21. Файлы.

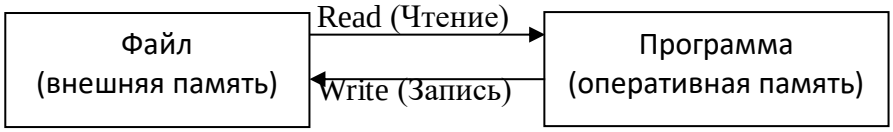
21.1 Основные определения

Файл – **поименованная совокупность данных на внешнем носителе**. От других структурированных стандартных типов данных файл отличается тем, что **число компонентов файла при его создании не фиксируется** (и ничем не ограничен сверху)**.**

Над файлами обычно выполняются операции: открытие файла, чтение из файла (*ввод*), запись (или дозапись) в файл (*вывод*), позиционирование (на нужную компоненту), закрытие файла.

Чтение (ввод) подразумевает считывание единицы информации (компонента файла) в оперативную память (в область данных программы).

Операция **записи** (вывод) подразумевает, что единица информации (блок, запись и т.п.) пересылается из оперативной памяти в файл.



Различаются логический и физический файл.

***Физический* файл** находится на внешнем носителе независимо от программиста и его программы. Такой файл иногда называется набором данных. Физические файлы имеют на внешнем носителе свои имена. Имена записываются в виде имя+расширение. Имя физического файла в программе (или полный путь к файлу) должно быть в тексте программы заключено в апострофы (т.е. оформление в виде строковой константы ’имя.расширение’).

***Логические* файлы** существуют только в программах. Логический файл это представление (модель) физического файла в программе. Это представление в программе осуществляется (реализуется) с использованием специальных файловых переменных. Файловые переменные, как и все переменные, перед использованием должны быть выше по тексту объявлены в программе (в секции описания переменных), причем каждому физическому файлу в программе должна соответствовать файловая переменная определенного типа. Перед началом работы с файлом надо установить **соответствие между физическим и логическим файлом**.

Паскаль:

Assign(f , ’имя физического файла’);

Delphi:

AssignFile(f , ’имя физического файла’);

После выполнения этой процедуры, все действия в программе над файловой переменной (ввод или вывод) будут **на самом деле выполняться над ее синонимом** - физическим файлом:

файловая переменная

Assign(f ,…);
Read (f ,…);
Write (output ,…);

Стандартные логические файлы *Input* и *Output* при (их имена, как известно, можно не указывать в процедурах ввода и вывода – их имена там подразумеваются по умолчанию) открытии по умолчанию устанавливаются на устройство *CON* (устройство консоли, для которого вывод осуществляется на экран дисплея, а ввод с клавиатуры), то есть автоматически выполняются следующее связывание:

Assign(Input,’Con’);
или
AssignFile(Input,’Con’); - в дельфи

Для файлов нужно различать **две характеристики**: способ организации и метод доступа.

Способ организации определяет порядок размещения на внешнем носителе компонент (записей) файла друг относительно друга. По способу организации различают файлы с последовательной и прямой организацией.

При ***последовательной*** организации файловые записи располагаются в таком порядке, в котором их собираются обрабатывать. Как говорят, в таких файлах физическое следование элементов на внешнем носителе полностью совпадает с логическим порядком обработки этих элементов. В файле с такой организацией элемент, который должен обрабатываться по порядку 3-им, на внешнем носителе должен располагаться также третьим от начала файла.

При ***прямой*** организации даже обрабатываемые подряд записи файла могут храниться в произвольном порядке на внешнем носителе. Например, запись, обращение к которой должно произойти в последнюю очередь, может храниться на внешнем носителе в любом месте относительно начала файла – первой, последней, и т.д. При этом каждая запись должна снабжаться специальным *признаком* или *ключом,* который должен позволить прямо найти нужную запись (или номер нужной записи должен легко вычисляться).

Примечание: все файлы в Паскале – файлы с последовательной организацией.

2-я характеристика - ***метод доступа*** - характеризует *порядок обработки записей файла*. Различают *последовательный* и *прямой* метод доступа.

Файлы с ***последовательным методом доступа*** могут обрабатываться только в одном направлении – от первой компоненты к последней или от последней к первой (строго в порядке расположения записей). Естественно, что за один сеанс (цикл) обработки такого файла (за один проход в одном направлении) каждый элемент может быть обработан лишь один раз. Повторно обработать элемент можно лишь путем повторного (опять сначала) просмотра файла. При этом номер записи, обрабатываемой при текущем обращении, не указывается (подразумевается запись, непосредственно следующая за последней обработанной записью).

у них последовательная организация и последовательный метод доступа

Типичный **шаблон обработки текстовых файлов** имеет следующий вид:

возвращает TRUE, если достигнут коней файла *f*

Пока не конец файла **вып**
нач.
считать/записать текущую запись (строку)
обработать очередную запись (строку)
кон
	while not EOF(f) do
	begin
	Readln(f, имя_переменной-строки);
	обработать очередную запись (строку)
	end;

Для файлов с ***прямым методом доступа*** возможен непосредственный (прямой) доступ к записи по ее номеру. В таких файлах возможна обработка записей в любом порядке и в любом направлении. В Паскале файлы с прямым доступом создаются из записей одинаковой *длины*. Именно это позволяет компилятору, зная номер записи, определить (перед обращением к этой записи) ее смещение относительно начала файла.

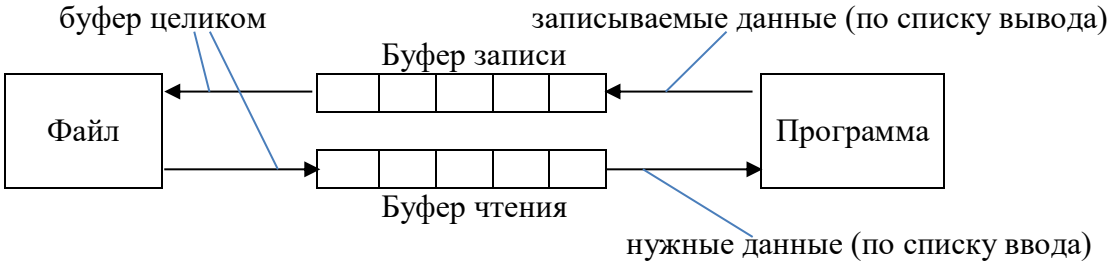
Типичный **шаблон обработки типизированных файлов** имеет следующий вид:

компилятор делает

Пока не конец файла **вып**
нач
вычислить адрес записи по ее номеру
встать на нужную запись
считать/записать запись
обработать запись
кон
	while not EOF(f) do
	begin
	Адрес = (номер-1) * (размер записи)
	seek(f, номер);
	Read(f, имя_переменной);
	обработать считанную в память запись
	end;

Ввод-вывод в Паскале ***буферизирован***. Память под буфер файла выделяется в момент т.н. открытия файла в системной области памяти (не программы). При *вводе* информация из физического файла считывается в буфер в количестве, определяемом размером буфера. При этом только первое чтение происходит непосредственно их физического файла. Второе и последующее чтение до исчерпания буфера будут происходить не из файла, а из этого самого буфера. Когда буфер будет исчерпан, то в него снова закачивается информация и т.д. При *выводе* выводимая информация записывается не сразу в физический файл, а сначала в буфер до его заполнения. Как только буфер

заполнится, то все его содержимое записывается (в рамках одной операции записи) в физический файл и т.д.



NB: - очистка буфера записи - flush(f);

- очистка буфера чтения - while keypressed do readkey;

Наличие буферов в/в позволяет **оптимизировать** работу жестких дисков.

Для разных типов файлов **размер буфера** выбирается по-разному. Для текстовых файлов размер буфера принимается равным *128* байт. С точки зрения оптимизации времени обращения к диску эта величина слишком мала, т.к. для сокращения числа физических обращений к диску целесообразно увеличение объема размера буфера ввода-вывода. Лучше всего, если размер буфера кратен размеру сектора на жестком диске (самое лучшее, если размер буфера равен размеру кластера). Процедура

SetTextBuf(Var f:text;Var Buf [;size:word]); дает программисту возможность назначить свой буфер ввода-вывода *Buf* необходимого объема *Size* текстовому файлу *f*.

Для типизированных файлов размер буфера принимается равным размеру компонент. И нет никаких средств для его коррекции.

Размер буфера для нетипизированных файлов принимается равным выбранному при открытии размеру компоненты файла.

21.2 Типы файлов в TP.

Замечание: содержимое одной файловой переменной нельзя присваивать другой (массивы и записи - можно).

Замечание: в случае необходимости передачи файловой переменной как параметра в процедуру или функцию, можно передавать ее только по адресу (как формальный параметр-переменную, т.е. со словом *Var*). При этом для обеспечения совместимости фактических и формальных параметров необходимо тип формального параметра объявлять не в заголовке подпрограммы, а выше по тексту в секции Type.

Для всех типов файлов **надо различать** следующее

- описание файловой переменной в программе;
- что является компонентами файла;
- как определяется конец файла;
- можно или нельзя использовать в программе текущий указатель при работе с файлом;
- способ организации файла (в Паскале всегда последовательный);
- метод доступа к компонентам файла;
- как определяется размер и положение буфера файла;
- внутреннее представление компонентов файла.

Рассмотрим, как по этим характеристикам разные типы файлов отличаются друг от друга.

1) Текстовые файлы

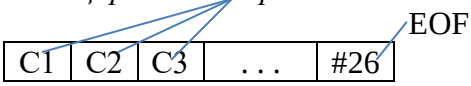
Объявление файла:

```
Var
  F : text;
или
  F : TextFile; {в Delphi}
```

файловая переменная

Текстовый файл можно рассматривать как последовательность символов, разбитую на строки переменной длины. Каждая такая строка завершается маркером конца строки, состоящим из двух символов: символа CR с кодом #13 (возврат каретки) и символа. LF с кодом #10 (перевод строки). Строки файла обычно имеют разный размер. К строкам файла возможен только последовательный доступ.

В конце каждого файла помещается символ конца файла (символ с номером 26). #26 (#$1A) – *конец файла: строки*



При каждой операции ввода/вывода явно номер записи (строки) не указывается, но подразумевается, что каждый раз действие выполняется над текущей строкой. При открытии файла текущей строкой является 1-я строка. После обработки 1-ой строки текущей станет 2-ая, после обработки 2-ой строки текущей станет 3-я и т.д.

Типичный **шаблон обработки текстовых файлов** имеет следующий вид:

возвращает TRUE, если достигнут коней файла

Пока не конец файла **вып**
нач.
считать/записать текущую запись
обработать очередную строку
кон
	while not EOF(f) do
	begin
	Readln(f, имя_переменной-строки);
	обработать очередную строку
	end;

Понятие текущего указателя к таким файлам неприменимо.

2) Типизированные файлы или компонентные файлы.

Содержимое типизированных файлов рассматривается как последовательность компонент одного и того же определенного типа (и естественно одинакового размера). На Паскале такие файлы описываются следующим образом:

Var
F: file of тип компонент;

Пример:
F: file of char;
F: file of string[10];

Особенности типизированных файлов следующие:

1) тип и размер всех компонентов файла одинаков.

2) возможен как прямой, так и последовательный доступ к компонентам (это вытекает из 1-ой особенности).

3) в отличии от текстовых файлов в конце типизированных файлов не размещается символ конца файла.

4) с каждым таким файлом связывают **текущий указатель** (указатель на текущий компонент файла), которым можно управлять для позиционирования по файлу. Компоненты типизированных файлов нумеруются начиная с нуля. Вначале (после открытия) текущий указатель размещается перед началом первой записи (она имеет номер 0). При этом текущий указатель равен нулю. После каждой операции ввода или вывода текущий указатель увеличивается на 1, т.е. **перемещается на следующий компонент**.

Замечание: для текстовых файлов было понятие текущей записи, но не было понятия текущего указателя.

С текущем указателем вы можете выполнять следующие действия:

1) Прочитать значение текущего указателя;
i := filepos(f); - прочитать текущий указатель. В точку вызова возвращается значение текущего указателя (номер текущей компоненты) в компонентах (записях).
i := filesize(f); - размер файла в компонентах
2) Установить его значение (принудительно):

Seek (f, i), - установить текущий указатель перед записью i, где i- номер нужной записи начиная с «0»)

Seek(f, filesize(f)); - встать за последней записью в файле

Т.о. с использованием текущего указателя можно обрабатывать компоненты в произвольном порядке, указывая (перед обращениями к записям файла) номера компонентов.

Замечание: **внутреннее представление** одних и тех же строк текста совершенно разное для текстовых файлов и типизированных. Рассмотрим пример:

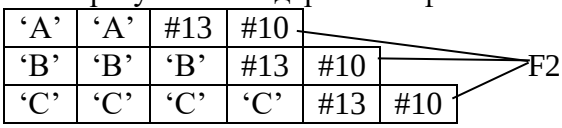
```
Var
  F1 : file of string[5];
  F2 : text;
```

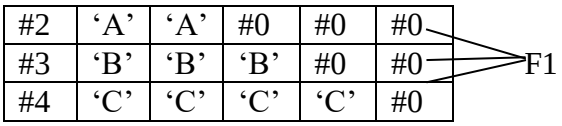
Пусть следующие три строки нужно расположить в каждом из двух файлах (F1 и F2):

'AA'
'BBB'
'CCCC'

типизированный
текстовый

В результате содержимое файлов F1и F2 будет следующим:





Из рисунка видно, что

- в типизированном файле хранится внутреннее представление строк (так же как в оперативной памяти);
- в типизированном файле в конце каждой строки значащие символы дополняются нуль – символами до размера компонента файла.

NB!!!: Эту разницу необходимо учитывать при посимвольной обработке текстовых файлов (когда он создавался как текстовый, например, а обрабатывается как типизированный файл из символов).

3) Нетипизированные файлы

Соответствующая файловая переменная описывается следующим образом:

```
Var
  F1 : file;
```

Эти файлы обладают следующими **особенностями**:

1) в отличии от двух ранее рассмотренных типов файлов, нетипизированный файл представляет из себя единую **неделимую** последовательность байтов. Эти файлы физически (явно) на компоненты не делятся. Деление этих файлов на компоненты производится на логическом уровне (мысленно) и при открытии файлов, где явно указывается размер компонента, на который мы хотим поделить файл. При каждом новом открытии файла можно делить файл на компоненты разных размеров (в разное время).
2) после того, как поделили файл на компоненты одинакового размера, для этого файла становится определенным понятие текущего указателя и можно этот файл обрабатывать как в режиме последовательного, так и в режиме прямого доступа
3) раз эти файлы не делятся явно на компоненты (и поэтому тип компоненты неизвестен), то в процедурах ввода-вывода для такого файла списки ввода и списки вывода отсутствуют. Поэтому используются специальные процедуры ввода и вывода (blockread, blockwrite).
4) любой файл, подготовленный (сохраненный) как текстовый или типизированный, можно открыть и работать с ним, как с нетипизированным. При этом в качестве буфера-приёмника/передатчика полученной информации из файла обычно является массив символов. При этом в процессе обработки такого файла не будет теряться время на преобразование типов и поиск управляющих последовательностей (типа конца строк),

достаточно лишь считать нужный объем информации их файла в определенную область памяти. За счет этого достигается максимально возможная скорость обработки файла.

21.3 Порядок работы с файлами

Общий порядок работы с файлами обычно имеет следующий вид:

0) В разделе описаний выполняется описание файловой переменной и буферной переменной (куда читать из файла или откуда писать в файл).
1) Связывание логического файла с физическим.
2) Открытие файла (для чтения файла, записи, дозаписи).
3) Позиционирование (установка на компоненту файла, с которой нужно начать обработку файла).
4) Обработка компонентов файла (обычно в цикле до нахождения конца файла).
5) Закрытие файла.

Замечания:

а) указанные действия с номерами 3 и 4 обычно выполняются в цикле (до конца файла).

б) при открытии файла обычно производится настройка на первую его компоненту файла и производится начальное заполнение буфера.

в) при закрытии файла происходит:

1) сбрасывание содержимого буфера записи в физический файл;

2) уничтожение (освобождение занимаемой памяти) внутренних буферов, связанных с файлом;

г) буфера, через которые производятся операции в/в, находятся в системной области (не в программе).

21.4 Связывание логического файла и физического.

становятся синонимами

```
Var
    f : file;
    . . .
    assign(файловая переменная, 'путь к физическому файлу');
```

```
Var
    f : text;
    Assign(f, ’con’); где con – консоль (при вводе – клавиатура; выводе - монитор);
```

Особенности этой процедуры:

1) Здесь не проверяется как был записан путь к физическому файлу (не проверяется, есть ли физический файл или он отсутствует). Такая проверка выполняется позже уже **при открытии файла**.

2) Процедура assign работает только с закрытыми файлами.

3) Связывание сохраняется и после закрытия файла (развязывание не предусмотрено).

21.5 Открытие файлов.

Замечание: Сразу после открытия текущий указатель (если он определен для данного типа файлов) устанавливается на 1-ю компоненту.

На Паскале файлы открываются отдельно (по-разному) для чтения и записи.

Для текстовых и типизированных файлов открытие производится с помощью следующих двух процедур:

Reset(f); - открыть существующий файл для чтения

Rewrite(f); - открыть файл для записи.

При выполнении этих процедур проверяется, существует или нет соответствующий физический файл. Если при открытии для чтения физического файла нет, то программа аварийно завершается с выводом сообщения об ошибке. Если при открытии для записи физического файла не существует, то он будет перед открытием создан (пустой). Если, наоборот, при открытии для записи файл уже есь, то он будет открыт для записи в самое начало - будет затираться).

Замечания:

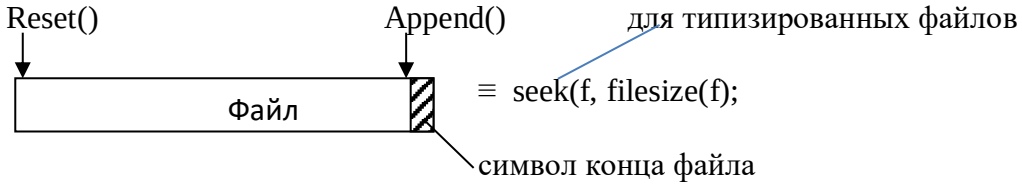
1) Для типизированных и нетипизированных файлов, открытых с помощью процедуры reset, могут быть использованы и операции записи, причем без переоткрытия файла. Для текстовых файлов возможно только одно направление операции (чтение или запись). Если в текстовом файле нужно поменять направление операции (например в файл, открытый для чтения, надо выполнить запись), то нужно файл вначале закрыть для чтения и затем открыть для записи. Для типизированного файла режим обращения к файлу (Чтение или Запись) можно установить (перед открытием файла) с помощью значения переменной Filemode из модуля System (0 – только чтение, 1 – только запись, 2 – чтение и запись), которая по умолчанию для типизированных и нетипизированных файлов устанавливается равной 2.
2) если открываемый файл ранее уже был открыт, то он вначале автоматически закроется, а потом заново откроется в том режиме, который был последний раз указан.

Для нетипизированных файлов существует две разновидности процедур открытия. Первая совпадает с той, что была приведена для текстовых и типизированных файлов (при этом размер буфера устанавливается равным 128 байтам). Вторая разновидность является расширенным вариантом первой:

Reset(f, размер компонент);
Rewrite(f, размер компонент);

Замечание: один и тот же нетипизированный файл можно открывать в одной программе в разное время с разными размерами компонента.

Для текстовых файлов имеется еще одна процедура открытия, которая открывает файл для записи, но для записи только начиная с конца файла:



После такого открытия текущая операция записи в файл будет затирать конец файла.

21.6 Позиционирование в пределах файлов.

7

1) **для типизированного и нетипизированного файлов**?

Позиционирование выполняется с использованием **текущего указателя** файла. Отметим вначале, что для работы с текущим указателем могут потребоваться следующие функции

I := filepos(f); --- возвращается позиция текущего указателя (номер текущей компоненты)
I := filesize(f); --- возвращается размер файла в компонентах

Собственно позиционирование выполняется с помощью процедуры Seek(f, i);

Seek(f, 0);	в начало файла ≡ reset(f);
Seek(f, filesize(f));	в конец за последней записью
Seek(f, 9);	в середину перед 10-й записью (нумерация с нуля, поэтому 10-я запись имеет номер 9)
Seek(f, filepos(f) - 2);	на две записи назад относительно текущей

С текущим указателе работает также процедура Truncate(f) – устанавливает в текущей позиции признак конца файла. С помощью Truncate(f) производится отсечение хвоста файла (с позиции текущего указателя) после закрытия файла. Обычно Truncate(f) используется в паре с seek: вначале вызывается seek(), а потом truncate():

номер компоненты, начиная с которой надо отсечь хвост (до конца файла)

```
seek(f, n);
truncate(f);
close(f);
```

отсечено будет даже в режиме чтения (filemode = 0)

2) **для текстовых файлов.**

Позиционирование в начало файла производится с помощью переоткрытия файла:

Reset(f);
Rewrite(f).

озиционирование в конец файла (только для записи в файл):
Append(f).

на строку номер *n*

Позиционирование в середину файла (мы знаем номер записи *n*, на которую надо встать).
Возможны два случая:

1) Мы знаем номер *L* записи, на которой сейчас находимся и знаем, что *L* < *n*:

```
For i := 1 to (n-L) do
  Readln(f);  ----------пустое чтение
```

2) Мы не знаем номер *L* текущей записи или знаем, что *L* > *n*. Надо вначале вернуться в начало файла, а потом пропустить несколько (n-1) записей:

```
Reset(f);
For i := to (n-1) do
  Readln(f);  ----------пустое чтение
```

21.7 Процедуры ввода/вывода.

21.7.1 Процедуры ввода/вывода для текстовых файлов.

Используются хорошо знакомые процедуры:
Readln(f[, список ввода]);
Writeln(f[, список вывода]);

Также имеются функции, которые позволяют отлавливать признаки конца строки и признаки конца файла.

EOLN(f)
EOF(f)
и другие.

Пример:
надо определить число строк в текстовом файле (использовать filesize для данного типа файла нельзя!)

```
Var
  F:text;
  S:string;
  I:longint; // счетчик строк
Begin
  Assign(f, 'text1.txt');
  Reset(f);
  I:=0; // счетчик - в ноль
```

`While not eof(f) do`
`  Begin`
`  Readln(f,s);`
`  I:=I+1;`
`  End;`
	`Repeat`
	`  Readln(f,s);`
	`  I:=I+1;`
	`Until(eof(f));`

ситуация «конец файла» здесь наступает перед попыткой считать запись за концом файла	ситуация «конец файла» здесь наступает *сразу после* считывания последней записи

Особенностью функций ***EOLN(f)*** **и** ***EOF(f)*** является то, что в отличии от случая работы с клавиатурой (когда EOLN возвращает значение истина, если последний считанный символ - признак конца строки) при работе с файлом EOLN(f) возвращает значение истина, если следующий считываемый символ в файле f будет символ конца строки. То же самое для EOF (только вместо конца строки - конец файла).

обенности есть и при работе с текстовыми файлами для процедур **Writeln и Readln:**

- в случае процедуры Readln без списка ввода при вводе с клавиатуры программа перейдет в состояние ожидания ввода до тех пор, пока вы не нажмете Ввод. В отличие от этого при вызове Readln(f) – программа не переходит в состояние ввода, а пропускает текущую строку в файле.

- в случае процедуры Writeln вызов Writeln без списка вывода при вводе с клавиатуры вызывает перевод курсора на экране на начало следующей строки (на экране), а вызов без списка вывода для файла типа Writeln(f) – записывает в файл пустую строку (символы с кодами 13 и 10).

Пример 1: копирование одного текстового файла в другой (в прямом порядке)

```
Var
  s:string; // буферная переменная
  f1:text; {файл - источник}
  f2:text; {файл - приемник}

begin
  assign(f1, 'c:\temp\f1.txt'); {входной файл}
  assign(f2, 'c:\temp\f2.txt'); {выходной файл}
  reset(f1);
  rewrite(f2);
  while (not eof(f1)) do  {пока не закончится f1}
    begin
      readln(f1,s);  {считывание строки из f1 в s}
      writeln(f2,s); {запись строки из s в f2}
  end;
  close(f1); close(f2);
  reset(f2);  {переоткрытие файла f2}
  while (not eof(f2)) do  {вывод f2 на экран}
    begin
      readln(f2,s);
      writeln('s=',s);
    end;
  readln; {приостановка после вывода очередной строки}
end.
```

видно, что по входному файлу надо двигаться в обратном направлении, а по выходному файлу – в прямом

Пример 2: копирование одного текстового файла в другой (но в обратном порядке)

```
Var
  конец_обработки: Boolean;
  i: word;   {число строк в файле}
  j: word; {вспомогательная переменная для пропуска строк в файле}
```

Алгоритм
конец_обработки:=ложь

i:=число строк в f1

используем функцию

```
function kol_str:word;
  var
    s:word; {счетчик строк}
  begin
    s:=0;
    while not eof(f1) do
      begin
        readln(f1);
        s:=s+1;
      end;
    kol_str:=s;
end;
```

while не конец_обработки do

  begin

начинаем с конца

считать i-ю строку из f1 в s

записать эту строку из s в f2

writeln(f2,s);

пропустить (i-1) строк в f1 (с начала)

считать очередную строку из f1 в s

readln(f1,s);

i:=i-1; {движемся назад}

  if i=0
    then
      конец_обработки:=истина
end;

встать на начало f1

reset(f1);

считать (i-1) строк из f1 (вхолостую)

for j:=1 to i-1 do
  readln(f1);

зная число строк в f1 можно цикл while заменить на цикл for:

```
for i:=kol_str  downto  1 do
  begin
    считать i-ю строку из f1 в s
    записать эту строку из s в f2
  end;
```

21.7.2 Процедуры ввода/вывода для типизированных файлов.

Ввод/вывод выполняется всего 2-мя процедурами:
Read(f, список ввода);
Write(f, список вывода); где f – имя файла.

Замечания:

- в списках ввода и вывода переменные могут быть только того же самого типа, что и тип компонента в типизированном файле.
- Eoln, Readln и Writeln – запрещены.
- Eof – разрешен (с ограничением для файлов из символов).

Пример (с ошибкой):

```
Var
  S : string[10];
  F : file of string[10];
Begin
  Assign(f, 'f1.txt');
  Reset(f);
  Read(f, s);
  Writeln(s);  {выводим строку s на экран}
End.
```

Здесь имеется **нарушение правил совместимости типов**: тип компонента файла (string[10]) и тип переменной (s), которая находится в списке ввода, не совпадают (а должны совпадать, т.к по сути мы передаем в процедуру Read адрес переменной *S*, по которому надо поместить в памяти значение определенного типа и размера). Последует сообщение об ошибке при компиляции.

Чтобы ошибки не было, надо писать так:

```
Type
  T = string[10];
Var
  S : t;
  F : file  of  t;
Begin
  Assign(f, 'f1.txt');
  Reset(f);
  Read(f, s);
End.
```

```
j := filesize(f1)-1;
For i := 1 to filesize(f1) do
  begin
    seek(f1, j);  {встали перед последней  строкой}
    read(f1, s);
    write(f2, s);
     j :=  j - 2;
  end;
```

Рассмотрим пример **копирования единичного типизированного файла** в другой.

Пример 3: копирование из F1 в F2 в прямом порядке

```
Type
  T = string[10];
Var
  S : t; {строка - буфер}
  F1, {файл - источник}
  F2 : file of t;  {файл - приемник}
Begin
  Assign(f1, 'f1.txt');
  Reset(f1);
  Assign(f2, 'f2.txt');
  Rewrite(f2);
  While not eof(f1) do или filepos(f1) <= filesize(f1) – 1 do
    Begin
      Read(f1, s);
      Write(f2, s);
    End;
  Close(f1);
  Close(f2);
End.
```

или

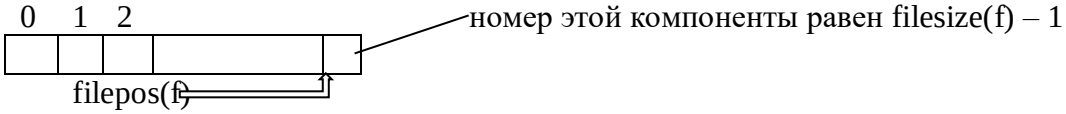
```
for i:=filesize(f1) downto 1 do
```

NB:
здесь мы и к f1 и к f2 обращаемся в режиме *последовательного* доступа, но можно использовать и *прямой* доступ

Пример 4: копирование из F1 в F2 в обратном порядке

```
Type
  t = string[10];
Var
  S : t;
  F1, F2 :  file of t;
  i: integer;
Begin
  Assign(f1, 'tex1.txt');{входной файл}
  Reset(f1);
  Assign(f2, 'text2.txt');{выходной файл}
  Rewrite(f2);
  For i := filesize(f1) downto 1 do
    begin
      seek(f1, i-1);  {встали на i-ю строку}
      read(f1, s);  write(f2, s);
  close(f1); close(f2);
end.
```

Замечание: так же как и для текстовых файлов цикл обработки типизированных файлов можно завершать с помощью функции ***eof***. Недостаток ее использования состоит в том, что эта функция eof реагирует на символ с кодом #26 (число 26 = признак конца файла). Если этот символ встречается в середине файла типа **f : file of char** (если файл обрабатывается как файл из однобайтовых чисел) и производится последовательная обработка, то ***eof*** обнаружит конец файла в середине файла по положению символа с кодом #26. Поэтому при обработке файла из символов более практично использовать не функцию eof, а функцию filepos(), которая возвращает текущий указатель файла (его надо в цикле сравнивать с размером файла).



если размер строк в файле > 255 символов, то нельзя использовать текстовые файлы

21.7.3 Процедуры ввода/вывода для нетипизированных файлов.

Нетипизированные файлы явно на компоненты не делятся. Поэтому при работе с такими файлами размер компонент и их тип неизвестны и нельзя записать список ввода и список вывода. Используются специальные процедуры ввода и вывода:

Blockread(f, buf, count, kolzap);
blockwrite(f, buf, count, kolzap);

где: входные параметры выходной параметр

1) f - имя файловой переменной.

2) buf – имя переменной (точнее ее адрес) в программе, которая используется как буфер для работы с файлом. Для чтения это область, куда информация записывается из файла за один вызов процедуры BlockRead. Для записи это область, откуда информация пишется в файл за один вызов процедуры BlockWrite. В качестве буфера обычто используется массив символов:

```
Var
 Buf: array[1..512] of char;
```

Содержимое (отдельные символы массива) никого не интересуют, этот массив рассматривается как область из 512 байт. На размер этого массива (буфера) накладывается следующее ограничение:

Recsize * count должно быть <= sizeof(buf) < 64Кб

Размер компонент файла, указанный при открытии в reset или rewrite

Reset(f, RecSize); Rewrite(f, RecSize);

3) count – заданное при вызове количество логических записей (компонент), которое считывается/записываетя в буфер из нетипизированного файла **за одну операцию ввода/вывода**.

Для того, чтобы размер файла в байтах вседа нацело делился на размер комонента, обычно выбирают размер компоненты равный 1. В этом случае значение параметра Count будет равно

1

Count = Sizeof(buf)

4) kolzap – **возвращенное** процедурой количество реально считанных (для Blockread) или реально записанных (для BlockWrite) *целых* логических записей в течении последней выполненной операции ввода-вывода в файл (из файла).

Если операция ввода/вывода заканчивается **нормально**, то значение, которое возвращается через переменную kolzap, должно совпадать с тем, что было указано в параметре count. То есть обычно после вызова

Kolzap = count.

Это равенство нарушается в следующих случаях:

1) Когда мы подошли близко к физическому концу файла (но еще не дошли до него), то Kolzap вернется в точку вызова, близким к нулю (и не равным Count). При этом число оставшися в файле записей меньше значения Count)
2) Когда при чтении/записи были физические сбои.

Рассмотрим **пример копирования одного нетипизированного файла** в другой.

```
Var
 F1, {источник}
 F2 : file;  {приемник}
 Buf : array[1..15] of char;  {буфер}
Kolzap,  {число прочитанных из f1 записей}
Kolprzap : word;  {число записанных в f2 записей}
Begin
 Assign(f1, 'f1.txt');
 Reset(f1, 1); {1 чтобы не сбиться со счета}
 Assign(f2, 'f2.txt');
 Rewrite(f2, 1); {что бы не сбиться со счета}
```

```
Repeat                                  надо бы тоже проверить sizeof(buf)  = kolzap
  Blockread(f1, buf, sizeof(buf), kolzap);
  Blockwrite(f2, buf, kolzap, kolprzap);
Until eof(f1) OR (kolzap = 0) OR (kolzap <> kolprzap);
```

В случае конца f1 — В случае ошибки ввода-вывода

число записей, которое должно было быть записано/считано, не совпадает с числом реально считанных/записанных записей

```
Close(f1); close(f2);
End.
```

Если размер логической записи (компоненты) не кратен размеру буфера, т.е.

RecSize*Count ≠ sizeof(buf)

то при последней операции ввода-вывода возникает та же **ситуация**, что и при ошибке. Поэтому, что бы не сбиться со счета размер логической записи принимают равным единице. Именно по этой причине в качестве 3-го параметра *blockread* передавался размер буфера в байтах.

В 3-ем параметре *blockwrite* указывался не размер буфера, а значение поля *kolzap* из blockread.Это сделано для того, что бы не писать в выходной файл больше, чем считано из входного.

У ситуации выхода из цикла repeat есть 2 признака:

1)обнаружен конец входного файла.

2)обнаружена ошибка ввода-вывода.

21.8 Закрытие файлов.

Выполняется при помощи вызова процедуры:

Close(f); {CloseFile(f); в Delphi}

Замечание:1 Нельзя можете закрыть более одного файла одновременно.

Замечание:2 Нельзя нормально закрыть файл, который еще не открыт. Это вызовет фатальную ошибку.

Замечание 3: Можно открыть файл, который еще не закрыт. В последнем случае (за нас) файл будет автоматически закрыт, а потом еще раз открыт.

21.9 Дополнительные функции работы с файлами

Подпрограммы ввода-вывода модуля *System*

Функция	**Описание**
Erase	Стирает физический файл (не удаляет)
Rename	Переименовывает физический файл
ChDir	Выполняет смену текущего каталога: chdir(имя); - вниз по иерархии chdir('..'); - вверх по иерархии
MkDir	Создает поддиректорий
RmDir	Удаляет поддиректорий (пустой)
GetDir	Получить текущий дир. на заданном диске
DeleteFile	Удаляет файл
Flush	Сбрасывает буфер (выходной) текстового файла

Подпрограммы ввода-вывода модуля *DOS*

Функция	**Описание**
DiskFree	Число св. байтов на диске
DiskSize	Размер диска в байтах
GetFAttr	Получение атрибутов неоткрытого файла
SetFAttr	Задание атрибутов неоткрытого файла
FSplit	Получение составных частей имени файла
FExpand	Формирование полного имени файла
FSearch	Поиск файла в списке директорий
FindFirst	Поиск в указанном или текущем директории первого файла, соответствующего заданному шаблону и атрибутам
FinfNext	Поиск следующего файла, соответствующего шаблону и атрибутам, заданным в последнем обращении к FindFirst

Пример: копирование всех **заданных шаблоном** (типа *.*) файлов (мы заранее не знаем их имена) в текущем каталоге, в один выходной файл (очевидно, что этот выходной файл не должен быть в том же каталоге)

Для решения задачи нам потребуются процедуры findfirst() и findnext(). Прототипы этих процедур имеют вид: входные параметры выходной параметр

```
FindFirst(Path: String; Attrib: Word; VAR SR: SearchRec);
FindNext(VAR SR: SearchRec);
```

Для работы с этими подпрограммами нам потребуются следующие предопределенные вещи:

1) **Константы** (атрибуты)

```
Const
  ReadOnly   = $01;   только для чтения
  Hidden     = $02;   скрытый
  SysFile    = $04;   системный (непереносимый)
  Volume ID  = $08;   метка диска
  Directory  = $10;   подкаталог
  Archive    = $20;   архивный
  AnyFile    = $3F;   сумма всех предыдущих
```

Первые шесть аттрибутов можно складывать или вычитать (вычитать надо из anyfile).

2) Переменная **DOSError**. Она используется для анализа ошибок, коды которых возвращает MS DOS. Значение этой переменной, равное нулю, соответствует отсутствию ошибки при выполнени последней операции ввода-вывода. Смысл некоторых ненулевых кодов следующий:

2 – файл не найден
3 – маршрут не найден
5 – доступ к файлу запрещен
6 – неправильная обработка
8 – недостаточно памяти
10 – неверные установки значений параметров среды
11 – неправильный формат
18 – файлов нет

При работе процедуры FindFirst возможны ошибки с номерами 2 и 18, а при работе FindNext – только 18.

3) Тип описан в модуле DOS

```
Type
  SearchRec = record
                Fill: array[1..21] of byte;   {резервное поле}
                attr: byte;                   {байт атрибутов}
                time: longint;                {время создания}
                size: longint;                {размер файла}
                name: string[12];             {имя файла}
              end;
```

Поля переменной этого типа содержат информацию о последнем файле, найденном с помощью FindFirst и FindNext.

Процедура FindFirst при заданных имени файла и атрибутах должна вызываться лишь один раз. Она записывает в поля переданной ей переменной SR типа SearchRec информацию о первом найденном файле, удовлетаоряющем заданным условиям. Эта информация в дальнейшем будет использоваться процедурой FindNext, которая всегда вызывается после FindFirst и заполняет поля переменной SR информацией о следующем найденном файле.

Контроль работы этих процедур ведется с помощью переменной DOSError: если файла нет, то DOSError<>0.

Текст нужной нам программы имеет следующий вид:

```
Uses
  crt, dos;
VAR
  fs: searchrec;
  s : string;  {буфер}
  f1, f2: text;  {вх. и вых. файлы}
begin
  clrscr;
  assign(f2, 'c:\temp\out_file.txt');  {важно, чтобы этот физ. файл находился не в текущей папке}
  rewrite(f2);
  {ищем первый файл – подготовка цикла}
  findfirst('*.*', anyfile, fs);
  {цикл, пока есть файлы}
  while doserror = 0 do
    begin
      writeln(f2);  {пропуск строки в f2}
      writeln(f2, 'Начало файла ', fs.name);
      writeln(f2);  {пропуск строки в f2}
      assign(f1, fs.name);
      reset(f1);
      {цикл до конца текущего файла}
      while not eof(f1) do
        begin
          readln(f1, s);
          writeln(f2, s);
        end;
      close(f1);
      {ищем следующий файл}
      findnext(fs);
    end;
  close(f2);
end.
```

В данном случае в теле цикла выполняется копирование всех файлов.
В общем случае в данном месте кроме проверки имени файла стоило бы поставить проверку атрибутов найденного файла типа такой:

```
if ((fs.attr and sysfile <> $00) AND (fs.attr <> Hidden))
  then обработать найденный файл
```

для отделения копируемых файлов в теле выходного файла (f2) перед началом каждого копируемого файла будем помещать некий идентификатор. В итоге структура файла f2 будет примерно следующей:

```
Пустая строка
Начало файла 1
Пустая строка
Строки файла 1
Пустая строка
Начало файла 2
Пустая строка
Строки файла 2
……..……..
```

связываем найденный физ. файл с логическим f1

открываем найденный файл для чтения

обрабатываем найденный файл (переписываем в f2)

21.10 Обработка ошибок ввода-вывода

Замечание:

Для отладки программ, содержащих обработку ошибок (особенно фатальных ошибок, которые в отличие от ошибок ввода/вывода ничем нельзя заблокировать) в Турбо Паскале имеется специальный оператор: RunError(код ошибки). При его выполнении программа завершается аварийно с выдачей сообщения об ошибке с заданным номером

Все рассмотренные ранее программы были написаны так, как будто бы возможные ошибки ввода-вывода игнорировались. Однако реально это не так - ошибки ввода-вывода возникают и надо уметь с ними бороться (или хотя бы реагировать на них). В TP стандартной реакцией на наличие ошибок в программе является следующее: программа аварийно завершается и выдает сообщение:

Runtime error <номер> <смещение>.

Такое сообщение будет полезно программистам при поиске места и причины ошибки в программе. Однако показывать такое сообщение пользователю программы будет не совсем дружественно.

Для того, чтобы программа из-за ошибки аварийно не завершалась и для того, чтобы не пугать пользователей непонятными им сообщениями, необходимо перехватить инициативу по обработке ошибок у системы и писать собственные обработчики ошибок стадии выполнения (исключений).

Традиционно обработка ошибок в программах осуществляется с помощью **установки сигнальных значений - флагов** (в случае ошибки) и последующего анализа места и причин появления этих флагов (функция IOResult, переменная DOSError).

Для перехвата ошибок ввода-вывода на Паскале необходимо:

1) Специальным образом оформить (обернуть) каждый фрагмент программы, который потенциально может вызвать ошибку. К таким фрагментам относятся - открытие файла, чтение из файла, запись в файл.

2) При наличии такого специального оформления фрагмента необходимо по завершению обернутого (специально оформленного) фрагмента проверить - была ошибка или нет. Если была, то вы должны выполнить ее обработку (хотя бы вывести свое сообщение).

Пример: Пусть в программе используется

reset(f);

тогда при попытки открытия файла физический файл может отсутствовать на диске. Следовательно, этот оператор в программе является потенциальным источником ошибки.

Специальное оформление (обертывание) потенциально опасных фрагментов программы состоит в том, чтобы заключить каждый опасный участок в специальный код, размещаемый **выше и ниже опасного фрагмента программы.** Таким конвертом для оборачивания фрагментов программы служат две директивы компилятора - {I+} и {I-}:

выше {$I-} - указывает на то, что необходимо отключить (на время) системный контроль ошибок ввода-вывода
опасный фрагмент reset(f);
ниже {$I+} - указывает на то, что необходимо включить системный контроль ошибок ввода-вывода

При наличии директивы компилятора {$I-} даже при наличии ошибок в/в программа аварийно не завершается и ход выполнения программы не нарушается (но после I+ надо тут же проверить, были ли ошибки - если были, то результатам последней операции в/в доверять нельзя).

Замечания:

1) В результате выполнения ошибочных операций (операций с ошибками, которые мы игнорируем) нормальной (значимой с точки зрения решаемой задачи) информации при вводе/выводе вы не получите, но программа продолжит работу.
2) По умолчанию действует директива {$I+}.
3) Необходимо стремиться к тому, что бы {$I+} расположить ближе к описанному фрагменту. Это связано с тем, пока она не включена все ваши действия по вводу-выводу будут происходить впустую.

Контроль ошибок в/в производится с помощью вызова функции без параметров IOResult . Если ошибка имеет место, то значение этой функции отлично от нуля. Поэтому типичный шаблон обработки ошибки ввода-вывода имеет вид:

```
{$I-}
операция ввода-вывода
{$I+}
If IOResult <> 0 Then {обработка ошибки};
```

о том, какая ошибка, можно судить по тому действию, которое завернуто в I+ и I-

Обычно обработка ошибки сводится к сообщению об ошибки и либо к повторению неудачной операции (с теми же параметрами или с новыми), либо к завершению вашей программы (с помощью процедуры HALT или EXIT).

Halt(<код ошибки>); - завершает работу всей программы с указанным кодом ошибки
exit; - завершает работу только текущего блока

Замечание: Опросить функцию IOResult (для каждого конкретного действия по вводу-выводу) можно **лишь один раз** после каждой операции ввода-вывода, т.к. она обнуляет свое внутреннее значение результата (кода ошибки) после каждого вызова (IOResult не запоминает результат своего предыдущего вызова). Поэтому целесообразно сохранять значение, возвращенное IOResult, в специально выделенной переменной.

Обработка ошибок ввода-вывода в Паскале и Delphi:

(синхронная – надо обработать каждое потенциальное место ошибки)

Паскаль

```
{$I-}
reset(f);
{$I+}
If IOResult <> 0
  then begin
         Writeln('Ошибка открытия файла');
         Halt (<код ошибки>);
       end;
```

Дельфи

```
try
  reset(f);      ShowMessage('Ош-ка открытия файла');
except
  on EFOpenError do
    begin
      writeln('Ошибка открытия файла');
    end;
end;
```

Можно (нужно)так:

```
M:
{$I-}
reset(f);
{$I+}
If IOResult <> 0
  then begin
         Writeln('Ошибка открытия файла');
         goto M;
       end;
Продолжение работы
```

Лучше так:

```
r := 0; //признак ошибки ввода-вывода
repeat
  {$I-}
  reset(f);
  {$I+}
  r := IOResult;
  if r <> 0
    then
      begin
         Writeln('Ошибка открытия файла');
         Writeln('Введите правильное имя');
         Readln(s);  //в s - имя файла
      end;
until  r = 0; //условие завершения цикла
Продолжение работы
```


на PL/I было (асинхронная)

ON ERROR

```
begin
  обработка ошибок
  переход в точку за ошибкой (goto M);
end;
```

использовались переменные типа метка для возврата назад (по **запомненному значению метки M**) после завершения обработки ошибки

M:=M1 или M:=M2

```
.....
M1:  <участок пограммы>
.....
M2:  <участок программы>
```

21.11 Обработка исключений в Delphi

С самого начала важно понимать разницу между **исключениями Delphi**, которые связаны с языком программирования, и **аппаратными исключениями,** связанными с аппаратными прерываниями. Вы можете, как говорится, обернуть аппаратное исключение в исключение Delphi и таким образом обработать событие (ситуацию). Например, окончание бумаги на принтере приводит к аппаратному исключению. Delphi в этом случае исключения не получает (не возбуждает?), но подпрограмма, где ситуация обнаружена, может **возбудить исключение**. Однако возбужденное исключение **не приведет к аппаратной ошибке**. Она (аппаратная ошибка) уже существует вне программы (независимо от нее).

При возникновении (возбуждении) исключения **выполнение программы прекращается** и специальный код в программе (системном модуле) цепочку блоков обработки исключений (сначала внутренний для той точки в программе, где возникло исключение, затем – более внешний, затем еще более внешний и т.д.) до тех пор пока не будет найден блок, содержащий нужный обработчик исключения. При таком перемещении по цепочке вызовов вызванные подпрограммы и их данные **выталкиваются из стека**. После обработки исключения управление не возвращается в точку, непосредственно следующую за местом возникновения исключения. Управление передается в точку, непосредственно следующую за блоком, где найден обработчик исключения.

С точки зрения Object Pascal каждая (предопределенная в языке или определенная программистом) исключительная ситуация – **это объект**, порожденный, как говорят, от базового класса **Exception**.. Для работы с такими объектами используются специальные конструкции языка – блоки try…except и try…finally.

Для **реакции на конкретный тип ситуации** применяется блок **try…except**:

```
try
  <Оператор1>
  <Оператор2>
  ……
except
  on Exception1  do <Оператор>;      ---------- обработчик Исключения1
  on Exception2  do <Оператор>;      ---------- обработчик Исключения2
  ……..
  else
    <Обработчик прочих исключений>
end;
```


Между конструкциями try и except размещается т.н. “**защищаемый**” (от ошибок) фрагмент программы. При отсутствии исключений этот фрагмент программы выполняется от начала до конца. Секция Except получает управление в случае возникновения одного из исключений. В ней находятся т.н. обработчики исключений. Каждый **обработчик исключения** представляет из себя директиву вида

on <исключительная ситуация > **do** <оператор – реакция на исключит. ситуацию>

Каждая такая директива связывает ситуацию (заданную своим именем после **on**), с группой операторов (записанных после **do**).

Пример:

```
try
  U:=220;
  R:=0;
  I:=U/R;                          ShowMessage('Деление  на ноль');
except
  on EZeroDivide  do begin Writeln('Деление  на ноль'); Halt;  end;
end;
```

В этом примере замена If…then на try…except внешне не дала существенной **экономии кода**. Однако если, например, операторов присваивания с делением на ноль будет не один, а несколько, то

выигрыш от использования try…except очевиден – достаточно лишь одного блока try на все операторы.

При возникновении исключения все директивы просматриваются последовательно сверху вниз в порядке их описания. Просмотр выполняется до тех пор, пока (после **on)** не будет обнаружено описание (имя) исключения, совпадающее с наступившим исключением. Если описание наступившей ситуации обнаружено, то это исключение обрабатывается, после чего управление передается на **оператор, следующий за блоком *try***.

Если возникла ситуация, не определенная нив в одной из директив, то выполняются те операторы, которые расположены **за словом *else*.** Если их нет, то ситуация считается **необработанной** и будет передана на следующий уровень обработки (в вызывающий блок). Для передачи обработки исключения «вышестоящим инстанциям» часто пользуются оператором ***raise:***

Пример:

```
try
  U:=220;
  R:=0;
  I:=U/R;
except
  Writeln('Ошибка!');
  Raise;
end;
```

Здесь в случае любой ошибки выводится сообщение . Сама же обработка ошибок предоставляется вышестоящим (по цепочке вызовов) блокам.

Оператор ***raise*** используется для **возбуждения исключения**. Более полный вид этого оператора имеет вид:


```
raise EInOutError.create('Ошибка ввода-вывода');
raise EFopenError.create{'Ошибка открытия файла')
raise EOutOfMemory.create('Не хватает памяти');
```

Оператор ***raise* такого вида вызывает следующее действие -** возбуждается конкретное исключение, т.е. **конструируется экземпляр конкретного вида исключения** (т.к.вызван метод create конкретного класса исключения). Команда raise указывает компилятору, что в данном месте необходимо прервать выполнение программы и начать «выталкивание» данных и вызванных подпрограмм из стека.

Кроме блока try…except используется и блок ***try…finally***. Он используется тогда, когда надо, например, возвратить выделенные программе ресурсы (например, память) даже в случае аврийного завершения. Дело в том, что если между выделением и освобождением памяти произошло исключение, то код , освобождающий память не будет выполнен. Синтаксис такого блока имеет вид:

```
try
  <оператор>
  <оператор>
  …….
finally
  <оператор>
  ……
end;
```

Порядок выполнения этой конструкции следующий. Операторы, следующие за try, выполняются в обычном порядке. Если за это время не произошло никаких исключений, то далее выполняются те операторы, которые расположены после finally. Если же между try и finally произошло исключение, то управление немедленно передается на операторы, следующие за finally (эти операторы часто называются ***кодом очистки***).

Допустим вы используете в программе после слова try операторы выделения динамической памяти. Тогда действия по освобождению выделенной памяти надо поместить после слова finally:

```
try
  getmem(p, 100);
  ……
finally
  freemem(p,20);
  ….
end;
```

В данном **случае ресурсы (память) будут освобождены в любом случае** – и при нормальном и при аварийном завершении. Поэтому блок try…finally называют блоком защиты ресурсов.

Замечание: блок try…finally ничего не производит с самим объектом – исключительной ситуацией. Задача try…finally – только прореагировать на аварийную ситуацию в работе программы и проделать определенные программистом действия прежде, чем управление буде передано локадбному обработчику исключения или обработчику исключения вышестоящего уровня. . Сама же исключительная ситуация продолжает «путешествие» по программе и вопрос о ее обработке остается открытым.

Часто в программах **сочетаются блоки try обоих рассмотренных типов** – в один из них помещается **общее** (освобождение ресурсов в finally), а в другой – **особенное** (реакция на конкретные исключения внутри except). В этом случае при возникновении исключений вначале выполняются действия, расположенные после finally, а только после этого выполняются действия, расположенные после except:

```
try
  < оператор >
  < оператор >
  ……..
  try
    < оператор >
    < оператор >
    ……..
  finally
    < оператор >
    < оператор >
    …...
  end;
  < оператор >
  < оператор >
  ……
except
  < оператор >
  < оператор >
  …..
end;
```

блок try…finally, вложенный в try…except
(возможен и обратный порядок вложения блоков)