ГЛАВА 8

Методы работы с внешней памятью. Файлы

Где-то далеко в памяти моей…

Слова из песни

Внимательный читатель уже должен был отметить один известный методический принцип, формулируемый обычно как «повторенье – мать ученья», который авторы не раз применяли на протяжении книги. Действительно, на многие важные положения мы сознательно обращали внимание неоднократно, иногда в точности повторяя первоначальную формулировку, иногда чуть ее видоизменяя. Хотим подчеркнуть: мы отнюдь не являемся сторонниками усваивания информации путем «зубрежки», на таком подходе, на наш взгляд, далеко не уедешь. Понимание и умение формулировать – вот истинные показатели уровня владения материалом! К величайшему сожалению бесчисленных поколений школяров всех веков и эпох, до сих пор не придумано способа гарантированного запоминания необходимой человеку информации. Вот и разрабатываем мы различные методики, мнемонические правила, создаем целые школы тренировки памяти. Вот и изощряются фантасты – кто придумает наиболее эффектный и элегантный прибор, способный «затолкнуть» человеку в голову требуемые сведения. Ну а пока все эти сказки еще не стали былью, человечеству, как и во многих других случаях, пришлось искать решение в создании «дополнительного органа» для хранения информации – «внешней» памяти. Сначала наскальные рисунки, потом глиняные таблички и папирусы, затем пергаментные и бумажные книги, наконец, в последние полвека эту роль взяли на себя компьютеры. Сегодня средних размеров жесткий диск персонального компьютера способен вместить содержимое десятков тысяч книг. Впрочем, перевести информацию в электронный вид – только полдела. Гораздо важнее обеспечить возможность работать с этой информацией в привычном человеку виде. Как мы отмечали в главе 3, общепринятый способ организации информации в долговременной памяти – файлы. Работа с файлами составляет существенную часть необходимых умений квалифицированного программиста. В данной главе мы рассмотрим основные приемы работы с файлами, методы чтения и записи информации, а также соответствующий аппарат языка Pascal.

1. Файлы: основные понятия

Не вдаваясь в подробности физического устройства гибких, жестких, CD- и DVD-дисков, флэш-драйвов, ZIP- и ZIV-дисков, стримеров и других внешних носителей, заметим, что с общих позиций принципы представления информации в них и принципы доступа к ней идентичны. В любом устройстве информация представляет собой двоичные последовательности, доступ к которым осуществляется по физическим адресам. Производители устройств создают специальные программы – драйверы, обеспечивающие выполнение всех необходимых низкоуровневых операций, а разработчики операционных систем предоставляют прикладным программистам высокоуровневый интерфейс, позволяющий оперировать с данными в терминах файлов. Напоминаем: под *файлом* понимается поименованный набор данных, размещенный во внешней[1] памяти компьютера. Очевидно, что над файлом нужно уметь производить как минимум два основных действия: писать в него и читать из него. Также очевидно, что файл должен иметь некоторую внутреннюю структуру, отражающую содержание хранимой в нем информации. Разберем эти моменты более подробно.

1.1. Записи файлов

Структурную единицу информации внутри файла принято называть *записью* (не следует путать это понятие с типом данных «запись» языка Pascal). Разделение файла на записи производится по двум причинам, первая из которых связана с физическими принципами представления данных и обмена информацией между оперативной и внешней памятью, а вторая – с тем или иным взглядом на логическую структуру информации файла.

Физические записи (их еще иногда называют *блоками*) представляют собой атомарную часть файла с точки зрения обмена с оперативной памятью. Другими словами, за один раз записывается на диск или считывается с диска ровно одна физическая запись. Количество физических записей в файле и размер каждой из них в общем случае задаются программистом средствами языка программирования.

---

[1] В данном случае под «внешней памятью» мы понимаем любые средства хранения информации, кроме оперативной памяти.

Все записи файла могут иметь либо одинаковый размер, то есть одинаковое число байт, – так называемые *записи фиксированной длины*, либо иметь разные размеры – *записи переменной длины*.

В различных языках программирования реализованы разные способы задания характеристик файла. Смысл вариаций состоит в том, что либо программисту даются средства прямо указать размеры записей и их число (и это он должен делать при объявлении каждого файла), либо применяется система умолчаний, при которой характеристики файла связываются с его типом. В языке Pascal можно найти как первый способ – так называемые *бестиповые файлы*, так и второй – *текстовые* и *типизированные файлы*, определения физических характеристик файла.

Разбиение файла на *логические записи* зависит от структуры информации. Это разбиение явным образом не отражается в представлении данных на носителе, а фиксируется в алгоритме обработки файла. При этом в различных контекстах обработки строение файла может считаться разным. Скажем, в файле, содержащем информацию о фамилиях, должностях и зарплатах сотрудников некоторого предприятия, в качестве логической записи в одном случае может рассматриваться строка из трех полей, а в другом нас могут интересовать лишь фамилия и должность конкретного работника.

В зависимости от потребностей обработки возможны различные соотношения между логическими и физическими записями. Во-первых, они могут совпадать. Во-вторых, одна логическая запись может представлять последовательность физических записей – в этом случае говорят, что логическая запись сегментирована. И, наконец, возможно, что в одной физической записи размещается несколько логических записей – это называется блокированием логических записей. Некоторые языки программирования имеют средства явной установки соотношения между физическими и логическими записями. В языке Pascal такого аппарата фактически нет, что ни в коей мере не затрудняет программирование, так как эта проблема решается сама собой выбором того или иного типа файла.

1.2. Физический и логический файл. Связывание

В зависимости от контекста употребления термин «файл» может иметь две различные смысловые окраски. Проиллюстрируем их на примере.

Допустим, наша программа предназначена для бухгалтерских расчетов. Одним из объектов обработки программы является список сотрудников предприятия, включающий фамилии и заработные платы. Указанная

информация по ряду подразделений хранится в виде файлов. Когда мы говорим о файле данных по конкретному подразделению, мы имеем в виду конкретный список фамилий и значений зарплаты, размещенный на конкретном физическом носителе и имеющий заданное имя. В этом случае часто употребляют термин *физический файл*. В то же время для составления программы бухгалтерских расчетов существенны не конкретные фамилии сотрудников и их заработные платы, а структура файла и формат его компонент, то есть логическое описание файла. В этом контексте говорят о *логическом файле*. Здесь вполне уместна аналогия с константой – физический файл – и переменной: логический файл.

Программа, ориентированная на обработку некоторого логического файла, в процессе выполнения имеет дело с одним из его экземпляров, то есть физическим файлом. Перед началом работы программы должно быть осуществлено ее *связывание* с конкретным физическим файлом. Процедуру связывания часто называют *открытием* физического файла. В открытие файла входит, в частности, поиск его на диске.

В процессе работы программа может завершить обработку одного физического файла и перейти к работе над другим аналогичным файлом. Для этого требуется «отключить» программу от первого файла или, как говорят, *закрыть* этот файл и открыть новый.

1.3. Методы доступа

Итак, мы выяснили, что работа с файлами заключается в выполнении операций над их записями, соответственно одним из существенных вопросов в этой работе является вопрос о способе доступа к записям. В принципе здесь возможны два подхода.

Работа в режиме так называемого *последовательного доступа* означает возможность выбора очередной записи только после обработки записи, физически расположенной перед ней. Здесь можно провести аналогию с воспроизведением очередной песни, записанной на магнитофонной кассете. Доступ к ней можно получить либо проиграв предыдущую, либо осуществив ее перемотку. Способ доступа, конечно же, диктуется алгоритмом решения конкретной задачи, многие из них требуют именно последовательной обработки данных. Кроме того, последовательный метод доступа обладает важным достоинством: он не требует указания адреса очередной записи и поиска ее на носителе.

Более универсальным методом доступа к записям файла является так называемый *прямой доступ*. Он означает возможность выбора в любой момент произвольной записи файла вне зависимости от ее местоположения

и того, какая запись обрабатывалась до этого. Продолжая «музыкальную» аналогию, этот способ доступа к информации можно сравнить с выбором песни на аудиодиске. Заметим, что при этом сохраняется возможность и последовательного прослушивания песен без указания местоположения (адреса) очередной песни.

Прямой доступ к записям является универсальным методом, на основе которого может быть запрограммирован любой алгоритм обработки файлов. Очевидно, реализация прямого доступа требует наличия в языке программирования средств «именования» записей и указания требуемой записи по ее «имени». В языке Pascal, разумеется, реализованы оба рассмотренных нами метода доступа, при этом в качестве имен записей выступают номера.

1.4. Доступ на чтение и на запись

Базовые операции в работе с файлами – чтение и запись данных. Давайте подумаем вот над каким вопросом: возможно ли в рамках одной «сессии» работы с файлом (от открытия до закрытия) поочередное выполнение этих операций? Вопрос не так прост, как может показаться. Ответ на него неоднозначен и, в целом, связан со способом организации работы с файлами и соглашениями, принятыми в конкретной системе программирования.

Поговорим для начала о модификации файла, то есть добавлении к нему новых записей и удалении ставших ненужными. Если удалять данные из файла не требуется, то и с добавлением никаких проблем не возникнет. Дописываем новые записи в конец файла, пока на диске есть место, – все в порядке. А вот если данные требуется в процессе работы удалять?

Во-первых, надо понять, а что значит «удалить запись»? Или, другими словами, чем «удаленная» запись отличается от «нормальной»? Отличить записи по содержимому возможно только в частных случаях. Например, если записи хранят числа и из задачи известно, что допустимы лишь положительные значения, то в удаленные записи можно разместить, скажем, число –1. В общем случае таких «выделенных» значений, которые можно использовать в качестве признаков, конечно же, нет. Выходов два: либо запоминать номера удаленных записей, что имеет смысл только при регулярной структуре файла (если записи переменной длины, номера смысла не имеют), либо в каждой записи размещать служебную информацию – признак удаленности.

Допустим, с удалением мы разобрались, теперь, во-вторых, надо решить, каким образом мы будем добавлять новые записи. Можно, как и ранее, дописывать их в конец, что естественным образом приведет к «разбуханию» файла и лишним затратам дискового пространства. Можно попробовать использовать освободившееся место в середине файла. Снова

получаем две ситуации: если записи постоянной длины, все в порядке – одну удалили, на ее место в точности «войдет» другая, при переменной же длине может оказаться, что свободное место внутри файла есть, но нет ни одного достаточного по размеру непрерывного фрагмента. Потребуется перепаковка – сдвиг фрагментов в начало файла, а значит, затраты времени.

Следствием рассмотренных сложностей является тот факт, что во многих языках программирования, и в том числе в языке Pascal, на файлы, допускающие модификацию во время работы, накладывается ограничение – такие файлы должны состоять из записей одинаковой длины.

Что касается файлов с записями переменной длины, то в связи с их нерегулярной структурой работа с ними, кроме частных случаев, возможна только в режиме последовательного доступа. По этой же причине «последовательные» файлы в рамках одной сессии могут использоваться либо только в режиме чтения, либо только в режиме записи.

2. Виды файлов в Object Pascal

В языке Pascal файлы разбиты на три категории по областям применения.

В особый класс выделены *текстовые файлы*. Это последовательные файлы с записями переменной длины, содержащие кроме текста специальные символы типа «переход на следующую строку». Важно понимать, что текстовый файл не является файлом «строк», то есть содержащим записи из констант типа `String`. Такой файл тоже можно создать, но он уже будет относиться к другому виду файлов. Для объявления текстовых файлов используется предопределенный идентификатор типа `Text`.

Типизированные файлы – это файлы прямого доступа с записями фиксированной длины. Каждая запись этого файла представляет собой константу одного общего для всего файла типа данных. Этот вид файлов можно рассматривать как основной и наиболее универсальный. Конкретный вид типизированного файла задается при объявлении типа, которое записывается следующим образом:

```
type
  <имя файлового типа> = file of <тип записи>;
```

например

```
type
  FileOfInteger = file of Integer;
```

Третий вид файлов – *бестиповые файлы*. Бестиповые файлы предназначены для программирования, что называется, на физическом уровне, то есть близком к уровню машинного языка. Они применяются в случае, когда требуется эффективная по времени и памяти реализация процедур обмена, такая, что можно пожертвовать наглядностью программы и дополнительными усилиями по программированию обмена. Кроме того, такая организация файла применяется, когда производится обмен без обработки информации, например в процедуре копирования. Бестиповые файлы объявляются следующим образом:

```
type
  <имя файлового типа> = file;
```

например

```
type
  FileToCopy = file;
```

Для работы с конкретным физическим файлом в программе требуется создать файл логический – объявить переменную файлового типа. Например,

```
var
  T: Text;
  FInt: FileOfInteger;
  FC: FileToCopy;
```

3. Операторы связывания логического и физического файлов

Средства работы с физическими файлами в языке Pascal отражают соответствующие возможности операционных систем типа DOS и семейства Microsoft Windows. Как известно, в этих системах физический файл идентифицируется путем доступа к нему и именем. Например,

```
C:\Temp\Example1\Main.pas
A:\BOOK\CHAPTER7.TXT
```

Связывание имен логического и физического файла производится процедурой языка Object Pascal, обращение к которой имеет следующий вид:

```
AssignFile(<имя логического файла>,
  <имя физического файла>);
```

Здесь `имя логического файла` есть ранее объявленная файловая переменная, а `имя физического файла` может быть представлено строковым выражением, задающим в конечном итоге путь к файлу.

Заметим, что эта процедура – одна из немногих, чье наименование изменилось по сравнению с предыдущими версиями языка[1]. Из соображений обратной совместимости можно использовать форму процедуры с именем Assign, однако делать это не рекомендуется.

Примеры обращения к процедуре:

```
AssignFile(T, 'A:\BOOK\CHAPTER7.TXT');
AssignFile(FC, Path + FileName);
```

Во втором примере предполагается, что предварительно сделаны объявления вида:

```
var
   Path: String[67];
   FileName: String[12];
```

и этим переменным присвоены некоторые значения.

Заметим, что в общем случае задание имени физического файла с помощью переменной предпочтительнее, чем с помощью константы. Записывая имя-константу, мы жестко привязываем программу к конкретному имени физического файла. Если же имя задано переменной, то его можно, например, вводить в диалоге с пользователем программы.

Процедура AssignFile носит, в определенном смысле, декларативный характер и не выполняет всех действий по открытию файла, в частности не производит поиск файла. Это делается отдельными процедурами Reset или Rewrite. Первая из упомянутых процедур открывает уже существующий файл, вторая – создает новый (пустой) файл. Существуют некоторые особенности использования этих процедур для открытия каждого из видов файлов (текстовых, типизированных и бестиповых). В этом разделе мы рассмотрим общие для всех видов файлов аспекты работы процедур открытия.

Обращения к процедурам открытия файла имеют вид:

```
Reset(<имя логического файла>);
Rewrite(<имя логического файла>);
```

например,

```
Reset(T);
Rewrite(FInt);
```

Процедура Reset осуществляет поиск файла с физическим именем, связанным в данный момент с файловой переменной, и подготавливает его

[1] Вторая процедура – CloseFile. В Borland Pascal 7.0 она называлась Close. В Object Pascal рекомендуется использовать новый вариант – CloseFile.

к работе. Если файл с заданным именем не найден, то в стандартной ситуации выдается сообщение об ошибке. Чтобы избежать выдачи маловразумительного для пользователя-непрограммиста сообщения, обычно применяют прием, основанный на использовании директивы {$I-}, позволяющей отказаться от вставки в исполняемый модуль команд, контролирующих выполнение процедур обмена. При этом, как мы уже не раз отмечали, для получения информации о результате операции можно использовать функцию IOResult. Ниже приведен фрагмент программы, запрашивающий имя файла до тех пор, пока не будет введено имя существующего на диске набора данных.

```
{ ================================================== }
{ Пример 8.1 }
{ Открытие файла }
program OpenFile;
var
   T: Text;
   FileName: String[80];

begin
   repeat
      Write('Введите имя файла: ');
      Readln(FileName);
      {$I-}
      AssignFile(T, FileName);
      Reset(T);
      {$I+}
   until IOResult = 0;
   . . .
end.
{ ================================================== }
```

Процедура создания нового файла Rewrite таит в себе опасность другого сорта. Если на диске уже существует файл с именем вновь открываемого файла, то старый набор данных будет уничтожен без какого-либо предупреждения. Таким образом, пользователь может случайно уничтожить полезную информацию, если он проявил невнимательность при выборе имени нового файла. Поэтому в общем случае при создании нового файла полезно проверить наличие файла с таким именем и предупредить об этом пользователя.

Рассмотрим пример программы, осуществляющей предварительную проверку наличия файла с именем, совпадающим с именем вновь создаваемого файла. Программа пытается открыть файл процедурой Reset и в случае успеха запрашивает у пользователя разрешение на удаление старого файла. Если

пользователь не дает разрешения, программа запрашивает новое имя для создаваемого файла. В примере использована функция UpCase(<символ>). Она преобразует буквы из строчных в прописные. Если аргумент не является строчной буквой, то результат совпадает с аргументом. Смысл применения функции – освободить пользователя от необходимости переключать регистр при ответе на запрос об удалении файла.

```
{ ================================================= }
{ Пример 8.2 }
{ Проверка наличия файла с именем вновь создаваемого файла }
program OpenWithCheck;
var
   T: Text;
   FileName: String[80];
   Answer: Char; { Ответ на вопрос об удалении файла }
   Exists: Boolean; { Признак наличия файла }
begin
   repeat
      WriteLn('Введите имя файла: ');
      ReadLn(FileName);
      {$I-}
      AssignFile(T, FileName);
      Reset(T);
      {$I+}
      Exists := IOResult = 0;
      if Exists then
      begin
         CloseFile(T); { Закрытие существующего файла }
         Write('Такой файл уже есть. Удалить? (Y/N) ');
         { Ввод кода клавиши, гарантирующий реакцию только на }
         { допустимый ответ: Y или y (Да), либо N или n (Нет) }
         repeat
           ReadLn(Answer);
         until (UpCase(Answer) = 'Y') or (UpCase(Answer) = 'N');
         WriteLn(Answer);
      end;
   until (not Exists) or (UpCase(Answer) = 'Y'); Rewrite(T);
   . . .
end.
{ ================================================= }
```

В вышеприведенном примере мы использовали процедуру закрытия файла, обращение к которой имеет следующий общий вид:

```
CloseFile(<имя логического файла>);
```

Закрытие файла следует производить перед переключением файловой переменной с одного физического файла на другой или перед завершением работы программы.

4. Текстовый файл

Текстовый файл представляет собой последовательный файл с записями переменной длины. В данном случае записи – это строки текста. Разделяются записи между собой специальным признаком «конец строки», формирующимся, например, при нажатии клавиши `Enter`.

При открытии файла определяется способ его обработки. Файл, открытый процедурой `Reset`, предназначен только для чтения его записей. В файл, созданный процедурой `Rewrite`, можно только записывать информацию.

Модификация уже существующего текстового файла возможна только путем добавления новых записей в конец файла. Для этого он должен быть открыт специальной процедурой

```
Append(<имя логического текстового файла>);
```

Мы уже практически знакомы с операциями обработки текстового файла. Дело в том, что ввод с клавиатуры – это не что иное, как ввод записей текстового файла, связанного с физическим файлом, «размещенным» на клавиатуре. Вывод же на дисплей – это вывод в текстовый физический файл, «размещенный» на дисплее. Эти стандартные файлы имеют предопределенные в библиотеке `System` (которая, напомним, всегда автоматически подключается к программе) имена: `Input` – для входного файла и `Output` – для выходного. Заметим, что имеется возможность связать эти логические файлы и с другими устройствами, например, назначив вывод из текстового файла `Output` на диск.

Общая форма записи операторов чтения и записи включает в себя упоминание имени логического файла:

```
Read([<имя логического текстового файла>,]
  <список ввода>);
Readln([<имя логического текстового файла>,]
  <список ввода>);
Write([<имя логического текстового файла>,]
  <список вывода>);
Writeln([<имя логического текстового файла>,]
  <список вывода>);
```

Если имя логического файла не указано, подразумевается `Input` для операторов чтения и `Output` для операторов записи.

Обратим еще раз внимание на важную особенность операторов обмена с текстовым файлом. Они обеспечивают автоматическое преобразование информации из текстового вида к типу переменных, стоящих в списке, при вводе и из внутреннего формата представления данных в текстовый вид при выводе.

Если, обрабатывая некоторый ранее созданный текстовый файл, мы не знаем заранее количество записей-строк, как же определить, когда файл закончится? Проблема состоит в том, что по самому файлу определить, сколько в нем строк, невозможно – в языке Pascal операции для этого не предусмотрено. Все, что нам, программистам, доступно, – это функция

```
Eof(<имя логического файла>);
```

которая возвращает значение **`True`** при достижении конца файла.

Аналогичного сорта проблема может возникнуть и при чтении отдельной записи. Составляя программу, разработчик должен знать формат каждой строки обрабатываемого файла (например, сколько в ней чисел, в каком формате эти числа представлены), чтобы правильно использовать оператор чтения. Если отсутствует информация о количестве элементов строки, то можно воспользоваться функцией

```
Eoln(<имя логического текстового файла>);
```

которая принимает значение **`True`**, если встретился признак «конец строки».

Заметьте, что в соответствии с описанием функция `Eof` применима не только к тестовым, но и к файлам других видов, в отличие от функции `Eoln`.

Вариантами этих функций, полезными при чтении числовой информации из текстового файла, являются булевские функции

```
SeekEof(<имя логического текстового файла>);
```

и

```
SeekEoln(<имя логического текстового файла>);
```

Первая из них выдает сообщение о конце файла, если в последних строках нет символов, отличных от пробелов или символов табуляции. Вторая функция сообщает о конце строки, если в ней не осталось непрочитанных символов, отличных от пробелов и табуляций.

Рассмотрим пример использования этих языковых средств. Допустим, с помощью текстового редактора подготовлен текстовый файл, содержащий вещественные числа. По некоторым причинам числа размещены в «свободном формате», то есть строки содержат разное количество чисел, содержат пробелы в конце строк, пустые строки могут встретиться в любом месте файла. Пример такого текста приведен ниже:

```
-2.3 34.5 3 45 45.56

34 -4.67
36.90

23 4 -89 4.67
12.45 4 567
-56.7

4
```

Требуется отформатировать этот текст, поместив на каждой строке только по три числа и удалив все лишние пробелы и пустые строки, то есть получить текст вида:

```
-2.30       34.50       3.00
45.00       45.56       34.00
-4.67       36.90       23.00
4.00        -89.00      4.67
12.45       4.00        567.00
-56.70      4.00
```

Эту задачу решает следующая программа:

```
{ ================================================== }
{ Пример 8.3 }
{ Форматирование текстового файла }
Program Format;
var
   Old, { исходный файл }
   New: Text; { результирующий файл }
   Count: Word;
   X: Real;
begin
   AssignFile(Old, 'OLD.TXT');
   Reset(Old);
   AssignFile(New, 'NEW.TXT');
   Rewrite(New);
   Count := 0;
   while not SeekEof(Old) do
      { пока в файле есть непустые строки }
   begin
      while not SeekEoln(Old) do { пока в строке есть числа }
      begin
         Inc(Count);
         Read(Old, X);          { читаем одно число из строки }
         Write(New, X:8:2, ' '); { записываем это число     }
         if Count = 3 then      { если записали 3 числа      }
         begin                  { то                         }
```

```
          Count := 0;
          WriteLn(New);       { переходим на новую строку }
        end;
      end;
    end;
    CloseFile(Old);
    CloseFile(New);
end.
{ ================================================ }
```

5. Типизированный файл

Типизированные файлы являются файлами прямого доступа с фиксированной длиной записи, определяемой типом записи при объявлении файла. Тип записи – это один из рассмотренных нами типов языка Pascal, за исключением файлового типа. Другими словами, нельзя создать «файл файлов», однако можно создать, например, такой экзотический файл, как «файл процедур», в котором будут храниться значения переменных процедурного типа. Примеры объявления конкретных файловых типов:

```
type
   FileOfReal = file of Real;

   Matrix = array[1..10, 1..20] of Word;
   FileOfMatrix = file of Matrix;

   Person = record
      Name: String;
      Age: Byte;
   end;
   PersonList = file of Person;
```

Типизированный файл может обрабатываться как в режиме последовательного, так и в режиме прямого доступа к записям. Во втором случае операторы обмена должны знать адрес требуемой записи. Как уже отмечалось, адресация записей производится по их порядковым номерам, которые присваиваются записям в процессе создания файла. Условно говоря, с файлом связан «курсор» – указатель позиции в файле, который показывает на текущую обрабатываемую запись. После открытия существующего файла курсор указывает на первую запись. Выполнение последовательных операторов обмена производит перемещение курсора на следующую по порядку запись. Выполнение операций обмена с прямым доступом переносит курсор на запись с номером, указанным в этом операторе.

Обмен с текущей (на которую указывает курсор) записью производится операторами:

```
Read(<имя логического файла>, <список ввода>);
Write(<имя логического файла>, <список вывода>);
```

Списки ввода и вывода должны состоять по крайней мере из одной переменной, принадлежащей к тому же типу, что и все записи файла. После выполнения этих операторов курсор передвигается на следующую запись.

Для осуществления прямого доступа используется процедура установки курсора на запись с заданным номером:

```
Seek(<имя логического файла>, <номер записи>);
```

Второй параметр процедуры представляет собой выражение типа `Longint`.

Чтобы определить номер записи, на которую в данный момент указывает курсор, используется функция

```
FilePos(<имя логического файла>);
```

Типизированные файлы допускают их модификацию путем замены любых записей, добавления новых с номерами, большими, чем максимальный текущий номер записи, а также удаления записей. При этом неважно, каким образом был открыт файл – был ли он вновь создан процедурой `Rewrite` или был открыт существовавший ранее файл процедурой `Reset`. В любом случае после открытия файла можно как читать записи, так и записывать в него информацию.

Рассмотрим более подробно приемы модификации типизированного файла. Поскольку все записи файла принадлежат одному типу, то замена записи производится просто: следует установить курсор на заменяемую запись и осуществить вывод в нее новой информации.

Вставка новых записей в файл возможна только путем добавления их в конец файла. Другими словами, нельзя вставить новую запись между двумя уже существующими записями («раздвинуть» записи). Чтобы добавить в конец файла новую запись, следует в процедуре `Seek` указать номер последней записи файла. Это число можно узнать, воспользовавшись функцией

```
FileSize(<имя логического файла>);
```

которая возвращает количество записей в типизированном файле (заметим, что она неприменима к текстовым файлам). Таким образом, пара процедур:

```
Seek(F, FileSize(F));
Write(F, Info);
```

добавляет в конец файла `F` новую запись с информацией из переменной `Info`.

Интересно, что номер добавляемой записи не обязательно должен быть на единицу большим общего числа записей в файле. Например, если файл имеет 50 записей, мы можем поместить в него запись с номером 60. При этом будут созданы и промежуточные 9 записей с номерами от 51 до 59, которые могут быть заполнены полезной информацией позже.

Удаление записей также возможно только в конце файла, то есть методом «отсечения» его «хвостовой» части. Для этого предназначена процедура

```
Truncate(<имя логического файла>);
```

которая отсекает все записи файла, начиная с той, на которую указывает в данный момент курсор.

Следует обратить внимание на одну важную особенность обмена с типизированными файлами. Обмен информацией между внешней и оперативной памятью всегда происходит без преобразования данных. При записи информация заносится на диск в своем внутреннем представлении. Таким образом, содержимое типизированного файла в общем случае нельзя посмотреть с помощью текстового редактора, точнее, посмотреть, конечно, можно, а вот понять – вряд ли. Отсутствие преобразования информации к текстовому виду при записи в типизированный файл существенно ускоряет процесс обмена.

В заключение обзора средств работы с типизированными файлами отметим, что функция определения конца файла `Eof` и процедура закрытия файла `CloseFile` также применяются для их обработки.

Рассмотрим пример работы с типизированным файлом.

```
{ ================================================== }
{ Пример 8.4 }
{ Создание и модификация типизированного файла }
Program TypedFile;
type
   Person = record { строка ведомости }
      Name: String[20];
      Salary: Word;
   end;
      PersonFile = file of Person;
var
   P: Person;
   F: PersonFile;
   S: String[80]; { буфер для вводимой текстовой строки   }
   StarPos,       { номер позиции строки, где находится * }
   Code: Integer; { код завершения процедуры Val           }
    { (не анализируется в данной программе)               }

{ Преобразование входной текстовой строки к типу Person }
```

```
procedure Perform;
begin
   StarPos := Pos('*', S);
   P.Name := Copy(S, 1, StarPos - 1);
   Val(Copy(S, StarPos + 1, Length(S) - StarPos), P.Salary,
     Code);
end; { Perform }

{ Заполнение типизированного файла }
procedure Create;
var
   T: Text;
begin
   AssignFile(T, 'LIST.TXT');
   Reset(T);
   while not SeekEof(T) do
   begin
      ReadLn(T, S);
      Perform;
      Write(F, P);
   end;
   CloseFile(T);
end; { Create }

{ Вывод ведомости на экран с печатью итоговой суммы }
procedure Display;
var
   Sum: Word;
begin
   Reset(F);
   Sum := 0;
   with P do
   begin
      while not Eof(F) do
      begin
         Read(F, P);
         Write(Name, ' ');
         WriteLn(Salary);
         Sum := Sum + Salary;
      end;
   end;
   Writeln(' ', 'Итого: ', Sum);
end; { Display }

{ Замена или добавление строки }
procedure ChangeOrAdd;
var
```

```
   N, i: Integer;
begin
   WriteLn('Введите новую строку');
   Readln(S);
   Val(Copy(S, 1, Pos('.', S) - 1), N, Code);
   if N > FileSize(F) then
   begin
      P.Salary := 0;
      for i := FileSize(F) + 1 to N - 1 do



      begin
         Str(i, P.Name);
         P.Name := P.Name + '.';
         Write(F, P);
      end;
   end;
   Perform;
   Seek(F, N - 1);
   Write(F, P);
end; { ChangeOrAdd }

begin
   AssignFile(F, 'LIST.PSL');
   Rewrite(F);
   Create;
   Display;
   Readln;
   ChangeOrAdd;
   Display;
   Readln;
end.
{ ================================================ }
```

Приведенная программа включает процедуру создания типизированного файла (информация читается из заранее подготовленного текстового файла), процедуру вывода на экран содержимого файла и процедуру замены или добавления новой записи в файл. Первые две процедуры работают с файлом в режиме последовательного доступа, третья – в режиме прямого доступа.

Информация, заносимая в файл, представляет собой «ведомость» на получение заработной платы. Первоначально ведомость подготавливается с помощью текстового редактора в следующем виде:

```
1. Иванов И.И.*5000
2. Петров П.П.*6500
```

```
3. Сидоров С.С.*7000
4. Королев Н.Н.*5500
```

Каждая строка начинается с номера, отделяемого от остальной части точкой. Номер строки используется при прямом доступе к записям. Фамилия сотрудника отделяется от его заработной платы с помощью символа « * ».

При выводе ведомости на экран печатается итоговая сумма по заработной плате. Для этого требуется представление заработной платы в числовом виде.

Замена или добавление новой строки ведомости производится по следующим правилам. Запрашивается новая строка, которая должна быть введена в вышеуказанном формате. Если номер строки не превышает числа строк в ведомости, то будет заменена соответствующая строка. Если номер строки больше числа строк в ведомости, то эта строка будет добавлена в файл. Причем допускается вводить строки с номерами, более чем на единицу превышающими число строк в ведомости. При этом будут созданы все промежуточные строки, в которые будут помещены только их номера. Например, добавление строки

```
7. Кузнецов А.А.*8000
```

к вышеуказанному файлу приведет к появлению на экране (в случае вызова процедуры печати) текста:

```
1. Иванов И.И. 5000
2. Петров П.П. 6500
3. Сидоров С.С. 7000
4. Королев Н.Н. 5500
5. 0
6. 0
7. Кузнецов А.А. 8000
Итого: 32000
```

В программе с целью сокращения объема опущены всевозможные проверки, обеспечивающие надежность работы. Например, не проверяется наличие текстового файла, не контролируется синтаксис вводимых строк и тому подобное.

6. Бестиповые файлы

Текстовые файлы – наиболее близкий и понятный человеку формат представления информации, файлы типизированные отражают тот факт, что человек любую информацию, с которой приходится работать, стремится упорядочить, структурировать. Именно поэтому два этих вида наиболее распространены. Однако существует ряд задач, в которых либо внутреннее устройство файла не имеет значения, например упоминавшаяся

уже задача копирования файла, либо интерпретация содержимого может меняться в зависимости от потребности, скажем, в одной ситуации запись обрабатывается в программе как строка символов, а в другой в ней требуется выделять числовые поля. Во всех этих случаях использование текстового или типизированного вида файлов является не слишком удобным. На помощь приходит бестиповый файл.

Бестиповые файлы, объявляемые как

```
type
  <имя файлового типа> = file;
```

являются файлами с прямым доступом и записями постоянной длины.

Открывая бестиповый файл, программист может указать длину записи файла в байтах. Это относится и к ранее созданным файлам, причем не требуется, чтобы указываемая длина совпадала с той длиной записи, которая указывалась (явно или неявно) при создании открываемого файла. Если длина записи не указана при открытии, то она по умолчанию принимается равной 128 байт. Задание длины записи производится в уже известных нам операторах открытия файла `Reset` и `Rewrite`, где она указывается вторым параметром. Например, процедура

```
Reset(F, 256);
```

открывает существующий файл F и задает длину его записи в 256 байт.

Процедура

```
Rewrite(F, 1);
```

создает новый файл с длиной записи, равной 1 байту.

Для ускорения обмена передаваемые записи могут блокироваться в одну физическую запись-блок. На необходимость блокирования записей указывают параметры процедур обмена, которые для бестиповых файлов имеют особую форму, не совпадающую с формой записи операторов обмена, рассмотренных нами ранее.

Оператор чтения блока с диска в оперативную память имеет вид:

```
BlockRead(<имя логического файла>, <буфер>,
  <число записей> [, <число прочитанных записей>]);
```

Размер читаемого блока определяется как произведение третьего параметра на размер записи. Второй параметр задает область памяти, в которую помещается блок. Это должна быть переменная, число байт в которой не меньше, чем в блоке.

Если количество байт информации файла не делится нацело на длину блока, то при чтении последнего блока признак «конец файла» может встретиться раньше, чем будет прочитан блок. Если последний параметр не

был задан, возникнет ошибка ввода-вывода, которая приведет к аварийному завершению программы (конечно, если не использовалась директива {$I-} и соответствующая обработка аварийной ситуации). Если же последний параметр задан, то в нем будет содержаться число прочитанных записей блока. Однако здесь надо учитывать одну тонкость. В этот параметр передается число полностью прочитанных записей. Если же запись состояла более чем из одного байта, то могло случиться так, что конец файла встретился до того, как запись была полностью прочитана. В этом случае эта «недочитанная» запись не учитывается.

Процедура записи блока на диск имеет вид:

```
BlockWrite(<имя логического файла>, <буфер>,
  <число записей> [, <число прочитанных записей>]);
```

Здесь первые три параметра имеют тот же смысл, что и в предыдущей процедуре. В последний параметр возвращается число реально переданных записей. В подавляющем большинстве случаев оно равно числу записей в блоке, то есть третьему параметру. Однако если при записи на диск возникла ситуация «диск полон», то в этот параметр вернется число записей блока, которые уместились на диске. Так же как и при чтении, если запись не удалось записать на диск целиком, она не войдет в общее количество.

Если последний параметр процедуры опущен, то при переполнении диска произойдет аварийное завершение программы.

Учитывая выше рассмотренные особенности операторов обмена, можно рекомендовать объявлять файлы с длиной записи, равной одному байту. В этом случае не возникнут неприятности, связанные с «недоучтенными» записями.

Все ранее рассмотренные процедуры (кроме Read и Write) для типизированных файлов так же применимы и к бестиповым файлам.

В качестве примера использования бестиповых файлов рассмотрим процедуру копирования файла.

```
{ ============================================== }
{ Копирование файла }
Program CopyFile;
var
   FromF, ToF: file;     { исходный и результирующий файл   }
   NumRead,              { число прочитанных записей блока  }
   NumWritten: Integer;  { число записанных записей блока   }
   NumCopied: Longint;   { общее число скопированных записей }
   Answer: Char;
   buf: array[1..2048] of Char; { буфер для размещения блока }
```

```
begin
   {$I-}
   AssignFile(FromF, ParamStr(1));
   Reset(FromF, 1); { длина записи - 1 байт }
   {$I+}
   if IOResult <> 0 then
   begin
      WriteLn('Исходный файл ',ParamStr(1), ' не найден');
      Halt;
   end;


   {$I-}
   AssignFile(ToF, ParamStr(2));
   { Проверка на наличие файла с заданным именем }
   Reset(ToF);
   {$I+}
   if IOResult = 0 then

   begin
      CloseFile(ToF);
      Write('Файл', ParamStr(2),'уже есть.Переписать? (Y/N)');
      repeat
         ReadLn(Answer);
      until (UpCase(Answer) = 'Y') or (UpCase(Answer) = 'N');
      if UpCase(Answer) = 'N' then
        Halt;
   end;

   Rewrite(ToF, 1); { длина записи - 1 байт }
   WriteLn('Копирую ', FileSize(FromF), ' байт...');
   NumCopied := 0;
   repeat
      BlockRead(FromF, buf, SizeOf(buf), NumRead);
      BlockWrite(ToF, buf, NumRead, NumWritten);
      NumCopied := NumCopied + NumWritten;
   until (NumRead = 0) or (NumWritten < NumRead);

   if NumWritten < NumRead then
      WriteLn('Диск полон. Скопировано ', NumCopied, ' байт')
   else
      WriteLn('Копирование завершено');

   CloseFile(ToF);
   CloseFile(FromF);
end.
```

```
{ ============================================== }
```

Сделаем пояснения по примеру. Чтобы выполнить копирование файла с помощью нашей программы, требуется запустить ее с параметрами командной строки, например, так

```
CopyFile c:\user\data.txt a:data.txt
```

В связи с таким способом вызова программы возникает проблема передачи параметров, записанных в командной строке. В языке Pascal имеется предназначенная для этих целей функция

```
ParamStr(<номер аргумента>);
```

аргументом которой является порядковый номер параметра из командной строки, а результатом – сам параметр в виде строковой (тип `String`) константы. Так, результатом обращения `ParamStr(1)` при условии вызова программы копирования выше приведенной командной строкой будет имя исходного файла `c:\user\data.txt` . Заметим, что вызов функции с номером параметра 0 даст первый элемент командной строки, то есть имя программы (в нашем случае `CopyFile`).

7. Некоторые возможности управления файловой системой

Язык Object Pascal включает набор процедур, отражающих возможности операционной системы по управлению файлами. В их число входят процедуры поиска, удаления файлов, смены текущего каталога и другие. Все эти процедуры позволяют разрабатывать собственные информационно-поисковые системы, базы данных и тому подобные программные комплексы обработки и хранения больших массивов информации.

В языке имеется более десятка соответствующих процедур и функций, часть из которых размещается в библиотеке `System`, часть – в библиотеке `SysUtils`. Кроме того, в библиотеке `SysUtils` определен ряд типов данных и переменных, облегчающих составление программ управления файлами. Мы не будем рассматривать все существующие средства, а ограничимся некоторым набором подпрограмм, необходимым для реализации примера.

В качестве примера рассмотрим программу удаления всех файлов с расширением .EXE из каталога, заданного при вызове программы в командной строке. Причем если каталог не указан, удаление будем

производить в текущем каталоге. Программа должна сообщать имена удаляемых файлов и их размеры в байтах. В завершение программа должна напечатать число свободных байт на логическом диске, где расположен каталог.

Рассмотрим стандартные средства, требуемые для реализации этой программы.

Процедура

```
ChDir(<каталог>);
```

устанавливает в качестве текущего каталог, заданный параметром. Если такого каталога нет, возникает ошибка, которая может быть обработана с использованием функции `IOResult`.

Процедура

```
Erase(<имя логического файла>);
```

удаляет физический файл, связанный с логическим файлом, заданным параметром. В момент удаления файл должен быть закрыт. В нашем примере мы просто не будем открывать удаляемый файл, лишь связав его с файловой переменной оператором `AssignFile`.

Рассмотренные процедуры размещаются в библиотеке `System`. Для использования следующих подпрограмм необходимо подсоединить библиотеку `SysUtils`.

Пара процедур `FindFirst` и `FindNext` обеспечивают поиск файлов. Прежде всего рассмотрим предопределенный в библиотеке тип данных `TSearchRec`, переменные которого используются в качестве параметров этих процедур. Это запись, объявленная как

```
type
  TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ...
  end;
```

Поля этой записи имеют следующий смысл:

Поле	Содержание
`Time`	Дата и время создания файла (упакованные)
`Size`	Размер файла в байтах
`Attr`	Атрибуты файла
`Name`	Имя файла

В примере для нас будут важны атрибуты `Name` и `Size`, смысл которых очевиден.

Процедура

```
FindFirst(<маска файла>, <атрибут>, <данные о файле>);
```

производит поиск первого файла с заданными характеристиками в каталоге, указанном в первом параметре. Вообще, первый параметр – это специальная «маска» или «шаблон» для поиска. В нем задается вид искомых файлов с использованием спецсимволов `'*'` и `'?'`. Например, маска «`*.PAS`» говорит, что требуется найти первый файл с расширением .PAS в текущем каталоге. Второй параметр задает атрибут файла в смысле классификации, принятой в операционной системе. Файлы могут быть системными, предназначенными только для чтения, скрытыми и тому подобное. Третий параметр представляет собой переменную типа `TSearchRec`, в которую заносится информация о найденном файле.

Процедура

```
FindNext(<данные о файле>);
```

находит очередной файл с характеристиками, заданными в предыдущей процедуре. Информация об этом файле заносится в единственный параметр процедуры, который должен быть переменной типа `TSearchRec`.

По завершении процедуры `FindFirst` и `FindNext` возвращают код, характеризующий результат выполнения операции. Код ноль означает, что операция завершилась успешно.

В заключение рассмотрим использованную в примере функцию:

```
DiskFree();
```

которая выдает количество свободных байт (тип результата `Int64`) на логическом диске. Диск задается параметром – целой константой. Константа 0 означает текущий диск, 1 – диск A, 2 – диск B и так далее.

Ну а теперь собственно текст программы.

```
{ ============================================== }
{ Удаление всех файлов с расширением .EXE из заданного каталога }
Program DelEXE;
uses SysUtils;
var
   Dir: String;
   DirInfo: TSearchRec;
   Code: Integer;
   f: file;
begin
   {$I-}
   { Установка нового текущего каталога }
```

```
   Dir := ParamStr(1);
   if Dir <> '' then
   begin
      ChDir(Dir);
      if IOResult <> 0 then
      begin
         WriteLn('Каталог не найден');
         Halt;
      end;
   end;
   { Поиск и удаление файлов }
   Code := FindFirst('*.EXE', faAnyFile, DirInfo);
   while Code = 0 do
   begin
      WriteLn('Удален файл ', DirInfo.Name, ' ',
      DirInfo.Size, ' байт');
      AssignFile(f, DirInfo.Name);
      Erase(f);
      Code := FindNext(DirInfo);
   end;

   WriteLn('Свободных ', DiskFree(0),' байт');
end.
{ ================================================ }
```

8. Выводы

В этой главе мы рассмотрели классические приемы работы с внешней памятью средствами языка Object Pascal, выяснили, что в нем на логическом уровне поддерживаются три вида файлов – типизированные, текстовые и бестиповые, что полностью покрывает все возможные потребности программистов по обмену данными между программой и внешней памятью. Заметим, что современные средства программирования, в частности Borland Delphi, содержат и более развитые механизмы осуществления файловых операций. Познакомившись в главе 10 с объектно-ориентированным программированием, желающие могут изучить класс `TFileStream` из модуля `Classes`, предоставляющий еще более удобные средства для чтения/записи данных.

Литература

1. Богатырев Р. Летопись языков Паскаль // Мир ПК. –№ 4.–2001.

2. Богатырев Р. От Паскаля к языку Zonnon: реализация новых идей на платформе .NET // Мир ПК.–2003.–№ 9.

3. Большая советская энциклопедия. Издание третье. – М.: Советская энциклопедия, 1970.

4. Брэдли Д. Программирование на языке ассемблера для персональной ЭВМ фирмы IBM.–М.: Радио и связь, 1988.

5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. – Бином, 1998.

6. Касперски К. Техника оптимизации программ. Эффективное использование памяти.– СПб.: БХВ-Петербург, 2003.

7. Кетков А., Кетков Ю. Практика программирования: Бейсик, Си, Паскаль. Самоучитель.–СПб.: БХВ-Петербург, 2001.

8. Кетков А., Кетков Ю. Практика программирования: Visual Basic, C++ Builder, Delphi. Самоучитель.–СПб.: БХВ-Петербург, 2002.

9. Кнут Д.Э. Искусство программирования. Т. 1. Основные алгоритмы.–М.:Вильямс, 2000.

10. Кнут Д.Э. Искусство программирования. Т. 3. Сортировка и поиск.–М.:Вильямс, 2004.

11. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы. Построение и анализ.–М.:МЦНМО, 1999.

12. Кэнту М. Delphi 7 для профессионалов.–Спб.:Питер, 2004.

13. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования.–М.: Мир, 1982.

14. Модернизация и ремонт персонального компьютера.– [http://www.allcompinfo.com]

15. Новичков А. Система генерации проектной документации Rational SoDA.– [http://www.citforum.ru/programming/digest/soda.shtml]

16. Олифер В.Г., Олифер Н.А. Сетевые операционные системы.–СПб.: Питер, 2002.

17. Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ.– 4-е изд.–СПб.: Питер; М.: Издательско-торговый дом «Русская Редакция», 2001.

18. Служба тематических толковых словарей. – [http://www.glossary.ru/]

19. Стивенс Р. Delphi. Готовые алгоритмы.– Спб.:Питер, 2004.

20. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика. –Вильямс, 2002.

21. Шень А. Программирование: теоремы и задачи.–М.:МЦНМО, 2004.

22. Юров В. Assembler.–СПб.: Питер, 2002.

23. Cantu M. Essential Delphi.– [www.marcocantu.com/edelphi]

24. Cantu M. Essential Pascal.– [www.marcocantu.com/epascal]

25. Cantu M. Mastering Borland Delphi 2005.–Sybex, 2005.

26. Cantu M. Mastering Delphi 7.–Sybex, 2003.

27. Clark R., Koehler S. The UCSD Pascal Handbook.–Prentice-Hall, 1982.

28. Dahl O., Dijkstra E., Hoare C.A.R. Structured Programming.–London, England: Academic Press, 1972.

29. Gutknecht J., Zueff E. Zonnon Language Report. Draft.–ETH Zurich, June 2003.

30. Gutknecht J., Zueff E. Zonnon for .NET, A Language and Compiler Experiment. // Computer Systems Institute, ETH Zürich, Switzerland. JMLC Conference.–August 2003.

31. Gutknecht J. Zonnon: A .NET Language Beyond C# // Moscow: Microsoft Conference, June 15-17, 2003.

32. Gutknecht J., Wirth N. The Oberon System // Software – Practice and Experience.–Vol.19, № 9.–1989.–p.857-893.

33. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-oriented Software Engineering.–Workingham, England: Addison-Wesley Publishing Company, 1992.

34. Jensen K., Wirth N. PASCAL – User Manual and Report, ISO Pascal Standard.–1974.

35. Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture.– Intel Corporation: 1997.

36. Meyer B. Object-oriented Software Construction.–Prentice Hall, 1988.

37. Rumbaugh H., Blaha M., Premarlani W., Eddy F., Lorensen W. Object-Oriented Modeling and Design.–Prentice-Hall, 1995.

38. Sedgewick R. Algorithms.–Reading, MA: Addison-Wesley, 1983.

39. Tesler L. Object Pascal Report. // Structured Language World, Vol.9, No.3.– 1985.–p. 10-14.

40. Thiriez H. Modelling of an interactive scheduling system in a complex environment // European Jornal of Operational Research. –1991.– №50.–p.37-47.

41. Wirth N. Program Development by Stepwise Refinement // Communications of the ACM vol.26(1).– January 1983.

42. Wirth N. The Programming Language Pascal // Acta Informatica, 1.–1971.–p. 35-63.

43. Wirth N. Programming in Modula-2.–Springer-Verlag, 1982.

44. Wirth N. The programming language Oberon // Software – Practice and Experience, Vol.18, № 7.–1988.–p.671-690.

45. Wirth N. Programming in Modula-2.–Springer, 1974.

46. Wirth N. Pascal and its Successors // Conference on Computer Pioneers.–Bonn, 2001.