Блог GunSmoker-a

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

12 ноября 2011 г.

Сериализация - потоки данных

Это второй пост в серии постов про сериализацию. В этой части мы рассмотрим потоки данных.

Оглавление

Общие сведения

Если вы вспомните определение файла, то оно звучало так: "файл - это устройство с последовательным доступом, к которому можно обратиться по имени". При этом физические файлы на диске являются лишь частным случаем. Отнимите из этого определения часть "имеющие имя" - и вы получите определение **потока данных**. Поток данных представляет собой "что-то" с последовательным доступом. Вы можете читать из него данные или записывать. Некоторые потоки поддерживают позиционирование (вроде физического файла на диске), другие - нет (вроде сетевого сокета).

Потоки данных являются де-факто стандартом для обмена данными в Delphi. Всюду в своих процедурах, где вам необходимо принимать или отправлять нетипизированный набор данных, используйте потоки данных. Многие механизмы Delphi по умолчанию умеют работать именно с потоками данных, предоставляя методы вроде `LoadFromStream` и `SaveToStream` (и иногда предоставляя к ним обёртки-переходники вроде `LoadFromFile` и `SaveToFile`).

Примечание: поток данных (**stream**) не следует путать с потоком кода (**thread**), который также иногда

называют *нитью*. Они не имеют между собой ничего общего, кроме слова "поток" в названии. Если какой-то текст говорит про "потоки", не уточняя кода или данных, то значение термина должно быть ясно из контекста. Эти понятия не пересекаются, так что здесь не должно быть никаких проблем с пониманием. Замечу, что подобная путаница возможна только в русском языке, где два разных понятия переводятся одинаково. В английском языке для них используются разные слова (stream и thread).

В Delphi все потоки данных реализованы как объекты (экземпляры) классов, наследуемых от `TStream`. `TStream` является абстрактным базовым классом, который поддерживает операции чтения, записи и позиционирования, но сам при этом не умеет делать ничего. Конкретная работа реализуется его классами-наследниками.

В Delphi есть широкий набор классов, предназначенный для работы с чем угодно: от файла на диске до блока памяти. Каждый такой класс-наследник реализует базовые методы `TStream` по-своему. К примеру, при чтении из потока данных: наследник для работы с файлами вызовет функцию чтения данных с диска, а наследник для работы с блоком памяти использует процедуру `Move` для копирования данных.

Вот (неполный) список классов-наследников `TStream` (примеры):

- `TFileStream` (для работы с файлами)
- `TResourceStream` (для работы с ресурсами программы)
- `TStringStream` (для работы со строками)
- `TMemoryStream` (для работы с буфером в памяти)
- `TBlobStream` (для работы с BLOB полями)
- `TWinSocketStream` (для работы с сетевым сокетом)

По сравнению с прошлой темой, где было всего три вполне конкретных файловых типа, это может быть немного непонятно: зачем нужен какой-то абстрактный класс и классы-наследники? Очень просто: пусть вы хотите уметь загружать растровые изображения (bitmap). Но ведь рисунок может лежать не только в файле, он может быть и в ресурсах программы и в памяти. Не писать же три разных метода, которые делают одно и то же? Вот поэтому и нужен абстрактный класс. Он объявляет общий контракт, которому обязуются следовать все его наследники. Поэтому вы можете спокойно написать (один раз) код, который грузит рисунок из `TStream`. А уж вызывающий подставит вам `TFileStream` для загрузки рисунка из файла, `TResourceStream` для загрузки из ресурса и `TMemoryStream` для загрузки из памяти. В определённом смысле все эти классы-наследники представляют собой простые переходники (от общей спецификации, определённой `TStream`, до конкретного метода доступа: файл, ресурс, память, сеть и так далее; и наоборот). В общем, полиморфизм в действии.

Здесь и ниже я буду говорить в основном про физические файлы и примеры приводить в расчёте на `TFileStream` - как наиболее типичный случай. Просто имейте в виду, что всё сказанное будет применимо и к другим потокам данных.

Кроме того, потоки обеспечивают поддержку для загрузки/сохранения компонентов и форм. Именно благодаря этому механизму работает загрузка .dfm файлов в run-time. Этот механизм работает автоматически. Впрочем, вы можете использовать его и в своих целях. Но на это мы посмотрим в другой раз, потому что он тесно связан с RTTI. Это будет темой одной из следующих статей.

Общие принципы работы с потоками данных

Чтение/запись

Класс `TStream` имеет два метода для чтения данных и два метода для записи данных. Работают они обычным образом: поток читает (или пишет) указанный блок данных и сдвигает текущую позицию на размер блока данных, так что следующая операция начинается там, где закончилась предыдущая. Ничего сложного.

Итак, для чтения класс `TStream` предлагает методы `Read` и `ReadBuffer`, а для записи - методы `Write` и

WriteBuffer. Эти методы используются одинаково: первым параметром указывается буфер (это нетипизированный параметр), а вторым параметром - его размер в байтах, например:

```
procedure LoadFromStream(AStream: TStream);
var
  I: array of Integer;
begin
  // Чтение содержимого AStream в массив I
  SetLength(I, AStream.Size div SizeOf(I[0]));
  AStream.ReadBuffer(I[0], AStream.Size);

  // <- тут какая-то работа с I
end;

procedure SaveToStream(AStream: TStream);
var
  I: array of Integer;
begin
  // <- тут какая-то работа с I

  // Запись массива I в поток данных AStream
  AStream.WriteBuffer(I[0], Length(I) * SizeOf(I[0]));
end;
```

Разница между методами с "Buffer" и без него заключается в том, что методы с суффиксом "Buffer" гарантируют выполнение операции до конца. Если же это невозможно (к примеру, в файле 12 байт, а вы командуете прочитать 24 байта), то будет возбуждено исключение (см. также ниже раздел про обработку ошибок).

А вот методы без суффикса "Buffer" допускают частичное выполнение операции. Они никогда не возбуждают ошибку, а вместо этого возвращают, сколько реально байт было прочитано или записано. Иногда, это может быть и 0 байт. К примеру, если в файле 12 байт, а вы вызываете Read, указывая 24 байта, то метод Read прочитает 12 байт и вернёт вам число 12 (это метод-функция). Ещё пример: если у вас поток связан с сетевым сокетом и вы вызываете Read, но пока никаких данных ещё не пришло: метод завершится тут же, возвращая 0.

Замечу, что некоторые потоки данных могут быть только для чтения или только для записи. К примеру, однонаправленный pipe может не допускать чтения, а ресурсы, очевидно, не разрешают запись. В таких случаях попытка вызова запрещённого метода приведёт к ошибке.

Помимо общих методов чтения/записи, потоки поддерживают специализированные методы для реализации чтения/записи компонентов - но, как я уже сказал, это тема для следующего раза.

Разумеется, некоторые из наследников TStream могут вводить свои специальные методы для чтения/записи. Мы посмотрим на некоторые примеры ниже.

Кроме чтения и записи TStream имеет и некоторые другие методы.

Позиционирование

Во-первых, это методы позиционирования. Вы можете вызвать свойство Size, которое вернёт вам размер потока данных в байтах (кроме того, вы можете устанавливать свойство Size, чтобы менять размер потока, но это поддерживается далеко не всеми видами потоков данных). Свойство Position указывает текущую позицию в потоке данных, где 0 соответствует началу, а значение, равное Size, - концу. Вы можете читать свойство Position, чтобы узнать текущую позицию, и записывать значение в Position, чтобы изменить текущую позицию (позиционирование поддерживается не всеми видами потоков). К примеру:

```
var
  Stream: TStream;
  SavedPos: Int64;
begin
  ...
  SavedPos := Stream.Position; // Где мы сейчас?
  Stream.Position := 0; // Перешли в начало потока данных
```

```
  // что-то сделали...
  Stream.Position := SavedPos; // Вернулись, где были до этого
  ...
  Stream.Position := Stream.Size; // Перешли в конец потока
  ...
end;
```

Кроме абсолютного позиционирования через свойство Position, потоки данных поддерживают относительное позиционирование в стиле файловых Seek-процедур: метод Seek. Этот метод имеет два параметра: позицию и точку отсчёта. Причём последнее может иметь значения soBeginning (отсчёт от начала потока, аналог абсолютного позиционирования), soCurrent (отсчёт от текущей позиции, относительное смещение) и soEnd (отсчёт от конца потока). Для soCurrent позиция может быть и положительным числом (смещение в сторону конца потока) и отрицательным (смещение в сторону начала), для soBeginning смещение может быть только положительным (или нулём), а для soEnd - только отрицательным (или нулём). Есть также устаревшие константы вида soFromBeginning, soFromCurrent, soFromEnd - не используйте их в новом коде. При этом метод Seek возвращает предыдущее значение текущей позиции (до позиционирования). К примеру:

```
var
  Stream: TStream;
  SavedPos: Int64;
begin
  ...
  SavedPos := Stream.Seek(0, soBeginning); // Перемещаемся в начало потока, одновреме
  // что-то сделали...
  Stream.Seek(SavedPos, soBeginning); // Вернулись, где были до этого
  ...
  Stream.Seek(0, soEnd); // Перешли в конец потока
  ...
end;
```

Копирование

У TStream есть ещё один метод: CopyFrom. Этот метод копирует указанное количество данных (в байтах) из указанного массива. Копирование производится с текущей позиции. Метод работает аналогично методу записи WriteBuffer, сдвигая текущую позицию на указанное количество байт и возбуждая исключение при ошибках. Использование CopyFrom позволяет избежать создания буфера, чтения в него данных из исходного потока, запись буфера в выходной поток и удаления буфера - все эти действия выполняются автоматически внутри метода CopyFrom. К примеру, вот простейший пример копирования файлов (см. ниже описание TFileStream):

```
procedure CopyFile(const ASourceFileName, ATargetFileName: String);
var
  Source: TFileStream;
  Dest: TFileStream;
begin
  // Открыли исходный файл на чтение
  Source := TFileStream.Create(ASourceFileName, fmOpenRead or fmShareDenyWrite);
  try
    // Создали целевой файл (в режиме записи)
    Dest := TFileStream.Create(ATargetFileName, fmCreate or fmShareExclusive);
    try
      // Копируем данные из потока в поток
      Dest.CopyFrom(Source, Source.Size);
    finally
      FreeAndNil(Dest);
    end;
  finally
    FreeAndNil(Source);
  end;
end;
```

У CopyFrom есть специальный случай: если последний параметр (размер) равен 0, то CopyFrom скопирует весь поток целиком - начиная с начала потока (вне зависимости от текущей позиции) и до конца. Так что если число байт для записи у вас не фиксировано, а как-то вычисляется, то вставьте перед вызовом

CopyFrom проверку на 0: иначе вы скопируете поток целиком вместо 0 байт.

Обработка ошибок

Здесь нет ничего сложного, потоки данных используют стандартную обработку исключений. Вы можете просто писать код, не производя проверок - он по умолчанию будет иметь обработку ошибок. Конечно же, вам нужно использовать try..finally для корректной обработки утечек ресурсов программы:

```
1  // Случай 1: код в рамках одной процедуры.
2
3  begin
4    Stream := ???.Create(...);
5    try
6      // Работа с потоком
7    finally
8      FreeAndNil(Stream);
9    end;
10 end;
11
12 ...
13
14 // Случай 2: использование в конструкторах и деструкторах объектов.
15
16 constructor TSomeClass.Create;
17 begin
18   inherited Create;
19   FStream := ???.Create(...);
20 end;
21
22 procedure TSomeClass.DoSomething;
23 begin
24   // Работа с потоком
25 end;
26
27 destructor TSomeClass.Destroy;
28 begin
29   FreeAndNil(FStream);
30   inherited Destroy;
31 end;
32
33 ...
34
35 SomeObj := TSomeClass.Create;
36 try
37   SomeObj.DoSomething;
38 finally
39   FreeAndNil(SomeObj);
40 end;
```

Во втором примере try..finally не используется при работе с потоком, потому что он вынесен во внешний код (вызывающий).

Все исключения, возбуждаемые самим потоком, наследуются от EStreamError. Наиболее типичными случаями являются ошибки EFileStreamError, EFCreateError, EFOpenError, EReadError и EWriteError.

Так что код обработки исключений потоков данных (если он вам нужен) может выглядеть как-то так:

```
1  try
2    Stream := ???.Create(...);
3    try
4      // Работа с TStream
5    finally
6      FreeAndNil(Stream);
7    end;
8  except
9    on E: EStreamError do
10     Application.MessageBox(PChar(Format('При работе с потоком произошла ошибка %s', [
11 end;
```

Всё вышеуказанное - это типичное поведение для работы с исключениями. Тут не должно быть ничего нового и нетипичного.

Так что единственное "ага!" в обработке ошибок - разница между `Read` и `ReadBuffer` и между `Write` и `WriteBuffer`. Запомните, что если вы вызываете методы без суффикса "Buffer", то вы должны либо явно проверять результат их вызова (прямо как с кодом на кодах ошибок), либо иметь в виду, что данные могут прочитаться/записаться не полностью.

Правила использования

Хотелось бы сказать о "правилах использования" - а, скорее, о наиболее типичных ошибках новичков при работе с потоками данных.

1. (Почти) всегда используйте `ReadBuffer` и `WriteBuffer`. Используйте `Read` и `Write` только если вам не важно выполнение операции до конца (к примеру, вы не знаете, сколько данных нужно прочитать). Иными словами, `ReadBuffer` и `WriteBuffer` должны быть вашим вариантом по умолчанию, а не наоборот.

2. Если вы в своей процедуре принимаете или отправляете какие-то данные - используйте `TStream`. Не используйте для этого нетипизированные параметры, указатели или конкретные экземпляры `TStream`. Т.е. вместо:

```
procedure A(AData: Pointer; ADataSize: Cardinal);
procedure B(const AData; ADataSize: Cardinal);
procedure C(AData: TFileStream);
```

должно быть:

```
procedure A(AData: TStream);
procedure B(AData: TStream);
procedure C(AData: TStream);
```

3. Частая ошибка новичков: они забывают про позиционирование. К примеру:

```
Stream.WriteBuffer(MyData, SizeOf(MyData)); // записали данные для объекта Obj
Obj.LoadFromStream(Stream); // ошибка: загрузка объекта "из конца потока"
```

Должно быть:

```
Stream.WriteBuffer(MyData, SizeOf(MyData)); // записали данные для объекта Obj
Stream.Position := 0; // сместились к началу потока
Obj.LoadFromStream(Stream); // правильно: загрузка того, что мы только что запис
```

4. `TStream` является абстрактным классом. Это значит, что в вашем коде не должно быть строк вида `TStream.Create`. Вы всегда должны использовать конкретного наследника - например, `TFileStream` или `TResourceStream`. Если вам нужен "просто поток", то вы можете использовать `TMemoryStream` - это создаст поток в памяти программы. Если при этом вы хотите использовать большой объём данных, то вы можете использовать `TFileStream` для временного файла. См. также.

5. Не забывайте, что потоки работают с большими данными, поэтому всё позиционирование осуществляется на базе `Int64` (8 байт/64 бита), а не `Integer` (4 байта/32 бита). Поэтому если вы по недосмотру где-то используете для позиционирования выражение/переменную типа `Integer`, то этим вы автоматически ограничите свои данные 2 Гб. Но этот момент также зависит и от вида используемого потока. К примеру, в 32-битных приложениях `TMemoryStream` не может работать с памятью больше 4 Гб, а `TResourceStream` ограничен 2 Гб - потому что он работает с ресурсами в исполняемых файлах формата PE. А любой файл формата PE не может быть больше 2 Гб.

6. Ну и под конец упомяну ещё такой момент: поскольку базовые методы чтения/записи работают с нетипизированными параметрами, то тут возможны ошибки при использовании динамических типов и указателей. Я уже приводил подобный пример в предыдущей статье, но приведу его ещё раз и тут:

```
var
  StatArray: array[0..15] of Integer;
  DynArray: array of Integer;
  AnsiStr: AnsiString;
  Str: String;
  PCh: PChar;
  Stream: TStream;
...
  // Неправильно (порча памяти):
  Stream.WriteBuffer(DynArray, Length(DynArray) * SizeOf(Integer));
  Stream.WriteBuffer(AnsiStr, Length(AnsiStr));
  Stream.WriteBuffer(Str, Length(Str) * SizeOf(Char));
  Stream.WriteBuffer(PCh, StrLen(PCh) * SizeOf(Char));

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

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

  // Неправильно (неверный размер 1):
  Stream.WriteBuffer(Pointer(DynArray)^, SizeOf(DynArray));
  Stream.WriteBuffer(DynArray[0], SizeOf(DynArray));
  Stream.WriteBuffer(Pointer(AnsiStr)^, SizeOf(AnsiStr));
  Stream.WriteBuffer(AnsiStr[1], SizeOf(AnsiStr));
  Stream.WriteBuffer(Pointer(Str)^, SizeOf(Str));
  Stream.WriteBuffer(Str[1], SizeOf(Str));

  // Неправильно (неверный размер 2):
  Stream.WriteBuffer(Pointer(Str)^, Length(Str));
  Stream.WriteBuffer(Str[1], Length(Str));
```

Практика

Все примеры ниже приводятся с полной обработкой ошибок. 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.

Как вы сейчас увидите, использование базовых методов потоков данных эквивалентно использованию нетипизированных файлов со всеми их преимуществами и недостатками. Впрочем, благодаря классам и наследованию, классы-наследники могут предоставлять другие методы и более удобный доступ для частных случаев - подробнее на это мы посмотрим ниже, а также в следующих статьях цикла. Также мы увидим как ООП предоставляет нам возможность использования некоторых интересных ходов для записи сложных динамических структур данных.

1. Одно значение: Double

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

  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
  try
    F.WriteBuffer(Value, SizeOf(Value));
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Value: Double;
begin
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    F.ReadBuffer(Value, SizeOf(Value));
  finally
    FreeAndNil(F);
  end;

  // Здесь Value = 5.5
end;
```

2. Одно значение переменного размера: String

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

  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
  try
    F.WriteBuffer(Pointer(Value)^, Length(Value));
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Value: AnsiString;
begin
```

```
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    SetLength(Value, F.Size);
    F.ReadBuffer(Pointer(Value)^, Length(Value));
  finally
    FreeAndNil(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). 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: TFileStream;
  Values: array of Double;
  Index: Integer;
begin
  SetLength(Values, 10);
  for Index := 0 to High(Values) do
    Values[Index] := Random * 10;

  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
  try
    for Index := 0 to High(Values) do
      F.WriteBuffer(Values[Index], SizeOf(Values[Index]));
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Values: array of Double;
  Index: Integer;
begin
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    SetLength(Values, F.Size div SizeOf(Values[0]));
    for Index := 0 to High(Values) do
      F.ReadBuffer(Values[Index], SizeOf(Values[Index]));
  finally
    FreeAndNil(F);
  end;

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

И снова, благодаря фиксированности размеров элементов, мы можем установить размер массива ещё до чтения из файла. Обратите внимание, что мы могли бы свести этот пример к предыдущему,

прочитав/записав весь массив за раз, вместо поэлементного копирования.

Также 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: TFileStream;
  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);

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

      F.WriteBuffer(Len, SizeOf(Len));
      if Len > 0 then
        F.WriteBuffer(Str[1], Length(Str) * SizeOf(Str[1]));
    end;
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Values: array of String;
  Str: WideString;
  Len: LongInt;
begin
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    while F.Read(Len, SizeOf(Len)) > 0 do
    begin
      SetLength(Str, Len);
      if Len > 0 then
        F.ReadBuffer(Str[1], Length(Str) * SizeOf(Str[1]));

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

```

```
53    // Здесь Values тождественно равны исходным данным из Button1Click
54  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 гарантированно всегда имеют один и тот же размер.

Ещё в этом примере показан вариант пример использования метода `Read`: идея в том, что если будет достигнут конец потока, то вызов метода `Read` вернёт 0. Т.е. это аналог функции EoF. Альтернативным решением является код

```
1  ...
2  while F.Position < F.Size do
3  begin
4    F.ReadBuffer(Len, SizeOf(Len));
5    SetLength(Str, Len);
6  ...
```

Вы также можете создать вспомогательную функцию

```
1  function EoS(Stream: TStream): Boolean; // End of Stream
2  begin
3    Result := (Stream.Position >= Stream.Size);
4  end;
5      ...
6      while not EoS(F) do
7      begin
8        F.ReadBuffer(Len, SizeOf(Len));
9        SetLength(Str, Len);
10     ...
```

И тогда этот пример будет эквивалентен примеру для нетипизированных файлов Pascal. Я буду использовать подпрограмму EoS в примерах ниже.

5. Запись - набор неоднородных данных:

```
1  type
2    TData = record
3      Signature: LongWord;
```

```
    Size: LongInt;
    Comment: String;
    CRC: LongWord;
  end;

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

  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
  try
    F.WriteBuffer(Value.Signature, SizeOf(Value.Signature));
    F.WriteBuffer(Value.Size, SizeOf(Value.Size));
    Str := Value.Comment;
    Len := Length(Str);
    F.WriteBuffer(Len, SizeOf(Len));
    if Len > 0 then
      F.WriteBuffer(Str[1], Length(Str) * SizeOf(Str[1]));
    F.WriteBuffer(Value.CRC, SizeOf(Value.CRC));
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Value: TData;
  Len: LongInt;
  Str: WideString;
begin
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    F.ReadBuffer(Value.Signature, SizeOf(Value.Signature));
    F.ReadBuffer(Value.Size, SizeOf(Value.Size));
    F.ReadBuffer(Len, SizeOf(Len));
    SetLength(Str, Len);
    if Len > 0 then
      F.ReadBuffer(Str[1], Length(Str) * SizeOf(Str[1]));
    Value.Comment := Str;
    F.ReadBuffer(Value.CRC, SizeOf(Value.CRC));
  finally
    FreeAndNil(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 WriteBufferDyn(F: TFileStream; const AData: WideString); overload;
var
  Len: LongInt;
begin
```

```
  Len := Length(AData);
  F.WriteBuffer(Len, SizeOf(Len));
  if Len > 0 then
    F.WriteBuffer(AData[1], Length(AData) * SizeOf(AData[1]));
end;

procedure WriteBufferDyn(F: TFileStream; const AData: array of Byte); overload;
...

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

procedure ReadBufferDyn(F: TFileStream; out AData: String); overload;
var
  Len: LongInt;
  WS: WideString;
begin
  F.ReadBuffer(Len, SizeOf(Len));
  SetLength(WS, Len);
  if Len > 0 then
    F.ReadBuffer(WS[1], Length(WS) * SizeOf(WS[1]));
  AData := WS;
end;

procedure ReadBufferDyn(F: TFileStream; out AData: TDynByteArray); overload;
...

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

Тогда чтение-запись свелись бы к:

```
    F.WriteBuffer(Value.Signature, SizeOf(Value.Signature));
    F.WriteBuffer(Value.Size, SizeOf(Value.Size));
    WriteBufferDyn(F, Value.Comment);
    F.WriteBuffer(Value.CRC, SizeOf(Value.CRC));
...
    F.ReadBuffer(Value.Signature, SizeOf(Value.Signature));
    F.ReadBuffer(Value.Size, SizeOf(Value.Size));
    ReadBufferDyn(F, Value.Comment);
    F.ReadBuffer(Value.CRC, SizeOf(Value.CRC));
```

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

Вообще, конечно же, более правильный код получится при использовании шаблона "декоратор". Суть заключается в создании класса, который реализует методы вида WriteBufferDyn/ReadBufferDyn, выполняя их над потоком, который ему указали. Например:

```
type
  TStreamDataDecorator = class
  private
    // Управление потоком, с которым мы будем работать
    FStream: TStream;
    FOwn: Boolean;
    procedure SetStream(const Value: TStream);

    // Вспомогательные методы: добавьте свои на каждый тип динамических данных
    procedure ReadBuffer(var ABuffer; ABufferSize: Integer); overload;
    procedure ReadBuffer(out AStr: String); overload;
    procedure WriteBuffer(const ABuffer; ABufferSize: Integer); overload;
    procedure WriteBuffer(const AStr: WideString); overload;
  public
    // Декоратор
    constructor Create(const AStream: TStream = nil; const AOwn: Boolean = True)
    destructor Destroy; override;

    // Просто для удобства
    property Stream: TStream read FStream write SetStream;
    property Own: Boolean read FOwn write FOwn;

```

```
23      // Вот ради чего всё делалось: высокоуровневый код
24      procedure Write(const AData: TData);
25      procedure Read(out AData: TData);
26      function EoS: Boolean;
27    end;
28
29  { TStreamDataDecorator }
30
31  constructor TStreamDataDecorator.Create(const AStream: TStream;
32    const AOwn: Boolean);
33  begin
34    inherited Create;
35    FStream := AStream;
36    FOwn := AOwn;
37  end;
38
39  destructor TStreamDataDecorator.Destroy;
40  begin
41    Stream := nil;
42    inherited Destroy;
43  end;
44
45  procedure TStreamDataDecorator.SetStream(const Value: TStream);
46  begin
47    if Assigned(FStream) and FOwn then
48      FreeAndNil(FStream);
49    FStream := Value;
50    FOwn := False;
51  end;
52
53  procedure TStreamDataDecorator.Read(out AData: TData);
54  begin
55    Assert(Assigned(FStream));
56
57    ReadBuffer(AData.Signature, SizeOf(AData.Signature));
58    ReadBuffer(AData.Size, SizeOf(AData.Size));
59    ReadBuffer(AData.Comment);
60    ReadBuffer(AData.CRC, SizeOf(AData.CRC));
61  end;
62
63  procedure TStreamDataDecorator.Write(const AData: TData);
64  begin
65    Assert(Assigned(FStream));
66
67    WriteBuffer(AData.Signature, SizeOf(AData.Signature));
68    WriteBuffer(AData.Size, SizeOf(AData.Size));
69    WriteBuffer(AData.Comment);
70    WriteBuffer(AData.CRC, SizeOf(AData.CRC));
71  end;
72
73  function TStreamDataDecorator.EoS: Boolean;
74  begin
75    Assert(Assigned(FStream));
76    Result := (FStream.Position >= FStream.Size);
77  end;
78
79  procedure TStreamDataDecorator.ReadBuffer(var ABuffer; ABufferSize: Integer);
80  begin
81    Assert(Assigned(FStream));
82
83    FStream.ReadBuffer(ABuffer, ABufferSize);
84  end;
85
86  procedure TStreamDataDecorator.ReadBuffer(out AStr: String);
87  var
88    Len: LongInt;
89    Str: WideString;
90  begin
91    ReadBuffer(Len, SizeOf(Len));
92    SetLength(Str, Len);
93    if Len > 0 then
94      ReadBuffer(Str[1], Length(Str) * SizeOf(Str[1]));
```

```
95    AStr := Str;
96  end;
97
98  procedure TStreamDataDecorator.WriteBuffer(const ABuffer; ABufferSize: Integer);
99  begin
100   Assert(Assigned(FStream));
101
102   FStream.WriteBuffer(ABuffer, ABufferSize);
103 end;
104
105 procedure TStreamDataDecorator.WriteBuffer(const AStr: WideString);
106 var
107   Len: LongInt;
108 begin
109   Len := Length(AStr);
110   WriteBuffer(Len, SizeOf(Len));
111   if Len > 0 then
112     WriteBuffer(AStr[1], Length(AStr) * SizeOf(AStr[1]));
113 end;
```

В этом примере реализовано 3 вещи: во-первых, мы реализовали хранение ссылки на поток данных, над которым будем выполнять операции. Во-вторых, мы реализовали несколько вспомогательных низкоуровневых операций в секции `private`. В-третьих, мы реализовали высокоуровневые операции `Read` и `Write` в секции `public`. Тогда сохранение и загрузка сводятся к такому простому коду:

```
1   // Сохранение в файл
2   procedure TForm1.Button1Click(Sender: TObject);
3   var
4     F: TStreamDataDecorator;
5     Value: TData;
6   begin
7     Value.Signature := 123;
8     Value.Size := SizeOf(Value);
9     Value.Comment := 'Example';
10    Value.CRC := 0987654321;
11
12    F := TStreamDataDecorator.Create(TFileStream.Create(ExtractFilePath(GetModuleN
13    try
14      F.Write(Value);
15    finally
16      FreeAndNil(F);
17    end;
18  end;
19
20  // Загрузка из файла
21  procedure TForm1.Button2Click(Sender: TObject);
22  var
23    F: TStreamDataDecorator;
24    Value: TData;
25  begin
26    F := TStreamDataDecorator.Create(TFileStream.Create(ExtractFilePath(GetModuleN
27    try
28      F.Read(Value);
29    finally
30      FreeAndNil(F);
31    end;
32
33    // Здесь Value = значениям из Button1Click
34  end;
```

Ну не красота-ли?

Имейте в виду, что именно этот подход вам нужно применять в реальных программах. В примерах ниже я буду применять процедуры вроде `WriteBufferDyn/ReadBufferDyn` исключительно по соображениям "меньше писать".

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: TFileStream;
  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;

  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
  try
    for Index := 0 to High(Values) do
    begin
      WriteBufferDyn(F, Values[Index].Name);
      F.WriteBuffer(Values[Index].Age, SizeOf(Values[Index].Age));
      F.WriteBuffer(Pointer(@Values[Index].Salary)^, SizeOf(Values[Index].Salary
    end;
  finally
    FreeAndNil(F);
  end;
end;

// Загрузка из файла
procedure TForm1.Button2Click(Sender: TObject);
var
  F: TFileStream;
  Values: TPersons;
  Value: TPerson;
begin
  F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
  try
    while not EoS(F) do
    begin
      ReadBufferDyn(F, Value.Name);
      F.ReadBuffer(Value.Age, SizeOf(Value.Age));
      F.ReadBuffer(Pointer(@Value.Salary)^, SizeOf(Value.Salary));

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

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

Для начала хочу сразу же заметить, что странное выражение для поля Salary сделано для обхода бага Delphi. Вообще, там должно стоять просто F.WriteBuffer(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 F.WriteBuffer(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 (F.WriteBuffer(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 потоков данных - 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 предыдущим обсуждениям, 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.

Также по аналогии с предыдущим примером я покажу, как будет выглядеть код, если вы введёте класс-декоратор. Само описание класса я опущу для краткости - оно аналогично (но не тождественно) предыдущему примеру:

```
type
  TStreamPersonDecorator = class
  ...
    procedure Write(const AData: TPerson);
    procedure Read(out AData: TPerson);
  ...
  end;

...

procedure TStreamPersonDecorator.Read(out AData: TPerson);
begin
  Assert(Assigned(FStream));

  ReadBuffer(AData.Name);
  ReadBuffer(AData.Age, SizeOf(AData.Age));
  ReadBuffer(AData.Salary, SizeOf(AData.Salary));
end;

procedure TStreamPersonDecorator.Write(const AData: TPerson);
begin
  Assert(Assigned(FStream));

  WriteBuffer(AData.Name);
  WriteBuffer(AData.Age, SizeOf(AData.Age));
  WriteBuffer(AData.Salary, SizeOf(AData.Salary));
end;

...
```

```
30
31  // Сохранение в файл
32  procedure TForm1.Button1Click(Sender: TObject);
33  var
34    F: TStreamPersonDecorator;
35    Values: TPersons;
36    Index: Integer;
37  begin
38    SetLength(Values, 3);
39    for Index := 0 to High(Values) do
40    begin
41      Values[Index].Name := 'Person #' + IntToStr(Index);
42      Values[Index].Age := Random(20) + 20;
43      Values[Index].Salary := Random(10) * 5000;
44    end;
45
46    F := TStreamPersonDecorator.Create(TFileStream.Create(ExtractFilePath(GetModul
47    try
48      for Index := 0 to High(Values) do
49        F.Write(Values[Index]);
50    finally
51      FreeAndNil(F);
52    end;
53  end;
54
55  // Загрузка из файла
56  procedure TForm1.Button2Click(Sender: TObject);
57  var
58    F: TStreamPersonDecorator;
59    Values: TPersons;
60    Value: TPerson;
61  begin
62    F := TStreamPersonDecorator.Create(TFileStream.Create(ExtractFilePath(GetModul
63    try
64      while not F.EoS do
65      begin
66        F.Read(Value);
67
68        SetLength(Values, Length(Values) + 1);
69        Values[High(Values)] := Value;
70      end;
71    finally
72      FreeAndNil(F);
73    end;
74
75    // Здесь Values тождественно равны исходным данным из Button1Click
76  end;
```

Мы также могли внести методы Read и Write, работающие с массивом из TPerson в класс-декоратор.

7. Массив из записей внутри записи - составные данные:

```
1   type
2     TCompose = record
3       Signature: LongInt;
4       Person: TPerson;
5       Count: Integer;
6       Related: TPersons;
7     end;
8
9     TComposes = array of TCompose;
10
11  procedure WriteBufferDyn(F: TFileStream; const APerson: TPerson); overload;
12  begin
13    WriteBufferDyn(F, APerson.Name);
14    F.WriteBuffer(APerson.Age, SizeOf(APerson.Age));
15    F.WriteBuffer(Pointer(@APerson.Salary)^, SizeOf(APerson.Salary));
16  end;
17
```

```
18  procedure WriteBufferDyn(F: TFileStream; const APersons: TPersons); overload;
19  var
20    Len: LongInt;
21    Index: Integer;
22  begin
23    Len := Length(APersons);
24    F.WriteBuffer(Len, SizeOf(Len));
25    for Index := 0 to High(APersons) do
26      WriteBufferDyn(F, APersons[Index]);
27  end;
28
29  procedure ReadBufferDyn(F: TFileStream; out APerson: TPerson); overload;
30  begin
31    ReadBufferDyn(F, APerson.Name);
32    F.ReadBuffer(APerson.Age, SizeOf(APerson.Age));
33    F.ReadBuffer(Pointer(@APerson.Salary)^, SizeOf(APerson.Salary));
34  end;
35
36  procedure ReadBufferDyn(F: TFileStream; out APersons: TPersons); overload;
37  var
38    Len: LongInt;
39    Index: Integer;
40  begin
41    F.ReadBuffer(Len, SizeOf(Len));
42    SetLength(APersons, Len);
43    for Index := 0 to High(APersons) do
44      ReadBufferDyn(F, APersons[Index]);
45  end;
46
47  // Сохранение в файл
48  procedure TForm1.Button1Click(Sender: TObject);
49  var
50    F: TFileStream;
51    Values: TComposes;
52    Index: Integer;
53    PersonIndex: Integer;
54  begin
55    SetLength(Values, 3);
56    for Index := 0 to High(Values) do
57    begin
58      Values[Index].Signature := 123456;
59      Values[Index].Person.Name := 'Person #' + IntToStr(Index);
60      Values[Index].Person.Age := Random(10) + 20;
61      Values[Index].Person.Salary := Random(10) * 5000;
62      Values[Index].Count := Random(10);
63      SetLength(Values[Index].Related, Random(10));
64      for PersonIndex := 0 to High(Values[Index].Related) do
65      begin
66        Values[Index].Related[PersonIndex].Name := 'Related #' + IntToStr(Index);
67        Values[Index].Related[PersonIndex].Age := Random(10) + 20;
68        Values[Index].Related[PersonIndex].Salary := Random(10) * 5000;
69      end;
70    end;
71
72    F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmCrea
73    try
74      for Index := 0 to High(Values) do
75      begin
76        F.WriteBuffer(Values[Index].Signature, SizeOf(Values[Index].Signature));
77        WriteBufferDyn(F, Values[Index].Person);
78        F.WriteBuffer(Values[Index].Count, SizeOf(Values[Index].Count));
79        WriteBufferDyn(F, Values[Index].Related);
80      end;
81    finally
82      FreeAndNil(F);
83    end;
84  end;
85
86  // Загрузка из файла
87  procedure TForm1.Button2Click(Sender: TObject);
88  var
89    F: TFileStream;
```

```
90    Values: TComposes;
91    Value: TCompose;
92  begin
93    F := TFileStream.Create(ExtractFilePath(GetModuleName(0)) + 'Test.bin', fmOpen
94    try
95      while not EoS(F) do
96      begin
97        F.ReadBuffer(Value.Signature, SizeOf(Value.Signature));
98        ReadBufferDyn(F, Value.Person);
99        F.ReadBuffer(Value.Count, SizeOf(Value.Count));
100       ReadBufferDyn(F, Value.Related);
101
102       SetLength(Values, Length(Values) + 1);
103       Values[High(Values)] := Value;
104     end;
105   finally
106     FreeAndNil(F);
107   end;
108
109   // Здесь Values тождественно равны исходным данным из Button1Click
110 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 код 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.

И снова: не забывайте, что это только пример. В реальных программах вам следует посмотреть в сторону класса-декоратора. Я не буду приводить тут пример: он делается по аналогии с предыдущими случаями. Просто добавьте новые методы чтения/записи, вот и всё.

Особенности

Посмотрим теперь на различные конкретные наследники класса `TStream` и то, что они нам предлагают.

Примечание: не все из нижеперечисленных возможностей существуют во всех версиях Delphi. Если у вас старая версия Delphi, то у вас могут отсутствовать некоторые описываемые классы, свойства или методы. В этом случае вам придётся обходится без них или писать аналоги самостоятельно.

THandleStream

`THandleStream` предназначен для работы с файлами в смысле операционной системы.

Короче говоря, `THandleStream` является оболочкой к файлам в стиле ОС. Объект этого типа можно связать с `THandle`, полученный любым способом - скажем, от `CreateFile`, `CreatePipe` и так далее: лишь бы на этот описатель можно было вызывать `ReadFile` и `WriteFile`.

Используйте `THandleStream` в двух случаях:

1. Функция ОС вернула вам описатель `THandle`, а код, который вы хотите вызвать, требует `TStream`. Тогда просто создайте `THandleStream`, передав в его конструктор описатель от функции ОС, и

передайте полученный объект коду. Например:

```
var
  ReadHandle, WriteHandle: THandle;
  ReadStream, WriteStream: TStream;
begin
  Win32Check(CreatePipe(@ReadHandle, @WriteHandle, nil, 0));
  try
    ReadStream := THandleStream.Create(ReadHandle);
    WriteStream := THandleStream.Create(WriteHandle);
    try
      ...
      Bitmap.SaveToStream(WriteStream); // Отправляем растровое изображение по р
      ...
    finally
      FreeAndNil(WriteStream);
      FreeAndNil(ReadStream);
    end;
  finally
    CloseHandle(WriteHandle);
    CloseHandle(ReadHandle);
  end;
end;
```

2. Вы хотите использовать возможность, которую предоставляет функция ОС, но не объект Delphi. См. первый пример в следующем пункте.

Сам THandleStream не имеет ограничений и поддерживает все операции: чтение, запись, позиционирование и изменение размера. Однако нижележащий дескриптор объекта ядра может поддерживать не все операции. К примеру, описатель от файла на диске может быть открыт только для чтения, а описатель pipe не поддерживает позиционирование.

Описатель, которым инициализирован объект, доступен через свойство Handle.

Обратите внимание, что сам THandleStream никогда не закрывает описатель (ему нельзя передать ответственность за него). Поэтому вы должны закрывать описатель вручную или использовать TFileStream (см. ниже).

TFileStream

TFileStream предназначен для работы с файлами на диске.

TFileStream наследуется от THandleStream, так что он получает все возможности своего предка. TFileStream не имеет ограничений на операции и поддерживает конструктор с передачей описателя. Например:

```
var
  FileHandle: THandle;
  Stream: TFileStream;
begin
  FileHandle := CreateFile('...\Temp.tmp', GENERIC_READ or GENERIC_WRITE, 0, nil, CRE
  Win32Check(FileHandle <> INVALID_FILE_HANDLE);
  Stream := TFileStream.Create(FileHandle);
  try
    ...
  finally
    FreeAndNil(Stream);
    // <- не нужно делать CloseHandle(FileHandle);
  end;
end;
```

В отличие от THandleStream, TFileStream закрывает открытый описатель файла, вне зависимости от

способа его инициализации.

Кроме того, TFileStream поддерживает перегруженный конструктор, позволяющий открывать файлы. У него есть два параметра: (имя файла) и (режим открытия + режим разделения).

Допустимые режимы открытия:

1. fmCreate - создаёт новый файл. Если такой файл уже есть, он удаляется перед созданием. Файл открывается в режиме записи.
2. fmOpenRead - открывает файл только для чтения.
3. fmOpenWrite - открывает файл только для записи.
4. fmOpenReadWrite - открывает файл для чтения-записи.

Режимы разделения:

1. fmShareExclusive - запретить совместное использование.
2. fmShareDenyWrite - запретить другим приложениям запись.
3. fmShareDenyRead - запретить другим приложениям чтение.
4. fmShareDenyNone - не вводить ограничения.

Режимы следует соединять друг с другом операцией or.

Примечание: у TFileStream есть вариант конструктора с дополнительным параметром, но этот (третий) параметр существует только для обратной совместимости и сейчас игнорируется. Не следует пытаться передавать в него режим разделения файла.

Флаги разделения используют устаревшую deny-семантику MS-DOS, в отличие от современного API. См. также: взаимодействие флагов режима открытия и разделения.

Типичный пример открытия файла выглядит так:

```
1 TFileStream.Create('MyFile.dat', fmOpenRead or fmShareDenyWrite);
```

А создания файла - так:

```
1 TFileStream.Create('MyFile.dat', fmCreate or fmShareExclusive);
```

При работе с файлами типа логов (для которых необходимы совместное использование или мониторинг) могут использоваться такие вызовы:

```
1 TFileStream.Create('MyFile.log', fmOpenRead or fmShareDenyNone);
2 TFileStream.Create('MyFile.log', fmCreate or fmShareDenyNone);
```

Далеко не все возможности функций открытия файлов ОС доступны через конструктор TFileStream, но зато он является универсальным для любых платформ. Предпочтительно использовать именно его для доступа к файлам вместо функций ОС. Если же вам нужны какие-либо возможности, недоступные через конструктор TFileStream, то используйте TFileStream, инициализировав его описателем файла от системной функции открытия файлов, как указано в примерах выше.

Если объект TFileStream создавался через конструктор с именем файла, то это имя файла доступно в свойстве FileName, иначе доступно только свойство Handle.

TMemoryStream

TMemoryStream реализует потоковую обёртку к данным в памяти программы. Т.е. к буферу в динамической памяти осуществляется последовательный доступ.

Используйте TMemoryStream, если вам нужен "просто поток" или вам нужен промежуточный буфер-поток.

TMemoryStream имеет конструктор без параметров, который создаёт пустой объект. Для заполнения потока после создания можно использовать обычные методы записи или же специальные методы LoadFromStream (аналог вызова CopyFrom(Stream, 0)) и LoadFromFile.

Также есть методы SaveToStream и SaveToFile.

TMemoryStream не имеет ограничений и поддерживает все операции, включая изменение размера.

TMemoryStream хранит данные в динамической куче процесса, выделяя память по мере необходимости. Он сам автоматически управляет памятью. Реально памяти может быть выделено больше, чем лежит данных в потоке - т.н. "capacity > size". Это стандартная оптимизация для "побайтовых записей".

Дополнительной возможностью TMemoryStream является предоставление свойства Memory, позволяющего обратиться к данным потока напрямую, через указатель, минуя последовательные методы чтения/записи. По этой причине вы можете рассматривать TMemoryStream как "переходник" между TStream и нетипизированным указателем.

Простейший пример использования TMemoryStream (в данном случае - для конвертации строки в TStream):

```
1  procedure LoadFromText(const AHandle: THandle; const AText: AnsiString);
2  var
3    Data: TMemoryStream;
4  begin
5    Data := TMemoryStream.Create;
6    try
7      Data.Write(PAnsiChar(AText)^, Length(AText));
8      Data.Position := 0;
9      LoadFromStream(AHandle, Data);
10   finally
11     FreeAndNil(Data);
12   end;
13 end;
```

Примечание: в данном примере более эффективной была бы другая конструкция. Данный пример по сути копирует данные строки в поток. Но поскольку реально нам нужно здесь только чтение, то можно поступить по другому: создать поток, который будет работать прямо поверх данных строки, без их копирования. Это будет настоящий адаптер. Мы посмотрим на такой пример ниже.

TResourceStream

TResourceStream предназначен для организации последовательного доступа к ресурсам в исполняемых файлах.

TResourceStream похож на TMemoryStream. Он тоже работает с памятью программы (только не с кучей, а с ресурсами), он поддерживает методы SaveToStream и SaveToFile, а также свойство Memory.

Но в отличие от TMemoryStream, TResourceStream поддерживает только методы чтения, но не записи, а также не поддерживает изменение размера. Иными словами, TResourceStream - это read-only.

Собственно для инициализации у TResourceStream есть два варианта конструктора, которые имеют по три параметра: описатель модуля, имя ресурса и тип ресурса. А разница между ними заключается в способе указания имени ресурса (второго параметра): по ID или по имени.

Ну и несколько примеров:

```
1  // Пример на использование метода SaveToFile
2
3  // Извлечение своего ресурса в отдельный файл:
4  procedure ExtractRes(const ResType: PChar; const ResName, ResNewFileName: String);
5  var
```

```
  Res: TResourceStream;
begin
  Res := TResourceStream.Create(HInstance, ResName, ResType);
  try
    Res.SaveToFile(ResNewFileName);
  finally
    FreeAndNil(Res);
  end;
end;

...

ExtractRes('BINFILE', 'MYFILE', 'C:\MyFile.bin');
ExtractRes(RT_RCDATA, 'MYDATA', 'C:\MyData.bin');
```

```
// Пример на использование свойства Memory

// Проигрывание MP3 прямо из ресурса, не выгружая его в файл (с использованием BASS)
var
  Res: TResourceStream;
  BkMusic: HSAMPLE;
...
  // Создали обёртку TStream
  Res := TResourceStream.Create(HInstance, 'BKMUSIC', RT_RCDATA);
  // Загружаем напрямую из памяти - нет копирования данных
  BkMusic := BASS_SampleLoad(True, Res.Memory, 0, Res.Size, 1, BASS_SAMPLE_LOOP);
  // Поехали! (C)
  if BkMusic <> 0 then
    BASS_ChannelPlay(BkMusic, False);
...
  if BkMusic <> 0 then
  begin
    BASS_ChannelStop(BkMusic);
    BASS_SampleFree(BkMusic);
    BkMusic := 0;
  end;
  FreeAndNil(Res);
...
```

TBytesStream

TBytesStream хранит данные потока в массиве байтов.

Используйте TBytesStream как переходник между TBytes и TStream.

Собственно, TBytesStream аналогичен TMemoryStream, только вместо блока памяти в куче он использует TBytes в качестве хранилища для данных. Он также не имеет ограничений на операции, поддерживая чтение, запись, позиционирование и изменение размера. Он поддерживает методы LoadFromStream, LoadFromFile, SaveToStream и SaveToFile, а также свойство Memory.

В отличие от TMemoryStream, у TBytesStream есть конструктор, который принимает переменную типа TBytes - это будут начальные данные потока. При этом не производится копирование данных (используется счётчик ссылок динамического массива). Все операции чтения-записи будут оперировать с исходными данными в оригинальной переменной типа TBytes. Однако если вы измените размер потока (либо явно через Size/SetSize, либо неявно через запись данных в конец потока данных), то **поток сделает копию данных** и будет работать уже с ней. При этом все будущие изменения в потоке не затронут оригинальной переменной типа TBytes.

Вы также можете передать nil в конструктор, чтобы инициализировать пустой поток. В этом случае он не будет связан с переменной.

Дополнительно TBytesStream вводит свойство Bytes оно работает аналогично свойству Memory, только имеет тип TBytes, а не Pointer. **Предупреждение:** не пытайтесь использовать Length для определения

размера данных. Размер хранилища может быть больше актуального размера ("Capacity > Size"). Используйте свойство Size для определения размера данных.

Простой пример использования TBytesStream как переходника (обратите внимание на усечение данных до их актуального размера, указанного в свойстве Size):

```
function DecodeBase64(const Input: AnsiString): TBytes;
var
  InStr: TPointerStream;
  OutStr: TBytesStream;
  Len: Integer;
begin
  InStr := TPointerStream.Create(PAnsiChar(Input), Length(Input));
  try
    OutStr := TBytesStream.Create;
    try
      DecodeStream(InStr, OutStr);
      Result := OutStr.Bytes;
      Len := OutStr.Size;
    finally
      FreeAndNil(OutStr);
    end;
  finally
    FreeAndNil(InStr);
  end;
  SetLength(Result, Len);
end;
```

TBytes является динамическим массивом, т.е. автоуправляемым типом. Явно освобождать его не нужно, заботиться о вопросах владения - тоже.

TStringStream

TStringStream предоставляет последовательный доступ к информации, хранящейся в обычной строке.

Используйте TStringStream для хранения данных в строках. Использование TStringStream даст вам в руки мощные возможности TStream. TStringStream удобен как промежуточный объект, который умеет хранить данные в строке, а также читать и писать их.

По сути, TStringStream является обёрткой к TBytesStream, которая просто конвертирует строку в байты и обратно. У TStringStream есть несколько вариантов конструкторов, которые инициализируют поток по разным типам строк. В Unicode версиях Delphi конструкторы также позволяют вам указывать кодировку для ANSI строк.

Методы чтения-записи TStringStream не затрагивают исходную строку, а всегда работают с копией данных (внутреннее хранилище в виде TBytes).

Ну и, конечно же, TStringStream предоставляет строко-ориентированные свойства и методы. Во-первых, это методы WriteString и ReadString, которые пишут и читают данные из потока в виде строки. При этом кодировка (в Unicode-ных версиях Delphi) контролируется свойством Encoding. И, равно как и предыдущие классы, TStringStream выставляет наружу хранилище в "родном" формате: DataString.

Простой пример использования TStringStream как переходника между строками и TStream:

```
function EncodeString(const Input: String): String;
var
  InStr, OutStr: TStringStream;
begin
  InStr := TStringStream.Create(Input); // <- использует текущую кодовую страницу ANS
  // Можно было так:
  // InStr := TStringStream.Create(Input, CP_UTF8); // <- использует UTF-8
  // или так:
  // InStr := TStringStream.Create(Input, TEncoding.Unicode); // <- использует UTF-16
```

```
  try
    OutStr := TStringStream.Create(''); // <- аналогичные замечания
    try
      EncodeStream(InStr, OutStr); // работает с потоками TStream - т.е. двоичными дан
      Result := OutStr.DataString;
    finally
      FreeAndNil(OutStr);
    end;
  finally
    FreeAndNil(InStr);
  end;
end;
```

Заметьте, что несмотря на наличие методов чтения строк и загрузки/сохранения данных из/в файлы, TStringStream не пригоден для работы с текстовыми файлами. Он не работает с BOM и не позволяет прочитать одну строку (в смысле line) от разделителя до разделителя (он читает только указанное количество символов). По этой причине для работы с текстовыми файлами используют вспомогательный класс - TStringList. Этому классу будет посвящена следующая статья (где и будут показаны методы работы с текстом), а здесь же я только приведу примеры шифрования/расшифровки текстового файла, использующие оба этих класса:

```
procedure EncodeTextFile(const ASourceFileName, ADestFileName: String);
var
  SourceFile: TStringList;
  SourceData: TStringStream;
  DestFile: TFileStream;
begin
  SourceData := nil;
  try
    // Загрузка данных
    SourceFile := TStringList.Create;
    try
      // "Правильная" загрузка текстового файла с учётом BOM и кодировок
      SourceFile.LoadFromFile(ASourceFileName);

      // Конвертируем строку в TStream
      SourceData := TStringStream.Create(SourceFile.Text, TEncoding.Unicode);
    finally
      FreeAndNil(SourceFile);
    end;

    DestFile := TFileStream.Create(ADestFileName, fmCreate or fmShareExclusive);
    try
      EncodeStream(SourceData, DestFile); // двоичное шифрование
    finally
      FreeAndNil(DestFile);
    end;
  finally
    FreeAndNil(SourceData);
  end;
end;

procedure DecodeToTextFile(const ASourceFileName, ADestFileName: String);
var
  SourceFile: TFileStream;
  DestData: TStringStream;
  DestFile: TStringList;
begin
  SourceFile := TFileStream.Create(ASourceFileName, fmOpenRead or fmShareDenyWrite);
  try
    DestData := TStringStream.Create('', TEncoding.Unicode);
    try
      DecodeStream(SourceFile, DestData); // двоичное дешиврование

      DestFile := TStringList.Create;
      try
        // Конвертируем TStream в строку
        DestFile.Text := DestData.DataString;
```

```
48
49          // "Правильное" сохранение текстового файла с BOM (используем UTF-8)
50          DestFile.SaveToFile(ADestFileName, TEncoding.UTF8);
51        finally
52          FreeAndNil(DestFile);
53        end;
54      finally
55        FreeAndNil(DestData);
56      end;
57    finally
58      FreeAndNil(SourceFile);
59    end;
60  end;
```

Конечно, на практике такой пример не имеет большого смысла, потому что гораздо проще просто работать с текстовым файлом как с двоичным - обработав его через TFileStream. Но более удачного и простого примера мне сейчас в голову не приходит, а код выше прекрасно показывает пример соединения трёх классов для работы.

TStreamAdapter

Понятие "потока данных" есть не только в Delphi, но и практически в любом другом современном языке. Разумеется, другие языки понятия не имеют, как работать с объектами Delphi, и наоборот: Delphi не знает, как устроены классы и объекты в других языках. К счастью, под Windows у нас есть COM и интерфейсы. С ними умеют работать почти все языки, так что это является де-факто стандартом межязыкового взаимодействия. И, конечно же, не могло быть иначе: для такой популярной концепции как "поток данных" существует свой интерфейс - IStream.

Иными словами, если вам нужно передать куда-то поток данных - вы используете IStream. Если вам кто-то передаёт поток данных, то это будет IStream.

Тут возникает маленькая проблемка: ваши Delphi объекты вообще-то не умеют работать с интерфейсом IStream: они работают с классом TStream. Что же делать?

Для этого в Delphi есть два класса-адаптера, которые конвертируют TStream в IStream и наоборот. При этом они являются тонкими оболочками, которые просто перенаправляют вызовы. Они конвертируют интерфейс, копирования данных потока не происходит: просто вызовы, скажем, класса конвертируются в вызовы интерфейса (и наоборот), работая с данными оригинального потока данных напрямую.

TStreamAdapter предоставляет переходник от TStream к IStream. Он принимает в конструкторе экземпляр TStream и выставляет наружу IStream, который вы можете передать в чужой код.

TStreamAdapter поддерживает те же операции, что и оригинальный поток, который в него завёрнут.

TStreamAdapter может взять на себе ответственность за удаление исходного потока данных, а может оставить её вам. Вот два примера использования TStreamAdapter, иллюстрирующие оба подхода:

```
1  var
2    Stream: TMemoryStream;
3    COMStream: IStream;
4  begin
5    // Готовим поток-источник: это TStream, который приходит от нашего Delphi-кода
6    Stream := TMemoryStream.Create;
7    try
8      Bitmap.SaveToStream(Stream);
9      Stream.Position := 0;
10
11     // Создаём адаптер. Исходный поток мы должны удалять сами
12     COMStream := TStreamAdapter.Create(Stream, soReference);
13
14     // Загрузка растра в Windows Imaging Component
15     FImagingFactory.CreateDecoderFromStream(COMStream, guid_null, WICDecodeMetadataCa
```

```
    ...
  finally
    COMStream := nil;
    FreeAndNil(Stream);
  end;
end;
```

```
var
  Stream: TFileStream;
  COMStream: IStream;
begin
  // Готовим поток-источник: это TStream, который приходит от нашего Delphi-кода
  Stream := TFileStream.Create('My.bmp', fmOpenRead or fmShareDenyWrite);

  // Создаём адаптер. Адаптер удаляет поток, мы не должны его удалять
  COMStream := TStreamAdapter.Create(Stream, soOwned);

  // Загрузка растра в Windows Imaging Component
  FImagingFactory.CreateDecoderFromStream(COMStream, guid_null, WICDecodeMetadataCach
  ...
end;
```

TOleStream

TOleStream представляет собой обратный класс к TStreamAdapter: переходник от IStream к TStream.

Используется он аналогично. Единственная разница - нет вопроса владения, поскольку интерфейсы относятся к автоуправляемым типам.

TOleStream поддерживает те же операции, что и оригинальный поток, который в него завёрнут.

Пример использования TOleStream:

```
// Открывает файл из композитного OLE-хранилища
function StreamFileRead(const APath: String): TStream;
var
  Strm: IStream;
  StorageToOpen: IStorage;
  FileToOpen: WideString;
begin
  // Пропущены проверки и подготовка
  ...

  // Получаем файл в виде IStream
  if Failed(StorageToOpen.OpenStream(PWideChar(FileToOpen), nil, STGM_READ or STGM_SH
  begin
    Result := nil;
    Exit;
  end;

  // Конвертируем IStream в TStream. Оригинальный поток (Strm) удалится сам когда над
  Result := TOleStream.Create(Strm);
  Result.Position := 0;
end;
```

Прочие потоки

Это далеко не все стандартные потоки, реализованные в Delphi. К примеру, есть ещё TWinSocketStream, TBLOBStream, TZCompressionStream и многие-многие другие. Многие из них являются переходниками, но много классов также предоставляют свою собственную функциональность.

Создание своих классов-потоков

Как бы много в Delphi ни было классов-наследников, всегда найдётся случай, когда вас не устраивают стандартные классы. В этом случае вам нужно реализовывать свой класс-наследник.

К примеру, помните пример для TMemoryStream? Там мы копировали данные строки в поток. Давайте сейчас напишем класс, который позволял бы работать с блоком памяти напрямую, без копирования. Разумеется, такой класс не может поддерживать операцию изменения размера, но чтение, запись и позиционирование - вполне.

Это несложно, если использовать в качестве базового класса TCustomMemoryStream - он уже умеет читать данные из блока памяти, поддерживает позиционирование, но не умеет писать и никак не управляет нижележащим хранилищем. Тогда мы получаем:

```
1  type
2    // Наш новый класс - наследуем способности TCustomMemoryStream
3    TPointerStream = class(TCustomMemoryStream)
4    public
5      // Нам нужно инициализировать поток данных блоком памяти
6      constructor Create(P: Pointer; Size: Integer);
7      // И научить его в него писать (а читать уже умеет TCustomMemoryStream)
8      function Write(const Buffer; Count: Longint): Longint; override;
9    end;
10
11 { TPointerStream }
12
13 constructor TPointerStream.Create(P: Pointer; Size: Integer);
14 begin
15   inherited Create;
16   SetPointer(P, Size);
17 end;
18
19 function TPointerStream.Write(const Buffer; Count: Integer): Longint;
20 var
21   Pos, EndPos: Int64;
22   Mem: Pointer;
23   Sz: Int64;
24 begin
25   Result := 0;
26
27   // Есть что писать?
28   Pos := Position;
29   if (Pos < 0) or (Count <= 0) then
30     Exit;
31
32   // Данные потока
33   Mem := Memory;
34   Sz := Size;
35
36   // Вычисляем конечную позицию и проверяем, что она в пределах блока памяти
37   EndPos := Pos + Count;
38   if EndPos > Sz then
39     raise EStreamError.Create('Ошибка записи данных в поток');
40
41   // Перенос данных в буфер (в текущую позицию)
42   System.Move(Buffer, Pointer(NativeUInt(Mem) + Pos)^, Count);
43
44   // Обновляем текущую позицию потока
45   Position := Pos;
46   Result := Count;
47 end;
48
49 ...
50
51 procedure LoadFromText(const AHandle: THandle; const AText: AnsiString);
52 var
53   Data: TPointerStream;
54 begin
55   // Нет копирования данных, оперирует сразу со строкой
```

```
56    Data := TPointerStream.Create(PAnsiChar(AText), Length(AText));
57    try
58      LoadFromStream(AHandle, Data);
59    finally
60      FreeAndNil(Data);
61    end;
62  end;
```

Вот список методов, которые вы можете захотеть переопределить в своих классах-наследниках:

- (protected) `function GetSize: Int64;` и `procedure SetSize(const NewSize: Int64);` - для управление размером хранилища. Если вы не зададите свой `GetSize`, то реализация по умолчанию будет использовать `Seek` для поиска конца потока и определения размера (равному текущей позиции в конце потока). `SetSize` по умолчанию просто ничего не делает.
- (public) `function Read(var Buffer; Count: Longint): Longint;` и `function Write(const Buffer; Count: Longint): Longint;` для чтения и записи данных. По умолчанию оба метода абстрактные. В любом наследнике вы должны их замещать.
- (public) `function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;` для перемещения по потоку. Реализация по умолчанию вызывает исключение. Вы должны заместить этот метод, если ваши данные поддерживают позиционирование. Обычно это так. Исключение составляют случаи вроде сетевых сокетов.

Вот и всё. Всего три категории и всего пять методов. Реализуйте их - и у вас будет готовый поток. Все прочие методы и свойства являются переходниками к вышеуказанным. К примеру, метод `GetPosition` (get-акцессор для свойства `Position`) реализован как `Result := Seek(0, soCurrent);`

В общем, как видите, создать свой класс-поток - это очень просто.

Преимущества и недостатки потоков данных

Плюсы:

- Де-факто стандарт языка Delphi
- Являются основой для других (более высокоуровневых) механизмов
- Имеют готовые оболочки для самых типичных случаев ("не надо писать самому" - в отличие от файлов в стиле Pascal)
- Гибкость
- Стандартная обработка ошибок на исключениях
- Поддержка произвольных файлов (нет ограничения на размер)
- Нет проблем с многопоточностью*
- Межъязыковая совместимость (через `IStream`)
- Поддержка Unicode и кодировок при работе с текстовыми данными (но не текстовыми файлами - нет поддержки BOM)
- Легко расширяются написанием классов-наследников

Минусы:

- Необходимость ручной сериализации данных
- Ориентированы на побайтовую обработку, слабо пригодны для работы с текстовыми файлами
- Часто неопытные программисты используют `Read` и `Write` вместо `ReadBuffer` и `WriteBuffer`, не делая проверку результатов. Часто это приводит к некорректному коду без обработки ошибок
- Круче кривая обучения: сам `TStream` не умеет делать ничего. Значит, чтобы делать в программах что-то полезное, нужно изучать многочисленные наследники `TStream`, чтобы знать кто что умеет и когда что кого нужно применять. К примеру, если вам нужно отправить растр по сети, то вы должны сообразить, что вы можете создать `THandleStream` для описателя сетевого сокета и использовать его в сочетании с методом `SaveToStream` объекта растра. Сравните это с файлами в стиле Pascal, где было всего три файловых типа, покрывавших все случаи использования

Вывод: если вы работаете в Delphi и хотите работать с любыми данными, то потоки данных должны быть вашим первым вариантом. Используйте что-то другое, только если это "другое" больше подходит для вашей задачи (к примеру, динамический массив для типизированных данных; также некоторые люди могут рассматривать файлы в стиле Pascal более подходящими для работы с текстовыми данными).

(*) - некоторые люди неверно читают это предложение. Обратите внимание, что **речь идёт о достоинствах и недостатках по сравнению с другими способами работы с файлами**. Например, файлами Паскаля. Поэтому, в этом предложении утверждается, что у вас не будет проблем если вы работаете с двумя разными потоками данных (двумя объектами) в двух разных потоках кода, а вовсе не (как читают это некоторые) то, что вы можете обращаться к одному объекту из разных потоков.

Тэги Delphi, Статья

20 комментариев:

nik0lay 14 ноября 2011 г., 12:33

День добрый.
Интересная статья, да и сама тема серии.
Именно так и делаю. Для себя принял строгое правило: все данные которые выходят за границу программы или dll, обязательно дополнять версией! То есть сначала пишу версию, а потом все что нужно. В качестве версии замечательно подходит TGUID, и размер стандартный и неповторяемость почти 100%. Что еще хорошо, если какой либо интерфейс пишет данные, то он может подписаться своим GUIDом. При загрузке данных, сначала читаем GUID, если знаем такой, то читаем все остальное, не знаем, ругаемся на не поддерживаемый формат данных.
Всем удачи!

Ответить

AlekVolsk 1 декабря 2011 г., 19:58

Офигеть, на Delphi пишу с момента ее выхода, и по прочтении статьи понял, что Delphi я не знаю...

Ответить

Fr0sT 27 декабря 2011 г., 14:46

Если вы в своей процедуре принимаете или отправляете какие-то данные - используйте TStream. Не используйте для этого нетипизированные параметры, указатели или конкретные экземпляры TStream. Т.е. вместо:
1 procedure A(AData: Pointer; ADataSize: Cardinal);
2 procedure B(const AData; ADataSize: Cardinal);
3 procedure C(AData: TFileStream);

должно быть:
1 procedure A(AData: TStream);
2 procedure B(AData: TStream);
3 procedure C(AData: TStream);

3 - соглашусь (хотя такое и может понадобиться для ограничения типа), а вот 1, 2 - нет. Оперируя с буфером, буфер и передаёшь, необходимость мутить дополнительные объекты потоков - излишня и раздражает. Поэтому точно так же, как есть LoadFrom/WriteToFile, надо делать перегруженные методы для непосредственной работы с блоком данных.

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.
Integer и Cardinal не меняют своих размеров на x64, только указатели и Native(U)Int. Но если уж нужна фиксация по размеру, то логичнее юзать UInt32.

Нет проблем с многопоточностью
С чего бы это? Отсутствие проблем может быть только для handlestream, и то только потому, что ОС заботится о синхронизации доступа. А всё остальное, тот же memorystream, абсолютно так же подвержен проблемам, как и любой буфер в куче.

Ответить

Александр Алексеев 27 декабря 2011 г., 16:33

>>> 3 - соглашусь (хотя такое и может понадобиться для ограничения типа), а вот 1, 2 - нет. Оперируя с буфером, буфер и передаёшь, необходимость мутить дополнительные объекты потоков - излишня и раздражает.

Мне кажется, ты неявно за меня немножко домыслил то, что я не говорил. Обрати внимание, что речь идёт про отправку и приём данных.

Разумеется, могут быть случаи, когда в контексте задачи имеет смысл только буфер памяти - в этом случае 1 и 2 более чем уместны и даже, более того, п3 не пригоден.

Я же говорю как раз о ситуациях, когда данные могут быть где угодно, но человек всё равно пишет Pointer - и всё тут. В результате вместо того, чтобы просто передать поток, вызывающему придётся данные загрузить самому.

>>> Integer и Cardinal не меняют своих размеров на x64, только указатели и Native(U)Int

То, что генерик тип не меняется на одной конкретной платформе ещё не означает, что он не меняется вообще.

>>> Но если уж нужна фиксация по размеру, то логичнее юзать UInt32.

В общем-то да, но тогда придётся объяснять где взять этот тип на Delphi 7.

>>> С чего бы это?

Сравни это с переменной FileMode. Речь про это. Если угодно: "нет никаких **специальных** проблем".

Ответить

Fr0sT 27 декабря 2011 г., 18:23

Мне кажется, ты неявно за меня немножко домыслил то, что я не говорил. Обрати внимание, что речь идёт про отправку и приём данных.
Возможно, да, а возможно, и нет. Да, приём и отправка данных - это как раз то, с чем я работаю. И мне как-то ни разу не хотелось юзать поток для хранения данных. Потому как для сетевого обмена, например, особенно если это - сплошной поток (какая ирония!), stream не подходит, а скорее нужна очередь. Да, функционала очереди можно добиться от стрима парой новых методов, но не суть. В принципе, я вообще не очень люблю, когда над указателями и блоками памяти навешивают свои типы. Взять, к примеру, TBytes, с которым плотно работает TEncoding. Понятно, что это слизано с .Нет, где другого средства нет. А вот в нативке, если у тебя уже есть буфер с данными, куда засунуть этот TBytes? Придется так или иначе копировать, а это затраты времени и памяти. Так же, к примеру, и с Format - иногда бывает, что нужно инжектить результат в имеющийся символьный буфер, ан фигу. Слава Небесам, что есть FormatBuf - а ведь его могло бы и не быть!
То, что генерик тип не меняется на одной конкретной платформе ещё не означает, что он не меняется вообще.
А где они меняются? Ну, кроме 16-битной платформы (и то не уверен, а искать неохота).
В общем-то да, но тогда придётся объяснять где взять этот тип на Delphi 7.
Ну что ж, кто хочет совместимости, тот должен быть готовым к куче дефайнов! Конечно, в качестве примера с

этим можно и не заморачиваться, но вообще хорошо бы упомянуть, что (U)Int## как раз и предназначены для простого и наглядного объявления переменной фиксированного размера.
Сравни это с переменной FileMode. Речь про это.
Хм, что-то не понял, при чем тут FileMode

Ответить

Александр Алексеев 27 декабря 2011 г., 20:48

>>> А где они меняются?

Этот вопрос суть раскрывает. На красный свет тоже можно переходить. Только это не делает это правильным.

>>> Хм, что-то не понял, при чем тут FileMode

Ну как же. Попробуй открыть файл с нужным доступом. У тебя не получится сделать это потокобезопасно, потому что FileMode - глобальная переменная.

А с потоками данных никакой проблемы нет. Поток данных, созданный во вторичном потоке, никак не связан (и не влияет) с потоком данных в главном потоке - если, конечно, ты сам их не свяжешь (например, натравив оба потока данных на один источник). Но это как бы уже ты сам себе злобный буратино, а сам по себе поток данных проблем не привносит. В отличие от FileMode.

Ответить

Fr0sT 28 декабря 2011 г., 10:16

Попробуй открыть файл с нужным доступом. У тебя не получится сделать это потокобезопасно, потому что FileMode - глобальная переменная
Ааа, ты имеешь в виду, для Паскалевских файлов? Я просто ни разу это не юзал. Тогда не спорю, это в самом деле не потокобезопасно.

Ответить

Александр Алексеев 28 декабря 2011 г., 10:17

Ну, как бы плюсы и минусы я писал в сравнении с другими методами работы с файлами. Такая логика.

Ответить

Анонимный 16 марта 2012 г., 14:59

Спасибо за такой полезный материал!

Ответить

Umpire 24 апреля 2012 г., 15:08

Нет проблем с многопоточностью К сожалению есть проблемы, по крайней мере в TmemoryStream. К свойствам Position и Size нельзя обращаться из разных потоков, потому как их геттеры не просто чтитают какие-то внутренние переменные, а это функции которые используют Seek, что в совю очередь приводит к неконтролируемому изменению Position.

Ответить

Александр Алексеев 24 апреля 2012 г., 15:21

Добавил примечание для тех, кому лень читать комментарии.

Ответить

Анонимный 24 апреля 2012 г., 15:36

:), Не знаю как у других у меня слово "многопоточность" ассоциируется только с Thread'ами, а не со стримами. Так что примечание это хорошо, но лучшеб как-то перефразировать именно пункт. Судя по коментариям не я один такой непонятливый.

Ответить

Александр Алексеев 24 апреля 2012 г., 15:45

Нет, слово "многопоточность" употреблено именно в смысле потоков кода.

Ещё раз.

Эта статья - часть серии. В предыдущей статье были рассмотрены файлы Паскаля. Для них характерна следующая вещь, которая была записана в минуса: для файлов Паскаля нельзя гарантировано открыть два разных файла в двух разных потоках в заданном режиме. Потому что FileMode - глобальная переменная.

Иными словами, с файлами Паскаля нельзя безопасно работать из нескольких потоков.

Так вот такой проблемы с потоками данных нет. Тут нет глобальных переменных, а каждый объект изолирован друг от друга. Поэтому нет никаких проблем открыть сколько угодно файлов в сколько угодно потоках кода - и это всегда будет работать корректно.

А про один объект из нескольких потоков речи вообще не было. И в целом этот момент должен быть понятен: для обращения к одному объекту из нескольких потоков существуют методы Lock и Unlock, которых у TStream нет.

Лично мне это представляется очевидным:

```
Stream.Position := 5;
Stream.Write(I, SizeOf(I)); // кто сказал, что этот вызов будет писать в позицию 5?
```

О какой безопасном доступе можно говорить без Lock/Unlock?

Ответить

Анонимный 24 апреля 2012 г., 18:47

Да я не спорю. Если пошла такая пьянка. Я искал потокобезапасный ТкакойнибудьStream, неохота было велосипед изобретать. Забил в яндекс многопоточность+Tstream. 3 ссылка на ваш блог, ну не сильно разбираясь ищу поиском постранице что тут про многопоточность говорят - **НЕТ ПРОБЛЕМ** :) ну думаю раз gunsmoker говорит что нет проблем, думаю нихрена себе TStream уже безопасный, ну думаю круто и парится не надо все сделано уже до нас. Написал кусок кода, стал запускать и поползли призраки, думаю блин пахнет кидаловом. Стал разбираться полез в исходники Стримов, и смотрю что там не пахнет безопасностью. Ну думаю, надо уведомить общественность об этом открытии :). Ведь все люди ленивы, я статью целеком не читал, не говоря о серии статей или коментариях, ведь проблема у меня не что выбрать стрим или не стрим. Итого я благодарен Вам за разъяснения и статьи и блог и вообще. Но для программиста средней горемычности которого пугают **ПРОБЛЕМАМИ МНОГОПОТОЧНОСТИ** во всех статьях "о многопоточности" этот пункт дезинфа :)

Ответить

Victor Fedorenkov 9 июня 2012 г., 9:16

В TStreamAdaptore есть досадная ошибка непозваляющая работать с большими потоками данных. Вот исправление:

```
//Исправленный StreamAdapter
TFixStreamAdapter = class(TStreamAdapter)
public
function Seek(dlibMove: Int64; dwOrigin: Integer;
```

```
out libNewPosition: Int64): HRESULT; override; stdcall;
end;

...


{ TFixStreamAdapter }

//Ошибка была в этом методе
function TFixStreamAdapter.Seek(dlibMove: Int64; dwOrigin: Integer;
out libNewPosition: Int64): HRESULT;
var
NewPos: LargeInt;
begin
try
if (dwOrigin < STREAM_SEEK_SET) or (dwOrigin > STREAM_SEEK_END) then
begin
Result := STG_E_INVALIDFUNCTION;
Exit;
end;

//Для того что бы вызвался Int64 вариант метода Seek,
//приводим dwOrigin к TSeekOrigin
NewPos := Stream.Seek(dlibMove, TSeekOrigin(dwOrigin));
if @libNewPosition <> nil then libNewPosition := NewPos;
Result := S_OK;
except
Result := STG_E_INVALIDPOINTER;
end;
end;
```

Ответить

Victor Fedorenkov 9 июня 2012 г., 13:30

Кстати,у TOleStream та же беда, Seek реализован только для Int, пришлось делать наследника для поддержки seek'ов для Int64

Ответить

Artyom Stetsenko 16 октября 2012 г., 11:23

Александр, при работе с TFileStream у Вас не указано про случай когда в вновь созданный файл небходимо писать из другог процесса/объекта, например как тут http://www.programmersforum.ru/showthread.php?t=214427

Ведь fmCreate = $FFFF то есть любое логическое сложение с ним даст только fmCreate, который имеет эксклюзивный доступ.

Ответить

Александр Алексеев 16 октября 2012 г., 15:23

Значит у вас старая версия Delphi. В новых Delphi этот баг исправлен. fmCreate определена как $FF00.

Ответить

Анонимный 9 ноября 2014 г., 21:33

Не подскажете, как лучше обработать ситуацию если файл не удалось создать, ведь объект все-таки будет

создан. Его нужно удалить в блоке обработки exception?

Ответить

Александр Алексеев 12 ноября 2014 г., 17:59

Если файл не может быть открыт или создан, то объект не будет создан.

Открытие файла производится в конструкторе TFileStream. При ошибке конструктор выбрасывает исключение. Выброс исключения в конструкторе означает отмену создания объекта.

Таким образом, никаких дополнительных действий делать не нужно.

Ответить

Введите комментарий...

Подпись комментария: Аккаунт Goog

Публикация **Просмотр**

Можно использовать некоторые HTML-теги, например:

Жирный
Курсив
Ссылка

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)

Поиск по блогу

Google™ Пользовательский поиск

Поиск

Архив

Архив

Тэги

Delphi (185) Статья (72) задачки (38) ты можешь это

Подписка

сделать (30) начинающим (26) обработка ошибок (26) случайные мысли (26) EurekaLog (22) блог (17) прочее (16) Windows (14) не делай так (11) роботы/киберпанк (9) журнал (6) 7 (4) Tiburon (4) Vista (4) Королевство Delphi (4) TasksEx (3) x64 (2) Коты (2) кроссплатформенность (1) работа (1)

Шаблон "Simple". Технологии Blogger.