Gilding Pale Streams

This month we look at extending the TStream class

by Julian Bucknall

We're bang in the middle of moving here at TurboPower. Lots of moving boxes are sitting around, the pictures on the wall aren't there any more, and people are getting rid of years of accumulated rubbish. We've got some snazzy new offices in downtown Colorado Springs (until now we've been up on the North end of town in a high tech corridor with WorldCom, HP, Agilent and now Intel). It'll be good to be downtown for a change: new lunchtime eateries, everything within walking distance, a different kind of working crowd. By the time you read this we should have been there, up and running, for several weeks.

And then to crown it all, Wordware, my publishers, sent me the proof of my book *Tomes of Delphi: Algorithms and Data Structures*. They wanted it proofed and indexed in one week. Unfortunately the proof wasn't very good, and so I didn't get away with just indexing; I had to read the whole thing and compare against my original documents. Anyway, after a hectic week or so, I managed to get everything done (at least as much as I could) and it has now gone off to the printers. It should be available by the time you read this.

So, unfortunately, I have less time than usual to write this article. Since I've been hitting the heavy stuff recently in these columns, I thought I'd make this one a little easier going, both for me to write and for you to read.

Introduction

A couple of months back, I said that, in my view, the `TStream` hierarchy was one of the most important class hierarchies provided by Borland in the Delphi VCL. I've provided a few examples of this over the years I've been writing these columns (and, checking back with my *Collection 2000* CD, so has Brian Long), but this month I want to summarize and expand on this theme. So, without further ado, we'll sit on the grassy bank *al fresco* and look into streams.

The ancestor of the entire stream hierarchy is of course the `TStream` class. To most people, the `TStream` class comprises the basic interface: virtual abstract methods to read a buffer (`Read`), to write a buffer (`Write`), and to seek to a particular position in the stream (`Seek`). There's also a `Position` property that, when read, provides you with the current position of the stream and, when written, simply performs a `Seek`. There's a `Size` property as well that returns the current size of the stream when reading, and (from Delphi 3 onwards) truncates the stream on being written to.

There is also a lot more going on in the `TStream` class than you might think. For starters, it contains the entire streaming capabilities of Delphi's DFM files (and Kylix's XFM files): reading and writing components, reading and writing properties, and so on. My intent in this article is not to explore the component-writing capabilities of `TStream`, though (for more information on this, Brian Long wrote an article on the subject back in 1996); it is to expand on other stream types we can devise and write.

The three basic methods used with a stream (`Read`, `Write`, and `Seek`) are all virtual and abstract, meaning that `TStream` is itself an abstract class: you cannot create instances of it. These methods should be overridden in the descendants. The other two static methods we'll note are `ReadBuffer` and `WriteBuffer`. These methods are completely implemented in the ancestor. The `ReadBuffer` method differs from the `Read` method in that it will raise an exception if it cannot read the requested number of bytes (the `Read` method will merely return the number of bytes it did manage to read from the stream). `ReadBuffer` is a simple implementation: a call to `Read` followed by raising an exception if the number of bytes read did not equal the number requested. The `Write` and `WriteBuffer` methods have the same relationship.

The two main stream properties, `Position` and `Size`, are also completely implemented in the `TStream` ancestor. The implementations of the access methods for the `Position` property (the getter and setter, if you like) are trivial in the extreme. They are coded as simple calls to the `Seek` method: the getter merely calls it with an offset of zero from the current position and returns the result, the setter calls it with an offset of the position wanted from the start of the stream.

The getter for the `Size` property is coded as three calls to `Seek`: the first seeks to the current position and saves the return value, the second seeks to an offset of zero from the end of the stream, and the third seeks back to the original position. The setter for the `Size` property (remember: Delphi 3 and above and Kylix only) is a virtual method that does nothing in the ancestor, and that presumably is overridden in descendants where setting size is a meaningful operation (it doesn't make sense in a read-only stream for example). Notice though the reliance on the `Seek` method. We'll be discussing how to get around this for streams that cannot seek.

The `TStream` class, then, is a true ancestor class. You must create and instantiate descendants of `TStream` in order to provide streaming capabilities to your application. Delphi comes with several descendants: a memory stream, a string stream, a file stream, a resource stream, a BLOB stream, and so on. It must be said that these form a fairly complete hierarchy and it seems hard to come up with others. Nevertheless, there's a whole class of stream descendants we can write, as you'll see.

Anyway, one I needed pretty well straight away was a file stream that remembered its own name. The standard file stream doesn't store its name, so if an error occurs and you want to use the filename as part of an exception message, you're out of luck. This descendant is extremely trivial to code (a replaced `Create` constructor and a new property to get at the filename) and is shown in Listing 1.

Writing Filters

We could, with enough time and inclination, come up with other trivial descendants like this but, in reality, it hardly seems worth it. On the other hand there's a whole class of descendants we could write that would be meaningful. These descendants are known as *filters*.

Let's take a simple example. Suppose you wanted to write a stream that compresses data as you write to the stream (the stream being viewed as a write-only stream). Seems easy enough, provided you have a handy compression algorithm already implemented, but from which stream would you descend? A file stream, so that writing to the stream writes compressed data to disk? Sure, but next week you might need to compress data being written to a BLOB. Would you then have to copy the same code but just make a descendant of `TBLOBStream` instead? It certainly seems wasteful of effort.

Better is to write a filter. A filter is a stream descendant that uses an internal stream to which it delegates all of its reads, writes and seeks. It does not, in and of itself, know how to read, write or seek; it defers all that to an already open stream that you provide when calling the constructor. In this way, your code does not actually see or directly interact with the internal stream, instead your requests are filtered through the filter stream. The filter stream may indeed alter the data you see. The internal stream may be a memory stream or a file stream or any other type of stream; it makes no difference.

Let's see how this works with a simple filter: a read-only filter. A read-only filter only allows you to read data from the stream. You cannot write to it or even seek within it: the stream should be viewed as a sequence of bytes, a flow of data; once read you can't go back and reread it.

This `TStream` descendant sounds simple enough. Simply override

➤ *Listing 1: A file stream that knows its name.*

```
type
  TaaFileStream = class(TFileStream)
    {a file stream that remembers its name}
    private
      FName : string;
    protected
    public
      constructor Create(const aFileName : string; aMode : word);
      property Name : string read FName;
  end;
constructor TaaFileStream.Create(const aFileName : string; aMode : word);
begin
  FName := aFileName;
  inherited Create(aFileName, aMode);
end;
```

➤ *Listing 2: The TaaReadFilter class.*

```
type
  TaaReadFilter = class(TStream)
    {a read-only filter}
    private
      FSize       : longint;
      FStream     : TStream;
      FGotSizeReq : boolean;
    protected
    public
      constructor Create(aStream : TStream;
        aSize : longint);
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function Write(const Buffer; Count : longint) :
        longint; override;
  end;
constructor TaaReadFilter.Create(aStream : TStream;
  aSize : longint);
begin
  Assert(aStream <> nil,
    'TaaReadFilter.Create: the stream cannot be nil');
  inherited Create;
  FStream := aStream;
  if (aSize = -1) then
    FSize := FStream.Size
  else
    FSize := aSize;
end;
function TaaReadFilter.Read(var Buffer; Count : longint) :
  longint;
begin
  Assert(not FGotSizeReq,
    'TaaReadFilter.Read: cannot read whilst getting size');
  Result := FStream.Read(Buffer, Count);
end;
```

```
function TaaReadFilter.Seek(Offset : longint; Origin : word)
  : longint;
begin
  case Origin of
    soFromBeginning :
      if FGotSizeReq then begin
        Result := FStream.Position;
        if (Offset = Result) then
          Exit;
        FGotSizeReq := false;
      end;
    soFromCurrent :
      if (Offset = 0) and (not FGotSizeReq) then begin
        Result := FStream.Position;
        Exit;
      end;
    soFromEnd :
      if (Offset = 0) and (not FGotSizeReq) then begin
        Result := FSize;
        FGotSizeReq := true;
        Exit;
      end;
  else
    Assert(false, Format(
      'TaaReadFilter.Seek: invalid origin (%d)', [Origin]));
  end;
  Assert(false,
    'TaaReadFilter.Seek: a read-only filter cannot seek');
  Result := 0;
end;
function TaaReadFilter.Write(const Buffer; Count : longint)
  : longint;
begin
  Assert(false,
    'TaaReadFilter.Write: a read-only filter cannot write');
  Result := 0;
end;
```

the Read method to pass read requests to the underlying stream, override the Write and Seek methods to raise exceptions. There is one slight problem though: the Seek method could be used in a 'read-only' sense. Where am I in the stream? How big is the stream? Both of these operations involve calls to Seek. We are not altering the position of the stream in these cases, we are merely reading some important values. Recall also how the Size property get method works: read the current position, seek to the end to get the stream size, seek to the original position. We should therefore override Seek to cater for these special cases: reading the current position (Seek is called with offset 0 and origin soFromCurrent), seeking to the end of the stream to get the stream size (Seek is called with offset 0 and origin soFromEnd), seeking to the current position (Seek is called with the offset equal to the current position and the origin soFromBeginning).

Unfortunately, we will have to store some state for reading the Size property: the last two calls to Seek are designed to change the position of the stream, albeit temporarily. We will have to store the fact that the seek to the end of the stream took place, and then if the very next call to a stream method is not the one we're expecting (a call to reposition the stream where we were), we'd raise an exception.

Listing 2 shows the resulting TaaReadFilter class. As you can see, the Create constructor takes in another, already opened, stream. It is this stream that will be our delegate. The Read method merely calls the Read method of the delegate stream, unless the stream's state is in mid GetSize. The Write method will raise an exception. The Seek method is where the fun stuff we've been discussing occurs. Notice that if we've been asked to seek to the end of the stream, we return the previously determined size of the stream (the Create constructor accepts the size of the stream: zero or more is the actual size, -1 means that we can read the size of the stream from the stream itself, and MaxLongint means that the stream size is unknown).

Having written a read-only filter, it is a matter of moments to design a write-only stream. With such a stream we can maintain the current size of the stream quite easily: just count up the number of bytes written to the stream. We still have to play the same games with the Seek method, the Read method is merely raises an exception, whereas Write gets the underlying stream to do the work. Listing 3 shows the implementation of the TaaWriteFilter class.

Buffering Filters

Well, so far nothing really exciting. Let's now create a descendant of the TaaReadFilter class. This descendant will buffer the data in the underlying stream for us. Why do this? Well, there is one stream class that calls out for it: the TFileStream. Every call to Read in a TFileStream is translated to call the operating system API directly. This is extremely inefficient. Better would be to read the underlying stream in large buffers full and then dole it out, as and when required. This in turn should be much more efficient: we are only accessing the OS API once in a while for large buffers (say 8Kb worth), and the rest of the time we're just moving data around in memory.

Listing 4 shows the TaaReadBufferFilter class. We obviously need to override the Read method to dole out data from our buffer, filling it wherever required from the underlying stream. The constructor therefore has to create the buffer, the destructor to free it. We also must override the Seek method for one particular case: getting the position of the filter.

➤ *Listing 3: The TaaWriteFilter class.*

```
type
  TaaWriteFilter = class(TStream)
    {a write-only filter}
    private
      FSize       : longint;
      FStream     : TStream;
      FGotSizeReq : boolean;
    protected
    public
      constructor Create(aStream : TStream);
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function Write(const Buffer; Count : longint) :
        longint; override;
  end;
constructor TaaWriteFilter.Create(aStream : TStream);
begin
  Assert(aStream <> nil,
    'TaaWriteFilter.Create: the stream cannot be nil');
  inherited Create;
  FStream := aStream;
end;
function TaaWriteFilter.Read(var Buffer; Count : longint) :
  longint;
begin
  Assert(false,
    'TaaWriteFilter.Read: a write-only filter cannot read');
  Result := 0;
end;
function TaaWriteFilter.Seek(Offset : longint; Origin :
  word) : longint;
begin
  case Origin of
    soFromBeginning :
      if FGotSizeReq then begin
        Result := FSize;
        if (Offset = Result) then
          Exit;
        FGotSizeReq := false;
      end;
    soFromCurrent :
      if (Offset = 0) and (not FGotSizeReq) then begin
        Result := FSize;
        Exit;
      end;
    soFromEnd :
      if (Offset = 0) and (not FGotSizeReq) then begin
        Result := FSize;
        FGotSizeReq := true;
        Exit;
      end;
  else
    Assert(false, Format('TaaWriteFilter.Seek: invalid '+
      'origin (%d)', [Origin]));
  end;
  Assert(false,
    'TaaWriteFilter.Seek: a read-only filter cannot seek');
  Result := 0;
end;
function TaaWriteFilter.Write(const Buffer; Count : longint)
  : longint;
begin
  Assert(not FGotSizeReq, 'TaaWriteFilter.Write: cannot
    write whilst getting size');
  Result := FStream.Write(Buffer, Count);
  inc(FSize, Result);
end;
```

Recall that we are reading the data out of the stream in large buffers-worth. The stream's position is therefore very inaccurate: it will always report a multiple of our buffer's size and nothing else. We therefore need to trap the request for the current position (offset is zero, origin is `soFromCurrent`) to report where we are, as if the buffer were not there.

Designing a write-only buffered stream is a little more involved. The `Write` method will accept data into the buffer and, whenever required, write it out to the underlying stream. Also, just like in the read-only buffered stream case, the constructor should allocate a buffer and the destructor should destroy it, but now the destructor has some extra work to do. Consider the problem: we half fill up the buffer and then free the stream. This data in the buffer only has one chance to get written out to the delegate stream: during the `Destroy` method. We should therefore make an attempt to write it out before we start freeing the buffer.

There's another problem, though. The `Write` method of a `TStream` descendant is supposed to signal that it couldn't write all the data by returning the number of bytes it did manage to write. In general, our buffered stream will always manage to write data: it just gets put in the buffer. So consider the moment that you write data to the buffered stream and it fills up the buffer and has to write it out to the underlying stream. This latter stream returns that it could only write less bytes than requested. What should we do? We certainly can't return the number of bytes that were written to the underlying stream. Instead we return the number of bytes that we managed to add to the buffer to fill it up. We also slide along the data in the buffer so that it just contains data that wasn't written. If the underlying stream were a file stream, and we've just filled up the disk, our buffer is full, and all subsequent calls to `Write` will return zero as the number of bytes written. The attempt to write the last buffer of data in the `Destroy` is also guaranteed to fail in this case, but we can do nothing here except ignore it (we should not raise exceptions in destructors: too many things go wrong if you do that; for example, the object being destroyed is not freed if an exception is raised in the destructor).

The `TaaWriteBufferFilter` class is shown in Listing 5. Again we need to override the `Seek` method to calculate the corrected current position of the filter since the stream will again only report multiples of the buffer size. Having seen a couple of examples of filter classes, we should show a brief bit of code that explains how to use them. Listing 6 demonstrates the use of the buffered filters to copy one file to another. Notice especially that you should not destroy the stream being filtered until you have destroyed the filter. Doing it the other way round could lead to access violations or seg faults.

Text Filters

You may view some of these filters as being, well, dinky, and yearn for something a little meatier. How

➤ *Listing 4: The read-only buffered filter.*

```
type
  TaaReadBufferFilter = class(TaaReadFilter)
    {a read-only buffered filter}
    private
      FBuffer : PChar;
      FBufEnd : longint;
      FBufPos : longint;
    protected
      function rbfReadBuffer : boolean;
    public
      constructor Create(aStream : TStream; aSize:longint);
      destructor Destroy; override;
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
  end;
constructor TaaReadBufferFilter.Create(aStream : TStream;
  aSize : longint);
begin
  inherited Create(aStream, aSize);
  GetMem(FBuffer, BufferSize);
end;
destructor TaaReadBufferFilter.Destroy;
begin
  if (FBuffer <> nil) then
    FreeMem(FBuffer, BufferSize);
  inherited Destroy;
end;
function TaaReadBufferFilter.rbfReadBuffer : boolean;
begin
  {read the next bufferful from the stream}
  FBufEnd := FStream.Read(FBuffer^, BufferSize);
  FBufPos := 0;
  {return true if at least one byte read, false otherwise}
  Result := FBufEnd <> FBufPos;
end;
function TaaReadBufferFilter.Read(var Buffer; Count :
  longint) : longint;
var
  UserBuf     : PChar;
  BytesToGo   : longint;
  BytesToRead : longint;
begin
  {reference the buffer as a PChar}
  UserBuf := @Buffer;
  {start the counter for the number of bytes read}
  Result := 0;
  {if needed, fill internal buffer from underlying stream}
  if (FBufPos = FBufEnd) then
    if not rbfReadBuffer then
      Exit;
  {calculate number of bytes to copy from internal buffer}
  BytesToGo := Count;
  BytesToRead := FBufEnd - FBufPos;
  if (BytesToRead > BytesToGo) then
    BytesToRead := BytesToGo;
  {copy bytes from internal buffer to user buffer}
  Move(FBuffer[FBufPos], UserBuf^, BytesToRead);
  {adjust the counters}
  inc(FBufPos, BytesToRead);
  dec(BytesToGo, BytesToRead);
  inc(Result, BytesToRead);
  {while there are more bytes to copy, do so}
  while (BytesToGo <> 0) do begin
    {advance the user buffer}
    inc(UserBuf, BytesToRead);
    {fill the internal buffer from the underlying stream}
    if not rbfReadBuffer then
      Exit;
    {calculate number of bytes to copy from internal buffer}
    BytesToRead := FBufEnd - FBufPos;
    if (BytesToRead > BytesToGo) then
      BytesToRead := BytesToGo;
    {copy bytes from internal buffer to user buffer}
    Move(FBuffer^, UserBuf^, BytesToRead);
    {adjust the counters}
    inc(FBufPos, BytesToRead);
    dec(BytesToGo, BytesToRead);
    inc(Result, BytesToRead);
  end;
end;
function TaaReadBufferFilter.Seek(Offset : longint;
  Origin : word) : longint;
begin
  if (Offset = 0) and (Origin = soFromCurrent) then
    Result := FStream.Position - FBufEnd + FBufPos
  else
    Result := inherited Seek(Offset, Origin);
end;
```

```
type
  TaaWriteBufferFilter = class(TaaWriteFilter)
    {a write-only buffered filter}
    private
      FBuffer : PChar;
      FCurPos : PChar;
    protected
      function wbfWriteBuffer : boolean;
    public
      constructor Create(aStream : TStream);
      destructor Destroy; override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function Write(const Buffer; Count : longint) :
        longint; override;
  end;
constructor TaaWriteBufferFilter.Create(aStream : TStream);
begin
  inherited Create(aStream);
  GetMem(FBuffer, BufferSize);
  FCurPos := FBuffer;
end;
destructor TaaWriteBufferFilter.Destroy;
begin
  if (FBuffer <> nil) then begin
    if (FCurPos <> FBuffer) then
      wbfWriteBuffer;
    FreeMem(FBuffer, BufferSize);
  end;
  inherited Destroy;
end;
function TaaWriteBufferFilter.wbfWriteBuffer : boolean;
var
  BytesToWrite : longint;
  BytesWritten : longint;
begin
  BytesToWrite := FCurPos - FBuffer;
  BytesWritten := FStream.Write(FBuffer^, BytesToWrite);
  if (BytesWritten = BytesToWrite) then begin
    Result := true;
    FCurPos := FBuffer;
  end else begin
    Result := false;
    if (BytesWritten <> 0) then begin
      Move(FBuffer[BytesWritten], FBuffer^,
        BytesToWrite - BytesWritten);
      FCurPos := FBuffer + (BytesToWrite - BytesWritten);
    end;
  end;
end;
function TaaWriteBufferFilter.Seek(Offset : longint;
  Origin : word) : longint;
begin
  if (Offset = 0) and (Origin = soFromCurrent) then
    Result := FStream.Position + (FCurPos - FBuffer)
  else
    Result := inherited Seek(Offset, Origin);
end;
function TaaWriteBufferFilter.Write(const Buffer;
  Count : longint) : longint;
var
  UserBuf      : PChar;
  BytesToGo    : longint;
  BytesToWrite : longint;
begin
  {reference the buffer as a PChar}
  UserBuf := @Buffer;
  {start the counter for the number of bytes written}
  Result := 0;
  {if needed, empty internal buffer into underlying stream}
  if (BufferSize = FCurPos - FBuffer) then
    if not wbfWriteBuffer then
      Exit;
  {calculate number of bytes to copy to internal buffer}
  BytesToGo := Count;
  BytesToWrite := BufferSize - (FCurPos - FBuffer);
  if (BytesToWrite > BytesToGo) then
    BytesToWrite := BytesToGo;
  {copy the bytes from user buffer to internal buffer}
  Move(UserBuf^, FCurPos^, BytesToWrite);
  {adjust the counters}
  inc(FCurPos, BytesToWrite);
  dec(BytesToGo, BytesToWrite);
  inc(Result, BytesToWrite);
  {while there are more bytes to copy, do so}
  while (BytesToGo <> 0) do begin
    {advance the user buffer}
    inc(UserBuf, BytesToWrite);
    {empty the internal buffer into the underlying stream}
    if not wbfWriteBuffer then
      Exit;
    {calculate number of bytes to copy to internal buffer}
    BytesToWrite := BufferSize;
    if (BytesToWrite > BytesToGo) then
      BytesToWrite := BytesToGo;
    {copy bytes from user buffer to internal buffer}
    Move(UserBuf^, FCurPos^, BytesToWrite);
    {adjust the counters}
    inc(FCurPos, BytesToWrite);
    dec(BytesToGo, BytesToWrite);
    inc(Result, BytesToWrite);
  end;
end;
```

➤ *Listing 5: The write-only buffered filter.*

```
FSIn := nil;
FSOut := nil;
BFIn := nil;
BFOut := nil;
try
  FSIn := TFileStream.Create('AAStrms.pas', fmOpenRead);
  BFIn := TaaReadBufferFilter.Create(FSIn, -1);
  FSOut := TFileStream.Create('test1.tst', fmCreate);
  BFOut := TaaWriteBufferFilter.Create(FSOut);
  BytesRead := BFIn.Read(B, sizeof(B));
  while (BytesRead <> 0) do begin
    BFOut.Write(B, BytesRead);
    BytesRead := BFIn.Read(B, sizeof(B));
  end;
finally
  BFOut.Free;
  FSOut.Free;
  BFIn.Free;
  FSIn.Free;
end;
```

➤ *Listing 6: Simple use of the buffered filters.*

about a filter that parses up the underlying stream as text into individual lines? Again we'd have to write two descendant classes: one for reading and one for writing. Because we have no idea how many characters there are per line we'd have to do some buffering, otherwise reading a line would be too inefficient. We shall use the buffered classes as ancestors, that way we get the buffering we'd require for free.

Let's tackle the read-only text filter to start with. With this filter it doesn't seem to make sense to enable the `Read` method, and instead we should write and provide a `ReadLine` method so that the user can get an entire line in one go. After all it's the ability to read a text file in terms of its lines that we're trying to achieve. However, I can certainly see the need for the ability to read an arbitrary number of bytes from a text stream, so we'll leave it implemented just in case.

Of course, all the real work will be done in the `ReadLine` method. This method will return a string containing the current line. Ah, but what terminates the current line? A carriage return, line feed character pair (CR/LF)? A single LF character? In the Windows universe, it's the former delimiter, in the Linux cosmos, the latter. What we'll do is to cater for both. `ReadLine` will scan characters from the current position until it reaches an LF character. It'll then check the previous character to see if it's a CR character. Either way it can calculate the length of the line just read, create the return string, and then position the stream just after the CR/LF or LF.

There's one more problem, though. How do we know that we've reached the end of the

```
type
  TaaReadTextFilter = class(TaaReadBufferFilter)
    {a read-only text filter}
    private
      FStrBuilder : TaaStringBuilder;
    protected
    public
      constructor Create(aStream : TStream; aSize :
        longint);
      destructor Destroy; override;
      function ReadLine : string; virtual;
      function AtEndOFStream : boolean;
  end;
constructor TaaReadTextFilter.Create(aStream : TStream;
  aSize : longint);
begin
  inherited Create(aStream, aSize);
  FStrBuilder := TaaStringBuilder.Create;
end;
destructor TaaReadTextFilter.Destroy;
begin
  FStrBuilder.Free;
  inherited Destroy;
end;
function TaaReadTextFilter.AtEndOFStream : boolean;
begin
  Result := FSize = Position;
end;
function TaaReadTextFilter.ReadLine : string;
const
  CR = ^M;
  LF = ^J;
var
  Ch : char;
  BytesRead : longint;
begin
  {read characters until we get an LF}
  BytesRead := Read(Ch, sizeof(Ch));
  while (BytesRead <> 0) and (Ch <> LF) do begin
    {if it's not a CR character, append it to the current
      line}
    if (Ch <> CR) then
      FStrBuilder.Add(Ch);
    BytesRead := Read(Ch, sizeof(Ch));
  end;
  {return the string}
  Result := FStrBuilder.AsString;
end;
```

➤ *Listing 7: The read-only text filter.*

stream? It reeks of the sledgehammer if we raise an exception in the ReadLine method when that point is reached. We could continually test to see if the value of the Position property equals that of the Size property, but, as I'm sure you're now aware, that would require four calls to Seek for every test. My preference is to have a special function, called AtEndOfStream, say, that returns true once the end of the stream is reached. Inside this function, we'll test the current position against the pre-calculated size of the stream.

Listing 7 shows the TaaReadTextFilter class. Remember the important thing about this class: it can be used with any other stream whatsoever. Obviously the first use most people would have is to read a text file, but it can equally as well be used for a memory stream containing text, or a BLOB, and so on.

Pretty good and very useful. The next most obvious move is to code a write-only text filter. We will implement the Write method in the usual way, and we'll have to write a WriteLine method to write a string of characters, followed by the end-of-line terminator. What's this going to be? In the read-only text stream we were able to auto-detect the end-of-line marker and deal with it, but this time we can't. The class needs to know which type of end-of-line marker to use: a CR/LF or a single LF. An ideal job for a property, which is what we'll do. We'll be clever and have the default calculated in the constructor dependent on the platform on which we're compiling.

Apart from that, the WriteLine method is very simple. We call the overridden Write method to write the passed string, and then we write either a CR/LF character pair or a single LF character. Listing 8 shows the write-only text filter, TaaWriteTextFilter.

Debug Filter

Having written these text streams, I can identify another useful filter: a debug filter. This is a class that logs each and every call to the Read, Write and Seek methods to a file for later perusal. This log can be interesting to read and may point out the need for buffering, for example, or show up problems in the use of a stream. The code is trivial to implement: the Create constructor opens up a log file that you specify by name, using a write-only text filter; the Destroy destructor closes it; the Read method logs the number of bytes requested, and the number of bytes actually read; the Write method does a similar job as Read, and Seek method logs the offset and origin requested and the returned position. Listing 9 shows this class (note that it would be an easy exercise to log the actual data read or written as well).

Regex Filter

What next? A couple of months ago I implemented a regular expression engine for matching patterns to strings. This seems a natural fit for the read-only text filter: a

➤ *Listing 8: The write-only text filter.*

```
type
  TaaLineDelimiter = ( {possible line delimiters}
     ldLF,            {..line feed}
     ldCRLF);         {..carriage return line feed}
  TaaWriteTextFilter = class(TaaWriteFilter)
    {a write-only text filter}
    private
      FLineDelim : TaaLineDelimiter;
    protected
    public
      constructor Create(aStream : TStream);
      procedure WriteLine(const S : string);
      property LineDelimiter : TaaLineDelimiter
        read FLineDelim write FLineDelim;
  end;
constructor TaaWriteTextFilter.Create(aStream : TStream);
begin
  inherited Create(aStream);
  {$IFDEF Win32}
  FLineDelim := ldCRLF;
  {$ENDIF}
  {$IFDEF Linux}
  FLineDelim := ldLF;
  {$ENDIF}
end;
procedure TaaWriteTextFilter.WriteLine(const S : string);
const
  cLF : char = ^J;
  cCRLF : array [0..1] of char = ^M^J;
begin
  if (length(S) > 0) then
    Write(S[1], length(S));
  case FLineDelim of
    ldLF   : Write(cLF, sizeof(cLF));
    ldCRLF : Write(cCRLF, sizeof(cCRLF));
  end;
end;
```

```
type
  TaaDebugFilter = class(TStream)
    {a debug filter}
    private
      FLog    : TaaWriteTextFilter;
      FFile   : TFileStream;
      FStream : TStream;
    protected
      function dfGetOriginStr(aOrigin : word) : string;
    public
      constructor Create(aStream : TStream; const aLogName :
        string);
      destructor Destroy; override;
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function Write(const Buffer; Count : longint) :
        longint; override;
  end;
constructor TaaDebugFilter.Create(aStream : TStream; const
  aLogName : string);
begin
  Assert(aStream <> nil,
    'TaaDebugFilter.Create: the stream cannot be nil');
  inherited Create;
  FStream := aStream;
  FFile := TFileStream.Create(aLogName, fmCreate);
  FLog := TaaWriteTextFilter.Create(FFile);
end;
destructor TaaDebugFilter.Destroy;
begin
  FLog.Free;
  FFile.Free;
  inherited Destroy;
end;
function TaaDebugFilter.dfGetOriginStr(aOrigin : word) :
  string;
begin
  case aOrigin of
    soFromBeginning : Result := 'soFromBeginning';
    soFromCurrent   : Result := 'soFromCurrent';
    soFromEnd       : Result := 'soFromEnd';
  else
    Result := Format('Invalid origin [%d]', [aOrigin]);
  end;
end;
function TaaDebugFilter.Read(var Buffer; Count : longint) :
  longint;
begin
  FLog.WriteLine(Format('READ:  Count requested: %d',
    [Count]));
  Result := FStream.Read(Buffer, Count);
  FLog.WriteLine(Format('Bytes read: %d', [Result]));
end;
function TaaDebugFilter.Seek(Offset : longint; Origin :
  word) : longint;
var
  OriginStr : string;
begin
  OriginStr := dfGetOriginStr(Origin);
  FLog.WriteLine(Format('SEEK:  Offset: %d, Origin: %s',
    [Offset, OriginStr]));
  Result := FStream.Seek(Offset, Origin);
  FLog.WriteLine(Format('Returned position: %d', [Result]));
end;
function TaaDebugFilter.Write(const Buffer; Count : longint)
  : longint;
begin
  FLog.WriteLine(Format('WRITE: Count requested: %d',
    [Count]));
  Result := FStream.Write(Buffer, Count);
  FLog.WriteLine(Format('Bytes written: %d', [Result]));
end;
```

➤ *Listing 9: The debug filter.*

descendant that will only return lines that fit the specified regular expression pattern.

This descendant also gives me the opportunity to add a fix to my latest regular expression engine code. When I introduced the optimization I discussed in the April *Algorithms Alfresco* I managed to add a bug whereby if the regular expression started with an alternation (that is: this subexpression OR that one) it would fail during the matching process. The version included on this month's disk fixes the bug.

Back to the regex filter class. We shall need a regex property, of course. This would need compiling into the transition table form prior to use. Apart from that, the `ReadLine` method would be overridden so that it calls the inherited `ReadLine` method to read the individual lines in the stream. Only lines that match the regex will be returned, so, in effect, the `ReadLine` method would call the inherited method until the line returned matched the regex pattern. Apart from that, the regex class implementation is fairly trivial. Listing 10 shows the class.

```
type
  TaaRegexTextFilter = class(TaaReadTextFilter)
    {a read-only regex text filter}
    private
      FRegexEngine : TaaRegexCompiler;
    protected
    public
      constructor Create(aStream : TStream; aSize : longint; const aRegex :
        string);
      destructor Destroy; override;
      function ReadLine : string; override;
  end;
constructor TaaRegexTextFilter.Create(aStream : TStream; aSize : longint;
  const aRegex : string);
begin
  inherited Create(aStream, aSize);
  FRegexEngine := TaaRegexCompiler.Create(aRegex);
end;
destructor TaaRegexTextFilter.Destroy;
begin
  FRegexEngine.Free;
  inherited Destroy;
end;
function TaaRegexTextFilter.ReadLine : string;
var
  S : string;
begin
  S := inherited ReadLine;
  while (FRegexEngine.MatchString(S) = 0) do begin
    if AtEndOfStream then begin
      Result := '';
      Exit;
    end;
    S := inherited ReadLine;
  end;
  Result := S;
end;
```

➤ *Listing 10: The regex filter.*

Encryption Filter

Let's move away from text filters for now and show another possibility for stream filters. Back in June 2000, I implemented DES encryption in *Algorithms Alfresco*. Let's reuse that code to create a write-only filter that encodes data when it's written, and then decodes it through a read-only stream. We shall need to specify a key to the `Create` constructor of both classes so that we can encode and decode the data when required. Apart from that it seems fairly easy, except that we must remember that DES encodes data in blocks of 64 bits (8 bytes). Therefore, our `Write` method needs to collect data in a buffer and then encrypt it as it writes out the data to the underlying stream. The buffer can be as small as 8 bytes (a DES block) or as large as we need. It makes sense to go for the latter: that way we can encrypt and write data in larger blocks. For the read-only filter we can read and decrypt in large buffers rather than 8 bytes at a time.

There is one small problem, though. If you refer back to the original article, you can see that the final block (which may be smaller than 8 bytes) goes through a complicated bit of business with the penultimate block during the encoding. We may not be able to fiddle with this penultimate block, since it may have already been written to the underlying stream. So we need to be careful with how we write the buffer out to the stream.

The best solution is to always keep a block in hand. When we encrypt and write a buffer's worth, we in fact process the entire buffer except for the last block. So if the buffer is 8Kb in size, we encrypt and write out 8,088 bytes and keep the remaining 8 bytes in hand (a single DES block) to start off the next buffer's worth. At the end, when the filter is destroyed we know we always have the penultimate block ready for all the bit twiddling and obfuscation we need to do.

On reading we need to do the same kind of processing; that is, keeping a DES block in hand ready for the final bit un-twiddling.

Window Filter

Another fairly handy filter that I've wished for in the past is a filter that only shows you part of another stream. I call this a window filter.

It always seems that, when I write a routine that processes data in a stream, the first thing I code in the routine is a `Seek` to position `0`, to reset the stream at the beginning. What if the stream were already positioned correctly at a particular place? Well, I've messed it up. OK, let's remove the call to `Seek` at the start of the routine. But, here I will get the opposite problem: I always have to remember to position the stream at the correct place before calling the routine. It seems I lose if I do, and I lose if I don't. Essentially I cannot assume the routine is a black box: I have to remember whether the black box does or does not reposition.

A better solution is to have a filter that only exposes part of the underlying stream, from a particular offset onwards. That way I can position the underlying stream to my heart's content, create a window filter over it, and pass the filter to the routine. If the routine resets the filter it won't go back before my desired position. If it doesn't reset the filter, I am still covered.

To create a `TaaWindowFilter` object we pass in the 'zero' position of the underlying stream, the offset that the filter is to call zero. The `Create` constructor will, if necessary, reposition the underlying stream to this offset if the stream is positioned before the zero position. After that, the only method we need to really work at is the `Seek` method: we shouldn't allow seeks before the zero position (even though there is data there) and we should take care of seeks from the end of the stream (since the filter would be smaller in size than the underlying stream). The `Seek` method should also position the underlying stream correctly (by adding in the zero position value to every seek, and subtracting it from the return value). Listing 11 shows this simple filter.

Summary

After this exploration into the world of streams and filters, I hope you see the possibilities in using filters to alter the properties of any stream. We don't have to worry about creating similar-looking descendants of file streams and memory streams, for example, we just create a single descendant of `TStream` and wrap it around any

➤ *Listing 11: The window filter.*

```
type
  TaaWindowFilter = class(TStream)
    {a window filter}
    private
      FStream  : TStream;
      FZeroPos : longint;
    protected
    public
      constructor Create(aStream : TStream; aZeroPos :
        longint);
      destructor Destroy; override;
      function Read(var Buffer; Count : longint) : longint;
        override;
      function Seek(Offset : longint; Origin : word) :
        longint; override;
      function Write(const Buffer; Count : longint) :
        longint; override;
  end;
constructor TaaWindowFilter.Create(aStream : TStream;
  aZeroPos : longint);
begin
  Assert(aStream <> nil, 'TaaWindowFilter.Create: the stream
    cannot be nil');
  Assert(aZeroPos >= 0, 'TaaWindowFilter.Create: the zero
    position cannot be negative');
  inherited Create;
  FStream := aStream;
  FZeroPos := aZeroPos;
  if (FStream.Position < aZeroPos) then
    FStream.Seek(aZeroPos, soFromBeginning);
end;
destructor TaaWindowFilter.Destroy;
begin
  inherited Destroy;
end;
function TaaWindowFilter.Read(var Buffer; Count : longint) :
  longint;
begin
  Result := FStream.Read(Buffer, Count);
end;
function TaaWindowFilter.Seek(Offset : longint; Origin :
  word) : longint;
var
  NewPos : longint;
begin
  case Origin of
    soFromBeginning :
      NewPos := FStream.Seek(Offset + FZeroPos,
        soFromBeginning);
    soFromCurrent :
      NewPos := FStream.Seek(Offset, soFromCurrent);
    soFromEnd :
      NewPos := FStream.Seek(Offset, soFromEnd);
  else
    Assert(false,
      'TaaWindowFilter.Seek: invalid Origin value');
    NewPos := 0;
  end;
  if (NewPos < FZeroPos) then
    NewPos := FStream.Seek(FZeroPos, soFromBeginning);
  Result := NewPos - FZeroPos;
end;
function TaaWindowFilter.Write(const Buffer; Count :
  longint) : longint;
begin
  Result := FStream.Write(Buffer, Count);
end;
```

stream. This wrapper class filters the data in the underlying stream in various interesting and useful ways: I'm sure you'll find others.

Julian Bucknall is a spare time Shakespeare scholar. Email him at julianb@turbopower.com