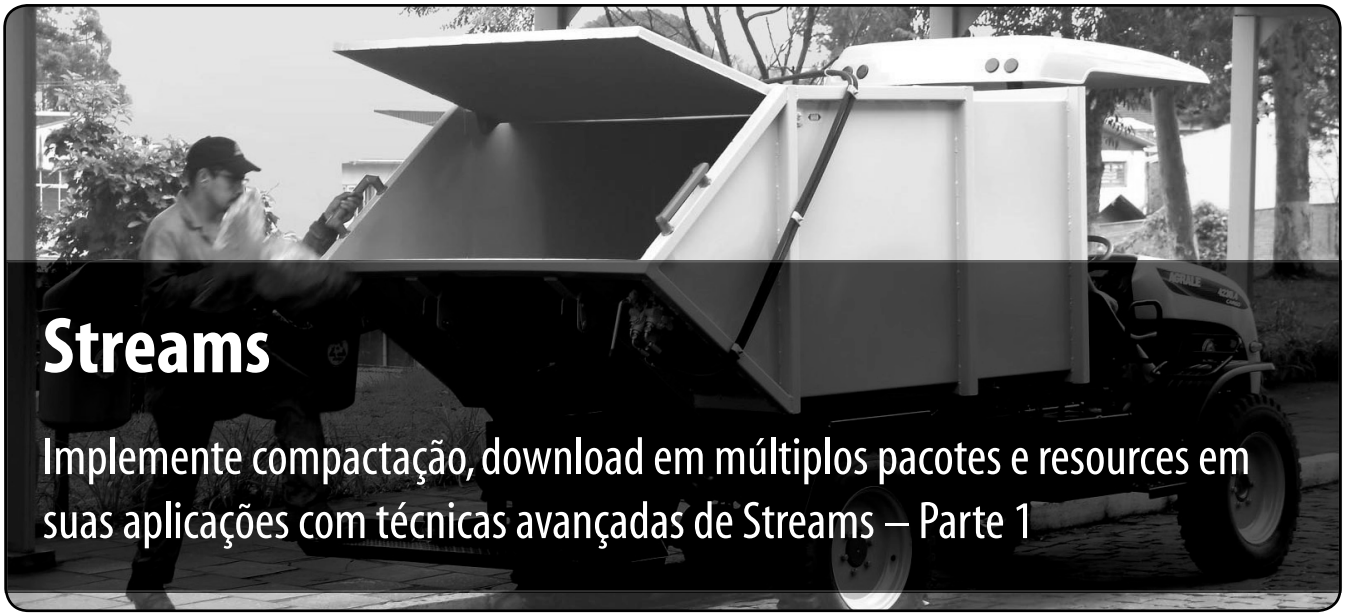
Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi .NET



Streams

Implemente compactação, download em múltiplos pacotes e resources em suas aplicações com técnicas avançadas de Streams – Parte 1

Gustavo Chaurais
(gustavoc@macrovision.com)
é Borland Delphi 7 Advanced Certified, Borland Delphi 2005 for Win32 Certified, Borland Delphi 2006 for Win32 Certified e Borland Delphi Instructor Certified. Foi palestrante das três últimas edições da Borland Conference Brasil e de outros grandes eventos nacionais. Além disso, é membro da coordenadoria do GIESSC. Hoje, ministra cursos, presta consultoria e atua como Software Engineer do projeto InstallAnywhere para a empresa norte-americana Macrovision Corporation.

Aí está o grande pesadelo de grande parte dos programadores Delphi. Muitos, quando ouvem falar em *streams*, desistem do código na hora! Mas, por que será que causa tanto medo? Talvez, isso se deva ao fato de estarmos trabalhando diretamente com *bits* e *bytes*. E isso nos remete aos "malucos escovadores de bits" de décadas atrás. Hoje, temos orientação a objetos, a adoção do *garbage collector* pelas mais diversas linguagens, dentre outros conceitos e técnicas que deixam nossa vida muito mais fácil. Por isso, desacostumamos com o fato de que, por trás de tudo, ainda temos os mesmos ponteiros, *bits* e *bytes*.

Neste artigo, veremos que *streams* são, na verdade, muito simples e podem ser utilizados por qualquer programador. Todos os que já trabalharam com *arrays*, listas (*TStringList*, *TObjectList*), *datasets*, ou qualquer tipo de coleção, não devem encontrar dificuldade para utilizar todos os recursos da classe *TStream*.

Com exemplos práticos e objetivos, abordaremos também dois assuntos extremamente relacionados: *Compression* e *Resources*. Veremos que não é necessário fazer o download nenhuma biblioteca para trabalhar com arquivos compactados, uma vez que o Delphi já implementa a sua. Construiremos, passo a passo, uma aplicação para compactação de múltiplos arquivos.

Quanto aos *Resources*, além do exemplo clássico de embutir uma DLL dentro de seu executável, aprenderemos também, a fundo, a alterar os recursos de outras aplicações, bem como a modificar suas *resourcestrings*. Enfim, tudo o que há de mais avançado no assunto.

Streams

Imagine um *dataset* bidirecional, como o *TClientDataSet* ou o *TTable*. Porém, em vez de armazenar registros e campos de um banco de dados, armazenaremos *bytes*, um após o outro, de um arquivo

que está no disco. Podemos pegar seu tamanho final, temos a possibilidade de navegar entre os *bytes*, etc., como se fossem registros.

Streams podem ser entendidos como uma seqüência de *bytes* na memória. Esses *bytes* podem ser tanto de um programa que foi carregado, como de uma *string* qualquer, um arquivo de recurso, ou como uma seqüência de *bytes* qualquer que desejamos representar. Analisemos código presente na **Listagem 1**.

A classe *TFileStream* herda de *TStream* e serve para representarmos arquivos do disco na memória. Seu uso é muito simples. No seu construtor, devemos passar: o caminho para o arquivo a ser representado e o modo de abertura desse arquivo. (ver nota "Modos de Abertura de Arquivos")

Vale lembrar que o arquivo ficará bloqueado para outros *streams* (e outros programas) até que o *stream* seja fechado. Como não existe um método *Close*, a única maneira de fazermos isso é chamando seu destrutor (ou *Free*). Portanto, é muito importante fechar o *stream* para que o arquivo seja liberado.

No exemplo citado, estamos criando dois *streams* do tipo *TFileStream* e chamando o método *CopyFrom* do que está criando o arquivo. Portanto, estamos simplesmente criando uma cópia do arquivo. Além disso, notamos que esse método necessita da quantidade de *bytes* que gostaríamos de copiar. No caso, utilizamos o arquivo inteiro (*Size*). Vejamos agora outro tipo de *stream*, também descendente de *TStream*, o *TStringStream*. Vamos modificar a entrada de dados para uma *string* na memória. (**Listagem 2**)

A classe *TStringStream* se difere da anterior apenas pela fonte de dados, que agora é uma *string*. Esta classe possui uma propriedade especial *DataString*, pela qual é possível sabermos qual a *string* está sendo armazenada na memória. O resultado deste exemplo será um arquivo texto contendo: *string* no *stream*.

Listagem 1. Trecho de código utilizando Streams

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStreamIn: TFileStream;
  FileStreamOut: TFileStream;
begin
  FileStreamIn := TFileStream.Create('caminho para arquivo de entrada',
    fmOpenRead);
  FileStreamOut := TFileStream.Create('caminho para arquivo de saída',
    fmCreate);
  try
    FileStreamOut.CopyFrom(FileStreamIn, FileStreamIn.Size)
  finally
    FileStreamIn.Free;
    FileStreamOut.Free;
  end;
end;
```

Listagem 2. Trecho de código utilizando Streams

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StringStreamIn: TStringStream;
  FileStreamOut: TFileStream;
begin
  StringStreamIn:= TStringStream.Create('string no stream');
  FileStreamOut:= TFileStream.Create('caminho para arquivo de saída',
    fmCreate);

  try
    FileStreamOut.CopyFrom(StringStreamIn, StringStreamIn.Size);
  finally
    StringStreamIn.Free;
    FileStreamOut.Free;
  end;
end;
```

Modos de Abertura de Arquivos

fmCreate
Caso o arquivo não exista, será criado. Caso exista, será zerado para que se comece um novo.

fmOpenRead
Apenas para leitura.

fmOpenWrite
Apenas para escrita.

fmOpenReadWrite
Leitura e escrita.

Cabe aqui apresentar as duas propriedades mais importantes de qualquer *stream*: *Position* e *Size*. Como em um *dataset*, *streams* possuem um cursor interno para a leitura e a escrita. No exemplo anterior, não precisamos lidar com tal cursor porque o método *CopyFrom* fez todo o trabalho. No entanto, se lermos o valor de *Position* antes de chamá-lo, veremos que seu valor é *0*. Após a cópia, este muda para o número de *bytes* copiado (no caso, o tamanho do arquivo). E a propriedade *Size*, como era de se esperar, retorna o tamanho do que foi carregado no *stream*. Ambas são do tipo *Int64*, um tipo especial de inteiro capaz de trabalhar com números de até *64bits*.

Experimente acrescentar a seguinte linha logo após a criação do *TStringList*:

```
StringStreamIn.Position := 7;
```

Ao rodar o programa, você verá uma exceção, pois como estamos tentando copiar todo o tamanho do arquivo (*StringStreamIn.Size*) e começando da posição *7*, ele tentará ler um conteúdo que não existe. Para corrigir isso, modifique a linha de cópia para:

```
FileStreamOut.CopyFrom(StringStreamIn,
  StringStreamIn.Size - StringStreamIn.
  Position);
```

Agora, a cópia será feita a partir da posição *7*, ou seja, o conteúdo final do arquivo será: *no stream* Além do método *CopyFrom* a maneira mais comum de lermos e escrevermos algo em um *stream* é utilizando seus métodos *Read* e *Write*. Analisemos o código da **Listagem 3**.

Existem diversas maneiras de manipular dados com *Read* e *Write*, porém, consideramos esta a mais simples. A primeira coisa a ser feita é definirmos um *buffer*. Um *buffer* precisa ser um local da memória livre para armazenarmos *bytes*. Neste caso, poderíamos utilizar um ponteiro para algum lugar (como um *PChar*) e trabalharmos com seu valor (variavel^), alocando e desalocando memória para ele (*GetMem*), ou um simples *array* (de *Byte*, *Char* ou outra estrutura de 8 bits), que também é uma região reservada da memória. Não poderíamos utilizar um *string* porque, apesar de também representar uma região alocada da me-

mória, possui algumas informações no primeiro caractere que não interessam neste caso. Portanto, decidimos utilizar um *array*.

O tamanho do *buffer* você é quem escolhe. Estamos trabalhando com 1024 *bytes* (0..1023 – 1KB). Esta será a região reservada na memória para transferência de dados entre nossos *streams*.

> **Nota**: Se preferir utilizar um *PChar* como *buffer*, é aconselhável escolher um tamanho até 61440 *bytes* (240Kb).

Com o *buffer* definido, podemos agora chamar o método *Read* do *stream*. Passamos para ele onde devem ser colocados os *bytes* lidos (*buffer*) e a quantidade máxima de *bytes* que gostaríamos de ler. O retorno do método é o número de *bytes* lidos. Então, este é número que passaremos para o método *Write*, pois queremos escrever exatamente o que foi lido e está no *buffer*. A quantidade de *bytes* lidos será *0* somente quando não houver mais nada para ler. Portanto, a cópia será feita com sucesso.

Figura 1. Exemplo de tela de download

Existem ainda alguns métodos não muito utilizados na classe *stream*. O método *seek* é utilizado para a movimentação do cursor a partir de um ponto (análogo ao método *MoveBy* do *TDataSet*). *ReadBuffer* e *WriteBuffer* utilizam *Read* e *Write*, respectivamente, e disparam uma exceção caso a quantidade de *bytes* a ser lida/escrita for diferente da recebida, ou se a quantidade de *bytes* a ser lida/escrita for igual a *0*. O restante dos métodos é utilizado pela IDE para a leitura/escrita de componentes e seus recursos em *streams*.

Das classes descendentes de *TStream* comumente utilizadas, podemos destacar: *TFileStream*, *TStringStream*, *TMemoryStream* e *TResourceStream*. As duas últimas serão demonstradas posteriormente e armazenam, respectivamente, *bytes* genericamente na memória e o conteúdo de um *resource*. Vale acrescentar que a classe *TFileStream* é bastante útil quando estamos lidando com uma quantidade muito grande de *bytes*, pois o armazenamento é feito diretamente no disco rígido, não na memória *RAM*. Operações no disco são, contudo, mais lentas do que na memória.

Uma última dica: muito cuidado ao misturar *TStringStreams* com outros *streams*. As *strings* do Delphi são *null-terminated*, ou seja, depois de todo o texto vem sempre um caractere nulo (*#0*). No caso de programas e arquivos não-texto, esse caractere é muito comum e não indica, de maneira alguma, o final do arquivo. Portanto, se você tentar copiar um *TFileStream* (carregando um arquivo binário) para um *TStringStream*, seu *DataString* apresentará os caracteres até apenas o caractere nulo (*#0*).

Com o que temos até agora já conseguimos fazer muita coisa no que se refere ao tratamento de arquivos (cópias, criação, leitura, modificação, etc...), no entanto, vamos praticar esse conhecimento através de um exemplo mais sólido.

Na internet, existem disponíveis diversos programas que aceleram a velocidade de *downloads*. Seu funcionamento básico compreende estabelecer várias conexões com o servidor e quebrar o *download* em pedaços. Posteriormente, o arquivo final é montado. Este é o exemplo que iremos construir.

Listagem 3. Trecho de código copiando a partir da posição 7

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStreamIn: TFileStream;
  FileStreamOut: TFileStream;
  Buffer: array[0..1023] of Byte;
  BytesRead: Integer;
begin
  FileStreamIn := TFileStream.Create('caminho para arquivo de entrada',
    fmOpenRead);
  FileStreamOut := TFileStream.Create('caminho para arquivo de saída',
    fmCreate);
  try
    repeat
      BytesRead := FileStreamIn.Read(Buffer, 1024);
      FileStreamOut.Write(Buffer, BytesRead);
    until BytesRead = 0;
  finally
    FileStreamIn.Free;
    FileStreamOut.Free;
  end;
end;
```

Listagem 4. Criação da classe de download

```
TFTPThread = class(TThread)
private
  FLimit: Integer;
  FOrder: Integer;
  FTotal: Integer;
  FFileToGet: string;
  FIdFTP: TIdFTP;
  FStream: TStream;
  FFormToWarn: TForm1;
  procedure IdFTPWork(Sender: TObject; AWorkMode:
    TWorkMode; const AWorkCount: Integer);
protected
  procedure Execute; override;
public
  constructor Create(Order, Total: Integer;
    FileToGet: string; FormToWarn: TForm1);
  function GetStream: TStream;
end;
```

Criando um exemplo de download

Crie um novo projeto, salve-o como "FTPThread.dpr" e, no formulário principal, acrescente dois componentes *TEdit* (*edFTP* e *edLocal*) e um *TButton* (*btGet*). Vamos começar criando a classe da *Thread* que fará o *download* do arquivo. Declare abaixo da declaração do formulário uma classe como mostrado na **Listagem 4**. Desenhe uma tela semelhante a **Figura 1**.

Vamos precisar de: um atributo para saber qual é a ordem da *Thread* atual na lista de *Threads* que vão fazer parte

do *download* (*FOrder*); outro para saber a quantidade total de *Threads* (*FTotal*); esses dois serão utilizados para se calcular o limite de *bytes* que essa *Thread* fará o *download* (*FLimit*); o arquivo a ser baixado (*FFileToGet*); a *stream* na qual gravaremos o pedaço do arquivo (*FStream*); o componente que fará o *download* efetivamente (*TIdFTP*) e o formulário a ser avisado quando o *download* for concluído (*FFormToWarn*).

Não se esqueça de adicionar duas *Units* na sua cláusula *Uses*: *IdComponent* e *IdFTP*. Essas serão necessárias, pois faremos uso de um componente *TIdFTP* (que está na aba *Indy Clients*). A implementação do seu construtor será bem simples **(Listagem 5).**

Note que, como estamos passando *False* para o construtor da *Thread*, ela iniciará imediatamente, sem a necessidade de chamarmos seu método *Resume*. Implemente seu principal método conforme a **Listagem 6**.

Nossa primeira tarefa será criar e configurar o componente *TIdFTP*. Modifique as propriedades *Host*, *Username* e *Password* para as configurações de seu servidor FTP. O evento *OnWork* está sendo configurado para podermos finalizar o *download* quando o limite for atingido e será implementado a seguir. Conectamos então com o servidor e pegamos o tamanho do arquivo. O cálculo do limite será feito dividindo-se o tamanho do arquivo pelo número de *Threads*, sendo que a última *Thread* da lista ficará com o resto da divisão.

Internamente, o componente *TIdFTP* irá pegar a posição atual do *stream* de saída para saber de onde deve continuar o *download*. Portanto, criamos o *stream*, configuramos sua posição e iniciamos o *download*. Ao final, voltamos para a posição inicial e configuramos o *Size* do *stream*, pois o componente pode ter feito o *download* de alguns *bytes* a mais, que são desnecessários. Note que, agora estamos utilizando um *TMemoryStream*, pois o arquivo só será montado mais tarde. Você poderia utilizar aqui um *TFileStream* caso desejasse fazer uso de arquivos temporários para cada *Thread*. Por fim, chamamos o método *ThreadDone* do nosso formulário, que será implementado na seqüência. Implemente o tratador do evento *OnWork* do componente *TIdFTP* de acordo com adiante.

```
if AWorkMode = wmRead then
  if AWorkCount >= FLimit then
    FIdFTP.Abort;
```

Basicamente, estamos abortando a operação caso o limite de *bytes* tenha sido alcançado. Este evento é chamado cada vez que alguns *bytes* são carregados do servidor. Também pode ser utilizado no acompanhamento do progresso do *download*. Finalmente, como o formulário irá precisar montar o arquivo final, ele precisará dos *bytes* baixados por esta *Thread*, por isso precisamos do método *GetStream*, descrito a seguir.

```
function TFTPThread.GetStream: TStream;
begin
  Result := FStream;
end;
```

Nossa classe está pronta. Agora vamos à classe do formulário, que irá gerenciar as *Threads* e montar o arquivo final. Primeiramente, adicione antes da declaração da classe do formulário uma nova *class*.

```
  TFTPThread = class;
```

E na seção *private* da classe de seu formulário inclua:

```
  FThread1: TFTPThread;
  FThread2: TFTPThread;
  FThread3: TFTPThread;
  FThread4: TFTPThread;
  FThreadsDone: Integer;
....
```

Crie os objetos como mostrado a seguir:

```
procedure TForm1.btGetClick(Sender: TObject);
begin
  FThread1:=TFTPThread.Create(
    1, 4, edFTP.Text, Self);
 FThread2:=TFTPThread.Create(
    2, 4, edFTP.Text, Self);
 FThread3:=TFTPThread.Create(
    3, 4, edFTP.Text, Self);
 FThread4:=TFTPThread.Create(
    4, 4, edFTP.Text, Self);
end;
```

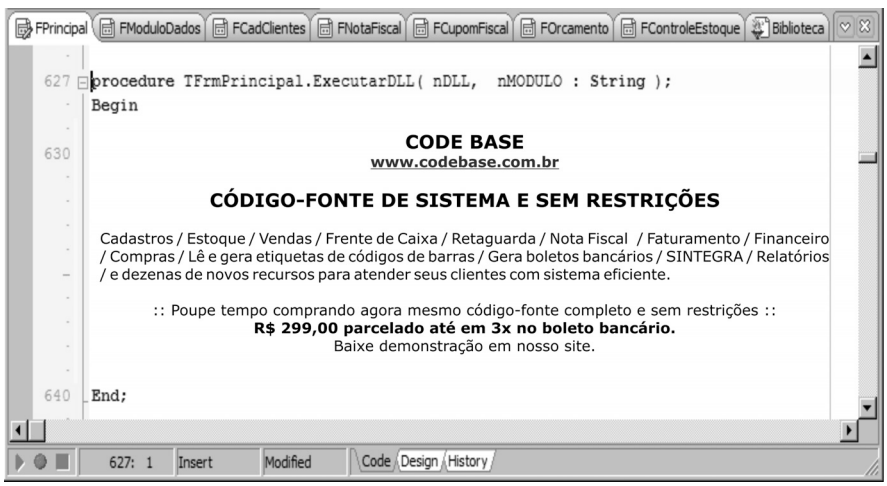
Finalmente, declare e implemente o método *ThreadDone* (**Listagem 7**).

Listagem 5. Código do método construtor

```
constructor TFTPThread.Create(Order,
  Total: Integer; FileToGet:
  string; FormToWarn: TForm1);
begin
  FOrder := Order;
  FTotal := Total;
  FFileToGet := FileToGet;
  FFormToWarn := FormToWarn;
  inherited Create(False);
end;
```

Listagem 6. Método principal da classe

```
procedure TFTPThread.Execute;
var
  ASize, AStart: Integer;
begin
  FIdFTP := TIdFTP.Create(nil);
  try
    FIdFTP.Host := 'localhost';
    FIdFTP.Username := 'Anonymous';
    FIdFTP.Password := '';
    FIdFTP.OnWork := IdFTPWork;
    FIdFTP.Connect;

    ASize := FIdFTP.Size(FFileToGet);

    FLimit := ASize div FTotal;
    AStart := FLimit * (FOrder - 1);
    if FOrder = FTotal then
      Inc(FLimit, ASize mod FTotal);

    FStream := TMemoryStream.Create();
    FStream.Position := AStart;
    FIdFTP.Get(FFileToGet, FStream, True);
    FStream.Size := AStart + FLimit;
    FStream.Position := AStart;

    Synchronize(FFormToWarn.ThreadDone);
  finally
    FIdFTP.Free;
  end;
end;
```



Listagem 7. Código do método ThreadDone

```
procedure TForm1.ThreadDone;
var
  AFileStream: TFileStream;
begin
  Inc(FThreadsDone);

  if FThreadsDone = 4 then
  begin
    AFileStream := TFileStream.Create(edlocal.Text, fmCreate);
    try
      Copy(FThread1.GetStream, AFileStream);
      Copy(FThread2.GetStream, AFileStream);
      Copy(FThread3.GetStream, AFileStream);
      Copy(FThread4.GetStream, AFileStream);
    finally
      AFileStream.Free;
    end;
  end;
end;
```

Listagem 8. Código do método Copy

```
procedure TForm1.Copy(const FromStream, ToStream: TStream);
var
  Buffer: array[0..255] of Byte;
  BytesRead: Integer;
begin
  repeat
    BytesRead := FromStream.Read(Buffer, 256);
    ToStream.Write(Buffer, BytesRead);
  until BytesRead = 0;

  FromStream.Free;
end;
```

Listagem 9. Código exemplo de compressão

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StringStream: TStringStream;
  FileStreamOut: TFileStream;
  CompressionStream: TCompressionStream;
begin
  StringStream := TStringStream.Create('string no stream');
  FileStreamOut := TFileStream.Create('caminho para arquivo de saída', fmCreate);
  CompressionStream := TCompressionStream.Create(clDefault, FileStreamOut);

  try
    CompressionStream.CopyFrom(StringStream, StringStream.Size)
  finally
    StringStream.Free;
    CompressionStream.Free;
    FileStreamOut.Free;
  end;
end;
```

Listagem 10. Outra forma de compressão de arquivos

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileStreamIn: TFileStream;
  StringStream: TStringStream;
  DecompressionStream: TDecompressionStream;
  ReadBytes: Integer;
  Buffer: array [0..1023] of Byte;
begin
  FileStreamIn := TFileStream.Create('caminho para arquivo de entrada', fmOpenRead);
  StringStream := TStringStream.Create('');
  DecompressionStream := TDecompressionStream.Create(FileStreamIn);

  try
    repeat
      ReadBytes := DecompressionStream.Read(Buffer, 1024);
      StringStream.Write(Buffer, ReadBytes);
    until ReadBytes = 0;
  finally
    DecompressionStream.Free;
    StringStream.Free;
    FileStreamIn.Free;
  end;
end;
```

Cada vez que uma *Thread* termina seu serviço, este método é chamado e quando as quatro terminarem, o arquivo é montado. A montagem do arquivo é feita normalmente, conforme os exemplos anteriores. Para isso utilizamos o método *Copy*. Então, declare e implemente-o. O código completo podemos ver na **Listagem 8**.

Neste caso, o método *CopyFrom* do *stream* não funcionaria porque precisamos manter as posições iniciais de cada *stream*, o que não acontece com o tal método.

Fica como sugestão de treino a criação de uma barra de progresso para o acompanhamento e o tratamento de alguns erros, como servidor indisponível, etc. Para a barra de progresso, verifique o quanto já foi carregado em cada *Thread* a cada *x* segundos. Dessa maneira, você pode também calcular a velocidade do seu *download* pegando a quantidade de *bytes* carregados nesse segundo.

Inicie um programa e faça o *download* de algum arquivo. Se tudo ocorrer bem, o arquivo final será montado corretamente e no local certo.

Compression com zLib

Certamente, quem nunca precisou, um dia vai precisar fazer a compactação de algo via programação. Para isso, muitos utilizam bibliotecas disponíveis na internet, acessam utilitários externos, etc. O que muita gente não sabe é que o Delphi já possui nativamente a biblioteca *zLib* para fazermos a compactação de arquivos. E sua utilização, com o conteúdo que temos até agora, é bastante simples. Utilizaremos, simplesmente, as classes *TCompressionStream* e *TDecompressionStream* da *unit zLib*. Analisemos o código da **Listagem 9**.

Conforme podemos perceber, a única mudança em relação ao primeiro exemplo foi a adição da classe *TCompressionStream*. Esta deve ser criada com base em um *stream* de saída (qualquer tipo) e devemos informar o nível de compressão desejado: *clNone*, *clFastest*, *clDefault*, *clMax*. De nenhum à compressão máxima, respectivamente. Ao chamarmos o método *CopyFrom* do *TCompressionStream*, ele irá ler os *bytes* do *stream* de entrada

e, à medida que a gravação é feita, a compressão é também realizada.

Vale lembrar aqui a importância do fechamento do *stream* de compressão (através do método *Free*). Caso contrário, o arquivo não será gravado corretamente. Além disso, tanto o *TCompressionStream* quanto seu *stream* de saída devem ser fechados corretamente. A leitura, nesse caso, seria o contrário. (**Listagem 10**)

Agora, em vez do *TCompressionStream*, estamos utilizando o *TDecompressionStream*. Sua operação, em vez de ser referente ao *stream* de saída, será no de entrada. E, mais uma diferença: a propriedade *Size*, neste caso, nos informa o tamanho do *stream* compactado. Se utilizarmos um *CopyStream* copiando esse número de *bytes*, não estaremos copiando todos os *bytes*, pois, o *stream* descompactado provavelmente será maior do que isso. Portanto, como não sabemos previamente o tamanho final, utilizamos os métodos *Read* e *Write* para conseguir ler todo o *stream*.

Como você deve ter reparado, essas classes não dão nenhum suporte para a adição de múltiplos arquivos em um mesmo *stream*. Temos que fazer isso manualmente. E é esse exemplo que construiremos a seguir. Vamos criar um programa similar ao *WinZip*, mas o padrão de compressão será nosso.

Primeiramente, vamos tentar entender como será o formato desse arquivo compactado. Para cada arquivo a ser adicionado no arquivo compactado, criaremos uma entrada. Ela terá a seguinte forma: [LENGTH DO NOME][NOME DO ARQUIVO][SIZE DO ARQUIVO][CONTEUDO DO ARQUIVO]. Assim, saberemos exatamente o que e quanto ler na hora de descompactar em múltiplos arquivos. O conteúdo do arquivo completo será uma entrada após a outra.

Para o *length* do nome do arquivo e para o *size* do arquivo, utilizaremos um tipo *Integer* e um tipo *Int64*, respectivamente. Para gravá-los, poderíamos facilmente utilizar um *IntToStr* e gravar sua representação *string*. Porém, 2097152 *bytes* (2MB), por exemplo, ocuparia sete casas. Em vez disso, vamos armazenar este número como ele é visto pelo computador (forma binária), o que ocuparia apenas três casas, pois número é representado por 32, 0 e 0. Isso vai nos fazer economizar um bocado de espaço no arquivo final.

Crie um novo projeto, salve-o como "Compression.dpr", inclua uma nova *Unit* e salve-a como "Compressor.pas". Nessa nova *Unit*, adicione *SysUtils*, *Classes*, *ZLib e Math* na sua seção *Uses* e declare o *Record* que irá representar uma entrada no arquivo e o tipo *TBuffer*, como podemos ver adiante.

```
TCompressionEntry = record
  Path: string;
  FileSize: Int64;
end;
TBuffer = array[0..1023] of Byte;
```

O tipo *TBuffer* é necessário para podermos passá-lo como parâmetro, o que não é permitido com *arrays*. Declare também uma classe *TCompressor*. Esta será a classe responsável pela compressão dos arquivos. Adicione a essa classe os seguintes métodos utilitários e implemente-os (**Listagem 11**).

Não se esqueça de adicionar a diretiva *overload* aos dois métodos *IntToBin*. Tais métodos não serão explicados a fundo porque são apenas utilitários e fogem um pouco ao contexto do exemplo, porém, serão extremamente úteis. Por exemplo, o método *CopyToBuffer* faz-se necessário pois, quando utilizamos o tradicional *StrPCopy*, caso haja algum caractere nulo no meio da seqüência, este será interpretado erroneamente como caractere final.

Listagem 11. Métodos da classe TCompressor

```
function TCompressor.CopyToBuffer(const From: string;
  Count: Integer): TBuffer;
var
  i: Integer;
begin
  for i := 1 to Count do
    Result[i - 1] := From[i];
end;
function TCompressor.CopyFromBuffer(const From: TBuffer;
  Count: Integer): string;
var
  i: Integer;
begin
  Result := '';
  for i := 0 to Count - 1 do
    Result := Result + From[i];
end;
function TCompressor.IntToBin(Int: Integer): string;
var
  i: Integer;
begin
  Result := '';
  for i := 3 downto 0 do
  begin
    Result := Result + Chr(Int shr (i * 8));
  end;
end;
function TCompressor.IntToBin(Int: Int64): string;
var
  i: Integer;
begin
  Result := '';
  for i := 7 downto 0 do
  begin
    Result := Result + Chr(Int shr (i * 8));
  end;
end;
function TCompressor.BinToInt(Bin: string): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := 0 to 3 do
  begin
    Result := Result or Ord(Bin[i + 1]) shl ((3 - i) * 8);
  end;
end;
function TCompressor.BinToInt64(Bin: string): Int64;
var
  i: Integer;
begin
  Result := 0;
  for i := 0 to 7 do
  begin
    Result := Result or Ord(Bin[i + 1]) shl ((7 - i) * 8);
  end;
end;
```

Listagem 12. Código do método CloseStreams

```
procedure TCompressor.CloseStreams;
begin
  FreeAndNil(FStreamWrite);
  FreeAndNil(FStreamRead);
  FreeAndNil(FInternalStream);
end;
```

Listagem 13. Construtor da classe TCompressor

```
constructor TCompressor.Create(const FilePath:
  string; CleanFile: Boolean);
begin
  FFilePath := FilePath;
  if CleanFile then
    RestartStream(fmCreate)
  else
    RestartStream(fmOpenRead);
  FStreamRead := TDecompressionStream.Create(
    FInternalStream);
  try
    PopulateEntries;
  finally
    CloseStreams;
  end;
end;
```

Listagem 14. Método PopulateEntries

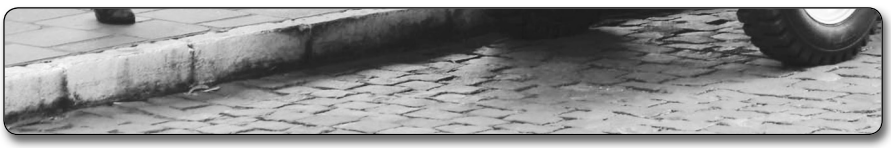
```
procedure TCompressor.PopulateEntries;
var
  Buffer: TBuffer;
  NameLength: Integer;
begin
  SetLength(FEntries, 0);

  while FStreamRead.Read(Buffer, 4) > 0 do
  begin
    SetLength(FEntries, Length(FEntries) + 1);
    NameLength := BinToInt(CopyFromBuffer(Buffer, 4));
    FStreamRead.Read(Buffer, NameLength);
    FEntries[High(FEntries)].Path := CopyFromBuffer(
      Buffer, NameLength);
    FStreamRead.Read(Buffer, 8);
    FEntries[High(FEntries)].FileSize := BinToInt64(
      CopyFromBuffer(Buffer, 8));
    ReadFile(FEntries[High(FEntries)].FileSize, False);
  end;
end;
```

Listagem 15. Método ReadFile

```
procedure TCompressor.ReadFile(Count: Int64;
  WriteResult: Boolean);
var
  Buffer: TBuffer;
  BytesRead: Integer;
begin
  repeat
    BytesRead := FStreamRead.Read(Buffer, Min(Count,
      1024));
    if WriteResult then
      FStreamWrite.Write(Buffer, BytesRead);
    Dec(Count, BytesRead);
  until (Count = 0) or (BytesRead = 0);
end;
```

Listagem 16. Método CopyEntireStream

```
procedure TCompressor.CopyEntireStream;
var
  Buffer: TBuffer;
  BytesRead: Integer;
begin
  repeat
    BytesRead := FStreamRead.Read(Buffer, 1024);
    FStreamWrite.Write(Buffer, BytesRead);
  until BytesRead = 0;
end;
```



Adicione *FInternalStream*, *FStreamWrite* e *FStreamRead*, os três do tipo *stream*, à declaração de sua classe. O primeiro referenciará o *stream* do arquivo final e, os outros dois serão os *streams* de compressão e descompressão, respectivamente. Declare e implemente o método como segue:

```
procedure TCompressor.RestartStream(Mode:
Word);
begin
  FInternalStream := TFileStream.Create(
    FFilePath, Mode);
end;
```

Este será utilizado para iniciar o *stream* principal e poderá ser sobrescrito em uma classe descendente para a gravação em outro tipo de *stream*. Adicione a diretiva *virtual* a ele. Aproveite para declarar e implementar o método *CloseStreams*, onde liberamos de memória todos os *streams* criados. Veja o código completo na **Listagem 12**.

Para as entradas, declare um atributo *FEntries* (*array of TCompressionEntry*) e adicione as propriedades:

```
    property EntriesCount: Integer read
GetEntriesCount;
    property Entries[Index: Integer]:
TCompressionEntry read GetEntry; default;
```

GetEntriesCount deve retornar simplesmente o *Length* de *FEntries* e *GetEntry* retorna *FEntries[Index]*. O construtor da classe deve ser declarado e implementado como na **Listagem 13**.

Declare *FFilePath* (*string*) para salvarmos o nome do arquivo. O parâmetro *CleanFile* irá definir se criaremos ou não um novo arquivo. Chamamos o recém criado *RestartStream* para inicializar o *stream* interno e criamos o *stream* de descompressão. Após isso, populamos o *Array* de entradas e fechamos os *streams*. Escolhemos por fechar os *streams* ao final de cada método para liberar o arquivo enquanto nenhuma operação está ocorrendo. O método *PopulateEntries* possui a implementação como na **Listagem 14**.

Sua implementação, por mais que pareça complicada, é, na verdade, bastante simples. Lemos quatro *bytes* (*Integer*) e armazenamos em *NameLength*. Lembrando que precisamos converter o valor lido de *TBuffer* para *string* (*CopyFromBuffer*) e, como este número estará na forma binária, precisamos restaurar o seu conteúdo

inteiro (*BinToInt*). Após isso, iremos ler o nome do arquivo e armazenamos na entrada. Finalmente, lemos oito *bytes* (*Int64*) para o tamanho do arquivo e, também o armazenamos na entrada. Como estamos lendo apenas a descrição de cada entrada e não seu conteúdo, chamamos *ReadFile* apenas para ler e descartar o conteúdo do arquivo, pois este não será necessário agora. Veja como fica nosso método *ReadFile* agora observando a **Listagem 15**, ou seja, simplesmente lemos o conteúdo do arquivo e, se necessário, gravamos no *stream* de saída.

Antes de iniciar a implementação da adição de um arquivo, vamos ao método *CopyEntireStream*. (**Listagem 16**). Este método copia todo o conteúdo do *stream* de leitura para o *stream* de escrita. Vamos então, finalmente, ao método *Add* previsto na **Listagem 17**.

Basicamente, iniciamos com a preparação dos *streams*. Note que estamos criando os *streams* de compressão e descompressão para o mesmo *stream* interno. Sim, isso é permitido. Após isso, copiamos todo o conteúdo atual de volta para o *stream*. Você deve estar se perguntando: mas não seria só uma questão de posicionar o *stream* de saída no final do arquivo? Pois a resposta é: não. O *stream* de compressão precisa ser único no arquivo inteiro. Se fecharmos ele, abrirmos o arquivo novamente, e continuarmos escrevendo, quando formos ler, o *stream* de descompressão pensará que o final do arquivo chegou quando fechamos o *stream* a primeira vez, ou seja, não irá funcionar.

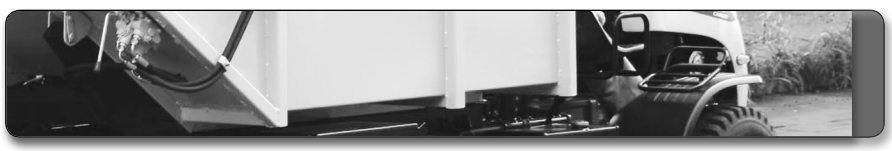
Após isso, instanciamos um *stream* temporário para a leitura do arquivo a ser adicionado, inserimos uma entrada no *Array* e escrevemos o tamanho do nome, o nome e o tamanho do arquivo no *stream* de escrita. Finalmente, escrevemos o conteúdo do arquivo. Agora veja o método de extração de um arquivo (**Listagem 18**).

Listagem 17. Método Add da classe

```
function TCompressor.Add(const FilePath: string):
  TCompressionEntry;
var
  Buffer: TBuffer;
  ATempStream: TFileStream;
  AFileName: string;
begin
  RestartStream(fmOpenReadWrite);
  FStreamRead := TDecompressionStream.Create(
    FInternalStream);
  FStreamWrite := TCompressionStream.Create(clMax,
    FInternalStream);
  CopyEntireStream;

  try
    ATempStream := TFileStream.Create(FilePath, fmOpenRead);
    try
      AFileName := ExtractFileName(FilePath);

      SetLength(FEntries, Length(FEntries) + 1);
      Result.Path := AFileName;
      Result.FileSize := ATempStream.Size;
      FEntries[High(FEntries)] := Result;

      Buffer := CopyToBuffer(IntToBin(Length(AFileName)), 4);
      FStreamWrite.Write(Buffer, 4);

      Buffer := CopyToBuffer(AFileName, Length(AFileName));
      FStreamWrite.Write(Buffer, Length(AFileName));

      Buffer := CopyToBuffer(IntToBin(ATempStream.Size), 8);
      FStreamWrite.Write(Buffer, 8);

      FStreamWrite.CopyFrom(ATempStream, ATempStream.Size);
    finally
      ATempStream.Free;
    end;
  finally
    CloseStreams;
  end;
end;
```

Listagem 18. Método de extração dos arquivos no stream

```
procedure TCompressor.Extract(Index: Integer; const
  ExtractTo: string);
var
  Buffer: TBuffer;
  ATempStream: TStream;
  Entry: TCompressionEntry;
  i: Integer;
begin
  RestartStream(fmOpenRead);
  FStreamRead := TDecompressionStream.Create(
    FInternalStream);
  try
    Entry := FEntries[Index];

    for i := 0 to Index - 1 do
      ReadEntry(False);

    FStreamRead.Read(Buffer, 4);
    FStreamRead.Read(Buffer, BinToInt(CopyFromBuffer(
      Buffer, 4)));
    FStreamRead.Read(Buffer, 8);

    ATempStream := GetDestStream(
      IncludeTrailingPathDelimiter(ExtractTo) +
      Entry.Path);
    try
      ATempStream.CopyFrom(FStreamRead, Entry.
        FileSize);
    finally
      CleanDestStream(ATempStream);
    end;
  finally
    CloseStreams;
  end;
end;
```

Conclusão

E com isso finalizamos a primeira parte deste artigo. Até a próxima edição, onde continuaremos a construção do exemplo. ●