8. Дизайн «сверху – вниз», модульное программирование, процедуры и функции.

Рассмотрим на примерах несколько проблем, связанных с повторным использованием одного и того же алгоритмического фрагмента.

Пример 1. Предположим, мы разработали программу для преобразования суммы денег из долларов в рубли. Естественно, удобно иметь интерфейс, позволяющий вводить различные суммы по очереди и получать результат. Следовательно, потребуется расширить задачу. Алгоритмическое описание структуры решения новой расширенной проблемы может выглядеть так:

Повторять
    Ввести величину суммы в долларах;
    Вычислить сумму в рублях;
    Запросить: «Будут ли еще данные для обработки? »
    Считать ответ
Пока – не ответ=нет

Так как алгоритм для преобразования суммы уже разработан, то его можно использовать непосредственно. Это повторное использо-вание, во-первых, бережет время, во-вторых, алгоритм уже проверен на корректность, что освобождает от дополнительной тестировки и уменьшает возможность ошибок.

Пример 2. В программе по алгоритму в нескольких местах решаются различные квадратные уравнения. Текст фрагмента алгоритма, и, следовательно, кода программы, решающего эту локальную проблему, придется вставить во всех соответствующих местах кода решения полной задачи.

Рассмотрим еще один пример. Пусть в программе надо выполнить несколько задач по выбору, в общем случае, независимых. Это можно организовать с помощью т.н. *меню* пользователя, например,

Меню:

1. Решение квадратного уравнения;
2. Решение линейного уравнения;
3.Закончить сеанс.

Выберите вариант нажатием на клавиши 1 — 3.

Пусть выбор считывается в переменной *«ответ»*. Тогда алгоритм для реализации такого меню может выглядеть так:

Повторять
Очистить экран дисплея;
Вывести текст меню на экран;
Считать *ответ*
Выбор по *ответ*.
1. {Считать коэффициенты;
Решить квадратное уравнение; Выдать результат;}
2. { Считать коэффициенты;
Решить линейное уравнение; Выдать результат;}
3. Завершить программу.
Конец (Выбора)
Пока—не Ложь

Этот алгоритм можно непосредственно реализовать на Турбо-Паскале с помощью оператора «Case». Цикл будет повторяться до тех пор, пока не будет выбран третий пункт меню, так как логическое условие «ложь» никогда не станет истинным. В каждый альтернативный пункт оператора выбора нужно подставить соответствующий программный текст, реализующий решение задачи, указанной в том пункте меню. При малом количестве вариантов выбора программа может еще сохранить читабельный вид, но если пунктов много, а каждый пункт, в свою очередь, выполняет сложную и большую задачу, то трудно будет охватить текст программы целиком, что, следовательно, затруднит его корректировку.

Наилучшим механизмом для повторных вызовов одного и того же алгоритма (примеры 1, 2), а также вызовов самостоятельных программных блоков в различных местах кода является механизм т.н. *модульного программирования*, в котором используется организация решения задачи в виде *процедур* и *функций*.

Процедура и функция, т.н. *подпрограмма* — это независимый блок программы. Подпрограмме назначается имя и передается список *входных параметров* (или *аргументов*), которые используются во время обработки данных и выполнения операторов, реализующих решение задачи. В общем случае подпрограмма может возвращать результирующие значения в *выходных параметрах*.

В приведенных примерах использование концепта подпрограммы может значительно упростить формирование и реализацию алгоритма. В каждом случае решения частных задач можно оформить как отдельную подпрограмму, предварительно проверенную и откорректированную, так, например,

- Блок преобразования суммы — как функцию;
- Блок решения квадратного уравнения — как процедуру;
- Решение задач каждого пункта меню — как процедуру.

В качестве параметра функции для перевода суммы в рубли можно передать величину суммы в долларах, а она возвратит вычисленное значение в рублях. Эту функцию затем можно будет вызывать в любом месте программы. Вышеприведенный алгоритм можно записать в виде:

Повторять
    Ввести величину суммы в долларах;
    Вычислить функцию dollarsTOrubles(dollars);
    Вывести значение функции dollarsTOrubles;
    Запросить: «Будут ли еще данные для обработки?»
    Считать ответ
Пока–не ответ=нет

Независимо от данной программы эту функцию можно использовать и в других программах, где требуется валютное

преобразование суммы. Другое преимущество данного подхода заключается в том, что валютный курс со временем может меняться, и тогда придется внести корректировку только в описание функции *dollarsTOrubles*.

Преимущество модульного программирования в основном заключается в том, что оно бережет время, так как, решив проблему один раз и оформив это решение в виде подпрограммы, ту же проблему для другого частного случая можно решить с помощью уже готовой подпрограммы с новыми параметрами. Кроме того, при таком подходе алгоритм нужно протестировать только один раз, что снижает возможность допущения ошибок.

Сформулируем основные преимущества модульного программирования более детально:

✓ При решении большая задача обычно разбивается на последовательность меньших под-задач, которые, в свою очередь, могут быть разбиты на еще меньшие под-задачи, и т.д. Такой подход обычно называется методом «сверху-вниз». Процесс разбиения продолжается до тех пор, пока под-задачи не станут настолько малыми, что могут быть решены одним программистом. Важность подхода «сверху-вниз» появляется в случаях, когда проблемная задача должна быть разработана группой программистов, каждая из которых реализует определенную часть системы, оформленную в виде одной или нескольких подпрограмм. Работая над одной подпрограммой, программист может сконцентрировать внимание на решении только этой проблемы, что дает больше возможности избежать ошибок. Кроме того, подпрограмму можно протестировать на корректность независимо от других частей главной программы.

✓ При подходе «сверху-вниз» может оказаться, что в конкретном учреждении некоторые проблемы возникают очень часто. Например, часто требуется сортировка файла данных по какому-либо признаку. Таким образом, можно построить библиотеку таких часто используемых подпрограмм, к которым можно будет обращаться из различных программ. Такой подход явно бережет время и уменьшает возможность ошибок, так как используются уже протестированные подпрограммы.

✓ Существует много специальных проблемных сфер, и не каждый программист может и должен знать все из них. Например, про-

граммист, работающий над научными прикладными программами, часто использует математические функции, такие, как синус, косинус, но может не знать, как их написать. Аналогично, программист, работающий в коммерческой прикладной сфере, может не знать, как разработать алгоритмы различной эффективности для сортировки, однако специалист может их написать и поместить в публичную библиотеку функций и процедур, которой смогут пользоваться все программисты.

8.1. Процедуры и функции, определенные пользователем.

Турбо-Паскаль позволяет пользователю оформлять свои собственные процедуры/функции – самостоятельные блоки, решающие локальные задачи, которые являются составными частями какой-то большой проблемы.

Процедура/функция определяется по имени и состоит из группы операторов, заключенных в программные скобки «Begin End». Фактически, подпрограмма имеет ту же структуру, что и главная программа. Процедура/функция начинает выполняться после ее вызова из главного кода, т.е. указания имени с параметрами, если они присутствуют. Подпрограммы поддерживают структуру дизайна программы, позволяя самостоятельное развитие ее отдельных частей. Как уже было отмечено, основное различие между процедурой и функцией заключается в том, что функция является носителем значения величины, и, следовательно, может стоять на месте переменной, а процедура просто выполняет последовательность операций и может стоять на месте оператора.

Таким образом, в Турбо-Паскале поддерживается два основных типа подпрограмм – процедура и функция. Рассмотрим каждый из них по-отдельности.

8.1.1. Простые процедуры без параметров.

Процедуры без параметров выполняют определенную последовательность операций. Например, следующая процедура *lines4* выводит на экран четыре пустые строки:

```
Procedure lines4;
Begin
    Writeln; Writeln; Writeln; Writeln;
End; { lines4 }
```

Чтобы процедура выполнила те операции, которые заложены в ее *теле,* ее надо вызвать из главной программы, например,

```
Begin
    Writeln(' Это первая строка вывода!');
    lines4;
    Writeln(' Это шестая строка вывода!');
End.
```

При этом, описание процедуры должно предшествовать телу главной программы, оно располагается в блоке объявления. Полный код программы будет иметь вид:

```
Program proc_example;
{ Описание процедуры }
Procedure lines4;
{ Пропускает четыре пустые строки на экране }
Begin
    Writeln; Writeln; Writeln; Writeln;
End; { lines4 }

Begin { Главная программа }
    Writeln(' Это первая строка вывода!');
    Lines4;
    Writeln(' Это пятая строка вывода!');
End.
```

Рекомендуется давать в комментариях хотя бы минимальную информацию о цели и назначении каждой процедуры, хотя ее имя может подсказывать это по смыслу.

Когда главная программа начинает выполняться и доходит до оператора вызова процедуры, активизируется тело процедуры, выполняются все ее операторы, и после ее завершения управление передается опять главной программе.

Процедура может объявить собственные переменные. Они доступны и определены только в пределах данной процедуры и теряют смысл и значение в главной программе. Такие переменные называются *локальными,*

Рассмотрим пример:

```
Procedure lines4;
{ Пропускает четыре пустые строки на экране }
Var i:Integer;
Begin
    For i:=1 to 4 Do Writeln;
End; { lines4 }
```

Локальные переменные (а также другие элементы программы, такие, как константы или другие процедуры или функции), объявленные в пределах процедуры, теряют свои значения вне процедуры. Для их идентификаторов можно использовать любое имя, даже если оно объявлено в главной программе или других процедурах. Изменение значений локальной переменной в пределах процедуры никак не отразится на значении одноименной переменной из другого программного блока, и наоборот, изменение значения одноименной переменной из глобального блока не отразится на переменной внутри процедуры.

С другой стороны, переменные, которые объявляются вне пределов любой процедуры, называются *глобальными* и доступны также внутри процедуры, но использование внешних переменных не рекомендуется, так как в больших программах трудно контролировать, какие именно переменные являются общими и какие из них обновляются в процессе выполнения процедуры.

Таким образом, общий формат описания процедуры без параметров имеет вид:

```
Procedure Имя — процедуры;
```

```
{ Блок локальных объявлений }
Begin
    { Тело Процедуры }
    End;
```

8.1.2. Процедуры с параметрами.

Процедура *lines4,* рассмотренная в предыдущем разделе, была бы гораздо более полезней, если бы можно было задавать количество пропускаемых пустых строк, что давало бы больше гибкости и удобства. Эту возможность, т.е. возможность передавать процедуре входные значения внешних переменных и данных, можно осуществить с помощью *аргументов, параметров процедуры.* Например, в нашем примере процедуре можно передать количество пустых строк для вывода на экран, скажем *n.* Теперь описание процедуры будет иметь вид:

```
Procedure lines(n:Integer);
{ Пропускает n пустых строк на экране }
Var i:Integer;
Begin
    For i:=1 to n Do Writeln;
End; {lines }
```

Как и в предыдущем случае, эта подпрограмма также не возвращает значения. Ей передается целый параметр *n,* который определяет количество пропускаемых строк. Список параметров задается перечислением идентификаторов *формальных* аргументов с указанием типа: знак — разделитель — двоеточие «:». Вызов процедуры осуществляется так же, как и в предыдущем случае, только с заданием значения для параметра, например,

```
Var m,n:Integer;
Begin
…
m :=6;
n:=3;
…
```

```
lines(n);
...
lines(n+m);
...
End.
```

Однако вызов

```
lines(4.0);
```

недопустим, так как тип фактического аргумента не совпадает с типом формального параметра, объявленного в заголовке процедуры.

Таким образом, общий формат описания процедуры с параметрами имеет вид:

```
Procedure Имя — процедуры(Список —параметров:Тип);
{ Блок локальных объявлений }
Begin
    { Тело процедуры }
End;
```

А общие правила формирования и вызова процедур заключаются в следующем:

- Параметры, к которым обращается процедура, заключаются в скобки;
- Объявление параметров располагается в заголовке процедуры между ее именем и символом «;».
- Когда процедура вызывается, за ее именем следуют круглые скобки;
- Передаваемые параметры располагаются между этими круглыми скобками;
- Параметры перечисляются в том же порядке, в каком они указаны в заголовке описания процедуры.

Программа-пример: Суммирование двух целых чисел.

```
Program add_numbers;
```

```
{ Суммирует два целых числа }
Var number 1, number2 : Integer;

    Procedure calc_answer(first, second : Integer );
    Var result: Integer;
    Begin
        result := first + second;
        Writeln('Сумма равняется = ', result)
        End;
    Begin
        Write(' Введите два целых числа:');
        Read(numberl, number2);
        calc_answer(numberl, number2)
End.
```

8.1.3. Процедуры и параметры-значения (копии).

В рассмотренных выше примерах процедурам передаются не сами параметры, а их копии. Значения соответствующих переменных могут изменяться в теле процедуры, но это не отразится на значениях переменных — оригиналов.

Рассмотрим предыдущий пример с некоторыми изменениями:

```
Program valueParameters;
Var number1, number2 : Integer;

Procedure calc_answer(numberl, number2 : Integer );
{ Суммирует два целых числа }
Var result: Integer;
Begin
    result := number1+ number2;
    Writeln(' Сумма равняется = ', result);
    { Изменение значений локальных переменных }
    Inc(numberl);
    Dec(number2);
{Вывод измененных значений локальных переменных}
      Writeln('Значения переменных number1 и number2
```

```
                внутри процедуры: ', number1,' ', number2);
End;

Begin    { Главная программа }
    Write(' Введите два целых числа:');
    Read(number1, number2);
    { Вывод значений переменных—оригиналов }
    calc_answer(number1, number2);
    Writeln(' Значения переменных number1 и number2 в
    ' главной программе:', number1 ,' ', number2);
End.
```

Несмотря на то, что в теле процедуры значения переменных number1 и number2 изменяются, главная программа выведет на экран именно те значения для них, которые были введены пользователем на запрос оператора «Read»,

Таким образом, параметры-значения используются только для передачи данных процедуре. Процедура не может влиять на значения таких параметров вне ее пределов, что, следовательно, предотвращает нежелательные побочные случайные эффекты при вызове процедуры из главной программы.

8.1.4. Функция-подпрограмма, возвращающая значение.

Подпрограмма, возвращающая значение, которое является функцией ее аргументов, называется *функцией.* В этом случае дополнительно определяется тип возвращаемого результата. Рассмотрим функцию, которой передаются координаты точки (x,y), и которая возвращает расстояние от начала координат. Она будет записываться следующим образом:

```
Function distance(x,y:Real):Real;
{ Вычисляет расстояние от (x,y) до начала координат }
Var dist:Real; { Локальная переменная }
Begin
    dist:=Sqrt(x*x + y*y);
```

```
    distance: = dist;
End; { distance }
```

Отметим следующие моменты:

• Заголовок описывается ключевым словом «Function»;

• В заголовке же объявляется тип функции, т.е. тип возвращаемого результата;

• Функции передаются два параметра, они перечисляются через запятую;

• Используется локальная переменная *dist* для временного хранения значения функции. В данном простом примере это не обязательно, но в общем случае такая страховка предотвращает нежелательные побочные эффекты. А здесь можно было прямо написать:

```
distance:=Sqrt(x*x + y*y);
```

• Когда функция вызывается из главной программы, формальные параметры заменяются фактическими значениями действительного типа, при этом первым аргументом идет *x*-координата, а вторым – *y*.

• Так как функция возвращает значение, она может быть использована в выражениях только в качестве переменной.

Общий формат объявления функции имеет вид

```
Function Имя-Функции (Список-параметров): Тип;
```

Отметим, что тип функции может быть только скалярным или строковым (о типах смотрите ниже), Недопустимо, например, определить функцию Типа массива, записи, файла или других структурированных типов. Передаваемые параметры могут быть структурированными (и это также касается процедур), только эти типы должны быть предварительно объявлены в разделе «Type». Например, следующее перечеркнутое объявление нелегально:

```
Function time (t:record hour, min, sec: Integer End): Boolean;
```

И наконец, приведем примеры вызова и использования функций:

```
Var a,b,c,d,x,y:Real;
...
    a:-3.0;
    b:=4,4;
    c := 5.1;
    d:=2.6;
    x = distance(a, b);
    y = distance(c, d);
    ...
    if(distance(3, 6) > distance(x, y)) Then Writeln('Сообщение 1');
     ...
```

Программа-пример: Вычисление квадратного корня.

```
Function sqRoot(x,accuracy:Real):Real;
{ Вычисляет квадратный корень с точностью до accuracy }
{ Если x<0, возвращает 0 }
Var xold, xnew: Real; { Локальные переменные }
Begin
    if(x <= 0) Then sqRoot:=0
    Else
    Begin
        xold := x; { x как первое приближение }
        xnew :==0.5*(xold+x/xold);
        While Abs((xold-xnew)/xnew) > accuracy Do
        Begin
            xold := xnew;
            xnew := 0.5*(xold+x/xold);
        End; { While }
        sqRoot := xnew;
     End; { If-Else }
End; { sqRoot}
```

Программа-пример: Сумма целых чисел.

Следующая функция возвращает сумму первых *n* целых чисел, где *n* передается как параметр:

```
Function summa(n:Integer):Integer;
{ Суммирует n целых чисел }
Var sum, i:Integer;
Begin
    sum:=0;
    For i:= 1 to n Do sum := sum + i;
    summa:=sum;
End; { summa }
```

Пример вызова функции из главной программы:

```
Program testSum;
Var sum,i:Integer;
Begin
    Write(' Введите целое число:');
    Read(n);
    Writeln(' Сумма чисел от 1 до ', n,' = ', summa(n));
End.
```

Программа-пример: Возведение в степень.

Функция *power* возвращает степень ее первого входного параметра, где величину степени задает второй входной параметр. При этом, второй параметр целочисленного типа и может принимать значение 0 или быть отрицательным.

```
Function power(x:Real; n: Integer): Real;
{ Возведение в степень }
Var prod:Real;
    i: Integer;
```

```
Begin
    prod:=l;
    If n<>0
    Then
    Begin
    For i:= 1 to abs(n) Do prod:=prod*x;
            If n<0 Then prod:=l/prod;
            power :=prod;
    End
    Else power:=0;
End; { power }
```

Пример вызова функции из главной программы:

```
Program testPower;
Var x,pr:Real;
    n:Integer;
Begin
    Write(' Введите основу и степень (целое число):');
    Read(x,n);
    Writeln( x:7:3,' в степени ', n,' равняется ', power(x, n));
End.
```

8.1.5. Процедуры и параметры-переменные (оригиналы).

До сих пор все рассмотренные параметры были входными. Значения соответствующих переменных могут изменяться в теле процедуры, но это не отразится на значениях переменных – оригиналов. Это происходит из-за того, что подпрограмма оперирует не самими фактическими параметрами, а копиями их значений.

Единственный способ получить обратно информацию, определенную в результате выполнения подпрограммы, – это использование функции, которая возвращает одно значение скалярного или строкового типа. Но часто требуется, чтобы подпрограмма возвращала несколько значений или значения структурированных данных. Например, подпрограмме задается

сумма денег в копейках, а она должна возвратить сумму в рублях и копейках.

Если требуется, чтобы подпрограмма возвращала значение параметра, то ей нужно передать ссылку на фактический параметр, т.н. *параметр—переменную* (оригинал). Для этого перед их объявлением в заголовке процедуры (в скобках) нужно вставить ключевое слово «Var». В результате соответствующие переменные будут восприниматься компилятором как оригиналы, а не как копии.

Рассмотрим пример процедуры для решения квадратного уравнения

```
Procedure sqEqSolution(
    a, b, c:Real; { Входные параметры }
    Var rootl,root2:Real { Выходные параметры } );
{ Процедура решает квадратное уравнение }
{ Если действительных корней нет, выдается сообщение }
Var discr:Real; { Локальная переменная }
Begin
    discr := b*b-4*a*c;
    if(discr<0)
    Then Begin
            Writeln(' Действительных корней нет! ');
            Exit
        End
    Else Begin
            rootl := (-b + Sqrt(discr))/(2*a);
            root2:= (-b - Sqrt(discr))/(2*a);
        End;
    End; { sqEqSolution }
```

Отметим, что корни *(roots)*, которые являются выходными параметрами, объявляются как параметры-переменные, однако коэффициенты являются входными параметрами, и, следовательно, объявляются как параметры-значения.

Процедура может быть использована в главной программе следующим образом:

```
Var a, b, c, r1, r2:Real;
```

```
Begin
    Write('Введите коэффициенты квадратного уравнения:');
    Read(a,b,c);
    SqEqSoIution (a,b,c,rl,r2);
    If rl=r2
        Then Writeln('Один действительный корень:', rl:10:3,'.')
        Else Writeln('Корни: rl =', rl: 10:3,',  r2 = ', r2:10:3,'.');
End.
```

Подпрограмму можно было оформить как логическую функцию, которая, в случае мнимых корней, возвращала бы значение «ложно», а иначе – «истинно». При отрицательном дискриминанте действие функции ограничивается только присвоением ей значения *False,* а в случае наличия действительных корней генерируется значение *True* и дополнительно вычисляются и корни:

```
Var a, b, c, rl, r2:Real;

Function sqEqSolutionFun (
    a, b, c:RealI; { Входные параметры }
    Var root 1,root2: Real { Выходные параметры }
        ):Boolean; { Тип функции }
{ Функция решает квадратное уравнение }
{ Если действительных корней нет, }
    { генерируется значение False }
Var discr:Real; { Локальная переменная }
Begin
    discr := Sqr(b) - 4*a*c;
    If discr<0 Then
        Begin
                sqEqSolutionFun:=FaIse;
                Exit; { Выход из функции }
        End
        Else
        Begin
                sqEqSolutionFun:=True;
```

```
                root1: (-b + sqrt(discr))/(2 * a);
                root2 := (-b - sqrt(discr))/(2 * a);
            End
End; { sqEqSolutionFun }

Begin
    Write(' Введите коэффициенты a,b,c:');
    Read(a,b,c);
    if SqEqSolutionFun (a,b,c,rl,r2)
    Then If rl=r2
        Then Writeln('Один корень: ', rl:10:3,'.')
        Else Writeln('Корни:rl = ', rl:10:3,', r2 = ', r2:10:3,'.')
    Else Writeln( Действительных корней нет!');
End.
```

Различие между параметрами-значениями и параметрами-переменными хорошо видно на примере процедуры, которая взаимозаменяет значения своих параметров:

```
Var number 1, number2 : Integer;

Procedure swap ( var valuel, value2 : Integer );
Var temp : Integer;
Begin
    temp := valuel;
    valuel := value2;
    value2 := temp
End;

Begin
    number1 := 10;
    number2 := 33;
    Writeln('Число первое =', number1,', второе = ', number2 );
    swap( number1, number2);
    Writeln('После обмена:');
    Writeln("Число первое = ', number1,', второе = ', number2 );
End.
```

Программа выведет на экран следующий результат:
Число первое = 10, второе = 33
Число первое = 33, второе = 10

Если бы *value1* и *value2* были объявлены как параметры—значения, то результат вывода первой процедуры «Writeln» не отличался бы от результата второй.