# Блог GunSmoker-a



...when altering one's mind becomes as easy as programming a computer, what does it mean to be human?..



17 октября 2011 г.

## Сериализация - файлы в стиле Pascal

Это первый пост в серии постов про сериализацию. В этой части мы рассмотрим т.н. файлы в стиле Pascal.

# Оглавление



# Общие сведения

В языке Pascal (и, следовательно, Delphi) есть встроенные средства по работе с файлами, не зависящие от нижележащей операционной системы.

В отличие от файловых API уровня операционной системы, файлы Pascal ориентированы на работу с *типами данных языка* (а не нетипизированным байтовым потоком, как это имеет место для API операционной системы, которая понятия не имеет про типы данных языка Pascal) и включают в себя высокоуровневую оболочку для работы со строками и наборами однородных данных. Поэтому работа с файлами Pascal удобнее, чем обращение к API операционной системы.

В отличие от других, более современных методов, использующих мета-информацию, файлы Pascal отличаются простотой использования. Поэтому, хотя другие методы работы с файлами могут упростить повседневную работу, но кривая их изучения существенно круче.

Вообще надо сказать, что файловый тип появился в языке именно для обучения программированию. Т.е. он прост и прозрачен. На его примере можно объяснить работу с файлами, не затрагивая сложных для новичков деталей вроде режимов доступа, синхронизации и т.п. С другой стороны это же означает ограничение гибкости.

Сразу же замечу, что я не буду рассказывать для совсем уж начинающих - тут будет подробное изложение. Предполагается, что вы уже прочитали какую-то "книжку по Delphi".

Всего в языке есть три файловых типа:

- текстовые файлы (`var F: TextFile;`)
- типизированные файлы (`var F: file of Тип-Данных;`)
- нетипизированные файлы (`var F: file;`)

Нетипизированные файлы являются самым общим типом, а текстовые и типизированные файлы - обёрткой над нетипизированными файлами, частным случаем. Они предоставляют более удобные способы работы, путём наложения ограничений на содержимое файла.

Для начала стоит немного прояснить терминологию. Вообще говоря, файлы в системе однородны: любой файл - это набор байтов. Не существует какого-то жёсткого разделения файлов по типам. Тем не менее, часто используются следующие термины для какой-то классификации файлов по содержимому.

Под **текстовым** файлом понимают файлы, содержащие читабельный для человека текст в кодировках ANSI, UTF-8 или Unicode. Текст представлен в явном виде, при этом некоторые специальные символы (с кодами от 0 до 31) являются специальными - они всегда имеют один и тот же смысл и обычно отвечают за форматирование текста: перенос строк, табуляцию и т.п. Обычно текстовыми файлами не называют *документы* - файлы, содержащие текст, но оформленные в формате определённой программы (вроде Word), потому что кроме самих данных (текста) такие файлы содержат служебную мета-информацию (заголовки, данные форматирования, атрибуты и т.п.).

В Pascal и Delphi в настоящий момент поддерживаются только текстовые файлы в кодировке ANSI. Файловые типы Pascal считаются устаревшим средством - появившись в языке давно, они уже не развиваются. В те далёкие дни люди мало волновались о кодировках, а о Unicode и UTF-8 и слыхом не слыхивали. Поэтому вы не сможете работать с текстовыми файлами, отличными от канонического ANSI-формата (кодовая страница ANSI, без BOM).

Примечание: начиная с Delphi XE2 (год выхода 2011), файлы Pascal впервые за пару десятков лет получили обновление API: теперь становится возможным указывать кодировку файла, которая может быть отличной от системной кодовой страницы, но вы всё ещё не можете использовать BOM.

Как я уже сказал, "текстовый файл" - это просто удобный ярлык, навешиваемый на файл, чтобы как-то охарактеризовать его содержимое. Это не однозначная и неизменная характеристика. Вы можете открыть текстовый файл в двоичном режиме и вы можете открыть произвольный бинарный файл как текстовый (весь вопрос в осмысленности результатов от таких действий; если первое ещё имеет смысл, то открытие двоичного файла как текстового - весьма сомнительная операция).

**Типизированные** файлы содержат записи одного типа и фиксированной длины. Чаще всего компонентами файла выступают именно записи (`record`), но это может быть и любой другой тип фиксированного размера. К примеру, текстовый файл можно открыть как типизированный - `file of AnsiChar`.

Наконец, под **нетипизированными** файлами понимают файлы произвольной, нерегулярной структуры. Это означает открытие файла в двоичном режиме - как набор байт, без какой-либо предопределённой структуры. Именно в этом режиме работают файловые API операционной системы. В подобном режиме можно открыть любой файл и это всегда будет иметь смысл.

Следует заметить, что понятие "типизированный файл" может трактоваться в двух смыслах. Во-первых, про файл можно говорить, что он типизированный в смысле строгого определения типизированного файла Pascal. Т.е. это файл, определяемый как "`file of что-то`", состоящий из набора блоков одинаковой длины. Иными словами, это файл-массив однотипных элементов. Но на практике "типизированный файл" может употребляться и в более широком смысле - как файл с однородными данными. При этом файл не обязан состоять исключительно из последовательности записей, и сами записи файла не обязаны иметь одинаковый размер. К примеру, в начале файла может быть записан заголовок (сигнатура, контрольная сумма, число записей, версия файла и т.п.), а за ним идти набор записей одинакового вида (скажем, три числа и строка), но записи будут иметь переменную длину (строка разной длины). Такой файл могут

называть типизированным (в смысле однородности его данных), но надо понимать, что он не будет типизированным в смысле языка Pascal - и работать с ним нужно будет как с нетипизированным, двоичным файлом. Применение термина "типизированный файл" к файлам нерегулярной структуры не корректно. Примером такого файла является .exe файл: в нём содержится первичный заголовок, который ссылается на дополнительные заголовки, в нём есть секции кода и данных (произвольных размеров), оглавление ресурсов (и сами ресурсы) и т.п. Все части файла имеют разную длину и разную структуру.

# Общие принципы работы с файлами Pascal

Работа с любыми типами файлов имеет общие элементы, которые мы сейчас и рассмотрим.

Во-первых, работа с файлами в стиле Pascal почти всегда следует одному шаблону (кратко):

1. AssignFile
2. Reset/Rewrite/Append
3. работа с файлом
4. CloseFile

Подробно:

1. Вы начинаете с объявления файловой переменной нужного типа. В зависимости от выбранного вами вида файла, вам нужно использовать `var F: TextFile;` `var F: file of Тип-Данных;` или `var F: file;` (текстовый, типизированный и двоичный файл соответственно), где "Тип-Данных" является типом данных **фиксированного** размера.

2. Вы ассоциируете переменную с именем файла. Делается это вызовом `AssignFile`. Которая имеет прототип:

```
1 function AssignFile(var F: File; FileName: String; [CodePage: Word]): Integer; o
```

Примеры вызова:

```
1 var
2   F1: TextFile;
3   F2: file of Integer;
4   F3: file;
5 begin
6   AssignFile(F1, 'MyFile.txt');
7   AssignFile(F2, 'C:\MyFile.dat');
8   AssignFile(F3, '..\..\MyFolder\MyFile.exe');
9 end;
```

Как видите - вы можете указывать любое допустимое имя файла с путём или без, но по соображениям надёжности кода я рекомендую вам всегда указывать полностью квалифицированное имя файла.

Параметр `CodePage` является необязательным и он появился только начиная с Delphi XE 2. Он применим только для текстовых файлов. Если он не указан, то используется `DefaultSystemCodePage`, что для "русской Windows" равно Windows-1251.

Он указывает кодовую страницу для выполнения перекодировки текста перед записью и после чтения данных из файла. Например:

```
1 var
2   F1: TextFile;
3 begin
4   // Только для Delphi XE 2 и выше:
5   AssignFile(F1, 'MyFile.txt', CP_UTF8); // подразумевается, что MyFile.txt - UT
6 end;
```

Текстовый файл должен быть в указанной вами кодовой странице. Кодовая страница - любая из принимаемых функцией WideCharToMultiByte (или её аналога на не Windows платформах). Файл не может иметь BOM. Так что на практике эта возможность полезна только для открытия ANSI файлов в кодировке, отличной от системной, но не текстовых файлов в UTF-8 и UTF-16, которые почти всегда имеют BOM.

Данная операция НЕ открывает файл. Она просто записывает имя файла в файловую переменную. Так что когда вы откроете файл в будущем (как правило - сразу же после этой операции), то будет открыт именно файл с указанным тут именем.

Данная операция всегда выполняется успешно, но вам нужно быть внимательным и никогда не вызывать её для уже открытых файлов - это приведёт к утечке открытых описателей файла (фактически, AssignFile просто тупо затирает нулями переменную перед выполнением своей работы; так что, чтобы там у вас ни было - оно будет потеряно). Собственно, эта операция нужна не столько для ассоциации с файловым путём, сколько для инициализации переменной. Как правило, за всё время жизни файловой переменной, эта операция применяется к ней единственный раз - первым действием.

В Pascal вместо AssignFile используется Assign. Это ровно эта же процедура, просто в Delphi Assign переименовали в AssignFile, чтобы избежать конфликтов имён с другим кодом, который тоже использует имя Assign (к примеру - форма, компоненты, да и любой TPersistent). Assign всё ещё существует в Delphi, так что любой старый код может быть перекомпилирован в Delphi и он будет работать. Но для нового кода вам лучше использовать AssignFile.

3. Далее файл открывается. Открывать файл можно в трёх режимах: чтение, запись и чтение-запись. Как несложно сообразить, при открытии файла в режиме чтения из него можно только читать, но не писать - и так далее. Таким образом, всего у файловой переменной может быть 4 режима: закрыт, открыт на чтение, открыт на запись и открыт на чтение-запись (любые другие значения указывают на отсутствие инициализации файловой переменной - т.е. закрытый файл):

```
const
  fmClosed = $D7B0;
  fmInput= $D7B1;
  fmOutput = $D7B2;
  fmInOut= $D7B3;
```

Открытие файла выполняется с помощью функций Reset, Rewrite и Append.

Что касается общих моментов: Rewrite создаёт новый файл; Reset открывает существующий файл, а Append является модификацией Reset и открывает файл для дополнения: т.е. после открытия файла переходит в его конец для дозаписи данных. Append применима только к текстовым файлам.

Используются все эти процедуры в целом одинаково:

```
var
  F1: TextFile;
  F2: file of Integer;
  F3: file;
begin
  // Инициализация:
  AssignFile(F1, 'MyFile.txt');
  AssignFile(F2, 'C:\MyFile.dat');
  AssignFile(F3, '..\..\MyFolder\MyFile.exe');

  // Открываем файлы:
  Append(F1);
  Reset(F2);
  Rewrite(F3);
end;
```

Хотя конкретные типы файлов имеют свои особенности использования (см. ниже), но обычно эти подпрограммы работают следующим образом:

1. `Rewrite` создаёт новый файл. Если файл с таким именем уже существует, то он удаляется и вместо него создаётся новый файл. Если файловая переменная уже открыта к моменту вызова `Rewrite`, то файл закрывается и пересоздаётся. После открытия файл позиционируется на начало файла, а функция `EOF` возвращает `True` - указывая на конец файла.
2. `Reset` открывает существующий файл в режиме, указываемом в глобальной переменной `FileMode` (по умолчанию - чтение-запись; возможные значения - только чтение, только запись и чтение-запись). Если файл не существует или не может быть открыть в нужном режиме (заблокирован, отказ доступа) - то возникнет ошибка. Если файловая переменная уже открыта к моменту вызова `Reset`, то файл закрывается и открывается заново. После открытия файл позиционируется на начало файла, а функция `EOF` возвращает `True`, если файл существует и имеет размер 0 байт или (обычно) `False` - если файл существует и имеет ненулевой размер.
3. `Append` применимо только к текстовым файлам и будет рассмотрена ниже.

Глобальная переменная `FileMode` является именно глобальной переменной, а потому установка доступа не является потоко-безопасным действием. В любом случае, переменная может содержать комбинацию (через "or") следующих флагов:

```
const
  fmOpenRead       = $0000;
  fmOpenWrite      = $0001;
  fmOpenReadWrite  = $0002;
  fmExclusive      = $0004;

  fmShareCompat    = $0000 platform;
  fmShareExclusive = $0010;
  fmShareDenyWrite = $0020;
  fmShareDenyRead  = $0030 platform;
  fmShareDenyNone  = $0040;
```

Вы можете указать один флаг вида `fmOpenXYZ` и один флаг из второй группы. Первый флаг определяет режим открытия: только чтение (`fmOpenRead`), только запись (`fmOpenWrite`) или чтение-запись (`fmOpenReadWrite`), вторые флаги определяют режим разделения файла: без разделения (`fmShareExclusive`), запретить другим читать (`fmShareDenyRead`), запретить другим писать (`fmShareDenyWrite`) и без ограничений (`fmShareDenyNone`). И два флага являются специальными. Флаги разделения используют устаревшую deny-семантику MS-DOS, в отличие от современного API. См. также: взаимодействие флагов режима открытия и разделения.

Здесь общие моменты заканчиваются и начинаются различия - все три процедуры имеют небольшие особенности в зависимости от используемого типа файловой переменной, так что мы подробнее рассмотрим их ниже.

4. Затем идёт работа с файлом - чтение и/или запись данных. Эти операции специфичны для разных типов файловых переменных. Их мы отдельно рассмотрим ниже.

5. В итоге после работы файл нужно закрыть. Делается это вызовом `CloseFile`:

```
var
  F1: TextFile;
  F2: file of Integer;
  F3: file;
begin
  // <- открытие и работа с файлом
  CloseFile(F3);
  CloseFile(F2);
  CloseFile(F1);
end;
```

После этого файл будет полностью закрыт, все изменения полностью сброшены на диск, а файловую переменную можно использовать заново - связать её с другим файлом (через AssignFile), открыть и т.д., но, как правило, с файловой переменной работают лишь один раз.

Примечание: аналогично AssignFile (см. выше), CloseFile является переименованной Close, которая тоже всё ещё доступна, но чаще всего замещается другим кодом (к примеру - Close у формы). Всегда используйте CloseFile в новом коде.

Итак, на этом мы закончили разбор общих принципов работы с файлами Pascal. Но прежде чем перейти к обсуждению способов работы с каждым конкретным типом файловых переменных, нужно обсудить ещё один важный общий момент: обработку ошибок.

# Обработка ошибок

Тут надо сказать, что обработка ошибок работы с файлами Pascal весьма запутывающа. Дело в том, что обработку ошибок подпрограммы ввода-вывода файлов Pascal производят в двух режимах - которые переключаются весьма нестандартно: опцией компилятора. Она называется I/O Checking и расположена на вкладке Compiling (Compiler в старых версиях Delphi) в опциях проекта в Delphi (Project/Options). Что ещё хуже - эта настройка может быть изменена прямо в коде, используя директиву компилятора. В коде можно использовать {$I+} для включения этой опции и {$I-} для выключения.

Итак, если эта опция выключена (либо в коде стоит {$I-}), то обработку ошибок любых функций по работе с файлами в стиле Pascal (кроме AssignFile, которая всегда успешна) нужно проводить так: вам доступна функция IOResult, которая возвращает статус последней завершившейся операции. Если она вернула 0 - то операция была успешно выполнена (файл открыт, данные записаны и т.п.). Если же она возвращает что-то иное - произошла ошибка. Какая именно ошибка - зависит от значения, которое она вернула, которое (значение) представляет собой код ошибки. Возможные коды ошибок можно посмотреть здесь (только не закладывайтесь на неизменность этих кодов и неизменность таблицы). Помимо ошибок ввода-вывода, вы можете получать и системные коды ошибок. Их список можно увидеть в модуле Windows. Откройте его и запустите поиск по "ERROR_SUCCESS" (без кавычек). А ниже вы увидите список системных кодов ошибок. Наиболее частые ошибки при работе с файлами - ERROR_FILE_NOT_FOUND, ERROR_PATH_NOT_FOUND, ERROR_ACCESS_DENIED, ERROR_INVALID_DRIVE, ERROR_SHARING_VIOLATION. Но вообще код может быть почти любым. В любом случае, вам доступен только код ошибки, но не её текстовое описание. Если произошла ошибка, то все последующие вызовы функций ввода-вывода будут игнорироваться, пока вы не вызовете IOResult.

Итак, это старый режим обработки ошибок, который пришёл в Delphi из языка Pascal. С ним код выглядит примерно так:

```
1  {$I-}
2  var
3    F: TextFile;
4  begin
5    AssignFile(F, Edit1.Text);
6    Rewrite(F);
7    if IOResult <> 0 then
8    begin
9      Application.MessageBox('Произошла ошибка открытия файла', 'Ошибка', MB_OK or MB_I
10     Exit;
11   end;
12   WriteLn(F, 'Текст для записи в файл');
13   if IOResult <> 0 then
14   begin
15     Application.MessageBox('Произошла ошибка записи данных в файл', 'Ошибка', MB_OK o
16     Exit;
17   end;
18   CloseFile(F);
19 end;
```

Вызов функции очищает код ошибки, так что если вы хотите обработать ошибку, нужно поступать так:

```
{$I-}
var
  F: TextFile;
  ErrCode: Integer;
begin
  ...
  ErrCode := IOResult;
  if ErrCode <> 0 then
  begin
    Application.MessageBox(PChar(Format('Произошла ошибка открытия файла: %d', [ErrCo
    Exit;
  end;
  ...
end;
```

Поскольку ошибка блокирует вызовы функций ввода-вывода, то обработку ошибок удобно делать один раз - в конце работы. Например:

```
{$I-}
var
  F: TextFile;
  ErrCode: Integer;
begin
  // Работа с файлом
  AssignFile(F, Edit1.Text);
  Rewrite(F);
  WriteLn(F, 'Текст для записи в файл');
  CloseFile(F);

  // Обработка ошибок
  ErrCode := IOResult;
  if ErrCode <> 0 then
    Application.MessageBox(PChar(Format('Произошла ошибка работы с файлом: %d', [ErrC
end;
```

Этот код основан на том факте, что если при работе с файлом возникнет ошибка (к примеру - при открытии файла в момент вызова `Rewrite` из-за того, что в `Edit1` указано недопустимое имя файла), то все нижеследующие функции не будут ничего делать, а код ошибки будет сохранён до вызова `IOResult`.

Обратите внимание, что правило "ничего не делать при ошибке" не распространяется на `CloseFile`. Вот почему мы вставили обработку ошибок в самый конец, а не до `CloseFile`.

Окей, с этим способом всё. Теперь, если вы включаете опцию I/O Checking или используете {$I+}, то функция `IOResult` вам не доступна, а всю обработку ошибок берут на себя сами функции ввода-вывода: если при вызове любой из них возникнет ошибка - функция сама обработает её путём возбуждения исключения (если модуль `SysUtils` не подключен - то возбуждением run-time ошибки; подробнее - тут). Исключение имеет класс `EInOutError`:

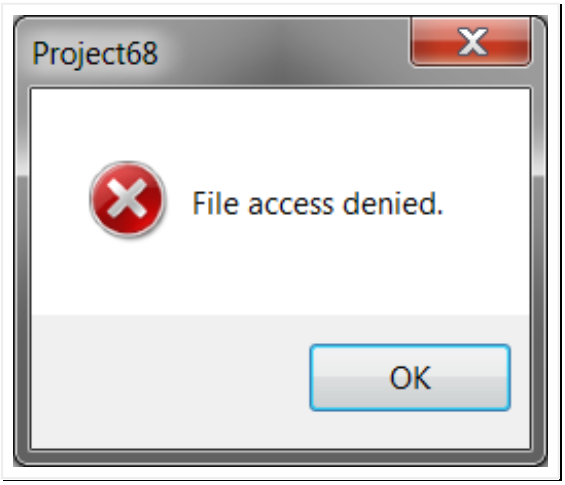
```
type
  EInOutError = class(Exception)
  public
    ErrorCode: Integer;
  end;
```

Собственно, код в этом режиме всегда будет выглядеть так:

```
{$I+}
var
  F: TextFile;
  ErrCode: Integer;
begin
  AssignFile(F, Edit1.Text);
  Rewrite(F);
  try
```

```
    WriteLn(F, 'Текст для записи в файл');
  finally
    CloseFile(F);
  end;
end;
```

А при возникновении проблем - у вас будет исключение, которое в стандартной VCL Forms программе в конечном итоге обрабатывается показом сообщения:




Я допускаю, что вы можете не очень быть знакомы с работой с исключениями (и в таком случае вам можно начать с этой статьи), но я бы всё же рекомендовал использовать режим {$I+} по одной очень простой причине: с исключениями поведение по умолчанию - показ ошибки; в режиме {$I-} - скрытие и игнорирование. Иными словами, если в {$I-} по незнанию или из-за лени вы опустите обработку ошибок, а при выполнении программы ошибка всё же возникнет - вы останетесь ни с чем: код просто не работает и вы понятия не имеете почему. С исключениями же такое невозможно: вам нужно специально прикладывать усилия, чтобы скрыть ошибку, так что это не может произойти случайно, по недосмотру.

В любом случае, вот код приведения второго режима к шаблону первого:

```
{$I+}
var
  F: TextFile;
  ErrCode: Integer;
begin
  try
    AssignFile(F, Edit1.Text);
    Rewrite(F);
    try
      WriteLn(F, 'Текст для записи в файл');
    finally
      CloseFile(F);
    end;

  except
    on E: EInOutError do
      // Произошла ошибка - обработаем. В данном случае - показом сообщения, но в общ
      Application.MessageBox(PChar(E.Message), 'Ошибка', MB_OK or MB_ICONERROR);
  end;
end;
```

На этом разговор об обработке ошибок при работе с файлами Pascal можно считать законченным.

Персональное примечание: напомню, что именно запутанность обработки ошибок в файлах Pascal привела к обнаружению вируса Virus.Win32.Induc.a.

В любом случае, это всё. И нам наконец-то можно приступить к обсуждению непосредственно работы с

файлами.

# Текстовые файлы

Итак, текстовый файл - это обычный файл, открываемый через переменную типа `TextFile`, для которого делается предположение о чисто текстовом содержимом. Тип `TextFile` в Delphi имеет псевдоним `Text` - это текстовый тип в Pascal. И снова: тип был переименован, старый `Text` всё ещё существует по соображениям совместимости, но в новых программах нужно использовать тип `TextFile`.

Когда вы открываете текстовый файл, его содержимое интерпретируется специальным образом: предполагается, что файл содержит последовательности читабельных символов, организованных в строки, при этом каждая строка заканчивается маркером end-of-line ("конец строки"). Иными словами, текстовый файл трактуется не как просто `file of Char`, а с дополнительной смысловой нагрузкой. Для текстовых файлов есть специальные формы `Read` и `Write`, на которые мы посмотрим чуть позже.

В Delphi есть три стандартные глобальные переменные типа текстовых файлов: `Input` для чтения или `Output` для вывода - это специальные имена для стандартных каналов ввода-вывода; ещё есть `ErrOutput` - стандартный канал вывода ошибок, по умолчанию он направляется в канал вывода, но вызывающая программа может отделить его. Соответственно, `Input` открывается только для чтения и обычно представляет собой клавиатуру в консольных приложениях (если ввод не был перенаправлен), а остальные два - только для записи (и обычно представляют собой экран). Эти файловые переменные открываются автоматически. В Windows они открываются только для консольных программ, но на других платформах это может быть и не так.

В любом случае, для открытия текстовых файлов можно использовать `Rewrite` (создание), `Reset` (открытие) и `Append` (дозапись в существующий файл). Все три используются одинаково - им передаётся переменная файлового типа. Больше аргументов у них нет. Особенность текстовых файлов: `Rewrite` открывает файл (вернее - создаёт) в режиме только запись, `Reset` всегда открывает файл в режиме только чтение, а `Append` открывает файл в режиме только запись. Текстовые файлы нельзя открыть в режиме чтение-запись.
`Append` аналогична `Reset`, только устанавливает текущую позицию файла в конец и открывает в режиме записи, а не чтения.

После открытия вам доступны процедуры `Write`, `WriteLn`, `Read` и `ReadLn` для записи и чтения строк (варианты функций с *Ln допустимы только для текстовых файлов). Эти подпрограммы не являются настоящими функциями и процедурами, а представляют собой магию компилятора. Первым параметром у них идёт файловая переменная - она указывает файл, с которым будет производится работа (чтение строк или запись), а далее идёт произвольное число параметров - что пишем или читаем.

Если опустить файловую переменную - будет подразумеваться консоль (`Input` и `Output`). Это используется для ввода-вывода в консольных программах. Вам необязательно работать со стандартными каналами ввода-вывода именно через файлы Pascal, вы можете использовать и API.

Иными словами, вызовы этих функций имеют форму:

```
1 Write(F, что-то);
2 WriteLn(F, что-то);
3 Read(F, что-то);
4 ReadLn(F, что-то);
```

Где "что-то" - это одна или более строковых или числовых переменных, указанных через запятую - как обычные аргументы. Мы рассмотрим это чуть позже.

При этом, краткая форма:

```
1 Write(что-то);
2 WriteLn(что-то);
3 Read(что-то);
4 ReadLn(что-то);
```

эквивалентна:

```
1 Write(Output, что-то);
2 WriteLn(Output, что-то);
3 Read(Input, что-то);
4 ReadLn(Input, что-то);
```

В Windows Input, Output и ErrOutput доступны только для консольных программ. Попытка использовать их в GUI приложении приведёт к ошибке (операция над закрытой файловой переменной). Это - частая ошибка новичков. Они забывают указать первым параметром файловую переменную. Эта ошибка не страшна в Windows, поскольку вы тут же увидите проблему при запуске программы, но достаточно коварна на других платформах, где стандартные каналы ввода-вывода могут быть открыты для всех программ. Таким образом, если вы забудете указать файловую переменную, то ваша программа будет работать, не выдавая ошибки - но будет работать неправильно.

Функции с постфиксом Ln отличаются от своих собратьев тем, что используют разделитель строк. Иными словами, Write записывает данные в файл, а WriteLn дополнительно после этого вписывает в файл перенос строки. Т.е. несколько вызовов Write подряд будут писать данные в одну строку. Аналогично, Read читает данные из файла, а ReadLn после этого ищет конец строки и делает переход к следующей строке в файле. В качестве разделителя строк используется умолчание для платформы, если вы пишете в файл, или Enter, если вы работаете с консолью. К примеру, для Windows разделителем строк является последовательности из двух символов #13#10 - известные как CR (carriage return) и LF (line feed), они имеют коды 13 и 10, соответственно. По историческим причинам выбор разделителя строк зависит от платформы. Вы можете изменить умолчание вызовом SetLineBreakStyle, но замечу, что работа с файлами другой платформы - это достаточная головная боль. Я не буду это подробно рассматривать. Часто наилучшее решение - предварительная нормализация данных и файлов. В частности, в Delphi есть функция AdjustLineBreaks.

Read читает все символы из файла вплоть до конца строки или конца файла, но она не читает сами маркеры. Чтобы перейти на следующую строчку - используйте ReadLn. Если вы не вызовите ReadLn, то все вызовы Read после встречи маркера конца строки будут возвращать пустые строки (для чисел - нули). Опознать конец строки или конец файла можно с помощью функций EoLn и EoF (или функций SeekEoLn и SeekEoF - и эти функции не следует путать с функцией Seek). Если же читаемая строка длиннее, чем аргумент у Read/ReadLn, то результат обрезается без возбуждения ошибки. Но если вы читаете числа, то Read пропускает все пробелы, табуляторы и переносы строк, пока не встретит число. Иными словами, при чтении чисел они должны отделяться друг от друга пробелами, табуляторами или размещаться на отдельных строках. При чтении строк вы должны переходить на следующую строку сами.

Write записывает значения в файл. При этом каждый параметр у Write может быть строковым, логическим или числовым (включая числа с плавающей запятой) типом и иметь опциональный спецификатор форматирования значения. При этом все не текстовые данные конвертируются в строки перед записью в файл согласно указанным спецификаторам форматирования.

В общем случае параметр выглядит так:

OutExpr [: MinWidth [: DecPlaces ] ]

Где в квадратных скобках указываются опциональные (необязательные) части. OutExpr - это само выражение для записи. Оно может быть переменной, константой или непосредственным значением. MinWidth и DecPlaces являются спецификаторами форматирования и должны быть числами. MinWidth указывает общую длину вывода и должно быть числом большим нуля. По необходимости слева добавляется нужное количество пробелов. А DecPlaces указывает число знаков после десятичной точки и применимо только при записи чисел. Если эти данные не указаны, то используется научный формат представления чисел. Форматирование значений при выводе - наследие Pascal, где не было функций форматирования строк. В современных программах предпочтительнее использовать функцию Format и её варианты. Этот современный вариант форматирования данных является стандартным решением в Delphi и предоставляет больше возможностей.

Write пишет значения друг за другом, без разделителей. Поэтому, когда вы записываете числа, вам нужно вставлять пробелы, табуляторы или использовать WriteLn.

При этом нужно быть аккуратным при чтении и записи смеси чисел и строк. Оптимальнее всего размещать каждое значение на отдельной строке в файле. Связано это с тем, что если вы записали в файл в одну строчку, скажем, число, текст и число, то при чтении из файла прочитается число, а затем текст - но в этот текст будет включено второе число - поскольку нет никакой возможности отличить текст от числа. Иными словами, при чтении строк читаются все данные до ближайшего маркера (конца строки или файла).

Далее необходимо заметить, что переменные текстового файлового типа имеют буфер. Все данные, записываемые в файл, на самом деле записываются в этот буфер. И лишь при закрытии файла или переполнения буфера данные сбрасываются на диск. Аналогичные действия производятся и при чтении файла. Это делается для оптимизации посимвольного ввода-вывода. Буфер можно сбросить и вручную в любой момент - с использованием подпрограммы Flush. По умолчанию размер буфера равен 128 байтам, но его можно изменить на произвольное значение вызовом SetTextBuf.

Текстовые файлы не поддерживают позиционирование.

# Типизированные файлы

Типизированный файл - это очень простая БД в виде "array of что-то". Как уже было сказано, "что-то" должно иметь фиксированный размер в байтах, поэтому строки и динамические массивы хранить нельзя (но можно - короткие строки или статические массивы символов). Для открытия файлов доступны Rewrite и Reset. Здесь нет никаких особенностей по сравнению с вышеуказанными общими принципами. Запись и чтение из файла осуществляется с помощью Write и Read.

В отличие от текстовых файлов, типизированные и нетипизированные файлы поддерживают позиционирование. Вы можете установить текущую позицию в файле с помощью Seek. Процедура принимает два параметра - файл и номер позиции, на которую нужно переместиться. Положение отсчитывается не в байтах, а в размере записи файла. Иными словами, если вы работаете с, к примеру, file of Integer, то Seek(F, 0) переместит вас в начало файла, Seek(F, 1) - ко второму элементу (т.е. через 4 байта от начала файла), Seek(F, 2) - к третьему (через 8 байт), а Seek(F, FileSize(F)) - в конец файла. Т.е. функция FileSize тоже возвращает размер файла не в байтах, а в записях. Этот размер совпадает с размером в байтах только для file of byte и аналогичных типов - с однобайтовыми записями. Текущую файловую позицию (снова в записях) всегда можно узнать вызовом FilePos.

Ещё одной особенностью типизированных (и нетипизированных) файлов является функция Truncate. Она удаляет содержимое файла за текущей позицией. После её вызова функция EoF возвращает True.

# Нетипизированные файлы

Нетипизированные файлы заполняют пробел в файлах Pascal, позволяя открывать произвольные файлы в двоичном режиме и осуществлять побайтовый доступ. Данный тип файлов не имеет никакого преимущества перед другими методами работы с файлами. В частности, почти всегда для нетипизированного доступа к файлу оказывается предпочтительнее TFileStream (мы рассмотрим его в другой раз).

Для нетипизированных файлов нам доступны Rewrite и Reset - равно как и для типизированных файлов. Но тут есть одно важное отличие: для нетипизированных файлов эти процедуры принимают два параметра. Первый параметр, как обычно, файловая переменная. А второй параметр - размер блока. Размер блока измеряется в байтах и является аналогом размера записи у типизированных файлов. Размер блока влияет на все подпрограммы работы с нетипизированными файлами, которые принимают размеры или позицию. Все они подразумевают указание размеров/позиции в блоках, а не байтах. Вы можете указать 1, чтобы производить измерения в байтах. Плохая новость - второй параметр является опциональным, его можно не указывать. Проблема тут в том, что если вы его не укажете, то размер блока по умолчанию будет 128 байт - не самое очевидное поведение.

Далее, для чтения и записи в нетипизированный файл вместо Read и Write используются функции BlockRead и BlockWrite. Обе они используются одинаково: первый параметр - файловая переменная, второй параметр - что пишем/читаем, третий параметр - сколько пишем/читаем (в блоках). Функция возвращает сколько реально было прочитано/записано. Если блок прочитан/записан целиком, то результат равен третьему параметру. У обеих функций есть перегруженные варианты, у которых результат функции возвращается четвёртым параметром.

Замечу, что второй параметр - нетипизированный. Это значит, что компилятор не выполняет проверок типа. И вам лучше бы не напутать, что туда передавать. Я в первую очередь сейчас говорю про указатели и динамические типы. К примеру:

```
var
  StatArray: array[0..15] of Integer;
  DynArray: array of Integer;
  AnsiStr: AnsiString;
  Str: String;
  PCh: PChar;
  F: file;
begin
  AssignFile(F, 'test');
  Rewrite(F, 1);

  // Неправильно (порча памяти):
  BlockWrite(F, DynArray, Length(DynArray) * SizeOf(Integer));
  BlockWrite(F, AnsiStr, Length(AnsiStr));
  BlockWrite(F, Str, Length(Str) * SizeOf(Char));
  BlockWrite(F, PCh, StrLen(PCh) * SizeOf(Char));

  // Правильно (при условии, что размер данных > 0):
  BlockWrite(F, StatArray, Length(StatArray) * SizeOf(Integer));
  BlockWrite(F, StatArray, SizeOf(StatArray));
  BlockWrite(F, StatArray[0], Length(StatArray) * SizeOf(Integer));
  BlockWrite(F, StatArray[0], SizeOf(StatArray));
  BlockWrite(F, Pointer(DynArray)^, Length(DynArray) * SizeOf(Integer));
  BlockWrite(F, DynArray[0], Length(DynArray) * SizeOf(Integer));
  BlockWrite(F, Pointer(AnsiStr)^, Length(AnsiStr));
  BlockWrite(F, AnsiStr[1], Length(AnsiStr));
  BlockWrite(F, Pointer(Str)^, Length(Str) * SizeOf(Char));
  BlockWrite(F, Str[1], Length(Str) * SizeOf(Char));
  BlockWrite(F, PCh^, StrLen(PCh) * SizeOf(Char));
  BlockWrite(F, PCh[0], StrLen(PCh) * SizeOf(Char));

  // Неправильно (неверный индекс):
  BlockWrite(F, AnsiStr[0], Length(AnsiStr));
  BlockWrite(F, Str[0], Length(Str) * SizeOf(Char));
  BlockWrite(F, PCh[1], StrLen(PCh) * SizeOf(Char));

  // Неправильно (неверный размер 1):
  BlockWrite(F, Pointer(DynArray)^, SizeOf(DynArray));
  BlockWrite(F, DynArray[0], SizeOf(DynArray));
  BlockWrite(F, Pointer(AnsiStr)^, SizeOf(AnsiStr));
  BlockWrite(F, AnsiStr[1], SizeOf(AnsiStr));
  BlockWrite(F, Pointer(Str)^, SizeOf(Str));
  BlockWrite(F, Str[1], SizeOf(Str));

  // Неправильно (неверный размер 2):
  BlockWrite(F, Pointer(Str)^, Length(Str));
  BlockWrite(F, Str[1], Length(Str));

  CloseFile(F);
end;
```

Фух, достаточно сложно. Но только если вы не понимаете как работают указатели.

Следует заметить, что достаточно часто вместо использования типизированного файла оказывается проще открыть файл в двоичном режиме (как нетипизированный) и загрузить/сохранить все данные за раз - вместо организации цикла. К примеру (обработка ошибок убрана):

```pascal
type
  TSomething = Integer; // или запись или что угодно - элемент нашей БД.

// Было:
var
  Data: array of TSomething;
  F: file of TSomething;
  Index: Integer;
begin
  // Запись:

  // <- заполнение Data
  AssignFile(F, 'Test.bin');
  Rewrite(F);
  for Index := 0 to High(Data) do
    Write(F, Data[Index]);
  CloseFile(F);

  // Чтение:
  AssignFile(F, 'Test.bin');
  Reset(F);
  SetLength(Data, FileSize(F));
  for Index := 0 to High(Data) do
    Read(F, Data[Index]);
  CloseFile(F);
end;

// Стало:
var
  Data: array of TSomething;
  F: file;
begin
  // Запись:

  // <- заполнение Data
  AssignFile(F, 'Test.bin');
  Rewrite(F, 1);
  BlockWrite(F, Pointer(Data)^, Length(Data) * SizeOf(TSomething));
  CloseFile(F);

  // Чтение:
  AssignFile(F, 'Test.bin');
  Reset(F, 1);
  SetLength(Data, FileSize(F) div SizeOf(TSomething));
  BlockRead(F, Pointer(Data)^, Length(Data) * SizeOf(TSomething));
  CloseFile(F);
end;

// Или:
var
  Data: array of TSomething;
  F: file;
begin
  // Запись:

  // <- заполнение Data
  AssignFile(F, 'Test.bin');
  Rewrite(F, SizeOf(TSomething));
  BlockWrite(F, Pointer(Data)^, Length(Data));
  CloseFile(F);

  // Чтение:
  AssignFile(F, 'Test.bin');
  Reset(F, SizeOf(TSomething));
  SetLength(Data, FileSize(F));
  BlockRead(F, Pointer(Data)^, Length(Data));
  CloseFile(F);
end;
```

Разумеется, нетипизированные файлы поддерживают позиционирование - ровно как и типизированные

файлы. Только не забывайте про измерение в блоках, а не байтах.

# Прочие подпрограммы и особенности

Хотя типы файловых переменных являются "чёрным ящиком", но внутренне они представлены записями `TTextRec` (для текстовых файлов) или `TFileRec` (для всех прочих файлов). Обе записи объявлены в модуле `System`:

```
type
  TFileRec = packed record
    Handle: NativeInt;
    Mode: Word;
    Flags: Word;
    case Byte of
      0: (RecSize: Cardinal);   //  files of record
      1: (BufSize: Cardinal;     //  text files
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PAnsiChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of WideChar;
      );
  end;

  TTextRec = packed record
    Handle: NativeInt;
    Mode: Word;
    Flags: Word;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PAnsiChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of WideChar;
    Buffer: TTextBuf;
    CodePage: Word;
    MBCSLength: ShortInt;
    MBCSBufPos: Byte;
    case Integer of
      0: (MBCSBuffer: array[0..5] of AnsiChar);
      1: (UTF16Buffer: array[0..2] of WideChar);
  end;
```

Вам не нужны эти записи для обычной работы, но иногда могут быть ситуации, когда вам нужно получать доступ к полям записи. Сделать это можно простым приведением типа:

```
var
  F1: TextFile;
  F2: file;
begin
  TTextRec(F1).Handle; // <- системный описатель открытого файла
  TFileRec(F2).Mode; // <- режим открытого файла
end;
```

Примечание: структура записей зависит от версии вашей Delphi. Прежде чем использовать код, который работает с этими записями или объявляет аналогичные хак-записи - убедитесь, что код написан для вашей версии Delphi или адаптируйте объявления и код.

# Практика

Все примеры ниже приводятся с полной обработкой ошибок для режима {$I+}. Примеры используют файл в папке с программой. Это удобно для тестирования и экспериментов, но, как указано ранее, этого нужно избегать в реальных программах. После тестирования и перед использованием кода в релизе вы должны заменить папку программы на подпапку в Application Data.

# Практика: текстовые файлы

1. Одно значение: `Double`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Value: Double;
begin
  Value := 5.5;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    Write(F, Value); // запишет число в научном формате; если у вас числа тольк
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Value: Double;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    Read(F, Value);
  finally
    CloseFile(F);
  end;

  // Здесь Value = 5.5
end;
```

Обратите внимание, что запись чисел всегда использует фиксированный формат числа - вне зависимости от региональных установок. Иными словами, конвертация чисел в строки и обратно выполняется процедурами `Str` и `Val`. Это имеет как плюсы, так и минусы.

С одной стороны, файл, созданный на одной машине, без проблем прочитается на другой.

С другой стороны, текстовый файл, очевидно, предназначен для редактирования человеком. Иначе зачем делать его текстовым? А человек вправе ожидать, что числа будут использовать "правильный формат". К примеру, для России это - использование запятой в качестве разделителя целой и дробной частей, а не точки. Простого решения этой проблемы для текстовых файлов в стиле Pascal нет.

2. Одно значение переменного размера: `String`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Value: String;
begin
```

```
  Value := 'Example string';

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    WriteLn(F, Value);
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Value: String;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    ReadLn(F, Value);
  finally
    CloseFile(F);
  end;

  // Здесь Value = 'Example string'
end;
```

Ключевой момент при записи данных с динамическим размером - размещать каждое такое значение на отдельной строке. Тогда размер данных = размеру строки.

3. Набор однородных значений: `array of Double`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Values: array of Double;
  Index: Integer;
begin
  // Генерируем случайные 10 чисел
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := Random * 10;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    for Index := 0 to High(Values) do
      Write(F, Values[Index], ' ');
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Values: array of Double;
  Value: Double;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    while not SeekEof(F) do
    begin
      Read(F, Value);

      // Добавление нового элемента в динамический массив
```

```
      SetLength(Values, Length(Values) + 1);
      Values[High(Values)] := Value;
    end;
  finally
    CloseFile(F);
  end;

  // Здесь Values тождественно равны исходным данным из Button1Click
end;
```

Обратите внимание, что записываем мы все значения в одну строчку - поэтому нам нужно вставить разделитель (пробел). Кроме того, мы могли бы также писать каждое число на отдельной строке - используя `WriteLn`. Тогда пробел был бы уже не нужен.

Что касается чтения, то мы могли бы читать ровно как и писать - циклом. Но это подразумевает, что у вас жёсткая структура файла. Т.е. записали все числа в одну строчку - значит, и человек после редактирования файла должен оставить все числа в одной строке. И так далее.

Поэтому вместо этого я показал, как вы можете считать файл произвольной структуры - лишь бы в нём были бы числа. Для этого мы делаем цикл чтения, пока не будет встречен конец файла (`while not SeekEof(F) do`), а в самом цикле считываем и добавляем в массив каждое число. При этом мы пользуемся тем фактом, что `Read` при чтении числа будет пропускать все пробельные символы, включая переносы строк. Вот почему нам не нужно явно вызывать `ReadLn`.

Ещё один момент - использование `SeekEoF` вместо просто `EoF`. Если бы мы использовали `EoF`, то тогда в массив добавлялся бы ноль в конец, если в конце файла стоит несколько пробелов или пустых строк.

4. Набор однородных значений переменного размера: `array of String`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Values: array of String;
  Index: Integer;
begin
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := 'Str #' + IntToStr(Index);

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    for Index := 0 to High(Values) do
      WriteLn(F, Values[Index]);
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Values: array of String;
  Value: String;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    while not Eof(F) do
    begin
      ReadLn(F, Value);

```

```
      SetLength(Values, Length(Values) + 1);
      Values[High(Values)] := Value;
    end;
  finally
    CloseFile(F);
  end;

  // Здесь Values тождественно равны исходным данным из Button1Click
end;
```

Здесь всё оказывается ещё проще - динамические данные должны размещаться на отдельной строке, так что мы просто используем WriteLn/ReadLn. Обратите внимание, что в этом случае, поскольку мы записываем строки, то нет никакой возможности отличить пустую строку в конце файла - часть ли это данных или просто человек случайно добавил её. Если в ваших данных пустые строки недопустимы, то вы можете заменить EoF на SeekEoF, как это сделано в предыдущем примере.

5. Запись - набор неоднородных данных:

```
type
  TData = record
    Signature: LongWord;
    Size: LongInt;
    Comment: String;
    CRC: LongWord;
  end;

// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Value: TData;
begin
  Value.Signature := 123;
  Value.Size := SizeOf(Value);
  Value.Comment := 'Example';
  Value.CRC := 0987654321;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    WriteLn(F, Value.Signature);
    WriteLn(F, Value.Size);
    WriteLn(F, Value.Comment);
    WriteLn(F, Value.CRC);
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Value: TData;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    ReadLn(F, Value.Signature);
    ReadLn(F, Value.Size);
    ReadLn(F, Value.Comment);
    ReadLn(F, Value.CRC);
  finally
    CloseFile(F);
  end;

  // Здесь Value = значениям из Button1Click
end;
```

Тут всё достаточно прозрачно - каждое поле на новой строке.

6. Набор (массив) из записей - иерархический набор данных:

```
type
  TPerson = record
    Name: String;
    Age: Integer;
    Salary: Currency;
  end;

  TPersons = array of TPerson;

// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: TextFile;
  Values: TPersons;
  Index: Integer;
begin
  SetLength(Values, 3);
  for Index := 0 to High(Values) do
  begin
    Values[Index].Name := 'Person #' + IntToStr(Index);
    Values[Index].Age := Random(20) + 20;
    Values[Index].Salary := Random(10) * 5000;
  end;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Rewrite(F);
  try
    for Index := 0 to High(Values) do
    begin
      WriteLn(F, Values[Index].Name);
      WriteLn(F, Values[Index].Age);
      WriteLn(F, Values[Index].Salary);
    end;
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TextFile;
  Values: TPersons;
  Value: TPerson;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.txt');
  Reset(F);
  try
    while not SeekEoF(F) do
    begin
      ReadLn(F, Value.Name);
      ReadLn(F, Value.Age);
      ReadLn(F, Value.Salary);

      SetLength(Values, Length(Values) + 1);
      Values[High(Values)] := Value;
    end;
  finally
    CloseFile(F);
  end;

  // Здесь Values тождественно равны исходным данным из Button1Click
end;
```

Пример почти полностью эквивалентен предыдущему. Обратите внимание, что мы не используем

каких либо разделителей между записями (элементами массива) - они идут сплошным потоком. Следующая запись начинается за предыдущей. Общая длина данных, как и ранее, определяется по размеру самого файла - через признак его конца.

7. Массив из записей внутри записи - составные данные:

```
type
  TCompose = record
    Signature: LongInt;
    Person: TPerson;
    Count: Integer;
    Related: TPersons;
  end;

  TComposes = array of TCompose;
```

С вложением самих записей проблем не возникает - я даже не буду писать пример, т.к. он эквивалентен предыдущему. Просто выпишите в ряд `WriteLn` полей `TCompose`. Для полей-записи вместо одного `WriteLn` вам нужно будет написать несколько - по одному на каждое поле вложенной записи. Ну а `ReadLn` будут зеркальным отражением `WriteLn`.

Но вот вложение массива записей уже является непреодолимым препятствием для общего случая. Дело в том, что в текстовом файле нет возможности как-то указать размер вложенных данных, ведь обычная техника записи динамических данных в текстовый файл - использование разделителей (чаще всего - переноса строк). В частных случаях вы можете найти решение. Скажем, отделять поле `Related` пустой строкой от следующего элемента/поля. Но в общем случае приемлемого решения нет - вам нужно использовать нетипизированный файл. В некоторых случаях вы можете предложить введение формата в текстовый файл. Вроде INI, XML, JSON и т.п. Но на это мы посмотрим в следующий раз.

# Практика: типизированные файлы

1. Одно значение: `Double`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file of Double;
  Value: Double;
begin
  Value := 5.5;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
  Rewrite(F);
  try
    Write(F, Value);
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file of Double;
  Value: Double;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
  Reset(F);
  try
    Read(F, Value);
```

```
28    finally
29      CloseFile(F);
30    end;
31
32    // Здесь Value = 5.5
33  end;
```

В данном случае число записывается в двоичном формате, а не текстовом, так что все проблемы с региональными настройками обходят нас стороной.

Тут надо сделать примечание, почему вообще для этого примера выбран именно тип Double, а не Extended, который в Delphi является де-факто стандартом для чисел с плавающей запятой. Дело в том, что Extended зависит от платформы. Его размер может меняться. Так что если вы компилируете, скажем 32-битную и 64-битную программы - у них размер Extended будет разным. Что означает, что из одной программы вы не сможете прочитать данные, созданные в другой. Это не проблема, если вы планируете работу только в одной платформе - можете спокойно использовать Extended. В противном случае вам нужно использовать Double. Ну а если вам нужно прочитать Extended, созданный на другой платформе, то вы можете использовать тип TExtended80Rec (появился, начиная с Delphi XE2, где, собственно, и появилась поддержка нескольких платформ) вместо Extended.

Аналогично, по этой же причине вам следует избегать Integer и Cardinal при работе с типизированными файлами - потому что это generic-типы, размер которых может меняться. Используйте вместо них LongInt и LongWord соответственно.

2. Одно значение переменного размера: String. Сделать это для типизированных файлов невозможно. В типизированный файл (в смысле файловых типов языка Pascal) нельзя записывать данные переменного (динамического) размера. Для динамических данных нужно использовать либо текстовые, либо нетипизированные файлы. Что вы можете сделать - так это использовать какое-то ограничение.

   К примеру, если брать строки, то вы можете использовать ShortString - это ограничит ваши данные ANSI и 255 символами. Ещё вариант - статический массив символов. Скажем array[0..4095] of Char (AnsiChar/WideChar). Обратите внимание, что запись в файл ShortString или массива из символов - это не аналог текстовых файлов, потому что кроме значимого текста в файле появляется т.н. padding - мусорные данные, не несущие смысловой нагрузки, а служащие для дополнения данных до нужного размера. Вы можете вызывать FillChar или ZeroMemory для предварительной очистки данных перед записью - чтобы визуально подчеркнуть неиспользуемость дополнения.

   Надо понимать, что чем больше (по байтовому размеру) вы возьмёте тип, тем больше места у вас будет тратиться зря (на padding), если в основной массе у вас короткие строки. С другой стороны, если вы возьмёте недостаточно большой тип, то ваши данные будут обрезаться. Так что подобный вариант далеко не всегда возможен - а только, если вы можете предложить подходящее ограничения размера.

   Здесь и далее я не буду приводить пример - он всегда эквивалентен предыдущему примеру с данными фиксированного размера. Только замените типы.

3. Набор однородных значений: array of Double

```
1   // Сохранение в файл
2   procedure TForm1.Button1Click(Sender: TObject);
3   var
4     F: file of Double;
5     Values: array of Double;
6     Index: Integer;
7   begin
```

```
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := Random * 10;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
  Rewrite(F);
  try
    for Index := 0 to High(Values) do
      Write(F, Values[Index]);
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file of Double;
  Values: array of Double;
  Index: Integer;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
  Reset(F);
  try
    SetLength(Values, FileSize(F));
    for Index := 0 to High(Values) do
      Read(F, Values[Index]);
  finally
    CloseFile(F);
  end;

  // Здесь Values тождественно равны исходным данным из Button1Click
end;
```

Обратите внимание, что нам не нужен разделитель, поскольку элементы имеют фиксированный размер. Кроме того, нет проблемы с пробелами в конце, ранее решаемой SeekEoF. Кроме того, фиксированность элементов позволяет узнать длину массива заранее - по размеру файла, что в итоге позволяет написать более эффективный код.

4. Запись - набор неоднородных данных:

```
type
  TData = record
    Signature: LongWord;
    Size: LongInt;
    Comment: String;
    CRC: LongWord;
  end;
```

Аналогично второму примеру, записать неоднородные данные в типизированный файл невозможно. Если вы объявите file of LongWord, то вы не сможете записать в него строку и наоборот. В общем, нужно использовать текстовые или нетипизированные файлы.

Вы можете подумать, что вы могли бы объявить file of TData - ну, с заменой динамических строк на фиксированные аналоги, конечно же. А затем использовать первый пример. Да, это будет работать для конкретного объявления TData, но только этот пример - не на запись в файл record-а, а на запись неоднородных данных.

Два остальных примера (с массивами записей и вложенными записями) также не дадут ничего нового - либо вы используете фиксированные элементы, и тогда эти примеры сводятся к предыдущим, либо - нет, и тогда использование типизированных файлов невозможно.

# Практика: нетипизированные файлы

1. Одно значение: Double

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Value: Double;
begin
  Value := 5.5;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    BlockWrite(F, Value, SizeOf(Value));
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Value: Double;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Reset(F, 1);
  try
    BlockRead(F, Value, SizeOf(Value));
  finally
    CloseFile(F);
  end;

  // Здесь Value = 5.5
end;
```

Обратите внимание на 1 у Rewrite/Reset. Конечно, мы могли бы использовать вместо неё SizeOf(Double). Но это фактически означало бы, что мы используем нетипизированный файл как типизированный. А в чём тогда смысл примера?

2. Одно значение переменного размера: String

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Value: AnsiString;
begin
  Value := 'Example';

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    BlockWrite(F, Pointer(Value)^, Length(Value));
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Value: AnsiString;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
```

```
  Reset(F, 1);
  try
    SetLength(Value, FileSize(F));
    BlockRead(F, Pointer(Value)^, Length(Value));
  finally
    CloseFile(F);
  end;

  // Здесь Value = 'Example'
end;
```

Данный случай прост - размер данных определяется по размеру файла. Для примера я выбрал AnsiString, а не String по двум причинам - во-первых, String - это псевдоним либо на AnsiString, либо на UnicodeString, в зависимости от версии Delphi. Иными словами, тут получается ситуация, аналогичная ситуации с Extended в разделе примеров для типизированных файлов. Так что вам нужно использовать явные типы - AnsiString, WideString (или UnicodeString), а не String, иначе файл, созданный в одном варианте программы, нельзя будет прочитать в другом варианте программы.

Во-вторых, используя AnsiString, я показал, как вы можете загрузить в строку весь файл целиком, "как есть". Хотя, если подобный подход использовать в реальных программах, то уж лучше использовать array of Byte или хотя бы RawByteString - чтобы подчеркнуть двоичность данных.

3. Набор однородных значений: array of Double

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Values: array of Double;
  Index: Integer;
begin
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := Random * 10;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    for Index := 0 to High(Values) do
      BlockWrite(F, Values[Index], SizeOf(Values[Index]));
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Values: array of Double;
  Index: Integer;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Reset(F, 1);
  try
    SetLength(Values, FileSize(F) div SizeOf(Values[0]));
    for Index := 0 to High(Values) do
      BlockRead(F, Values[Index], SizeOf(Values[Index]));
  finally
    CloseFile(F);
  end;

  // Здесь Values тождественно равны исходным данным из Button1Click
end;
```

Данный пример эквивалентен примеру с типизированными файлами. Только теперь мы используем

двоичный доступ, без учёта типа - так что нам приходится указывать явно все размеры. Вообще говоря, это общий принцип - работа с нетипизированными файлами и фиксированными размерами эквивалента типизированным файлам с учётом коэффициента размера.

И снова, благодаря фиксированности размеров элементов, мы можем установить размер массива ещё до чтения из файла.

Обратите внимание, что не имеет значения, какой индекс используется внутри выражения у `SizeOf`. Более того, не требуется даже наличие (существование) этого элемента. Это потому, что мы не обращаемся к нему - мы только просим у компилятора его размер. Это, по сути, константа. Так что всё выражение вообще не вычисляется - оно просто заменяется числом. Это удобный трюк для написания подобного кода, потому что это удобнее, чем писать тип явно: `SizeOf(Double)`. Почему? А что, если мы изменим объявление типа с `Double` на `Single`? И забудем обновить `SizeOf`? Тогда это приведёт к порче памяти - т.к. писаться или читаться будет больше, чем реально есть байт в элементе. Это выглядит не очень страшно для массива из `Double`, но рассмотрите вариант, скажем, строки - изменение размера `Char` гораздо более вероятно. А вот если мы используем форму `SizeOf` как в примере, то такой проблемы не будет - размер изменится автоматически.

4. Набор однородных значений переменного размера: `array of String`

```
// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Values: array of String;
  Index: Integer;
  Str: WideString;
  Len: LongInt;
begin
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := 'Str #' + IntToStr(Index);

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    for Index := 0 to High(Values) do
    begin
      Str := Values[Index];
      Len := Length(Str);

      BlockWrite(F, Len, SizeOf(Len));
      if Len > 0 then
        BlockWrite(F, Str[1], Length(Str) * SizeOf(Str[1]));
    end;
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Values: array of String;
  Str: WideString;
  Len: LongInt;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Reset(F, 1);
  try
    while not EoF(F) do
    begin
      BlockRead(F, Len, SizeOf(Len));
      SetLength(Str, Len);
```

```
46        if Len > 0 then
47          BlockRead(F, Str[1], Length(Str) * SizeOf(Str[1]));
48
49        SetLength(Values, Length(Values) + 1);
50        Values[High(Values)] := Str;
51      end;
52    finally
53      CloseFile(F);
54    end;
55
56    // Здесь Values тождественно равны исходным данным из Button1Click
57  end;
```

С записью набора динамических данных возникает проблема - как отличить один элемент от другого? Мы не можем более использовать переход на другую строку, как это было с текстовыми файлами. Тут есть несколько вариантов.

Самый простой и очевидный - ввести разделитель данных. Т.е. элементы отделяются друг от друга специальным символом. В качестве такого чаще всего выступает нулевой символ (#0) - это аналог разделителя строк в текстовых файлах. Тогда чтение-запись сведётся к примеру два. Но я не стал показывать этот путь, т.к. он очевидно вводит ограничение на возможные данные: теперь данные не могут содержать в себе разделитель (каким бы вы его ни выбрали). Конечно, вы можете его экранировать, но гораздо проще будет выбор другого подхода.

И я его показал - это явная запись размера данных до записи самих данных. Т.е. мы пишем два значения для каждого элемента: длину и сами данные.

Кроме того, в этом же примере показано, как можно сделать так, чтобы внутри программы работать с хорошо знакомым `String`, а в файле хранить фиксированный тип (`AnsiString/RawByteString` или `WideString/UnicodeString`). Вообще говоря, даже если вы работаете на Delphi 7 или любой другой версии Delphi до 2007 включительно - я бы рекомендовал всегда писать Unicode-данные в формате `WideString` во внешние хранилища.

Обратите внимание, что в качестве счётчика длины используется `LongInt`, а не `Integer` - по причинам, указанным выше для типизированных файлов: `String`, `Extended`, `Integer` и `Cardinal` могут менять свои размеры в зависимости от окружения - поэтому мы используем другие типы, которые гарантировано всегда имеют один и тот же размер.

5. Запись - набор неоднородных данных:

```
1   type
2     TData = record
3       Signature: LongWord;
4       Size: LongInt;
5       Comment: String;
6       CRC: LongWord;
7     end;
8
9   // Сохранение в файл
10  procedure TForm1.Button1Click(Sender: TObject);
11  var
12    F: file;
13    Value: TData;
14    Len: LongInt;
15    Str: WideString;
16  begin
17    Value.Signature := 123;
18    Value.Size := SizeOf(Value);
19    Value.Comment := 'Example';
20    Value.CRC := 0987654321;
21
22    AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
23    Rewrite(F, 1);
```

```
  try
    BlockWrite(F, Value.Signature, SizeOf(Value.Signature));
    BlockWrite(F, Value.Size, SizeOf(Value.Size));
    Str := Value.Comment;
    Len := Length(Str);
    BlockWrite(F, Len, SizeOf(Len));
    if Len > 0 then
      BlockWrite(F, Str[1], Length(Str) * SizeOf(Str[1]));
    BlockWrite(F, Value.CRC, SizeOf(Value.CRC));
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Value: TData;
  Len: LongInt;
  Str: WideString;
begin
  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.dat');
  Reset(F, 1);
  try
    BlockRead(F, Value.Signature, SizeOf(Value.Signature));
    BlockRead(F, Value.Size, SizeOf(Value.Size));
    BlockRead(F, Len, SizeOf(Len));
    SetLength(Str, Len);
    if Len > 0 then
      BlockRead(F, Str[1], Length(Str) * SizeOf(Str[1]));
    Value.Comment := Str;
    BlockRead(F, Value.CRC, SizeOf(Value.CRC));
  finally
    CloseFile(F);
  end;

  // Здесь Value = значениям из Button1Click
end;
```

В отличие от типизированных файлов, для нетипизированных файлов нет никаких проблем с записью неоднородных данных - вы просто пишете одно за другим. Данные фиксированного размера самодокументируются, а для динамических данных вы пишете сначала их размер, а затем сами данные. При чтении повторяете всё это в обратном порядке.

Кстати, я бы вынес запись динамических данных в отдельные служебные подпрограммы:

```
procedure BlockWriteDyn(var F: file; const AData: WideString); overload;
var
  Len: LongInt;
begin
  Len := Length(AData);
  BlockWrite(F, Len, SizeOf(Len));
  if Len > 0 then
    BlockWrite(F, AData[1], Length(AData) * SizeOf(AData[1]));
end;

procedure BlockWriteDyn(var F: file; const AData: array of Byte); overload;
...

// и так далее для каждого типа данных переменного размера, который вы используе

procedure BlockReadDyn(var F: file; out AData: String); overload;
var
  Len: LongInt;
  WS: WideString;
begin
  BlockRead(F, Len, SizeOf(Len));
  SetLength(WS, Len);
  if Len > 0 then
    BlockRead(F, WS[1], Length(WS) * SizeOf(WS[1]));
```

```
  AData := WS;
end;

procedure BlockReadDyn(var F: file; out AData: TDynByteArray); overload;
...

// и так далее для каждого типа данных переменного размера, который вы используе
```

Тогда чтение-запись свелись бы к:

```
    BlockWrite(F, Value.Signature, SizeOf(Value.Signature));
    BlockWrite(F, Value.Size, SizeOf(Value.Size));
    BlockWriteDyn(F, Value.Comment);
    BlockWrite(F, Value.CRC, SizeOf(Value.CRC));
...
    BlockRead(F, Value.Signature, SizeOf(Value.Signature));
    BlockRead(F, Value.Size, SizeOf(Value.Size));
    BlockReadDyn(F, Value.Comment);
    BlockRead(F, Value.CRC, SizeOf(Value.CRC));
```

Выглядит существенно проще и красивее, не так ли? Иллюстрация силы выделения кода в подпрограммы.

6. Набор (массив) из записей - иерархический набор данных:

```
type
  TPerson = record
    Name: String;
    Age: Integer;
    Salary: Currency;
  end;

  TPersons = array of TPerson;

// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Values: TPersons;
  Index: Integer;
begin
  SetLength(Values, 3);
  for Index := 0 to High(Values) do
  begin
    Values[Index].Name := 'Person #' + IntToStr(Index);
    Values[Index].Age := Random(20) + 20;
    Values[Index].Salary := Random(10) * 5000;
  end;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    for Index := 0 to High(Values) do
    begin
      BlockWriteDyn(F, Values[Index].Name);
      BlockWrite(F, Values[Index].Age, SizeOf(Values[Index].Age));
      BlockWrite(F, Pointer(@Values[Index].Salary)^, SizeOf(Values[Index].Salary
    end;
  finally
    CloseFile(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: file;
  Values: TPersons;
```

```
44    Value: TPerson;
45  begin
46    AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
47    Reset(F, 1);
48    try
49      while not EoF(F) do
50      begin
51        BlockReadDyn(F, Value.Name);
52        BlockRead(F, Value.Age, SizeOf(Value.Age));
53        BlockRead(F, Pointer(@Value.Salary)^, SizeOf(Value.Salary));
54
55        SetLength(Values, Length(Values) + 1);
56        Values[High(Values)] := Value;
57      end;
58    finally
59      CloseFile(F);
60    end;
61
62    // Здесь Values тождественно равны исходным данным из Button1Click
63  end;
```

Для начала хочу сразу же заметить, что странное выражение для поля Salary сделано для обхода бага Delphi. Вообще, там должно стоять просто BlockWrite(F, Values[Index].Salary, SizeOf(Values[Index].Salary)), но в настоящий момент это выражение даёт ошибку "Variable required", поэтому используется обходной путь: мы берём указатель и разыменовываем его. Вообще говоря, это NOP-операция. А смысл её заключается в потере информации о типе. Это достаточно частый трюк, когда мы хотим запустить свои шаловливые руки под капот языка, минуя информацию типа, но в данном случае он используется для более благих целей: обхода бага компилятора. Вы можете использовать BlockWrite(F, Values[Index].Salary, SizeOf(Values[Index].Salary)), если ваша версия компилятора это позволяет, или просто выбрать другой тип данных (не Currency).

В любом случае, надо заметить, что достаточно часто при записи/чтении массива записей новички пытаются сделать такую вещь, как запись элемента целиком (BlockWrite(F, Values[Index], SizeOf(Values[Index]))). Это будет работать для записей фиксированного размера, не содержащих динамические данные (указатели). Ровно как это работает для типизированных файлов. Но если в записях у вас встречаются строки, динамические массивы и другие данные-указатели, то этот подход не будет работать. Собственно, если вы используете типизированные файлы, то компилятор даже не даст вам объявить такой тип данных (file of String, например, или file of Запись, где Запись содержит String). Но суть нетипизированных файлов - в прямом доступе, минуя информацию типа. Так что по рукам за это вам никто не даст. Вместо этого код будет просто вылетать или давать неверные результаты. А проблема тут в том, что для динамических данных, поле - это просто указатель. Записывая элемент "как есть" вы запишете в файл значение указателя, но не данные, на которые он указывает. Запись в файл произойдёт нормально, но в файле вы не найдёте своих строк. Чтение из файла тоже пройдёт отлично. Но как только вы попробуете обратиться к прочитанной строке - код вылетит с access violation, потому что указатель строки указывает в космос, на мусор.

Аналогично обсуждению типизированных файлов, самый простой способ решения проблемы (но не всегда достаточный) - замена String в записях на ShortString или статический массив символов. Я не буду рассматривать этот вариант, т.к. он сводится к предыдущим примерам с записью данных фиксированного размера.

Вместо этого в примере я показал уже известную технику: запись длины строки вместе с её данными. Это избавляет вас от всех недостатков ShortString/массива символов, но даёт новый недостаток: теперь вы не можете сохранить данные одной строчкой, вам нужно писать их поле-за-полем.

7. Массив из записей внутри записи - составные данные:

```
1  type
2    TCompose = record
3      Signature: LongInt;
```

```
    Person: TPerson;
    Count: Integer;
    Related: TPersons;
  end;

  TComposes = array of TCompose;

procedure BlockWriteDyn(var F: file; const APerson: TPerson); overload;
begin
  BlockWriteDyn(F, APerson.Name);
  BlockWrite(F, APerson.Age, SizeOf(APerson.Age));
  BlockWrite(F, Pointer(@APerson.Salary)^, SizeOf(APerson.Salary));
end;

procedure BlockWriteDyn(var F: file; const APersons: TPersons); overload;
var
  Len: LongInt;
  Index: Integer;
begin
  Len := Length(APersons);
  BlockWrite(F, Len, SizeOf(Len));
  for Index := 0 to High(APersons) do
    BlockWriteDyn(F, APersons[Index]);
end;

procedure BlockReadDyn(var F: file; out APerson: TPerson); overload;
begin
  BlockReadDyn(F, APerson.Name);
  BlockRead(F, APerson.Age, SizeOf(APerson.Age));
  BlockRead(F, Pointer(@APerson.Salary)^, SizeOf(APerson.Salary));
end;

procedure BlockReadDyn(var F: file; out APersons: TPersons); overload;
var
  Len: LongInt;
  Index: Integer;
begin
  BlockRead(F, Len, SizeOf(Len));
  SetLength(APersons, Len);
  for Index := 0 to High(APersons) do
    BlockReadDyn(F, APersons[Index]);
end;

// Сохранение в файл
procedure TForm1.Button1Click(Sender: TObject);
var
  F: file;
  Values: TComposes;
  Index: Integer;
  PersonIndex: Integer;
begin
  SetLength(Values, 3);
  for Index := 0 to High(Values) do
  begin
    Values[Index].Signature := 123456;
    Values[Index].Person.Name := 'Person #' + IntToStr(Index);
    Values[Index].Person.Age := Random(10) + 20;
    Values[Index].Person.Salary := Random(10) * 5000;
    Values[Index].Count := Random(10);
    SetLength(Values[Index].Related, Random(10));
    for PersonIndex := 0 to High(Values[Index].Related) do
    begin
      Values[Index].Related[PersonIndex].Name := 'Related #' + IntToStr(Index);
      Values[Index].Related[PersonIndex].Age := Random(10) + 20;
      Values[Index].Related[PersonIndex].Salary := Random(10) * 5000;
    end;
  end;

  AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
  Rewrite(F, 1);
  try
    for Index := 0 to High(Values) do
```

```
76      begin
77        BlockWrite(F, Values[Index].Signature, SizeOf(Values[Index].Signature));
78        BlockWriteDyn(F, Values[Index].Person);
79        BlockWrite(F, Values[Index].Count, SizeOf(Values[Index].Count));
80        BlockWriteDyn(F, Values[Index].Related);
81      end;
82    finally
83      CloseFile(F);
84    end;
85  end;
86
87  // Загрузка из файла
88  procedure TForm1.Button2Click(Sender: TObject);
89  var
90    F: file;
91    Values: TComposes;
92    Value: TCompose;
93  begin
94    AssignFile(F, ExtractFilePath(GetModuleName(0)) + 'Test.bin');
95    Reset(F, 1);
96    try
97      while not EoF(F) do
98      begin
99        BlockRead(F, Value.Signature, SizeOf(Value.Signature));
100       BlockReadDyn(F, Value.Person);
101       BlockRead(F, Value.Count, SizeOf(Value.Count));
102       BlockReadDyn(F, Value.Related);
103
104       SetLength(Values, Length(Values) + 1);
105       Values[High(Values)] := Value;
106     end;
107   finally
108     CloseFile(F);
109   end;
110
111   // Здесь Values тождественно равны исходным данным из Button1Click
112 end;
```

Как видите - здесь нет никаких проблем, вы просто соединяете воедино техники из предыдущих примеров. Мы используем технику с записью счётчика длины для динамических данных в двух местах: при записи строк и при записи массивов (поле `Related`).

Кроме того, хотя я мог бы написать весь код в цикле, друг за другом, я всё же выделил новые подпрограммы - исключительно ради удобства. Код теперь выглядит компактно и аккуратно. Он прозрачен и его легко проверить. А если бы я внёс под из подпрограмм в главные циклы, то получилась бы слабочитаемая мешанина кода.

Заметьте, что вы всё ещё должны писать записи по отдельным полям. И если вы меняете объявление записи - вам лучше бы не забыть поменять код, сериализующий и десериализующий запись.

Из последнего утверждения следует, что обеспечение обратной совместимости указанными методами (чтение файла одной версии программы в более новой версии программы и наоборот) - большая головная боль. Так просто она не обеспечивается, вам специально нужно реализовывать поддержку обратной совместимости. Проще всего это делать на нетипизированных файлах. Вы пишете в начало файла заголовок, в котором указываете версию содержимого файла. Тогда любая программа сможет прочитать заголовок и определить версию данных. А дальше - либо читать их, либо откинуть как не поддерживаемые. Впрочем, это разговор для другого раза.

# Преимущества и недостатки файлов в стиле Pascal

Плюсы:

- Отлично подходят для начального изучения языка
- Вы наверняка знаете, как с ними работать, ибо это первый способ работы с файлами, который все изучают
- Удобство работы с (текстовыми) данными: форматирование и переменное число аргументов
- Гибкий подход, позволяющий работать с текстовыми, типизированными и произвольными данными
- Встроенная буферизация даёт прирост производительности при записи небольших кусочков из-за экономии на вызовах в режим ядра
- Могут быть расширены на поддержку любых файловых устройств, а не только дисковых файлов (т.е. IPC, pipes, сетевых каналов и т.п.) - путём написания своих адаптеров ввода-вывода, называемых "Text File Device Drivers". Подробнее см. Text File Device Drivers в справке Delphi

Минусы:

- Необходимость ручной сериализации данных
- Неудобная (и неоднозначная) обработка ошибок
- Поведение кода зависит от директив компилятора
- Нет поддержки Unicode и кодировок (улучшено начиная с Delphi XE2)
- Проблемы с обработкой больших файлов (более 2 Гб)
- Проблемы с глобальными переменными
- Проблемы с многопоточностью
- В некоторых случаях требуется ручной сброс буфера
- Недостаточная гибкость для некоторых задач
- Нестандартный код

Пояснение по последнему пункту: файлы Pascal пришли в Delphi из её предшественника. В них есть странная обработка ошибок, волшебные функции с переменным числом параметров, нестандартное для языка форматирование строк - всё это выбивается из обычного кода Delphi, оно выглядит иначе. Разнородный код - обычно это не хорошо.

Вывод: это отличный выбор для изучения языка и работы с файлами, но лично я для практических задач предпочитаю держаться подальше от файлов в стиле Pascal, но кто-то может считать это удобным способом. Если текстовые файлы ещё как-то оправдывают своё существование (если вы закроете глаза на их минусы), то типизированные и нетипизированные файлы практически зеркалируют потоки данных, но не имеют над ними никаких преимуществ.

Тэги Delphi, Статья

# 6 комментариев:

**Анонимный** 17 октября 2011 г., 14:36

Монументально.

И это в форме "не для начинающих"...

Ответить

**Анонимный** 17 октября 2011 г., 16:14

В этот раз без Луны? :)

Ответить

**GunSmoker** 18 октября 2011 г., 3:34

:D

Как-нибудь в следующий раз.

Ответить

**balmo** 18 октября 2011 г., 14:18

Спасибо.

Александр, а про баг компилятора (я так понял -- неверное выравнивание?) где можно прочитать подробнее?

Ответить

**GunSmoker** 18 октября 2011 г., 19:26

>>> *Александр, а про баг компилятора (я так понял -- неверное выравнивание?) где можно прочитать подробнее?*

Да нет, не выравнивание вроде. В общем, баг в том, что нельзя передать Currency как нетипизированный параметр. По мне выглядит как баг, потому что объяснения я не вижу, но может это и as designed. Я не спец по типам с плавающей запятой. Currency - довольно хитрый тип. На самом деле это Int64, просто он притворяется типом для дробей.

Насчёт почитать - не в курсе. На QC такого не нашёл.

Ответить

**Анонимный** 16 мая 2012 г., 14:01

Дано текстовый файл f, каждое слово которого отделено от других символом пробела. Записать в файл g сколько слов в тексте встречаются слова, начинающиеся и заканчивающиеся одинаковой буквой.

Ответить

Введите комментарий...

**Подпись комментария:** Аккаунт Goog ▼

**Публикация** **Просмотр**

Можно использовать некоторые HTML-теги, например:

**Жирный**

*Курсив*
<a href="http://www.example.com/">Ссылка</a>

Вам необязательно регистрироваться для комментирования - для этого просто выберите из списка "Анонимный" (для анонимного комментария) или "Имя/URL" (для указания вашего имени и (опционально) ссылки на сайт). Все прочие варианты потребуют от вас входа в вашу учётку (поддерживается OpenID).

Пожалуйста, по возможности **используйте "Имя/URL" вместо "Анонимный"**. URL можно просто не указывать.

Ваше сообщение может быть помечено как спам спам-фильтром - не волнуйтесь, оно появится после проверки администратором.

# Ссылки

Создать ссылку

Следующее Главная страница Предыдущее

Подписаться на: Комментарии к сообщению (Atom)