Глава 11. Файлы

Файл (file) – именованная область данных на носителе информации.

Википедия

Чтобы усовершенствовать ум, надо больше размышлять, а не заучивать.

Рене Декарт

Файлы являются последним из ещё не рассмотренных сложных типов Паскаля. Сначала дадим следующее определение. Структура данных называется динамической, если она может менять свой размер и/или взаимное расположение своих частей в процессе выполнения программы. Все ранее рассмотренные типы стандарта Паскаля описывали статические структуры данных, а вот файл является *динамической* структурой данных.

Ранее в языке Free Pascal мы встречались с псевдодинамическими структурами данных. Например, на первый взгляд переменная

```
var X: string[80];
```

может менять свой размер, принимая на хранение строки длиной от нуля до 80 символов. Однако, при порождении переменной X ей сразу отводится 81 байт памяти, и в дальнейшем размер этой памяти не меняется. То же самое можно сказать и о множествах, например, переменная-множество

```
var Y: set of 1..100;
```

может содержать от нуля до 100 элементов, но при порождении этой переменной сразу отводится место для хранение *всех* элементов (т.е. полного множества).

«Настоящими» динамическими структурами являются файлы, они могут увеличиваться и уменьшаться в размерах во время счёта программы. Правда, изменение в размерах производится порциями памяти, которые называются кластерами, но об этом далее.

Файлы принадлежат новому, внешнему классу памяти ЭВМ. Переменные этого класса могут существовать до начала счёта программы, и продолжать существовать после завершения счёта. Ясно, что такие файлы не могут *принадлежать* программе, т.е. располагаться на её памяти, именно поэтому данный класс памяти мы ранее и не рассматривали. Таким образом, можно считать, что в самом стандарте Паскаля все структуры данных статические.

В стандарте Паскаля, наряду с внешними, были определены и так называемые *внутренние* файлы, которые могут быть как статического, автоматического и динамического классов памяти. Как и остальные переменные Паскаля, они заведомо уничтожаются после окончания программы. Такие файлы оказались бесполезными в практическом программировании, они отсутствуют во всех конкретных реализациях Паскаля и рассматриваться не будут.

Основное назначение файлов очевидно, из них в программу могут поступать её входные данные и в файлы записываться выходные данные (результаты выполнения) программы. Файлы также могут использоваться для промежуточного хранения программой данных большого объёма, так как размер файлов может значительно превышать размер памяти самой программы.

Файлы принадлежат файловой системе, это важная часть операционной системы (ОС), управляющей всей работой компьютера. Каждый файл имеет в файловой системе имя. Имя файла не является именем в смысле языков программирования, оно скорее похоже на адрес сайта в интернете, например:

```
c:\Free_Pascal\bin\i386-win32\a.pas ❶
..\bin\a.pas ❷
```

Имя задаёт путь, который надо пройти в древовидной структуре файловой системы, чтобы достичь нужного файла. Наверное Вы уже знаете, что имя файла может быть абсолютным и относительным (например, просто a.pas),. Относительный путь отсчитывается от текущего или рабочего каталога (working directory) файловой системы, а абсолютный путь – от корня (начала) файловой системы. В приведённом примере ❶ это абсолютный, а ❷ – относительный путь.

Только файловая система может порождать и уничтожать файлы, а также менять их содержимое. Таким образом, для работы с файлом программе приходится обращаться к служебным функциям

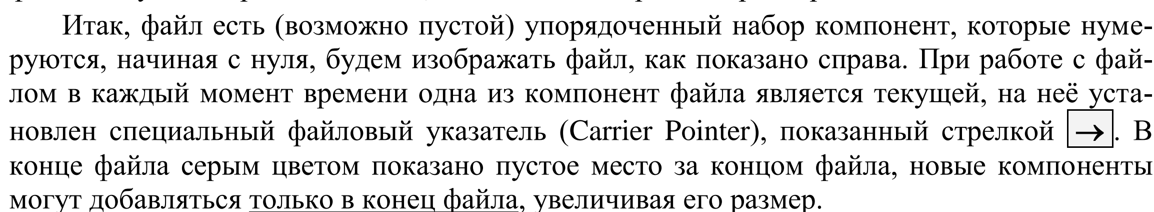
файловой системы (так называемым системным вызовам), передавая им *запросы* на те или иные операции с файлами. Заметим, что мы уже встречались с такой ситуацией при работе с потоками ввода/вывода, которые тоже принадлежат файловой системе. Стандартные процедуры `read` и `write` обращались к файловой системе для выполнения операций по чтению и записи очередной порции данных потока.

Сравним файл, как абстрактную структуру данных, с массивами и записями (см. Таблицу 11.1). Будем говорить, что файл состоит из компонент, хотя в стандарте Паскаля они называются *записями*, что создаёт путаницу с типом **`record`**. Так как весь файл и его компоненты не являются переменными языка программирования, то, как уже говорилось, для выполнения над ними операции приходится вызывать служебные функции ОС. Файловая система, конечно, может выполнять операции как над файлами целиком, так и над входящими в него компонентами.

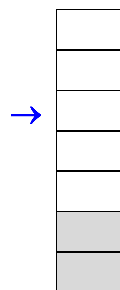
Таблица 11.1. Сравнение массивов, записей и файлов.

Массив	Запись	Файл
Является переменной Паскаля	Является переменной Паскаля	Не является переменной Паскаля
Состоит из элементов	Состоит из полей	Состоит из компонентов
Каждый элемент переменная Паскаля	Каждое поле переменная Паскаля	Компонента не является переменной Паскаля
Все элементы одного типа	Поля могут быть разных типов	Компоненты могут быть разного типа [1]
Элементы не имеют имён	Все поля имеют имена	Компоненты файла не имеют имён
Элементы в массиве упорядочены	Поля в записи не упорядочены	Компоненты файла упорядочены

Теперь следует понять, что сама файловая система и язык программирования могут видеть структуру файла по-разному. Для файловой системы каждый файл является просто упорядоченной последовательностью (возможно и пустой) компонент минимального размера – байтов. Практически так же видит файл и язык С. Язык Паскаль считает, что каждый файл состоит из компонентов *одного* типа, но разные файлы могут иметь компоненты *разного* типа и, следовательно, разного размера. Например, один файл имеет небольшие компоненты типа `real`, а другой – большие типа массива `Mas=`**`array`**`[1..1000]` **`of`** `integer;`. А вот, например, в языке Фортран даже компоненты одного файла могут быть разного типа и, следовательно, разного размера.



Итак, файл есть (возможно пустой) упорядоченный набор компонент, которые нумеруются, начиная с нуля, будем изображать файл, как показано справа. При работе с файлом в каждый момент времени одна из компонент файла является текущей, на неё установлен специальный файловый указатель (Carrier Pointer), показанный стрелкой →. В конце файла серым цветом показано пустое место за концом файла, новые компоненты могут добавляться только в конец файла, увеличивая его размер.



Каждый файл описывается в Паскале *файловой переменной*, тип которой обязательно должен иметь имя (причину этого мы вскоре поймём). Синтаксис описания типа файла:

<тип файла> ::= **`file of`** <имя типа компоненты>;

Отметим, что в Паскале всегда считаются описанными две стандартные файловые переменные `input` и `output`, они имеют стандартный тип текстовых файлов с именем `text`.

В языке Free Pascal переменные `input` и `output` имеют синонимы `stdin` и `stdout` соответственно (как в языке С), кроме тогоещё есть стандартная файловая переменная `stderr` (есть синоним `erroutput`), она предназначена для вывода сообщений об ошибках во время выполнения программы, обычно она, как и файловая переменная `output`, связывается с экраном консоли.

Компоненты файла могут быть только статического типа. Напоминаем, что тип называется статическим, если размер его переменных известен до начала счёта программы и не меняется во время

[1] Компоненты файлов могут быть разного типа только с точки зрения языков программмирования. С точки зрения файловой системы все компоненты одного типа, это байты – минимальные адресуемые области памяти компьютера.

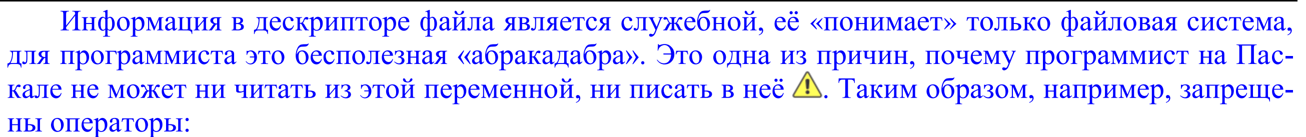
счёта. В языке Free Pascal не являются статическими, например, упоминавшиеся ранее динамические массивы. Примеры описания файлов:

```
type Fint=file of integer;
     Mas=array[1..100000] of real;
     FMas=file of Mas;
var Fi: Fint; FM1,FM2: FMas;
```
[1]

Во время счёта программы в файловой переменной содержится вся необходимая информация для работы с файлом: имя и месторасположение файла в файловой системе, позиция текущей компоненты файла, режимы работы с файлом и т.д. Более уместно называть их не файловыми переменными, а, как в других языках, файловыми дескрипторами (описателями). Вся необходимая информация должна быть помещена в файловый дескриптор до начала работы с компонентами файла.

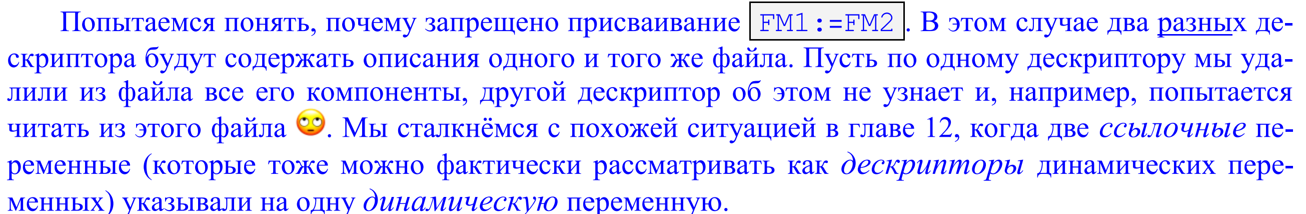
Итак, в этих примерах компонентами файла `Fi` являются целые числа, а файлов `FM1` и `FM2` – массивы вещественных чисел. Здесь стоит обратить внимание на такой важный момент. Компонентами файла `Fi` целых чисел являются не строки, изображающие целые числа (т.е. лексемы целых чисел), а закодированные целые числа во внутреннем машинном представлении. В языке Free Pascal для типа `integer` это всегда 16 бит или (в режиме `{$mode objfpc}`) 32 бита так называемого дополнительного кода целого числа. Например, лексема целого числа `-2078` состоит из 5 символов, она имеет такие 2-й и 16-й коды длиной 32 бита (с общепринятыми цифрами `0..9,A..F`):

$1111\ 0111\ 1110\ 0010_2 = 0F7E2h = F7E2_{16}$



Информация в дескрипторе файла является служебной, её «понимает» только файловая система, для программиста это бесполезная «абракадабра». Это одна из причин, почему программист на Паскале не может ни читать из этой переменной, ни писать в неё ⚠. Таким образом, например, запрещены операторы:

`FM1:=FM2; if FM1=FM2 then` … и т.д.



Попытаемся понять, почему запрещено присваивание `FM1:=FM2`. В этом случае два разных дескриптора будут содержать описания одного и того же файла. Пусть по одному дескриптору мы удалили из файла все его компоненты, другой дескриптор об этом не узнает и, например, попытается читать из этого файла 😳. Мы столкнёмся с похожей ситуацией в главе 12, когда две *ссылочные* переменные (которые тоже можно фактически рассматривать как *дескрипторы* динамических переменных) указывали на одну *динамическую* переменную.

Вам, быть может, приходилось открывать один документ, скажем, в редакторе Word, дважды, при этом второй раз Вам предлагается открыть копию этого документа и только по чтению [2]. Таким образом, с файловым дескриптором можно делать только одно: передавать его *по ссылке* в подпрограмму (обязательно поймите и умейте объяснять, почему невозможна передача *по значению*).

Ясно, что перед работой с файлом его дескриптор надо связать с конкретным файлом в файловой системе. В стандарте Паскаля предусмотрен простой механизм такого связывания, оно всегда выполняется статически, т.е. до начала счёта программы. Скажем, для заголовка программы

```
program MyProg(input,output,Fint);
```

При запуске на счёт можно задать такое статическое связывание

```
C:>MyProg.exe >a.txt Fint<b.int
```

Это похоже на перенаправление стандартных потоков ввода/вывода: перед началом счёта поток `output` связывается с файлом `a.txt`, а поток `Fint` связывается с файлом `b.int`.

Такой способ связи файловых переменных с конкретными файлами имеет существенный недостаток. Действительно, имя файла, с которым надо связать файловую переменную, часто неизвестно до начала счёта. Это имя может, например, запрашиваться у пользователя: «Введите имя Вашего

---

[1] Вскоре мы увидим, что безымянные файловые типы хотя и допускаются, но бесполезны.

[2] В отличие от ОС Windows, операционные системы семейства Unix обычно разрешают открывать один и тот же файл на разные дескрипторы, причём не только по чтению, но и по записи. Результаты при этом будут самые непредсказуемые, но там программисту разрешено делать практически все глупости, его действия мало контролируются.

файла». Естественно, что все конкретные реализации Паскаля отказываются от такого *статического* (до начала счёта) связывания в пользу *динамического* связывания (уже во время счёта).

Например, язык Free Pascal статически (перед началом счёта) связывает только стандартные файловые переменные input, output и stderr, все остальные файловые переменные связываются динамически. Таким образом, это существенное отступление от стандарта и дальше мы будем изучать работу с файлами, как это принято в языке Free Pascal.

Итак, перед началом работы с файлом надо связать файловую переменную с конкретным файлом или устройством, это делается с помощью стандартной процедуры языка Free Pascal:

Assign(<файловая переменная>,{<имя файла> | <имя устройства>})

Допускается и более «осмысленное» для этой процедуры имя AssignFile. Имя файла задаётся строковым выражением (или просто строковой константой), например:

```
Assign(Fint,'MyIntFile.int')
```

Устройства задаются строковыми выражением со стандартными именами, установленным для них в операционной системе, например (для Windows):

```
Assign(output,'PRN');  { Принтер, есть синоним 'LPT' }
Assign(output,'');     { Экран консоли :) (странное имя) }
Assign(output,'NUL');  { Вывод в «никуда» }
```

В последнем примере файловая переменная output связывается со специальным потоком ввода/вывода, который имеет имя NUL. Это «пустой» поток, при записи в него данных они просто пропадают (выбрасываются), а при чтении сразу возникает ошибка eof, т.е. фиксируется попытка чтения из пустого потока (или пустого файла). Например, при запуске программы на счёт

```
C:>MyProg >NUL
```

весь вывод стандартного файла output просто пропадёт :(.

Особым случаем является связывание выходного потока output одной программы с входным потоком input другой. В ОС Windows это можно задать так:

```
C:\>A.exe | B.exe { output 'A.exe' → input 'B.exe' }
```

Программы A.exe и B.exe могут работать как последовательно (сначала заканчивается A.exe, её вывод где-то сохраняется операционной системой (во временном файле) и потом подаётся на вход B.exe), так и параллельно (скажем, на двух процессорных ядрах).

Таким образом, перед началом работы с файлом надо связать файловую переменную с конкретным файлом или устройством.

Итак, поток ввода/вывода подсоединён или к файлу (хранилищу данных) или к устройству (поставщику или приёмнику данных). Несмотря на их схожесть, это всё-таки разные сущности, например, содержимое файла можно читать много раз, а с клавиатуры данные поступают только единожды. Несмотря на это, операционные системы семейства Unix считают почти все устройства файлами специального вида. Такая унификация, как считается в этой ОС, помогает сделать работу с потоками единообразной. ОС Windows придерживается другого мнения.

Надо учесть, что процедура Assign производит только *формальное* связывание, не проверяется, что указанный файл существует, может ли Ваша программа его читать и писать в него и т.д. Делается единственная проверка, что директория, в которой расположен файл, существует, и программа может читать содержимое этой директории, например:

```
Assign(F,'C:\MyDirectory\MyFile.int');
```

В этом случае сам путь 'C:\MyDirectory\' должен существовать и эта директория должна быть доступна Вашей программе по чтению.

```
Assign(F,'MyFile.int');  { доступна текущая директория }
```

После связывания можно открывать файл для работы. Предварительно, однако, нам надо определить для файла два важных понятия: *способ доступа* и *режим работы*.

Способ доступа к компонентам файла. Основным способом работы с компонентами является *обмен данными* между компонентой файла (это чужая память) и переменной Паскаля. Программе доступны операции *чтения* копии компоненты в свою переменную (её тип, естественно, должен

совпадать с типом компоненты файла), и операция *записи* копии своей переменной в компоненту файла. Различают последовательный и прямой способ доступа.

1. Последовательный доступ производит чтение компонент файла строго последовательно, начиная с первой (с нулевым номером) компоненты, и при каждом чтении указатель файла смещается к следующей позиции. В любой момент, однако, можно дать команду установить указатель опять на начало файла и читать компоненты заново. Начать запись возможно только в пустой файл, компоненты записываются в файл тоже строго последовательно.
2. Прямой доступ производит чтение и запись компонент файла по их номерам в файле (начиная с нулевого номера). Обычно сначала процедурой Seek устанавливают указатель на нужную компоненту файла, а потом выполняют операцию чтения или записи, например

```
Seek(Fi,20); read(Fi,X);
{ Здесь в X читается 21-я компонента файла }
Seek(Fi,0); {эквивалентно Reset(Fi)}
```

Обратите внимание, что прямой доступ невозможен, если компоненты файла имеют разный тип (и, вероятно, разную длину). Процедура `Seek(Fi,20)` в предыдущем примере приказывает файловой системе установить указатель на `20*sizeof(integer)=20*4` т.е. 80-й байт файла. При переменной длине компонент длина каждой из них известна только программисту, поэтому дать такую команду файловой системе невозможно. Прямой доступ невозможен и в текстовых файлах, так как они состоят из строк символов переменной длины. Ясно также, что прямой доступ не имеет смысла для больших файлов, хранящихся на устройствах с последовательным доступом, например, на так называемых стримерах (кассетах магнитной ленты), это крайне малоэффективно.

Язык Free Pascal считает, что при работе с не текстовыми файлами используется (если это возможно) прямой способ доступа.

Обычно файловая система предоставляет программисту ещё так называемый режим прямого отображения файла (или его непрерывной части) на память программы MMF (Memory Mapped Files). При этом всему файлу или его части ставиться в соответствие непрерывный участок оперативной памяти. Например, можно отобразить часть файла `Fi[0..10000]` на участок оперативной памяти, который считать массивом, тогда читать компоненты файла можно оператором `x:=↑Fi[i]`, а писать оператором `Fi↑[i]:=x` (на самом деле здесь всё несколько сложнее).

В стандарте Паскаля определён только последовательный доступ, именно его мы и будем здесь рассматривать. Теперь определим режимы работы с файлом.

Режим работы с компонентами файла. В языке Free Pascal существуют два основных и один дополнительный режим работы.

11.1. Режим только чтение компонент файла

Учитесь и читайте. Читайте книги серьёзные. Жизнь сделает остальное.
Ф.М. Достоевский

В языке Free Pascal режим работы с *текущим* файлом (но не с устройством!) хранится в стандартной переменной с именем Filemode, она описана в модуле Crt (или WinCrt) как типизированная константа:

```
const Filemode: byte=2; {значение по умолчанию}
{     Filemode=0  – только чтение;
      Filemode=1  – только запись;
      Filemode=2  – чтение и запись. }
```

Заметим, что у файловой системы (истинного хозяина файла) может быть своё «мнение», в каком режиме можно работать с файлом. Даже если сама программа установит `Filemode=2`, но для данного программиста в файловой системе файл закрыт на запись, то, естественно, запись в такой файл невозможна.

Обычно при работе на персональном (личном) компьютере для пользователя установлен в операционной системе режим полного доступа (так называемый режим суперпользователя). Некомпетентный пользователь при этом может легко испортить операционную систему (например, удалив важные

файлы) и вывести компьютер из строя. Исходя из этого, на планшетах и смартфонах (где у пользователей часто недостаточные познания в функционировании компьютера) установлен режим ограниченного доступа и, например, многие файлы открыты только на чтение и даже просто не видны.

Запись из программы в файл, открытый в режиме `Filemode=0` блокируется. Ясно, что это безопасный режим, именно в нём и надо работать, если нам не надо менять файл. Для открытия файла в этом режиме надо присвоить `Filemode:=0` и затем вызвать стандартную процедуру

```
Reset(<файловая переменная>)
```

В языке Free Pascal процедура Reset (и рассматриваемая далее ReWrite) имеют второй необязательный параметр:

```
Reset(<файловая переменная>[,<размер буфера>])
```

Буфер используется при работе с безтиповыми файлами (см. разд. 11.6), для типизированных файлов он бесполезен.

Файловая переменная должна быть предварительно связана с конкретным файлом вызовом процедуры Assign, этот файл должен существовать, хотя и может быть пустым. Если файл не существует, то фиксируется ошибка времени выполнения «Exitcode=2. File not found».

Повторное открытие одного файла для другой файловой переменной в ОС Windows влечёт ошибку времени выполнения «Exitcode=5. File access denied». Аналогичная ошибка возникает, если файл уже используется другой программой: «Процесс не может получить доступ к файлу, так как файл используется другим процессом».

Программист может предварительно убедится, что файл существует, вызвав логическую функцию FileExists,[1] например:

```
if FileExists('a.int') then Reset(Fi) else …
```

Важно понять, что читать из файла можно только существующие компоненты, иначе возникает ошибка. Следовательно, если Вы не уверены, что следующая компонента в файле существует, перед каждым чтением надо вызывать функцию `eof(<файловая переменная>)`. Когда эта функция вернёт значение **true**, читать из файла нельзя (это не значит, что в файле совсем не осталось данных, просто в нём осталось меньше байт, чем размер компоненты файла).

В качестве примера рассмотрим следующую задачу. Используя файловый дескриптор файла целых чисел, надо найти сумму всех *положительных* чисел файла, которые больше последнего числа этого файла.

Осознание задачи. Кто-то уже описал в программе файловый дескриптор и связал его с конкретным файлом, нам осталось только найти искомую сумму. Следовательно, это *подзадача*, результат которой – одно целое число, значит, надо написать функцию. Ясно, что эта функция будет читать файл дважды: сначала требуется найти последнее число, а лишь затем начинать суммирование.

В качестве предусловия будем считать, что файл существует и наша программа может из него читать. Поймите, что наша функция не сможет определить, существует ли этот файл, так как не знает его имени, а знает только файловый дескриптор. Кроме того, предполагается, что искомая сумма не выходит за диапазон значений целого типа.

Спецификация задачи. Особым случаем является пустой файл, будем в этом случае выдавать ответ минус единица, надо понять, что такого ответа не может быть ни для одного непустого файла. [i] [см. сноску в конце главы]

Итак, где-то выше уже описано

```
type Fint=file of integer;
var  Fi: Fint;
```

Теперь опишем нашу функцию.

```
function SumPosGtLast(❶ var F: Fint): integer;
  var Sum,Last,x: integer;
begin ❷ Reset(F); SumPosGtLast:=-1; { пустой }
```

[1] Эта функция находится в модуле с именем Sysutils, предварительно надо его подключить: `uses Sysutils;`. Там же есть полезная функция DirectoryExists.

```
  if not eof(F) then begin { не пустой }
    repeat read(F,Last)
    until eof(F); { Last=последний }
  ❸ Reset(F); Sum:=0;
    repeat read(F,x); { файл не пустой! }
      if (x>0) and (x>Last) then Sum:=Sum+x
    until eof(F);
    SumPosGtLast:=Sum
  end
end;
```

Как мы уже говорили, файловую переменную можно передавать только по ссылке, так что попытка опустить ❶ **var** будет ошибкой (Вы должны уметь объяснить, почему ⚠). Далее, про дескриптор F известно только то, что он уже связан с конкретным файлом процедурой Assign. Всё остальное (существует ли такой файл, открыт ли он, открыт ли только на чтение или ещё и на запись) функции неизвестно. Поэтому оператор ❷ `Reset(F)` будет иметь следующий эффект.

1. Дескриптор не связан с файлом – ошибка (у нас не обрабатывается).
2. Файл не существует – ошибка (у нас не обрабатывается).
3. Файл закрыт на чтение (Filemode=1) – ошибка (у нас не обрабатывается).

Мы уже знаем, как можно обработать такие ошибки в языке Free Pascal. Надо отключить стандартную реакцию на ошибки ввода/вывода директивой {$I-} и самим обрабатывать такие ошибки, например:

```
begin {$I-}
  Reset(F); k:=IOResult;
  if k<>0 then begin { были ошибки }
{ разные k=Exitcode для разных ошибок }
    if k=2 { файл не найден }
      then SumPosGtLast:=-2;
    if k=104 { файл закрыт на чтение }
      then SumPosGtLast:=-3;
    if k=10 { ошибка при чтении файла }
      then SumPosGtLast:=-4;
    …
  end else begin { всё хорошо } {$I+}
    SumPosGtLast:=-1;
          …
```

4. Файл не открыт – указатель на начало, режим работы в FileMode.
5. Файл как-то открыт – указатель на начало, режим работы в FileMode.

Отметим, что учащиеся часто после первого просмотра файла забывают ставить второй оператор ❸ `Reset(F)` для нового просмотра файла с начала.

Заметим, что по предусловию мы не проверяли, будет ли *итоговая* вычисленная сумма `Sum<=Maxint`, однако, если в языке Free Pascal заменить тип функции на int64 ($Maxint64 \approx 2^{63}$) и учесть, что максимальный размер файла в широко распространённой файловой системе NTFS равен примерно 2^{44} байт <= 2^{42} чисел типа longint, то всё будет хорошо (докажите это!).

Теперь вспомним, что после отладки можно выполнять этап *оптимизации*. При втором просмотре файла наша функция для каждой компоненты файла проводит две проверки: `x>0` и `x>Last`. Можно, однако, проводить в цикле только одну проверку, для этого заменим выделенный в рамку фрагмент функции на следующий:

```
if Last<0 then Last:=0;
repeat read(F,x);
  if x>Last then Sum:=Sum+x
until eof(F);
```

Можно надеяться, что скорость работы функции возрастёт.

В нашей функции `SumPosGtLast` мы просматривали файл два раза. В языке Free Pascal можно ограничиться одним просмотром, если предварительно узнать у файловой системе длину файла с помощью функции `FileSize` (см. разд. 11.5):

```
function SumPosGtLast(var F: Fint): integer;
  var Sum,Last,x,n: integer;
begin Reset(F); SumPosGtLast:=-1; { пустой }
  n:=FileSize(F);
  if n>0 then begin { не пустой }
    Seek(F,n-1); read(F,Last); { Last=последний }
    if Last<0 then Last:=0;
    Reset(F); Sum:=0;
    for n:=0 to n-2 do begin
      read(F,x);
      if x>Last then Sum:=Sum+x
    end;
    SumPosGtLast:=Sum
  end
end;
```

Можно надеятся, что при большом размере файла это будет работать много быстрее.

На этом примере Вы должны понять, что хороший программист должен подумать о многих вещах, даже при решении такой простой задачи.

В других языках может быть иной механизм контроля ошибок работы с файлом. Например, в языке Фортран в операторе чтения можно явно задать метки программы, на которые надо перейти, когда будет достигнут конец файла или будет ошибка чтения. В Фортране каждый оператор начинается с 8-позиции строки, в первых 6-ти позициях может стоять целочисленная метка, со знака **!** начинается комментарий:

```
       open (F,file='a.int')  ! открыт файл F
       read (F,END=10,ERR=20) x,y,z
          . . .
    10 ! При конце файла
    20 ! При ошибке чтения
```

11.2. Режим записи и чтения компонент файла

Только самые мудрые и самые глупые не поддаются обучению.

Конфуций, V век до н.э.

Для открытия файла в этом режиме (в языке Free Pascal должно быть значение `Filemode=2`) предназначена процедура стандарта Паскаля

`ReWrite(<файловая переменная>)`

Файловая переменная должна быть предварительно связана с некоторым файлом, этот файл может быть пустым и даже не существовать. Этот оператор имеет следующий эффект:

1. Дескриптор не связан с файлом – ошибка (у нас не обрабатывается).
2. Файл закрыт на запись – ошибка (у нас не обрабатывается).
3. Файл не существует – порождается новый пустой файл.
4. Файл существует – из файла удаляются все его компоненты и он становится пустым.

В стандарте Паскаля это единственный способ создать новый файл. Теперь понятно, почему режим называется для «записи и чтения», а не для «чтения и записи»: без предварительной записи в файл читать из него невозможно, так как он пустой. И только после записи в этот файл какого-то количества компонент, можно выполнить операцию `Reset`, установить указатель на начало файла, и только затем читать его компоненты.

На языке Free Pascal в режиме `Filemode=2` процедура `Reset(<файловая переменная>)` открывает существующий файл в режиме прямого доступа по чтению и записи (см. раздел 11.5). Открыт файл только для последовательного доступа в этом языке невозможно. В языке Free Pascal можно также попросить файловую систему выполнить различные операции над файлом целиком, например, удалить файл с помощью вызова процедуры

`Erase(<файловая переменная>)`

Переименовать файл можно с помощью вызова процедуры

`Rename(<файловая переменная>,<новое имя>)`

В качестве примера решим такую задачу. Даны две файловых переменных, связанных с какими-то файлами целых чисел, причём второй файл может и не существовать. Необходимо, чтобы второй файл содержал все нечётные положительные числа из первого файла, сам первый файл при этом должен остаться неизменным.

При осознании задачи мы понимаем, что, если второй файл был не пуст, то из него надо предварительно все целые числа удалить. Особым является случай пустого исходного файла, сделаем спецификацию, что в этом случае второй файл тоже надо сделать пустым. Понятно, что наша подзадача реализуется в виде процедуры:

```
procedure CopyOddPositive(❶var F1,F2: Fint);
  var X: integer;
begin Reset(F1); ❷ReWrite(F2);
  while not eof(F1) do begin { просмотр файла F1 }
    read(F1,X);
    if ❸odd(X) and (X>0) then Write(F2,X)
  end;
  ❹Close(F2); {❺Close(F1)}
end;
```

Файловые переменные можно передавать только по ссылке ❶ **var**. Операция ❷ `ReWrite(F2)` очищает существующий или порождает новый пустой файл. Почему-то много проблем у учащихся возникает с функцией проверки нечётности целого числа ❸ `odd(X)`. Многие вместо этого пишут `X mod 2<>0`, но здесь есть сложная операция деления (если её не удалит оптимизирующий компилятор), а функция `odd` для проверки нечётности никакого деления не производит. Некоторым компромиссом может служить одновременная проверка чётности и положительности оператором вида

```
if X mod 2=1 then Write(F2,X)
```

Поймите, как это работает. И, наконец, чтобы разобраться с операцией ❹ `Close(F2)` нам придётся познакомиться с работой с файлами на более глубоком уровне.

11.3. Файлы и потоки ввода/вывода

Преподавание программирования – дело почти безнадёжное, а его изучение – непосильный труд.

Чарльз Уэзерелл.
«Этюды для программистов».

Итак, весь обмен данными между программой и «внешним миром» производится через потоки ввода/вывода. Каждый поток может быть подключён либо к некоторому устройству, которое поставляет и принимает порции данных, либо к файлу (хранилищу данных).[1] Именно поэтому процедуры `read` и `write` одинаковы для работы с файлами и с устройствами, а файловые переменные «на самом деле» описывают потоки ввода/вывода.

Например, как Вы уже знаете, стандартный поток `output` можно подключить к устройствам (экрану, принтеру, «пустому» устройству `NUL`), получателю данных (например, другой программе), а также к текстовому файлу. Правда, до сих пор это делалось только при запуске программы на счёт, например:

---

[1] Как уже упоминалось, можно обобщить понятие потока и подключать его к абстрактным «поставщикам» и «получателям» данных, например, к порождающей функции или сетевому устройству.

```
C:>MyProg.exe >a.txt
```

Теперь с помощью файловых переменных мы может динамически подключать (нестандартные) потоки ввода/вывода как к устройствам, так и к файлам. Например, подключение файла целых чисел к файлу:

```
type Fint=file of integer;
var Fi: Fint;
begin Assign(Fi,'MyIntFile.int'); Reset(Fi);
```

Попробуем понять, в каком режиме (см. разд. 4.2) будет открыт входной поток Fi. Так как доступ к файлу (например, на диске) производится на несколько порядков медленнее, чем к обычной (оперативной) памяти ЭВМ, то естественно, что надо открыть этот поток с буферизацией. При этом мы будем «привозить» из файла не по одному целому числу (это 2, 4 или 8 байт), а сразу по многу таких чисел, размещая их в буфере, расположенном в оперативной памяти. Как правило, диск устроен так, что при обращении к нему читается так называемый кластер (как правило, это сектор дорожки диска). Сейчас минимальный размер кластера обычно 4 Кб, таким образом с диска в буфер читается сразу 2048, 1024 или 512 целых чисел.

В отличие от символов, среди целых чисел нет никаких «управляющих» или «служебных», поэтому поток открывается без контроля. И, наконец, никакие копии с целых чисел снимать не надо, так что поток открывается без эха. Обычно в таких же режимах (с буферизацией, без контроля и без эха) открываются файловые потоки и на вывод.

Наличие буфера сильно увеличивает скорость работы программы, так как время чтения данных даже с быстрого, так называемого «твердотельного» диска SSD (Solid-State Drive) в *сотни* раз медленнее, чем из оперативной памяти. Когда диск сильно тормозит работу программы, можно делать и так называемую *двойную* буферизацию. Программа при этом заводит два буфера и сначала заполняет с диска первый из них. Затем, пока программа читает данные из первого буфера, параллельно с этим заполняется второй буфер. Когда первый буфер закончится, программа переключается на второй буфер, а в первый читается следующая порция данных, и т.д. Это так называемый асинхронный режим работы.

Стандарт языка Паскаль ISO /IEC 7185:1990 предполагает ещё более быструю работу с компонентами файла, без использовании буферной переменной. Файловая переменная при этом считается ссылочной, в каждый момент времени эта ссылка указывает на текущую позицию компоненты файла в буфере, например, цикл для суммирования компонент файла целый чисел F:

```
Sum:=0;
while not eof(F) do begin
   read(F,X); Sum:=Sum+X
end;
```

в стандарте Паскаля можно записать так

```
Sum:=0;
while not eof(F) do begin
   Sum:=Sum+F↑; get(F)
end;
```

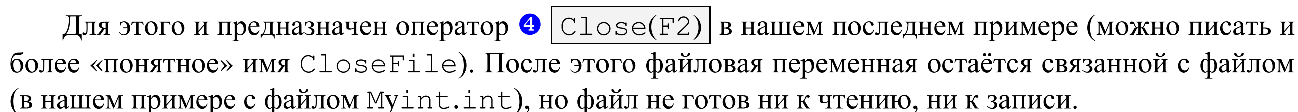
Здесь не используется вспомогательная переменная X, в которую копируется целое число из буфера ввода (который всё равно расположен в памяти *моей* программы ⚠). Мы просто приходим на нужное место буфера по ссылке `F↑` и берём оттуда число, а потом устанавливаем ссылку на следующее число буфера с помощью вызова процедуры `get(F)`. Таким образом, экономится перемещение целого числа из буфера в переменную X. Начиная с версии Free Pascal 3.0.2 желающие могут использовать такой метод доступа к файлам, задав директиву `{$mode ISO}`. Необходимо, однако, учитывать, что, к сожалению, в этом случае нужно и описывать файлы по стандарту Паскаля в виде параметров заголовка программы **program**. Кроме того, по стандарту станет работать и функция округления round, а также операция взятия остатка **mod** ☹, так что пользы от этого не будет.

Итак, при чтении из потока использование буферизации повышает скорость работы программы, а вот при записи в поток буферизация приводит к дополнительному эффекту. Дело в том, что буфер, сначала полностью заполняется данными, а затем они записываются в поток (в файл на диск, посы-

лаются на принтер и т.д.). Буфер, однако, является массивом, который располагается в памяти программы и заведомо уничтожается при её окончании (для Паскаля – при выходе на последний **end.**). Таким образом, если он заполнен не полностью, то последняя порция данных в поток не запишется и «погибнет» при окончании программы, а маленькие выходные файлы вообще останутся пустыми.

Для закрытия файла и принудительной записи буфера надо использовать стандартную процедуру

```
Close(<файловая переменная>)
```

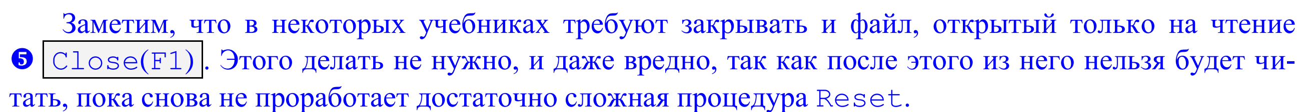


Для этого и предназначен оператор ❹ `Close(F2)` в нашем последнем примере (можно писать и более «понятное» имя `CloseFile`). После этого файловая переменная остаётся связанной с файлом (в нашем примере с файлом `Myint.int`), но файл не готов ни к чтению, ни к записи.

Кроме того, можно принудительно записать (сбросить) буфер в файл с помощью процедуры

```
Flush(<файловая переменная>)
```

В программе на языке Free Pascal *некоторые* открытые файла принудительно закрываются при окончании программы (нормальном или аварийном), а другие файлы не закрываются, так что лучше самим писать оператор `Close`. Отметим, что использование `Flush` вместо `Close` более предпочтительно, если предполагается *дальнейшая* работа с файлом, так как иначе надо снова открывать файл процедурой `Reset`.



Заметим, что в некоторых учебниках требуют закрывать и файл, открытый только на чтение ❺ `Close(F1)`. Этого делать не нужно, и даже вредно, так как после этого из него нельзя будет читать, пока снова не проработает достаточно сложная процедура `Reset`.

Можно выполнять операции над файлами, давая команды непосредственно файловой системе из командной строки. Например, для ОС Windows, скопировать файл `a.txt` в файл `b.txt`:

```
C:>copy a.txt b.txt
```

или уничтожить файл `a.int`

```
C:>del a.int
```

и т.д.

11.4. Текстовые файлы

Главный враг знания не невежество, а иллюзия знания.

Стивен Хокинг

Мы уже знакомы с важным понятием текста – упорядоченной последовательности символьных строк переменной длины. Текстовые файлы имеют в Паскале стандартный тип с именем `text` (в языке Free Pascal есть более «понятный» синоним `TextFile`). К текстовым файлам можно подключать и стандартные потоки `input` и `output`, правда, при этом могут меняться режимы их работы. Например, по умолчанию поток `output` подключён к текстовому экрану консоли *без буферизации*, а при подключении к текстовому файлу

```
Assign(output,'a.txt'); ReWrite(output);
```

переключается в режим работы *с буферизацией*. Кроме того, при вводе из текстовых файлов определено понятие конца строки и можно пользоваться функцией `eoln(<файловая переменная>)`.

К сожалению, текстовые файлы можно открыть в режиме только для чтения (`Reset`) и только для записи (`ReWrite`). Режима записи с последующим чтением и чтения с последующей записью для текстовых файлов нет. При чтении после `ReWrite` возникает ошибка `104` (File not open for input – Файл не открыт для чтения), а при записи после `Reset` будет ошибка `105` (File not open for output – Файл не открыт для записи).

Этот недостаток в языке Free Pascal частично исправлен определением дополнительного режима работы – дозаписи данных в конец *существующего* текстового файла. Для этого применяется процедура `Append`, например

```
Assign(F,'a.txt'); Append(F);
```

Этот оператор имеет следующий эффект.

1. Дескриптор не связан с файлом – ошибка (у нас не обрабатывается).
2. Файл не существует – ошибка (мы говорили, как это проверить).

3. Файл закрыт на чтение и/или запись – ошибка (у нас не обрабатывается).

4. В остальных случаях файл открывается на запись и указатель ставится на свободное место за концом файла.

При запуске программы (например, `A.exe`) на счёт тоже можно подключить стандартный поток вывода `output` к текстовому файлу (например, `a.txt`) в режиме дозаписи с конец:

```
C:>A.exe >>a.txt
```

В качестве примера рассмотрим следующую задачу. Даны две файловых переменных, уже связанных с какими-то текстовыми файлами. Написать на языке Free Pascal процедуру,[1] которая дописывает все строки первого файла, которые начинаются с символа `'*'`, в конец второго, оставляя первый файл неизменным. Осознаём, что пустые строки не начинаются со `'*'`, поэтому их во второй файл не переписываем. На случай, если длина строки окажется больше, чем 255, включим режим `{$H+}`, в котором строки трактуются как тип `AnsiString` и имеют длину до 2^{31} байт.

```
procedure Modify(var F1,F2: text); {$H+}
  var S: string;
begin Reset(F1); Append(F2);
  while not eof(F1) do begin { просмотр файла F1 }
    readln(F1,S);
    if (length(S)>0) and (S[1]='*') then Writeln(F2,S)
  end;
  Close(F2)
end;
```

Обратите внимание, что вместо `read` использовалась процедура `readln`, а вместо `write` – `writeln`. Типичной ошибкой учащихся является использование процедур `readln` и `writeln` для не текстовых файлов, там никаких «строк» нет!

11.5. Режим прямого доступа к компонентам файла

Есть познанное знание – то, о чём мы знаем, что знаем. Есть также познанное незнание – то, о чём мы знаем, что не знаем. Но есть ещё и непознанное незнание – это то, о чём мы не знаем, что не знаем.

Дональд Рамсфельд, министр обороны США у президента Джорджа Буша.

Прямой доступ отличается от последовательного тем, что всегда можно поставить указатель файла на любой его компонент, а также на свободное место за концом файла. При этом считается, что время выполнения такой операции не зависит от позиции файла, на которую ставится указатель.

Этот режим реализован в большинстве языков программирования. Рассмотрим, как это сделано в языке Free Pascal, где его можно применять только для не текстовых файлов. Это связано с тем, что в текстовых файлах, компонентами которых, по существу, являются строки символов *переменной* длины, неизвестно, где в файле начинаются его компоненты-строки.

Для организации прямого доступа в язык Free Pascal добавлены следующие подпрограммы (`F` – файловая переменная).

1). Функция `FileSize(F)` – возвращает число компонент файла (0 для пустого).[2]

2). Функция `FilePos(F)` – возвращает номер текущего компонента файла (начиная с 0).

3). Процедура `Seek(F,n)` – ставит указатель на `n`-ю позицию файла `F` (начиная с 0).

4). Процедура `SeekEof(F)` – ставит указатель на свободную позицию за концом файла.

---

[1] На стандарте Паскаля написать такое будет значительно сложнее, так как там нет режима `Append`.

[2] Чтобы получит размер файла не надо читать его от начала до конца, т.к. этот размер хранится в каталоге файловой системы, а может быть и в файловой переменной (дескрипторе файла).

5). Процедура Truncate(F) – удаляет все компоненты файла, начиная с текущей (включительно), указатель теперь стоит на свободной позиции за концом файла, это усечение файла.

Рассмотрим следующую операцию над файлами. Для файла целых чисел вычислить сумму значений компонент, расположенных после максимального числа этого файла. Результатом будет одно целое число, значит надо реализовать эту операцию как функцию.

Сделаем следующую спецификацию. Когда в файле несколько максимумов, будем брать первый из них. Для пустого файла и случая, когда единственный максимальный компонент в конце файла, будем возвращать ответ ноль.

```
function SumAfterMax(var F: Fint): integer;
  var Sum,MaxPos,Max,x,i,NumComp: integer;
begin Reset(F); Sum:=0;
  NumComp:=FileSize(F); { число компонент }
  if NumComp>0 then begin { файл не пустой }
    MaxPos:=0; read(F,Max);
    for i:=1 to NumComp-1 do begin
      read(F,x);
      if x>Max then begin
         Max:=x; MaxPos:=i
      end;
      Seek(F,MaxPos); { на Max компоненту }
    end;
    for i:=MaxPos+1 to NumComp do begin
      read(F,x); Sum:=Sum+x
    end
  end;
  SumAfterMax:=Sum
end;
```

Наш алгоритм прост, но не эффективен. В качестве задачи перепишите эту функцию в стандарте Паскаля (без использования FileSize и Seek), и чтобы ответ находился за один просмотр файла.

11.6. Бестиповые файлы

Многие вещи нам непонятны не потому, что наши понятия слабы; но потому, что сии вещи не входят в круг наших понятий.

Козьма Прутков

Строгая типизация повышает надёжность программы, но ограничивает её применимость к похожим структурам данных. Например, в задачах копирования или сравнения файлов алгоритмы, в сущности, не зависят от конкретного типа компонент. Исходя из этого, во многих языках программирования допускается использование абстрактных (бестиповых) файлов. Тем более, что для самой файловой системы все файлы бестиповые (точнее, однотипные, просто последовательность компонент-байт). Например, рассмотрим фрагмент программы на языке Free Pascal

```
type Fdouble=file of double;
var FD: Fdouble; F: text; x: double;
begin
  Assign(FD,'AnyFile'); ReWrite(FD);
  repeat read(x); Write(FD,x) until x=0;
  Close(FD); Assign(F,'AnyFile'); ReSet(F);
  read(F,x) ⚠
```

Здесь мы записали файл вещественных чисел (во внутреннем машинном представлении, по 8 байт на число), закрыли его и затем открыли как текстовый файл и прочли лексему вещественного числа ⚠. Разумеется, чаще всего при этом ничего хорошего не получится, но надо ясно понимать, что файловой системе это «всё равно».

Бестиповые файлы описываются совсем просто, например

```
var F1,F2: file;
```

Рассмотрим, например, на языке Free Pascal задачу копирования файла F1 в файл F2 (старое содержимое F2 уничтожается):

```
type Buf=array[1..1000] of byte;
var  F1,F2: file; B: Buf; { буфер обмена }
     NumRead,NumWrite: word;
begin
  Assign(F1,'From.dat'); Assign(F2,'To.dat');
  Reset(F1,1); ReWrite(F2,1);
  { 1 – единица обмена (блок)=1 байт }
  repeat
    BlockRead(F1,B,1000,NumRead);
    BlockWrite(F2,B,NumRead,NumWrite)
{  обмен по 1000 блоков=1000 байт за раз }
  until  (NumRead<>1000) or (NumWrite<>NumRead);
  if NumWrite<>NumRead
    then Write('Мало памяти для записи')
    else Write('Успешно скопировано')
end.
```

Процедура BlockRead пытается читать в переменную B 1000 блоков (по одному байту) за раз, при этом в переменной NumRead возвращается число <u>на самом деле</u> прочитанных блоков. Последнее чтение вернёт длину прочитанных блоков `NumRead<>1000`, это признак конца цикла копирования. В редком случае длина файла F1 кратна числу блоков, тогда будет ещё одно чтение, которое вернёт `NumRead=0` и в файл F2 запишется 0 байт. Процедура BlockWrite записывает из переменной B в файл NumRead блоков (т.е. ровно столько, сколько считала процедура BlockRead). В параметре NumWrite возвращается число блоков, записанное в файл F2, когда `NumWrite<>NumRead`, то исчерпано место на диске (ну, или диск сломался 🐷). По хорошему, эту ошибку надо было бы обработать.

Вопросы и упражнения

Программировать – значит понимать.
Кристин Нюгард

1. Чем записи Паскаля отличаются от массивов?
2. Что такое инкапсуляция имён полей внутри записи?
3. Какие внешние (по отношению к полям записи) имена мы может использовать при описании этих полей?
4. Для чего нужен оператор присоединения?
5. Что такое вариантная часть записи и для чего она нужна?
6. Для чего в вариантой части записи предназначен селектор?
7. Почему элемент множества не является переменной?
8. Почему в Паскале мощность множества не может превышать 256?
9. Что такое базовый тип множества?
10. Опишите свою процедуру write, чтобы она выводила множества типа
    ```
    type SetB=set of byte;
    ```
    Все элементы множества выводить через пробел, длина одной строки не более 80 символов.
11. Чем динамическая структура данных отличается от статической?
12. Что такое внешний класс памяти?
13. Как программа может работать с компонентами файла?
14. Что такое файловая переменная, и чем она отличается от других переменных Паскаля?
15. Почему в программе нельзя читать значение файловой переменной?
16. Как файловые переменные связать с конкретными файлами?
17. Чем отличаются файлы от устройств ввода/вывода?
18. Какие Вы знаете способы доступа из программы к компонентам файла?

19. Какие Вы знаете режимы работы с файлом?
20. Для чего при работе с файлом может потребоваться буферизация?
21. Какова взаимосвязь между файлами и потоками ввода/вывода?
22. Какие файлы надо закрывать, и когда это нужно делать?
23. Из каких файлов нельзя читать данные процедурой `readln`?
24. Для каких файлов невозможен прямой доступ к их компонентам?
25. Как узнать, какой тип имеет файл в файловой системе?

---

[i] Для продвинутых читателей. Исследуем одну важную проблему обработки динамических структур данных, которые могут быть пустыми. Разберём для этого такую задачу. Пусть есть файл символов, это не текстовый файл типа `text`, а файл типа:

```
type FC=file of char;
```

Требуется реализовать функцию, которая возвращает самый «большой» (с наибольшим номером в алфавите) символ из этого файла. На этапе спецификации возникает проблема, что возвращать для пустого файла? Так как все символы алфавита «равноправны», то мы не можем выделить один из них как признак того, что файл был пустым. Эта проблемы общезначима для всех языков программирования, посмотрим, как она может быть решена.

1. Дополнительный параметр. Напишем требуемую функцию с заголовком:

```
function MaxChar(var f: FC; var Err: boolean): char;
```

Эта функция с побочным эффектом, она возвращает в параметре `Err` значение **`true`** для пустого файла. Придётся использовать эту функцию так:

```
c:=MaxChar(f,Err);
if Err then Write('Файл пуст')
       else Write('Max. символ=',c)
```

2. Переход к старшему типу результата. Напишем функцию с заголовком:

```
function MaxChar(var f: FC): integer;
```

Эта функция возвращает значение `-1` для пустого файла и `ord(c)` для максимального символа `c`. Эту функцию придётся использовать так:

```
var n: integer; c: char;
n:=MaxChar(f);
if n=-1 then Write('Файл пуст')
   else Write('Max. символ=',chr(n))
```

Такой приём популярен в языках C и C++, там целый и символьный типы просто совместимы по сравнению и присваиванию, фактически они не различаются.

3. Использование глобальной переменной. В системном блоке `System` языка Free Pascal описана глобальная для программы переменная с именем `ErrorCode` (код ошибки), в языке C аналогичная переменная имеет имя `ErrNo`, в ОС Windows определена такая же переменная с именем `GetLastError` и т.д. Функции, у которых есть рассматриваемые проблемы с возвращаемым результатом, будут иметь побочный эффект: они присваивают этой переменной ноль, если успешно завершили работу, и ненулевые значения для различных исключительных ситуаций. Пусть, например, для пустого файла выберем номер ошибки `-1`, тогда напишем функцию с заголовком:

```
function MaxChar(var f: FC): char;
```

Придётся использовать эту функцию так:

```
c:=MaxChar(f);
if ErrorCode<>0
   then Write('Файл пуст')
   else Write('Max. символ=',c)
```

4. Использование нового типа данных. Некоторые языки позволяют на базе существующего типа данных описать новый тип, в него добавлены одно или несколько «особых» значений, которые функция может возвращать при ошибке. Например, в языке Haskell на основании типа `Type` можно определить новый тип

```
data NewType Type = Error | Just Type
```

Теперь в типе `NewType` кроме всех значений типа `Type` появилась новая константа с именем `Error`, которую функция может возвращать при ошибке. Тогда, например, функция с именем `FirstListElement`, возвращающая первый элемент списка значений типа `Type` может быть описана как

```
FirstListElement [] = Error
FirstListElement (x,List) = Just x
```

Ключевое слово `data` позволяет в языке Haskell может определять и свои собственные новые типы данных (аналог перечислимого типа Паскаля), например

```
data Boolean = False | True
```

5. Возбуждение исключительной ситуации. Многие языки (например, Free Pascal, C++ и другие) позволяют функциям не завершать свою работу нормально (т.е. возвратом вычисленного результата), а возбуждать так называемую *исключительную ситуацию* (или просто исключение).

Например, на языке Free Pascal есть (именованные) стандартные исключения, которые определяет сама вычислительная система, например, деление на ноль (имя EZeroDivide), выход значения за допустимый диапазон (ERangeError), ошибка ввода/вывода (EInOutError) и т.д. (см. главу 17). Кроме того, программист может описать и своё исключение (в нашем случае поданный на обработку пустой файл). Далее в программе можно самому возбудить исключительную ситуация оператором **`raise`**.

При возникновении исключения в программе и/или функции программист может написать свою функцию для обработки этого исключения. Сейчас это самый универсальный путь (см. главу 17).