

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 2



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 GIES-SC. Hoje, ministra cursos, presta consultoria e atua como Software Engineer do projeto InstallAnywhere para a empresa norte-americana Macrovision Corporation.

Na edição anterior, iniciamos a criação de uma aplicação que faz uso intensivo de streams (Edição 93), que implementa compactação, download em múltiplos pacotes e resources. Desta vez, utilizaremos um *stream* temporário para escrever o arquivo que estamos lendo. Note que estamos utilizando *GetDestStream* e *CleanDestStream* para isso (veja última listagem da edição anterior). Portanto, poderemos estender essa classe futuramente para gerar a saída em um *stream* qualquer, não só em arquivos. Seguem suas implementações (não se esqueça da diretiva *virtual* neles. Veja na **Listagem 1** a implementação das funções *GetDestStream* e *CleanDestStream*.

Seu funcionamento também é básico: descartamos do arquivo *Index – 1* entradas para posicionar o cursor do *stream* de leitura; depois lemos o tamanho do nome, o nome e o tamanho do arquivo, também para posicionar o cursor; e,

finalmente, lemos o conteúdo do arquivo através de um *CopyFrom* do *stream* temporário.

Na **Listagem 2** podemos ver a implementação de outro método importante, *ReadEntry*. Estamos apenas lendo uma entrada e descartando-a caso *WriteResult* seja false. Para finalizar, vejamos o método *Delete* descrito na **Listagem 3**.

Este é provavelmente o método mais complicado da classe. Vamos precisar de um *stream* temporário que será, na verdade, o mesmo arquivo final, só que sem o arquivo *deletado*. Tivemos de fazer isso porque, ao *deletarmos* o arquivo no mesmo *stream*, depois não conseguimos reduzir o seu *size* para o tamanho novo (menor). Esse é o comportamento padrão de um *TCompressionStream*.

O primeiro passo será então copiar para este arquivo novo todas as entradas até a de número *Index (ReadEntry(True))*. Depois ignoramos uma entrada (*ReadEntry(False)*) e copiamos o resto do

arquivo (*CopyEntireStream*). Em seguida, substituímos o arquivo final atual pelo novo e *repopulamos* as entradas, pois estas foram modificadas.

Pronto. Isso é tudo o que precisávamos para termos uma classe totalmente funcional com compactação para múltiplos arquivos. Agora precisamos construir uma *interface* com o usuário.

Criando um exemplo completo de compactação

No projeto atual, você deve ter ignorado o *Form* padrão, gerado pelo Delphi. Agora, volte a ele, mude seu nome para "fmCompression" e adicione um componente *TToolBar*. A este, adicione cinco *TToolButtons* ("tbNew", "tbOpen", "tbAdd", "tbRemove" e "tbExtract"). Adicione também um *TDBGrid* ("dbgEntries"), um *TClientDataSet* ("cdsEntries"), um *TDataSource* ("dsEntries"), dois *TOpenDialogs* ("OpenDialog" e "AddFileDialog") e um *TSaveDialog* ("SaveDialog"). Opcionalmente, adicione um *TImageList* para adicionar imagens ao *TToolBar*.

Configure *dbgEntries*, apontando-o para *dsEntries* e este para o *cdsEntries*. Na propriedade *Options* de *dbgEntries*, acrescente *dgMultiSelect*. Adicione o seguinte filtro aos componentes *OpenDialog* e *SaveDialog*: "ClubeDelphi Zip File (*.cdz)|*.cdz". Seu filtro inicial deve ser configurado para "*.cdz". Dê também um título aos diálogos. Adicione a opção *ofAllowMultiSelect* à propriedade *Options* do componente *AddFileDialog*.

Abra o *Fields Editor* do componente *cdsEntries* e adicione os seguintes campos: "PATH" (*string* - *size*: 256), "FILE_SIZE" (*largeint*) e "ENTRY_INDEX" (*integer* - *visible*: *False*). Agora clique com o botão direito em *cdsEntries* e selecione *Create DataSet*.

O componente *cdsEntries* armazenará as entradas do *TCompressor*. Estamos utilizando um *TClientDataSet* com dados em memória pela facilidade em se trabalhar com os dados e com o *TDBGrid*. Opcionalmente, você pode configurar a propriedade *IndexFieldNames* do *cdsEntries* para "PATH" para ordenar as entradas por este campo. Posicione os componentes conforme a **Figura 1**.

Antes da implementação de qualquer evento, declare *FCompressor* (*TCompressor*)

Listagem 1. Métodos *GetDestStream* e *CleanDestStream*

```
function TCompressor.GetDestStream(const DestFilePath:
  string): TStream;
begin
  Result := TFileStream.Create(DestFilePath, fmCreate);
end;
procedure TCompressor.CleanDestStream(const DestStream: TStream);
begin
  DestStream.Free;
end;
```

Listagem 2. Método *ReadEntry*

```
procedure TCompressor.ReadEntry(WriteResult: Boolean);
var
  Buffer: TBuffer;
  NameLength: Integer;
begin
  FStreamRead.Read(Buffer, 4);
  if WriteResult then
    FStreamWrite.Write(Buffer, 4);
  NameLength := BinToInt(CopyFromBuffer(Buffer, 4));
  FStreamRead.Read(Buffer, NameLength);
  if WriteResult then
    FStreamWrite.Write(Buffer, NameLength);
  FStreamRead.Read(Buffer, 8);
  if WriteResult then
    FStreamWrite.Write(Buffer, 8);
  ReadFile(BinToInt64(CopyFromBuffer(Buffer, 8)), WriteResult);
end;
```

Listagem 3. Método *Delete*

```
procedure TCompressor.Delete(Index: Integer);
var
  I: Integer;
  ATempStream: TFileStream;
begin
  RestartStream(fmOpenRead);
  try
    ATempStream := TFileStream.Create(FFilePath + '.tmp', fmCreate);
    try
      FStreamWrite := TCompressionStream.Create(cIMax, ATempStream);
      FStreamRead := TDecompressionStream.Create(FInternalStream);
      for i := 0 to Index - 1 do
        ReadEntry(True);
        ReadEntry(False);
        CopyEntireStream;
      finally
        CloseStreams;
        ATempStream.Free;
      end;
    finally
      CloseStreams;
    end;
  end;
  DeleteFile(FFilePath);
  RenameFile(FFilePath + '.tmp', FFilePath);
  RestartStream(fmOpenRead);
  FStreamRead := TDecompressionStream.Create(FInternalStream);
  try
    PopulateEntries;
  finally
    CloseStreams;
  end;
end;
```

na seção *private* da classe *TfmCompression* e vamos à implementação do método *ReloadEntries* conforme a **Listagem 4**.

Este método é muito simples, estamos apagando os dados de *cdsEntries* e adicionando a ele todas as entradas do compressor. Além de estarmos controlando a atualização do *dbgEntries* através dos métodos *DisableControls* e *EnableControls*. Vamos implementar as operações *New* e *Open*. Para isto, manipule o evento *OnClick* do botão *tbNew* de acordo com a **Listagem 5**.

O evento *OnClick* do botão *tbOpen* deve

ser implementado da mesma maneira. Porém, troque *SaveDialog* por *OpenDialog* e de *True* para *False* na construção de *FCompressor*. Agora, uma vez criado *FCompressor*, é necessário destruí-lo. Faremos isso através do evento *OnDestroy* do formulário:

```
if Assigned(FCompressor) then
  FreeAndNil(FCompressor);
```

Finalmente, vamos as outras operações. Começamos pelo *Add*. Portanto manipule o evento do botão *tbAdd* seguindo a codificação prevista na **Listagem 6**.

Listagem 4. Código do método ReloadEntries

```
procedure TfmCompression.ReloadEntries;
var
  i: Integer;
  Entry: TCompressionEntry;
begin
  cdsEntries.DisableControls;
  cdsEntries.EmptyDataSet;
  try
    for i := 0 to FCompressor.EntriesCount - 1 do
    begin
      Entry := FCompressor.Entries[i];

      cdsEntries.Append;
      cdsEntriesPATH.AsString := Entry.Path;
      cdsEntriesFILE_SIZE.AsLargeInt := Entry.
        FileSize;
      cdsEntriesENTRY_INDEX.AsInteger := i;
      cdsEntries.Post;
    end;
  finally
    cdsEntries.EnableControls;
  end;
end;
```

Listagem 5. Código do botão Novo

```
procedure TfmCompression.tbNewClick(Sender: TObject);
var
  AFileName: string;
begin
  if SaveDialog.Execute then
  begin
    AFileName := SaveDialog.FileName;
    if Assigned(FCompressor) then
      FCompressor.Free;
    FCompressor := TCompressor.Create(AFileName, True);
    Caption := ExtractFileName(AFileName);
    ReloadEntries;
  end;
end;
```

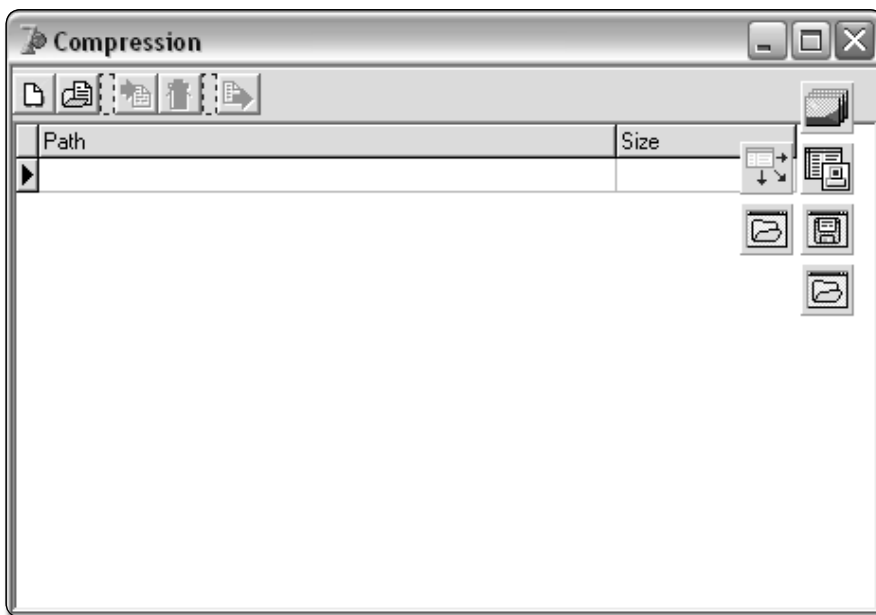


Figura 1. Interface gráfica do projeto Compression



Mais um método bastante simples, no qual passamos ao *FCompressor* todos os arquivos a serem adicionados. Para a operação de *Extract*, implemente o evento *OnClick* do botão *tbExtract* (Listagem 7)

A função *SelectDirectory* não é muito conhecida. Esta serve para que o usuário possa selecionar um diretório, como um diálogo qualquer. Passamos o *Caption*, o diretório inicial e para onde enviar o diretório selecionado. Então, para cada linha selecionada no *dbgEntries* estamos chamando o método *Extract* do compressor passando o *Index* da entrada, armazenado no método *ReloadEntries*.

Novamente, o caso mais complicado. Manipule o evento *OnClick* do botão *tbDelete* para usando o código da Listagem 8.

Para entender o código anterior, precisamos pensar em uma operação na qual estão sendo removidos vários registros de uma coleção. Caso removamos um registro e, posteriormente, tentemos remover outro de índice maior, o segundo registro estará sendo removido de maneira incorreta. Portanto, vamos fazer a operação pegando do maior índice a ser removido para o menor.

Por facilidade, vamos utilizar um *TStringList* para armazenar os índices. A primeira tarefa é então alimentar o *TStringList* com os índices das entradas a serem removidas. A segunda compreende procurar o próximo índice mais alto da lista, guardá-lo e salvar sua posição na coleção de índices. Após isso, *deletamos* a entrada e a excluimos da coleção. Quando todos os registros forem removidos, simplesmente carregamos todas as entradas novamente.

Pronto. Nossa versão simplificada do *WinZip* para trabalhar com arquivos *.cdz* está pronta para ser utilizada. Você pode agora tentar adicionar outros recursos como criptografia ao seu arquivo compactado. Além disso, como criamos uma classe, você pode embutir essa ferramenta em sua aplicação para transportar arquivos com mais facilidade.

Trabalhando com Resources

Todo programa Windows de interface gráfica apresenta uma série de recursos que não são apenas código compilado, como:

ícones, cursores, sons, imagens, dentre outros. O próprio Windows nos permite inserir, em nossos executáveis, qualquer tipo de arquivo. Estes podem ser compilados em arquivos de recursos (.RES) e então embutidos em uma aplicação.

Esta é a receita de bolo para a utilização de arquivos de recursos:

1. Criamos um arquivo .RC, referenciando os arquivos que irão constar no .RES;
2. Compilamos o arquivo .RC para .RES;
3. No Delphi, utilizamos a diretiva {\$R Arquivo.RES} para importar o arquivo de recursos;
4. Fazemos uso da classe *TResourceStream* para a leitura dos arquivos.

Nosso primeiro exemplo é um clássico da customização dos recursos de uma aplicação. Certamente, se nunca utilizou, irá utilizá-lo um dia em sua aplicação.

É muito comum fazermos uso de bibliotecas externas ao nosso programa. Contudo, o fato de termos de distribuí-las junto ao nosso executável nos incomoda bastante. Portanto, vamos incluí-las em nossa aplicação. Deste modo, poderíamos embutir também o executável final e extrair todos eles antes da execução propriamente dita.

Crie (com o próprio bloco de notas) um arquivo com a extensão .RC no disco (por exemplo: *Recursos.RC* na pasta da aplicação). Seu conteúdo ficaria parecido com o seguinte:

```
DLL DLLFILE caminho_para_biblioteca.dll
```

A primeira coluna contém a chave que identificará o arquivo como recurso e poderá ser representada por qualquer palavra. A segunda coluna contém o tipo de arquivo. Os mais comuns são: *BITMAP*, *ICON*, *CURSOR*, *JPEG*, *WAVE*, *TEXT* e *RCDATA* (genérico). A terceira coluna contém o caminho (absoluto ou relativo) para o arquivo. Lembre-se de que inúmeras linhas (entradas) podem ser adicionadas ao arquivo .RC. Aponte para qualquer biblioteca em seu sistema para testar.

Com o fonte .RC pronto, devemos compilá-lo para um .RES. Para isso, você pode utilizar o programa *BRCC32.exe*, na pasta *bin* do *Delphi*. Abra uma linha de comando e digite:

Listagem 6. Código do botão Add

```
procedure TfmCompression.tbAddClick(Sender: TObject);
var
  i: Integer;
  Entry: TCompressionEntry;
begin
  cdsEntries.DisableControls;
  try
    if AddFileDialog.Execute then
      begin
        for i := 0 to AddFileDialog.Files.Count - 1 do
          begin
            Entry := FCompressor.Add(AddFileDialog.
              Files[i]);
            cdsEntries.Append;
            cdsEntries.PATH.AsString := Entry.Path;
            cdsEntries.FILE_SIZE.AsLargeInt := Entry.
              FileSize;
            cdsEntries.ENTRY_INDEX.AsInteger := FCompressor.
              EntriesCount - 1;
            cdsEntries.Post;
          end;
        end;
      finally
        cdsEntries.EnableControls;
      end;
    end;
  end;
```

Listagem 7. Código do botão de extração

```
procedure TfmCompression.tbExtractClick(Sender: TObject);
var
  ExtractTo: string;
  i: Integer;
begin
  if SelectDirectory('Extract to', 'c:\', ExtractTo)
  then
    begin
      cdsEntries.DisableControls;
      try
        for i := 0 to dbgEntries.SelectedRows.Count-1 do
          begin
            cdsEntries.GotoBookmark(Pointer(dbgEntries.
              SelectedRows.Items[i]));
            FCompressor.Extract(cdsEntries.ENTRY_INDEX.
              AsInteger, ExtractTo);
          end;
        finally
          cdsEntries.EnableControls;
        end;
      end;
    end;
  end;
```

Listagem 8. Código para remoção de arquivo

```
procedure TfmCompression.tbRemoveClick(Sender: TObject);
var
  i: Integer;
  ToRemove: TStrings;
  EntryIndexToRemove, iToRemove: Integer;
begin
  cdsEntries.DisableControls;
  ToRemove := TStringList.Create;
  try
    for i := 0 to dbgEntries.SelectedRows.Count-1 do
      begin
        cdsEntries.GotoBookmark(Pointer(dbgEntries.
          SelectedRows.Items[i]));
        ToRemove.Add(cdsEntries.ENTRY_INDEX.AsString);
      end;
    while ToRemove.Count > 0 do
      begin
        EntryIndexToRemove := -1;
        iToRemove := -1;
        for i := 0 to ToRemove.Count - 1 do
          begin
            if StrToInt(ToRemove[i]) > EntryIndexToRemove
            then
              begin
                iToRemove := i;
                EntryIndexToRemove := StrToInt(ToRemove[i]);
              end;
            end;
          end;
        FCompressor.Delete(EntryIndexToRemove);
        ToRemove.Delete(iToRemove);
      end;
    finally
      cdsEntries.EnableControls;
      ToRemove.Free;
    end;
  end;
  ReloadEntries;
end;
```

```
[caminho para a pasta bin]\brcc32.exe  
[arquivo .RC]
```

Na pasta atual será gerado um arquivo *.RES* com o mesmo nome de seu fonte *.RC*. Crie um novo projeto e salve-o como "ResourceDLL.dpr". Na *unit* do formulário principal, digite:

```
($R caminho_para_o_arquivo_de_recurso_  
compilado.RES)
```

Adicione um botão ao formulário e implemente seu evento *OnClick* igual a **Listagem 9**. Inicie o programa e clique no botão. A DLL será extraída para a mesma pasta, com o nome *extracted.dll*.

Relembrando o conceito de *streams* do início do artigo, finalmente fechamos as principais classes que o assunto abran-

ge. O *TResourceStream* é muito parecido com o *TMemoryStream* e necessita de três parâmetros para sua construção. O primeiro é o *HModule*, ou seja, o *handle* do módulo retornado no carregamento de uma aplicação/DLL. No caso, o arquivo de recursos foi compilado junto à aplicação corrente, portanto, utilizaremos *HInstance*. No caso de quisermos carregar recursos de outras aplicações passaríamos o *HModule* da aplicação desejada. O segundo parâmetro é a chave que identifica o recurso, informada no *.RC*. E o terceiro é o tipo de recurso, também informado no *.RC*.

A classe *TResourceStream*, bem como o *TMemoryStream*, herdam de *TCustomMemoryStream*. Esta última introduz dois

métodos especiais: *SaveToFile* e *SaveToStream*. Isso é muito útil porque não precisamos de um *TFileStream* para salvar seu conteúdo. O método *SaveToFile* já faz isso para nós. E esta é a operação presente em nosso exemplo. Estamos simplesmente criando o *stream* e chamando seu método *SaveToFile*. Muito simples, não?

Como dica, podemos utilizar o plug-in *Simple Resource Editor* (**Figura 2**), de minha autoria, para a manipulação dos arquivos *.RC*. Com ele você pode facilmente adicionar novos arquivos (buscando-os por diálogos especiais), compilar para arquivos *.RES*, dentre outras funcionalidades bastante úteis. O assistente lhe ajuda inclusive a, rapidamente, selecionar o tipo de recurso a ser adicionado. Seu download pode ser feito através do *CodeCentral* da *CodeGear* (cc.codegear.com), procurando-se pelo autor *Gustavo Chaurais*. Seu código fonte está incluso e é muito interessante para estudo.

Listagem 9. Código do botão de extração dos recursos

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Stream: TResourceStream;  
const  
    EXTRACTED_DLL_NAME = 'extracted.dll';  
begin  
    Stream := TResourceStream.Create(HInstance, 'DLL',  
    'DLLFILE');  
    try  
        Stream.SaveToFile(EXTRACTED_DLL_NAME);  
        MessageDlg('Arquivo extraído para: ' +  
            IncludeTrailingPathDelimiter(ExtractFilePath(  
                Application.ExeName)) + EXTRACTED_DLL_NAME,  
            mtInformation, [mbOk], 0);  
    finally  
        FreeAndNil(Stream);  
    end;  
end;
```

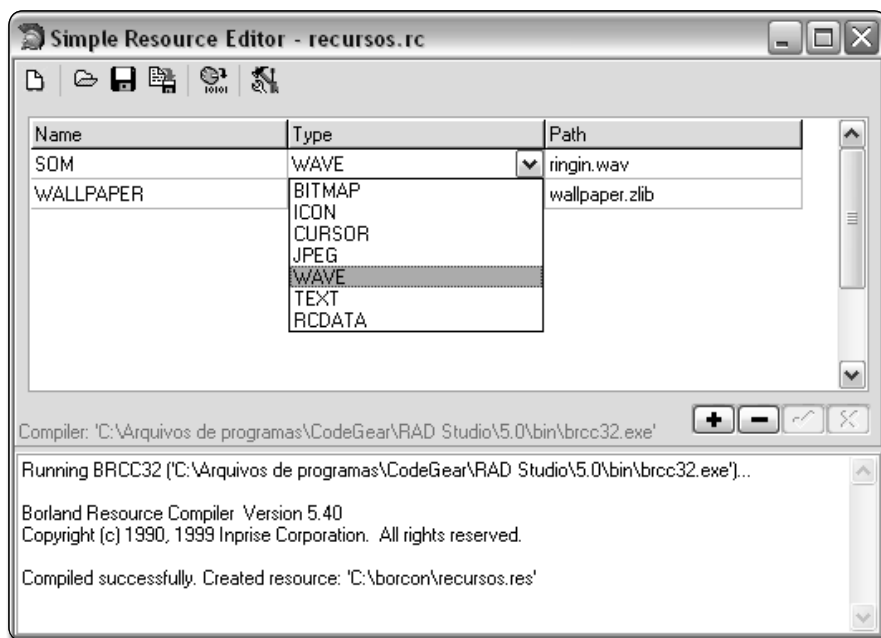


Figura 2. Simple Resource Editor

ClubeDelphi PLUS www.devmedia.com.br/dubeddelphi/portal.asp
Acesse agora o mesmo portal do assinante ClubeDelphi e assista a uma vídeo aula de Adriano Santos que mostra como trabalhar com o plug-in Simple Resource Editor.
www.devmedia.com.br/articles/viewcomp.asp?comp=5476&hl=

Vamos agora misturar um pouco as coisas. Construiremos um exemplo que utilizará um arquivo comprimido através do exemplo *Compression*.

Crie um novo projeto e salve-o como *ResourceZip.dpr*. Adicione a ele uma nova *Unit* e salve-a como *ResCompressor.pas*. Adicione também a *Unit Compressor.pas*, construída anteriormente.

Na seção *Uses* de *ResCompressor*, insira: *Compressor*, *Classes*, *SysUtils* e *Windows*. Agora, declare uma classe *TResCompressor*, estendendo *TCompressor*. Declare e implemente o seguinte método (com a diretiva *reintroduce*):

```
constructor TResCompressor.Create(const  
    ResName:  
        string);  
begin  
    FResName := ResName;  
    inherited Create('', False);  
end;
```

E declare *FResName* na seção *private* como sendo do tipo *string*. Declare também e implemente o seguinte método:

```

procedure TResCompressor.
ExtractToStream(Index:
Integer; const ExtractTo: TStream);
begin
FDestStream := ExtractTo;
inherited Extract(Index, '');
end;

```

Adicione também *FDestStream* à seção *private* da classe. Este método será utilizado para extrairmos um *resource* para um *stream* qualquer.

Você se lembra dos três métodos virtuais da classe *TCompressor*? Pois, tenha certeza de que não estão na seção *private* da classe *TCompressor* (devem estar preferencialmente em *protected*) e redeclare-os na classe *TResCompressor* com a diretiva *override*. O atributo de *TCompressor*, *FInternalStream*, também deve ser colocado preferencialmente na sua seção *protected*.

O método *RestartStream* deve ser utilizado para a inicialização do *stream* interno. Portanto, o inicializaremos como um *TResourceStream* conforme segue:

```

procedure TResCompressor.RestartStream(
Mode: Word);
begin
FInternalStream := TResourceStream.Create(
HInstance, FResName, RT_RCDATA);
end;

```

O método *GetDestStream* é utilizado para a inicialização de uma *string* onde será gravado o arquivo extraído. Neste caso, quando for chamado o método *ExtractToStream*, o *stream* desejado será gravado localmente em *FDestStream*. Podemos então simplesmente retorná-lo. Veja:

```

function TResCompressor.GetDestStream(const
DestFilePath: string): TStream;
begin
Result := FDestStream;
end;

```

E, uma vez que o *stream* veio de fora para o *ExtractToStream*, não é nossa responsabilidade limpá-lo. Provavelmente, quem chamou o método vai querer ler o que foi extraído. Portanto, o método *CleanDestStream* ficará vazio. Note a seguir:

```

procedure TResCompressor.
CleanDestStream(const
DestStream: TStream);
begin
end;

```

Para fechar esta classe, você pode, opcionalmente, disparar uma exceção caso sejam chamados os métodos *Add*, *Delete* e *Extract*. Para tanto, declare-os como *virtual* na classe *TCompressor* e *override* na classe *TResCompressor*.

Agora, vamos construir nosso arquivo de recursos. Abra o programa *Compression* e crie um arquivo contendo algumas figuras do tipo *jpg*. Crie um arquivo *.RC* e adicione a ele a seguinte linha:

```

IMAGESZIP RCDATA caminho_para_arquivo_
compactado.cdz

```

Compile o *.RC* para um *.RES* e o refencie no formulário principal, através da diretiva *{\$R Arquivo.RES}*. Adicione ao formulário principal da aplicação um *TPaintBox* (aba *System*). Na sua *Unit*, insira *jpeg* na seção *Uses* e manipule o evento *OnCreate* digitando o código da **Listagem 10**.

O que este método está fazendo se refere à criação de um *TResCompressor*, o qual irá carregar o *.RES*, contendo o arquivo das figuras compactado. Após sua abertura, podemos chamar seu método *ExtractToStream* passando um índice aleatório. Finalmente, carregamos à figura em *FJpeg* (declare-a na classe do formulário). Utilizamos um *TMemoryStream* para carregá-la. Agora, no evento *OnPaint* do *TPaintBox*, implemente o código seguinte.

```

procedure TForm1.PaintBox1Paint(Sender:
TObject);
begin
if Assigned(FJpeg) then
PaintBox1.Canvas.Draw(0, 0, FJpeg);
end;

```

Pronto. Agora, toda vez que você entrar no programa, uma figura aleatória de sua lista será carregada no *TPaintBox*. Você pode utilizar isso no *splash screen* de sua aplicação, por exemplo.

E, se você ainda não cansou de exemplos, vamos finalizar com um muito pouco conhecido: como alterar *resources* de outras aplicações.

Ainda no exemplo anterior, adicione a seguinte linha na seção *Interface* da *Unit* do formulário:

```

resourcestring
TITLE = 'not changed';

```

Resource Strings são como constantes. No entanto, podemos modificar seu conteúdo utilizando programas que alterem *resources* como o *Resource Hacker* ou o *Workshop Editor*. Essas são muito utilizadas em programas que desejamos internacionalizar, pois podemos alterar seu conteúdo para outra língua sem ter de recompilar a aplicação toda. Adicione como primeira linha do manipulador do evento *OnCreate* do formulário o seguinte:

```

Caption := TITLE;

```

Crie um novo projeto e salve-o como "ResourceUpdate.dpr". No formulário principal, adicione: três *TLabel*, três *TEdit* ("edProgram", "edResKey" e "edValue"), três *TButton* ("btList", "btRCDATA" e "btString") e um *TMemo* ("mmList"). Posicione os componentes e configure seus *Captions* conforme a **Figura 3**.

Vamos iniciar pela listagem dos recursos. Implemente o tratamento para o evento *OnClick* do *btList* observando o código da **Listagem 11**.

Estamos, primeiramente, utilizando a função *LoadLibraryEx* para carregar

Listagem 10. Código OnCreate do formulário

```

procedure TForm1.FormCreate(Sender: TObject);
var
AResCompressor: TResCompressor;
AIndex: Integer;
AMemoryStream: TStream;
begin
AResCompressor := TResCompressor.Create('IMAGESZIP');
try
AMemoryStream := TMemoryStream.Create;
try
Randomize;
AIndex := RandomRange(0, AResCompressor.EntriesCount);
AResCompressor.ExtractToStream(AIndex, AMemoryStream);

AMemoryStream.Position := 0;
FJpeg := TJpegImage.Create;
FJpeg.LoadFromStream(AMemoryStream);
finally
AMemoryStream.Free;
end;
finally
AResCompressor.Free;
end;
end;

```

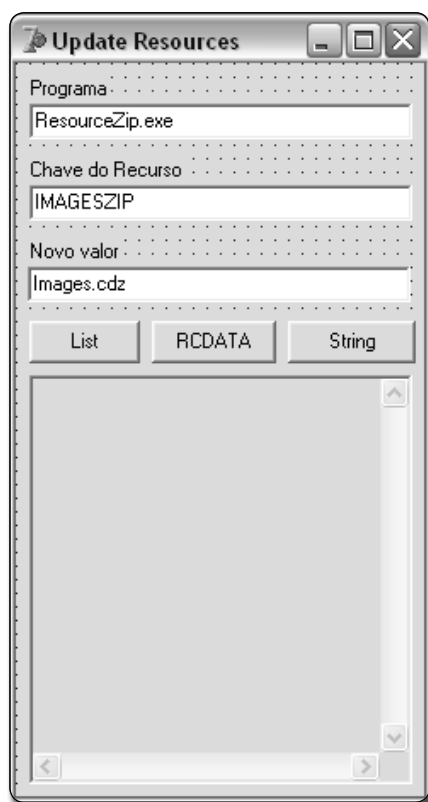


Figura 3. Interface do exemplo ResourceUpdate

o programa. Esta função pode ser executada tanto para bibliotecas quanto executáveis e retorna o *handle* do módulo. Após isso, chamamos o método *EnumResourceNames*, uma vez para o tipo *RCDATA* e uma vez para o tipo *STRING*. Passamos também um ponteiro para a função que será chamada para cada recurso (*@EnumResNamesProc*) e um ponteiro para a lista de *strings* do *mmList*.

Feito isso, declare (logo acima do método recém implementado, sem ser da classe *TForm1*) a função *EnumResNamesProc* (**Listagem 12**).

Quando o recurso for do tipo *string*, na verdade, ele será um *StringTable*. Essas tabelas são a maneira que *strings* são adicionadas aos arquivos de recurso. Em cada uma, teremos até 16 *strings*. Cada *string* é identificado por uma chave própria, que é calculada da seguinte forma: $([\text{NOME DO RECURSO}] - 1) * 16$. Posteriormente, podemos utilizar a função *LoadString* passando esta chave para lermos cada *string*.

O método implementado, conforme comentado, será chamado para cada recurso encontrado. Se o recurso for um *RCDATA*, apenas adicionamos na lista o nome do recurso. Caso seja um *String Table*, varremos a tabela lendo cada *string* através do método *LoadString*. Para pegarmos o nome de recurso da tabela, utilizamos a função *LoWord*, pois, nos interessa apenas os *bytes* menos significativos.

A lista já está pronta para ser preenchida. Você verá primeiro os recursos *RCDATA* do aplicativo e, depois, os recursos do tipo *String Table*. No segundo caso, aparecerá também a chave identificadora de cada *string*.

Agora, declare e implemente o método *UpdateResourceInProgram* (**Listagem 13**).

Este é o método que será utilizado para a modificação efetiva dos *resources*. Ele consiste em chamarmos *BeginUpdateResource* para iniciar a alteração, o qual nos retorna o *Handle* do programa. Se a alteração for do tipo *RCDATA*, iniciamos um *stream* para a leitura do arquivo que irá substituir o recurso atual; reservamos memória para todo este valor (note que o arquivo será totalmente carregado em memória); lemos o arquivo no *buffer* e, por fim, chamamos *UpdateResource*. Este é o método que irá modificar o recurso e, para ele, devemos passar: o *Handle* da aplicação a modificar, o tipo de recurso, o nome do recurso (é interessante utilizar a função *MakeIntResource* para a formatação do valor), a linguagem do recurso (0 = Neutral), o *buffer* com os *bytes* e o tamanho a ser gravado.

Caso estejamos interessados na alteração de uma *string*, chamamos o método a ser implementado *GetStringTable* e gravamos, agora, toda a *String Table* novamente na posição correta (retornada por *GetResourceNameOfStrId*). Para aplicar as atualizações, chamamos *EndUpdateResource*.

Finalmente, implemente os métodos relacionados à construção do *String Table* conforme a **Listagem 14**.

Para a montagem do *String Table*, é necessário lermos todas as *strings* novamente, adicionando-as na tabela e alterando somente a *string* desejada. Perceba que as *strings* estão presentes na tabela da seguinte forma: [LENG-

Listagem 11. Código do evento *OnClick* do botão *btList*

```
procedure TForm1.btListClick(Sender: TObject);
var
  AHandle: THandle;
begin
  mmList.Clear;
  AHandle := LoadLibraryEx(PChar(edProgram.Text), 0, LOAD_LIBRARY_AS_DATAFILE);
  try
    mmList.Lines.Add('RCDATA RESOURCES');
    EnumResourceNames(AHandle, RT_RCDATA, @EnumResNamesProc, Integer(mmList.Lines));
    mmList.Lines.Add('');
    mmList.Lines.Add('STRING TABLES');
    EnumResourceNames(AHandle, RT_STRING, @EnumResNamesProc, Integer(mmList.Lines));
  finally
    FreeLibrary(AHandle);
  end;
end;
```

Listagem 12. Função *EnumResNamesProc*

```
function EnumResNamesProc(Module: HMODULE; ResType, ResName: PChar;
  Strings: TStrings): Boolean; stdcall;
var
  InitialString, i: Integer;
  Buffer: array[0..1023] of Char;
begin
  if ResType = RT_STRING then
  begin
    InitialString := (LoWord(Cardinal(ResName)) - 1) * 16;
    for i := 0 to 15 do
    begin
      if LoadString(Module, InitialString + i,
        Buffer, 1024) <> 0 then
        Strings.Add('> ' + IntToStr(InitialString + i)
          + ' - ' + string(Buffer));
    end;
  end
  else if ResType = RT_RCDATA then
  begin
    Strings.Add('> ' + ResName);
  end;
  Result := True;
end;
```

Listagem 13. Método UpdateResourceInProgram

```

procedure TForm1.UpdateResourceInProgram(const
  ResType: PChar);
var
  TempStream: TStream;
  ResHandle: THandle;
  Buffer: PChar;
  StringTable: WideString;
begin
  ResHandle := BeginUpdateResource(PChar(
    edProgram.Text), False);
  try
    if ResType = RT_RCDATA then
    begin
      TempStream := TFileStream.Create(
        edValue.Text, fmOpenRead);
      try
        GetMem(Buffer, TempStream.Size);
        TempStream.Read(Buffer^, TempStream.Size);
        UpdateResource(ResHandle, RT_RCDATA,
          MakeIntResource(edResKey.Text), 0,
          Buffer, TempStream.Size);
      finally
        TempStream.Free;
      end;
    end;

    FreeMem(Buffer);
  end
  else if ResType = RT_STRING then
  begin
    StringTable := GetStringTable(StrToInt(
      edResKey.Text), edValue.Text);
    UpdateResource(ResHandle, RT_STRING,
      MakeIntResource(GetResNameOfStrId(StrToInt(
        edResKey.Text))), 0, PWideChar(StringTable),
      Length(StringTable));
  end;
  finally
    EndUpdateResource(ResHandle, False);
  end;
end;

```

Listagem 14. Implementação dos métodos GetStringTable e GetResNameOfStrId

```

function TForm1.GetStringTable(StrId: Integer; const
  NewValue: string): WideString;
var
  ResName: Integer;
  Offset: Integer;
  StartPos: Integer;
  i: Integer;
  AHandle: THandle;
  Buffer: array[0..1023] of Char;
  StrRead: string;
begin
  ResName := GetResNameOfStrId(StrId);
  Offset := StrId mod 16;
  StartPos := (ResName - 1) * 16;
  Result := '';
  AHandle := LoadLibraryEx(PChar(edProgram.Text), 0,
    LOAD_LIBRARY_AS_DATAFILE);
  try
    for i := 0 to 15 do
    begin
      if i = Offset then
      begin
        Result := Result + Char(Length(NewValue)) +
          NewValue;
      end
      else
      begin
        if LoadString(AHandle, StartPos + i, Buffer,
          1024) <> 0 then
        begin
          StrRead := string(Buffer);
          Result := Result + Char(Length(StrRead)) +
            StrRead;
        end;
      end;
    end;
    Result := Result + StringOfChar(#0, Length(Result));
  finally
    FreeLibrary(AHandle);
  end;
end;

function TForm1.GetResNameOfStrId(StrId:
  Integer): Integer;
begin
  Result := (StrId div 16) + 1;
end;

```

TH][STRING]. O cálculo da posição inicial é feito com base no nome do recurso correspondente à *String Table* em questão. Já o *Offset* representa a posição da *string* a ser alterada na tabela. Para finalizar, devemos gerar o mesmo número de caracteres nulos correspondentes ao tamanho gerado. Assim, fechamos a construção de um *String Table* completo. Já o método *GetResNameOfStrId* é um simples cálculo para se saber o *resource name* do *String Table* com base no *id* de um *string*.

Teste seu programa apontando para o executável anteriormente criado *ResourceZip.exe* e clicando em *btList*. Você verá a lista com os recursos do programa. Para alterar um *string*, coloque no segundo *TEdit* o *id* da *string* (copie da lista) e dê um novo valor. Por exemplo, procure por “not changed”, correspondente ao *resourcestring* anteriormente adicionado, mude seu valor e execute o programa. Você verá que a *string* foi modificada com sucesso, através da barra de títulos. Gere também outro arquivo *.cdz* com diferentes figuras *jpeg* e faça uma modificação apontando para o nome do recurso (*IMAGESZIP*) e para o caminho ao arquivo *.cdz*. Abra o programa e você verá as imagens novas.

Conclusão

Por mais que achemos que sabemos tudo, sempre há algo para aprender. *Streams* não são um conteúdo difícil, porém, necessitam ser exercitados. Além disso, aumentamos um pouco mais nosso “arsenal” de possibilidades, com a utilização de poderosos compactadores de arquivos e arquivos de recurso bastante flexíveis. Agora, é só esperar a oportunidade certa para aplicar o que foi explicado em um cenário real. ●

Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

