Лекция 15. Процедуры в языке Delphi (продолжение 3)

Области видимости. Локальные и глобальные описания

Еще одной важной деталью, касающейся использования подпрограмм, является видимость переменных. Само понятие видимости подразумевает под собой тот факт, что переменная/константа/тип/процедура, объявленная в одном месте программы может быть доступна, или наоборот, недоступна, в другом.

Все **описания внутри кода** программы, подпрограммы или модуля доступны ***ниже*** своего описания **в коде этой же** программы, подпрограммы или модуля, в том числе **в подпрограммах**, описанных **ниже** по коду **в коде этой же** программы, подпрограммы или модуля. Описания, сделанные **вне** процедуры, но доступные в ней, являются **глобальными** для этой процедуры.

Но все описания констант/типов/переменных/процедур внутри процедуры (они так и называются – **локальными**) доступны только внутри этой процедуры с момента описания, но не в программе ее вызывающей и не программе/подпрограмме/модуле, содержащем ее описание. Все описания из раздела *Interface* (и только лишь этого раздела) модуля доступны в программе или другом модуле, с момента подключения (*uses*) к ним этого модуля.

В **разных** областях видимости могут существовать **одинаковые** имена. Например, две переменные с одним и тем же именем. Разумеется, компилятор еще на стадии проверки синтаксиса не допустит, чтобы в программе были объявлены одноименные переменные в рамках одной области видимости (скажем, 2 глобальных переменных *X*, или 2 локальных переменных *X* в одной и той же подпрограмме). Речь в данном случае идет о том, что произойдет, если в одной и той же подпрограмме будет 2 переменных *X*, одна – глобальная, а другая – локальная. В таком случае в действие вступает правило близости, т.е. какая переменная описана ближе (локальнее) к месту использования в коде, та и есть верная. Применительно к подпрограмме ближней оказывается локальная переменная *X*, и именно она будет задействована внутри подпрограммы.

```
Program GlobLocal;
{$AppType CONSOLE}

Uses Unit1;  //см.код этого модуля ниже, там тоже есть X

var
  X: integer;

Procedure Proc1;
Var X: integer;
begin
  X := 12; // Здесь значение будет присвоено локальной переменной X
  GlobLocal.X := 13; // Так можно обратится к глобальной
                     //(с префиксом имени программы
  Unit1.X := 14;     // или модуля)

  Writeln('X=', X); //12
end;

begin
  X := 11; // Здесь значение будет присвоено глобальной переменной X
           // из данной программы GlobLocal
  Writeln('X=', X); //11
```

```
  Proc1;
  Writeln('X=', X); //13

  Writeln('X=', Unit1.X); //14    X из подключенного модуля Unit1
  Unit1.X :=15;

end.
```

```
Unit Unit1;
Interface

Var X: integer;

Implementation

Initialization
  X:= 5;
Finalization
  Writeln('X=', X); //15
  Readln
End.
```

Как видно из примера, хотя глобальная переменная перекрыта локальной с тем же именем, но к глобальной переменной тоже можно получить доступ, указав имя программы/модуля, в котором она описана, в качестве префикса, но по умолчанию, обращение идет к локальной переменной.

Что касается видимости подпрограмм, то она определяется аналогичным образом: подпрограммы, объявленные в самой программе, видны всюду ниже по коду в этой программе. Те же подпрограммы, которые объявлены внутри процедуры или функции, доступны только внутри нее:

```
Program ProjLocal;

Procedure Proc1;
  Procedure SubProc;
  begin
  end;
begin
 SubProc; // Верно. Вложенная процедура здесь видна
end;

begin
  Proc1; // Верно
  SubProc; // Ошибка! Процедура SubProc недоступна за пределами Proc1
end.
```

В том случае, когда имена встроенной и некой глобальной процедуры совпадают, то, по аналогии с переменными, в области видимости встроенной процедуры, именно она и будет выполнена.

Процедурный тип

В некоторых случаях целесообразно имя одной процедуры/функции сделать параметром другой процедуры/функции, для описания типа такого параметра и создается процедурный тип. В таком параметре нет необходимости при вызове одной конкретной процедуры/функции из другой, в отличие от случая, когда вызывать необходимо будет разные, но однотипные, процедуры/функции.

Допустим, для заданного одномерного вещественного массива *A*, состоящего из *N* элементов, надо найти три суммы:

а) $\sum_{i=1}^{N} A_i \sin A_i$ б) $\sum_{i=1}^{N} \sqrt[3]{|A_i|+1}$ в) $\sum_{i=1}^{N} A_i^2$

Для вычисления каждой напишем процедуры.

```
program Project1;
{$APPTYPE CONSOLE}
const nmax = 20;
type mas = array [1..nmax] of real; // этот тип необходим для
                            // использования массива как параметра процедур
procedure proc1(const a: mas; const n: byte; out sum: real);
var i: byte;
begin
  sum:=0;
  for i:=1 to n do  sum:=sum+a[i]*sin(a[i]);
end;
procedure proc2(const a: mas; const n: byte; out sum: real);
var i: byte;
begin
  sum:=0;
  for i:=1 to n do  sum:=sum+exp(1/3*ln(abs(a[i])+1));
end;
procedure proc3(const a: mas; const n: byte; out sum: real);
var i: byte;
begin
  sum:=0;
  for i:=1 to n do  sum:=sum+sqr(a[i]);
end;
var   A: mas;  N,i:byte;  s1,s2,s3: real;
begin
  //ввод длины массива и ввод/вывод элементов массива
  repeat
    write('N(1..',nmax,')=?'); readln(N);
  until (N>0) and (N<=nmax);
  writeln('A(',N,'):');
  for i:=1 to N do  read(A[i]);
  readln;
  writeln('Vy zadali massiv A(',N,'):');
  for i:=1 to N do  write(' ',A[i]:5:1);
  writeln;
        // поиск трех сумм тремя процедурами
  proc1(a,n,s1);  proc2(a,n,s2);  proc3(a,n,s3);
        // вывод результатов
  writeln('Summy a)', s1:5:2, ' b)', s2:5:2, ' c)', s3:5:2);
  write('Press Enter...':75); readln;
end.
```

Единственное, чем отличаются эти три процедуры – это функция, применяемая к A_i для вычисления очередного слагаемого. Сделаем эту функцию **параметром** одной общей для всех трех функций процедуры. Для этого следует определить еще один тип – **процедурный**. Назовем его ***func***. Этому типу должны соответствовать все три функции:

```
program Project2;
{$APPTYPE CONSOLE}
const nmax=20;
type
  mas = array [1..nmax] of real;
  func = function (x: real):real; //процедурный тип выглядит как
                                  // заголовок процедуры/функции без имени
function f1(x: real):real; // три разных функции
begin
  f1:=x*sin(x);
end;
function f2(x: real):real;
begin
  f2:=exp(1/3*ln(abs(x)+1));
end;
function f3(x: real):real;
begin
  f3:=sqr(x);
end;
     // одна общая процедура с параметром процедурного типа:
procedure proc(const a: mas; const n: byte; f: func; out sum: real);
var i: byte;
begin
  sum:=0;
  for i:=1 to n do
    sum:=sum+f(a[i]);
end;

var   A: mas;  N,i:byte;  s1,s2,s3: real;
begin
  //ввод длины массива и ввод/вывод элементов массива
  repeat
    write('N(1..',nmax,')=?'); readln(N);
  until (N>0) and (N<=nmax);
  writeln('A(',N,'):');
  for i:=1 to N do
    read(A[i]);
  readln;
  writeln('Vy zadali massiv A(',N,'):');
  for i:=1 to N do
    write(' ',A[i]:5:1);
  writeln;
     // поиск трех разных сумм одной процедурой (три вызова):
  proc(a,n, f1, s1);    proc(a,n, f2, s2);    proc(a,n, f3, s3);

     // вывод результатов
  writeln('Summy a)', s1:5:2, ' b)', s2:5:2, ' c)', s3:5:2);
  write('Press Enter...':75); readln;
end.
```

Сейчас в *Delphi* дальняя модель памяти используется по умолчанию, но в более ранних версиях и в *Turbo* и *Borland Pascal* после заголовков следует написать **far;** для правильного формирования адреса этих функций в целях использования в качестве параметра другой процедуры/функции. Либо описания функций надо сделать между директивами компилятору **{$F+}** и **{$F-}.**

Теперь одна процедура в зависимости от переданного ей имени функции находит разные суммы вида $\sum_{i=1}^{N} f(A_i)$

а) $f1(x) = x\sin(x)$ б) $f2(x) = \sqrt[3]{|x|+1}$ в) $f3(x) = x^2$

Соглашения о вызовах

В разных языках, и даже внутри одного языка, могут использоваться разные стандарты вызова процедур (общего вида и функций), передачи параметров и завершения работы процедур. При вызове процедуры надо вычислить в определенном порядке и разместить в памяти (регистры и стек вызовов) значения всех параметров процедуры и передать управление в процедуру. В том же стеке будет выделена память для локальных переменных, таким образом в языках высокого уровня создаются переменные, существующие только во время работы процедуры. При завершении работы процедуры, использованную процедурой память (в стеке вызовов, и сегментные регистры процессора и *ESP* с *EBP*) надо освободить, и вернуть управление вызывающей программе, используя адрес возврата, хранимый в том же стеке. Возвращаемое функцией значение обычно хранится в регистре *EAX*, либо, если слишком велико для размещения в регистре, то оно размещается на вершине стека, а значение в регистре *EAX* будет указывать на него.

Вот неполный перечень соглашений о вызовах:

Стандарт	**Порядок вычисления параметров, и их размещение**	**Освобождает стек**
registr (по умолчанию в *Pascal и Delphi*), частный случай «быстрого вызова» *fastcall*, не имеющего единого межъязыкового стандарта	Слева направо, но первые три параметра *по возможности* в регистрах *EAX, EDX, ECX,* из остальных последний в вершине стека	Сама подпрограмма
pascal (используется в *Pascal* и *Delphi*)	Слева направо, последний параметр в вершине стека В *Delphi* в функциях *неявно* добавляется первый параметр *Result* для возврата результата функции, что позволяет делать результатом функции значения не только простого типа (как в *Pascal*).	Сама подпрограмма
stdcall (*Standart Call,* используется для вызова функций *WinAPI (Windows Application Programming Interfaces*), хранящихся в *dll*-файлах)	Справа налево, первый параметр в вершине стека	Сама подпрограмма
cdecl (*C-Declaration,* вызов в языке Си)	Справа налево, первый параметр в вершине стека	Вызывающая программа

Когда надо указывать/учитывать используемый стандарт?

1) При создании *dll*-файла, чтобы его можно было использовать в программе, написанной на другом языке;

2) При импортировании процедур/функций из *dll*-файла, например, для смены кодовой страницы при работе с консолью из «ядра *Windows*» были вытащены две функции с указанием стандарта *stdcall*;

3) При размещении вызовов функций в списке фактических параметров при вызове другой процедуры/функции. Например, как вы думаете, в каком порядке будут выведены 5 чисел первыми шестью вызовами (процедуры отличаются только используемым стандартом)? Как изменится ***i*** в результате последних трёх вызовов?

```
program ProjCall;
{$APPTYPE CONSOLE}

Function F(k: integer):integer;
begin
  writeln(k);
  F:=k
end;

Procedure P1(a,b,c,d,e:integer); register;
begin
  writeln('Ok');
end;
Procedure P2(a,b,c,d,e:integer); pascal;
begin
  writeln('Ok');
end;
Procedure P3(a,b,c,d,e:integer); stdcall;
begin
  writeln('Ok');
end;
Procedure P4(a,b,c,d,e:integer); cdecl;
begin
  writeln('Ok');
end;
Procedure P5(a,b,c,d,e:integer); // по умолчанию
begin
  writeln('Ok');
end;

Function F2(var i: integer; j: integer):integer;
begin
  writeln('i=',i,' j=',j, ' i+j=', i+j);
  if i mod 2 = 0 then i:=i+j
  else i:=i*j;
  F2:=i
end;

var i: integer;
begin
    P1(F(1),F(2),F(3),F(4),F(5));  //register 45321
    P2(F(1),F(2),F(3),F(4),F(5));  //pascal   12345
    P3(F(1),F(2),F(3),F(4),F(5));  //stdcall  54321
    P4(F(1),F(2),F(3),F(4),F(5));  //cdecl    54321
    P5(F(1),F(2),F(3),F(4),F(5));  //         45321 = register
    Writeln('Writeln: ',F(1),F(2),F(3),F(4),F(5), 'Ok');//12345

    i:=1;
    P1(F2(i,1),F2(i,2),F2(i,3),F2(i,4),F2(i,5)); //register
    writeln('Result1 i=',i);                      // 55=(1*4+5)*3*2+1
    i:=1;
    P2(F2(i,1),F2(i,2),F2(i,3),F2(i,4),F2(i,5)); //pascal
    writeln('Result2 i=',i);                      // 25=(1*1*2+3)*4+5
    i:=1;
    P3(F2(i,1),F2(i,2),F2(i,3),F2(i,4),F2(i,5)); //stdcall
    writeln('Result3 i=',i);                      // 47=(1*5*4+3)*2+1

    Write('Press ENTER'); readln;
end.
```

Нисходящее и восходящее тестирование программ

Подготовка тестов обычно идет параллельно восходящей/нисходящей разработке программы. При нисходящей разработке алгоритм большинства подпрограмм заменяется **заглушками** (простейший код для имитации ее работы), которые постепенно (по мере их разработке и тестирования уже написанной программы) заменяются реальным алгоритмом. Примеры оформления заглушек в виде процедур – на следующей странице. Программа тестируется так, как будто уже целиком закодирована, при этом текст заглушек должен меняться так, чтобы присваиваемые в нем результаты соответствовали проводимому тесту. Например, если в нижерасмотренном примере заменить строку

```
writeln('Процент точек = ',p:5:1);
```

на

```
if p=0 then writeln ('Нет точек, в указанной области. p=0.0%')
else writeln('Процент точек = ',p:5:1);
```

то в заглушке для проверки двух альтернативных решений нужны значения как минимум для двух тестов, но все кроме одного должны быть закомментированы (не доступны) в каждый момент времени:

```
{заглушка для теста 1}
p:=20; sa:= 5.867;
{заглушка для теста 4 с p=0 и max(sa)}
//p:=0; sa:= 14.142;
```

```
testN := 1;{выбор теста по номеру}
case testN of
  1: begin p:=20; sa:= 5.867; end;
  4: begin p:=0; sa:= 14.142; end;
End;
```

После замены очередной заглушки реальным алгоритмом выполнения подзадачи, вся программа вновь тестируется с помощью всего набора тестов для всей программы, и также для этой конкретной подпрограммы могут быть созданы отдельные тесты для всесторонней проверки ее работы на разных наборах входных и выходных данных, которые в масштабах всей программы могут быть промежуточными.

При восходящей разработке сначала создается код подпрограмм, для тестирования которых пишутся отдельные программы – **драйвера**, в каждой из которых выполняется ввод исходных данных, необходимых для проверки алгоритма конкретной подпрограммы, а после ее выполнения выводятся полученные результаты. Затем отлаженные части собираются вместе для создания алгоритма выполнения всей задачи, и тестируется общая логика работы программы.

Нисходящее тестирование (и разработка)	**Восходящее тестирование (и разработка)**
Достоинства: + Сначала разрабатывается, кодируется и тестируется логика основной программы (с *заглушками*), и **программа сразу готова к демонстрации.**	Достоинства: + **Подпрограммы тестируются** отдельно вне основного кода программы (с помощью *драйверов*) и **без посредников**.
Недостатки: – **Тестирование подпрограмм** выполняется **не «напрямую»** хотя и сразу «на своем месте», а значит, для ее тестирования, надо создать такой набор исходных данных для всей программы, чтобы до тестируемой подпрограммы они дошли в нужном виде. Программа постепенно растет и усложняется, **усложняя и процесс создания тестов.**	Недостатки: – Для тестирования приходится писать **«лишний» код** (драйвера), который не войдет в окончательный состав кода программы. – **Программа не готова для демонстрации,** тестирования и отладки как единое целое пока полностью не собрана, что выполняется в самом конце.

Как и разработку, тестирование можно выполнять смешанным способом (метод «сэндвича»), что позволяет сочетать достоинства, убрав часть недостатков.

Заглушки на примере программы из Лабораторной работы №2:

Программа №1 без процедур, но с заглушкой

```
program points1;
{$APPTYPE CONSOLE}
const
  nmax=20; {верхняя граница массива}
var
  n, i: integer;
  r, p, sa: real;
  x, y: array [1..nmax] of real;
  {описания других промежуточных данных}
begin
  {A0.1 - ввод-вывод входных данных}
  writeln(' ':9, 'Точки');
  writeln('Кол-во точек n:'); readln(n);
  writeln('Критич.удаление r:'); readln(r);
  writeln('Абсциссы и ординаты:');
  for i:=1 to n do readln(x[i], y[i]);
  for i:=1 to 20 do writeln; {рзделение}

  writeln('Количество точек =', n:3);
  writeln('Критич.удаление = ', r:5:1);
  writeln('Абсциссы и ординаты: ');
  for i:=1 to n do
    writeln(' ':5,x[i]:5:1,' ':8,y[i]:5:1);

  {A0.2 - обработка}
  {заглушка для теста 1}
  p:=20;
  sa:= 5.867;

  {A0.3 - вывод результатов}
  writeln('Процент точек = ',p:5:1);
  writeln('Среднее удаление = ',sa:6:3);
  write('Press Enter...'); readln;
end.
```

Программа №2 с процедурой-заглушкой

```
program points2;
{$APPTYPE CONSOLE}
const  nmax=20; {верхняя граница массива}
type
  mas = array [1..nmax] of real;
//-------------------------------------
procedure A02(const n: integer;
              var x,y: mas; const r: real;
              out p, sa: real);
{var описание промежуточных данных}
begin
      {заглушка для теста 1}
      p:=20;
      sa:= 5.867;
end;
//-------------------------------------
var  n, i: integer;   r, p, sa: real;
     x, y: mas;
begin
  {A0.1 - ввод-вывод входных данных}
  writeln(' ':9, 'Точки');
  writeln('Кол-во точек n:'); readln(n);
  writeln('Критич.удаление r:'); readln(r);
  writeln('Абсциссы и ординаты:');
  for i:=1 to n do readln(x[i], y[i]);
  for i:=1 to 20 do writeln; {рзделение}

  writeln('Количество точек =', n:3);
  writeln('Критич.удаление = ', r:5:1);
  writeln('Абсциссы и ординаты: ');
  for i:=1 to n do
    writeln(' ':5,x[i]:5:1,' ':8,y[i]:5:1);
  {A0.2 - обработка: вызов процедуры}
  A02(n, x, y, r, p, sa);
  {A0.3 - вывод результатов}
  writeln('Процент точек = ',p:5:1);
  writeln('Среднее удаление = ',sa:6:3);
  write('Press Enter...'); readln;
end.
```

Программа №3 с процедурой-заглушкой, находящейся в отдельном модуле

Главная программа

```
program points3;
{$APPTYPE CONSOLE}
Uses UnPoints;
Var   n, i: integer;   r, p, sa: real;
      x, y: mas;
begin
  {A0.1 - ввод-вывод входных данных}
  writeln(' ':9, 'Точки');
  writeln('Кол-во точек n:'); readln(n);
  writeln('Критич.удаление r:'); readln(r);
  writeln('Абсциссы и ординаты:');
  for i:=1 to n do readln(x[i], y[i]);
  for i:=1 to 20 do writeln; {рзделение}

  writeln('Количество точек =', n:3);
  writeln('Критич.удаление = ', r:5:1);
  writeln('Абсциссы и ординаты: ');
  for i:=1 to n do
    writeln(' ':5,x[i]:5:1,' ':8,y[i]:5:1);

  {A0.2 - обработка . Вызов процедуры}
  A02(n, x, y, r, p, sa);
  {A0.3 - вывод результатов}
  writeln('Процент точек = ',p:5:1);
  writeln('Среднее удаление = ',sa:6:3);
  write('Press Enter...'); readln;
end.
```

Модуль

```
unit UnPoints;
Interface
const
  nmax=20; {верхняя граница массива}
type
  mas = array [1..nmax] of real;

procedure A02(const n: integer;
              var x,y: mas; const r: real;
              out p, sa: real);

Implementation

procedure A02(const n: integer;
              var x,y: mas; const r: real;
              out p, sa: real);
{var описание промежуточных данных}
begin
      {заглушка для теста 1}
      p:=20;
      sa:= 5.867;
end;

end.
```

Нисходящую разработку и составление тестов можно увидеть в примере выполнения Типового расчета.