Delphi e Lógica de Programação para Iniciantes

por: Diego Garcia
contato: drgarcia1986@gmail.com
site: http://drgarcia1986.wordpress.com

Sumário

Introdução ao Delphi

O Que é o Delphi

Uma forma simples de definir o **Delphi** seria dizer que o Delphi é uma ferramenta RAD (em português **Desenvolvimento Rápido de Aplicações**) para desenvolvimento desktop em **Object Pascal** (ramificação da linguagem de programação **Pascal**, grosseiramente definida como *Pascal Orientado a Objetos*). Começaremos este estudo conhecendo um pouco da IDE (em português **Ambiente Integrado de Desenvolvimento**) do Delphi e criando nosso primeiro *Hello World.* Entre as muitas versões do **Delphi**, iremos seguir nosso estudo utilizando a versão **Turbo Delphi** por ser uma versão gratuita.

Criando um novo Projeto

Para o nosso primeiro *Hello World* iremos criar um formulário simples com um botão que quando for clicado apresentará a mensagem Hello World em um caixa de mensagem, porem antes, iremos nos habituar melhor a IDE do **Turbo Delphi.** Vamos começar inserindo um novo projeto em:

File -> New -> VCL Forms Application – Delphi For Win32,

isso fará com que um novo projeto seja criado e um formulário vazio apareça no centro da tela. Desta forma acabamos de iniciar um projeto de um aplicação comum, poderíamos iniciar desta mesma forma um projeto de uma nova dll, uma unit com um conjunto de funções, um pacote de um projeto maior (bpl), etc. Esta área onde o formulário foi criado é chamada de **Design**, entraremos em detalhes em breve, vamos nos atentar agora as outras áreas da desktop do Delphi.

IDE / Desktop

Pelo fato do Delphi ser uma ferramente RAD para desenvolver aplicações visuais, sua IDE foi projetada para facilitar o acesso a todas propriedades e eventos do projeto em desenvolvimento. O IDE é composto por várias partes principais, onde temos:

Design

Esta é a área onde será feita o design da interface gráfica do projeto. É aqui onde é desenhados os formulários, onde são incluídos componentes como botões, caixas de texto, etc. Está área se encontra no centro da tela.

Code

Está é a área onde será feita toda a codificação dos projetos criados. Toda inteligencia da aplicação se encontra em sua codificação. Está área se encontra junto com a área de **Design** podendo alternar entre uma o outro através de uma aba na parte inferior ou atrás da tecla **F12**.

Structure

Está é a área de estrutura do formulário. Todos componentes criados em um determinado formulário serão apresentados de forma hierárquica nesta área, sendo que, caso estejamos na área

Code a estrutura apresentada será a do código do formulário, apresentando todos métodos e variáveis criadas. Está área se encontra no canto superior esquerdo.

Object Inspector

Está é a área de propriedades e eventos de um componente selecionado na área de **Design**, é nesta área onde serão definidas todas as características de um componente do projeto, desde o nome até o evento disparado no momento em que o componente receber um clique do mouse por exemplo. Está área se encontra no canto inferior esquerdo.

Project Manager

Esta é a área de gerenciamento do projeto, todos os formulários, units, datamodules, etc, que forem criadas para o projeto corrente, estarão disponíveis nesta área, sendo possível criar um grupo de projeto e fazer o gerenciamento de mais de um projeto por vez. Está área se encontra no canto superior direito.

Tool Palette

Está é a área onde se encontrão os componentes disponíveis para o desenvolvimento de uma aplicação. Os componentes disponíveis podem mudar de acordo com o tipo de unit ou a areá de visualização atual (code ou design). Esses componentes podem ser visuais ou não, como por exemplo um botão sendo um componente visual e uma query sendo um componente não visual. Está área se encontra no canto inferior direito.

Olá Mundo!

Agora que já nos habituamos a IDE do Delphi, vamos desenvolver nosso primeiro *Hello World*, para isso vamos inserir em qualquer parte do nosso formulário um componente **TButton** que consiste em um botão comum e vamos clicar duas vezes neste componente, isto fará com que alterne da areá Design para a areá Code com o método **Button1Click** previamente implementado (entraremos em detalhes posteriormente). Agora digite a seguinte linha de código:

```
ShowMessage('Olá Mundo!');
```

Este código fara com que, quando o botão que inserimos no formulário for clicado, a aplicação ira executar o método *ShowMessage* que consiste em mostrar uma caixa de mensagem simples para o usuário.

Salve este projeto em *C:\Curso\HelloWorld* (caso você não possua esse diretório, você pode criá-lo) e clique em *Run -> Run* no menu principal (ou simplesmente pressione a tecla **F9**), isto fara com que o Delphi compile o projeto, gerando seu arquivo binário e já o execute.

Logica e Pascal

A partir de agora, iremos estudar a parte mais importante deste curso, a parte de lógica de programação e fundamentos da linguagem pascal. A grande maioria de desenvolvedores de software espalhados pelo mundo concordam que basta ter a lógica bem assimilada para que seja possível desenvolver softwares em qualquer linguagem, isto por que, a solução do problema sempre será a mesma, o que muda é sintaxe usada para chegar a essa solução.

Premissas

Antes de começar a nos aprofundar no fantástico mundo do pascal, vamos nos atentar a algumas regras da linguagem.

Fim de comando

Sempre no final de cada comando é necessário utilizar o caractere **;** (ponto e virgula)

Variavel := 1 +2;

Atribuindo valores a variáveis

Assim como foi demonstrado no exemplo anterior, para se atribuir o valor a uma variável, basta usar os caracteres **:=** (dois pontos e igual) .

Variavel := Valor;

Comentários

Uma boa pratica de programação é o uso dos comentários para identificar certos trechos de código que podem passar a ser complexos para entender depois de um determinado tempo ou caso outra pessoa tenha a necessidade de dar manutenção neste código. As duas maneiras mais comuns de comentar um código é com o comentário simples de uma linha e o bloco de comentários.

```
//este é um comentário simples

{Este é um bloco de comentários
possibilitando colocar mais de uma linha
no comentário}
```

Variáveis / Constantes

De uma maneira simples, variáveis e constantes são regiões de memória onde damos um nome e armazenamos valores de determinados tipos, com a diferença que as variáveis podem ter seus valor alterado durante a execução de um aplicativo, enquanto que as constantes permanecem sempre com o mesmo valor com qual foram declaradas.

Em um primeiro momento o uso de constantes pode parecer pouco útil, porem, imagine uma aplicação que realize diversos cálculos com uma taxa de impostos fixa, não conhecendo o uso das constantes, o comportamento esperado seria que, para cada calculo feito, fosse digitado o valor desta taxa, porem, imagine que um dia essa taxa possa mudar, teríamos que vascular o código inteiro atrás de ocorrências da taxa antiga, correndo o risco de deixar algum calculo passar.

Definição de escopo

Quando dizemos *escopo de uma variável* estamos nos referindo ao alcance desta variável, ou até onde ela está acessível. Em Object Pascal (assim como na maioria das outras linguagens de programação) uma variável só pode ser acessada dentro do seu escopo. Se criarmos uma variável dentro de uma função especifica, o escopo desta variável limita-se ao corpo desta função, ou seja, esta variável só poderá ser acessada dentro desta função.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MinhaVariavel : String;
begin
  MinhaVariavel := 'Conteudo da Variavel';
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  MinhaVariavel := 'Atualizando a Variavel';
end;
```

O uso da variável 'MinhaVariavel' no método 'FormCreate' não é permitido

Na codificação do formulário existe a área **Private** e a área **Public**, as variáveis, contantes, funções, etc, que forem declaradas no área **private** estarão acessíveis somente no formulário atual, enquanto que as declaradas em **public** estarão visíveis para todos os formulários que utilizarem o formulário onde a variável foi declarada. Saber em qual escopo uma variável se enquadra no projeto, ajuda a fazer um melhor uso da memória da maquina.

Tipos mais comuns

Uma lista com os tipos mais comuns de dados também conhecidos como tipos primitivos que usaremos no decorrer do nosso estudo

- Integer – Numero inteiro (de -2147483648 a 2147483647)
- Double – Ponto flutuante com 64 bits (utilizado para números grandes e com casas decimais)
- Boolean – Valor Booleano (true ou false)
- String – Cadeia de caracteres (utilizado para textos)
- Date – Referencia para data ou hora.

Declarando variáveis

Assim como já explicado anteriormente, as variáveis são espaços de memoria que armazenam valores que podem ser alterados no decorrer da aplicação. Para declarar uma variável primeiro precisamos saber qual escopo ela se mostra necessária e qual é o seu tipo.

A declaração de uma variável sempre deve vir logo após as palavras reservadas **VAR**, **PRIVATE** ou **PUBLIC** sendo a primeira a mais comum, com a sintaxe:

NomeDaVariavel : Tipo;

Uma variável pode ser de qualquer tipo reconhecido pelo Delphi, inclusive tipos criados pelo próprio desenvolvedor. Uma importante observação é que devido ao fato do Object Pascal ser uma linguagem fortemente tipada, não é possível declarar uma variável de um determinado tipo e no decorrer da aplicação atribuir a ela um valor de um tipo diferente do seu tipo declarado.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  UmNumeroQualquer : Integer;
begin
  UmNumeroQualquer := 0;
  UmNumeroQualquer := 'Um Texto';
end;
```

A segunda linha deste código ocasionara uma erro de compilação.

Outra regra com relação as variáveis diz respeito ao nome atribuído a variável na sua declaração, não sendo permitido o uso de palavras reservadas pelo delphi e nomes iniciados com números.

Declarando constantes

A declaração de constantes é muito semelhante a declaração de variáveis, com a principal diferença que, na declaração da constante não é obrigatório especificar o tipo da constante e no momento da declaração já deve ser atribuído um valor. A sintaxe mais comum para a declaração de uma constante é a seguinte:

NomeDaConstante = Valor

Apesar das regras de escopo se aplicarem também as constantes, a declaração de uma constante deve ser feita logo após a palavra reservado **CONST**, mesmo o escopo da constante sendo global (areá Public do formulário).

```
procedure TForm1.Button1Click(Sender: TObject);
const
  ValorDePi = 3.14159265;
begin

end;
```

Assim como as variáveis, o nome de uma constante não pode ser uma palavra reservada do delphi e nem pode começar com números.

Declarando Arrays

Array (também conhecido como vetor) nada mais é do que uma estrutura de memoria utilizada para armazenar vários dados de um mesmo tipo. É possível utilizar um array para armazenar uma lista de valores como por exemplo uma lista de alunos. Um array comum possui um tamanho fixo e o tipo de dado que será armazenado em suas posições, essas definições são feitas no momento da declaração do array. Um array é tratado como uma variável pelo Delphi seguindo as mesmas regras de uma variável comum. A sintaxe para a declaração de um array segue o seguinte padrão:

NomeDoArray : Array[posiçãoInicial .. posiçãoFinal] of Tipo;

Ou seja, podemos definir uma vetor de *strings* (maiores detalhes mais a frente) com 5 posições, para atribuir o nome de 5 pessoas diferentes, isto ficaria da seguinte forma:

```
procedure TForm1.Button1Click(Sender: TObject);
Var
  Pessoas : Array[0..4] of String;
begin
  Pessoas[0] := 'Carlos';
  Pessoas[1] := 'João';
  Pessoas[2] := 'Maria';
  Pessoas[3] := 'Sonia';
  Pessoas[4] := 'Carmen';
end;
```

Note que começamos nosso array na posição 0 e finalizamos na posição 4, fazendo com que nosso array contivesse 5 posições

Analisando o exemplo, fica simples de entender que o array armazena uma estrutura de variáveis tendo cada uma seu índice correspondente dentro do array.

Operadores Lógicos

Operadores lógicos são utilizados para definir valores *booleanos* (0 ou 1, true ou false) que são usados principalmente em estruturas de decisão, sendo praticamente impossível a construção de uma aplicação sem o uso desses elementos. Os principais e mais comuns operadores lógicos são estes:

- Igual (=)
- Diferente (<>)
- E (and)
- Ou (or)
- Negação (not)

Para ajudar na compreensão dos operadores lógicos, vamos aprender a utilizar estruturas de decisão para que possamos demonstrar as diversas maneiras de se utilizar operadores lógicos.

Estrutura de decisões

Em determinadas aplicações é necessário que o aplicativo seja capaz de tomar diferentes caminhos de acordo com as condições que apareçam no decorrer de sua execução. Para que isso seja possível existem as estrutura de decisões, tendo como principal representante o **IF/ELSE**.

IF/ELSE

O IF/Else é a maneira mais comum de tomar uma decisão no fluxo de uma aplicação através de um teste lógico. Fazendo uma abstração a vida real, o uso do IF seria algo como:

Se isso for verdadeiro Faça
Esta Operação
Caso contrario
Esta Outra

Esta é a maneira mais simples e correta de entender o uso de um IF. Voltando ao Pascal, a sintaxe básico do IF é a seguinte:

IF <condição> Then
...
Else
...

Sendo que o uso do ELSE não é obrigatório e o ELSE pode ser combinado com outro IF, exemplo:

IF <condição> Then
...
Else If <condição> Then
...

Utilizaremos o IF para entendermos melhor como trabalhar com operadores lógicos, lembrando que o IF testa condições boleanas, sendo assim, podemos utilizar o IF para validar variáveis booleanas da mesma forma que utilizamos o IF para validar operações lógicas.

Igual (=)

O operador lógico de igualdade é utilizado para validar se o valor de duas variáveis é igual, independente do tipo da variável desde que as duas variáveis sejam do mesmo tipo, veja neste exemplo quais são os resultados possíveis para uma comparação de igualdade:

1	=	1	True (Verdadeiro)
1	=	2	False (Falso)

Utilizando variáveis, constantes e estrutura de decisões, podemos demonstrar o uso deste operador lógico da seguinte maneira.

```
procedure TForm1.Button1Click(Sender: TObject);
Const
  NUMERO_ESPERADO = 50;
Var
  numeroSugerido : Integer;
begin
  numeroSugerido := 50;

  if numeroSugerido = NUMERO_ESPERADO then
    ShowMessage('Sua sugestão está correta')
  else
    ShowMessage('Este não é o numero esperado');
end;
```

Seguindo o fluxo do exemplo acima, ao clicar no botão do formulário seria apresentada a mensagem *Sua Sugestão está correta*, mas caso o valor da variável *numeroSugerido* ou da constante *NUMERO_ESPERADO* fosse alterado fazendo com que tivessem valores diferentes, a aplicação apresentaria a mensagem *Este não é o numero esperado*.

Diferente (<>)

O operador de diferença funciona exatamente como o operador de igualdade, porem, executando a operação inversa. Se na igualdade validamos se o valor de duas variáveis são iguais, na diferença validamos se o valor da duas variáveis são diferentes, fazendo com que o resultado desta validação seja *verdadeiro* para variáveis distintas e *falso* para variáveis semelhantes, sendo assim, os possíveis resultados são os seguintes:

1	<>	1	False (Falso)
1	<>	2	True (Verdadeiro)

Para melhor ilustrar esta questão, usaremos o exemplo a baixo:

```
procedure TForm1.Button1Click(Sender: TObject);
Const
  NOME_COMUM = 'João';
Var
  nomePessoa : String;
begin
  nomePessoa := 'João';

  if nomePessoa <> NOME_COMUM then
    ShowMessage('Este não é um nome comum')
  else
    ShowMessage('Este nome é comum');
end;
```

A execução do exemplo acima, faria com que ao clicar no botão do formulário, fosse apresentada a mensagem *Este nome é comum* pois a primeira validação do IF iria retornar *False* devido ao fato do valor da variável *nomePessoa* e da constante *NOME_COMUM* serem iguais e como nossa decisão esta sendo tomada a partir de uma diferença, o fluxo da aplicação seguiria para o *Else*.

Maior (>) / Maior ou igual (>=)

É possível verificar se um valor é maior do que outro e neste mesmo processo é possível também verificar se os valores são iguais, isto através do sinal de *maior* (>) e o *maior ou igual* (>=).

1	>	0	True (Verdadeiro)
1	>	1	False (Falso)
1	>	2	False (Falso)
1	>=	0	True (Verdadeiro)
1	>=	1	True (Verdadeiro)
1	>=	2	False (Falso)

Menor (<) / Menor ou igual (<=)

Ao contrario do m*aior / maior ou igual*, o *menor* (<) e o *menor ou igual* (<=) verifica se um valor é menor do que outro e verifica também se é menor ou igual.

1	<	0	False (Falso)
1	<	1	False (Falso)
1	<	2	True (Verdadeiro)
1	<=	0	False (Falso)
1	<=	1	True (Verdadeiro)
1	<=	2	True (Verdadeiro)

Ou (or)

Em determinadas situações é necessário verificar mais de uma condição em um fluxo da aplicação, onde, caso apenas uma esteja de acordo (verdadeira) já é o suficiente para executar uma determinada sequência de código. Isso se torna possível através do operador lógico **OU** (or) que verifica se pelo menos uma das condições verificadas é verdadeira.

1 > 0	or	'A' = 'B'	True (Verdadeiro)
1 < 2	or	'C' <> 'D'	True (Verdadeiro)
1 = 3	or	10 >= 15	False (Falso)

Utilizando o *Ou* a operação sempre resultara em *verdadeiro* caso pelo menos uma das comparações resulte em *verdadeiro*. Veja uma exemplo do uso do *ou*

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MAX_FALTAS = 20;
  NOTA_MEDIA = 6;
var
  FaltasAluno : Integer;
  MediaAluno  : Double;
begin
  FaltasAluno := 25;
  MediaAluno  := 7;

  if (MediaAluno < NOTA_MEDIA) or (FaltasAluno > MAX_FALTAS) then
    ShowMessage('Aluno reprovado');

end;
```

obs: Em uma comparação **OR** os termos sempre são analisados da esquerda para direita, portanto, caso algum termo resulte em um valor **true** os outros termos serão ignorados.

E (and)

Diferente do *ou* o *E* (*and*) só resulta em verdadeiro caso todas as condições resultem em verdadeiro.

1 > 0	and	'A' = 'B'	False (Falso)
1 < 2	and	'C' <> 'D'	True (Verdadeiro)
1 = 3	and	10 >= 15	False (Falso)

Usando o mesmo tema exemplo dado para o *ou*, vamos mudar um pouco para podermos demonstrar o uso do *E*.

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MAX_FALTAS = 20;
  NOTA_MEDIA = 6;
var
  FaltasAluno : Integer;
  MediaAluno  : Double;
begin
  FaltasAluno := 25;
  MediaAluno  := 7;

  if (MediaAluno >= NOTA_MEDIA) and (FaltasAluno <= MAX_FALTAS) then
    ShowMessage('Aluno Aprovado')
  else
    ShowMessage('Aluno Reprovado');

end;
```

Analisando esse exemplo, fica claro que a aplicação apresentara a mensagem '*Aluno Reprovado*', devido ao fato do valor da variavel *FaltasAluno* maior do que o valor da constante *MAX_FALTAS*, porem poderíamos mudar o valor da variável *FaltasAluno* para 20, desta forma a aplicação iria apresentar a mensagem *'Aluno Aprovado'*, já que as duas validações estariam testadas no IF seriam verdadeira.

Obs: Assim como no **OR** o **AND** compara os termos da esquerda para direta, sendo assim, caso algum termo resulte em **false** os outros termos não serão analisados.

Negação (not)

Podemos utilizar o operador lógico de *negação* (*not*) para negar alguma condição booleana, a grosso modo, o uso do *not* faz com que se 'inverta' o resultado de uma comparação booleana.

not True	False (Falso)
not False	True (Verdadeiro)
not (1=1)	False (Falso)
not (2 > 3)	True (Verdadeiro)

Vamos mudar um pouco nosso exemplo do aluno, para podermos demonstrar o uso do *not*:

```
procedure TForm1.Button1Click(Sender: TObject);
const
  MAX_FALTAS = 20;
  NOTA_MEDIA = 6;
var
  FaltasAluno   : Integer;
  MediaAluno    : Double;
  AlunoAprovado : boolean;
begin
  FaltasAluno := 25;
  MediaAluno  := 7;

  if (MediaAluno >= NOTA_MEDIA) and (FaltasAluno <= MAX_FALTAS) then
    AlunoAprovado := true
  else
    AlunoAprovado := false;;

  if not AlunoAprovado then
    ShowMessage('Aluno reprovado')
  else
    ShowMessage('Aluno aprovado');

end;
```

Operadores Matemáticos Básicos

Assim como em qualquer linguagem o Pascal também trabalha com operações matemáticos. Limitaremos nosso estudo as 4 operações básicas e o resto da divisão:

Soma	+
Subtração	-
Divisão	/
Multiplicação	*
Resto	mod

Precedência

A precedência das operações matemáticas no pascal segue a mesma precedência da matemática:

Primeiro	*, /, mod
Segundo	+, -

Porem, podemos forçar a execução de uma operação matemática de menor procedência antes de uma de maior utilizando parenteses:

*10 * 5 + 3*
*Primeiro seria executada a multiplicação 10 * 5 resultando em 50 e este valor seria somando ao 3 resultando em 53*

*10 * (5 + 3)*
Desta forma, primeiro seria executado a soma 5 + 3 resultando em 8 e esse valor seria multiplicado ao 10 resultando em 80.

Exercícios

Para os exercícios a seguir, o uso de variáveis é livre, independente da quantidade e do tipo de variáveis que forem utilizadas:

1. Utilizando a ideia do aluno sendo aprovado ou não, faça uma rotina para que sejam somadas 3 notas, tire a média delas e verifique se o aluno foi aprovado, reprovado ou se pode fazer exame para recuperar nota. Utilize uma constante para armazenar a média de aprovação do curso e a media para exame. Mostre para o usuário o resultado e depois altere o valor das variáveis para chegar em diferentes resultados.
2. Calcule quanto um funcionário recebe em um ano (ignorando 13º salário e férias), para realizar esse calculo crie uma constante para armazenar o salário mensal bruto do funcionário e a porcentagem de tributação paga mensalmente. Mostre para o usuário o resultado e depois altere o valor das constantes para chegar em diferentes resultados.

Estruturas de laço (loop)

Utilizamos *laços* quando queremos executar um bloco de instruções um numero determinado de vezes, mesmo esse numero sendo conhecido ou não.

While

O laço *while* é executado enquanto uma determinada condição for verdadeira:

Enquanto for verdadeiro faça
Esta operação

O *while* é utilizado geralmente quando não se sabe o numero de interações do laço. Para exemplificar o uso do while, vamos fazer uma contagem regressiva começando em 10 e terminado em 0.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NumeroAtual : Integer;
begin
  NumeroAtual := 10;

  while NumeroAtual >= 0 do
  begin
    ShowMessage(IntToStr(NumeroAtual));
    NumeroAtual := NumeroAtual -1;
  end;

end;
```

For

Diferente do laço *while* o laço *for* é utilizado quando se sabe a quantidade de interações, pois na sintaxe da sua implementação é especificado a quantidade de interações.

Para contador valendo x até y faça
Esta operação

Diferente do *while* o *for* é executado até que a condição seja verdadeira. O *for* é um laço largamente utilizando em operações com arrays por utilizar um contador que vai incrementando automaticamente a cada iteração.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Nomes : Array[0..4] of String;
  I : Integer;
begin
  Nomes[0] := 'João';
  Nomes[1] := 'Maria';
  Nomes[2] := 'Carlos';
  Nomes[3] := 'Julia';
  Nomes[4] := 'Ronaldo';

  for I := 0 to 4 do
  begin
    ShowMessage(nomes[i]);
  end;
end;
```

Note que foi necessário declarar a variável I para utiliza la como contador.

No exemplo acima especificamos na implementação do laço que a variável *I* começaria valendo 0 (zero) e o laço iria ser executado até que o variável *I* estivesse valendo 4 (quatro). O resultado seria cinco mensagens na tela uma com cada nome atribuído no array de nomes seguindo a ordem de atribuição.

Seria possível fazer com que no exemplo acima a aplicação apresentasse os nomes na ordem contraria a de atribuição? Começando pelo índice 4 e terminando no índice 0 ? sim é possível, ao alterar a implementação do laço *for* trocando o *TO* por *DOWNTO* o contador passa a ser decrementado ou invés de ser incrementado.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Nomes : Array[0..4] of String;
  I : Integer;
begin
  Nomes[0] := 'João';
  Nomes[1] := 'Maria';
  Nomes[2] := 'Carlos';
  Nomes[3] := 'Julia';
  Nomes[4] := 'Ronaldo';

  for I := 4 downto 0 do
  begin
    ShowMessage(nomes[i]);
  end;
end;
```

Repeat

Até agora estudamos laços que primeiro checam uma condição para depois realizar sua primeira iteração, no laço *repeat* esta regra muda, primeiro a aplicação vai realizar um iteração no laço para depois checar se a condição permite a execução da próxima iteração. Assim como o *for* o *repeat* é executado até que a condição seja verdadeira.

Repita
estes comandos
até esta condição

Mesmo que inicialmente a condição de parada do laço seja verdadeira antes de sua primeira iteração, os comandos são executados pelo menos uma vez, diferente dos laços anteriores. Para exemplificar o uso laço *repeat* vamos fazer uma contagem de 0 (zero) até um determinado numero.

```
procedure TForm1.Button1Click(Sender: TObject);
const
  NUMERO_FINAL = 10;
var
  PararLaco   : Boolean;
  NumeroAtual : Integer;
begin
  NumeroAtual := -1;

  repeat
    NumeroAtual := NumeroAtual + 1;
    ShowMessage(IntToStr(NumeroAtual));
  until NumeroAtual = NUMERO_FINAL;
end;
```

Break / Continue

Caso seja necessário sair de um laço interrompendo sua execução, basta usar o comando b*reak*, caso a intenção seja apenas pular desta iteração para a próxima, voltando para o começo do laço basta utilizar o comando *continue*

Exercícios

1. Faço uma contagem regressiva utilizando o laço *for*.
2. Faça uma contagem utilizando o laço *while* começando em 0 (zero) até achar 3 números pares.
3. Faça uma contagem nos 3 tipos de laços começando em 0 (zero) e terminando em 10 (dez) mostrando na tela mensagens somente com os números impares.
4. Faça uma contagem nos 3 tipos de laços começando em 0 (zero) e terminando em 10 (dez), porem, force a saída do laço após apresentar o numero 5.

Funções básicas

O pascal já disponibiliza algumas funções já implementadas, funções essas que são comuns em qualquer linguagem.

Funções com strings

Função	Aplicação	Sintaxe
Length	Recuperar o tamanho de uma string ou de um Array	VarInteira := Length(varTexto);
Copy	Copia parte de uma string	StrFinal := copy(varTexto, posiçãoInicial, quantosCaracteres);
Pos	Recupera a posição de um determinado caractere (ou conjunto) dentro de uma string	VarInterira := Pos('A',varTexto);
Trim	Elimina espaços a direita e a esquerda de uma string (utilize TrimLeft para eliminar somente a esquerda e TrimRight somente a direita)	StrFinal := Trim(varTexto);
StringReplace	Substitui parte de uma string por outra	StrFinal := StringReplace(varTexto, caracterASubstituir, novoCaracter, outrasOpcoes);

Funções com Números

Função	Aplicação	Sintaxe
Inc	Incrementa o valor de uma variavel do tipo inteiro	Inc(varInteiro);
Random	Retorna um numero inteiro aleatório.	VarInteira := Random(range); *obs: utilizar sempre a função Randomize antes de usar a random.*

Funções com Data e Hora

Função	Aplicação	Sintaxe
Date	Retorna a data atual	VarData := Date;
Now	Retornar a data e hora atual	VarDataTempo := now;
IncDay	Incrementa dias a uma data	VarData := incDay(varData, dias); *obs: caso não informe a quantidade de dia, a função ira incrementar somente 1*
DaysBetween	Retorna numero de dias entre uma data e outra	VarInteira := DaysBetween(varData, varProximaData);

Funções de conversão de tipo (cast)

Função	Aplicação	Sintaxe
IntToStr	Converte um integer para uma string	VarStr := IntToStr(varInteira);
FloatToStr	Converte um double para uma string	VarStr := FloatToStr(varDouble);
StrToInt	Converte uma string para um Integer	VarInteira := StrToInt(varStr);
StrToFloat	Converte uma string para um double	VarDouble := StrToFloat(varStr);
DateToStr	Converte um Date para uma String	VarStr := DateToStr(varData);
DateTimeToStr	Converte um DateTime (data e hora) para uma string	VarStr := DateTimeToStr(varDataEHora);
TimeToStr	Converte um Time para uma string	VarStr := TimeToStr(varHora);
StrToDate	Converte uma String em um Date	VarData := StrToDate(varStr);
FloatToStrF	Converte um Double para uma String formatada	VarStr := FloatToStrF(varDouble,formato, precisao, casas decimais);

Exercícios

1. Crie uma constante com uma palavra qualquer e utilizando laços e funções do pascal apresente uma mensagem com essa palavra escrita ao contrario.
2. Crie uma constante com o texto '133,25' e substitua a virgula por um ponto (**.**) em uma variável.
    1. faça usando pos e copy
    2. faça usando o StringReplace
3. Crie uma constante com um nome qualquer e mostre uma mensagem dizendo a quantidade de caracteres que aquele nome possui.
4. Armazenar uma numero aleatório (de 0 a 4) em uma variável, incrementar esse numero a data atual, apresentar uma mensagem com a data incrementada, gerar outro numero aleatório, incrementar na data que já foi incrementada e mostrar uma mensagem com a quantidade de dias entre a data atual e data incrementada.
5. Refazer o exercício do salário anual do funcionário, apresentando o resultado final formatado em reais.

Componentes

Uma das vantagens de se utilizar uma ferramente como o Delphi é a facilidade em manipular componentes visuais, tanto suas propriedades como seus eventos, este fato faz com que muitas vezes confundam o Delphi como uma linguagem orientada a eventos, sendo que o Delphi como já dito anteriormente é baseado em Object Pascal, que é uma linguagem orienta a objetos.

Propriedades de componentes

As propriedades de um componente são a parte mais importante do componente, são elas que determinam a forma como o componente vai ser apresentado, quais informações são armazenadas no componente, tamanho, cor, etc. Existem duas formas de manipular as propriedades de um componente, em tempo de design e em tempo de execução, sendo a primeira no momento do design do projeto utilizando a *Object Inspector* e a segunda totalmente dinâmica, de acordo com o fluxo de execução da aplicação, exemplo:

componente.propriedade := valor;

A maioria das propriedades são comuns entre os componentes facilitando a assimilação de novos componentes. Esta lista contem algumas das propriedades mais comuns.

Propriedade	Propósito
Text	Texto digitado no componente, por exemplo um texto digitado em uma caixa de texto.
Caption	Texto de título, por exemplo o texto de um botão
Name	O nome do componente.
Width	Largura do componente
Height	Altura do componente
Enable	Se o componente está ou não desabilitado
Visible	Se o componente está ou não visível.

Eventos dos componentes

Os eventos dos componentes geralmente ditam a forma como a aplicação vai se comportar. É possível executar rotinas em diversos eventos diferentes de qualquer componente, ao enviar foco, ao tirar foco, ao pressionar uma tecla, ao clicar, ao passar com o mouse encima, etc. Para verificar quais eventos um determinado componente disponibiliza e implementar rotinas para estes eventos, basta selecionar o componente e na *Object Inspector* mudar para a aba de eventos.

Componentes Básicos

Componente	Utilização
TEdit	Caixa de texto
TLabel	Texto fixo
TButton	Botão comum
TBitBtn	Botão que permite imagem
TMaskEdit	Caixa de texto com mascara, utilizado geralmente para campos de telefone, data, etc.
TMenuItem	Utilizado para fazer Menu em formulário
TGroupBox	Caixa para agrupamento de componentes, funciona também como um contender
TRadioButton	Opção de seleção, utilizado para seleções únicas, por exemplo Sexo: Masculino / Feminino.
TCheckBox	Opção de checagem, utilizado para múltiplas seleções.

TComboBox	Caixa de seleção única baseado em texto, utilizado geralmente quando as opções não são totalmente definidas.
TImage	Para apresentar imagem.
TListView	Listagem, pode ser utilizado para varias finalidades possibilitando o uso de colunas, imagens, etc.
TMemo	Caixa de texto para textos longos que permitam quebra de linha.
TOpenDialog	Caixa de dialogo padrão do windows para abertura de arquivos
TSaveDialog	Caixa de dialog padrão do windows para gravação de arquivo.

Exercícios

1. Refazer o exercício do salário anual, fazendo uma interface com caixas de texto para o salario mensal, um combo para armazenar as taxas e deixar selecionável, um label para apresentar o resultado final e colocar uma caixa de checagem para determinar se devera calcular o a taxa ou não. Incluir uma caixa de texto para informar o valor de isenção do imposto de renda e um label para informar se o funcionário precisa declarar imposto de renda ou não.
2. Criar uma interface semelhante a um cadastro contendo as seguintes informações: Nome, Data de Nascimento, Sexo, Cidade (em um combo com as seguintes opções São Paulo, Campinas, Piracicaba), se possui dependentes, telefone e ao final através de uma rotina programada no clique de um botão, colocar todas as informações em um memo, separando o nome do sobrenome e calculando a idade através da data de nascimento. Validar na saída do campo de data de nascimento se a data digitada é uma data valida.
3. Salvar as informações do exercício anterior em um arquivo de texto utilizando o componente SaveDialog para escolher o diretório e utilizar a função SaveToFile do componente memo para salvar o arquivo.

Banco de Dados

DataModule

Uma boa prática de desenvolvimento quando se esta trabalhando com banco de dados é o uso de *DataModules*, que consiste em uma especie de formulário não visual para armazenar componentes de acesso a dados, centralizando esses componentes e tornando mais fácil o acesso através dos outros formulários. Para adicionar um *datamodule* ao projeto atual, basta clicar com o botão direito encima do projeto no *Project Manager* e depois ir em *Add New -> Other -> DataModule*.

BDE

Uma das tecnologias que o Delphi trabalha nativamente para acesso ao banco de dados é a tecnologia BDE, que permite o acesso a vários bancos de dados diferentes, tendo como principal o paradox e os arquivos dbf.

TTable

O *ttable* é o componente utilizado para abertura de uma tabela que seja possível ser aberta via tecnologia **bde**. A maioria dos métodos que veremos para o componente *ttable* estão disponíveis para outros componentes de acesso a banco de dados, obtendo os mesmos resultados.

Abrindo e fechando um tabela

Para abrir uma tabela e carregar seu conteúdo na memória, é necessário primeiramente preencher algumas propriedades do componente *ttable* como por exemplo:

Propriedade	**Finalidade**
DatabaseName	Alias (bde ou odbc) ou diretório onde se encontra a tabela
TableName	Nome da tabela
Exclusive	Método de abertura da tabela, exclusivo (true) ou compartilhado (false), por padrão esta opção vem setado como false
Active	Determina se uma tabela vai estar aberta ou não. Setando como active = true em tempo de design é possível ver os registros da tabela ainda em tempo de design.

Obs: Qualquer uma dessas propriedades pode ser alterada em tempo de execução

Após preencher as principais propriedades do componente basta executar o método *open* do componente, com a seguinte sintaxe:

tabela.open;

Caso não ocorra nenhum erro na abertura da tabela a aplicação continuara com o fluxo normal. Para fechar a tabela basta utilizar o método *Close*.

tabela.close;

Lembrando que, utilizando acesso via BDE, cada tabela aberta consome parte da memória da BDE correndo o risco de ocasionar falhas com um numero grande de tabelas abertas.

Inserindo Registros

Para inserir registros em uma tabela pelos componentes de tabela do delphi, basta utilizar o método *Append*, fazendo com que insira uma nova linha em branco na tabela, sendo assim, após o *append* é necessário preencher os valores paras os campos da tabela. Após preencher as informações dos campos, basta utilizar o método *post* para efetivar as informações no banco de dados, ou o método *cancel* para cancelar as alterações e remover a nova linha inserida.

```
Tabela.Append;
Tabela.FieldByName('Nome').value := nome;
Tabela.FieldByName('Fone').value := telefone;
Tabela.post;
```

Os campos que não forem preenchidos neste processo serão gravados como nulo no banco.

Editando Registros

Para editar um registro ativo em um componente *ttable* basta utilizar o método *Edit*, fazendo com que o registro entre em modo de edição. Após deixar o registro em modo de edição o método de gravação é o mesmo do *append*, *post* para efetivar as alterações e *cancel* para cancelar a edição.

```
Tabela.edit;
Tabela.FieldByName('Nome').value := nome;
Tabela.FieldByName('Fone').value := telefone;
Tabela.post;
```

Apagando registros

Para apagar o registro ativo em um componente *ttable* basta utilizar o método *Delete*.

```
Tabela.delete;
```

Navegação entre registros

Até agora vimos como abrir e fechar uma tabela, inserir novos registros, editar e deletar o registro ativo, porem ainda não vimos como navegar entre os registros gravados na tabela, para isso existe um conjunto de métodos específicos para isso. A navegação entre registros se dá através de um ponteiro, ou seja, o registro ativo é na verdade o registro em que esta o ponteiro da tabela.

Método	**Finalidade**
First	Move o ponteiro para o primeiro registro
Last	Move o ponteiro para o ultimo registro
Next	Move o ponteiro para o próximo ponteiro
Prior	Move o ponteiro para o registro anterior

Para saber se um o ponteiro esta no ultimo registro, basta usar o método *EOF*, quer retorna *true* se o ponteiro já está no ultimo registro.

Procurando registros

Uma das coisas mais comuns é buscar um registro em uma tabela e o componente *ttable* disponibiliza algumas maneiras de realizar esta busca.

Locate

A forma mais comum e mais simples de se procurar um registro em uma tabela é uso do método *locate*. O método *locate* é uma busca sem uso de índices e por qualquer campo da tabela. Retornar *true* se encontra o registro e move o ponteiro para a primeira ocorrência desse registro. A Sintaxe do método é a seguinte:

tabela.locate(campo,valor,opcoes);

Um exemplo pratico do uso do *locate* para procurar um registro uma tabela.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Nome : String;
begin
  Nome := Edit1.Text;
  if Table1.Locate('nome_pessoa',Nome,[]) then
    ShowMessage(Nome + ' já cadastrado');
end;
```

FindKey

O *findkey* é um método de busca através do uso de índices, sendo assim é necessário que a tabela esteja indexada pelo campo que se deseja fazer a busca. Para alterar o índice ativo de uma tabela existem duas propriedades, o *indexName* que deve ser preenchida com o nome do índice e a propriedade *indexFieldName* que deve ser preenchida com o nome do campo a se indexar. Lembrando que, em uma banco de dados paradox, para indexar um campo, este deve possuir um índice previamente criado. A sintaxe do uso do *findKey* é a seguinte:

tabela.findkey(valor);

Um exemplo prático do uso do método *findkey* indexando uma tabela por um campo determinado.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  cpf : String;
begin
  cpf := Edit1.Text;

  Table1.IndexFieldNames := 'cpf_pessoa';
  Table1.Refresh;

  if Table1.FindKey([cpf]) then
    ShowMessage('Pessoa encontrada');
end;
```

Assim como o *locate* o *findKey* retorna *true* se acha o registro e move o ponteiro para o registro encontrado.

FindNeareast

Assim como o *findkey* o *FindNeareast* é uma pesquisa basada no índice ativo da tabela, com uma diferença, enquanto o *findkey* procura exatamente o termo passado como parâmetro, o *findNeareast* busca por parte do termo, simulando um *like* de sql.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Nome : String;
begin
  Nome := Edit1.Text;
  Table1.FindNearest([Nome]);
end;
```

Outra diferença entre os dois métodos é que o *findnearest* não tem retorno, ele simplesmente move o ponteiro para o registro encontrado.

TQuery

Até agora aprendemos a abrir uma tabela e manipular seus registros, porem, algumas vezes é necessário juntar informações de varias tabelas em uma única para conseguir chegar em um determinado resultado, algo que uma simples query já resolveria. Para essa finalidade existe o componente *TQuery* que funciona muito semelhante ao componente *ttable* mas com algumas particularidades especificas para o uso de instruções SQL. Para tornar nosso aprendizado mais dinâmico, vamos centrar as *particularidades* deste componente, aproveitando o que já conhecemos de navegação do componente *ttable*.

Carregando / Executando uma query

Existem duas formas de executar uma query com o componente *tquery*, sendo cada uma especifica para uma finalidade, mas antes de entrar neste detalhes, primeiro precisamos saber como especificar qual query será executada, para isso existe a propriedade *sql* do componente *tquery*, propriedade esta que contem alguns métodos como por exemplo o método *add*, que será o que usaremos para informar nossa query ao componente.

```
var
  ComandoSql : String;
begin
  ComandoSql := 'SELECT * FROM PESSOA';
  Query1.SQL.Clear; //LIMPANDO O VALOR DA PROPRIEDADE
  Query1.SQL.Add(ComandoSql);
```

Sabendo como "informar" nossa query ao componente, precisamos saber agora qual será a finalidade da query, se for somente uma query de consulta (select por exemplo) será necessário um ponteiro no resultado para que possamos manipular o registro, para isso usaremos o método OPEN do componente *tquery*, caso seja uma query de manipulação de dados (insert, delete, update, ou mesmo comandos de DDL), um ponteiro não se mostra necessário, sendo assim usaremos o método *Exec* do componente tquery.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ComandoSql : String;
begin
  ComandoSql := 'SELECT * FROM PESSOA';
  Query1.SQL.Clear; //LIMPANDO O VALOR DA PROPRIEDADE
  Query1.SQL.Add(ComandoSql);
  Query1.Open;
end;
```

Após abrir a query, a navegação dos dados segue o mesmo padrão do componente ttable.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ComandoSql : String;
begin
  ComandoSql := 'INSERT INTO PESSOA VALUES (';
  ComandoSql := ComandoSql + QuotedStr(Edit1.Text) +')';

  Query1.SQL.Clear;
  Query1.SQL.Add(ComandoSql);
  Query1.ExecSQL;
end;
```

Exercícios

1. Criar uma tabela paradox com o nome de pessoa, com o campo id (autoincremento) e o campo nome (varchar 35), criar uma aplicação para inserir dados nesta tabela (via componente *ttable*) e criar uma rotina para preencher um memo com todos as pessoas cadastradas nesta tabela.
2. Repetir o exercícios anterior usando o componente *tquery* no lugar do componente *ttable*

Componentes DataControl

Componentes datacontrol são componentes criados para facilitar a integração entre componentes visuais e componentes de acesso a dados.

TDataSource

O componente TD*ataSource* serve como ponte entre os *dataSets* (ttable, tquery, etc.) e os componentes DataControl. Portanto, para cada *dataset* que será associado a um componente *dataControl* é necessário criar o seu componente *tdataSource* correspondente. A propriedade que especifica a qual *dataset* este *TdataSource* se refere-se é a propriedade *DataSet*, basta preencher essa propriedade com o nome do *dataset.* Feito isto, já podemos começar a usar os componentes *datacontrol* associados a um tabela.

TDBGrid

O componente *tdbGrid* é um componente que apresenta os dados de uma tabela em um formato parecido com uma planilha, permitindo ou não edição direta dos dados. Com o *TdbGrid* podemos especificar quais campos serão mostrados, cor da linha, fonte, etc. Para integramos um *tdbgrid* a uma tabela, basta *no object Inspector* preencher a propriedade *dataSource* do *dbgrid* com o nome do *datasource* que está apontando para a tabela que será exibida no *dbgrid* e depois especificarmos pela propriedade *columns* os campos que serão apresentados.

TDBEdit

O componente *TDBEdit* se assemelha a um componente *tedit* com a diferença que este componente pode estar ligado diretamente a um campo de um *dataset.* Assim como o componente *TDBGrid*, para ligar um um *TDBEdit* a um campo de um *dataset,* basta preencher sua propriedade *datasource* com o nome do *datasource* desejado e a propriedade *DataField* com o nome do campo a qual esse *tdbedit* irá se referenciar. Ao editar o conteúdo deste componente, automaticamente o conteúdo do campo está sendo editado, deixando o registro ativo em modo de edição.

TDBText

O componente *TDBText* se assemelha a um componente *tlabel*, e funciona exatamente igual ao componente *TDBEdit* com a diferença que não proporciona uma forma de editar o conteúdo da informação, se limitando a ser um componente somente para visualização.

Exercício

1. Criar um cadastro simples de pessoa com os campos Id (autoincremento) nome e e-mail (varchar 35). Criar a interface com dbgrid, dbedit e dbtext. Usar botões para editar, inserir ou apagar um registro.

Navegação

TabOrder

O que determina a ordem de foco dos campos quando se pressiona a tecla tab em um formulário é a propriedade *tabOrder* dos componentes, basta ordenar os componentes na ordem desejada de foco. Lembrando que cada contêiner do formulário possui sua ordem individual.

VirtualKeys

Algo muito comum é determinar ações especificas em componentes de acordo com uma determinada tecla que for pressionada, existem duas formas de se controlar isto, uma delas (talvez a mais fácil) é através do uso de *virtualKey,* que nada mais é do que constantes do Delphi para auxiliar neste propósito. Basta comparar a tecla pressionada com a *virtualKey* desejada. Algo importante a se notar neste momento é que o uso de virtualkey só se torna possível no evento *onKeyDown* dos componentes. Um exemplo desta técnica:

```
procedure TForm1.DBGrid1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if key = VK_F4 then
  Begin
    //aqui vão rotinas de exclusão de registro
  End;
end;
```

note que a variável Key (que representa a tecla pressionada) é na verdade um parâmetro da função onKeyDown

No exemplo a cima, ao se pressionar a tecla *F4*, alguma rotina poderá ser executada.

Código ASCII na navegação

Outra forma de verificar a tecla pressionada e condicionar a navegação a isto é o uso da verificação do código ascii da tecla. Funciona da mesma forma que os *virtualKeys* porem, ao invés de se comparar com uma *virtualKey* se compara ao código ascii de uma tecla qualquer e o evento mais comum a se usar esta verificação é o evento *onKeyPress*.

```
procedure TForm1.DBEdit1KeyPress(Sender: TObject; var Key: Char);
begin
  if key = #13 then
    Button1.SetFocus;
end;
```

Esta rotina verifica se a tecla pressionada é tecla enter (#13 da tabela ascii), caso seja, envia o foco do formulário para outro componente.

Exercício

1. Refaça o exercício do cadastro simples, melhorando a navegação com as técnicas passadas neste tópico.

Formulários

Os formulários no Delphi são na verdade objetos que devem ser criados na memória antes do uso e consequentemente limpados da memória quando seu uso passa a não ser mais necessário. Para inserir um novo formulário ao projeto atual, basta fazer na mesma forma que se adiciona um *DataModule*, através do *Project Manager.* O Delphi consegue gerencial quais formulários serão criados no momento em que a aplicação será executada ou os que será criados dinamicamente, por padrão, sempre que inserimos um novo formulário ao projeto, este será setado para ser criado automaticamente. Para manipular este gerenciamento bastar ir no menu:

Project -> Options -> Forms

A lista da esquerda determina os formulários que serão criados automaticamente quando a aplicação for carregada e a lista da direita são os fomulários disponíveis no projeto. Note que ainda existe a indicação *Main Form* com um combo contendo o nome dos formulários do projeto, o

formulário que for identificado como *Main Form* passa a ser o formulário principal da aplicação, isso significa que será o primeiro a ser criado e que se por algum motivo for descarregado a aplicação será encerrada.

Show e ShowModal

Existem duas forma de abrir um formulário, o método *show* e o método *ShowModal.* A principal diferença entre os dois métodos é que, com o *show* o formulário é aberto mas continua a ser possível manipular o formulário anterior, enquanto que, com o *ShowModal* os formulários anteriores passam a ficar desabilitados, permitindo a manipulação somente do ultimo formulário aberto.

Criando formulários em tempo de execução

Assim como dito anteriormente, quando um formulário não for setado para ser criado junto com a aplicação é necessário cria-lo na memória e limpa-lo quando este deixar de ser necessário. Existem inúmeras formas de se fazer isso, portanto, pode acontecer casos em que você se depare com outras forma e questione se esta é a mais correta do que a outra, a resposta é, depende, mas não entraremos nesses detalhes. Vamos seguir o seguinte exemplo para criar um formulário em tempo de execução:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  //verifico se o formulário ainda não esta criado
  if Form2 = nil then
    Form2 := TForm2.Create(Application);

  Form2.ShowModal; //abro o formulário

  FreeAndNil(Form2); //limpo o formulário da memória
end;
```

Exercícios

1. Refaça o exercício do cadastro simples adicionando o campo cidade (inteiro) na tabela de pessoa e criando uma tabela de cidades com id (autoinc) e nome (varchar 35), no cadastro adicione um campo para inserir o código da cidade e crie um formulário para pesquisar as cidades cadastradas.