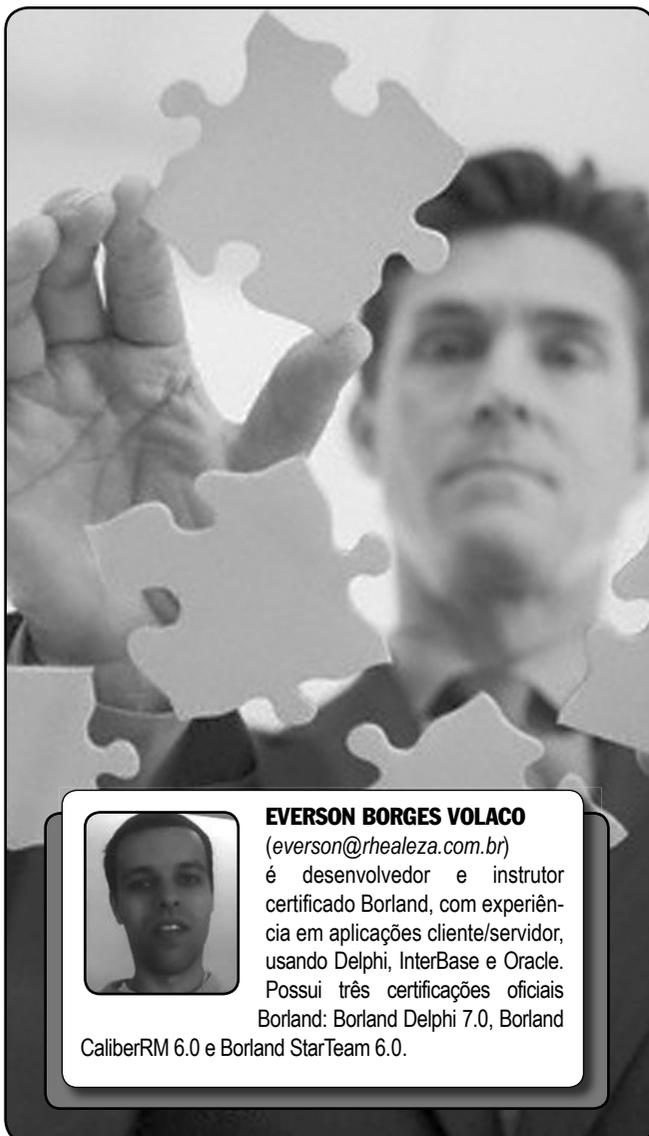


15 Dicas de Delphi

Dicas de Delphi para Desenvolvedores Iniciantes



EVERSON BORGES VOLACO

(everson@rhealeza.com.br)

é desenvolvedor e instrutor certificado Borland, com experiência em aplicações cliente/servidor, usando Delphi, InterBase e Oracle. Possui três certificações oficiais Borland: Borland Delphi 7.0, Borland CaliberRM 6.0 e Borland StarTeam 6.0.

Este artigo mostrará 15 dicas de Delphi (Win32) para desenvolvedores iniciantes e intermediários. Veremos os mais variados assuntos, desde o IDE até opções para facilitar a depuração de aplicações. Usuários que estão iniciando com o Delphi ou que possuem pouca experiência no mesmo, terão algumas dicas úteis para uso no dia a dia durante o desenvolvimento.

Nota: Todas as dicas são focadas no Delphi 7, porém, a maioria pode ser aplicada em qualquer versão do Delphi em aplicações Win32.

1. Definindo o carregamento dos formulários da aplicação

Quando criamos uma aplicação no Delphi, todos os formulários que construímos a partir da opção *New Form* por padrão ficam definidos como *Auto-create forms* dentro da janela *Options* do projeto (*Project > Options > Forms*).

Isso significa que, se você tiver, por exemplo, 10 formulários em sua aplicação, todos serão carregados em memória no momento que a aplicação for iniciada. Dependendo do tamanho de sua aplicação você pode ter sérios problemas de performance.

Uma boa prática é definir o formulário principal da aplicação como *Auto-create forms* e os demais como *Available forms* (**Figura 1**).

Uma observação apenas é que se o formulário a ser chamado estiver definido dentro da seção *Auto-create forms*, não precisamos criar a mesma via código, isso é, basta fazer a chamada ao método *Show* (ou *ShowModal*) para visualizar o formulário.

Já se o formulário estiver definido como *Available forms* precisaremos criar o mesmo através de seu construtor (método *Create*) antes de invocar o método *Show* para mostrá-lo na tela.

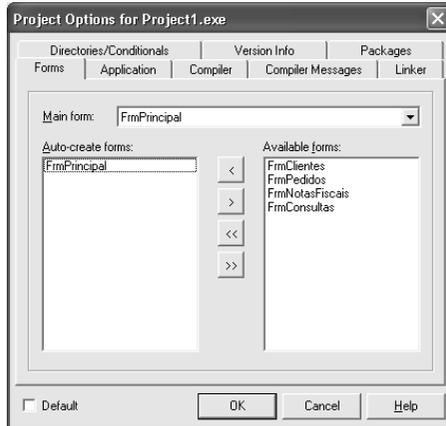


Figura 1. Definindo o comportamento de criação dos formulários do projeto

Nota: Você pode especificar a seção para que um formulário seja adicionado por padrão quando for criado (opção *New Form*) durante o projeto. Para isso, acesse a opção *Tools > Environment Options > Designer* e defina a opção *Auto create forms & data modules* disponível no item *Module creation options*.

2. Criando e destruindo formulários

Quando instanciamos um formulário em nossa aplicação através do seu construtor (método *Create*) e fazemos sua chamada através do método *ShowModal* para mostrar o mesmo ao usuário da aplicação, é importante destruímos sua instancia quando o usuário fechar o mesmo.

Uma boa opção para garantir que o formulário sempre será destruído, mesmo que algum erro ocorra, é utilizar a instrução *try...finally...end*. Essa instrução garante que todo o código escrito dentro da seção *finally* será executado mesmo que algum erro grave ocorra na aplicação. Veja o exemplo na **Listagem 1**.

Listagem 1. Usando o bloco *try...finally...end* para criação de um formulário

```
FrmClientes := TFrmClientes.Create(Self);
try
  FrmClientes.ShowModal;
finally
  FrmClientes.Release;
  FrmClientes := nil;
end;
```

No código da listagem anterior, criamos o *FrmClientes* e dentro do bloco *try* realizamos a chamada ao mesmo. Como estamos utilizando o método *ShowModal*, a aplicação só continuará a execução do código acima após o usuário ter fechado o *FrmClientes*. Quando essa operação ocorrer, o código do bloco *finally* será executado, realizando a destruição da instância do *FrmClientes*.

Nota: A instrução *try...finally...end* não funciona com o método *Show*, pois o mesmo não “trava” a aplicação até o formulário ser fechado. Nesse caso, o código do bloco *finally* será executado logo após a chamada ao

método *Show*, não permanecendo assim o formulário disponível para o usuário realizar suas operações.

3. Utilizando Data Modules

Outra boa prática é fazer o uso de Data Modules na aplicação para separar os componentes de acesso a dados da interface com o usuário (formulários). Utilizando Data Modules, podemos separar as regras de negócio que fazem acesso ao banco de dados de nossos formulários além de podermos reutilizar os *DataSets* em diferentes formulários.

Dependendo do tamanho de sua aplicação é aconselhável ainda criar vários Data Modules, cada um contendo os *DataSets* referente a um módulo ou segmento da aplicação. Normalmente criam-se um Data Module principal (por exemplo, *DMPrincipal*) o qual contém o componente de conexão com o banco de dados além de um ou dois *DataSets* auxiliares.

Nota: *DataSets* auxiliares não possuem nenhuma instrução SQL pré-definida; os mesmos são utilizados por funções distribuídas na aplicação para realizarem operações no banco de dados através de SQLs passados em tempo de execução.

São criados então, “n” outros Data Modules, contendo cada um diversos *DataSets*, sendo todos ligados a conexão disponível no Data Module principal. Na **Figura 2** temos um exemplo de como seria a criação dos Data Modules no carregamento da aplicação.

Nota: Neste exemplo, é necessário fazer a chamada ao *Create* do Data Module, para ter acesso aos seus métodos e componentes.

4. Habilitando opções básicas do IDE

Quando instalamos o Delphi algumas opções básicas e interessantes do IDE não vêm marcadas por padrão. Você pode estar definindo essas opções através do menu

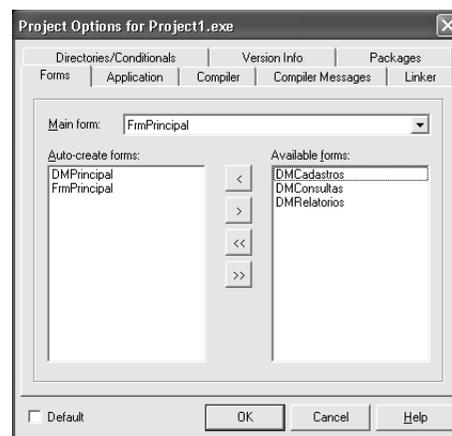


Figura 2. Utilizando Data Modules para o armazenamento dos componentes de acesso a dados da aplicação

Tools>Environment Options e Tools>Debugger Options.

Por exemplo, para que possamos visualizar a caixa de informações referente a compilação do projeto, mensagens de *warnings*, *hints* e *errors* marque a opção *Show compiler progress* disponível na aba *Preferences* da janela *Environment Options*.

Outra opção bastante útil que também não vem marcada por padrão é a *Show component captions* a qual, quando marcada passa, a mostrar o nome dos componentes não visuais dentro do formulário ou Data Module em tempo de design.

5. Passando instruções SQL em tempo de execução

Haverá situações onde precisaremos montar e enviar para o banco de dados instruções SQL de acordo com opções selecionadas pelo usuário na aplicação em tempo de execução. Componentes como *Query*, *SQLDataSet* e *ClientDataSet* possuem uma propriedade denominada *Params* a qual permite especificarmos parâmetros para filtros dentro da instrução SQL.

Porém, há situações onde toda a instrução SQL precisa ser montada em tempo de execução, tornando o uso da propriedade *Params* bastante complicada para ser utilizada. Uma saída é utilizar o caractere dois pontos (:) para definir os parâmetros via código ou atribuir os valores para os filtros diretamente dentro da instrução SQL.

Veja na **Listagem 2** dois exemplos de utilização de parâmetros em instruções SQL atribuídas em tempo de execução.

Listagem 2. Atribuindo parâmetros em tempo de execução

```
SQLDataSet1.Close;
SQLDataSet1.CommandText := 'SELECT CODIGO, NOME, '+
  'FONE FROM CLIENTES WHERE CODIGO = ' + Edit1.Text;
SQLDataSet1.Open;
ou
SQLDataSet1.Close;
SQLDataSet1.CommandText := 'SELECT CODIGO, NOME, '+
  'FONE FROM CLIENTES WHERE CODIGO = :COD';
SQLDataSet1.Params[0].AsInteger := StrToInt(Edit1.Text);
SQLDataSet1.Open;
```

Em ambas as instruções SQL, passamos o valor do *Edit1* como filtro para o campo CODIGO da tabela CLIENTES. Na primeira instrução, fazemos a atribuição do valor diretamente ao campo. Repare que apesar do campo CODIGO ser do tipo numérico (*integer*) não fazemos a conversão do valor armazenado na propriedade *Text* do *Edit1* para *Integer*.

Nesses casos, devemos passar o valor numérico como *string* dentro do código Delphi. Já no segundo caso, utilizamos o caractere dois pontos (:) para definir um parâmetro de nome COD. Terminada a montagem da instrução SQL utilizamos a propriedade *Params* para atribuir o valor do *Edit1* ao parâmetro.

Para esse caso, utilizamos a propriedade *AsInteger* e fazemos a conversão de tipo da propriedade *Text* do *Edit1* de *string* para *integer* (*StrToInt*). Para o primeiro caso, dependendo do tipo do campo que será utilizado no filtro dentro da instrução SQL, devemos passar o valor para o filtro de maneiras diferentes. Veja exemplos na **Tabela 1**.

Tipo	Valor	Exemplo
Integer	100	'WHERE CODIGO = ' + '100';
String	Cliente	'WHERE TIPO = uotedStr('Cliente');
Date	01/01/2006	'WHERE DTVENDA = ' + QuotedStr('01/01/2006')

Tabela 1. Exemplos de passagem de parâmetros

Para campos do tipo *string* ou *Date* você precisa passar o valor do filtro entre duas aspas simples, sendo isso que faz internamente o método *QuotedStr*. Você pode utilizar ainda, para o mesmo propósito, #39 ou '''. Veja os exemplos a seguir:

```
'SELECT * FROM FORNECEDORES WHERE UF = #39 + 'PR' + #39;
'SELECT * FROM FORNECEDORES WHERE CIDADE = ' + ''' +
  'Curitiba' + ''';
```

6. Passando novas instruções SQL através do ClientDataSet

Quando utilizamos o trio de componentes *SQLDataSet*, *DataSetProvider* e *ClientDataSet* para fazer acesso ao banco de dados, definimos a instrução SQL diretamente na propriedade *CommandText* do *SQLDataSet*. Podem haver situações onde podemos ter a necessidade de alterar a instrução SQL do *SQLDataSet* em tempo de execução. Entretanto, dependendo da arquitetura da aplicação essa alteração pode ser bastante complicada.

Imagine em uma aplicação três camadas onde o *SQLDataSet* fica na aplicação servidora e o *ClientDataSet* fica na aplicação cliente. Para esses casos o *DataSetProvider* possui uma opção que habilita enviarmos uma nova instrução SQL a partir da propriedade *CommandText* do *ClientDataSet*.

Essa propriedade chama-se *poAllowCommandText* e está disponível dentro da propriedade *Options*. Modificando para *True*, podemos modificar a instrução SQL definida em tempo de design no *SQLDataSet* através do *ClientDataSet* em tempo de execução.

7. Manipulando arquivos texto com TStringList

Quem ainda não teve a necessidade de trabalhar com arquivos textos no Delphi mais cedo ou mais tarde vai acabar se deparando com essa necessidade. O Delphi traz o tipo de dado *TextFile* o qual permite manipularmos arquivos texto a partir de aplicações Delphi.

Você pode, porém, manipular arquivos texto de uma forma mais simples através do uso da classe *TStringList*. A classe permite criar e manipular em memória uma lista de *strings*, como por exemplo, um arquivo texto.

Através do método *LoadFromFile* podemos carregar um arquivo texto para dentro de uma variável do tipo *TStringList*. Através do índice de cada item (ou linha) da lista, podemos acessar e/ou alterar o conteúdo desse item.

Realizada a manipulação no arquivo texto, podemos utilizar o método *SaveToFile* para salvar novamente o arquivo no sistema operacional. Veja na **Listagem 3** um exemplo da utilização dessa técnica.

Listagem 3. Trabalhando com arquivos textos no Delphi

```

procedure TfrmPrincipal.btnCarregarArquivoClick(
  Sender: TObject);
var
  { Variável que recebe o conteúdo do arquivo texto }
  arquivo: TStringList;
  i: Integer;
begin
  { Instancia a variável arquivo }
  arquivo := TStringList.Create;
  try
  { Carrega o conteúdo do arquivo texto para a
    memória }
  arquivo.LoadFromFile('c:\temp\arquivo1.txt');
  { Realiza um loop em toda a lista }
  for i := 0 to arquivo.Count - 1 do
  begin
  { Mostra o valor atual da linha }
  ShowMessage('O conteúdo original da linha ' +
    IntToStr(i) + ' é ' + arquivo[i]);
  { Atribui um novo valor para a linha corrente }
  arquivo[i] := 'Novo conteúdo da linha: ' +
    IntToStr(i);
  end;
  { Salva as alterações no arquivo }
  arquivo.SaveToFile('c:\temp\arquivo1.txt');
  finally
  { Libera a instancia da lista da memória }
  FreeAndNil(arquivo);
  end;
end;

```

8. Utilizando MultiSelect no DBGrid

Sem dúvida um dos componentes mais utilizados no desenvolvimento de aplicações com Delphi é o *DBGrid*. Veremos nessa dica como utilizar a opção de *MultiSelect* do componente. Habilitando essa opção, o usuário poderá selecionar mais de um registro (ao mesmo tempo) dentro do grid.

Adicione um *DBGrid* e altere a propriedade *Options* > *dg-MultiSelect* para *True*. Acesse um banco de dados qualquer e mostre no *DBGrid* registros de alguma tabela desse banco. Adicione um botão ao formulário e digite o código da **Listagem 4** em seu evento *OnClick*.

Listagem 4. MultiSelect no DBGrid

```

procedure TfrmPrincipal.btnMostrarSelecionadosClick(
  Sender: TObject);
var
  i: Integer;
  aux: string;
begin
  for i := 0 to DBGrid1.SelectedRows.Count - 1 do
  begin
  ClientDataSet1.GotoBookmark(pointer(
    DBGrid1.SelectedRows.Items[i]));
  aux := aux + IntToStr(ClientDataSet1.RecNo) +
    ' - ' + ClientDataSet1.FieldName(
    'CUSTOMER').AsString + #13;
  end;
  ShowMessage('Linhas selecionadas: ' + #13 + aux);
end;

```

No código da listagem anterior utilizamos a propriedade de *SelectedRows* do *DBGrid* para varrer todos os registros selecionados pelo usuário. Utilizamos o método *GotoBookmark* do *ClientDataSet* para posicionar o cursor no registro corrente selecionado. Veja na **Figura 3** a aplicação de exemplo em execução.

9. Utilizando as propriedades FetchOnDemand e PacketRecord do ClientDataSet

Uma das principais características do *ClientDataSet* é traba-



Figura 3. Trabalhando com multiseleção de registros em um *DBGrid*

lhar com os dados em memória, desconectado do banco de dados. Quando trazemos uma grande quantidade de registros do banco de dados para serem mostrados em um *DBGrid* por exemplo, o *ClientDataSet* primeiro carrega os registros em memória para só depois mostrá-los na tela para o usuário.

Dependendo do número de registros, o usuário pode demorar a visualizar os mesmos dentro do *DBGrid*. Isso ocorre porque, por padrão, a propriedade *PacketRecord* do *ClientDataSet* vem definida com o valor “-1”. Através desta propriedade podemos controlar o número de registros por pacote que serão buscados no servidor de banco de dados.

Essa propriedade está diretamente vinculada a propriedade *FetchOnDemand*, que também por padrão vem definida como *True*. Você pode diminuir o tempo para visualização de grandes quantidades de registros na tela alterando o valor da propriedade *PacketRecord*. Faça a seguinte simulação: crie uma aplicação para mostrar todos os registros de uma tabela grande do banco de dados em um *DBGrid* utilizando *ClientDataSet*.

Você verá que os dados demorarão a aparecer no *DBGrid* devido ao fato do *ClientDataSet* ter que carregar os registros antes em memória. Volte a aplicação em tempo de design e altere a propriedade *PacketRecords* para “10”. Rode novamente a aplicação. Os registros aparecerão instantaneamente no *DBGrid*.

O que ocorreu na verdade é que apenas os 10 primeiros registros foram carregados no *ClientDataSet*. A medida que navegamos pelos registros no *DBGrid* os demais registros vão sendo buscados no banco de dados, sempre em pacotes de 10 registros. Você pode ainda, se preferir, trazer apenas os 10 primeiros registros e através de uma opção na aplicação ir buscando os demais registros somente quando necessário.

Para isso, basta alterar a propriedade *FetchOnDemand* para *False* e utilizar o método *GetNextPacket* para buscar o próximo pacote de registros do servidor de banco de dados.

Nota: Essas propriedades são muito interessantes em aplicações multicamadas, possibilitando assim um melhor controle dos *ResultSets* retornados pela aplicação servidora e passando ao usuário final uma sensação de melhor performance da aplicação.

10. Propriedade UniDirectional do componente TQuery (BDE)

Essa dica é destinada aos desenvolvedores que utilizam BDE em conjunto com os componentes *DataSetProvider* e *ClientDataSet*. Programadores que utilizam o Delphi 5, versão a qual não possui a tecnologia dbExpress, para a criação de aplicações multicamadas ou o conjunto *Query + DataSetProvider + ClientDataSet*, precisam tomar cuidado com a duplicidade dos registros no buffer.

O *Query* possui uma propriedade denominada *UniDirectional*, a qual por padrão é definida como *False*. Dentro desse cenário como a tecnologia BDE não é unidirecional por natureza, como o dbExpress por exemplo, podemos ter duplicidade dos registros em cache, visto que tanto o *Query* como o *ClientDataSet* armazenarão os registros retornados pelo servidor de banco de dados.

Essa característica pode degradar a performance da aplicação, e o complicado é que nenhuma mensagem de erro é retornada para o desenvolvedor. Para esse tipo de aplicação é aconselhável a alteração da propriedade *UniDirectional* para *True*, fazendo assim com que o cache dos registros seja realizado apenas pelo *ClientDataSet*.

11. Atualizando informações da aplicação

É comum utilizarmos componentes como o *ProgressBar* e o *Gauge* para manter o usuário da aplicação informado sobre o progresso de um determinado processo dentro de nossa aplicação. Porém, dependendo do processo a aplicação pode travar sendo definida como *não respondendo* para o sistema operacional.

Para atualizar as informações da tela durante a execução de um processo pesado e/ou demorado de nossa aplicação podemos utilizar o método *ProcessMessages* da variável *Application*. Veja o exemplo na **Listagem 5**.

Listagem 5. Usando ProcessMessages para não travar o sistema

```
procedure TFormPrincipal.btnExecutarClick(
  Sender: TObject);
var
  i: Integer;
begin
  ProgressBar1.Min := 0;
  ProgressBar1.Position := ProgressBar1.Min;
  ProgressBar1.Max := 100000;
  for i := 1 to 100000 do
  begin
    ProgressBar1.StepIt;
    Label1.Caption := IntToStr(i);
    Application.ProcessMessages;
  end;
end;
```

No código da listagem anterior, fazemos um loop de 1 até 100000 atualizando a cada passagem do loop um *ProgressBar* e um *Label*. Caso não utilizarmos o método *ProcessMessages* o *Label* não será atualizado na tela em tempo real para o usuário da aplicação.

12. Controlando a versão da sua aplicação

Podemos definir e controlar o número da versão de nos-

sa aplicação Delphi. Dentro da janela de opções do projeto (*Project > Options*) na aba *Version Info* basta habilitar a opção *Include version information in project* e informar o número da versão corrente de sua aplicação dentro da seção *Module version number*.

Além do número da versão podemos entrar com diversas outras informações como nome da empresa, nome do produto, descrição, entre outras. O Delphi permite ainda que você crie novas chaves/valores dentro da seção *Key/Value* através da opção *Add Key* no menu de contexto.

Todas as informações são armazenadas no arquivo executável gerado pela nossa aplicação quando compilamos a mesma. Para acessar essas informações a partir do Windows, basta selecionar o arquivo executável e clicar sobre a opção *Propriedades* disponível no menu de contexto.

Através do número da versão do aplicativo podemos controlar atualizações e identificar a versão do aplicativo que está rodando em cada cliente. Você pode capturar esse número de versão em tempo de execução para mostrar ao usuário em uma tela do tipo *Sobre* ou *About*, por exemplo. Para isso, utilize o método da **Listagem 6**.

Listagem 6. Pegando a versão do arquivo

```
function GetVersaoArq: string;
var
  VerInfoSize: DWORD;
  VerInfo: Pointer;
  VerValueSize: DWORD;
  VerValue: PVSFixedFileInfo;
  Dummy: DWORD;
begin
  VerInfoSize := GetFileVersionInfoSize(PChar(
    ParamStr(0)), Dummy);
  GetMem(VerInfo, VerInfoSize);
  GetFileVersionInfo(PChar(ParamStr(0)), 0,
    VerInfoSize, VerInfo);
  VerQueryValue(VerInfo, '\\', Pointer(VerValue),
    VerValueSize);
  with VerValue^ do
  begin
    Result := IntToStr(dwFileVersionMS shr 16);
    Result := Result + '.' + IntToStr(
      dwFileVersionMS and $FFFF);
    Result := Result + '.' + IntToStr(
      dwFileVersionLS shr 16);
    Result := Result + '.' + IntToStr(
      dwFileVersionLS and $FFFF);
  end;
  FreeMem(VerInfo, VerInfoSize);
end;
```

13. Debugger – Adicionando condição a um Breakpoint

Um dos principais recursos disponíveis no *debugger* do Delphi sem dúvida alguma é o *breakpoint*. Através dele, podemos fazer com que nossa aplicação pare em um determinado ponto do código para que possamos depurá-lo a fim de encontrar um erro ou uma falha na lógica.

Porém, existem situações onde o problema encontra-se dentro de um loop (comando *while..do* ou *for..do*), e ao adicionarmos o *breakpoint* temos que depurar dentro do loop utilizando as teclas F7 ou F8 até o momento em que o erro ocorre.

O Delphi permite definir uma condição para a parada do *breakpoint* dentro do código onde, quando tal condição for verdadeira, o aplicativo pára na linha em questão. Veja o código de exemplo da **Listagem 7**.

Listagem 7. Exemplo para uso do breakpoint com condição

```

procedure TForm1.btnExecutarClick(
  Sender: TObject);
var
  x, y: Integer;
begin
  x := 1;
  y := 1;
  while x < 100 do
  begin
    y := y + x;
    inc(x);
  end;
  Edit1.Text := IntToStr(y);
end;

```

Vamos adicionar um *breakpoint* na linha $y := y + x$. Digamos que um erro está ocorrendo quando o valor de y é igual a 16. Nesse caso, deveríamos ir executando a aplicação utilizando a tecla F8 (*Step Over*), correto? Bem, nesses casos podemos definir uma condição do tipo ($y = 16$), onde somente quando a mesma for verdadeira o *breakpoint* para na linha de código.

Para isso, selecione a opção *View > Debug Windows > Breakpoints* para abrir a janela *Breakpoint List*. Selecione o *breakpoint* em questão e acesse a opção *Properties* disponível no menu de contexto. Dentro da janela *Source Breakpoint Properties* adicione a condição “ $y = 16$ ”, dentro do campo *Condition* (**Figura 4**).

Ao rodar a aplicação o *breakpoint* só irá parar na linha de código quando a condição for verdadeira, evitando assim a depuração desgastante através da tecla F8 dentro do loop.

14. Trabalhando com packages em runtime

Caso sua aplicação esteja ficando com o tamanho do arquivo executável muito grande, você pode diminuir utilizando a opção de *packages* em runtime. Quando criamos uma aplicação Delphi, por menor que ela seja, ao gerarmos o arquivo executável, todos os pacotes os quais contêm os componentes que utilizamos na aplicação são embutidos dentro do arquivo.

Podemos distribuir esses pacotes separadamente da aplicação, fazendo com que dessa maneira nosso arquivo executável fique menor. Entre na janela *Options* do projeto (*Project > Options*) e dentro da aba *Packages* marque a opção *Build with runtime packages*.

Rode a aplicação. O arquivo executável gerado será bem



Figura 4. Definindo uma condição para a parada do breakpoint

menor, entretanto, os pacotes dos componentes que foram utilizados na aplicação terão que ser distribuídos junto com o executável para que a aplicação funcione.

Para saber quais pacotes (arquivos *bpl*) devem ser levados junto com o executável da aplicação acesse a opção *Project > Information for NomeProjeto* (**Figura 5**).

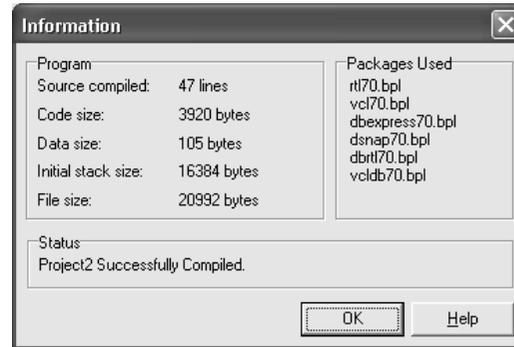


Figura 5. Verificando a lista dos packages utilizados pela aplicação

15. Enviando e-mail com anexo utilizando componentes Indy

Nessa última dica, veremos com enviar e-mails com arquivos em anexo utilizando os componentes *Indy*. Para enviar e-mails a partir de uma aplicação Delphi adicione um *IdSMTP* (*Indy Clients*) e um *IdMessage* (*Indy Misc*). Adicione ainda um botão e no seu evento *OnClick* digite o código da **Listagem 8**.

Listagem 8. Enviando e-mail com anexo

```

procedure TForm1.btnEnviarClick(Sender: TObject);
begin
  IdMessage1.From.Address :=
    'everson@rhealeza.com.br';
  IdMessage1.Recipients.EmailAddresses :=
    'joao.silva@gmail.com';
  IdMessage1.Subject := 'Assunto - Enviando e-mails ' +
    'com anexo utilizando Indy';
  IdMessage1.Body.add('Corpo do email');
  IdMessage1.ContentType := 'Text';
  TIdAttachment.Create(IdMessage1.MessageParts,
    'c:\temp\Arquivo.txt');
  IdSMTP1.host := 'seuhost';

  // caso o servidor use autenticação:
  IdSMTP1.Password := 'senha';
  IdSMTP1.Username := 'usuario@dominio';
  IdSMTP1.AuthenticationType := atLogin;

  IdSMTP1.Connect;
  try
    IdSMTP1.send(IdMessage1);
  finally
    IdSMTP1.Disconnect;
  end;
end;

```

Conclusão

Vimos neste artigo várias dicas que podem ser utilizadas no dia a dia do desenvolvedor Delphi. Algumas dessas dicas podem ser consideradas básicas para alguns, porém, a muitos novos desenvolvedores começando a trabalhar com Delphi, onde tenho certeza que elas poderão ajudá-los durante o desenvolvimento e aprendizado. Um abraço e até a próxima. ■