

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА

Л.С. Зеленко, Т.И. Михеева

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ПРОГРАММИРОВАНИЕ (II часть)

*Курс лекций
для студентов заочной формы обучения*

УДК 681.3

Алгоритмические языки и программирование. II часть: Курс лекций / Л.С. Зеленко, Т.И. Михеева. Самар. гос. аэрокосм. ун-т. Самара, 2001. С. 122.

ISBN 5-7883-0164-5

Дано описание основных возможностей процедурного языка высокого уровня Паскаль (версия 7.0) для персональных компьютеров, работающих под управлением операционной системы MS DOS. Изложены основные приемы программирования, особенности создания программ, приведено большое количество примеров.

Курс лекций предназначен для студентов заочной формы обучения специальности 22.02 – «Автоматизированные системы обработки информации и управления». Данный курс лекций будет также полезен студентам других специальностей, знакомых с основами программирования на алгоритмических языках и занимающихся как по очной, так и заочной форме обучения. Работа подготовлена на кафедре ИСТ.

Печатается по решению редакционно-издательского совета Самарского государственного аэрокосмического университета имени академика С.П. Королева.

Рецензент канд. техн. наук, доц. Л.А. Ж а р и н о в а

ISBN 5-7883-0164-5

© Зеленко Л.С., Михеева Т.И., 2001

© Самарский государственный
аэрокосмический университет, 2001

ОГЛАВЛЕНИЕ

1. ПОНЯТИЕ ПОДПРОГРАММЫ: ПРОЦЕДУРЫ И ФУНКЦИИ.....	6
1.1. Процедуры.....	7
1.1.1. Описание процедуры.....	7
1.1.2. Вызов процедуры.....	9
1.2. Функции.....	10
1.2.1. Описание функции.....	10
1.2.2. Вызов функции.....	11
1.2.3. Побочный эффект функции.....	15
1.3. Локальные и глобальные параметры.....	17
1.4. Передаваемые параметры процедур и функций.....	21
1.4.1. Параметры-значения.....	23
1.4.2. Параметры-переменные.....	24
1.4.3. Процедуры и функции как параметры.....	34
1.4.4. Бестиповые параметры процедур и функций.....	40
1.5. Процедура или функция?.....	43
1.6. Опережающее описание процедур и функций.....	45
1.7. Внешние процедуры и функции.....	46
1.8. Рекурсивные процедуры и функции.....	48
1.8.1. Когда не нужно использовать рекурсию.....	49
2. РАСПРЕДЕЛЕНИЕ ПАМЯТИ ПРИ ВЫПОЛНЕНИИ ПРОГРАММ..	54
2.1. Распределение памяти при работе подпрограмм.....	58
3. МОДУЛИ.....	61
3.1. Структура модулей.....	61
4. ВВОД-ВЫВОД И ФАЙЛОВАЯ СИСТЕМА.....	63
4.1. Понятие формального файла.....	64
4.2. Физические файлы.....	65
4.3. Буфер ввода-вывода.....	66
4.4. Организация работы с файлом.....	67

4.5. Запись в файл.....	67
4.6. Чтение из файла	69
4.7. Файловые типы языка Паскаль	71
4.8. Общие процедуры работы с файлами.....	72
4.8.1. Процедура связывания	73
4.8.2. Процедура открытия.....	74
4.8.3. Процедура закрытия	75
4.8.4. Процедура переименования.....	76
4.8.5. Процедура удаления	78
4.8.6. Функция анализа состояния файла	78
4.9. Текстовые файлы.....	80
4.9.1. Чтение и запись в текстовых файлах	82
4.9.1.1. Процедуры Read / ReadLn	83
4.9.1.2. Комплексный ввод	85
4.9.1.3. Процедуры Write / WriteLn.....	86
4.9.1.4. Форматы вывода данных.....	87
4.9.2. Процедуры и функции для работы с текстовыми файлами. 89	
4.9.2.1. Процедура установки размера буфера.....	90
4.9.2.2. Процедура освобождения буфера.....	93
4.9.2.3. Процедура открытия файла для дозаписи.....	93
4.9.2.4. Функции анализа состояния текстового файла	94
I. Функция EOLn.....	94
II. Функция SeekEOLn	96
III. Функция SeekEOF.....	96
4.10. Компонентные файлы.....	97
4.10.1. Чтение и запись в компонентном файле	99
4.10.2. Прямой и последовательный доступ к файлам	102
4.10.3. Функции определения размера и позиции в файле	104
4.10.4. Процедура позиционирования	105

4.10.5. Процедура отсечения конца файла	108
4.11. Нетипизированные файлы	109
4.11.1. Открытие нетипизированного файла	110
4.11.2. Чтение и запись в нетипизированном файле	110
4.12. Примеры программ	114
СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ.....	120
ПЕРЕЧЕНЬ ИСПОЛЬЗУЕМЫХ ОБОЗНАЧЕНИЙ	120

1. ПОНЯТИЕ ПОДПРОГРАММЫ: ПРОЦЕДУРЫ И ФУНКЦИИ

В основе конструкций языка Паскаль, называемых процедурами и функциями, лежит понятие *подпрограммы*. В настоящее время оно трактуется не только как средство, позволяющее избежать повторных записей одинаковых частей программы, а как одна из базовых управляющих структур в программировании, позволяющая реализовать нисходящую концепцию при *структурном программировании*.

В процессе совершенствования искусства программирования разработчики программ создавали их методом последовательных уточнений, который заключается в следующем: на каждом этапе задачу разделяют на несколько подзадач, определяя тем самым некоторое количество компонент, логически связанных между собой, но имеющих самостоятельное значение. Концепция процедуры позволяет выделить подзадачу как явную подпрограмму, делает структуру задачи более понятной и дает возможность вести разработку одновременно коллективу программистов.

Подпрограмма является одним из фундаментальных инструментов, оказывающих влияние на стиль, качество и надежность разработки программных средств. Поэтому не следует колебаться: оформлять или нет некоторое действие как подпрограмму (даже когда к ней обращаются лишь единожды), - оформлять. Использование подпрограмм приводит к улучшению «читаемости» программы: как правило, в коротких блоках разбираться легче, чем в длинных. При разделении этапов разработки на подпрограммы программу будет легче передать другому человеку или проверить ее.

Оформляя фрагмент программы как подпрограмму, программист не только экономит на времени написании программы, но и на памяти, занимаемой программой (механизм распределения памяти между переменными рассмотрим чуть позже).

Механизм подпрограмм – это мощное средство абстракции, позволяющее расширять язык путем включения новых операций и операторов. Записывая

вызов подпрограммы, мы исходим из того, что подпрограмма делает, и не уделяем внимания тому, как она это делает реально.

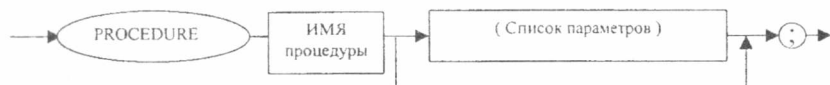
В Паскале подпрограммы делятся на две группы: процедуры и функции, которые имеют много общего и некоторые особенности. Рассмотрим каждую группу подпрограмм в отдельности.

1.1. Процедуры

Процедура – это специальным образом оформленная последовательность операторов, которой присвоено имя. Процедура хранится (описывается) в декларативной части программы, программа по отношению к ней выступает как внешняя. Процедура может быть вызвана для выполнения в исполнительской части программы из любых ее точек по имени.

1.1.1. Описание процедуры

Процедуры вводятся в программу после своего описания в декларативной части программы (см. п.1.3, часть I). Описание процедуры служит для определения части программы и сопоставления с ней некоторого имени, так что эту часть можно затем активировать по ее имени. Описание выглядит как программа, но вместо заголовка программы употребляется заголовок процедуры. Синтаксическая диаграмма заголовка процедуры приведена ниже:



Описание процедуры начинается с зарезервированного слова *Procedure*, за которым следует имя процедуры, которое выбирается в соответствии с правилами определения имен (идентификаторов). Далее в скобках идет список параметров. Список параметров может быть пустым, в этом случае отсутствуют и сами скобки. Через параметры процедура осуществляет связь с программой и

другими процедурами. Список параметров представляет собой список идентификаторов с указанием их типов.

Procedure <Имя_Процедуры> [(<Список_Параметров>)] ;

Пояснение: квадратные скобки в описании показывают необязательность этой части оператора.

Содержательная часть процедуры представляет собой блок, который содержит раздел описания данных (декларативная часть) и раздел операторов (исполнительная часть).

```
Procedure <Имя_Процедуры> ( ПараметрЗначение1 : ТипЗначение1;
                           ПараметрЗначение2 : ТипЗначение2;
                           Var ПараметрПеременная1 : ТипПеременная1;
                           Var ПараметрПеременная2 : ТипПеременная2;
                           ... ) ;
```

Label <Метки_Используемые_в_Процедуре>	}	<i>Декларативная часть процедуры</i>
Const <Константы_Используемые_в_Процедуре>		
Type <Типы_Используемые_в_Процедуре>		
Var <Переменные_Используемые_в_Процедуре>		
<Описание вложенных процедур и (или) функций>		
Procedure...		
Function...		

Begin	}	<i>Исполнительная часть процедуры</i>
< Тело процедуры >		
End; {Имя процедуры}		

Структура процедуры копирует структуру программы в целом, за исключением завершающей процедуру точки с запятой вместо точки после последнего *End* программы. Порядок следования разделов описаний подчиняется тем же правилам, по которым строится вся программа.



Имя процедуры в программе должно быть уникальным, как и любой идентификатор программы. Область его действия – блок программы, содержащий данное описание процедуры. Он должен отличаться от всех идентификаторов блока.



Идентификаторы, описанные в заголовке процедуры в качестве формальных параметров, не должны описываться в декларативной части процедуры.

Таким образом, при описании процедуры определяются все *объекты*, с которыми она работает, и *действия* над этими объектами. Описание процедуры, расположенное в декларативной части программы, само по себе никакого действия не производит.

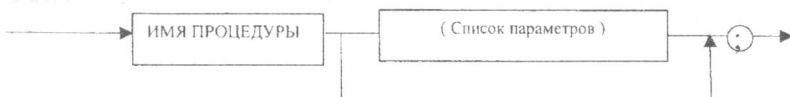
1.1.2. Вызов процедуры

Процедура может быть активирована в исполнительной части программы по ее имени. При вызове процедура будет содержать список параметров, если он присутствовал при описании этой процедуры в декларативной части. Типы используемых параметров при вызове процедуры не указываются.

<Имя_Процедуры> [(<Список_Параметров>)] ;

Пояснение: квадратные скобки в описании показывают необязательность этой части оператора.

Синтаксическая диаграмма, описывающая вызов процедуры, приведена ниже:



Например, необходимо написать процедуру вычисления корней квадратного уравнения $ax^2 + bx + c = 0$. Заголовок этой процедуры будет выглядеть следующим образом:

Procedure Roots (A,B,C : Real; Var Root1,Root2 : Real);

где *A*, *B* и *C* – коэффициенты квадратного уравнения, *Root1* и *Root2* – корни этого уравнения.

Вызов процедуры осуществляется по ее имени.

Roots (A, B, C, Root1, Root2);

Параметры при вызове отделяются друг от друга запятыми, а весь список заключается в скобки.

При отсутствии параметров в процедуре их не должно быть и при ее вызове. Например, процедура вывода титульной страницы на экран содержит постоянные сведения, не зависящие от каких-либо параметров, следовательно, при описании процедуры ее можно объявить как

Procedure Title; и вызвать по имени **Title**.

1.2. Функции

Функция – это часть программы (в некотором смысле аналог процедуры), определяющая единственное скалярное (целое, символьное, логическое, вещественное), строковое или ссылочное значение, являющееся результатом работы функции.

1.2.1. Описание функции

Все сказанное ранее о процедурах справедливо и для функций. Вместе с тем имеется ряд отличий. Одно из них чисто внешнее: заголовок начинается с зарезервированного слова *Function*.

Синтаксическая диаграмма заголовка функции приведена ниже:



Как и у процедуры, список параметров функции может быть пустым, в этом случае отсутствуют и сами скобки.

Function <Имя_Функции> [(<Список_Параметров >)] :

<Тип_Результата>;

Пояснение: квадратные скобки в описании показывают необязательность этой части оператора.

Содержательная часть функции, как и у процедуры, представляет собой блок, который содержит декларативную и исполнительную часть.

Другие отличия более существенны:

1. В заголовке функций *явно* указывается тип результата, вычисленного с помощью этой функции. Указатели функций используются в выражениях, следовательно, при анализе выражения обязательно знать тип результата, вычисляемого функцией.
2. В теле функции обязательно должен присутствовать оператор присваивания, в левой части которого стоит идентификатор этой функции. Такое присваивание «возвращает» результат функции. Если такой оператор отсутствует, то значение, вычисляемое функцией, не определено, что приводит к аварийному завершению программы.
3. Если имя функции появляется где-либо в выражении внутри самой функции, то речь идет об ее рекурсивном выполнении.

1.2.2. Вызов функции

Функция активируется в исполнительной части программы по ее имени. Функция может участвовать в выражениях в качестве операнда. В операторе присваивания (при нерекурсивном вызове) функция стоит в правой его части. При вызове функция будет содержать список параметров, если он присутствовал при ее описании в декларативной части программы. При вызове функции типы передаваемых параметров не указываются.

<Переменная_Типа_Результат_Функции>

:= <Имя_Функции> [(<Список_Параметров >)] ;

Пояснение: квадратные скобки в описании показывают необходимость этой части оператора.

Например, необходимо написать функцию вычисления среднего арифметического элементов вектора (рис. 1.1).

```

{===== Среднее арифметическое элементов вектора =====}
Const
    n_max = 50;           { Количество элементов вектора }
Type
    Vector = Array [1..n_max] of Real;
Var b : Vector;
    Number : Byte;         { Количество элементов }
    Aver : Real;           { Среднее арифметическое }
{=====Вычисление среднего арифметического вектора=====}
Function Average( n : Byte; { Заданное количество элементов }
                 Var a: Vector): Real; { Вектор }
Var
    I : Byte;             { Параметр цикла }
    Sum : Real;           { Сумма элементов вектора }
Begin
    Sum := 0;
    For i := 1 to n do
        Sum := Sum + a[i]; { Нахождение суммы вектора }
    Average := Sum / n;    { Нахождение среднего арифметического }
                        { Возвращение результата через имя }
End { Average };
Begin
    { Главный begin программы }
    Readln (Number);       { Ввод рабочего размера вектора }
    For i := 1 to Number do { Ввод элементов вектора }
        Readln (b[i]);

```

Рис. 1.1. Среднее арифметическое элементов вектора (см. с. 13)


```

{===== Первый способ получения результата =====}
Aver := Average (Number, b); { Активация функции }
                                { Сохранение результата в }
                                { переменную Aver }
Write ('Среднее арифметическое ', Number);
Write (' элементов вектора = ', Aver:12:4);

{===== Второй способ получения результата =====}
Write ('Среднее арифметическое ', Number);
Write (' элементов вектора = ', Average (Number, b) :12:4);
{ Результат вычислен временно и нигде не сохранен!!! }
End.

```

Рис.1.1. Окончание

При работе с функциями следует иметь в виду, что функция не задает какого-то логически завершенного этапа вычислительного процесса - она задает лишь правило вычисления некоторого значения, принимаемого в качестве значения определяемой функции, ничего не говоря о том, что делать дальше с этим значением (и даже о том, где его следует сохранить для дальнейшего использования). Транслятор по своему усмотрению может решать вопрос о том, куда помещать вычисленное значение любой функции: в фиксированную ячейку памяти, в определенный регистр, который будет использоваться для этой цели, и т.д. Поэтому и вызов функции представляет вычисляемое значение, которое обычно используется в качестве операнда какой-либо операции. Необходимо четко помнить, что вызов функции может использоваться только в качестве компонента (возможно, единственного) какого-либо выражения.



Основным результатом функции не могут быть переменные производных типов: массивы, множества, записи, файлы (рис. 1.2).

{===== Получение нового вектора из старого =====}

Type

Vector = array[1..7] of Real; Синтаксическая ошибка:

~~Function NewVector~~

*Результатом функции
не может быть массив.*

~~(Var a : Vector): Vector;~~

Procedure NewVector

Правильное описание

(Var a, NewVec : Vector);

{===== Вставка числа в упорядоченный файл =====}

{===== и определение номера этого элемента в файле =====}

~~Function Insert~~

Синтаксическая ошибка:

~~(x : Byte;~~

*Результатом функции
не может быть файл.*

~~Var PosX : Word);~~

~~File_of_Byte;~~

Function Insert

Правильное описание

(Var f : File_of_Byte;

x : Byte):Word;

{===== Поиск компонента файла по заданному ключу =====}

Type

~~Rec = Record~~

~~Name : String;~~

~~X : Real;~~

~~End;~~

File_of_Rec = File Of Rec; Синтаксическая ошибка:

Function Search

*Результатом функции
не может быть запись.*

(Var F : File_of_Rec;

S : String) : Rec;

Рис. 1.2. Примеры описания функций с различными типами выходных параметров (см. с. 15)

Function Search**Правильное описание**

```
(Var F : File_of_Rec;  
    S : String;  
    Var R : Rec) : Boolean;
```

Рис. 1.2. Окончание

1.2.3. Побочный эффект функции

Поскольку свой единственный результат функция присваивает своему идентификатору, то все остальные параметры должны быть входными и (или) возвратными. Если операторы функции производят какие-то изменения с входными параметрами или у функции больше одного выходного параметра (см. рис. 1.2), то возникает то, что называется «побочным эффектом» (то есть эффектом, производимым функцией помимо своей основной работы — вычисления значения). Термин «побочный эффект» взят из фармакологии, когда при приеме лекарства при лечении какого-либо заболевания могут возникнуть побочные действия: лекарство либо способствует лечению и других болезней, либо обостряет их. При этом значимость побочного эффекта может не уступать главному эффекту. Как и у лекарств, побочные эффекты функций могут быть как весьма удобными и полезными, так и вызывать различные неприятности.

Побочный эффект функции может проявляться как в изменении значений глобальных переменных, так и в изменении фактических параметров, вызываемых по имени (за счет того, что в этом случае функция получает непосредственный доступ к этому фактическому параметру, может не только использовать его, но и изменять).



Вряд ли стоит рекомендовать использование функций с побочным эффектом, так как это часто затрудняет понимание

программы, значительно усложняет ее верификацию, снижает надежность.

Опасность и неприятность в использовании функций, дающих побочный эффект, иллюстрирует программа, приведенная на рис. 1.3.

```

{===== Побочный эффект функции =====}
Var
  a, z : Integer;           { Глобальные параметры }
Function Sneaky (x : Integer) : Integer;
Begin
  z := z + x;               { Побочный эффект для z }
  Sneaky := x;
End { Sneaky };
Begin                       { Главный begin программы }
{===== Первый способ получения результата =====}
  z := 10;   a := Sneaky (z);
  WriteLn (a:4, z:4);
  

_ _ 1 0 _ _ 2 0


{===== Второй способ получения результата =====}
  z := 10;   a := Sneaky (100);
  WriteLn (a:4, z:4);
  

_ 1 0 0 _ 1 1 0


```

Рис. 1.3. Побочный эффект функции (см. с. 17)

```
{===== Третий способ получения результата =====}
```

```
z := 10;    a := Sneaky (z);
```

```
a := a * Sneaky (100);
```

```
WriteLn (a:4, z:4);
```

```
1 0 0 0  _ 1 2 0
```

```
End.
```

Рис. 1.3. Окончание

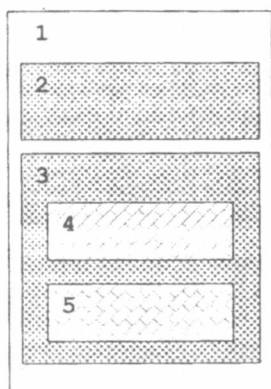
1.3. Локальные и глобальные параметры

Любая подпрограмма должна обладать определенной независимостью в смысле использования переменных, типов, констант, поэтому при использовании в программе процедур или функций происходит разделение данных и их типов на глобальные и локальные.

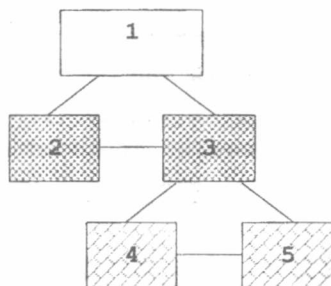
Локальные параметры подпрограмм — это константы, типы, переменные, процедуры и функции, описанные и работающие только внутри самой процедуры и никогда за ее пределами. К локальным также относятся данные, которыми процедура (функция) обменивается, т.е. объявленные в списке параметров.

Понятие «глобальные параметры» возникает вследствие блочной структуры программ (рис. 1.4). Каждая процедура (функция) представляет собой блок со своей декларативной частью и может содержать внутри этого блока описание других процедур и функций и обращения к ним.

Самый внешний (1), определяемый программой блок, содержит внутри себя подблоки (2, 3, 4, 5), реализованные с различным уровнем вложенности. При этом объекты, описанные внутри 2, 4 и 5-го блоков, являются локальными в своем блоке и недоступны внешним блокам. Объекты блока 3 являются с одной стороны локальными в своем блоке и глобальными для блоков 4 и 5.



а



б

Рис. 1.4. Блочная структура программы:

а - блочная структура программы, б - иерархия блоков

Уровень вложенности процедур и функций практически не ограничен, но на практике использовать больше двух не рекомендуется.



В случае совпадения имен локальных и глобальных параметров локальное имя экранирует глобальное.

Изменение глобальной переменной в процедуре вызывает соответствующие изменения и вне области ее действия. В этой связи, если в процедуре (функции) используется цикл, то параметр цикла должен быть описан внутри процедуры как локальный параметр. Это позволит избежать семантических ошибок при циклическом вызове процедур (см. рис. 1.8).



При необходимости сохранять локальную информацию от одного вызова подпрограммы к другому нельзя передавать эту информацию через локальные переменные.

Для сохранности между вызовами информация должна храниться вне подпрограммы, то есть в виде глобальной переменной (или переменных). Но в этом случае приходится отводить глобальные переменные по сути под

локальные данные. Чтобы решить эту проблему, необходимо использовать статические локальные переменные (локальные переменные со стартовыми значениями). Они вводятся точно так же, как их глобальные аналоги - типизированные константы (рис. 1.5).

Особенность переменных, объявленных таким образом, заключается в том, что хотя по методу доступа они являются строго локальными, свои значения они хранят в сегменте данных вместе с глобальными переменными. Поэтому значения переменных *R*, *Int*, *W*, *i* и *a* сохраняются неизменными до следующего вызова процедуры и после него. В них можно накапливать значения при многократном обращении процедуры, их можно использовать как флаги каких-то событий и т.п. При этом нужно помнить, что *инициализация переменных R, Int, W, i и a происходит только раз* при первом вызове процедуры (функции).

```
{===== Изменение статических локальных переменных =====}
Procedure Example ( z : Real;  Ch : Char );
Const R : Real = 25.0;           { Статические локальные переменные }
    Int : Integer = -100;
    W : Word = 5;
    i : Byte = 1;
    a : Array [1..3] of Char = ('a', 'b', 'c');
Var   j : Byte;                  { Обычный локальный параметр      }
Begin
    R := R + z;  Int := Int + W;
    Write('R=', R:7:2, ' Int=', Int:4, ' W=', W:2);
    a[i] := Ch;  WriteLn (' a[' , i, ']= ', a[i]);
    For j := 1 to 3 do Write (a[j]);
    WriteLn;
    Inc(i);
End { Example };
```

Рис. 1.5. Изменение статических локальных переменных (см. с. 20)

Begin

{===== Первый результат =====}

Example (10, 'M');

R= 3 5 . 0 0 Int= - 9 5 W= 5 a[1]= M

Mbc

{===== Второй результат =====}

Example (10, 'И');

R= 4 5 . 0 0 Int= - 9 0 W= 5 a[2]= И

Мис

{===== Третий результат =====}

Example (10, 'Р');

R= 5 5 . 0 0 Int= - 8 5 W= 5 a[3]= Р

МИР

End.

Рис. 1.5. Окончание



Область действия меток перехода всегда локальна. Метка описывается в том блоке, где используется.



Нельзя планировать переходы с помощью оператора *GoTo* из вложенной процедуры в охватывающую или в основной блок программы или из программы в процедуру (рис. 1.6).


```

Procedure P1;
  Procedure P2;
    Label L1; { Описываем метку L1 как локальную в процедуре P2 }
  Begin {P2}
    L1 : . . . { Действие, выполняемое по метке L1 }
  End {P2};
Begin {P1}
Goto L1; { Попытка передать управление по метке L1 }
{ из процедуры P1 в процедуру P2 }
О Ш И Б К А ! ! !
{ Переход по метке L1 в этом случае невозможен }
End {P1};

```

Рис. 1.6. Ошибки при использовании оператора перехода GO TO

1.4. Передаваемые параметры процедур и функций

Кроме локальных и глобальных параметров у процедур и функций выделяют формальные и фактические параметры.

Формальные параметры – это абстрактные объекты, по отношению к которым выполняется процедура или функция. Формальные параметры – это параметры, используемые при описании процедуры или функции.

Procedure <Имя_Процедуры> [(<Формальные_Параметры>)];

В отличие от них **фактические параметры** – это конкретные объекты, с которыми работает процедура или функция. Фактические параметры находятся в списке параметров при вызове.

<Имя_Процедуры> [(<Фактические_Параметры>)];

Между формальными и фактическими параметрами устанавливается взаимно однозначное соответствие.



Типы фактических параметров должны соответствовать типам формальных параметров.



Количество фактических параметров при обращении к подпрограмме должно совпадать с количеством формальных параметров.



Соответствие между фактическими и формальными параметрами устанавливается по их номерам в списке слева направо: первому слева формальному параметру ставится в соответствие первый слева фактический параметр, второму слева формальному – второй слева фактический и т. д.

Различие между формальными и фактическими параметрами не только терминологическое, но и по существу: описание процедуры (функции) задает формальную схему обработки, а *вызов процедуры (функции)* осуществляет *настройку этой формальной схемы на конкретную обработку для фактических данных*. Формальные параметры являются своего рода шаблонами, заполнение которых задают фактические параметры.

При вызове процедуры (функции) все формальные параметры заменяются соответствующими фактическими, создается динамический аналог блока процедуры (функции), который и выполняется.

Совпадение имен фактических и формальных параметров никаких неприятностей в программах не вызывает.

Хотя внешне все формальные параметры – это идентификаторы, но функционально они различаются: через одни параметры подпрограмма получает исходные данные, через другие отдает свои результаты, а некоторые параметры служат и для того и для другого одновременно.

По этому принципу параметры можно разделить на три группы:

- 1) *входные* - через которые процедура получает исходные данные;
- 2) *выходные* - через которые процедура отдает результаты;

3) *возвратные* - через которые процедура получает данные и отдает результаты.

Параметры процедур и функций можно перечислять в любом порядке, однако принято описывать сначала входные, затем возвратные, а в конце выходные.

В языке Паскаль различают формальные параметры четырех видов:

- *параметры-значения;*
- *параметры-переменные;*
- *параметры-процедуры;*
- *параметры-функции.*

1.4.1. Параметры-значения

Если в начале раздела параметров нет никакого служебного слова, то речь идет о *параметрах-значениях*. Такой формальный параметр представляется в вызываемой процедуре некоторой локальной переменной, которая размещается в автоматической («стековой») памяти и, следовательно, недоступна после выполнения процедуры основной программе. В качестве начального значения для формального параметра пересылается текущее значение соответствующего фактического параметра. Значение формального параметра может изменяться при выполнении процедуры, однако никакого влияния на значение фактического параметра это не оказывает.

При вызове процедуры (функции) на место параметра-значения могут подставляться фактические параметры в виде *констант, переменных, стандартных функций, арифметических выражений*. Рассмотрим это на примере процедуры нахождения корней квадратного уравнения, где *A, B, C* - коэффициенты квадратного уравнения, *Root1, Root2* - его корни.

```
Procedure Roots (A,B,C:Real; Var Root1,Root2:Real);
```

- константы

```
Roots ( ConstA, ConstB, ConstC, Root1, Root2 );
```

- конкретные числовые значения

```
Roots ( 15, 44, 371, Root1, Root2 );
```

- переменные

```
Roots ( A1, B1, C1, Root1, Root2 );
```

- стандартные математические или «библиотечные» функции

```
Roots ( Sqr(A), Abs(B), Sqrt(C), Root1, Root2 );
```

- арифметические выражения

```
Roots ( A+Sqr(B), B+A, 4-Sqr(C), Root1, Root2 );
```

При этом необходимо помнить о соответствии типов формальных и фактических параметров.



Параметр-значение может быть только *входным* параметром процедуры, использование его в качестве результата вычислений даст непредсказуемый ответ.

1.4.2. Параметры-переменные

Если формальный параметр является результатом работы процедуры или изменяется в результате ее работы, то перед этим формальным параметром ставится служебное слово *Var*. После этого на все время выполнения процедуры формальный параметр становится синонимом этой фактической переменной. Любые операции с таким формальным параметром выполняются непосредственно над соответствующим фактическим параметром. Значение *параметра-переменной* после завершения выполнения процедуры остается доступным для следующих вычислений, несмотря на то что динамический экземпляр блока процедуры прекращает свое существование.



Параметр-переменная используется для представления результатов работы процедуры, т.е. может быть *выходным* или *возвратным* параметром.



Var действует до ближайшей точки с запятой, то есть перед каждым выходным параметром разного типа должен быть *Var*.



В качестве фактических параметров, соответствующих формальным параметрам-переменным, запрещается использовать поле признака из записи с вариантами, чтобы не возникали зависящие от реализации проблемы вычисления адресов.

```
Type  Array_of_Int = array [1..100] of Integer;
      Array_of_Char = array [1..100] of Char;
      File_of_Byte  = File of Byte;
```

Var

```
Arr_Int : Array_of_Int;    { Массив целых чисел          }
Arr_Char : Array_of_Char;  { Массив символов        }
n : Byte;                  { Рабочая длина вектора    }
F : File_of_Byte;          { Файл целых чисел      }
```

{===== Вычисление суммы элементов массива =====}

~~Procedure SumElemArr~~

~~(n : Byte;~~
 ~~Var A:Array_of_Int;~~
 ~~Summa : Integer);~~

Семантическая ошибка:

После выполнения процедуры значение переменной *Summa* будет не определено.

Procedure SumElemArr

(n:Byte;
 Var A:Array_of_Int;
 Var Summa : Integer);

Правильное описание

Переменная *Summa* является результатом работы процедуры (выходной параметр) и должна быть описана с *Var*.

{===== Преобразование кодов в символы =====}

Procedure Translate

~~(n : Byte;
Var A:Array_of_Int;
B:Array_of_Char);~~

Семантическая ошибка:

После выполнения процедуры значение массива B будет не определено, т.к. массив описан без Var.

Procedure Translate

(n :Byte;
Var A:Array_of_Int;
Var b:Array_of_Char);

Правильное описание



Параметры, имеющие файловый тип (или сложный тип с файловыми компонентами), должны быть описаны как параметры-переменные.

{===== Вычисление суммы элементов файла =====}

~~Procedure SumElemFile~~

~~(F : File_of_Byte;
Var Summa : Word);~~

Синтаксическая ошибка:

Файловая переменная F описана без Var.

Procedure SumElemFile

(Var F : File_of_Byte;
Var Summa : Word);

Правильное описание

Разницу между параметрами-значениями и параметрами-переменными иллюстрирует пример, приведенный на рис. 1.7.

```

Var  A, B : Integer;
Procedure One (X : Integer; Var Y : Integer);
    { X – параметр-значение, Y – параметр-переменная }
Begin
    X := 1;    Y := 1;
    Writeln ('Печать результатов в процедуре');
    Writeln ('x = ', X:3, ' y = ', Y:3);
End { One };

Begin                                     { Начало основной программы }
    A := 1;    B := 1;
    One (A, B);                           { Вызов процедуры One }
    Writeln ('Печать результатов в программе');
    Writeln ('a = ', A:3, ' b = ', B:3);
End.

```

Результаты работы этой программы будут выглядеть следующим образом:

Печать результатов в процедуре

x = _ _ 1 y = _ _ 1

Печать результатов в программе

a = _ _ 1 b = _ _ 0

Рис. 1.7. Различие между параметром-значением и параметром-переменной

При активации процедуры для каждого локального параметра выделяется место в автоматической (стековой) памяти. Глобальное значение фактического

параметра копируется туда, а при выходе из процедуры эта память освобождается и значение переменной становится недоступным. В случае если передаваемый параметр занимает много памяти (например, массив), передача (копирование) его значений в стек займет дополнительное время и память стека. Для таких данных лучше описывать такой параметр как параметр-переменную, т.е. с предшествующим словом *Var*.

Рассмотрим вопрос о передаче в качестве параметров подпрограммы массивов данных (векторов и матриц). Передача параметров в этом случае имеет ряд особенностей.



В целях экономии времени и памяти при передаче массивов рекомендуется использовать в качестве параметров в заголовке процедуры *параметры-переменные*.



Рекомендуется в качестве параметра передавать рабочий размер массива.



Параметр цикла должен быть описан внутри процедуры.

Описание его глобально, а использование внутри процедуры может привести к непредсказуемым результатам (рис. 1.8).

```
{===== Запись сумм строк матрицы в вектор =====}  
Const N_Max = 10;  
Type Vector = array [1 .. N_Max] of Integer;  
      Matrix = array [1..N_Max, 1.. N_Max] of ShortInt;  
Var b : Vector;           { Вектор целых чисел }  
    a : Matrix;           { Матрица целых чисел }  
    i, j : Byte;          { Глобальные параметры цикла }  
    n : 1 .. N_Max;       { Рабочий размер матрицы и вектора }
```

Рис. 1.8. Запись сумм строк матрицы в вектор (см. с. 29)


```

Procedure SumStrMatrix      {=== Вычисление суммы ===}
                             {== k-й строки матрицы ==}
  ( n : Byte;               { Количество элементов в строке }
                                { матрицы (рабочий размер) }
    k : Byte;               { Номер строки матрицы }
    Var A: Matrix;          { Матрица. Var перед матрицей - в }
                                { целях экономии времени и памяти }
    Var Summa : Integer );  { Сумма строки (вектора) матрицы }

  Var i : Byte;             { Обязательное описание }
                                { параметра цикла в процедуре }

Begin
  Summa := 0;               { Нахождение суммы k-й строки }
  For i := 1 to n do       { (вектора) матрицы }
    Summa := Summa + a[k,i];
  End { SumStrMatrix };

Begin                      { Главный begin программы }
  Readln(n);               { Ввод рабочего размера матрицы }
  For i := 1 to n do       { Ввод значений матрицы }
    For j := 1 to n do
      Readln(a[i,j]);
    For i := 1 to n do { Если не описать параметр цикла }
                        { в процедуре, в этом месте }
                        { программы возникнет }
                        { неопределенность со значением }
                        { переменной i }
      SumStrMatrix (n, i, a, b[i]); Семантическая
                                ошибка !!!
    For i := 1 to n do { Печать элементов вектора сумм }
      Write( b[i] );
    End.

```

Рис. 1.8 Окончание



При передаче массивов через параметры процедуры после идентификатора формального параметра можно использовать только тип, заранее определенный в разделе типов.

```
Procedure SumStrMatrix
( n : Byte;
  k : Byte;
  A: Array [1..100] of Integer;
  Var Summa : Integer );
```

Синтаксическая ошибка:

Нельзя конструировать типы в описании параметров

```
Procedure SumStrMatrix
( n : Byte;
  k : Byte;
  Var A:Array_of_Int;
  Var Summa : Integer );
```

Правильное описание

В последних версиях языка Паскаль разрешается использовать массивы открытой структуры без описания его размера в параметрах процедуры (рис. 1.9).

```
{===== Сумма элементов вектора открытого размера =====}
Var
  b : Array [1..5] of Real;      { Вектор содержит 5 элементов }
  c : Array [-10..10] of Real;   { Вектор содержит 21 элемент }
  d : Array [0..7] of Real;      { Вектор содержит 8 элементов }
  Sb, Sc, Sd : Real;            { Суммы элементов векторов b, c, d }
Procedure Summa                  {=== Вычисление суммы ===}
                                {=== открытого массива ===}
  ( Var a: Array of Real;        {Вектор открытого типа}
    Var S : Real );              { Сумма вектора }
Var i : Byte;                    { Параметр цикла }
Begin
```

Рис. 1.9. Сумма элементов векторов открытого размера (см. с. 31)

```

S := 0;
For i := 0 to High(a) do      { Определение размера вектора с }
                             { помощью функции High           }
    S := S + a[i];           { Нахождение суммы вектора       }
End { Summa };
Begin                         { Главный begin программы      }
    . . .                    { ===== Ввод элементов вектора ===== }
    Summa (b, Sb);           { Индексация элементов от 0 до 4       }
    Summa (c, Sc);           { Индексация элементов от 0 до 20      }
    Summa (d, Sd);           { Индексация элементов от 0 до 7       }
    . . .                    { = Печать сумм элементов векторов = }
End.

```

Рис. 1.9. Окончание



При нерекурсивном выполнении имя функции не может появляться в правой части оператора присваивания, поэтому, если результат функции переопределяется в теле функции несколько раз, то необходимо ввести промежуточную переменную для вычислений (того же типа, что и результат функции), а результат присвоить при выходе из функции (рис. 1.10).

```

{===== Возведение числа в целую степень =====}
Type  Natural = 0 .. MaxInt; { Натуральные числа }
Var   x, PowerX : Real;      { Число и его степень }
Function Power {= функция возведения числа в степень =}
    ( Base : Real;           { Основание }
      Exponent:Natural):Real; { Степень }
Var i : Byte;               { Параметр цикла }
Begin
    Power := 1;
    For i := 1 to Exponent do Синтаксическая ошибка:
        Power := Power * Base; Рекурсивный вызов функции
    End { Power };

```

Рис. 1.10. Возведение числа в целую степень (см. с. 32)

Function Power	<u>Правильное решение</u>
(Base : Real; Exponent:Natural):Real;	
Var i : Byte;	{ Параметр цикла }
Result : Real;	{ Переменная для хранения результата }
Begin	
Result:= 1;	
For i := 1 to Exponent do	
Result := Result * Base;	
Power := Result;	{ Возвращение результата через имя }
	{ функции }
End { Power };	

Рис. 1.10. Окончание



Процедура не должна выводить внутри себя диагностику и результаты, если она сама по себе не предназначена для этого. В случае необходимости процедура должна иметь *сигнальный параметр*.

Например, в задаче, рассмотренной на рис. 1.11, необходимо найти среднее арифметическое положительных элементов вектора. В случае, если положительные элементы в векторе отсутствуют, их количество равно нулю и при вычислении среднего арифметического произойдет деление на ноль. Чтобы этой ошибки не возникло, в параметры процедуры добавляется сигнальный параметр *Flag*. Во время выполнения процедуры получение среднего арифметического будет зависеть от результата сигнального параметра.

```

{===== Среднее арифметическое =====}
{===== положительных элементов вектора =====}
Const N_Max = 10;           { Количество элементов вектора }

Type
  Vector = Array [1..N_Max] of Real; { Тип - вектор }
Var
  A : Vector;
  n : Byte;           { Рабочая длина вектора }
  Exist : Boolean;     { Сигнальный параметр }
  Average : Real;      { Среднее арифметическое }
Procedure Calculation
( n : Byte;           { Рабочая длина вектора }
  Var A : Vector;     { Вектор }
  Var Result : Real;   { Среднее арифметическое }
  Var Flag : Boolean ); { Сигнальный параметр }
Var
  S : Real;           { Сумма положительных элементов }
  Count,             { Количество положительных элементов }
  i : Byte;           { Параметр цикла }
Begin
  S := 0; Count := 0; { Инициализация суммы и количества }
  For i := 1 to n do { Цикл по всем элементам вектора }
    If A[i] > 0 then begin { Если элемент вектора > 0, то }
      S := S + A[i]; { увеличим сумму на этот элемент, }
      Inc(Count); { увеличим количество положительных }
                  { элементов на 1 }
    End { If };
  If Count = 0 then { Если количество положительных элементов }
    Flag := False { вектора равно 0, то "опускаем" флаг }
  Else begin { Иначе }
    Result := S / Count; { Вычисляем среднее арифметическое }
    Flag := True; { "Поднимаем" флаг }
  End;
End { Calculation };

```

Рис. 1.11. Использование сигнального параметра в процедуре (см. с. 34)

```

Begin                                     { Главный begin программы }
. . .                                     { = Ввод рабочей длины и значений элементов вектора = }
Calculation (n, A, Exist, Average);
If Exist then Write('Среднее = ', Average:8:4)
Else Write('Положительных чисел в векторе нет');
End.

```

Рис. 1.11. Окончание

1.4.3. Процедуры и функции как параметры

Параметрами процедур и функций могут выступать сами процедуры и функции. Точно так же, как передавалось значение, может передаваться некая функция его обработки. При программной реализации алгоритмов вычислительной математики передача через параметры самих функций оказывается весьма эффективной (например, в задаче интегрирования любой функции вида $F(x)$) (см. рис. 1.15).

Функциональный (или **процедурный**) тип — это не тип возвращаемого значения, а тип заголовка подпрограммы в целом. При описании типа заголовка название подпрограммы не ставится, так как оно не играет роли. Но обязательно перечисляются типы параметров и, если тип описывает функцию, тип результата. Идентификаторы параметров могут быть выбраны произвольно, основная смысловая нагрузка падает на их типы и порядок следования (рис. 1.12).

Фактический параметр, соответствующий формальному *параметру-процедуре* (функции), должен быть идентификатором, определение которого содержится в декларативном блоке программы. Процедура (функция), задаваемая фактическим параметром, и процедура (функция), задаваемая формальным параметром, должны иметь совпадающие списки параметров. Кроме того, функция, задаваемая фактическим параметром, и функция,

задаваемая формальным параметром, должны иметь идентичные типы результатов.

```

Type
  FuncType = Function (x : Real):Real; { Функциональный тип }
  ProcType = Procedure                { Процедурный тип.      }
              (a, b, h : Real;        { Сама процедура использу }
               f : FuncType;          { ет в качестве параметра }
               Var Res : Real);       { функциональный тип    }
  NoParamProcType = Procedure;        { Процедурный тип      }
                                      { без параметров        }
  NoParamFuncType = Function : Real;  { Функциональный тип  }
                                      { без параметров      }

```

Рис. 1.12. Процедурный и функциональный типы

После объявления функционального (процедурного) типа его можно использовать в описаниях параметров подпрограмм и, конечно, необходимо написать те реальные процедуры и функции, которые будут передаваться как параметры. Требование к ним одно: они должны компилироваться в режиме с ключом *{SF+}* или указываться с ключевым словом *Far*, что равнозначно. Поскольку по умолчанию принят режим *{SF-}*, такие процедуры обрамляются парой соответствующих директив (рис. 1.13).

```

{SF+}                                { Включение режима SF+      }
{= функция вычисления квадратного корня из экспоненты =}
Function SqrtExp (x : Real):Real;
Begin
  SqrtExp := Sqrt( Exp ( x ) );
End { SqrtExp };
{SF-}                                { Отключение режима SF-    }
                                      { Восстановление по умолчанию }

```

Рис. 1.13. Использование ключей компиляции *{SF+}*



Не всякую функцию (процедуру) можно подставить в вызов в качестве фактического параметра.



Нельзя подставлять в вызов следующие типы подпрограмм:

- процедуры с директивами *InLine* и *Interrupt* из-за особенностей их машинного представления;
- вложенные процедуры и функции;
- стандартные процедуры и функции.

Последнее ограничение легко обходится переопределением стандартной функции (рис. 1.14).

```
{===== Функция вычисления косинуса =====}
Function CosX (x : Real):Real;    Far;
Begin
    CosX := Cos (x);
End { CosX };
```

Рис. 1.14. Переопределение стандартной функции

Рассмотрим использование функционального типа на примере вычисления суммы интегралов от произвольной функции $f(x)$. Вычисление интеграла будем производить методом средних прямоугольников. Точность вычисления задается количеством участков разбиения (рис. 1.15).

```
{===== Сумма интегралов =====}
Const
    LimA = 0;                                { Пределы интегрирования }
    LimB = 2 * Pi / 3;
    LimC = 10 / 3;
    LimD = 8.5;
Type FuncType = Function (x:Real):Real;    { Функциональный }
                                              { тип }
                                              }
```

Рис. 1.15. Вычисление суммы интегралов (см. с. 37, 38)


```

Var  Int1, Int2,                               { Первый и второй интегралы, }
      Summa : Real;                             { их сумма }

{$F+}

{= функция вычисления квадратного корня из экспоненты =}

Function SqrtExp (x : Real):Real;

Begin

  SqrtExp := Sqrt( Exp ( x ));

End { SqrtExp };

{===== функция вычисления косинуса =====}

Function  CosX   (x   :   Real):Real;

Far;

Begin

  CosX := Cos (x);

End { CosX };

{$F-}

{===== Вычисление интеграла методом прямоугольников =====}

Procedure Integral

  (LowLimit, HiLimit : Real;                     { Нижний и верхний }
                                     { пределы интегрирования }
   n : Word;                                     { Количество участков }
                                     { разбиения }
   F : FuncType;   { Функциональный тип }
   Var Result : Real); { Результат — интеграл }

Var  x,                                     { аргумент функции }
      Step : Real;                         { шаг вычисления }

Begin

  Result := 0;

  Step := (HiLimit - LowLimit) / n;

```

Рис. 1.15. Продолжение

```
X:= LowLimit + 0.5 * Step;
While x < HiLimit do begin
    Result := Result + F(x);
    x := x + Step;
End;
Result := Result * Step;
End { Integral };
Begin                                     { Главный begin программы }
{ Вызов процедуры Integral с сохранением результата в Int1 }
{ Количество участков разбиения равно 200 }
Integral (LimA, LimB, 200, CosX, Int1);
{ Вызов процедуры Integral с сохранением результата в Int2 }
{ Количество участков разбиения равно 500 }
Integral (LimC, LimD, 500, SqrtExp, Int2);
Summa := Int1 + Int2;
Write ('Сумма интегралов = ', Summa:8:4);
End.
```

Рис. 1.15. Окончание

Применение процедурного типа не ограничивается только описанием параметров-процедур или функций, могут быть описаны и переменные этого типа. Это особенно удобно при организации меню, когда необходимо сделать выбор того или иного действия (выбор подпрограммы) (рис. 1.16) или когда переменная-подпрограмма является полем какой-либо сложной структуры (например, записи).

```
Var
  F : FuncType;      { Переменная функционального типа }
  Number : Byte;      { Номер используемой функции }
  Result : Real;      { Результат: вычисленный интеграл }
{===== Описание всех используемых функций =====}
. . .
{===== Описание процедуры вычисления интеграла =====}
. . .
Begin                { Главный begin программы }
  Repeat
    Write ('Введите номер функции ');
    ReadLn (Number);
  Until ( Number < 5 ) And (Number > 0);
  Case Number of
    1 : F := CosX;    { Косинус }
    2 : F := ExpX;    { Экспонента }
    3 : F := SinCosX; { Произведение синуса и косинуса }
    4 : F := SinX;    { Синус }
  End {Case};
  {- Вызов процедуры Integral с выбранной и сохраненной -}
  {- в переменной F функцией и сохранением результата -}
  {----- вычисления интеграла в переменной Result -----}
  Integral (LimA, LimB, 100, F, Result);
  Write('Интеграл = ', Result:10:3);
End.
```

Рис. 1.16. Вычисление интеграла от заданной функции



Попытка использовать переменную-подпрограмму, не присвоив ей значение, может привести к «зависанию» программы, эта ошибка не диагностируется и может привести к непредсказуемым последствиям.

Используя сложные структуры, можно хранить в них не только данные, но и процедуры (функции) их обработки. Причем в любой момент можно изменить процедуру или функцию, понимаемую под полем *Action* (рис. 1.17.).

```
Type
  ProcType = Procedure (<список параметров>);
  RecType = Record
    Param1, Param2 : Word;
    Action : ProcType;
  End; { RecType }
Var
  Rec1, Rec2 : RecType;
```

Рис. 1.17. Использование процедурного типа в качестве поля записи



С целью повышения надежности программ в качестве параметров процедур типа *ProcType* (см. рис. 1.17) лучше использовать параметры-значения, несмотря на то что это несколько снижает гибкость использования процедур.

1.4.4. Бестиповые параметры процедур и функций

При необходимости создать процедуру, фактическим параметром которой при нескольких ее вызовах могут быть переменные различных типов, можно использовать в качестве формального параметра бестиповой параметр. В этом

случае в заголовке процедуры или функции тип этого параметра не указывается.

```
Procedure ProcDemo ( Var X, Y );
```

```
Function FuncDemo ( A : Real; Var B ):Real;
```

Бестиповыми могут быть *только параметры-переменные*, так как передается не значение, а адрес. Через бестиповые переменные X, Y, B (см. выше) можно передавать различные конструкции: простые переменные, массивы, записи, строки и т.д. Но при этом подпрограмма должна явно задавать тип, к которому внутри нее приравниваются бестиповые параметры-переменные. На рис. 1.18 приведен пример использования таких параметров.

```
{===== Нахождение суммы элементов вектора =====}
{===== с использованием бестиповых переменных =====}
Var
  A1 : Array [1..50] of Byte;           { 50 элементов      }
  A2 : Array [-10..10] of Byte;         { 21 элемент        }
  A3 : Array ['a'..'z'] of Byte;        { 26 элементов      }
  St : String;                          { Строка            }
  S1, S2, S3,                           {Суммы векторов     }
  S4      : Word;                       {Сумма кодов строки }
{$R-}                                   { Отключение режима проверки индексов }
Procedure Summa (Var A; {Бестиповая переменная-вектор}
                 n : Byte; {Количество элементов вектора}
                 Var S : Word); {Сумма элементов вектора }
Type
  Vector = Array [-1..1] of Byte; { Описание типа вектора }
                                     { произвольного размера }
                                     { с типом элементов. }
                                     { идентичным глобальному }
```

Рис. 1.18. Использование бестиповых параметров-переменных (см. с. 42)

```

Var i : Byte;
                                                    { Параметр цикла }
Begin
  S := 0;
  For i := 1 to n do
                                                    { Нахождение суммы }
    S := S + Vector ( A ) [i];
End { Summa };
{$R+}
                                                    { Включение режима проверки индексов }
Begin {Главный}
  {-- Ввод рабочей длины и значений элементов векторов --}

  St := 'I love Pascal!';

  Summa (A1, 50, S1);
                                                    { Вычисление суммы S1 всех (50) }
                                                    { элементов вектора A1 }

  Summa (A2[0], 10, S2);
                                                    { Вычисление суммы S2 10 элементов }
                                                    { вектора A2, начиная с номера 0 }

  Summa (A3['d'], 15, S3);
                                                    { Вычисление суммы S3 15 элементов }
                                                    { вектора A3, начиная с номера 'd' }

  Summa (St[1], 14, S4);
                                                    { Вычисление суммы кодов символов }
                                                    { строки St, начиная с номера 1 по 14 }

  {----- Печать результатов вычислений -----}

End.

```

Рис. 1.18. Окончание

Переменная *A* в процедуре *Summa* (см. рис. 1.18) не боится несовместимости типов, но корректно процедура будет работать только с массивами, элементы которых будут иметь тип *Byte* или явно приводимый к нему (как, например, коды символов). Переопределение типов данных производится с помощью специально введенного типа *Vector*, к нему приводится тип любой переменной, передаваемой через параметр-переменную

А. Нумерация индексов в типе *Vector* может быть любой, значения $-1..1$ были заданы чисто в иллюстративных целях. При отключенном режиме проверки выхода индекса за границу массива $\{SR-\}$ можно обращаться к массиву с любым индексом. Передавая через переменную *A* строку *St*, мы получаем доступ к этой строке, но не к символам, а к их кодам (к байтам).

Рассмотренный пример можно распространить и на записи, и на ссылки, и на числовые переменные, но не рекомендуется передавать таким образом переменные множественного или перечислимого типа, так как они имеют свои особенности в машинном представлении.

1.5. Процедура или функция?

Как уже отмечалось, между процедурами и функциями много общего как с синтаксической, так и с семантической точек зрения.

```
{===== Вычислить k! / (2k)! + 5! =====}
Var
  Result : Real;           {Результат}
  K : Byte;                {Число для вычисления факториала}
  F1, F2, F3 : LongInt;    {Переменные для хранения значений}
                           {факториалов}
{===== Вычисление факториала с помощью функции =====}
Function Fact_Fun(n:Byte):LongInt;
Var i : Byte; F : LongInt;
Begin
  F := 1;
  For i := 2 to n do
    F := F * i;
  Fact_Fun := F;
End { Fact_Fun };
```

Рис. 1.19. Вычисление факториалов (см. с. 44)

```

{===== Вычисление факториала с помощью процедуры =====}
Procedure Fact_Proc(n:Byte; Var F:LongInt);
Var i : Byte;
Begin
    F := 1;
    For i := 2 to n do
        F := F * i ;
End { Fact_Proc };
Begin { Главный }
    Write ('Введите число');
    ReadLn (k);
    F1 := Fact_Fun (k);           { Вычисление факториалов }
    F2 := Fact_Fun (2*k);        { с помощью функции }
    F3 := Fact_Fun (5);
    Result := F1 / F2 + F3;
    Write ('Результат = ', Result:12:7);
    Fact_Proc (k, F1);           { Вычисление факториалов }
    Fact_Proc (2*k, F2);        { с помощью процедуры }
    Fact_Proc (5, F3);
    Result := F1 / F2 + F3;
    Write ('Результат = ', Result:12:7);
End.

```

Рис. 1.19. Окончание

На самом деле во многих случаях можно почти с одинаковым успехом использовать как процедуры, так и функции, если результатом является (единственное) скалярное значение. В этой связи возникает вопрос, какой же вид подпрограммы следует предпочесть: процедуру или функцию? Ответ на

него зависит от того, как мы дальше собираемся использовать подпрограмму, главное при этом не забывать о побочном эффекте функции. Если вычисляемое значение чаще используется в качестве промежуточных значений при вычислениях, то лучше использовать функцию, если же это значение необходимо присвоить некоторой переменной, то можно использовать и процедуру (рис. 1.19).

1.6. Опережающее описание процедур и функций

Текст программы транслируется в выполнимый код последовательно сверху вниз. При этом переменные, типы, константы и подпрограммы должны описываться до того, как начнутся их упоминания в операторах и выражениях. В противном случае компилятор объявит имя неизвестным и придется перемещать описания подпрограмм вверх по тексту программы. В случаях с процедурами и функциями этого можно избежать, используя опережающее описание директивой *Forward*.

```
Procedure <Имя_Процедуры> (<Список параметров>);  
                                                    Forward;  
  
Function <Имя_Функции> (<Список параметров>)  
                                                    : ТипРезультата;  
                                                    Forward;  
  
Procedure <Имя_Процедуры>; { Список параметров не нужен }  
< Тело процедуры >  
Function <Имя_Функции>; { Достаточно указать только имя }  
< Тело функции >
```

Директива *Forward* объявляет заголовок подпрограммы, откладывая описание содержимого «на потом». Месторасположение этого описания уже не играет роли, и в нем можно не указывать параметры, а ограничиться лишь именем подпрограммы. Основное описание не может иметь никаких директив (*Forward*, *External* и др.). Директива *Forward* существует в языке в основном для развязки закольцованных вызовов (рис. 1.20).

```
Procedure First (Parameter : TypeParam); Forward;  
Procedure Second (Parameter : TypeParam); Forward;  
Begin  
    First (Parameter); { Процедура Second вызывает процедуру First }  
End { Second };  
Procedure First;  
Begin  
    Second (Parameter); { Но сама процедура First вызывает Second }  
End { First };
```

Рис. 1.20. Развязка закольцованных вызовов процедур

1.7. Внешние процедуры и функции

Паскаль - язык не слишком коммуникабельный по отношению к другим языкам программирования. Он не поддерживает генерацию объектных файлов в формате Obj и вследствие этого не может поставлять написанные на нем процедуры и функции другим программам, написанным на другом языке. Единственное, что можно - это использовать при компиляции и компоновке программ внешние подпрограммы в виде Obj-файлов, созданных другими компиляторами. Obj-файлы должны при этом удовлетворять определенным требованиям к используемой модели памяти и способу передачи значений. Гарантируется совместимость кодов, полученных компилятором Turbo Assembler. Возможен импорт объектных кодов, полученных в Turbo C, но на практике он трудно реализуем.

Команды подстыковки объектных файлов в программу, написанную на языке Паскаль, задаются директивой компилятора {\$L ИмяФайла,Obj}. Процедуры и функции, реализованные в файлах, указанных в директиве,

должны быть объявлены своими именами и специальным словом `External` (рис. 1.21).

```
{ $L ExtFile.Obj } { Подключение внешнего файла ExtFile.Obj }  
Procedure Proc1; External;  
Procedure Proc2 (a : Byte; Var b : Real); External;  
Function Func1 (c : Integer) : Real; External;
```

Рис. 1.21. Компоновка файла объектных кодов

Подключенные таким образом процедуры ничем не отличаются от других, написанных в тексте самой программы. Обычно директиву подключения `Obj`-файлов и объявления подключенных подпрограмм удобно размещать рядом, при этом порядок описаний заголовков может быть произвольным.

```
{ $I func.pas } { Компилятор поставит на это место текст функции }  
. . .  
Begin  
. . .  
F := Func (x);  
. . .  
End.
```

Рис. 1.22. Подключение файла для компиляции

Язык Паскаль допускает открытие в любой данный момент времени не более восьми входных файлов. Включаемый файл должен содержать завершенный блок предложений на Паскале.

Компилятор языка Паскаль дает возможность подключать процедуры и функции, написанные на исходном языке, но расположенные во внешних файлах. Их подключение осуществляется директивой `{ $I ИмяФайла.Pas }` в произвольном месте (обычно после раздела процедур и функций), после чего процедуры и функции могут непосредственно использоваться (рис. 1.22).

1.8. Рекурсивные процедуры и функции

Использование рекурсии является преимуществом языка Паскаль. Объект называется *рекурсивным*, если он содержит сам себя или определен с помощью самого себя.

Рекурсия является особенно мощным средством в математических определениях. Известны примеры рекурсивных определений натуральных чисел, древовидных структур и некоторых функций.

Пример №1. Натуральные числа:

- 1 есть натуральное число;
- целое число, следующее за натуральным, есть натуральное число.

Пример №2. Функция вычисления факториала $n!$ для неотрицательных целых чисел $n = 0, 1, 2, 3, 4, \dots$:

- $0! = 1$,
- если $n > 0$, то $n! = n * (n - 1)!$

Очевидно, что мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания, поэтому бесконечные вычисления можно описать с помощью конечной рекурсивной программы, даже если эта программа не содержит явных циклов:

$$P \equiv \Pi [S_i, P].$$

Однако лучше всего использовать рекурсивные алгоритмы в тех случаях, когда решаемая задача, или вычисляемая функция, или обрабатываемая структура данных определены с помощью рекурсии. В общем виде рекурсивную процедуру P можно изобразить как композицию Π базовых операторов S_i (не содержащих P) и самой P .

Необходимое и достаточное средство для рекурсивного представления программ - это само описание процедур (функций), так как оно позволяет присваивать какому-либо оператору имя, с помощью которого можно вызывать

этот оператор. Если процедура P содержит явное обращение к самой себе, то она называется *прямо рекурсивной*; если P содержит обращение к процедуре Q , которая содержит (прямо или косвенно) обращение к P , то P называется *косвенно рекурсивной*. Поэтому использование рекурсии не всегда сразу видно из текста программы.

1.8.1. Когда не нужно использовать рекурсию

Рекурсивные алгоритмы наиболее пригодны в случаях, когда поставленная задача или используемые данные определены рекурсивно. Но это не значит, что при наличии таких рекурсивных определений лучшим способом решения задачи непременно является рекурсивный алгоритм. Следует избегать использования рекурсии в тех случаях, когда вычисляемые значения определяются с помощью простых рекуррентных соотношений. Рассмотрим широко известный пример вычислений факториалов $f = n!$:

n =	0	1	2	3	4	5	...
Fn =	1	1	1*2=2	1*2*3=6	1*2*3*4=24	1*2*3*4*5=120	...

«Нулевое» число определяется явным образом как $f_0 = 1$, а последующие числа обычно определяются рекурсивно - с помощью предшествующего значения:

$$f_{i+1} = (i+1) * f_i.$$

Эта формула предполагает использование рекурсивного алгоритма для вычисления n -го факториального числа (рис. 1.23).

```
Function Factorial (n : Byte) : LongInt;
Begin
  If n > 0 then Factorial := n * Factorial (n-1)
    else Factorial := 1;
End { Factorial };
```

Рис. 1.23. Рекурсивная функция вычисления факториала

Совершенно ясно, что здесь рекурсию можно заменить обычной итерацией, а именно следующей подпрограммой (рис. 1.24):

```
Function Factorial (n : Byte):LongInt;  
  Var F : LongInt; i : Byte;  
Begin  
  F:=1;  
  For i := 2 to n do  
    F := F * i;  
  Factorial := F;  
End { Factorial };
```

Рис. 1.24. Итеративное вычисление факториала

Рассмотрим более подробно вычисление $n!$ через рекурсию. Если в функцию передается число $n > 0$, то запоминается известное значение выражения в ветви *then*, а для вычисления неизвестных вызывается та же функция, но с «предшествующим» аргументом. При этом вновь запоминаются (но в другом месте памяти!) известные значения членов и происходят новые вызовы. Так происходит до тех пор, пока выражение не станет полностью определенным (в примере - это присваивание в ветви *else*), после чего алгоритм начинает «раскручиваться» в обратном направлении, изымая из памяти «отложенные» значения. Поскольку на каждом очередном шаге все члены выражений уже будут известны, то через n таких «обратных» шагов будет получено конечное значение.

Есть и другие, более сложные рекурсивные схемы, которые можно и должно переводить в итеративную форму. Примером служит вычисление чисел Фибоначчи, определяемых с помощью рекуррентного соотношения

$$\text{Fib}_n = \text{Fib}_{n-1} + \text{Fib}_{n-2} \quad \text{для } n > 1$$

и $\text{Fib}_1 = 1$, $\text{Fib}_0 = 0$. Ряд чисел Фибоначчи выглядит следующим образом: 0 1 1 2 3 5 8 13 21 34 и т.д.

n =	0	1	2	3	4	5	6	...
Число Фибоначчи	0	1	1	1+1 = 2	1+2 = 3	2+3 = 5	3+5 = 8	...

При непосредственном, «лобовом» подходе мы получим программу, представленную на рис. 1.25.

```

Function Fib (n : Byte):LongInt;
Begin
  If n = 0 then Fib:=0
  Else If n = 1 then Fib := 1
        else Fib := Fib (n-1) + Fib (n-2);
End { Fib };

```

Рис. 1.25. Вычисление чисел Фибоначчи

При вычислении Fib_n обращение к функции $Fib(n)$ приводит к рекурсивным активациям (вызовам) этой процедуры. Сколько раз? Каждое обращение к функции при $n > 1$ приводит к двум дальнейшим обращениям, т.е. общее число обращений растёт экспоненциально (рис. 1.26). Для нахождения пятого числа из ряда Фибоначчи необходимо произвести 15 вызовов функции $Fib(n)$. При больших числах такая программа непригодна для практического использования.

Числа Фибоначчи можно вычислять по итеративной схеме, при которой использование вспомогательных переменных **Fib** (Fib_n), **Fib1** (Fib_{n-1}) и **Fib2** (Fib_{n-2}) позволяет избежать повторного вычисления одних и тех же значений (рис. 1.27).

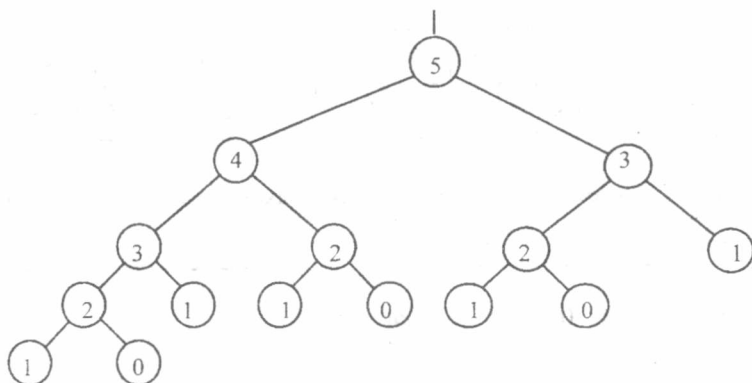


Рис. 1.26. Рекурсивные обращения к процедуре вычисления чисел Фибоначчи (для $n = 5$)

```

Fib := 1; Fib1 := 1; Fib2 := 0;
For i := 2 to n do begin
    Fib := Fib1 + Fib2;
    Fib2 := Fib1;
    Fib1 := Fib;
End;
Write ( n, '-е число Фибоначчи = ', Fib );

```

Рис. 1.27. Итеративное вычисление чисел Фибоначчи (1 вариант)

Эту программу можно упростить, убрав переменную *Fib2* (рис. 1.28).

```

Fib := 1; Fib1 := 0;
For i := 2 to n do begin
    Fib := Fib + Fib1;
    Fib1 := Fib - Fib1;
End;
Write ( n, '-е число Фибоначчи = ', Fib );

```

Рис. 1.28. Итеративное вычисление чисел Фибоначчи (2 вариант)

Из приведенных примеров видно, что *следует избегать рекурсии*, когда имеется *очевидное* итеративное решение поставленной задачи.

Но это не означает, что всегда нужно избавляться от рекурсии любой ценой, во многих случаях она вполне применима. Тот факт, что рекурсивные процедуры можно реализовать на нерекурсивных по сути компьютерах, говорит о том, что для практических целей любую рекурсивную программу можно преобразовать в чисто итеративную.

В некоторых алгоритмах использование рекурсии в программе придает изящность, но всегда при этом расходуется больше памяти, так как каждый «отложенный» вызов функции или процедуры - это дополнительный набор значений всех локальных переменных этой функции, размещенный в стеке. Если, например, будет 100 рекурсивных вызовов функции, то в памяти должно разместиться 100 наборов локальных параметров этой функции. Локальные параметры процедур и функций располагаются в стеке, в Паскале размер стека не может превышать 64К, а это не так уж много.

Необходимым условием для работы рекурсивных процедур является наличие условия окончания рекурсивных вызовов (например, проверка значения изменяющегося параметра). Действия, связанные с такой проверкой, уже не должны содержать рекурсивных вызовов. Если это условие не будет выполняться, то *глубина рекурсии* станет бесконечной, что неизбежно приведет к аварийному завершению программы. На рис. 1.25 прекращение рекурсивного вызова функции вычисления факториала произойдет по ветке *else*.

Программы, в которых рекурсия полностью оправдана, - это задачи из области «искусственного интеллекта» (алгоритмы с возвратом, переборные алгоритмы). Здесь нужно строить алгоритмы, которые находят решение определенной задачи не по фиксированным правилам вычисления, а методом «проб и ошибок». Обычно этот процесс разделяется на отдельные подзадачи. Часто эти подзадачи наиболее естественно описываются с помощью рекурсии. Процесс проб и ошибок можно рассматривать в общем виде как поисковый

процесс, который постепенно строит и просматривает (а также обрезает) дерево подзадач. Во многих случаях такие деревья поиска растут очень быстро, обычно экспоненциально, в зависимости от заданного параметра. Соответственно увеличивается стоимость поиска. Часто дерево поиска можно обрезать, используя только эвристические соображения, и тем самым сводить количество вычислений к разумным пределам. К наиболее известным чисто рекурсивным задачам относятся *задача о ходе коня* (совершить обход всех полей шахматного поля из заданной точки, причем каждое поле посещается ровно один раз), *задача о восьми ферзях* (расставить на шахматной доске восемь ферзей так, чтобы они не били друг друга) и некоторые другие.

2. РАСПРЕДЕЛЕНИЕ ПАМЯТИ ПРИ ВЫПОЛНЕНИИ ПРОГРАММ

При написании программ всегда необходимо предварительно рассчитывать требования к памяти конкретной программы. По умолчанию считается, что программа может работать со всей свободной памятью с момента запуска программы.

В реальном режиме распределению подлежит первый мегабайт ОЗУ. В этом режиме существует традиционная схема адресации, при которой размеры массивов и других данных главной части не могут превышать 64 Кбайт. В защищенном режиме адресации применяется другая система адресации и доступно все ОЗУ.

Существуют три способа распределения памяти:

1. **Статический:** память выделяется для глобальных переменных и типизированных констант один раз и после этого не перераспределяется.
2. **Автоматический:** используется так называемая стековая память, идет перераспределение в соответствии с блочной структурой программы:

при входе в процедуру память выделяется, при выходе — освобождается.

3. **Динамический:** память выделяется с помощью процедур *New* и *GetMem* и освобождается по запросу программы с помощью *Dispose* и *FreeMem*.

Общая структура памяти представлена на рис. 2.1.

Адресуемое пространство памяти в операционной системе *MS DOS* организовано **сегментами**: последовательными блоками по 64Кбайт каждый. Если известен сегмент, то дальнейшее уточнение места объекта происходит по его смещению, то есть по номеру байта от начала сегмента. Таким образом, любая ячейка памяти адресуемого пространства характеризуется парой чисел СЕГМЕНТ : СМЕЩЕНИЕ (например, видеопамять в цветном режиме располагается в памяти с адреса \$B800:0000). При этом сегмент может начинаться с любого физического адреса, что порождает множество способов адресации ячейки памяти.

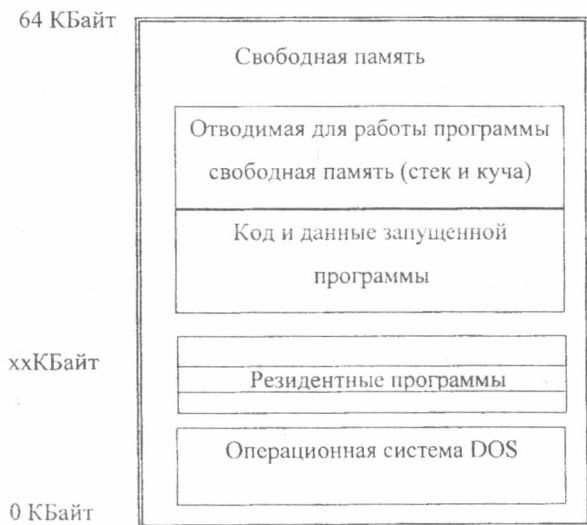


Рис. 2.1. Общая структура памяти

Рассмотрим распределение памяти для выполняемого кода программ. При запуске программы (Ехе-файла) операционная система *MS DOS* организует в памяти нечто вроде анкеты длиной в 256 байт на этот файл, называемой *PSP* (*Program Segment Prefix*).

Внешняя граница памяти MS DOS

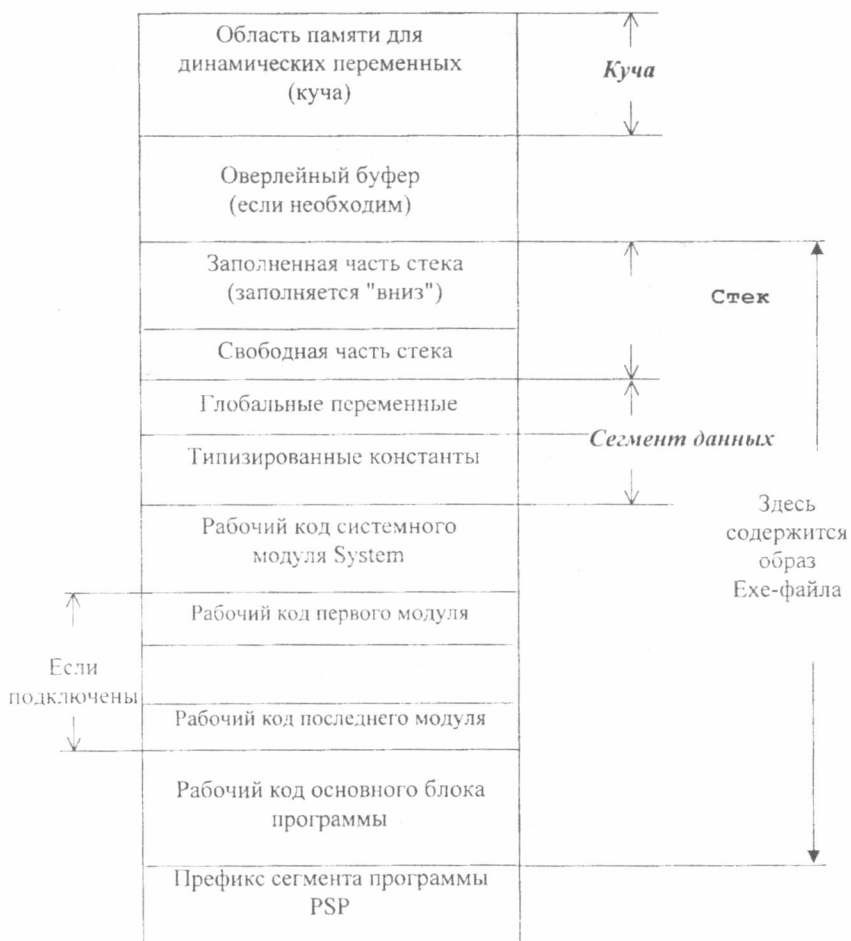


Рис. 2.2. Распределение памяти для выполняемого кода программ

После PSP начинается код Ехе-файла. Он может содержать более одного сегмента, когда выполняется какой-либо из блоков кода (основной блок, процедура из модуля и т.д.), он считается размещенным в некотором сегменте кода (реальное значение кода хранится в регистре SC процессора).

Статические глобальные переменные основного блока и все типизированные константы, включая локальные, располагаются в сегменте данных, который запоминается регистром DS процессора.

Следом за сегментом данных следует область *стека*. В нем размещаются локальные переменные и параметры-значения процедур и функций во время их работы по вызову. Сегмент стека содержится в регистре SS процессора. Стек заполняется от верхней границы по направлению к началу, то есть старту сегмента. Специальный регистр SP процессора содержит смещение указателя стека в сегменте SS. Размер стека и его верхняя граница могут быть назначены директивой компилятора \$M, которая должна быть в первых строках основной программы (в модулях она игнорируется).

{ \$M Стек, МинимумКучи, МаксимумКучи }



Общий объем переменных и типизированных констант в сумме не может превышать 64К.



Паскаль отводит под стек только один сегмент, поэтому размер стека не может превышать 64К.

Диапазон значений области стека - от 1024 до 65535 байт (от 1К до 64К). Выше стека программа отводит себе место под буфер работы оверлеев (перекрывающихся частей программы), если они не используются, то буфер не выделяется. Еще выше располагается область памяти для размещения динамических переменных и структур данных, называемая *кучей*.

Диапазон значений обоих параметров кучи - от 0 до 655360 (0К . . 640К). Максимальное значение кучи может быть больше, чем объем реально

свободной памяти, в этом случае кучей будет использоваться вся свободная память, но всегда должно выполняться условие:

МинимумКучи <= МаксимумКучи.

Если директива компилятора \$M не установлена, то значения минимального (нижнего) и максимального (верхнего) значений кучи устанавливаются соответственно 0 и 655360, что означает, что под кучу выделяется все свободное пространство памяти. Реальная необходимость для установления размеров стека и кучи возникает только при запуске субпроцессов и резидентных программ.

2.1. Распределение памяти при работе подпрограмм

Рассмотрим внутренний механизм передачи параметров подпрограмм. При вызове процедуры или функции каждой локальной переменной, описанной в подпрограмме, и каждому параметру-значению отводится место для хранения данных в специальной области памяти, называемой стеком. Эти места принадлежат переменным ровно столько времени, сколько времени выполняется подпрограмма, причем ячейки, соответствующие параметрам-значениям, сразу заполняются конкретным содержимым, задаваемым при вызове подпрограммы. Внутренние локальные переменные *принимают случайные значения (не определены по значению)*, поэтому присваивание значений таким переменным должно выполняться явно внутри процедуры.

По-другому передаются параметры-переменные. Вместо копии программа «получает разрешение» работать с тем местом, где постоянно (то есть со времени работы всего программного блока) хранится значение переменной, указанной в вызове на месте параметра-переменной. Все действия с параметром-переменной в подпрограмме, по сути, являются действиями над поставленной в вызов переменной (то есть параметры-переменные являются, по существу, глобальными параметрами). В этом заключается причина того, что при вызове на место параметра-значения можно подставить и непосредственные значения, и константы, и переменные, а на место

параметров-переменных можно подставлять только идентификатор переменной.

Рассмотрим пример, в котором функционируют как локальные, так и глобальные переменные (рис. 2.3).

Var <i>x, y, z</i> : Real;	{ Глобальные параметры }
Procedure Example (<i>x, y</i> : Real;	{ Локальные параметры }
Var Result : Real);	{ Глобальный параметр }
Var <i>z</i> : Real; <i>Temp</i> : Integer ;	{ Локальные параметры }
Begin	шаг 4.1
<i>x</i> := 44; <i>y</i> := -11;	шаг 4.2
WriteLn ('Значение <i>x</i> = ', <i>x</i> :8:4);	шаг 4.3
WriteLn ('Значение <i>y</i> = ', <i>y</i> :8:4);	
<i>z</i> := <i>x</i> + <i>y</i> ;	шаг 4.4
WriteLn ('Значение <i>z</i> = ', <i>z</i> :8:4);	шаг 4.5
WriteLn ('Temp = ', Temp);	шаг 4.6
Result := <i>z</i> ;	шаг 4.7
End ;	шаг 4.8
Begin	шаг 1
<i>x</i> := 25; <i>y</i> := - 60;	шаг 2
WriteLn ('Значение <i>x</i> = ', <i>x</i> :8:4);	шаг 3
WriteLn ('Значение <i>y</i> = ', <i>y</i> :8:4);	
Example (<i>x, y, z</i>);	шаг 4
WriteLn ('Значение <i>x</i> = ', <i>x</i> :8:4);	шаг 5
WriteLn ('Значение <i>y</i> = ', <i>y</i> :8:4);	
WriteLn ('Значение <i>z</i> = ', <i>z</i> :8:4);	шаг 6
End .	шаг 7

Рис. 2.3. Распределение памяти между локальными и глобальными переменными

Рассмотрим работу программы в пошаговом режиме (рис. 2.4).

Шаг	Действие	Значение локальных переменных	Значение глобальных переменных
1	Выделение статической памяти	Стековая память недоступна	$x=?$ $y=?$ $z=?$
2	Задание значений глобальных переменных x, y	Стековая память недоступна	$x=25$ $y=-60$ $z=?$
3	Печать значений глобальных переменных x, y		$x=25$ $y=-60$ $z=?$
4	Вызов процедуры <i>Example</i> (x, y, z);	Выделение стековой памяти	$x=25$ $y=-60$ $Z \rightarrow \text{Result}=?$
4.1	Выделение стековой памяти	$x=?$ $y=?$ $z=?$ $\text{Temp}=?$	Глобальные переменные x, y экранированы локальными x, y $\text{Result}=?$
4.2	Задание значений локальных переменных x, y	$x=44$ $y=-11$ $z=?$ $\text{Temp}=?$	
4.3	Печать значений локальных переменных x, y	$x=44$ $y=-11$ $z=?$ $\text{Temp}=?$	
4.4	Вычисление значения локальной переменной z	$x=44$ $y=-11$ $z=33$ $\text{Temp}=?$	
4.5	Печать значения локальной переменной z	$x=44$ $y=-11$ $z=33$ $\text{Temp}=?$	
4.6	Печать значения локальной переменной Temp (оно не определено!)	$x=44$ $y=-11$ $z=33$ $\text{Temp}=?$	
4.7	Вычисление значения глобальной переменной Result	$x=44$ $y=-11$ $z=33$ $\text{Temp}=?$	Переменные x, y экранированы $\text{Result}=33$
4.8	Освобождение стековой памяти	Стековая память недоступна	$x=25$ $y=-60$ $\text{Result} \rightarrow z=33$
5	Печать значений глобальных переменных x, y		$x=25$ $y=-60$ $z=33$
6	Печать значения глобальной переменной z		$x=25$ $y=-60$ $z=33$
7	Освобождение статической памяти		Статическая память недоступна

Рис. 2.4. Работа программы в пошаговом режиме

3. МОДУЛИ

3.1. Структура модулей

Модуль (*Unit*) - это специальным образом оформленная библиотека типов, констант, переменных, а также процедур и функций. Модуль, в отличие от программы, не может быть запущен самостоятельно: он может лишь использоваться в построении другой программы или другого модуля. В отличие от фрагментов, подключаемых к программе при компиляции директивой `{ $I ИмяФайла }`, модули предварительно компилируются независимо от использующей их программы. Результатом компиляции является файл с расширением *.Три*. Для того чтобы подключить модуль, необходимо и достаточно указать его имя директиве *Uses*.

Все системные библиотеки в Паскале реализованы как модули и для того, чтобы ими воспользоваться, необходимо перечислить их имена в директиве *Uses*:

```
Uses    CRT,           { модуль управления экраном в текстовом режиме }
        Graph,        { модуль управления графическими режимами }
        DOS,          { модуль для работы с дисковой операционной системой }
        My_Unit;      { модуль пользователя }
```

дальше можно использовать любые процедуры и функции, определенные внутри них. Общая структура модуля представлена ниже.

Unit <ИмяМодуля>;

Interface

```
Uses
    < Модули_Используемые_При_Объявлении >
Const
    < Библиотечные_Константы >
Type
    < Библиотечные_Типы >
Var
    < Библиотечные_Переменные >
Procedure
    < Заголовки_Библиотечных_Процедур >
Function
    < Заголовки_Библиотечных_Функций >
```

Раздел
объявлений

Implementation

Uses

< Модули_Используемые_При_Реализации >

Const

< Внутренние_Константы >

Type

< Внутренние_Типы >

Var

< Внутренние_Переменные >

<Тексты библиотечных процедур и функций >

Раздел

реализаций

Begin

Раздел

инициализации
модуляEnd

Модуль разделяется на четыре части:

заголовок модуля (*Unit*);

раздел объявлений или интерфейс (*Interface*);

раздел реализации (*Implementation*);

раздел инициализации (между *Begin* и *End*).

Все блоки, составляющие эти разделы, являются необязательными и могут как отсутствовать, так и появляться неоднократно. Обязательные слова в тексте модуля подчеркнуты. Если раздел инициализации отсутствует, то начинающее его слово *Begin* не ставится.

По имени, указанному в заголовке модуля, можно будет связать данный модуль с другими программами и модулями.



Имя модуля должно быть уникальным



Имя модуля должно соответствовать имени файла, в котором находится исходный текст модуля

Раздел объявлений, начинающийся со слова *Interface*, содержит описание типов, констант и переменных, которые будут привноситься в программу при подключении модуля. В нем также описываются заголовки подпрограмм, которые реализованы внутри модуля и доступны для внешнего использования при подключении модуля к другим программам или модулям. Тексты самих подпрограмм приводятся в разделе реализации *Implementation*, в разделе реализации список параметров можно опускать, достаточно только указать имя подпрограммы.

Раздел инициализации предназначен для присвоения каких-либо стартовых значений для используемых переменных или для выполнения каких-то одноразовых действий, которые должны производиться внутри программы.

Порядок подключения модулей может оказать существенное влияние на работу программы, например, если модули вводят одинаковые идентификаторы для определения разных понятий (библиотечные типы, данные и даже подпрограммы могут стать недоступными при совпадении имен в разных модулях). Этого можно избежать, если перед использованием идентификатора уточнять его местонахождение, указывая имя модуля. Так, если нужны переменные *x*, *y*, *z* из модуля *Unit1*, то обратиться к ним можно по уточненному имени: *Unit1.x*, *Unit1.y*, *Unit1.z*

4. ВВОД-ВЫВОД И ФАЙЛОВАЯ СИСТЕМА

Давая широкое определение термину «файл», можно сказать, что это организованное множество документов. Документами могут быть как карточки в библиотечном каталоге, так и экзаменационная или бухгалтерская ведомость, банковские счета, анкеты и т.п. Структура файлов также может быть различной и зависит от того, что хранится в файле и с какой целью он создается. Говоря о файлах, будем говорить об организации файлов, отвлекаясь от того, на каких носителях эта информация находится.

Любая программа предусматривает ввод, обработку и вывод информации. В случае обмена данными между программой и внешними устройствами одним

концом канала между входными и выходными данными является оперативная память компьютера, другим можно считать файл.

Понятие файла достаточно широко. Файлом можно назвать набор данных, хранящихся на диске, порт коммутации, принтер. Файл может быть источником информации, тогда мы говорим о чтении из файла, или приемником - в этом случае мы записываем в файл.

Важность такой структуры хранения данных как файл объясняется тремя причинами:

- любая программа может связываться с внешними устройствами только с помощью файлов;
- процесс обработки данных обычно длится недолго: программа загружается в оперативную память, выполняется, а по окончании работы память освобождается и поступает в распоряжение другой программы. При этом, если программа не изменяла данных, хранящихся на диске, эффект от ее работы будет таким же, как если бы она вообще не выполнялась;
- файлы могут хранить гораздо больше информации, чем оперативная память компьютера.

4.1. Понятие формального файла

Файлы на Паскале являются абстрактным обобщением реальных файлов, и программа не содержит информации о физической природе файла. Файл представляется в виде *переменной* какого-либо файлового типа языка Паскаль. Такую переменную будем называть формальным файлом, в отличие от фактического файла, который хранится на диске. *Связывание* формального и фактического файлов на время выполнения программы производит операционная система. Отношения между формальными файлами, описанными в программе, и фактическими файлами, привязываемыми к программе на время ее выполнения операционной системой, похожи на отношения между формальными и фактическими параметрами процедуры.

После связывания формального и физического файлов все обращения к файлу на диске осуществляются через файловую (формальную) переменную.



Файловые переменные, описанные и используемые в программе, не могут участвовать в операторах присваивания.



При использовании файловой переменной в качестве формального параметра заголовков процедур и функций она всегда должна описываться как *параметр-переменная (с префиксом Var)*.

Связывание осуществляется с помощью стандартной процедуры

Assign (Имя_Формального_Файла, Имя_Фактического_Файла).

Например, если мы хотим работать с текстовым файлом, расположенным на диске *C* и имеющим имя *Data.txt*, необходимо написать в программе:

```
Var f : Text;           { Объявляем файловую переменную f }
                        { (вводим формальный файл типа   }
                        { «Текст»)                         }
Begin
  Assign                 { Связываем формальный файл f с   }
                        { фактическим, имеющим имя Data.txt }
      (f, 'C:\Data.txt' ); { и расположенным на диске C   }
  . . .
End.
```

4.2. Физические файлы

Физический файл в *MS DOS* представляет собой именованный набор данных на жестком или гибком диске или внешнее устройство. При этом *MS DOS* не делает особого различия между «традиционными» файлами и устройствами (портами коммутации). Так же как и файл, внешние устройства имеют свои фиксированные имена.

Физические файлы-устройства организованы как текстовые файлы и связываются с формальной файловой переменной текстового типа.

Имена физических файлов строятся по принципу формирования любого идентификатора языка Паскаль. Например, «*PRN*» - имя обращения к принтеру, подключенному к порту *LPT1*. Файл с этим именем открывается для записи в него, но при открытии его для чтения операционная система не выдает ошибки, а сразу выдает признак конца файла.

Имя «*CON*» может принадлежать сразу двум устройствам: клавиатуре или экрану, в зависимости от того, какое действие выполняется с файлом – чтение или запись.

4.3. Буфер ввода-вывода

С файловой системой Паскаля связано понятие буфера ввода-вывода.

Ввод и вывод информации с диска и на диск осуществляется через буфер. *Буфер ввода-вывода* – это область памяти, отводимая при открытии файла. Буфер будет заполняться информацией, пока не заполнится весь, и только тогда произойдет «сброс» информации на диск. Аналогично при чтении из файла считывается не столько, сколько запрашивается, а сколько уместится в буфер.

Механизм буферизации позволяет быстро и эффективно обмениваться информацией с внешними устройствами.



Для текстовых и нетипизированных файлов размер буфера можно установить непосредственно в программе по своему усмотрению.

4.4. Организация работы с файлом

Структурно файл организован как последовательность элементов (рис. 4.1).

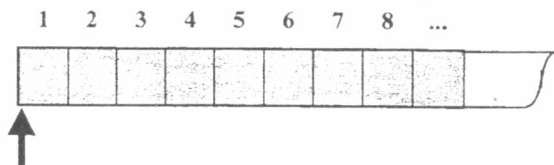


Рис. 4.1. Последовательный файл

Файл – это линейная структура, элементы в нем лежат один за другим. В файле не может быть «дыр».

У файла есть связанный с ним указатель, установленный перед элементом, с которым будет выполняться действие. При открытии файла он всегда находится в начале файла перед самым первым элементом. При выполнении действия с элементом файла (записи или чтения) указатель автоматически передвигается и устанавливается перед следующим элементом.



Элементы в файле нумеруются, начиная с 0.

После работы с последним элементом файла логическая функция *EOF (Имя_Формального_Файла)* выдает значение *True*.

Каждый цикл работы с файлом содержит, как правило, четыре действия:

1. Связывание формального и фактического файлов.
2. **Открытие файла (для записи или для чтения)**.
3. Непосредственно работа с файловыми данными (запись в файл или чтение из файла).
4. Закрытие файла.

4.5. Запись в файл

Файл создается или пополняется, когда в него что-нибудь *записывают*.

Каждая операция записи добавляет к файлу новый элемент.



Элементы могут добавляться только к концу последовательности.

При открытии файла для записи считается, что файл пуст, указатель стоит в начале (см. рис. 4.1) и указывает на позицию в файле, в которую будет производиться запись нового элемента. На рис. 4.2 изображена ситуация, когда в файле уже присутствуют 5 элементов, и указатель установлен на ту позицию, куда будет помещаться шестой элемент.

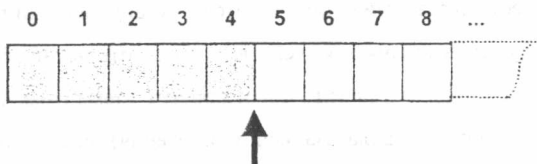


Рис. 4.2. Запись в файл

Операция установки указателя на начало файла и подготовки файла для записи в него новых элементов выполняется с помощью стандартной процедуры

Rewrite (Имя_Формального_Файла).

Эта операция одновременно уничтожает всю информацию, имеющуюся в файле.

Запись в файл осуществляется поэлементно с помощью процедуры

Write (Имя_Формального_Файла, Переменная_Для_Записи).



Записать в файл можно только весь элемент (компонент) целиком.

Если элемент имеет сложную структуру (массив, строка, запись), он также записывается в файл целиком, предварительно получив свои значения в оперативной памяти.

```

Var
  F : Text;           { Объявляем файловую переменную f }
                      { вводим формальный файл типа «Текст» }
  Ch : Char;          { Объявляем переменную символьного }
                      { типа для записи в файл }

Begin
  Assign (f, 'C:\Data.txt'); { Связываем формальный файл f с }
                             { фактическим, имеющим имя Data.txt и }
                             { расположенным на диске C }
  Rewrite(f);            { Открываем файл для записи }
  Ch := Readkey;          { Вводим символ с клавиатуры }
  Write (f, ch);          { Записываем в файл f значение }
                             { переменной ch }
  Close (f);              { Закрываем файл f }

End.
```

4.6. Чтение из файла

После создания файла мы можем прочитать из него информацию. Для этой цели необходимо вызвать процедуру чтения

Reset (Имя_Формального_Файла).

Эта процедура переместит указатель файла в его начало, как показано на рис. 4.1. При чтении из файла текущего элемента указатель автоматически сдвинется на одну позицию (один элемент) и будет готов к прочтению следующего элемента. При прочтении всех элементов указатель будет указывать на конец файла (рис. 4.3). В этой ситуации можно с помощью процедуры записи в файл дозаписать новый элемент.

Если файл, который мы открываем для чтения, пуст (не содержит ни одного элемента), то логическая функция *EOF* (*Имя_Формального_Файла*) выдаст значение *True*.

Чтение из файла осуществляется поэлементно с помощью процедуры

Read (*Имя_Формального_Файла*, *Переменная_Для_Чтения*).

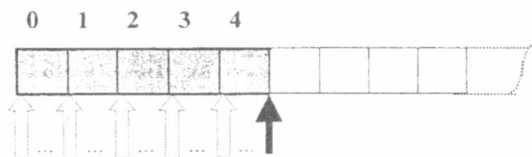


Рис. 4.3. Чтение из файла

```

Var
  F : Text;           { Объявляем файловую переменную f }
                        { вводим формальный файл типа «Текст» }
  Ch : Char;          { Объявляем переменную символьного }
                        { типа для записи в файл }
Begin
  Assign              { Связываем формальный файл f с }
    (f, 'C:\Data.txt'); { фактически, имеющим имя Data.txt }
                      { и расположенным на диске C }
  Reset(f);           { Открываем файл для чтения }
  Read (f, ch);        { Читаем из файла f элемент в }
                        { переменную ch }
  . . .
  Close (f);           { Закрываем файл f }
End.

```

Процедура *Read* считывает за одно прочтение один элемент файла. Прочитать из файла можно только весь элемент (компонент) целиком. Если элемент имеет сложную структуру (массив, строка, запись), он также читается из файла целиком, предварительно получив свое значение в файле.

4.7. Файловые типы языка Паскаль

Ввод-вывод информации в Паскале осуществляется через переменные файлового типа. Паскаль поддерживает три файловых типа:

- текстовые файлы (типа *Text*);
- типизированные (компонентные) файлы (типа *File of <Tun>...*);
- нетипизированные файлы (типа *File*).

Работа с этими типами файлов отличается друг от друга.

Обычно все файлы считаются файлами последовательного доступа. Однако к типизированным файлам (*File of <Tun>...*) и файлам без типа (*File*) помимо процедур, обеспечивающих последовательный доступ, можно применить процедуры прямого доступа (см. пп. 4.10 и 4.11).

Текстовый файл - это набор кодов *ASCII*, включая расширенные и управляющие коды. Текстовые файлы организованы по строкам, в конце текстового файла всегда находится специальный символ, называемый маркер конца файла. Примером текстового файла может служить сама программа на языке Паскаль. Такой файл создается путем набора символов с использованием клавиатуры.

Типизированные или компонентные файлы, в отличие от текстовых, хранят информацию не в символьном, а в цифровом, строковом, массивном, компонентном виде, т.е. в том виде, как она хранится в памяти компьютера.

Нетипизированные файлы могут содержать разнородные данные. Также как и типизированные, они хранят данные в машинном представлении. Но, если в типизированных файлах заранее определен тип хранения данных, то в нетипизированных хранится произвольный набор байт данных произвольной структуры.



Для всех типов файлов минимальной единицей хранения данных в них является байт.

Общие принципы работы с файлами едины, хотя и имеются различия в использовании некоторых процедур.

4.8. Общие процедуры работы с файлами

В табл. 4.1 приведены общие процедуры и функции, позволяющие работать с любым типом файлов. Специальные процедуры, характерные для какого-либо определенного типа файлов, будут описаны в соответствующих главах.

Таблица 4.1

Процедура или функция	Назначение
Assign (Var <i>f</i> ; FileName : String)	Предназначена для связывания файловой переменной <i>f</i> (формальный файл) с физическим файлом на диске, путь к которому хранится в переменной <i>FileName</i>
Close (Var <i>f</i>)	Предназначена для закрытия файла <i>f</i>
Reset (Var <i>f</i>)	Открывает файл <i>f</i> для чтения, устанавливает указатель на начало файла
Rewrite (Var <i>f</i>)	Открывает файл <i>f</i> для записи, устанавливает указатель на начало файла, стирает предыдущее содержимое файла, если оно было
Erase (Var <i>f</i>)	Предназначена для удаления файла <i>f</i> , файл при этом должен быть закрытым
Rename (Var <i>f</i> ; NewName : String)	Предназначена для переименования файла <i>f</i> , файл при этом должен быть закрытым. В <i>NewName</i> записывается новое имя файла
EOF (Var <i>f</i>): Boolean	Функция выдает результат True , если достигнут конец файла <i>f</i> , т.е. при прочтении последней записи

4.8.1. Процедура связывания

Процедура связывания

Assign (Var f; FileName : String)

Assign (Имя_Формального_Файла, Имя_Фактического_Файла)

устанавливает связь между формальным файлом (файловой переменной *f* заданного типа) и фактическим файлом (физическим файлом *MS DOS*), имя которого хранится в строковой переменной *FileName*. Строка *FileName* содержит имя файла, или полный путь к этому файлу и имя, или имя стандартного устройства.



Assign не занимается проверкой на корректность имени файла.

В связи с этим имя физического файла не должно содержать символов шаблонов «*» и «?» и должно быть уникальным. При попытке связать файл с некорректным именем возникнет ошибка при работе с этим файлом (открытие, закрытие, переименование).

Assign (f, 'Data.txt');	{ Связываем файл f с файлом Data.txt, }
	{ расположенным в текущем каталоге }
Assign (f, 'A:\ MyProg.pas');	{ Связываем файл f с файлом }
	{ MyProg.pas, расположенным на }
	{ диске A: }
FileName := 'D:\Gr614\Inanov.doc';	
Assign (f, FileName);	{ Связываем файл f с файлом }
	{ Ivanov.doc, расположенным на диске D }
	{ в каталоге Gr614. Имя файла и путь }
	{ к нему хранятся в строковой }
	{ переменной FileName }
Assign (f, 'PRN');	{ Связываем файл f с принтером }

При необходимости создать новый файл можно связать файловую переменную с несуществующим именем.

Assign (f, 'Data.txt');	{ Связываем файл <i>f</i> с файлом <i>Data.txt</i> , { расположенным в текущем каталоге	}
Rewrite (f);	{ Открываем файл <i>f</i> для записи	}
Write (f, v);	{ Пишем в файл <i>f</i> данные через переменную <i>v</i>	}
Close (f);	{ Закрываем файл <i>f</i>	}
Reset (f);	{ Открываем файл <i>f</i> для чтения	}
Read (f, v);	{ Читаем из файла <i>f</i> данные через { переменную <i>v</i>	}
Close (f);	{ Закрываем файл <i>f</i> (Обязательное { закрытие)	}
Rename (f, 'New.dat');	{ Даем файлу <i>f</i> новое имя «New.dat»	}
Erase (f);	{ Уничтожаем файл (удаляем с диска)	}

При связывании формальный файл *f* считается закрытым, размер выделяемого буфера ввода-вывода считается неопределенным.

Вне зависимости от того, сколько раз и какие действия будут выполняться с файлом на протяжении работы программы, связывание файла можно и достаточно проводить один раз перед началом выполнения операций с файлом.



Только после того как файл был связан, с ним можно производить какие-либо действия.

4.8.2. Процедура открытия

Процедура открытия

Reset (Var f)

Reset (Имя_Формального_Файла)

открывает формальный файл для чтения.

Процедура открытия

Rewrite (Var f)

Rewrite (Имя_Формального_Файла)

открывает формальный файл для записи.

Если открытие файла прошло успешно, то файл открыт соответственно для чтения или для записи и указатель устанавливается на начало файла, т.е. перед самым первым элементом.

Работа с файлом (запись в него или чтение) может производиться только после выполнения процедуры открытия (и только после нее!). Открытие одного и того же файла в программе может производиться сколько угодно раз. При этом закрытия файла не требуется.



Для корректной работы с данными, расположенными в файле, рекомендуется закрывать файл всегда после окончания работы с ним. Другими словами, количество вызовов в программе процедуры открытия должно соответствовать количеству вызовов процедуры закрытия.



Каждое новое открытие файла устанавливает указатель в начало. При этом процедура *Reset* сохраняет данные в файле неизменными, в то время как процедура *Rewrite* полагает, что файл пуст, и новые данные будут записываться в файл с первого элемента.

4.8.3. Процедура закрытия

Процедура

`Close ( Var f )`

`Close ( Имя_Формального_Файла )`

закрывает открытый формальный файл *f*.



Попытка закрыть уже закрытый файл приведет к ошибке.

После выполнения процедуры закрытия связь между формальным и фактическим файлом не нарушается, при этом файлы получают атрибут закрытого файла.

Если файл был открыт для чтения, он будет закрыт. Если файл был открыт для записи, то команда «закрыть файл» обеспечит запись в него оставшихся в буфере ввода-вывода данных. Если при открытом для записи файле произойдет сбой в программе, то данные, находящиеся в этот момент в буфере ввода-вывода, не перенесутся в файл и будут потеряны.



Закрытие файла гарантирует сохранность и полноту заполнения файла.

Данные, оставшиеся в буфере, не перенесутся в файл и в случае, если процедура закрытия файла вообще не будет написана.

Вызов процедуры закрытия файла всегда необходим перед вызовом процедур переименования файла (*Rename*) и удаления его с диска (*Erase*).

4.8.4. Процедура переименования

Процедура

Rename (Var f ; NewName : String)

Rename (Имя_Формального_Файла, Новое_Имя_Файла)

переименовывает фактический файл, связанный с формальным файлом *f*, и присваивает ему новое имя *NewName*.



Новое имя файла хранится в строковой переменной, должно быть уникальным и не должно содержать символов шаблонов «*» и «?».



Перед вызовом процедуры переименования файла необходимо его закрыть. Если файл закрыт не был, произойдет ошибка выполнения программы.

Предполагается, что исходный физический файл был связан с формальным файлом с помощью процедуры *Assign*. После переименования связь формального файла будет установлена с новым фактическим файлом. Старый файл при этом не резервируется, его имя будет замещено новым.

После переименования можно снова открывать файл и работать с ним, формальный файл будет теперь связан с новым именем.



Переименованный файл должен быть расположен в том же каталоге, что и исходный.

```

Var                                     { Объявляем файловую переменную f }
  f : file of String;                 { типа File of }

  S : String;                         { Объявляем переменную S строкового }
                                     { типа для чтения в нее из файла }

Begin

  Assign (f, 'Ivanov.pas');           { Связываем формальный файл f с }
                                     { фактическим, имеющим имя Ivanov.pas }
  Reset (f);                         { Открываем файл f для чтения }

  Read (f, S);                       { Читаем из файла f элемент (строку) в }
                                     { переменную S }
  Close (f);                         { Закрываем файл f }

  Rename (f, 'Petrov.pas');          { Переименовываем файл. Даем файлу }
                                     { новое имя Petrov.pas }

End.
```

Если в имени файла указывается путь к нему, то и у нового файла (в новом имени) должен быть указан этот же путь, т.к. процедура *Rename* предусматривает только переименование, но не перенос файла на новый диск.

Таким образом, если формальный файл *f* был связан с фактическим, имеющим имя *'Ivanov.pas'* и расположенным на диске *C:*, то попытка переименовать его в *'Petrov.pas'* и расположить на диске *A:* приведет к ошибке.

Assign (f, ' C: \ Ivanov . pas ');	{ Связываем файловую { переменную с файлом, { расположенным на диске C { Переименовываем файл и { переносим его на диск A }
Rename (f, ' A: \ Petrov . pas ');	
	!!! О Ш И Б К А !!!

4.8.5. Процедура удаления

Процедура

Erase (Var f)

Erase (Имя_Формального_Файла)

удаляет с диска фактический файл, связанный с формальным файлом *f*.

Предполагается, что исходный физический файл был связан с формальным файлом с помощью процедуры *Assign*. При попытке удалить фактический файл, предварительно не связанный с формальным файлом, возникнет ошибка.



Невозможно удалить с диска незакрытый файл, такая попытка вызовет ошибку.

Конструкция **Assign (f, FileName); Erase (f);** позволит удалить с диска фактический файл с именем *FileName*. Если файл с таким именем не существует, произойдет ошибка выполнения.

4.8.6. Функция анализа состояния файла

Логическая функция анализа состояния файла

EOF (Var f) : boolean

EOF (Имя_Формального_Файла) : boolean

проверяет, закончился ли файл, и возвращает *True*, если при чтении был достигнут конец файла, и *False* в противном случае. Функция возвращает *True*, если открываемый файл пуст или прочитан последний элемент файла. Состояние *EOF* автоматически обновляется при каждом обращении к

процедуре ввода данных. Для получения своего значения функция должна работать только с открытым файлом.

Назначение функции *EOF* - указывать на конец файла, поэтому для чтения из файла наиболее приемлемой будет конструкция:

While not EOF(f) do	{ Пока не конец файла <i>f</i>	}
Read(f, ...)	{ Читаем из файла <i>f</i>	}

Эта конструкция гарантирует, что как только будет прочитан последний элемент файла, т.е. будет достигнут конец файла, чтение из него прекратится, чего не гарантирует конструкция с использованием цикла *Repeat Until*. Для пустого файла такая конструкция вызовет ошибку.

Repeat	{ Повторяем	}
Read (f, ...);	{ Чтение из файла <i>f</i>	}
Until EOF (f);	{ До тех пор, пока не конец файла <i>f</i>	}

!!! О Ш И Б К А !!!

при пустом файле

4.9. Текстовые файлы

Одним из видов последовательных файлов является текстовый файл, компоненты которого - набор кодов *ASCII*, включая расширенные и управляющие коды.

Текстовые файлы организованы по строкам, в конце строки всегда находится специальный символ *EOLN (End Of LiNe)* - #13 (код 13 - *CR* - возврат каретки). Этот код может быть объединен с символом «прогона строки» #10 (код 10 - *LF*). Доступ к каждой строке возможен лишь последовательно, начиная с первой (рис. 4.4).

В конце текстового файла всегда находится специальный символ, называемый маркером конца файла (*EOF - End Of File*) - #26 (код 26 - обозначается символом ^Z). Пустой файл содержит только код 26.

Для указания типа таких файлов в языке Паскаль существует заранее определенный стандартный идентификатор *Text*.

Текстовые файлы играют особо важную роль в вычислениях и обработке данных: они обеспечивают взаимодействие между вычислительными системами и пользователем.

Особенностью работы с текстовым файлом является то, что параметры, значения которых вводятся и выводятся с помощью процедур *Read* или *Write*, могут быть не обязательно символьного типа *Char* или *String*, но и других простых типов: числовых, логических (рис. 4.4).



При записи чисел, строк и логических значений они преобразуются в символьный (текстовый) вид.

При чтении чисел и строк они автоматически преобразуются из текстового представления в машинное.

#13#10

Пример текстового файла #13#10

В файле могут храниться символьные строки #13#10

Или числа #13#10

Матрица #13#10

123 281 371 99 #13#10

4567 777 328 67 #13#10

15.44 821 3.14 10 #13#10

-0.456 #13#10

Конец текстового файла#13#10

#26

Рис. 4.4. Пример текстового файла

На рис. 4.4 коды #13, #10, #26 – управляющие, они занимают в файле по одному символу, на экран и принтер они не выводятся, но управляют процессом вывода.

Реально текстовый файл хранится в виде сплошной цепочки символов, включая управляющие, и разбивается на строки лишь при выводе содержимого файла на печать или экран.

При работе с текстовым файлом необходимо выполнить, как правило, четыре действия: связать, открыть, поработать, закрыть.

Var

```
F : Text; { Объявляем файловую переменную f типа Text }
```

```
Ch : Char;           { Объявляем вспомогательную переменную Ch      }
                     { символьного типа, для чтения в нее из файла    }
```

Begin

Assign (f, 'Data.txt');	{ Связываем формальный файл f с
	{ фактическим, имеющим имя Data.txt }

```
Reset(f); { Открываем файл для чтения }
```

Read (f, Ch);	{ Читаем из файла <i>f</i> первый символ в }
	{ переменную <i>Ch</i> }
Close (f);	{ Закрываем файл }
End.	

Текстовые файлы в Паскале - это не *File of Char*. Равенство между этими файлами можно поставить лишь со значительными оговорками.

4.9.1. Чтение и запись в текстовых файлах

После открытия файла чтение и запись в него осуществляется процедурами ввода и вывода.

Чтение

Read (Var f; Список ввода)

Read (f, Ch); или **Read (f, a,b,c,d, . . .);** и

ReadLn (f, S); или **ReadLn (f, a,b,c,d, . . .);**

Запись

Write (Var f; Список вывода)

Write (f, Ch); или **Write (f, a,b,c,d, . . .);** и

WriteLn (f, S); или **WriteLn (f, a,b,c,d, . . .);**

Имеются две стандартные файловые переменные для текстового файла: *Input* и *Output*. Переменная *Input* по умолчанию связана с клавиатурой, а *Output* - с экраном дисплея. Эти устройства ввода-вывода автоматически открываются при запуске программы и закрываются после окончания ее работы. Если при работе с файлами типа *Text* в процедуре или функции не указана файловая переменная, считается, что используется файловая переменная *Input* или *Output*.

Список ввода - это перечисленные через запятую имена переменных, в которые помещаются введенные значения. Количество элементов в списке ввода или вывода может быть любым, в том числе и пустым.

Оператор *Read (a, b)* полностью эквивалентен оператору *Read (Input, a, b)*, а оператор *Write (a,b)* - оператору *Write (Output, a,b)*.

4.9.1.1. Процедуры *Read / ReadLn*

Тип переменных при вводе из текстового файла (в том числе и с клавиатуры) может быть только целым, вещественным, символьным.



Данные сложных структур (массивы, записи, множества и др.) вводятся только *по элементам* или по полям, если это запись.



В текстовый файл нельзя записать весь массив или всю запись целиком. Попытка записать *Read (f, Vector)* или *Read (f, Rec)* приведет к ошибке!

При вводе из текстового файла два числа считаются разделенными, если между ними есть хотя бы один пробел, или символ (ы) табуляции (#9), или символ (ы) конца строки (#13).

```

Var
  I : Integer;           { Переменная целого типа }
  R : Real;              { Переменная вещественного типа }
  S : String;            { Строка }
  Ch : Char;             { Переменная символьного типа }
  Vector : array[1..5]   { Массив
                        of Byte;
  Rec = Record           { Запись
  X, Y : Real;           { Поля записи вещественного типа }
  N    : Word;           { Поле записи целого типа }
  S    : String;         { Поле записи типа строка }
End;
F : Text;               { Файловая переменная для текстового
                        { файла }
Begin
  Assign (f, ' Data.txt '); { Связываем формальный файл f с
                        { фактическим, имеющим имя Data.txt }
  Reset(f);              { Открываем файл для чтения }

```

Read (f, I, R);	{ Читаем из файла f два элемента в }
	{ переменные целого и действительного }
	{ типа }
Read (f, Ch);	{ Читаем из файла f элемент в }
	{ переменную символьного типа }
ReadLn (f, S);	{ Читаем из файла f всю строку целиком }
Read (f, Rec.X, Rec.N);	{ Читаем из файла f поля записи }
For i := 1 to 5 do	{ Читаем из файла f пять элементов в }
Read(f, Vector[i]);	{ массив }
Close (f);	{ Закрываем файл }
End.	

Если содержимое текстового файла читается в символьную переменную, то символ пробел, который являлся разделителем для чисел, теперь будет являться просто символом.

При чтении из файла строк можно воспользоваться процедурой *ReadLn*. При вводе строки с клавиатуры в конце ввода пользователь нажимает клавишу ввода и в файл записывается сначала вся строка, а затем код конца строки (#13). Процедура *ReadLn (f, S)* позволит прочитать из файла f в строковую переменную S всю строку целиком и распознает «разделитель», т.е. признак конца строки. Сам разделитель в строку не считывается. Если же в файл вводились строки одинаковой длины последовательно одна за другой без использования кода #13 в качестве разделителя, то процедура *Read(f, S)* прочтает в строку S количество символов, равное объявленной длине строки S.

Процедура *ReadLn* считывает значения в текущей строке и переводит позицию в начало следующей строки в независимости от того, есть или нет в ней непрочитанные еще элементы.

Так, при попытке чтения строки из текстового файла, содержащей числовые значения

333 444 555 777,

оператор *ReadLn(f, a, b)*; поместит в числовые переменные *a* и *b* первые два числа и переведет позицию в начало следующей строки. Два последних числа (555 и 777) прочитаны не будут.

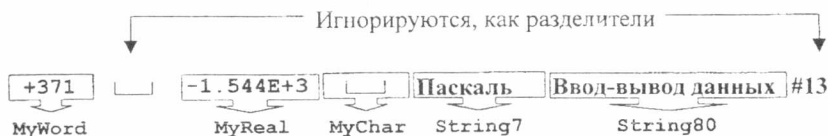
Вызов *ReadLn(f)* вообще пропустит строку в файле. Вызов *ReadLn* без указания файла будет ждать нажатия клавиши ввода.

4.9.1.2. Комплексный ввод

Список ввода в процедурах ввода предусматривает наличие более одного параметра, причем элементы этого списка могут быть разных типов. Такой комплексный ввод может содержать скрытые ошибки:

```
Read (MyWord, MyReal, MyChar, String7, String80);
```

При вводе разнотипные данные разобьются следующим образом:



Разделителем между числовыми данными может быть символ пробела () или признак конца строки (#13). В данном примере символьная переменная *MyChar* может принять только значение пробел (или #9 или #13), т.к. за символьной переменной следует переменная типа *String7*. Таким образом, вещественная переменная *MyReal* не будет прочитана, произойдет сбой. Переменная *String7* будет заполнена полностью, а *String80* заполнится только теми символами, которые находятся до разделителя #13.



Чтобы избежать сбоев при вводе, лучше отказаться от смешанных списков, по крайней мере, при вводе с клавиатуры.

4.9.1.3. Процедуры *Write* / *WriteLn*

Операторы *Write* и *WriteLn* предназначены для записи значений из списка вывода в текстовый файл или на экран, если файл не указан (предполагается файл *Output*). Значения переменных в списке, как и при вводе, могут быть лишь целые, вещественные, символьные и строковые. Сложные структуры данных, такие как массивы, множества, записи, выводятся по элементам, по полям.

Процедура *Write* выводит данные в текущую строку и не закрывает ее, т.е. следующие данные запишутся в ту же строку. Формально длина строки в текстовом файле не ограничена. Если вывод строки происходит на экран дисплея и длина строки превышает размер экрана, то продолжение печати строки будет происходить с начала следующей строки экрана. Если при выводе достигнут не только горизонтальный, но и вертикальный край экрана, т.е. символ окажется в правом нижнем углу экрана, то произойдет автоматическая прокрутка экрана на одну строку.

Процедура *WriteLn* выводит данные в текущую строку и закрывает ее, при этом вывод другой строки будет происходить с первой позиции следующей строки. В конец строки дописывается символ конца (#13, объединенный с #10).

Вызов процедуры *WriteLn* или *WriteLn (f)* без списка вывода переведет курсор на экране в начало следующей строки или создаст пустую строку в файле, содержащую только символ конца строки (#13).

Список вывода может содержать *Write* и *WriteLn*, может содержать константы, переменные, выражения, вызовы процедур, при условии, что они имеют соответствующие типы и разделены запятыми:

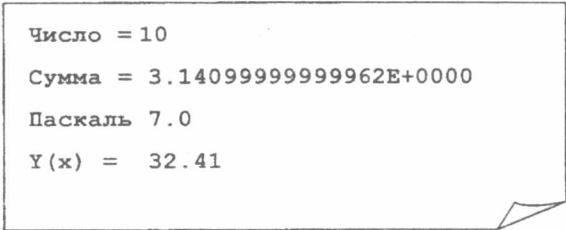
```
WriteLn ('Число = ', MyWord, #13#10' Сумма = ', MyReal);  
WriteLn (String7 + ' ' + MyChar + '.0');  
WriteLn ('Y(x) = ', MyWord*MyReal*cos(x) :7:2);
```

Символьные (строковые) константы, написанные в кавычках, выводятся на экран или в файл в том виде, в каком они записаны в вызове процедуры. Числовые значения будут выводиться по-разному.

Пусть в переменных хранятся следующие значения:

```
MyWord = 10, MyReal = 3.141, MyChar = "7",  
String7 = "Паскаль", x = 0.
```

Тогда информация на экране после отработки процедуры вывода будет выглядеть следующим образом:



```
Число = 10  
Сумма = 3.140999999999962E+0000  
Паскаль 7.0  
Y(x) = 32.41
```

$\{10 * 3,141 + 1 = 32,41\}$

Управляющие коды #13#10 переведут курсор в начало следующей строки. Переменная целого типа выведется в том виде, как она хранится, если число отрицательное, то число выведется со знаком минус. Переменная вещественного типа выведется в экспоненциальном виде, если не указан формат вывода, или согласно указанному формату (подробно см. п. 4.9.1.4).

4.9.1.4. Форматы вывода данных

При выводе значений на экран или в текстовый файл можно указывать формат вывода. Для строчных и символьных значений формат задается одним числом, отделенным от значения двоеточием:

```
Write (MyWord : 4);  
Write (MyChar : 2);
```

Число, стоящее после имени переменной, показывает количество позиций, отводимое на вывод значения. Так, для вывода переменной целого типа *MyWord* выделено 4 позиции. Если число занимает меньше позиций, оно будет прижато к правому краю, а перед ним будет поставлено недостающее

количество пробелов. Если `MyWord = 10`, напечатается число в 4 позиции следующим образом: `_ _ 1 0 .`

Но если наоборот (формат мал), то значение будет выводиться, игнорируя спецификацию.

Для вывода вещественного числа формат содержит уже два числа. Первое число соответствует общему количеству позиций, отводимых на вывод, второе — снова через двоеточие — количеству позиций на дробную часть.

`Write (MyReal : 7 : 2);`

Если `MyReal = 3.14`, напечатается число в 7 позициях, включая точку, при этом две позиции будут отведены на дробную часть: `_ _ _ 3 . 1 4 .`



Первое число формата должно быть больше второго, т.к. включает в себя:

- одну позицию на знак числа;
- количество позиций на целую часть числа;
- одну позицию на саму точку;

количество позиций на дробную часть.

Поэтому бессмысленны форматы типа

~~`Write ( MyReal : 5 : 4 );`~~

Если количество позиций в числе меньше выделенного количества, число прижимается к правому краю. При некорректном задании формата игнорируется только первый его параметр, а число знаков после точки устанавливается всегда корректно, но не превышает точности типа.

Если `MyReal = 3.1415`, а на дробную часть выделено позиций меньше, то число при выводе округлится до заданного числа знаков, хотя само число не изменится.

`Write (MyReal : 7 : 3);`

`_ _ 3 . 1 4 2 .`

Можно вывести вещественное число как целое, т.е. без дробной части. Для этого достаточно указать второй параметр равным 0. Если $\text{MyReal} = 1544.281$, то результат работы процедуры **Write (MyReal : 7 : 0);** будет выглядеть

— — — 1 5 4 4 .

При необходимости вывода вещественных значений в экспоненциальном формате достаточно задать только одно число, указывающее количество позиций, в которых надо разместить число.

Минимальное количество позиций равно 8. Эти 8 позиций складываются:

- 1) из знака числа,
- 2) цифры до точки,
- 3) самой точки,
- 4) хотя бы одной цифры после точки,
- 5-8) — четырех знаков под степень.

При формате меньше восьми позиций он автоматически устанавливается равным восьми.

Максимальное число позиций определено типом вещественного числа, и дальнейшее увеличение формата эффекта не даст.

4.9.2. Процедуры и функции для работы с текстовыми файлами

Кроме общих процедур и функций, предназначенных для любых типов файлов (см. табл. 4.1), в Паскале есть процедуры, ориентированные на работу с текстовыми файлами (табл. 4.2).

Таблица 4.2

Процедура или функция	Назначение
SetTextBuf (Var f:Text; Var Buf; [BufSize: Word])	Устанавливает размер буфера для файла f размером в BufSize байт

Продолжение табл. 4.2

Flush (Var f: Text)	Позволяет освобождать буфер до того, как буфер будет заполнен полностью.
EOLn (Var f: Text): Boolean	Возвращает значение True, если достигнут конец строки или конец файла.
Append (Var f: Text)	Открывает текстовый файл для дозаписи новых элементов в конец файла.
SeekEOLn (Var f: Text): Boolean	Возвращает значение True, если до конца строки остались только пробелы и (или) знаки табуляции
SeekEOF (Var f: Text): Boolean	Возвращает значение True, если до конца файла остались только пробелы, знаки табуляции и маркеры конца строки.

4.9.2.1. Процедура установки размера буфера

Процедура

SetTextBuf (Var f:Text; Var Buf; [BufSize: Word])

*SetTextBuf (Имя_Формального_Текстового_Файла,
Буфер_Ввода_Вывода,
[Размер_Буфера])*

устанавливает размер буфера ввода-вывода для текстового файла *f* размером в *BufSize* байт. По умолчанию размер буфера для текстового файла равен 128 байт. Увеличение буфера обмена необходимо при интенсивном обращении к физическим файлам.

Увеличить размер буфера можно только после выполнения процедуры связывания «формального» файла с фактическим файлом, до начала

выполнения процедур чтения или записи в файл. Лучше, если эта процедура будет использоваться до открытия файла.

Тип переменной *Buf*, в которой хранится буфер, не имеет значения, важен ее размер. Размер буфера определяется в необязательном параметре *BufSize*. Если в вызове процедуры число *BufSize* не указано, то размер буфера равен размеру переменной *Buf*.



Задание *BufSize* больше, чем размер самой *Buf*, приведет к потере данных.

```

Var
  f : Text;                { Файловая переменная для текстового }
                           { файла }
  Buf_Arr : array[1..1024] { Буфер размером в 1024 байта }
                           of Byte;
  Buf_S : String;          { Буфер размером в строку }

Begin
  Assign (f, ' Data.txt '); { Связываем формальный файл f с }
                           { фактическим, имеющим имя Data.txt }
  SetTextBuf(f, Buf_Arr);   { Устанавливаем буфер размером в }
                           { 1024 байта }
  Reset(f);                 { Открываем файл для чтения }

  Read (f, . . . );         { Читаем из файла f }

  . . .

  Close (f);                { Закрываем файл }

  Assign (f, ' New.txt ');  { Связываем формальный файл f с }
                           { фактическим, имеющим имя New.txt }
  SetTextBuf(f, Buf_S);     { Устанавливаем буфер размером в }
                           { 256 байт }
  Reset(f);                 { Открываем файл для чтения }

  Read (f, . . . );         { Читаем из файла f }

  . . .

  Close (f);                { Закрываем файл }

End.
```

Будучи однажды установленным для текстового файла f , буфер не меняет своего места и размера до следующего вызова *SetTextBuf* или *Assign* с той же файловой переменной f .

Переменная, отводимая под буфер, должна быть описана как глобальная.



```

Var f : Text;
Procedure SetNewFile ( Var f : Text );
Var
  Buf : array[1..1024] of Byte; { Буфер размером в 1024 байта }
  FileName : String;           { Имя файла }
Begin
  ReadLn (FileName);           { Вводим имя файла }
  Assign (f, FileName);        { Связываем формальный файл f с
                                { фактическим }
  SetTextBuf(f, Buf);          { Устанавливаем буфер размером в
                                { 1024 байта }
  Rewrite(f);                  { Открываем файл }
End {SetNewFile};

Begin                          { Начало исполнительной части }
  SetNewFile (f);              { Вызов процедуры SetNewFile }
  Write (f, ' . . . ');        { Попытка записи в файл f }

  Далее программа будет работать с ошибкой!!!

  Close (f);                   { Закрываем файл }
End.

```

Рис. 4.5. Пример использования процедуры *SetTextBuf*

Программа, представленная на рис. 4.5, содержит скрытую ошибку, смысл которой заключается в следующем. Изменение размера буфера происходит, как и положено, до выполнения процедур связывания и открытия, сама же

переменная *Buf* описана локально в процедуре *SetNewFile*, а попытка работы с ней осуществляется уже вне процедуры. Память, выделенная в стеке под переменную *Buf*, освобождается после окончания работы процедуры, и переменная становится недоступной для работы с ней. Область буфера становится общедоступной и быстро заполнится «мусором», который затем запишется в файл. При этом не возникнет никакой ошибки, что и является самым неприятным фактом в приведенном примере.



Размер буфера лучше всего делать кратным 512 байт, т.к. диск читается по секторам и дорожкам, и длина считываемой в одном месте порции информации колеблется от 512 до 4096 байт для гибких и жестких дисков.

4.9.2.2. Процедура освобождения буфера

Процедура

Flush (Var f : Text)

Flush (Имя_Формального_Текстового_Файла)

предназначена для освобождения буфера до того, как он будет заполнен полностью. Работает процедура с файлами, предварительно открытыми для записи с помощью процедур *Rewrite* или *Append*. Процедура *Flush* принудительно освобождает буфер и переносит его содержимое в файл, не дожидаясь заполнения буфера. При большом объеме буфера его содержимое может не попасть на диск в случае, если программа прервется в процессе работы. Этого можно избежать, если перед «аварийными» частями программы поставить вызовы процедуры *Flush*. Процедура *Flush* не закрывает файл.

4.9.2.3. Процедура открытия файла для дозаписи

Процедура

Append (Var f : Text)

Append (Имя_Формального_Текстового_Файла)

предназначена для открытия файла для дозаписи. Она применима только к существующим файлам. В отличие от процедуры *Rewrite*, также предназначенной для записи, процедура *Append* не уничтожает содержимое файла, а подготавливает файл для дозаписи новой информации в конец файла.



Попытка использовать процедуру *Append* для несуществующего файла приведет к ошибке.

4.9.2.4. Функции анализа состояния текстового файла

1. Функция *EOLn*

Функция *EOLn* (*End Of Line* — конец линии)

***EOLn* (Var f : Text) : Boolean**

***EOLn* (Имя_Формального_Текстового_Файла) : Boolean**

анализирует текущее положение в текстовом файле, открытом для чтения. Функция возвращает значение *True*, если достигнут конец строки (#13) или конец файла (#26), и *False* во всех других случаях.

Примером использования функции *EOLn* может служить следующая задача (рис. 4.6). В текстовом файле хранится матрица действительных чисел. Необходимо узнать количество строк и столбцов этой матрицы.

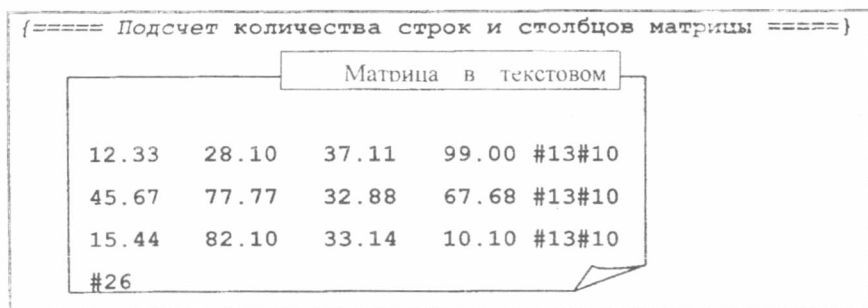


Рис. 4.6. Подсчет количества строк и столбцов матрицы (см. с. 95)

```

Var                                     { Текстовый файл }
F : Text;
FileName : String; { Имя файла }
x : Real;          { Переменная для хранения элемента файла }
N_Row,            { Количество строк в матрице }
N_Column : Byte;  { Количество столбцов в матрице }

Begin
  ReadLn (FileName); { Вводим имя файла }
  Assign (f, FileName); { Связываем формальный файл f с }
                        { фактическим }
  Reset(f); { Открываем файл для чтения }
  N_Row := 0; { Начальное значение для количества }
              { строк }
  While not EOF(f) do begin { Пока не конец файла f }
    N_Column := 0; { Начальное значение для количества }
                  { столбцов }
    While not EOLn(f) do begin { Пока не конец строки файла f }
      Read (f, x); { Читаем из файла f в переменную x, }
                  { двигаясь по строке }
      Inc (N_Column); { Увеличиваем счетчик столбцов }
    end;
    ReadLn (f); { Читаем из файла f признак конца строки }
               { (#13) }
    Inc (N_Row); { Увеличиваем счетчик строк }
  end;
  Close(f); { Закрываем файл }
End.

```

Рис. 4.6. Окончание

Функция *EOLn* может существовать и без параметра. В этом случае считается, что она работает со стандартным файлом *Input*, т.е. с клавиатурой. Для файла *Input* функция *EOLn* выдает значение *True* не перед, а после

прочтения кода конца строки (#13). Функция *EOLn* без параметра никогда не выдает значения *False*.

II. Функция *SeekEOLn*

Функция

***SeekEOLn* (Var f : Text) : Boolean**

***SeekEOLn* (Имя_Формального_Текстового_Файла) : Boolean**

так же, как и функция *EOLn*, анализирует текущее положение в текстовом файле, открытом для чтения.

Функция возвращает значение *True*, если до конца строки (#13) или конца файла (#26) в файле находятся только пробелы (#32) и (или) символы табуляции (#9). Функция также вернет *True*, если достигнут конец строки (#13) или конец файла (#26).

Таким образом, функция *SeekEOLn* как бы «видит» конец строки (или конец файла) издали, сквозь символы пробелов и табуляции. Данная функция наиболее удобна в случае, если в текстовом файле хранятся числовые значения, так как ускоряет обработку файла, «не обращая внимания» на пробелы в конце.

III. Функция *SeekEOF*

Функция

***SeekEOF* (Var f : Text) : Boolean**

***SeekEOF* (Имя_Формального_Текстового_Файла) : Boolean**

аналогично функциям, описанным выше, анализирует текущее положение в текстовом файле, открытом для чтения.

Функция *SeekEOF*, в отличие от функции *SeekEOLn*, возвращает значение *True* только в случае достижения конца файла (#26) или, если, начиная с текущей позиции и до конца, в файле находятся только пробелы (#32) и (или) символы табуляции (#9) и (или) символы конца строк (#13). Функция также вернет *True*, если достигнут конец файла (#26).

Так же, как и функция *SeekEOLn*, функция *SeekEOF* как бы «видит» конец файла издали. Символы пробелов, табуляции и концов строк являются для нее «прозрачными». Для файлов с числовыми значениями команда

```
While not SeekEOF (f) do . . .
```

будет выполняться быстрее, чем

```
While not EOF (f) do . . .
```

если в конце файла будут находиться пустые строки.

4.10. Компонентные файлы

Типизированные или компонентные файлы – это файлы с явно заданным типом компонента: *File Of Typ_Компонента*. Таким образом, все компоненты, их иногда называют записями, одного типа, одной структуры.

В качестве компонента может выступать как любой скалярный тип, так и тип, имеющий сложную структуру: массив, запись.

Type

```
Vector = array[1..5] of Integer; { Определяем новый тип
                                   массива Vector }
Matrix = array [1..10,1..15] of Real; { Определяем новый
                                         { тип массива Matrix }
Student = Record { Определяем новый тип записи }
    FIO      : String [ 20 ]; { Student }
    N_Gr     : Word;
    Session  : Vector;
End;
Faculty = File Of Student; { Файл с компонентом типа
                             { запись }
Marks = File Of Vector; { Файл с компонентом типа
                           { вектор }
MatrFile = File Of Matrix; { Файл с компонентом типа
                             { матрица }
IntFile = File Of Integer; { Файл с компонентом типа
                             { целое число }
RealFile = File Of Real; { Файл с компонентом типа
                           { действительное число }
Str_File = File Of String; { Файл с компонентом типа
                             { строка }
```



Компонентом файла не может быть сам файл или объект. Если файловый тип будет определен как элемент записи, а запись будет определена как компонент файла, это также будет недопустимо и вызовет ошибку.

Type

```

File_Faculty = File Of Faculty;    { Файл с компонентом типа }
                                     { файл записей }
File_Marks = File Of Marks;        { Файл с компонентом типа }
                                     { файл векторов }
File_File = File Of File Of Real;  { Файл с компонентом типа }
                                     { файл }

```

Как и с любым файловым типом Паскаля, с компонентным файлом можно выполнять четыре действия: связать, открыть, поработать, закрыть.

Type

```

Vector = array[1..5] of Integer;    { Определяем новый тип }
                                     { массива Vector }
Marks = File Of Vector;              { Тип файла с компонентом }
                                     { типа вектор }

```

Var

```

F : Marks;                           { Объявляем файловую }
                                     { переменную f типа File of ... }
V : Vector;                           { Объявляем вспомогательную }
                                     { переменную V такого же типа, }
                                     { как компонент файла, для }
                                     { чтения в нее из файла }

```

Begin

```

Assign (f, ' Session . dat ');        { Связываем формальный файл f }
                                     { с фактическим, имеющим имя }
                                     { Session . Dat }
Reset(f);                             { Открываем файл для чтения }

Read (f, V);                           { Читаем из файла f первый }
                                     { элемент в переменную V }

Close (f);                             { Закрываем файл }

```

End.

4.10.1. Чтение и запись в компонентном файле

Для компонентных файлов обе процедуры (*Reset* и *Rewrite*) устанавливают режим «чтение/запись» в соответствии со значением предопределенной системной переменной *FileMode* (оно по умолчанию равно 2), т.е. независимо от выбора процедуры открытия, файл открывается и для чтения и для записи.

Это касается только типизированных и нетипизированных файлов и будет неправильным для текстовых.

Этот порядок нарушится только в том случае, если значение *FileMode* будет равно 0, что соответствует режиму «чтение», или равно 1, что соответствует режиму «запись».

После открытия файла чтение из файла и запись в него осуществляется стандартными процедурами ввода и вывода.

Read (Var f; Переменные типа содержимого файла)

Процедура чтения очередного компонента из файла *f*.

Write (Var f; Переменные типа содержимого файла)

Процедура записи очередного компонента в файл *f*.

```
Reset(f);           { Открываем файл для чтения }
Read (f, V);        { Читаем из файла f первый элемент в }
                   { переменную V }
Read (f, a,b,c,d);  { Читаем из файла f четыре первых элемента в }
                   { переменные a, b, c, d }
                   { Переменные V, a, b, c, d должны быть }
                   { такого же типа, как компонент файла }
Rewrite(f);         { Открываем файл для записи }
Write (f, V);        { Пишем в файл f элемент, хранящийся в }
                   { переменной V }
Write (f, a,b,c,d);  { Пишем в файл f четыре элемента, хранящиеся }
                   { в переменных a, b, c, d }
                   { Переменные V, a, b, c, d должны быть }
                   { такого же типа, как компонент файла }
```

Первым аргументом процедур чтения и записи является формальный файл, предварительно связанный с физическим файлом на диске. Следующими аргументами будут одна или несколько буферных переменных, через которые и происходит «общение» между оперативной памятью и диском.

При создании файла буферная переменная работает как автобус, впускающий пассажиров на остановке: с появлением каждого следующего оператора записи *Write* содержимое файла увеличивается на один компонент (рис. 4.7).

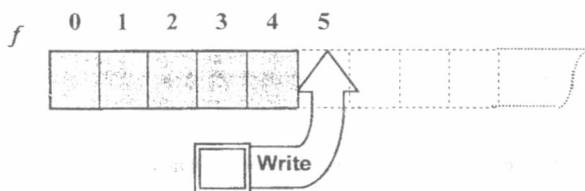


Рис. 4.7. Запись в файл одного компонента

При считывании из файла действие буферной переменной подобно окошку, через которое мы смотрим на файл. Процедуры *Read* и *Reset* передвигают его вперед-назад по файлу. *Read* сдвигает вперед на один элемент, *Reset* - в начало файла (рис. 4.8).

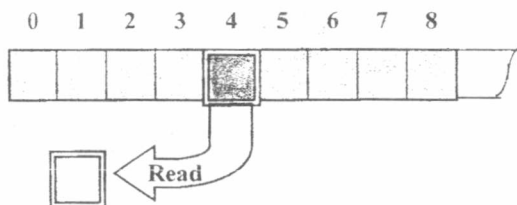


Рис. 4.8. Чтение из файла f одного компонента



Если компонентом файла является сложная структура (массив, запись), то чтение и запись происходит целиком всего компонента.


```

Const
    { N_Stud – количество студентов на факультете }
    N_Stud = 100;

Type
    Mark = 2..5;           { Диапазон оценок, получаемых студентами }
    Vector = array [1..5] of Mark; { Новый тип массива – Vector, }
                                   { содержит 5 оценок в сессию }
    Marks = File Of Vector; { Тип файла с компонентом типа вектор }

Var
    F : Marks;           { Объявляем файловую переменную f типа File of }
    V : Vector;          { Объявляем вспомогательную переменную V такого же }
                           { типа, как компонент файла, для работы с файлом }
    I, j : Byte;         { Параметры цикла }

Begin
    Assign (f, 'Session.dat'); { Связываем формальный файл f с }
                                { фактическим, имеющим имя }
                                { Session . Dat }
    Rewrite(f);               { Открываем файл для записи }

    For i:=1 to N_Stud do begin { Вводим оценки 100 студентов }
        Write ('Введите оценки за экзамены' );
        For j:=1 to 5 do
            Readln (V[j]);      { Вводим 5 оценок каждого студента в }
                                { массив V }
            Write (f, V);        { Пишем в файл f элемент типа Vector }
        End;
    Close (f);                 { Закрываем файл }

End.

```

Типизированные файлы имеют более строгую структуру хранения данных, нежели текстовые. При записи в них записывается машинное представление компонента. Компоненты ничем не разделены, и переход от одного компонента к другому осуществляется легко благодаря тому, что тип компонента, а значит и его размер (в байтах), жестко определены.



В типизированном файле нет понятия (символа) конца строки или конца файла, как в текстовом файле, поэтому для записи или чтения не имеют смысла процедуры *WriteLn* и *ReadLn*.

Их использование приведет к ошибке.



Типизированный файл, компонентом которого является символ: *File Of Char*, не одно и то же, что текстовый файл типа *Text*, т.к. не содержит управляющих символов и символа конца файла.

Типизированный файл, компонентом которого является строка: *File Of String [30]*, также не похож на текстовый файл, т.к. даже если строки будут содержать меньше, чем 30 символов, все равно для каждой выделится память в 31 байт (30 - на хранение символов строки и 1 - на хранение ее истинной длины). Часть строки будет заполнена нужными символами, в остальной части будет «мусор».



Размер буфера в типизированном файле устанавливается автоматически, исходя из размеров компонента, и не может быть изменен пользователем.

4.10.2. Прямой и последовательный доступ к файлам

Файл – последовательная структура данных. Элементы в файле «лежат» последовательно, друг за другом, доступ к ним осуществляется по очереди: пока не будет прочитан первый элемент, не может быть прочитан второй. Если необходимо прочитать из файла 10-й элемент, достаточно написать следующий фрагмент программы:

```

Reset(f);           { Открываем файл для чтения первого элемента }
For i:=1 to 10 do
  Read (f, V);       { Читаем из файла f первый элемент в переменную V }
                    { В переменной V останется последний 10-й элемент }

```

Таблица 4.3

Процедура или функция	Назначение
FileSize (Var f) : LongInt	Возвращает число компонентов в открытом файле f
FilePos (Var f) : LongInt	Возвращает номер компонента в открытом файле f, готового к чтению или записи
Seek (Var f; N : LongInt)	Устанавливает текущим компонентом в открытом файле f компонент с номером N, отсчитанный от нулевого
Truncate (Var f)	Отсекает часть открытого файла f, начиная с компонента, стоящего за только что считанным, и подтягивает на это место конец файла

При длинном файле такой способ доступа к элементу не рационален. В Паскале предусмотрены процедуры, обеспечивающие прямой доступ к файлу (см. табл. 4.3).



Процедуры и функции прямого доступа, описанные в табл. 3, могут использоваться только при работе с компонентными файлами. Для работы с текстовыми файлами эти процедуры не определены и компьютер выдаст сообщение об ошибке.

Прямой доступ к файлу более универсален, чем последовательный: он позволяет в любой момент получить доступ к произвольному компоненту файла вне зависимости от того, где в данный момент находится указатель файла и какой компонент был прочитан. Прямой доступ к файлу не исключает возможности последовательного доступа к компонентам.

Прямой доступ к файлу возможен благодаря тому, что каждый компонент файла имеет жестко заданную длину. Адресация компонентов осуществляется по их порядковым номерам, которые присваиваются компонентам в процессе создания файла.

4.10.3. Функции определения размера и позиции в файле

Функция

FileSize (Var f) : LongInt

FileSize (Имя_Формального_Файла) : LongInt

определяет размер открытого файла *f* (количество компонентов в нем). Для файла, изображенного на рис. 4.8, эта функция вернет значение 9. Для пустого файла функция *FileSize* вернет значение 0.

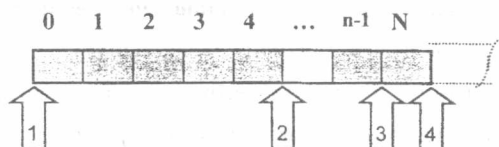
Функция

FilePos (Var f) : LongInt

FilePos (Имя_Формального_Файла) : LongInt

определяет позицию компонента в открытом файле *f*, готового к чтению или записи. Для файла, изображенного на рис. 4.7, эта функция вернет значение 5.

Позиция в файле – это номер условной границы между компонентами файла. При открытии файла номер текущей позиции всегда равен 0. После прочтения или записи компонента указатель сдвинется и позиция будет равна 1. После прочтения последней записи указатель будет иметь номер, равный количеству компонентов в файле (рис. 4.9).



Для указателя с номером 1 *FilePos* вернет значение 0.

Для указателя с номером 2 *FilePos* вернет значение 5.

Для указателя с номером 3 *FilePos* вернет значение *n*.

Для указателя с номером 4 *FilePos* вернет значение, равное *FileSize*.

Рис. 4.9. Позиции в файле

4.10.4. Процедура позиционирования

Процедура

Seek (Var *f* ; N : LongInt)

Seek (Имя_Формального_Файла , Номер компонента)

устанавливает указатель в открытом файле *f* непосредственно перед тем компонентом, который будет считан или записан. Данная процедура осуществляет прямой доступ к любому компоненту файла. Параметр *N* содержит номер условной границы между компонентами.

Для прочтения первого компонента файла необходимо записать

Seek (f , 0); { 0 – номер границы перед первым компонентом }

Для прочтения последнего компонента файла необходимо записать

Seek (f , FileSize(f) - 1); { *FileSize(f) - 1* – номер границы перед последним компонентом }

Команда *Seek (f , FileSize(f));* установит указатель после последнего компонента.

Процедура *Seek* по принципу позиционирования похожа на функцию *FilePos*, с той лишь разницей, что первая устанавливает указатель в заданную позицию, а вторая только анализирует эту позицию.



Попытка задать параметр N больше, чем реальное количество компонентов, вызовет ошибку ввода-вывода.

Рассмотрим программу, в которой необходимо в существующий файл записать новый элемент, не изменяя старые. Файл содержит оценки студентов в сессию, собранные в компонент типа вектор. Дозапись в файл в этом случае осуществляется в конец файла (рис. 4.10).

```

===== Запись в файл нового элемента =====
Type
  Mark = 2..5;           { Диапазон оценок, получаемых студентами }
  Vector = array [1..5] of Mark; { Новый тип массива – Vector, }
                                { содержит 5 оценок в сессию }
  Marks = File Of Vector; { Тип файла с компонентом типа вектор }

Var
  F : Marks;             { Объявляем файловую переменную f типа File of ... }
  V : Vector;            { Объявляем вспомогательную переменную V такого же }
                          { типа, как компонент файла, для работы с файлом }
  I : Byte;              { Параметр цикла }

Begin
  Assign (f, 'Session.dat'); { Связываем формальный файл f с }
                              { фактическим, имеющим имя }
                              { Session . Dat }
  Reset (f);                { Открываем файл для чтения !!! }
                              { Указатель встает в начало файла }
  Write ('Введите оценки за экзамены' );
  For i:=1 to 5 do
    Readln (V[i]);          { Вводим 5 оценок каждого студента в }
                              { массив V }
  Seek (f, FileSize( f )); { Ставим указатель в конец файла }
  Write (f, V);             { Дописываем в файл f новый компонент }
                              { типа Vector }
  Close (f);                { Закрываем файл }

End.

```

Рис. 4.10. Дозапись нового компонента

При необходимости записать новый компонент файла не в его конец, а в середину можно так же, как в примере на рис. 4.10, воспользоваться процедурой позиционирования (рис. 4.11). В этом случае компонент, на место которого производится запись, затирается.



Необходимо помнить, что при записи компонента в середину файла произойдет «стирание» старого компонента и запись на его место нового («раздвижки» файла не будет).

```
{=== Запись в файл нового элемента в заданную позицию ===}

Type
  Mark = 2..5;           { Диапазон оценок, получаемых студентами }
  Vector = array [1..5] of Mark; { Новый тип массива – Vector, }
                                   { содержит 5 оценок в сессию }
  Marks = File Of Vector; { Тип файла с компонентом типа вектор }

Var
  F : Marks;           { Объявляем файловую переменную f типа File of }
  V : Vector;          { Объявляем вспомогательную переменную V такого же }
                                   { типа, как компонент файла, для работы с файлом }
  I : Byte;            { Параметр цикла }
  N : Byte;            { Номер компонента в файле }

Begin
  Assign (f, 'Session.dat'); { Связываем формальный файл f с }
                                   { фактическим, имеющим имя }
                                   { Session . Dat }
  Reset (f);              { Открываем файл для чтения !!! }
                                   { Указатель встает в начало файла }
  Writeln ('Введите оценки за экзамены' );
  For i:=1 to 5 do
    Readln (V[i]);         { Вводим 5 оценок каждого студента в }
                                   { массив V }
  Writeln ('Введите номер компонента' );
```

Рис. 4.11. Запись нового компонента в заданную позицию файла (см. с. 108)

```

Readln (n);                               { Вводим номер компонента в файле }
                                           { оценок }
If n < FileSize(f) then begin { Проверяем, есть ли компонент }
                                           { с таким номером в файле }
    Seek (f, n);                          { Ставим указатель файла перед этим }
                                           { компонентом }
    Write (f, V);                         { Дописываем в файл f новый компонент }
                                           { типа Vector }
End
Else Write ('В файле нет компонента с номером', n);
Close (f);                                { Закрываем файл }
End.

```

Рис. 4.11. Окончание

4.10.5. Процедура отсечения конца файла

Процедура

Truncate (Var f)

Truncate (Имя_Формального_Файла)

отсекает часть открытого файла *f*, начиная с компонента, стоящего за только что считанным, и «подтягивает» на это место конец файла.

Конструкция

Seek (f, 0);

Truncate (f);

поставит указатель файла перед первым (с номером 0) компонентом, отсечет (удалит) все следующие компоненты и «подтянет» на эту позицию конец файла. Функция проверки конца файла *EOF* выдаст значение *True*.

Конструкция

Seek (f, FileSize(f));

Truncate (f);

фактически ничего не изменит в файле, т.к. указатель файла стоит после последнего компонента и отсекал в файле уже нечего.

После выполнения процедуры отсечения восстановить удаленные компоненты невозможно, кроме как создать их заново.



Процедура *Truncate* не работает с текстовыми файлами.

4.11. Нетипизированные файлы

Нетипизированные или бестиповые файлы – особый обобщенный тип файлов в Турбо-Паскале. В отличие от типизированных файлов с явно заданным типом компонента, бестиповые файлы описываются просто *File*, предполагая наличие в файле разнородных (разнотипных) данных:

Var

f : File;

Такой способ хранения данных позволяет, не задумываясь об их типе, записывать и считывать данные целыми блоками, преобразовывать считанные данные одного типа в другой. Такая вольность работы с данными накладывает жесткие ограничения на программиста, связанные с четкостью преобразования типов данных.

Для нетипизированных файлов приемлемы все общие процедуры работы с файлами (см. табл. 4.1). Кроме того, к нетипизированным файлам могут быть применены процедуры и функции работы с компонентными файлами (см. табл. 4.3).

Как и при работе с текстовым или компонентным файлом, с нетипизированным файлом, как правило, выполняют четыре действия:

- 1) Связывание формального и фактического файлов.
- 2) Открытие файла (для записи или для чтения).
- 3) Непосредственно работа с файловыми данными (запись в файл или чтение из файла).
- 4) Закрытие файла.

4.11.1. Открытие нетипизированного файла

Процедуры открытия нетипизированного файла имеют расширенный синтаксис.

Процедура открытия

Reset (Var f : File; BufSize : Word)

Reset (Имя_Формального_Файла , Размер_Буфера_Обмена)

открывает формальный файл *f* для чтения и устанавливает для него размер буфера обмена равным *BufSize*.

Аналогично процедура открытия

Rewrite (Var f : File; BufSize : Word)

Rewrite (Имя_Формального_Файла , Размер_Буфера_Обмена)

открывает формальный файл *f* для записи, устанавливает для него размер буфера обмена равным *BufSize*.

Так как данные из нетипизированного файла считываются блоками, то при открытии файла размер буфера автоматически (по умолчанию) устанавливается равным 128 байт. Однако в программе можно установить любой другой размер буфера, указав его явно в процедуре открытия файла. Минимальный размер буфера равен 1 байт, максимальный - 64К.

Параметр *BufSize* задает число байт памяти, считываемое или записываемое за одно обращение к файлу. Чем больше размер буфера, тем быстрее происходит обмен данными между диском и оперативной памятью компьютера. Но тем больше и расход памяти.

4.11.2. Чтение и запись в нетипизированном файле

Ввод и вывод в нетипизированных файлах осуществляется с помощью специальных процедур **BlockRead** и **BlockWrite**. Для чтения и записи стандартные процедуры *Read* и *Write* не пригодны. Процедуры ввода и вывода данных в нетипизированных файлах оперируют не с компонентом или строкой, как в других типах файлов, а с блоком, равным размеру буфера, определенным процедурой открытия файла.

Процедура

BlockRead (**Var** *f* : **File**;

Var *BufVar* ;

Number : **Word**

[; **Var** *CountBlock* : **Word**])

BlockRead (*Имя_Формального_Файла* ,

Буферная_Переменная_Для_Хранения_Блока,

Количество_Блоков_Для_Считывания,

Количество_Считанных_Блоков)

считывает из нетипизированного файла *f* в переменную *BufVar* блок или несколько блоков. Количество блоков (буферов), считываемых за один прием, определяется переменной *Number*. При этом необходимо соблюдение условия, чтобы $\text{Number} * \text{BufSize} < 64\text{K}$.

Необязательная переменная *CountBlock* содержит информацию о том, сколько блоков из числа определенных к считыванию, уже считалось текущей операцией *BlockRead*.

Процедура

BlockWrite (**Var** *f* : **File**;

Var *BufVar* ;

Number : **Word**

[; **Var** *CountBlock* : **Word**])

BlockWrite (*Имя_Формального_Файла* ,

Буферная_Переменная_Для_Хранения_Блока,

Количество_Блоков_Для_Записи,

Количество_Записанных_Блоков)

записывает в нетипизированный файл *f* из переменной *BufVar* блок или несколько блоков.

Необязательная переменная *CountBlock* содержит информацию о том, сколько блоков из числа определенных к записи уже записано в файл текущей операцией *BlockWrite*.

Если операция записи или чтения прошла успешно, то значение переменной *CountBlock* будет равно значению переменной *Number*. В случае сбой во время выполнения операции записи или чтения в переменной *CountBlock* будет находиться число полностью считанных блоков.

В примере, приведенном ниже, показано, как можно проконтролировать чтение и запись в нетипизированном файле.

```

Var
  F_in , F_out:File ;           { Файловые переменные }
  Count_In, Count_Out : Word;   { Переменные контроля записи и }
                                { чтения }
  Buf_In : . . . . . ;         { Буферная переменная для ввода }
                                { блока }
  Buf_Out : . . . . . ;        { Буферная переменная для вывода }
                                { блока }
  V:array[1..5] of Integer;     { Массив из 5 элементов }

Begin
  Assign (F_in, 'Source.dat');   {Связываем формальный файл F_in}
                                { с фактическим, имеющим имя }
                                { Source . Dat }
  Assign (F_out, 'Result.dat');  {Связываем формальный файл }
                                { F_out с фактическим, имеющим }
                                { имя Result . Dat }
  Reset(F_in,                    { Открываем файл F_in для чтения }
    SizeOf (V));                 { Устанавливаем размер буфера равным }
                                { размеру массива V }
  Rewrite(F_out,                 { Открываем файл F_out для записи }
    SizeOf (V));                 { Устанавливаем размер буфера равным }
                                { размеру массива V }
  BlockRead(F_in, Buf_In, 3, Count_In);

```

```
If Count_In <> 3 then
    Write('При чтении из файла произошел сбой');
BlockWrite(F_out, Buf_Out, 2, Count_Out );
If Count_Out <> 2 then
    Write('При записи в файл произошел сбой');

Close (F_in);                { Закрываем файлы }
Close (F_out);

End.
```



Если в процедуре чтения не указать необязательный параметр *CountBlock*, то невозможность считать заданное число блоков вызовет ошибку ввода-вывода.

4.12. Примеры программ

```

{===== функция проверки существования файла=====}

Function FileExists (FileName : String) : Boolean;
  Var f : file;                                { Файловая переменная }
                                              { Тип ее не имеет значения }
Begin
  Assign (f, FileName);                       {Связываем формальный файл F }
                                              { фактическим, имя которого передается }
                                              { в процедуру через параметры }
  {$I-}                                       { Отключаем контроль ввода }
  reset (f);                                 { Открываем файл F для чтения }
  {$I+}                                       { Включаем контроль ввода }
  If IOResult = 0 then begin { Проверяем не было ли ошибок при }
    { открытии файла }
    Close (f);                               { Закрываем файл }
    FileExists := True;                      { Функция возвращает значения True }
    { Файл с таким именем существует }
  End
  Else FileExists := False; { Функция возвращает значения False }
    { Файла с таким именем не существует }
End {FileExists};

```

Рис. 4.12. Функция проверки существования файла

В примере, приведенном ниже (см. рис.4.13), показано, как можно использовать функцию проверки существования файла. Файл при этом открывается для чтения, и проверка осуществляется для файла, расположенного в текущем каталоге. Возможно использование этой же процедуры для файла, открываемого для записи. В этом случае пользователь при условии существования файла с указанным именем выбирает открывать этот файл для записи и тем самым «затирать» его содержимое или же повторить ввод нового имени файла.

```

{ Использование функции проверки существования файла }

Var f : file;           { Файловая переменная }
                        { Тип ее не имеет значения }
    FileName : String;  { Имя файла }

Begin

    Write('Введите имя файла: ');

    ReadLn (FileName);   { Вводим имя файла }

    If FileExists(FileName) then begin { Проверяем, существует ли файл с именем FileName }
        Assign (f, FileName); { Связываем формальный файл с фактическим }
        Reset(f);             { Открываем файл для чтения }

        { Работа с файлом }

        Close (f);           { Закрываем файл }

    End

    Else

        Write('Файл с именем ', FileName, ' не существует');

End.

```

Рис. 4.13. Использование функции проверки существования файла

```

{===== Общий цикл работы с компонентным файлом =====}

.
.
.

Write('Введите имя файла: ');

ReadLn (FileName);   { Вводим имя файла }

If FileExists(FileName) then begin { Проверяем, существует ли файл с именем FileName }
    Assign (f, FileName); { Связываем формальный файл с фактическим }

```

Рис. 4.14. Общий цикл работы с компонентным файлом (см. с. 116)

```

Reset (f);           { Открываем файл для чтения }
n := 0;              { Номер текущей записи }

Repeat
    Seek (f, n);      { Устанавливаем указатель файла на }
                     { компонент с номером n }
    Read (f, v);      { Читаем из файла f один компонент в }
                     { переменную V }
    View (v);         { Процедура вывода текущего компонента }
                     { на экран }
    ch := ReadKey;     { Опрос клавиатуры }

    If ch = #0 then begin { Переход в режим расширенного опроса }
                     { клавиатуры }
        ch := ReadKey; { Повторный опрос клавиатуры }

        Case ch of
            { Выбор управляющей клавиши }

{←}      #75 : If n > 0 then dec(n);
{→}      #77 : If n < FileSize (f)-1 then inc(n);
{Home}   #71 : n := 0;
{End}    #79 : n := FileSize (f)-1;
{Ins}    #82 : ADD;      { Процедура добавления одного компонента }
{Del}    #83 : DEL(n);   { Процедура удаления n-го компонента }
{F4}     #62 : EDIT(n);  { Процедура редактирования n-го компонента}

        End;           { End Case }

    End;              { End If }

Until ch = #27;      { Цикл, пока не нажата клавиша Esc }

Close (f);           { Закрываем файл }

End

Else

    Write('Файл с именем', FileName, 'не существует');
.
.
.

```

Рис. 4.14. Окончание

Пример на рис. 4.14 содержит общий цикл работы с компонентным файлом. Будем полагать, что файл предварительно создан и необходимо выполнить некоторые действия, такие как:

- просмотр содержимого файл;
- добавление нового компонента в файл;
- удаление и редактирование компонента с заданным номером.

Процедура просмотра содержимого файла приведена на рис. 4.15. Для просмотра выбрана задача, в которой рассматривается файл, содержащий информацию о книгах, хранящихся в библиотеке: инвентарный номер книги, автор, название и год выпуска.

```

Type                                     {Предполагаемый тип компонента файла: }
Str30 = String [30];                   { информация о книге из библиотеки }

Tbook = record                          { Тип компонента файла - запись }
  Number : Word;                       { Номер книги в библиотеке }
  Author,                              { Автор книги }
  BookName : Str30;                    { Название книги }
  Year : Word;                         { Год выпуска книги }
End;

{===== Процедура показа одного компонента файла =====}

Procedure View (Book : Tbook);

Begin
  Writeln('Номер: ', Book.Number);
  Writeln('Автор: ', Book.Author);
  Writeln('Название: ', Book.BookName);
  Writeln('Год выпуска: ', Book.Year);
End {View};

```

Рис. 4.15. Процедура показа одного компонента файла

```

{===== Процедура удаления компонента из файла =====}
Procedure Del (n : Word);      { n – номер компонента файла }
  Var Book : Tbook;           { Переменная типа компонента файла }
Begin
  Seek(f, FileSize(f)-1);     { Устанавливаем указатель файла
                               { перед последним компонентом }
  Read(f, Book);              { Читаем последний компонент файла }
  Seek(f, n);                 { Устанавливаем указатель файла
                               { на компонент, который надо удалить }
  Write(f, Book);             { Пишем на это место последний
                               { компонент }
  Seek(f, FileSize(f)-1);     { Устанавливаем указатель файла
                               { перед последним компонентом }
  Truncate(f);               { Отсекаем часть файла с позиции
                               { последнего компонента }
End {Del};                   { Конец процедуры удаления }

```

Рис. 4.16. Процедура удаления компонента из файла

Один из алгоритмов удаления компонента из неупорядоченного файла приведен на рис. 4.16. Суть его заключается в следующем: читается последний компонент файла и записывается на место удаляемого компонента, затем последний компонент отсекается. Открытие и закрытие файла в процедуре не производится, так как данная процедура рассматривается в контексте общего цикла работы с файлом.

Редактирование записи рассматривается с позиции задачи о книгах из библиотеки (рис. 4.17).

Добавление нового элемента в конец файла или в заданную позицию показано на рис. 4.10 и 4.11 соответственно.

```

{===== Процедура редактирования компонента файла =====}
Procedure Edit (n : Word);    { n – номер компонента файла }
  Var Book : Tbook;          { Переменная типа компонента файла }
Begin

```

Рис. 4.17. Процедура редактирования компонента из файла (см. с. 119)

```

Seek(f, n);           { Устанавливаем указатель файла }
                      { на компонент, который надо изменить }
Read(f, Book);        { Читаем n – ый компонент файла }
Write('Номер');       ReadLn(Book.Number);
Write('Автор');       ReadLn(Book.Author);
Write('Название');    ReadLn(Book.BookName);
Write('Год выпуска'); ReadLn(Book.Year);
End {Edit};           { Конец процедуры редактирования }

```

Рис. 4.17. Окончание

На рис. 4.18 приведена процедура создания файла с запросом пользователю на подтверждение ввода очередного компонента.

```

{===== Процедура создания файла =====}
Procedure CreateFile (Var f : File);
                      { Тип файловой переменной не имеет значения }
  Var V : . . .      { Переменная V – типа компонент файла }
      Ch : Char;
Begin
  Rewrite (f);        { Открываем файл F для записи }
  Repeat
    { Ввод компонента файла }
    Write('Для окончания ввода нажмите Esc');
    Ch := ReadKey;
  Until Ch = #27;      { Цикл, пока не нажата клавиша Esc }
  Close(f);
End {CreateFile};

```

Рис. 4.18. Процедура создания файла

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Джадд Д.Р. Работа с файлами / Пер. с англ. - М.: Мир, 1975.
2. Грогоно П. Программирование на языке Паскаль / Пер. с англ. - М.: Мир, 1982.
3. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль / М.: МАИ, 1992.
4. Эрбс Х.-Э., Штольц О. Введение в программирование на языке Паскаль / Пер. с нем. - М.: Мир, 1989.
5. Вирт Н. Алгоритмы + структуры данных = программы / М.: Мир, 1985.
6. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя / Пер. с англ. - М.: Финансы и статистика, 1989.
7. Бутомо И.Д., Самочадин А.В., Усанова Д.Б. Программирование на алгоритмическом языке Паскаль для микроЭВМ / Л.: Издательство Ленинградского университета, 1985.

ПЕРЕЧЕНЬ ИСПОЛЗУЕМЫХ ОБОЗНАЧЕНИЙ



Обратите внимание! Полезная информация.



Это может привести к ошибке. Так лучше не делать.



Неправильное использование конструкций языка.

Может привести к ошибке.

Учебное издание

*Зеленко Лариса Сергеевна
Михеева Татьяна Ивановна*

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ПРОГРАММИРОВАНИЕ

Курс лекций для студентов заочной формы обучения

Редактор Т. К. Крeтeнннa

Корректор Н. С. Куприянова

Лицензия ЛР № 020301 от 30.12.96г.

Подписано в печать 10.09.2001 г. Формат 60×84 1/16.

Бумага газетная. Печать офсетная.

Усл. п.л. 7,20. Усл. кр.-отт. 7,32. Уч.-изд. л. 7,75.

Тираж 100 экз. Заказ 132.

Самарский государственный аэрокосмический университет имени
академика С.П.Королева.

443086 Самара, Московское шоссе, 34

ИПО Самарского государственного аэрокосмического университета.

443001 Самара, ул. Молодогвардейская, 151