Содержание

Ввод и вывод: файлы

В первой лекции мы уже рассматривали ввод информации с клавиатуры и вывод ее на экран[1)]. Однако процесс ввода с консоли весьма трудоемок, а результат вывода на консоль - недолговечен. К счастью, существует более удобный способ записывать, хранить, пересылать и по необходимости считывать информацию из постоянной памяти компьютера. Для этого применяются файлы.

Что такое файл

В последнее время студенты все реже задают этот вопрос, однако на него все-таки стоит дать короткий ответ.

Файл - это самостоятельная последовательность символов, записанная в постоянную *память компьютера*.

В английском языке слово "`file`" имеет вполне понятный смысл: "вереница", что очень хорошо отражает внутреннюю структуру любого файла. Файл - это именно вереница символов, причем связанных в определенной последовательности: символы файла не могут по своему желанию перепрыгивать с одного места на другое.

"Самостоятельность" файлов заключается в том, что они не зависят от работы какой-либо программы. И даже если выключить компьютер, файлы будут продолжать свое существование на *винчестере* или на дискете.

Файлы могут хранить в себе все, что поддается кодированию:

- исходные тексты программ или входные данные (тесты);
- машинные коды выполняемых программ (игры, вирусы, обучающие и сервисные программы, др.);
- информацию о текущем состоянии какого-либо процесса;
- различные документы, в том числе и Интернет-страницы;
- картинки (рисунки, фотографии, видео);
- музыку;
- и т.д. и т.п.

Когда нужно использовать файлы

Выбор между консолью и файлами вам придется делать каждый раз, когда вы станете писать очередную программу.

Между тем ответ на вопрос, вынесенный в заголовок этого пункта, прост.

- Файлы полезны, если объем входных данных превосходит посильный при ручном вводе. (Крайним является случай, когда входные или *выходные данные* заведомо не могут поместиться в оперативной памяти.)
- Файлы нужны, если приходится многократно вводить одну и ту же информацию, с минимальными изменениями или вовсе без изменений (например, при отладке программы).
- Файлы необходимы, если нужно сохранять информацию о результатах работы программы, полученных при вводе различных входных данных (то есть: при поиске ошибок в программе).

Например, если вашей программе необходимо получить два или три числа (пять - уже многовато) или строку длиной символов десять, вы вполне можете задавать такие данные с клавиатуры вручную. Если же вам (а еще вероятнее - не вам, а некоему усредненному и потому посредственному оператору) придется вводить, скажем, массив чисел `10x10`, то вероятность ошибки при ручном вводе возрастает многократно. Значит, возможность этой ошибки нужно исключить: записать данные в файл, который легко отредактировать в случае необходимости. Кроме того, однажды созданный файл можно использовать многократно (может быть, с незначительными изменениями).

Разновидности файлов

В языке Pascal имеется возможность работы с тремя видами файлов:

- *текстовыми* ;
- *типизированными;*
- *нетипизированными*.

Последние два типа объединяются под названием ***бинарные***: информация в них записывается по байтам и потому недоступна для просмотра или редактирования в удобных для человека текстовых редакторах, зато такие файлы более компактны, чем *текстовые*.

В отличие от *бинарных*, *текстовые файлы* возможно создавать, просматривать и редактировать "вручную" - в любом доступном текстовом редакторе. Кроме того, при считывании данных из *текстового файла* нет необходимости заботиться об их преобразовании: в языке Pascal имеются средства автоматического перевода содержимого *текстовых файлов* в нужный тип и формат, и это позволяет сэкономить немало времени и сил.

Описание файлов

В разделе `var` переменные, используемые для работы с файлами (***файловые переменные***), описываются следующим образом:

```
var f1,f2: text;  {текстовые файлы}
   g: file of <тип_элементов_файла>;     {типизированные файлы}
   in, out: file; {нетипизированные файлы}
```

Файловая переменная не может быть задана константой.

Текстовые файлы

В этой лекции мы ограничимся рассмотрением только *текстовых файлов*, а о *типизированных* расскажем позже (см. лекцию 7).

Оперирование файлами

С этого момента и до конца лекции под словом "файл" мы будем подразумевать " *текстовый файл* " (разумеется, если специально не оговорено обратное). Однако многие описываемые ниже команды пригодны не только для *текстовых*, но и для *бинарных файлов*.

Назначение файла

Процедура `assign(f,'<имя_файла>');` служит для установления связи между *файловой переменной* `f` и именем того файла, за действия с которым эта переменная будет отвечать.

На разных этапах работы программы одной и той же *файловой переменной* можно присваивать разные значения. Например, если в начале программы мы напишем

```
assign(f,'input.txt');
```

то переменной `f` будет соответствовать файл, из которого производится считывание входных данных, вплоть до того момента, когда в программе встретится, скажем, команда

```
assign(f,'output.txt');
```

после которой переменной `f` будет уже соответствовать тот файл, куда выводятся результаты.

Строка `'<имя_файла>'` может содержать полный путь к файлу. Если путь не указан, файл считается расположенным в той же директории, что и исполняемый модуль программы. Именно этот вариант обычно считается наиболее удобным.

Открытие файла

В зависимости от того, какие действия ваша программа собирается производить с открываемым файлом, возможно троякое его открытие:

`reset(f);`	- открытие файла для считывания из него информации; если такого файла не существует, попытка открыть его вызовет ошибку и аварийную остановку работы программы. Эта же команда служит для возвращения указателя на начало файла;
`rewrite(f);`	- открытие файла для записи в него информации; если такого файла не существует, он будет создан; если файл с таким именем уже есть, вся содержавшаяся в нем ранее информация исчезнет;
`append(f);`	- открытие файла для записи в него информации (указатель помещается в конец этого файла). Если такого файла не существует, он будет создан; а если файл с таким именем уже есть, вся содержащаяся в нем ранее информация будет сохранена, потому что запись будет производиться в его конец.

Закрытие файла

После того как ваша программа закончит работу с файлом, очень желательно закрыть его:

```
close(f);
```

В противном случае информация, содержащаяся в этом файле, может быть потеряна.

Считывание из файла

Чтение данных из файла, открытого для считывания, производится с помощью команд `read()` и `readln()`. В скобках сначала указывается имя*файловой переменной*, а затем - список ввода[1)]. Например:

`read(f,a, b,c);`	- читать из файла `f` три переменные `a`, `b` и `c`. После выполнения этой процедуры указатель в файле передвинется за переменную `c`;
`readln(f,a, b,c);`	- читать из файла `f` три переменные `a`, `b` и `c`, а затем перевести указатель ("курсор") на начало следующей строки; если кроме уже считанных переменных в строке содержалось еще что-то, то этот "хвост" будет проигнорирован.

Если вспомнить, что в памяти компьютера любой файл записывается линейной последовательностью символов и никакой разбивки на строки там реально нет, то действия процедуры `readln()` можно пояснить так: **читать все указанные переменные, а затем игнорировать все символы вплоть до ближайшего символа "конец строки" или "конец файла"**. Указатель при этом перемещается на позицию непосредственно за первым найденным символом **"конец строки"**.

Если же *символ конца строки* встретится где-нибудь между переменными, указанными в списке ввода, то обе процедуры его просто проигнорируют.

Считывать из *текстового файла* можно только переменные простых типов: целых, вещественных, символьных, - а также строковых. Численные переменные, считываемые из файла, должны разделяться хотя бы одним *пробельным символом*. Типы вводимых данных и типы тех переменных, куда этиданные считываются, обязаны быть совместимыми[2)]. Здесь действуют все те же правила, что и при считывании с клавиатуры.

Считываемые переменные могут иметь различные типы. Например, если в файле[3)] `f` записана строка

```
1 2.5 c
```

то командой `read(f,a,b,c);` можно прочитать одновременно значения для трех переменных, причем все - разных типов:

```
a: byte;
b: real;
c: char;
```

Замечание: Обратите внимание, что символьную переменную с пришлось считывать дважды, так как после числа " `2.5` " сначала идет символ пробела и только затем буква " `c` ".

Из файла невозможно считать переменную составного типа (например, если `a` - массив, то нельзя написать `read(f,a)`, можно ввести его только поэлементно, в цикле), файлового, а также логического.

Особенно внимательно нужно считывать строки (`string[length]` и `string`): эти переменные "забирают" из файла столько символов, сколько позволяет им длина (либо вплоть до ближайшего конца строки). Если строковая переменная неопределенной длины (тип данных `string`) является последней в текущей строке файла, то никаких проблем с ее считыванием не возникнет. Но в случае, когда необходимо считывать переменную типа`string` из начала или из середины строки файла, это придется делать с помощью цикла - посимвольно. Аналогичным образом - посимвольно, от пробела до пробела - считываются из *текстового файла* слова.

Есть еще один, гораздо более трудоемкий способ: считать из файла всю строку целиком, а затем "распотрошить" ее - разобрать на части специальной процедурой *выделения подстрок* `copy()`. После чего (где необходимо) применить процедуру превращения символьной записи числа в само число, применяя стандартную процедуру `val()`. Кроме того, совсем не очевидно, что длина вводимой строки не будет превышать 256 символов, поэтому такой способ приходится признать неудовлетворительным.

Запись в файл

Сохранять переменные в файл, открытый для записи командами `rewrite(f)` или `append(f)`, можно при помощи команд `write()` и `writeln()`. Так же как в случае считывания, первой указывается *файловая переменная*, а за ней - *список вывода*:

`write(f,a,b,c);`	- записать в файл `f` переменные `a`, `b` и `c` ;
`writeln(f,a,b,c);`	- записать в файл `f` переменные `a`, `b` и `c`, а затем записать туда же символ " **конец строки** ".

Выводить в *текстовый файл* можно переменные любых *базовых типов* (вместо значений *логического типа* выведется их строковый аналог `TRUE` или`FALSE`) или строки.

Структурированные типы данных можно записывать только поэлементно.

Пробельные символы

К ***пробельным символам*** (присутствующим в файле, но невидимым на экране) относятся:

- символ горизонтальной табуляции (`#9`);
- символ *перевода строки* (`#10`) (смещение курсора на следующую строку, в той же позиции);
- символ вертикальной табуляции (`#11`);
- символ *возврата каретки* (`#13`) (смещение курсора на начальную позицию текущей строки; в кодировке UNIX один этот символ служит признаком конца строки);
- символ конца файла (`#26`);

- символ пробела (#32).

Замечание: Пара символов #13 и #10 является признаком конца строки *текстового файла* (в кодировках DOS и Windows).

В *(текстовом) файле* границами чисел служат *пробельные символы*, и при считывании чисел эти *пробельные символы* игнорируются, сколько бы их ни было. Таким образом, если ввод многих чисел производится с помощью команды read(), то нет никакой разницы, как именно записаны эти числа: в одну строку, в несколько строк или вовсе в один столбик. В любом случае считывание пройдет корректно и завершится только по достижении конца файла.

Если же считывание тестового файла производится посимвольно, то нужно аккуратно отслеживать *пробельные*(особенно концевые) символы.

Поиск специальных *пробельных символов* (нас интересуют в основном #10, #13 и #26) можно осуществлять при помощи стандартных функций:

eof(f) - возвращает значение TRUE, если уже достигнут *конец файла* f (указатель находится сразу за последним элементом файла), и FALSE в противном случае;

seekeof(f) - возвращает значение TRUE, если "почти" достигнут *конец файла* f (между указателем и концом файла нет никаких символов, кроме*пробельных*), и FALSE в противном случае;

eoln(f) - возвращает значение TRUE, если достигнут конец строки в файле f (указатель находится сразу за последним элементом строки), и FALSE в противном случае;

seekeoln(f) - возвращает значение TRUE, если "почти" достигнут конец строки в файле f (между указателем и концом строки нет никаких символов, кроме *пробельных*), и FALSE в противном случае.

Ясно, что в большинстве случаев предпочтительнее использовать функции seekeof(f) и seekeoln(f): они предназначены специально для *текстовых файлов*, игнорируют концы строк (и вообще все *пробельные символы*) и потому позволяют автоматически обработать сразу несколько частных случаев.

Например, если по условию задачи файл входных данных может содержать только одну строку, то правильнее всего будет написать программу, обрабатывающую все возможные варианты:

- одна строка, заканчивающаяся символом конца файла;
- одна строка, заканчивающаяся несколькими концевыми пробелами, а затем - символом конца файла;
- одна строка, заканчивающаяся *символом конца строки*, а затем - символом конца файла (на самом деле получается, что в файле содержится не одна, а две строки, но вторая - пустая);
- одна строка, заканчивающаяся несколькими концевыми пробелами, затем - *символом конца строки*, а затем - символом конца файла.

Поскольку функции seekeof() и seekeoln() при каждой проверке пытаются проигнорировать все *пробельные символы*, то и результаты их работы отличаются от результатов работы функций *eof*() и eoln(). Эти различия нужно учитывать.

Например, для входного файла f, состоящего из двух строк 1_2_ _3_#13#10 (всего 9 символов, вторая строка пустая, подчеркивания здесь обозначают пробелы), следующие куски программ будут выдавать такие результаты:

	Содержимое результирующего файла g	**Длина файла** g

`while not eof(f) do`
`begin read(f,c); {c: char}`
`  write(g,c)`
`end;`
	1_2_ _3_#13#10	9 байт

`while not seekeof(f) do`
`begin read(f,c); {c: char}`
`  write(g,c)`
`end;`
	123	3 байта

`while not eof(f) do`
`while not seekeoln(f) do`
`begin read(f,c); {c: char}`
`  write(g,c)`
`end;`
	Зацикливание	

`while not seekeof(f) do`
`while not eoln(f) do`
`begin read(f,c); {c: char}`
`  write(g,c)`
`end;`
	1_2_ _3_	7 байт

`while not seekeof(f) do`
`while not  eoln(f) do`
`begin read(f,k); {k: byte}`
`  write(g,k)`
`end;`
	1230	4 байта

Пример использования файлов

Задача. В *текстовом файле* `f.txt` записаны (вперемешку) целые числа: поровну отрицательных и положительных. Используя только один вспомогательный файл, переписать в *текстовый файл* `h.txt` все эти числа так, чтобы:

1. порядок отрицательных чисел был сохранен;
2. порядок положительных чисел был сохранен;
3. любые два числа, стоящие рядом, имели разные знаки.

Решение

Если бы нам разрешили использовать два вспомогательных файла, мы бы просто переписали все положительные числа в один из них, а все отрицательные - в другой. А затем объединили бы два этих файла. В нашем же случае придется переписать во вспомогательный файл только положительные числа. Затем при "сборке" мы будем считывать из вспомогательного файла "все подряд", а из исходного - только отрицательные числа.

Реализация

```
program z1;
var f,g,h: text;
    k: integer;
begin
           assign(f,'f.txt');
           assign(g,'g.txt');
           assign(h,'h.txt');
{Переписываем положительные числа в доп.файл}
           reset(f);
           rewrite(g);
           while not eof(f) do
                  begin read(f,k);
                            if k>0 then write(g,k,' ');
                  end;
{Собираем числа в новый файл h.txt}
           reset(f); {Возвращаем указатель на начало файла f}
           reset(g);
           rewrite(h);
           while not eof(g) do
                   begin read(g,k);
                                 write(h,k,' ');
                                 repeat
                                        read(f,k)
                                 until k<0;
                                 write(h,k,' ');
                  end;
   close(f);
   close(g);
   close(h);
end.
```

Изменение реакции на ошибку

По умолчанию любая ошибка ввода или вывода вызывает аварийную остановку работы программы. Однако существует возможность отключить такое строгое реагирование; в этом случае программа сможет либо игнорировать эти ошибки (что, правда, далеко не лучшим образом отразится на результатах ее работы), либо обрабатывать их при помощи системной функции `IOResult: integer`.

Директива компилятора[1] `{$I-}` отключает режим проверки, соответственно директива `{$I+}` - включает.

Если при отключенной проверке правильности ввода-вывода ({$I-}) происходит ошибка, то все последующие операции ввода-вывода игнорируются - вплоть до первого обращения к функции IOResult. Ее вызов "очищает" внутренний показатель ("флаг") ошибки, после чего можно продолжать ввод или вывод.

Если функция IOResult возвращает 0, значит, операция ввода-вывода была завершена успешно. В противном случае функция вернет номер произошедшей ошибки.

Пример использования директив {$I}

```
flag:= false;
write('Введите имя файла: ');
repeat
          readln(s);      {s:string}
{$I-}
          assign(f,s);
          reset(f);
          case IOResult of
                    0: flag:= true;
                    3: write('Путь к файлу указан неверно. Измените путь: ');
                    5: write('Доступа к файлу нет. Измените имя файла: ');
                  152: write('Такого диска нет. Измените имя диска: ');
                  else write('Такого файла нет. Измените имя файла: ');
          end;
   until flag;
{$I+}
```

Номер ошибки	Описание ошибки		Генерирующие процедуры [2)]
2	File not found	Файл не найден	*append*, *erase*, *rename*, reset, *rewrite*
3	Path not found	Директория не найдена	*append*, chdir, *erase*, mkdir, *rename*, reset, *rewrite*, rmdir
4	Too many open files	Открыто более 15 файлов одновременно	*append*, reset, *rewrite*
5	File *access denied*	Отказ в доступе к файлу	*append*, blockread, blockwrite, *erase*, mkdir, read, readln, *rename*,reset, *rewrite*, rmdir, write, writeln
12	*Invalid* file *access code*	Попытка использовать *текстовый файл* как *типизированный* или наоборот	*append*, reset
16	Cannot remove*current directory*	Невозможно удалить заданную директорию	rmdir
100	Disk read error	Попытка чтения после конца файла	read, readln
101	Disk write error	Ошибка записи на диск (диск полон)	close, write, writeln
102	File not assigned	Файл не назначен	*append*, *erase*, *rename*, reset, *rewrite*
103	File not open	Файл не открыт{бинарные файлы}	blockread, blockwrite, cl

			ose, *eof*, filepos, filesize, read, seek,write
104	File not open for input	Файл не открыт для ввода { *текстовые файлы* }	*eof*, eoln, read, readln, seekeof, seekeoln
105	File not open for output	Файл не открыт для вывода { *текстовые файлы*}	write, writeln
106	*Invalid* numeric format	Неправильный числовой формат { *текстовые файлы* }	read, readln
152	Drive not ready	Задано неверное имя диска	*append*, *erase*, *rename*, reset, *rewrite*