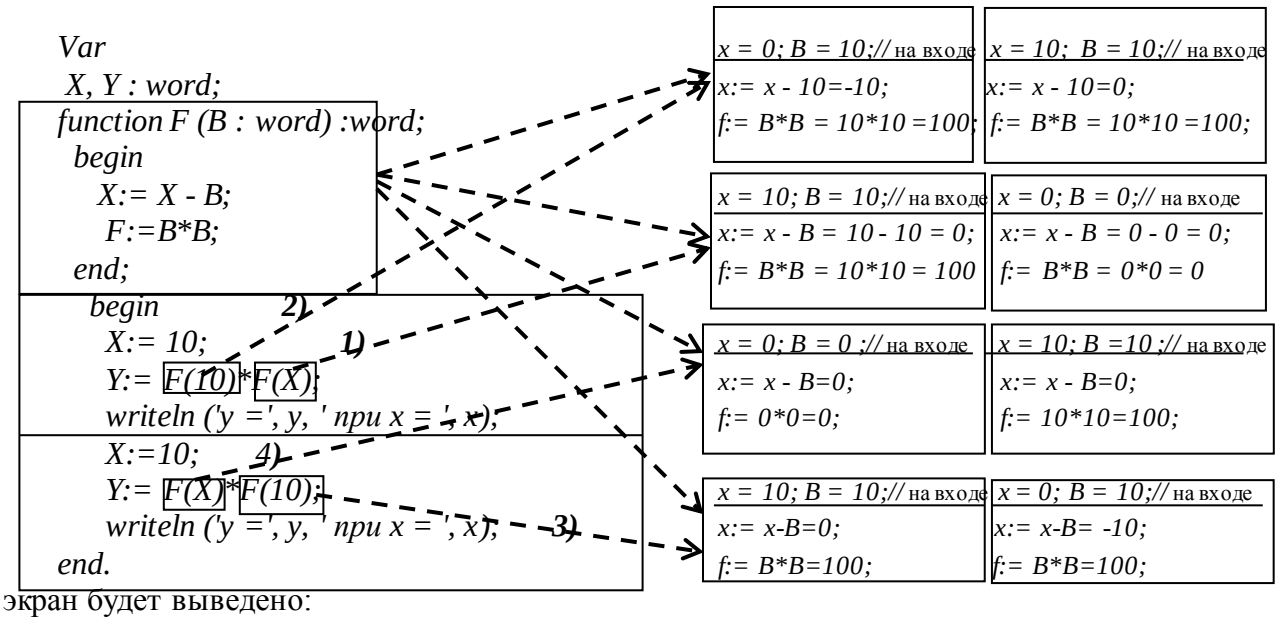
2 вариант. Необоснованный вызов в теле процедуры или функции процедур ввода–вывода, где этого никто (из тех, кто ее запускает) не ждет (обычно вывод выполняется в вызывающей части программы - при получении результатов - а не в вызываемой функции, где требуется только вычислять). Например, из текста программы может быть видно (ожидаться), что вывод результата должен быть только после вызова определенной подпрограммы (p1), однако при вызове другой подпрограммы (p2) тоже наблюдается вывод чего-то. Возникает вопрос, а что выводится? В результате ухудшается понятность программы.

Замечание. Обычно нежелательно выполнять вывод в программе (не в вызывающей функции - в main() как в Си). Лучше создать отдельную «печатающую» подпрограмму, выводимые ею строки сообщений оформить в виде массива строк, а номер выводимой строки (сообщения) передавать в «печатающую» подпрограмму (при ее вызове) в виде входного параметра (номера строки).

3 вариант. Изменение значений фактических параметров, которые не должны изменяться (по логике человека, использующего программу). Например, функция не может по определению изменять ничего кроме возвращаемого значения. Если же функция изменяет какой-либо из параметров (из заголовка), то это - побочный эффект.

4 вариант. Если вызов функции, использующей побочный эффект, находится в выражении, то значение этого выражения может **зависеть от порядка следования** в нем операндов.

ПРИМЕР:



На экран будет выведено:

y=10000 при x= -10

y=0 при x= 0

Объяснение: при включенной оптимизации при вычислении значения выражения типа A1*A2 (оно будет преобразовано в обратную польскую запись вида A1A2* - сначала операнды потом операция) на Паскале в стек вначале попадет A1, потом A2, при этом A2 окажется в вершине стека и будет вычислено и извлечено из стека первым (операнды операции будут вычисляться в порядке обратном порядку помещения их на стек, т.е. справа налево).

При наличии в программе побочных **эффектов невозможно предсказать по тексту** главного алгоритма, какие реальные действия в программе выполняются, если не анализировать ход выполнения каждой процедуры и функции, что **ухудшает понимание программы**.

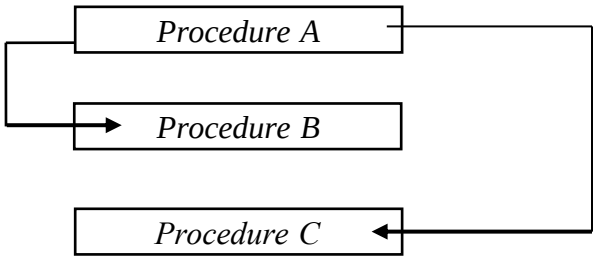
Вывод: Можно использовать побочный эффект для определенных целей (например, для обмена данными между главной программой и подпрограммами), но надо строго оговаривать все такие случаи.

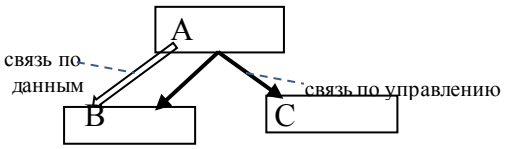
18.10 Опережающее определение процедур и функций.

При уточнении начального алгоритма методом сверху вниз (при разбиение исходной задачи на подзадачи) в программе будут встречаться описания подпрограмм, которые располагаются ниже описания подпрограмм, которые их вызывают. Так происходит потому, что по правилам разработки алгоритма методом сверху вниз подпрограммы (вызываемые - новые) в программе могут появляться только после того, как по мере уточнения (исходной программы) в этих (новых) подпрограммах появляется необходимость, т.е. новые подпрограммы будут размещаться по тексту программы после (ниже) описания тех подпрограмм, часть действий в которых оформляется как подпрограммы (вызываемые).

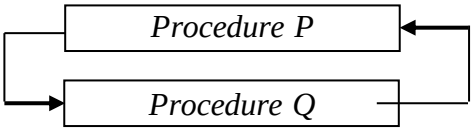
Рассмотрим два случая, когда возникает необходимость использования опережающего описания процедур и функций.

1-ый случай: пусть при уточнении подпрограммы (процедуры) *A* было решено, что она уточняется с помощью вызова двух подпрограмм (процедур) *B* и *C*.





2-й случай: сложная (взаимная, скрытая) *рекурсия*



Каждый из этих двух рассмотренных случаев связан с необходимостью определения порядка описания в программе используемых процедур: *A, B, C,* (1-й случай), *P*, *Q* (2-й случай).

Проблема в том, что компилятор рассматривает текст программы сверху вниз. При этом все используемые (вызываемые) процедуры, функции, константы, переменные и метки должны быть известны (описаны) *до момента их использования*. В противном случае компилятор сообщит, что *используется неизвестное имя,* и воспримет это как ошибку. В первом случае, если описать процедуры в том порядке, в котором они появлялись в поле зрения разработчика программы (A – B – C) , то в процедуре A неизвестными окажутся имена B и C. Во втором случае неизвестными окажутся всегда либо P либо Q.

Есть **две возможности устранения** подобных ошибок:

1). Изменить порядок описания процедур в тексте программы.

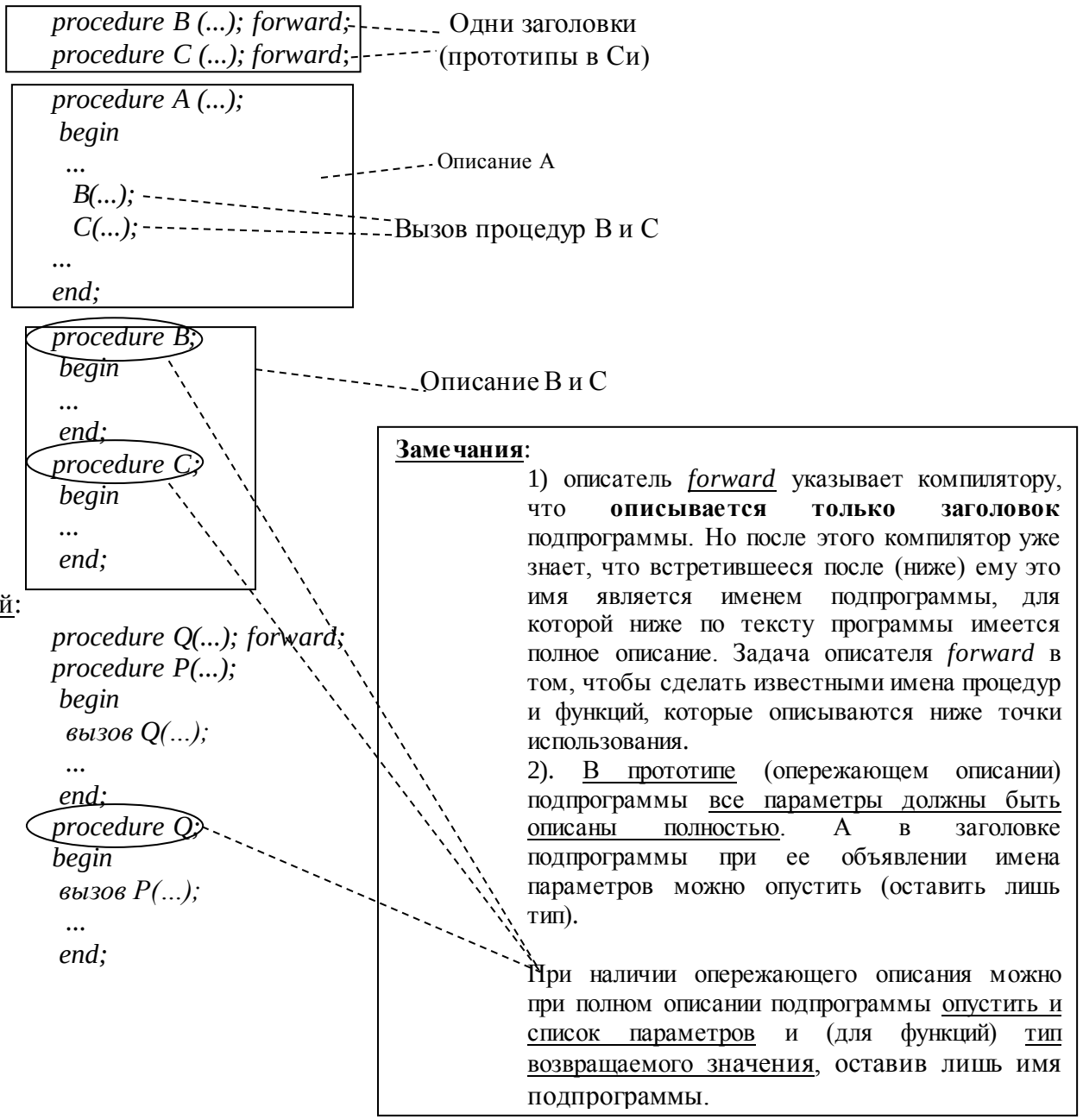
В *1*-ом случае, чтобы устранить ошибку нужно поместить описание процедур *B* и *C* перед описанием процедуры *A*.

Во *2*-ом случае перестановка *P* и *Q* не к чему не приведет.

2) Для устранения этой ошибки для процедур и функций есть описатель ***forward***.

При использовании описателя *forward* оба примера были бы записаны следующем образом:

1-й случай:



2-й случай:

```
procedure Q(...); forward;
procedure P(...);
 begin
  вызов Q(...);
  ...
 end;
procedure Q;
begin
 вызов P(...);
 ...
end;
```

Замечания:

1) описатель *forward* указывает компилятору, что **описывается только заголовок** подпрограммы. Но после этого компилятор уже знает, что встретившееся после (ниже) ему это имя является именем подпрограммы, для которой ниже по тексту программы имеется полное описание. Задача описателя *forward* в том, чтобы сделать известными имена процедур и функций, которые описываются ниже точки использования.

2). В прототипе (опережающем описании) подпрограммы все параметры должны быть описаны полностью. А в заголовке подпрограммы при ее объявлении имена параметров можно опустить (оставить лишь тип).

При наличии опережающего описания можно при полном описании подпрограммы опустить и список параметров и (для функций) тип возвращаемого значения, оставив лишь имя подпрограммы.

18.11 Рекурсия и итерация.

Рекурсия - это определение понятия самого через себя.

Например, идентификатор ::= <буква>|<идентификатор><буква> |<идентификатор><цифра>

В случае процедур и функций рекурсия используется в виде вызова функции самой себя в собственном теле (в разделе действий). Это делается обычно для последовательного понижения размерности (сложности) задачи, пока решение задачи не станет тривиальным (простым). Запись решения задачи в виде последовательности рекурсивных вызовов функции обычно выглядит более понятно (более просто) по сравнении с обычным (нерекурсивным) (итерационным) решением.

Например, одно и то же вычисление факториала можно сделать несколькими способами:

Нерекурсивное (итерационное) решение:

$$fact(n) = 1*2*3*4*...*n = \prod_{i:=1}^{n} i$$

Рекурсивное решение имеет вид:

$$fact(n) = \begin{cases} 1, \text{ при } n = 0 \\ fact(n-1)*n \text{ при } n > 0. \end{cases}$$

Понижение размерности

$4! = (3!)*4$

Примеры итерационных вычислений, например, при вычислении значений многоместных (n-арных) операций уже рассматривались выше (когда роль идет о вычислении n – арных операций)

Вычисление **с помощью итерации**:

```
                    Предопределенная
                       константа
Type                        |
        natural=0.. maxint;
function fact(n: natural): Longint;
var
  i: integer;{параметр цикла}
  f: Longint;{факториал}
begin
   f:= 1;
   for i:= 1 to n do
     f:= f * i;
   fact := f;
end;
```

переменная где формируется промежуточный результат функции

Рекурсивное определение той же функции:

```
type
  natural=0.. maxint;
function fact(n: natural): Longint;
begin
  if n = 0 then
    fact := 1
   else fact := n * fact(n-1); end;
end.
```

вычисление значения этого выражения будет производиться на рекурсивном возврате (после рекурсивного вызова)

Процесс рекурсивного вычисления значения в данном случае факториала организуется в соответствии с данным определением. Этот процесс сводится к последовательному вызову функции самой себя каждый раз с новым значением аргумента (цепочке рекурсивных вызовов). При этом каждый раз при каждом таком рекурсивном вызове в стеке откладывается копии всех локальных параметров процедуры или функции:

В общем случае **при вызове любой процедуры или функции** происходят следующие действия:

1). В стеке выделяется память и сохраняются значения локальных объектов подпрограммы (фактические параметры вызова и локальные для вызываемой подпрограммы переменные, а для функции - еще и возвращаемое значение);

2). Запоминается адрес возврата в вызывающую подпрограмму (или программу);

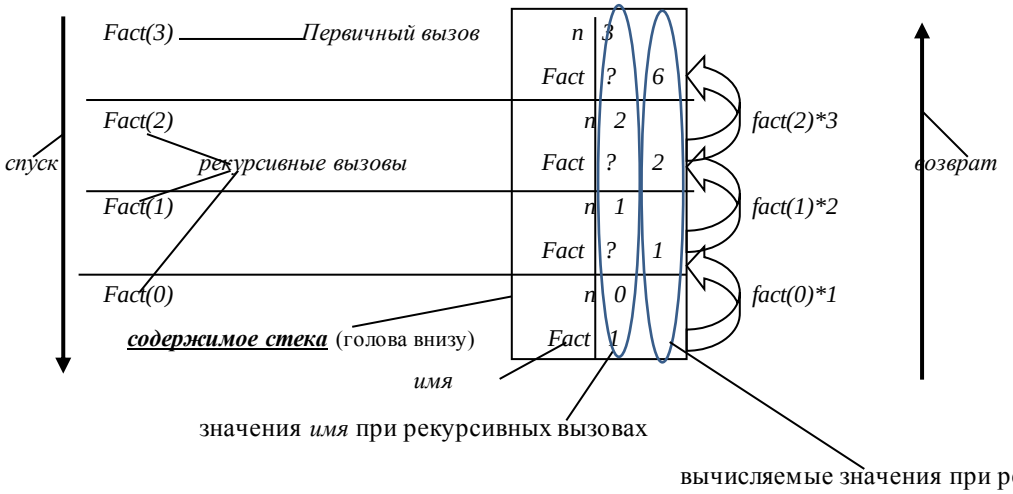
3). Управление передается вызванной подпрограмме.

В некоторый момент выполнения программы в стеке может присутствовать несколько (возможно даже одинаковых по имени) групп локальных объектов, соответствующих цепочке вызванных, но не завершенных подпрограмм. В каждый момент времени доступ возможен только к той группе объектов, которая включена в стек последней и находится в голове стека. Эта группа локальных объектов соответствует текущей вызванной подпрограмме. При завершении вызванной подпрограммы ее локальные объекты удаляются из вершины стека, и в голове стека оказываются (и становятся доступными) локальные объекты вызывающей подпрограммы.

Рассмотрим процесс рекурсивного определения более подробно на примере вызова y:=*fact(3):*

*Fact(3) → 3*Fact(2) → 3*(2* Fact(1)) → 3*(2*(1* Fact(0)))*

Нерекурсивный вызов Fact(0)=1 завершает цепочку рекурсивных вызовов.



Такой процесс последовательного вызова функции самой себя (естественно с разными значениями аргумента) называется *рекурсивным спуском*. Вспомним, что в процессе рекурсивного спуска при каждом рекурсивном вызове в стеке выделяется память под копию каждого локального объекта подпрограммы. Процесс рекурсивного спуска в данном случае будет продолжаться до тех пор, пока выражение в правой части оператора присваивания для вычисления возвращаемого значения не примет вполне определенный вид (в этом случае функция сама себя не вызывает). То есть в данном случае конец наступит, когда при очередном рекурсивном вызове аргумент станет равным *0*. После этого процесс вычислений начнет раскручиваться в обратную сторону - будет происходить *рекурсивный возврат*. Этот процесс обычно сопровождается ***освобождением памяти*** (в стеке), занимаемой очередной копией локальных объектов процедур или функций.

В общем случае действия в теле рекурсивных процедур и функций (у нас - в правой части оператора присваивания) могут выполняться либо на рекурсивном спуске (до рекурсивного вызова) , либо на рекурсивном возврате (после рекурсивного вызова). В нашем случае действие (вычисление факториала) происходит в процессе рекурсивного возврата и имеет следующий вид:

то, что получено после рекурсивного вызова (после возврата)

*fact(n) = fact(n-1) * n.*

Количество вызовов функцией самой себя при рекурсивном спуске называется *глубиной рекурсии*.

Необходимыми условиями для успешного завершения рекурсии процедур или функций являются следующие:

1). необходимо точно определить ситуацию, при которой должен завершиться рекурсивный спуск. В данном случае такая ситуация имеет вид: *n =0* - один признак у этой ситуации Но может быть и более сложный случай.

2) необходимо предусмотреть наличие в теле рекурсивной подпрограммы тех действий, которые должны приближать наступление указанной выше ситуации.

Считается, что каждую **рекурсию можно заменить *итерацией***, применив ***цикл***. Рекурсивное определение алгоритма обычно более просто и компактно, причем использует более простые действия (вместо возведения в степень - умножение). Можно считать, что при *рекурсивном* определении говорится, что нужно сделать, но не говорится как. При *итерационном* наоборот: строго формулируется, *что и как нужно сделать*.

У рекурсивной процедуры или функции имеются следующие **недостатки**: на выполнение рекурсии процедуры или функции требуется большое количество машинного времени и памяти.

Времени требуется больше потому, что многократно вызываются процедуры и функции, т.е. тратиться время на то, чтобы выйти из подпрограммы и снова войти в нее. **Память** тратится больше из-за того что при каждом рекурсивном вызове резервируется память в стеке для каждого локального объекта процедуры или функции. В итоге при большой глубине рекурсии стек может переполниться. Для контроля переполнения стека используется директива {$S+}. Для сокращения памяти в стеке (при этом увеличатся затраты памяти в сегменте данных) при рекурсивном вызове (при предполагаемом большом числе рекурсивных вызовов), целесообразно: а) использовать статические локальные переменные (на Паскале это типизированные константы, память под них выделяется не в стеке, а сегменте данных) б) длинные данные по возможности передавать по адресу - как параметры-переменные..

Замечание. Не каждый алгоритм может быть успешно реализован с помощью рекурсии. Успешно алгоритм может быть реализован рекурсивно, если удалось определить (придумать) 2 ситуации: 1) в которой процесс рекурсивного спуска будет продолжаться 2) в которой - завершится.

Пример: рекурсивная процедура поиска в массиве.

```
Type
  TR = array[1..100] of integer;
  TN = word;
Var
  R : TR; //глобальный массив
  Нашли: boolean; //глобальный признак
  A: TN; //глобальная переменная - число эл-тов массива
  Nomer: TN; //номер, где нашли
procedure p(N: TN);
 begin
   .....
  if (R[i] = Эталон
     then
        begin
          Нашли := True;
          Nomer := i;
        end
     else
         if  N > 1
            then
               P(N-1); //рекурсивный вызов
 end;
begin
 Нашли := False;
 A := 100;
 P(A);
 if Нашли = True
   then
      writeln('Нашли  в ', Nomer, ' элементе');
end.
```

18.12 Процедуры и функции как параметры.

Существует класс задач, где *исходными данными* (аргументами) для программ (подпрограмм) являются не переменные и константы, а *функции* и *процедуры*. К таким задачам относятся например задача создания неких универсальных подпрограмм типа, например, универсальной подпрограммы построения графика нескольких разных функций, определенных на нескольких одних и тех же интервалах. Вид такой подпрограммы не должен зависеть от вида функции, используемой для вычисления значения на интервале.

Если бы такую универсальную подпрограмму построения графика делать без передачи ей функции (для которой строить график) как параметра, то необходимо было бы сделать две вещи:

1) распознавать, для какой конкретно функции (интервала) надо строить график, например, используя меню вида: «выберите номер с именем нужного вида функции:

```
1: f1
2: f2
.....
```

После этого (ввода нужного номера - пусть будет имя *nomer*) вызвать оператор case вида:

```
case nomer of
    1: график (график (..., x*x, ...);
    2: график (график (..., sin(x*x), ...);
    3: график (график (..., ln(x*x), ...);
    ..................................................
```

Таких строк нужно было бы столько, сколько имеется функций, для которых надо строить график.

2) Для каждого конкретного вида функции надо было бы использовать свой специфический вызов этой функции для определения положения точки на графике и вывода ее (точку) на график.

Если в функцию построения графика передавать как параметр функцию, вычисляющую значение на интервале, то в общем виде заголовок этой функции (построения графика) можно было бы записать так:

функция *f* как формальный параметр

```
Procedure график(x : real; function f(x:real) : real);        (на Паскале так писать нельзя)
```

Костяк универсальной процедуры построения графика может иметь следующий вид:

```
function f1 (...):real;
    begin
     ...
    end;
function f2 (...):real;
    begin
     ...
    end;
function f3 (...):real;
    begin
     ...
    end;
procedure график (x:real; function f(x: real)): real);
    begin
     ...
     writeln(.....  , f(x), ...);
    end;
```

Описания функций, для которых надо строить графики

у процедуры 2 формальных параметра

для построения графика функции f1(x) вызов процедуры имел бы вид: график(x, f1(x));

В теле процедуры вывода графика используется один и тот же оператор вывода для всех функций (*f1, f2, f3).*

Конкретный вид функции указывался бы при вызове. В каждом конкретном случае при выполнении строчки

```
wtiteln(..., f(x), ...)
```

вызывалась бы та функция, которая передавалась бы в качестве фактического параметра.

На Паскале в качестве формальных параметров допускается использовать имена процедур или функций. При вызове подпрограммы в качестве фактических параметров процедур или функций можно (нужно) указывать имена соответствующих процедур или функций.

Но на Паскале **нельзя** описывать функцию как параметр в той форме, как это сделано выше (нельзя определять сложный тип прямо в заголовке). С этим мы уже сталкивались при передаче массивов как параметров - нельзя конструировать тип массива в заголовках процедур и функций..

Procedure f(a : array[1..10] of char); - для массива

Procedure график(x:real;...
function f(x: real: real);...); — неверно

На Паскале для этого случая нужно объявить **процедурный тип**.

Замечание. языке Си для этого используется указатель на функцию:

int f(int); ----- функция

int (*f)(int); -- указатель на функцию

Type
func = function(x: real): real; — Имя функции не указывается

Справа от "=" фактически приводится заголовок функции, только без имени.

Процедурный тип определяет, какой вид подпрограммы (процедуру, функцию) можно использовать в качестве параметра и с какими параметрами должна вызываться эта подпрограмма.

Правильный заголовок процедуры построения графика в нашем случае должен иметь следующий вид:

```
procedure  график (x: real; f: func);
 begin
  ...
  writeln (... , f(x), ...);
 end;
```

f := f1; или f:=@f1;

Парамерт (переменная) процедурного типа *f* **может получить** в качестве значения имя (адрес) функции, например *f1 (соответствующего по типу и числу параметров вида)*. Выполнения функции *f1* при таком присваивании не происходит (только адрес присваивается), но после присваивания вида f := f1 при вызове вида *f(x)* вызываться будет *f1(x)*.

В основной программе при таком заголовке вызов процедуры *график* может иметь вид:

график(a, f1); или график(a, f2); или график(a, f3), где a - аргумент функций f1, f2 и f3..

Установлены следующие **правила** использования подпрограмм в качестве параметров:

- правила **совместимости**: процедурные типы (и соответствующие им подпрограммы - как будущие фактические параметры) для совместимости по присваиванию должны иметь одинаковое число формальных параметров, а параметры на соответствующих позициях в заголовках, должны иметь эквивалентный тип. Кроме того, для функций должны совпадать типы возвращаемых значений;
- в Паскале, процедура или функция в своем теле (разделе действий) может обращаться только к той процедуре или функции, описание которых располагается перед описанием данной (вызывающей) процедуры или функции.
- они не должны объявляться внутри других процедур или функций;
- они не должны быть стандартными процедурами или функциями;
- для установки правильных связей между вызывающими и вызываемыми подпрограммами вызываемые процедуры или функции (передаваемые как параметры) должны быть скомпилированы с использованием **дальней модели вызовов** (для возможности вызова подпрограммы из других модулей - за пределами того модуля, где она описана) с формированием их полного адреса, что достигается двумя путями:

1-й путь: они должны компилироваться с директивой компилятору {SF+}

```
{$F+}
function f1(...): real;
 begin
 ...
 end;
{$F-}
```

Перед и после описания функции помещается директива

2-й путь:

```
function f1(...): real; far;
```

явно указана необходимость использования дальнего вызова в заголовке функции.

18.13 Директивы подпрограмм

Следующие директивы дают дополнительную информацию транслятору о размещении подпрограмм:

1. **Директива FORWARD** (смотри опережающее описание подпрограмм)
2. **Директивы FAR и NEAR**

Как правило, компилятор Turbo Pascal автоматически выбирает адресацию к подпрограмме:

- если подпрограмма находится в одном файле с основной программой, то она компилируется с "ближним" (near) адресом входа и возврата. При ближней модели вызовы выполняются быстрее, чем в дальней, но область действия ближних вызовов ограничена только тем модулем, в котором описана вызываемая подпрограмма; директива подпрограммы NEAR эквивалентна директиве компилятора {$F-}, которая выбирается по умолчанию; действие данной директивы распространяется только на одну подпрограмму.

- если в подпрограмме используются переменные процедурного типа, то она независимо от своего расположения должна компилироваться с получением "дальнего" адреса.

3. **Директива EXTERNAL**

Директива external позволяет использовать в программе подпрограммы, написанные на языке ассемблера, скомпилированные отдельно, а также для описания подпрограмм, импортируемых из DLL. Эти подпрограммы должны быть скомпонованы с основной программой, используя ключ {$L <имя файла>}. Здесь имя файла - имя того файла (с расширением .OBJ), в котором находятся скомпилированные объектные модули подпрограмм, написанных на языке ассемблера.

Пример:

```
function Max(x,y: real):real; external;
{$L ASM1.obj}
```

4. **Директива ASSEMBLER**

Директива assembler позволяет написать подпрограмму полностью на языке ассемблера. При этом во время компиляции подпрограмма будет автоматически скомпилирована встроенным ассемблером пакета Turbo Pascal. При отладке такой подпрограммы можно использовать встроенный отладчик пакета.

Например:

```
function Max(x,y: integer):integer; assembler;
 begin
  asm
     mov AX, X
     cmp AX, Y
     JG  @1
     mov AX, Y
     @1: mov @Result, AX
  end;
 end;
```

это не прямой доступ к регистрам

специальная переменная, в которую надо занести результат

можно использовать необъявленные метки

ассемблерная вставка заканчивается как составной оператор

5. **Директива INLINE**

Есть еще оператор *inline*. Он записывается после *begin*.

Директива *inline* позволяет включить в текст программы команды, записанные непосредственно в машинных кодах. В отличие от других подпрограмм подпрограмма с директивой inline непосредственно добавляется (подставляется - как открытая подпрограмма) всюду, где есть ее вызов (фактически она является макроопределением). Такие подпрограммы могут иметь параметры, которые можно использовать в тексте подпрограммы, получая их из стека через его голову (по имени - нельзя!). Такие подпрограммы достаточно эффективны по времени, но считается, что число команд не должно быть больше 10.

Машинные коды в процедуре записываются в круглых скобках побайтно через прямой слэш (/).

Пример:

```
function  Max3(x,y: integer):integer; inline;
 ($58/        { ≡ POP AX          ---- извлекли из стека X}
  $5A/        { ≡ POP DX          ---- извлекли из стека Y}
  $3B/$C2/    { ≡ CMP AX, DX      ---- сравнили}
  $7F/$02/    { ≡ JG              ---- перешли по «больше»}
  $8B/$C2);   { ≡ MOV AX,DX       ---- переслали результат}
 begin
   ..... {прочие действия в подпрограмме}
 end;
```

6. **Директива INTERRUPT**

Директива interrupt предназначена для процедур, обрабатывающих прерывания. Такие процедуры имеют стандартный заголовок:

```
procedure IntHandler(Flags,  CS, IP, AX,BX,  CX, DX, SI, DI, DS, ES, BP: Word); interrupt;
begin
…
end;
```

В заголовке отдельные параметры можно опускать (но только с начала списка), промежуточные параметры удалять нельзя, например:

```
procedure IntHandler(DI,  ES, BP: Word); interrupt;  {неправильный заголовок}
procedure IntHandler(DI,  DS, ES, BP: Word); interrupt;  {правильный заголовок}
```

Нельзя в заголовке процедуры обработки прерываний записывать и какие-либо другие параметры.

18.14 Отладка и тестирование программ, содержащих подпрограммы

Практикуются два вида тестирования: нисходящее тестирование и восходящее тестирование. При тестировании программы и ее подпрограмм обычно не обойтись без комбинации этих методов.

18.14.1 Нисходящее тестирование и подпрограммы-заглушки

Независимо от того, как создается программная система (основная программа, в которой имеются подпрограммы), программистом-одиночкой или коллективом программистов, не все подпрограммы обычно бывают готовы одновременно. Тем не менее и при этих условиях возможно выполнить тестирование обмена управляющими сигналами между основной программой и ее подпрограммами первого уровня, а также тестирование и отладку уже готовых программ. Процесс тестирования обмена управляющими сигналами между основной программой и ее подпрограммами называется *нисходящим тестированием.*

Поскольку для выполнения тестирования основная программа нуждается во всех подпрограммах первого уровня, отсутствующие пока подпрограммы необходимо заменить т.н. подпрограммами-заглушками (stub). Заглушка состоит из заголовка процедуры или функции, за которым следует тело, содержащее минимум операторов. Эти операторы должны выводить например сообщение о выполнении данной подпрограммы (которая пока в виде заглушки), а также присваивать простые значения любым выходным данным.

18.14.2 Восходящее тестирование и программы-тестеры

Конечно, готовую подпрограмму, если в ней обнаружатся неполадки, всегда можно заменить заглушкой, тем не менее, предварительное тестирование новой подпрограммы следует выполнять обязательно. Искать и исправлять ошибки в отдельной подпрограмме всегда легче, чем в готовой программной системе. Вы сможете протестировать новую подпрограмму, воспользовавшись небольшой программой-тестером.

Не тратьте много времени на создание элегантной программы-тестера — эта программа окажется не нужна в момент, когда тестирование новой подпрограммы будет завершено. Программа-тестер должна содержать только раздел описаний и минимум операторов, необходимых для тестирования единственной подпрограммы. Данная программа должна начинаться со считывания или присваивания значений всем входным параметрам, а также параметрам входа-выхода, после чего следует вызов тестируемой подпрограммы. В завершение программа-тестер должна отображать результаты работы подпрограммы.

После того как вы уверены, что подпрограмма работает должным образом, замените ее в программной системе заглушкой. Процесс раздельного тестирования самостоятельных подпрограмм, прежде чем собрать их в программную систему, называется *восходящим тестированием.*

Комбинация восходящего и нисходящего тестирований даст вам уверенность, что программная система после сборки окажется сравнительно свободна от ошибок. В результате заключительная отладка будет выполнена быстро и без проблем.

18.14.3 Рекомендации по отладке программ, содержащих подпрограммы

Вот несколько рекомендаций по отладке программных систем.

1. При создании исходного текста не забывайте использовать комментарии для документирования каждого параметра и локального идентификатора подпрограммы. Также используйте комментарии для описания операций, выполняемых подпрограммой.
2. Пусть выполнение каждой подпрограммы оставляет след в данных вывода программной системы. Для этого в каждую из подпрограмм, в начале ее тела, поместите оператор WriteLn, который бы отображал сообщение с именем данной подпрограммы.
3. В каждую из подпрограмм вставьте операторы, которые при входе в нее отображали бы значения всех входных параметров, а также параметров входа-выхода данной подпрограммы.
4. Вставьте операторы, которые бы отображали значения всех выходных данных подпрограммы после передачи управления основной программе. Осуществите ручную проверку этих значений, чтобы убедиться, что они корректны. Для процедур проследите, чтобы все выходные параметры, а также: параметры входа-выхода были объявлены как параметры-переменные.
5. Проследите, чтобы подпрограмма-заглушка присваивала значения каждому из выходных параметров.

Очень полезно планировать отладку в процессе создания каждой подпрограммы, а не добавлять отладочные операторы впоследствии. При создании подпрограмм, добавляйте в них отладочные операторы, о которых шла речь в рекомендациях 2 и 4 выше. Когда убедитесь, что подпрограмма работает как нужно, эти операторы несложно будет "изъять из обращения". Для этого проще всего превратить такие операторы в комментарии, заключив их в фигурные скобки, Поэтому же, если возникнут какие-либо проблемы, эти комментарии не составит труда снова сделать исполняемыми операторами, удалив фигурные скобки.

Другой способ включения-выключения отладочных операторов состоит в чтобы использовать глобальную константу типа Boolean (например, Debug), объявленную в основной программе. При этом, когда потребуется отладка, в основной программе должно быть следующее описание данной константы.

```
const
  Debug = True  {отладка включена}
```

А когда надобность в отладке минует, данное описание константы дол выглядеть так:

```
const
  Debug = False  {отладка выключена}
```

Затем в тело основной программы, а также в подпрограммы вставьте диагностические операторы WriteLn, управляемые оператором if с Debug в качестве условия.

18.14.4 Использование отладчика для трассировки процедур

В качестве альтернативы отображению начальных значений входных (и входа-выхода) параметров процедуры с использованием специальных отладочным операторов, для получения соответствующей информации можно воспользоваться отладчиком Turbo Pascal и его возможностью трассировки Trace into (<F7>). Если оператор, который должен быть выполнен следующим, представляет собой оператор вызова процедуры, нажатие клавиши <F7> обеспечит вход в эту процедуру. В результате в окне редактора появится тело этой процедуры с указателем выполнения на слове begin. Если вы укажете параметры процедуры как переменные для наблюдения, исходные значения этих параметров появятся в окне Watch. Затем, с выполнением каждого оператора процедуры (для этого каждый раз требуется . нажимать клавишу <F7>), любое новое значение, присвоенное выходному параметру или параметру входа-выхода, будет отображено в окне Watch.

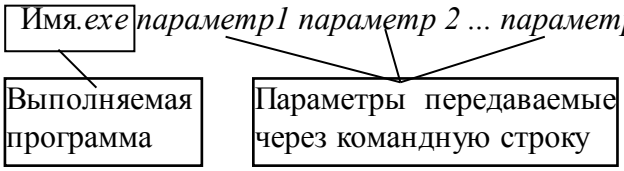
После того как процедура возвратит управление основной программе, вы можете воспользоваться отладчиком чтобы выяснить, какие значения были возвращены в программу. На этот раз в качестве переменных для наблюдения следует указать все фактические параметры, которые соответствуют выходным параметрам или параметрам входа-выхода. В результате значения, возвращенные процедурой, окажутся отображены в окне Watch.

18.14.5 Область действия идентификаторов и переменные в окне Watch

Значения, отображаемые в окне Watch, определяются с использованием обычных правил области действия имен для Pascal-идентификаторов. Соответственно, локальные переменные и формальные параметры процедуры будут помечены как Unknown identifier (Неизвестный идентификатор), пока не начнется выполнение данной процедуры. А по выходе из процедуры, ее локальные переменные и формальные параметры снова станут "неизвестными идентификаторами".

18.15. Получение доступа а параметрам командной строки, запуск внешних программ.

Обычно форма запуска внешних программ через командную строку имеет вид:



Каждой написанной на Паскале программе можно передавать параметры через *командную* строку. Параметры при запуске внешней программы записываются в командной строке справа от имени программы через пробел.

Параметры можно передавать не только через командную строку, но и при запуске программы из интегрированной среды, для чего *в меню OPTIONS* (для Турбо-Паскаля) ил *в меню Run* (для Free Pascal) надо указать соответствующее значение (в виде строки) для пункта *PARAMETERS*.

Чтобы параметры, передаваемые при запуске, воспринимались в вызванной программе, надо предусмотреть в этой программе ответы на следующие вопросы:

1. Сколько параметров передано? функция без параметров
2. Какие параметры переданы?

Ответ на первый вопрос: *paramcount* - возвращает количество переданных через командную строку параметров. Если справа после имени программы (при вызове) ничего ни указано, то *paramcount* - вернет значение 0. Когда указано N параметров, то ParamCount вернет значение *N*.

Для ответа на второй вопрос чтобы узнать, какие конкретно параметры переданы, есть другая функция: *Paramstr(<номер>).* Для каждого (по номеру) значения аргумента, функция возвращает значение параметра в виде *строки*. Если указать *Paramstr(0),* то функция в ответ вернет полный путь к программе. *Paramstr(ParamCount)* – вернет значение последнего параметра.

Пример вывода на экран всех параметров программы:

```
writeln('При запуске программы ', paramstr(0), ' указано ', paramcount, ' следующих параметров: ');
for i:=1 to paramcount do
  begin
    writeln(i:2, -й параметр: , paramstr(i));
  end;
```

надо помнить, что это строка (может понадобиться преобразование в число)

процедура VAL