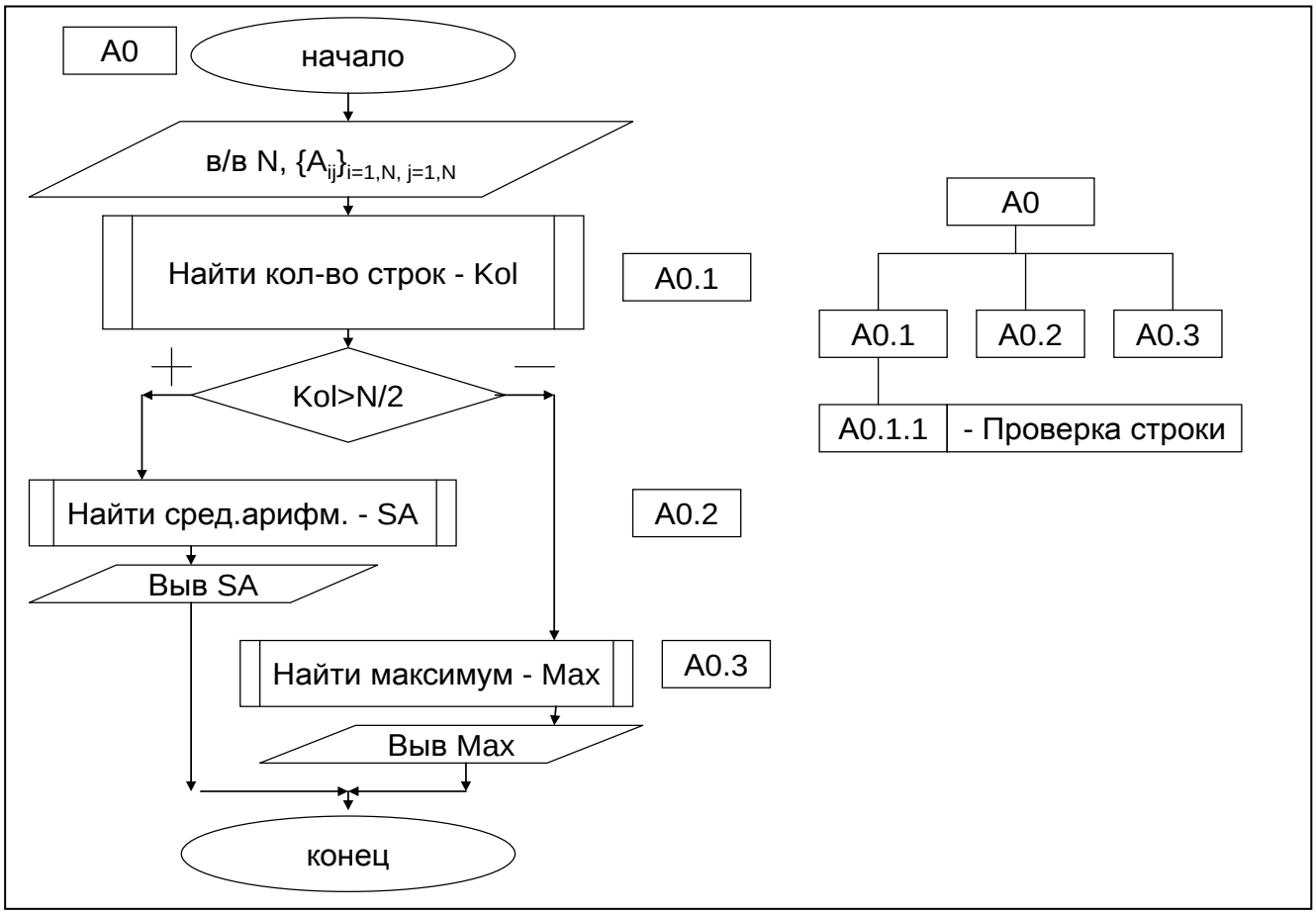
Лекция 14. Процедуры в языке Delphi (продолжение)

14.1 Процедуры *vs* Функции. Пример

Переделаем, уже написанную программу с процедурами общего вида, в программу с функциями.

Условие задачи:

Дана квадратная матрица *A* размером *N*x*N*.
Если количество (*Kol*) строк, в которых находятся только положительные элементы, больше *N*/2,
Найти *SA* – среднее арифметическое значений всех положительных элементов,
иначе
Найти *Max* – максимальное значение среди элементов, лежащих выше главной диагонали матрицы и на ней.



Все четыре выделенные подзадачи, сделаем функциями, поскольку во всех четырех есть чисто выходной параметр простого типа. Для этого внесем 5 изменений в каждую из процедур и изменим их вызовы. Какие это 5 изменений:

1) ключевое слово ***procedure*** **меняем** на ***function***; 2) **убираем** из списка параметров выбранный для возврата в качестве результата чисто выходной параметр (допустим, *X*), 2) **добавляя** его (*X*) описание в список локальных переменных, 3) а его (*X*) тип указываем как **тип результата** функции; 5) после всех

операторов тела бывшей процедуры добавляем еще один: **имени функции присваиваем** всё тот же *X*.

Например, есть процедура, у которой есть чисто выходной параметр *k* простого типа:

```
Procedure Proc( const a: real; b: byte;
                          var r: real; out k: integer);
Var d: double;
Begin
  <Тело процедуры>

End;

Вызов:
Proc( a,b,c,k);
```

Вносимые изменения:

Процедура и Функция

```
Function Proc( const a: real; b: byte;
                          var r: real; out k: integer) : integer;
Var d: double; k: integer;
Begin
  <Тело процедуры>
  Proc := k;     или    Result := k;
End;

Вызов:
Proc( a,b,c,k);        k := Proc( a,b,c);
```

Если бы у процедуры не было чисто выходного параметра, но был бы параметр-переменная *p* простого типа, то можно было бы сначала разделить его на

два параметра: чисто входной *p*1, для передачи начального значения и чисто выходной *p*2, а затем преобразовать процедуру в функцию по выше приведенному сценарию.

Если бы у процедуры вообще не было кандидата на результат, то можно всегда его добавить, поскольку результатом функций часто выступает оценка успешности её выполнения: *True* (или 0) при удачном выполнении, *False* (или ненулевой код ошибки), если выполнить по какой-либо причине не возможно, например, невозможен ввод при отсутствии нужного файла на диске или отсутствия соответствующих прав доступа к нему, случаи переполнения или деления на ноль при вычислении по сложным математическим формулам. То есть в простейшем случае из процедуры:

```
Procedure F(<список формальных параметров>);
Begin
        <тело процедуры>
End;
```

можно сделать функцию:

```
Function F(<список формальных параметров>): Boolean;
Begin
    F := True;
    Try
        <тело процедуры>
    Except
        F := False
    End;
End;
```

На примере четырех функций для решения выше приведенной задачи из предыдущей лекции:

Процедура поиска среднего арифметического положительных элементов (которые гарантированно там присутствуют):

```
Const Nmax = 10;
Type  Matr = array [1..Nmax, 1..Nmax] of real;

Procedure FindSA( N: byte; var A: Matr; out SA: real);
Var   i,j,K: byte; S: real;
Begin
   S:=0;    K:=0;
   for i:=1 to N do
      for j:=1 to N do
         if A[i,j]>0 then
         begin
            S:=S+A[i,j]; K:=K+1;
         end;
   SA:=S/K;
End;
```

Функция:

```
Function FindSA( N: byte; var A: Matr):  real;
Var   i,j,K: byte; S: real; // SA: real;
Begin
   S:=0;    K:=0;
   for i:=1 to N do
      for j:=1 to N do
         if A[i,j]>0 then
         begin
            S:=S+A[i,j]; K:=K+1;
         end;
   FindSA:= S/K;            // вместо: SA:=S/K; FindSA:=SA;
End;
```

Процедура поиска максимума выше и на самой главной диагонали:

```
Procedure FindMax( N: byte; var A: Matr; out Max: real);
Var    i,j: byte;
Begin
   Max:=A[1,N];
   for i:=1 to N do
      for j:=i to N do
         if A[i,j]>Max then
            Max:=A[i,j];
End;
```

Функция:

```
Function FindMax( N: byte; var A: Matr): real;
Var    i,j: byte; Max: real;
Begin
   Max:=A[1,N];
   for i:=1 to N do
      for j:=i to N do
         if A[i,j]>Max then
            Max:=A[i,j];
   FindMax := Max;
End;
```

Процедура проверки *i*-ой строки (все ли положительны):

```
Procedure Check_i(const N,i: byte; var A: Matr; out Flag: boolean);
Var    j: byte;
Begin
   Flag:=TRUE;    j:=1;
   while (j<=N) AND Flag do
   begin
      if NOT (A[i,j]>0) then
         Flag:= FALSE;
      inc(j);
   end;
End;
```

Функция:

```
Function Check_i( const N,i: byte; var A: Matr): boolean;
Var    j: byte; Flag: boolean;
Begin
   Flag:=TRUE;    j:=1;
   while (j<=N) AND Flag do
   begin
      if NOT (A[i,j]>0) then
         Flag:= FALSE;
      inc(j);
   end;
   Check_i := Flag;
End;
```

Процедура подсчета количества строк с только положительными элементами, вызывающая процедуру проверки *i*-ой строки:

A0.1 – Найти количество строк

```
Procedure FindKol(   const N: byte; var A: Matr;
                     out Kol: byte);
Var    i: byte; Flag: boolean;
Begin
   Kol:=0;
   For i:=1 to N do
   begin
      Check_i( N, i, A, Flag);
      if Flag then
         inc(Kol);
   end;
End;
```

	вх	пром	вых	
Kol	-	+	+	out
Flag	-	+	-	лок
i	-	+	-	лок
N	+	-	-	const
A	+	-	-	const/ var

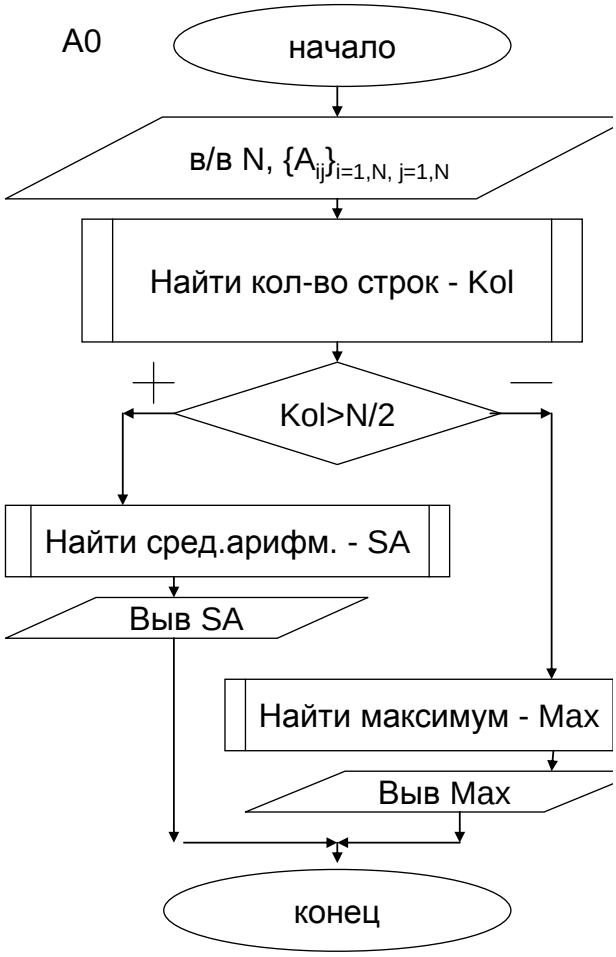
Функция, вызывающая функцию проверки строки:

A0.1 – Найти количество строк

```
Function FindKol( const N: byte; var A: Matr): byte;
Var    i, Kol : byte; Flag: boolean;
Begin
   Kol:=0;
   For i:=1 to N do
   begin
      Flag := Check_i( N,i,A);
      if Flag then
         inc(Kol);
   end;
   FindKol := Kol;
End;
```

	вх	пром	вых	
Kol	-	+	+	лок/ Result
Flag	-	+	-	лок
i	-	+	-	лок
N	+	-	-	const
A	+	-	-	const/ var

Вызывающая программа с процедурами общего вида:




```
Var
  N: Byte;
  A: Matr;
  Kol: byte;
  SA, Max: real;
Begin
   VvodVyvodNA( N, A);
   FindKol( N, A, Kol);
   if Kol > N/2 then
    begin
      FindSA( N, A, SA);
      writeln('Kol> N/2 and Sa=', SA:7:3);
    end
   else
    begin
      FindMax( N, A, Max);
      writeln('Kol<=N/2 and Max=',Max:5:1);
    end;

   writeln('Press ENTER'); readln;

End.
```

Вызывающая программа с функциями:

```
Var
  N: Byte;
  A: Matr;
  Kol: byte;
  SA, Max: real;

Begin
 VvodVyvodNA( N, A);
 Kol := FindKol( N, A);
 if Kol > N/2 then
  begin
   SA := FindSA( N, A);
   write('Kol> N/2 and ');
   writeln('Sa=', SA:7:3);
  end
 else
  begin
   Max := FindMax( N, A);
   write('Kol<=N/2 and ');
   writeln('Max=', Max:5:1);
  end;

 writeln('Press ENTER');
 readln;
End.
```

Можно сократить программный код, но не блок-схему:

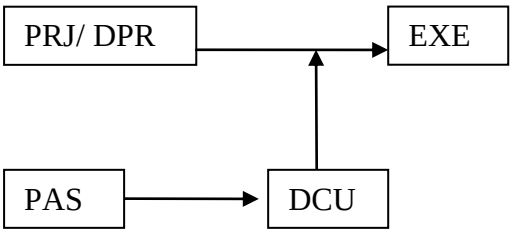
```
Var
  N: Byte;
  A: Matr;



Begin
 VvodVyvodNA( N, A);

 if FindKol( N, A) > N/2 then
  begin

   write('Kol> N/2 and Sa=');
   writeln(FindSA(N, A):7:3);
  end
 else
  begin

   write('Kol<=N/2 and Max=');
   writeln(FindMax(N, A):5:1);
  end;

 writeln('Press ENTER');
 readln;
End.
```

14.2. Внутренние и внешние описания процедур. Модули

Для использования **внутренних процедур** их описания размещаются в соответствующих разделах главной программы (***program***), описания процедур – в разделе процедур, а вызов процедур – в разделе операторов программы или другой процедуры того же модуля.

Описания **внешних процедур** размещаются **в отдельных файлах – модулях**, их еще называют библиотеками подпрограмм. Модули представляют собой отдельные единицы исходного текста, и могут быть разных видов:

1) статические модули (ключевое слово – ***unit***, исходный код располагается в файле с расширением ***pas*** (первые три буквы названия языка ***Pas****cal*, прародителя *Delphi*), после промежуточной компиляции становится *.***dcu** (***D****elphi* ***C****ompiled* ***U****nit*), а затем включается в состав общего с главной программой (*.*dpr* – ***D****elphi* ***Pr****oject*) исполняемого файла с расширением ***exe***),



2) динамически подключаемые библиотеки – ***DLL*** (***D****ynamic-****L****ink* ***L****ibrary*), которые тоже можно подключать и статически, но они не входят в состав общего с программой ***exe***-файла, что позволяет уменьшить его размер и подменять при необходимости *dll*-файл, не меняя самой программы, и кроме того, если какая-либо библиотека используется несколькими программами, то в оперативную память загружается только одна копия этой *DLL*. В *DLL* можно хранить описания процедур и функций, но не констант, типов или переменных, и можно использовать только базовые типы, но зато исходный код такой библиотеки может быть написан на языке программирования, отличном от языка программирования, на котором написана программа, к которой подключается откомпилированный *dll*-файл. В языке *Delphi* при написании исходного кода используется ключевое слово ***library***, исходный код располагается в файле с расширением ***dpr*** (***D****elphi* ***Pr****oject*), а после компиляции получает расширение ***dll***. Пример статического подключения *Kernal32.dll* для использования процедур изменения кодовой страницы:

```
// «вытащим» две функции из ядра Windows:
function SetConsoleOutputCP(wCodePageID: cardinal): cardinal;
                                        stdcall; external 'Kernel32.dll';
function SetConsoleCP(wCodePageID: cardinal): cardinal;
                                        stdcall; external 'Kernel32.dll';

begin
    // сменим кодовые страницы (Windows по умолчанию
    // для кириллицы использует кодировку 1251):
    setConsoleCP(1251);  // для ввода
    setConsoleOutputCP(1251); // для вывода
    writeln('По-русски, если шрифт сменить на Lucida Console');
    write('Нажмите ENTER для завершения'); readln;
end.
```

3) пакеты *.***bpl** (***B****orland* ***P****ackage* ***L****ibrary*) – это особым образом откомпилированные *DLL*, оптимизированные для совместного использования *Delphi*-программами, или средой разработки приложений *Delphi*, или и программами и средой. В отличие от *DLL* пакеты могут хранить и передавать программе типы (включая классы) и данные. Они разработаны специально для хранения компонентов, разного рода экспертов, редакторов сложных свойств и т.п.

В модуле (здесь и далее, и в типовом расчете подразумеваются ***unit***-файлы) обычно размещают константы, типы, процедуры и функции, которые могут быть использованы/вызваны из основной программы или других модулей. Файл с исходным текстом модуля должен состоять как минимум из трех частей:

1) заголовок модуля – ***unit***. Каждый модуль должен начинаться со строки, объявляющей, что данный блок текста является модулем, и задающей имя этого модуля. Имя модуля всегда должно соответствовать имени его файла, и наоборот. Например, если имя файла *unit1.pas*, то имя модуля *unit1*.

2) описание интерфейса. Второй функциональной строкой в модуле должен быть оператор ***interface***. Все описания, сделанные между этой строкой и оператором ***implementation*** данного модуля, доступны извне и может быть использовано другими программами и модулями. Именно в этой части описываются константы, типы, процедуры и функции, которые должны быть доступны основной программе или другим модулям. Для процедур и функций в этом разделе помещаются только их заголовки!

3) раздел реализации. Следует за описанием интерфейса и начинается с оператора ***implementation***, расположенного в отдельной строке. В этом разделе помещают полные описания (заголовки + тела) процедур и функций, упомянутых в разделе ***interface*** данного модуля. Кроме того, здесь могут быть описаны константы, типы, переменные, процедуры и функции, которые будут доступны только в пределах этого модуля.

4) Заканчивается модуль, как и основная программа, словом ***End*** и точкой.

Структура **типичного** модуля:

```
Unit unit1; {файл, значит, должен называться unit1.pas }
Interface
  uses <список используемых модулей>

    {описания, доступные извне:}
  const <описания констант>
  type <описание типов>

  <заголовки процедур и функций>

Implementation
  <полные описания процедур и функций,
           заголовки которых размещены в разделе interface>
End.
```

Помимо этих трех частей, модуль может иметь еще две необязательные части.

4) раздел инициализации. Начинается со слова **initialization** и содержит программный текст, необходимый для начала работы с этим модулем. Этот текст будет выполнен только один раз – перед началом выполнения основной программы.

5) раздел завершения. Начинается с ключевого слова **finalization** и располагается после раздела инициализации. Этот раздел содержит программный код, необходимый для завершения работы модуля. Этот код будет выполнен только один раз – при завершении работы программы.

Разделы инициализации и завершения подключаемых модулей выполняются в порядке, в котором эти модули открываются компилятором (т.е. первый модуль в описании **uses** основной программы, первый модуль в описании **uses** раздела **interface** этого модуля и т.д.).

Полная структура программного модуля:

```
Unit <имя модуля>;

Interface
  uses <список используемых модулей>

    {описания, доступные извне:}
  const <описания констант>
  type <описание типов>
  var <описание переменных>

  <заголовки процедур и функций>

Implementation
  uses <список используемых модулей>

     {описания, доступные внутри модуля:}
  const <описания констант>
  type <описание типов>
  var <описание переменных>

  <локальные для модуля процедуры и функции>

  <полные описания процедур и функций,
         заголовки которых размещены в разделе interface>

Initialization
  <операторы инициализации>
Finalization
  <операторы завершения>
End.
```

Для подключения модулей используется раздел ***uses***, в котором указывается список их имен через запятую. В модуле может быть два описания *uses*: одно в разделе *interface*, другое – в разделе *implementation*.

Иногда возникают ситуации, когда один модуль, например, *unit1*, использует другой модуль, например, *unit2*, который в свою очередь использует *unit1*. Наличие таких **взаимных (циклических) ссылок** говорит о просчетах, допущенных при проектировании структуры приложения. Эту проблему можно решить созданием третьего модуля, куда следует поместить необходимые для работы обоих модулей константы, типы, переменные, процедуры и функции. Если это невозможно, можно разместить в одном модуле ссылку на второй модуль в разделе *interface*, а во втором модуле – в разделе *implementation*.

Пример. Один модуль (unit) с процедурами

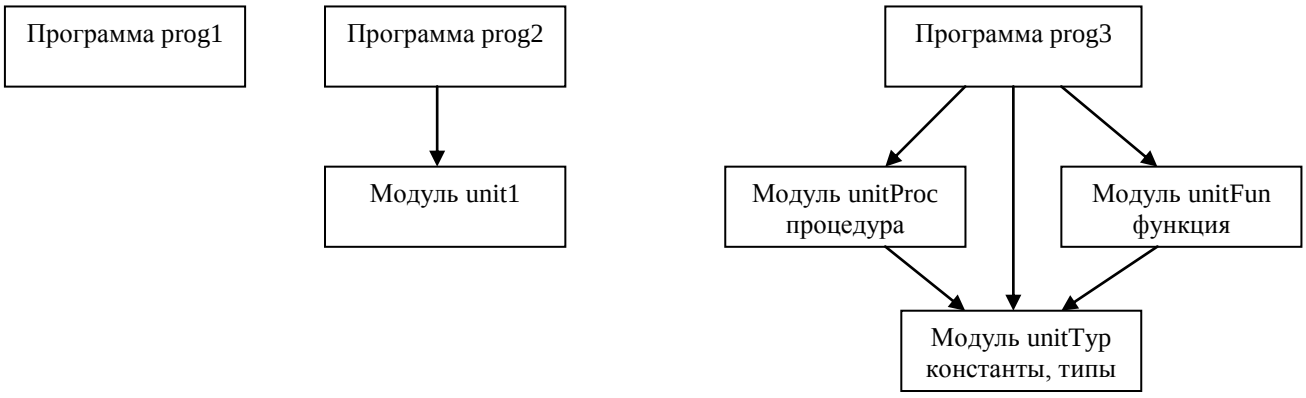
При описании **внешних процедур** описания глобальных объектов размещаются в соответствующих разделах внутри раздела ***interface*** модуля (*unit*), там же размещаются заголовки процедур, а описания процедур – в разделе ***implementation*** модуля (*unit*), а вызовы процедур остаются в разделе операторов вызывающей программы. В главной программе (*program*) указывается имя используемого модуля в разделе ***uses***.

Пример 1 на следующей странице.

Пример. Несколько модулей (unit) с процедурами

Создадим для каждой из двух процедур (см. Пример 2 ниже) собственный модуль. Описания глобальных констант и типов, необходимых всем остальным модулям, целесообразно также разместить в отдельном модуле (*unit*) в разделе ***interface***, при этом реализационная часть ***implementation*** останется пустой. В программе и модулях указываются имена используемых модулей в разделе ***uses***.

Связи между модулями в следующих двух примерах:



Пример 1:

Программа без модуля	**Программа с подключенным модулем**	**Модуль с константами, типами, процедурами и функциями.**

Program prog1;
{$APPTYPE CONSOLE}
 	
Const
nmax=20;
Type
ma = array[1..nmax] of shortint;
 	
Procedure psum(const n: byte;
var a: ma;
out S:integer);
Var i: byte;
Begin
S:=0;
For i:=1 to n do S:=S+a[i];
End;
 	
Function fsum(const n: byte;
var a: ma): integer;
Var i: byte; S:integer;
Begin
S:=0;
For i:=1 to n do S:=S+a[i];
Fsum:=S
End;
 	
Var a: ma; i: byte; sum:integer;
 	
Begin
Write(‘n=’); readln(n);
Writeln(‘A(’,n,’):’);
For i:=1 to n do read(a[i]);
Readln;
 	
Writeln(‘procedure psum’);
psum(n, a, sum);
Writeln(‘sum = ’, sum);
 	
Writeln(‘function fsum’);
Sum:=**fsum**(n, a);
Writeln(‘sum = ’, sum);
 	
Write(‘Press Enter’); readln;
End.
	Program prog2;
	{$APPTYPE CONSOLE}
	Uses unit1;
	
	{все константы,
	типы, процедуры и функции
	
	в отдельном модуле (unit)}
	
	Var a: ma; i: byte; sum:integer;
	
	Begin
	Write(‘n=’); readln(n);
	Writeln(‘A(’,n,’):’);
	For i:=1 to n do read(a[i]);
	Readln;
	
	Writeln(‘procedure psum’);
	psum(n, a, sum);
	Writeln(‘sum = ’, sum);
	
	Writeln(‘function fsum’);
	Sum:=**fsum**(n, a);
	Writeln(‘sum = ’, sum);
	
	Write(‘Press Enter’); readln;
	End.
		Unit unit1;
		Interface
		
		Const
		nmax=20;
		Type
		ma = array[1..nmax] of shortint;
		
		Procedure psum(const n: byte;
		var a: ma;
		out S:integer);
		Function fsum(const n: byte;
		var a: ma): integer;
		Implementation
		
		Procedure psum;
		Var i: byte;
		Begin
		S:=0;
		For i:=1 to n do S:=S+a[i];
		End;
		
		Function fsum;
		Var i: byte; S:integer;
		Begin
		S:=0;
		For i:=1 to n do S:=S+a[i];
		Fsum:=S
		End;
		
		End.
		
		В реализационной части допустимо опускать параметры процедур и функций, а также тип результата функции.

Пример 2:

<table>
<tr><th>Программа без модуля</th><th>Программа
с подключенными модулями</th><th>Модули с константами и типами, процедурами, функциями.</th></tr>
<tr><td rowspan="4">

```
Program prog1;
{$APPTYPE CONSOLE}

Const
  nmax=20;
Type
  ma = array[1..nmax] of shortint;

Procedure psum(const n: byte;
               var a: ma;
               out S:integer);
Var i: byte;
Begin
  S:=0;
  For i:=1 to n do S:=S+a[i];
End;

Function fsum(const n: byte;
              var a: ma): integer;
Var i: byte; S:integer;
Begin
  S:=0;
  For i:=1 to n do S:=S+a[i];
  Fsum:=S
End;

Var a: ma; i: byte; sum:integer;

Begin
  Write('n='); readln(n);
  Writeln('A(',n,'):');
  For i:=1 to n do read(a[i]);
  Readln;

  Writeln('procedure psum');
  psum(n, a, sum);
  Writeln('sum = ', sum);

  Writeln('function fsum');
  Sum:=fsum(n, a);
  Writeln('sum = ', sum);

  Write('Press Enter'); readln;
End.
```

</td><td rowspan="4">

```
Program prog3;
{$APPTYPE CONSOLE}
Uses unitTyp, unitProc, unitFun;

{все константы, типы,
процедуры и функции
в отдельных модулях (unit)}

Var a: ma; i: byte; sum:integer;

Begin
  Write('n='); readln(n);
  Writeln('A(',n,'):');
  For i:=1 to n do read(a[i]);
  Readln;

  Writeln('procedure psum');
  psum(n, a, sum);
  Writeln('sum = ', sum);

  Writeln('function fsum');
  Sum:=fsum(n, a);
  Writeln('sum = ', sum);

  Write('Press Enter'); readln;
End.
```

</td><td>

```
Unit unitTyp;
Interface

Const
  nmax=20;
Type
  ma = array[1..nmax] of shortint;

Implementation

End.
```

</td></tr>
<tr><td>

```
Unit unitProc;
Interface
Uses unitTyp;

Procedure psum(const n: byte;
               var a: ma;
               out S:integer);
Implementation

Procedure psum;
Var i: byte;
Begin
  S:=0;
  For i:=1 to n do S:=S+a[i];
End;

End.
```

</td></tr>
<tr><td>

```
Unit unitFun;
Interface
Uses unitTyp;

Function fsum(const n: byte;
              var a: ma): integer;
Implementation

Function fsum;
Var i: byte; S:integer;
Begin
  S:=0;
  For i:=1 to n do S:=S+a[i];
  Fsum:=S
End;

End.
```

</td></tr>
</table>

Местонахождение некоторых команд
Создать модуль (*unit*)
File → *New* → *Unit* (*Delphi for Win32*) или
File → *New* → *Other* → *New-Items* → *Delphi-Projects* → *Delphi-Files* → *Unit*
Добавить уже существующий модуль в проект:
Project → *Add to Project*
Удалить модуль из проекта:
Project → *Remove from Project*
Вставить *uses* в активный в редакторе кода файл можно вручную или из списка модулей проекта:
File → *Use Unit*…

Компиляция без запуска:
Project* → *Build – полная, с перекомпиляцией имеющегося исходного кода подключенных модулей и
Project* → *Compile **–** частичная, с перекомпиляцией только измененных файлов проекта.