Loops e 'Arrays' em VBA

Autor desconhecido -Tradução de Eduardo Machado (Good Guy)

Loops proporcionam a repetição de tarefas em um programa. Quando você está executando uma tarefa em um programa e se vê fazendo a mesma coisa repetidas vezes, você deve pensar sobre como um 'loop' ou uma sub-rotina poderia ser aplicado para facilitar a sua vida. Ao usar um 'loop' poderá poupar linhas e mais linhas de programação se a tarefa ou operação tiver que se repetir (ou mesmo se tiver uma grande probabilidade de se repetir).

Por que Precisamos de Loops

Vamos olhar para um exemplo de uma tarefa repetitiva onde os 'loops' economizariam linhas extensas de código.

```
strResposta = InputBox("Deseja Continuar? (S/N)")

' checa por S or N

If strResposta = "S" then Call Continue

If strResposta = "N" then Exit Sub

' S ou N não foi encontrado então continue perguntando ao usuário

strResposta = InputBox("Deseja Continuar? (S/N)")

If strResposta = "S" then Call Continue

If strResposta = "N" then Exit Sub

'... e assim por diante ...
```

Loops *Do ...Loop*

Para economizar linhas e mais linhas de programação acima, coloque o 'InputBox()' e as declarações 'IF' dentro de um loop.

```
Do

    strResposta = InputBox("deseja Continuar? (S/N)")

    If strResposta = "S" then Call Continue

    If strResposta = "N" then Exit Sub

Loop Until strResposta = "S" or strResposta = "N"
```

A vantagem do loop **Do ... Loop Until** parece muito clara neste exemplo. Este não é sempre o caso de situações de programação, contudo, assim mesmo usaremos exemplos para visualizar o lado prático dos 'loops'. Até o fim deste artigo, você perceberá de forma organizada as escolhas de sintaxe para se programar com loops. Como se pode observar acima, os exemplos fornecem o máximo de informação sobre o assunto. Uma pequena seção com referência baseada no Ajuda do Access também está disponível.

O **Do ...Loop** acima teve a lógica ao fundo do loop e, esta estrutura a forçou a desempenhar o mínimo programação de uma só vez. Quando a lógica está posicionada no topo do 'loop', no entanto, aí então o 'loop' pode funcionar com potencial no todo. No exemplo seguinte, um 'loop' pode correr tantas vezes quantas for necessário e, só executará se o usuário falhar em seguir as instruções. Isto ilustra como os 'loops' podem funcionar com declarações 'IF"

```
strResposta = InputBox("Deseja Continuar? (S/N)")

Do While (strResposta <> "S" And strResposta <> "N")

    strResposta = InputBox("Responda S ou  N") ' executada somente se o usuário não
digitar S ou N

Loop

'A declaração 'IF' se segue depois do 'loop'

If strResposta = "S" then Call Continue

If strResposta = "N" then Exit Sub
```

A sintaxe 'Do Loop' suporta a lógica da posição no início dos loops(**Do While... Loop**) ao invés de no final dos loops(**Do... Until Loop**). A lógica da posição no início ou no final de um loop determina se seu programa deve ou não deve executar o código durante o 'loop' pelo menos uma vez.

Uma outra sintaxe, chamada 'loop' **While...Wend**, posiciona a lógica no início do loop. Na verdade, não é diferente do Do While, portanto somente uma rápida menção se faz aqui e um exemplo se segue de forma resumida.

Os loops 'Do Loops' e 'While Wend' nem sempre são a melhor escolha de uma sintaxe de loop, porque o loop não vem com um contador embutido, você tem que programá-lo por ser necessário para o 'loop'. Isto funciona, é claro, mas requer duas linhas desnecessárias de código.

```
intCounter = 0

While intCounter < 10

    intCounter = intCounter + 1

    StrBanner = strBanner & "** Feliz Milênio **"

Wend
```

Loops For ...Next

Suponha que você precise de uma variável que consiste de ter "** Feliz Milênio **" repetindo-se 10 vezes em um *banner*. Veja como o 'loop' **For...Next** pode suprir esta necessidade.

```
For intCounter = 1 to 10

      StrBanner = strBanner & "** Feliz Milênio **"

Next
```

Os loops 'Do Loop' e 'For Next' ambos funcionam bem nestes exemplos de programação. Desde que ambos sejam loops, no entanto, **poderíamos usar um For no lugar de um DO e vice-versa?** As vantagens e desvantagens das duas estruturas devem se tornar bem evidentes nos exemplos abaixo.

Usando 'For' quando 'Do' é apropriado. Agora mesmo você pode achar que tanto 'For' quanto 'Do' esteja OK, mas se você já achou a constante 999 em seu programa dois meses depois que você a escreveu, provavelmente pareceria meio confuso. Neste caso, 'Do' é preferível, nisso é mais que auto-documentável (por exemplo, a condição para satisfazer o 'loop' e continuar o programa ficará mais claro.)

```
For intCounter = 1 to 999

    strResposta = InputBox("Deseja Continuar? (S/N)")

    If strResposta = "S" then Call Continue

    If strResposta = "N" then Exit Sub

Next
```

Loops *For Each ...Next*

Um outro format de loop é o **For Each... Next.** O Access 2000 fornece para ambas tanto a não-indexada quanto indexada(ou numerada) referências às suas coleções de objeto. Formulário, tabelas, consultas, botões, rótulos, caixas de texto, caixas de listagens e assim por diante são objetos nas coleções do Access.

Ao utilizar 'loops', você pode referenciar os objetos sem saber seus nomes de objetos específicos. Isto seria muito prático, por exemplo, se você quisesse fazer todos os rótulos e caixas de texto em sua aplicação aparecer com a fonte **Tahoma**, tamanho 12. Ajustar estes valores em todas as caixas de propriedades manualmente é trabalhoso com prováveis erros e acertos, mas um 'loop' não perderia um simples objeto.

Vamos agora criar um procedimento para forçar todos os rótulos e caixas de texto para exibir com fonte **Tahoma** de tamanho 12. Neste exemplo se aplica o loop **For Each...Next** para pesquisar tanto os rótulos quanto caixas de texto e alterar suas propriedades no formulário aberto. O exemplo abaixo ilustra um outro tipo de 'loop' que não tem nenhum contador.

```
    Dim ctl As Control

    For Each ctl In Me.Controls

        If TypeOf ctl Is Label Or TypeOf ctl Is TextBox Then

            ctl.FontName = "Tahoma"

            ctl.FontSize = 12

        End If

    Next ctl
```

O código acima pode ser colocado no procedimento de evento 'Form Load' para cada formulário. Pode ser usado exatamente como está, ou pode ser colocado em um procedimento público em um módulo padrão e depois evocado por uma linha de código no evento 'Form Load', desta maneira:

Call Tahoma(Me)

No último caso, você muda o **Me.Controls** para **frm.Controls** e dá a Sub Tahoma12 um argumento(frm as Form). Quase sem exceção, é uma prática de programação mais aceita colocar o código que pertencerá a mais de um formulário em um procedimento geral que aparece dentro de um módulo padrão.

'Arrays' e Loops

Se você não está familiarizado com **'arrays'**(matrizes), pode querer achar que um 'array' não é nada mais que uma lista. Todos nós temos listas em nossas vidas, tais como listas de shopping, lista de tarefas, listas de aniversários, listas de preços de itens do mercado e assim por diante. Todos os itens de um **'array'**, bem como os itens de uma lista, tem uma posição em algum lugar entre o primeiro e o último. Itens são numerados do mais baixo para o mais alto em **'arrays',** que torna o loop **For Next** particularmente útil ao programar **'arrays'.**

O seguinte exemplo calcularia o preço médio semanal para um estoque, baseado em um 'array' contendo o preço de estoque a cada cinco dias.

```
For intCounter = 1 to 5

    PrecoMedio = PrecoMedio + Price(intCounter)

Next

PrecoMedio = PrecoMedio / 5
```

Suponha que você quer que o nome de cada dia da semana apareça em um 'array'. A função seguinte cria um 'array' de nomes em uma Variant e então fornece os nomes para o usuário. Variants podem tanto ser do tipo Integer, String e assim por diante ou 'arrays' desses.

```
Public Function DiadaSemana(intDia As Integer) As String

    Dim DiaSemana As Variant

    DiaSemana = array("Dom","Seg","Ter","Qua","Qui","Sex","Sab")

    DiadaSemana = DiaSemana(intDia)

End Function
```

Um outro exemplo prático envolve um 'array' de datas para um negócio. No programa seguinte, um 'array' de datas foi desenvolvido em uma Variant e formatado e, uma data é então informada ao usuário.

```
Public Function DiaFinalNegocio(intFinal As Integer) As Date

    Dim DiaFinal As Variant

    DiaFinal = array(Format("31/03/12", "Short Date"), _

                        Format("30/06/12", "Short Date"), _

                        Format("30/09/12", "Short Date"), _

                        Format("31/12/12", "Short Date"))

    DiaFinalNegocio = DiaFinal(intFinal)

End Function
```

Como Utilizar 'Arrays' em Procedimentos

Um outro uso muito importante para 'arrays' é como argumentos em funções e sub-rotinas chamadas, como mostrado no exemplo seguinte. O propósito aqui é permitir que você use o procedimento com um número desconhecido de **elementos de 'array'** (De 0 até N) na lista de argumento.

Aqui está uma função que devolve a posição de um número do menor valor numérico em um 'array' de valores. Dois benefícios dignos de nota de 'arrays' neste caso são: (1) que o 'array' pode ser de qualquer comprimento; e (2) que todos os itens no 'array' podem ser referenciados por suas posições e valores. Aqui está o código que permite que você ache a menor nota em uma lista(ou 'array') das notas dos estudantes.

```
Public Function AMenor(Notas() As Single) As Integer

'encontre a posição da nota mais baixa em um 'array' de Notas

Dim intPosition As Integer, intPosicaoInferior As Integer

Dim sglMaisBaixa As Single

sglMaisBaixa = 9999999.9

For intPosition = LBound(Notas) To UBound(Notas)

    If sglMaisBaixa > Notas(intPosition) Then

        sglMaisBaixa = Notas(intPosition)

        intPosicaoInferior = intPosition

    End If

Next

AMenor = intPosicaoInferior

End Function

'Em seguida, um exemplo de como incluir 'arrays' em chamadas

Public Sub TestAMenor()

    Dim Notas(0 To 2) As Single

    Notas(0) = 95#

    Notas(1) = 75#

    Notas(2) = 50#

    Debug.Print AMenor(Notas); Notas(AMenor(Notas))

' Resultado: 2 50 representam a posição e o valor do AMenor nota

End Sub
```

No loop **For Next**, duas novas funções embutidas são apresentadas. São as L- e U-Bound que devolvem os limites inferiores e superiores do 'array'. Por causa de se estar usando estas funções, *o programa não tem que saber o tamanho exato do 'array' de Notas.* Mais acerca destas funções a seguir.

LBound and UBound: Seus Bons Amigos

As funções são extremamente úteis com 'array's. Você notou no exemplo acima que a *Função Amenor, poderia ser usada para qualquer número de Notas em qualquer curso?* Assim, o programa terá uma validade maior e, será mais fácil de manutenção. Aqui está mais sobre LBound e UBound para 'array's unidimensionais.

```
'LBound('array') devolve o início ou o índice Amenor do 'array' unidimensional

'UBound('array') devolve o final ou o índice maior do 'array' unidimensional.


            For intPosition = LBound(Notas) To UBound(Notas)
```

Option Base

Na seção de Declarações do código VBA, você pode incluir a Option Base 1. Isto instruiria o Access a sempre iniciar a numerar seus 'arrays' com um limite inferior igual a 1. Se você não adicionar esta declaração, o Access inicia seus 'arrays' em 0. Mesmo se você omitir a declaração, contudo, você pode substituir o zero ao usar a sintaxe:

Option Base 1

Dim Notas(1 to 3)

Option Base pode ser confuso porque algumas das funções Access embutidas usam a declaração, enquanto outras funções a ignoram completamente e sempre começam em zero. A função 'array' e a função Dim usam a Base, mas a função ParamArray ignora a base e sempre usa 0.

Estudantes às vezes ficam confusos com a posição da numeração e a **numeração dos elementos nos 'arrays'**. **Lembre-se de que Strings NUNCA começam a numeração da posição 0.** Na verdade, o 0(zero) causaria um erro.

Um Bom Conselho: Submeta Option base 0. Se você escolher usar **Base 1** você terá que saber lidar quando e onde a base 0 defaults(padrões) ocorrem. Se você escolher usar Base 0 sempre haverá menores complicações.

'Arrays' Multi-dimensionais

Nem todos os 'arrays' que você usa tem de ser de uma única dimensão. Como usuário Access, você já está familiarizado com folhas de dados que exibem dados em tabelas e consultas. As folhas de dados contém linhas e colunas que representam campos e registros, respectivamente.

Um 'array' multi-dimensional [**Dim NomeEstado(0 To 2, 0 To 49)**] pode conter o nome do estado, uma abreviação com 2(duas) letras do nome (RJ) e a taxa de vendas. Note que são três campos, mas a dimensão são de dois valores por causa da Option Base 0. Poderia haver 50 registros para os 50 estados. Se você tivesse este 'array' seu melhor uso poderia ser fornecer pesquisas de taxas de vendas muito rápidas.

Assim, como você consegue os dados embutidos em um 'array' multidimensional? O exemplo abaixo ilustra o método manual e exige a digitação e revisão cuidadosa, porque os dados foram incluídos na lógica (Veja a Sub TestAmenor() acima).

Na prática, conservamos os dados em tabelas e a lógica nos programas. Quando você mistura os dois você normalmente tem algumas péssimas conseqüências logo depois resultando em uma manutenção do programa. Por exemplo, se as taxas de vendas do estado estivessem em uma tabela, então você não teria que mudar o programa sempre que as referidas taxas mudassem.

O Access fornece um meio de converter os dados de uma tabela em 'arrays' bidimensionais. Ao usar a programação de recordset, você pode ter todos os registros ou alguns registros colocados dentro de um 'array'. Por exemplo, você pode obter os próximos 50 registros de uma tabela dentro da DadosEstado(2,49) desta maneira:

```
'3 colunas e 50 linhas dentro do DadosEstado(2,49)

Dim rs as DAO.Recordset

Set rs = CurrentDB.Openrecordset( _

         'SELECT Nome, SiglaEstado, Taxa FROM TaxaTblVendas', dbOpenDynaset)

DadosEstado = rs.GetRows(50)

rs.Close

Set rs = Nothing
```

LBound, UBound e 'Arrays' Multi-dimensionais

Estas funções servem tanto como 'arrays' multi-dimensionais bem como 'arrays' unidimensionais. Para aprender os limites inferiores e superiores das colunas do 'array', digite assim **LBound(DadosEstado,1) e UBound(DadosEstado,1).** O 1 significa a primeira dimensão(campos ou colunas). Similarmente, use **LBound(DadosEstado,2) e UBound(DadosEstado,2)** para a segunda dimensão (colunas ou registros)

```
intNumberFields = UBound(DadosEstado,1) + 1 ' número de campos/colunas


intNumeroColunas = UBound(DadosEstado,2) + 1 ' número de registros/colunas


intIndicePrimeiraCol = LBound(DadosEstado,1) ' campos iniciando o índice for-loop


intIndiceUltimaCol =  UBound(DadosEstado,1) ' campos finalizando o índice for-loop


intIndicePrimeiraLinha = LBound(DadosEstado,2) ' registros iniciando índice for-
loop

intIndiceUltimaLinha =  UBound(DadosEstado,2) ' registros finalizando o índice for-
loop
```

‘Array’ de Elementos e Dimensões

Estamos acostumados a ver folhas de dados do Access. Elas são análogas aos ‘arrays’ bidimensionais com colunas(campos) e linhas(registros). Uma folha de dados com uma simples coluna(campo) é análoga a um ‘array’ unidimensional. Um cubo é análogo a um ‘array’ tridimensional. ‘Arrays’ podem ter 1,2,3,... até 6 dimensões.

Às vezes os estudantes se tornam confusos acerca do número de dimensões e os números de elementos por dimensão. Os elementos se referem ao número de colunas ou linhas ou fatias de um ‘array’. Por exemplo, o ‘array’ bidimensional de taxas de vendas possue três campos(3 elementos de coluna) e 50 registros(50 elementos de linha).

Em nossos exemplos, temos declarado este ‘array’ com a seguinte sintaxe:

Dim DadosEstado(0 to 2, 0 to 49).

É comum para os programadores se referirem ao ‘array’ de taxas como um ‘array’ “3 por 50” para comunicarem sucintamente o número de dimensões e elementos por dimensão.

Material de Referência do Sistema de Ajuda do Access

Declaração For...Next

```
For contador = inicio To fim [Step step]
    [declarações]
    [Exit For]
    [declarações]
Next [contador]
```

Parte	Sintaxe da declaração do loop For ... Next
Contador	Exigido. Variável numérica usada como um contador de loop. A variável não pode ser um elemento boleano ou um elemento de um ‘array’.
Início	Required. Initial value of counter. Exigido. Valor inicial do Contador.
Fim	Exigido. Valor final do Contador.
Step	Opcional. O Contador de quantidade muda cada vez que se completa um loop. Se não especificado, o Step padrão fica em 1. O argumento Step tanto pode ser positivo quanto negativo. O valor do argumento Step determina o processamento do loop como se segue: (1) O loop executa se o Contador <= Fim e o Valor do Step é positivo

Parte	Sintaxe da declaração do loop For ... Next
	ou 0; (2) O loop executa se o Valor do Step for negativo e o Contador >= Fim;
declarações	Optional. One or more declarações between For and Next that are executed the specified number of times. Opcional. Um ou mais declarações entre For e Next que são executadas o número especificado de vezes.

Declaração Do...Loop

```
Do [{While | Until} condição]

    [declarações]

    [Exit Do]

    [declarações]

Loop


' Or, you can use this syntax:

Do

    [declarações]

    [Exit Do]

    [declarações]

Loop [{While | Until} condição]
```

Parte	Sintaxe da declaração do loop Do ... Loop
condição	Opcional. Expressão numérica ou expressão de string que é True(Verdadeira) ou False(Falsa). Se a condição for Null(Nula), a condição é tratada como False(Falsa).
declarações	Uma ou mais declarações que se repetem enquanto, ou até que uma condição seja True(Verdadeira).

Declaração While...Wend

```
While condição

    [declarações]

Wend
```

Parte	Sintaxe da declaração do loop While ... Wend
condição	Exigido. Expressão numérica ou expressão de string que avalia para True(Verdadeira) ou False(Falsa). Se a condição for Null(Nula), a condição é tratada como False(Falsa)
declarações	Opcional. Uma ou mais declarações executadas enquanto a condição for True (Verdadeira)

Declaração For Each...Next

```
For Each element In grupo

    [declarações]

    [Exit For]

    [declarações]

Next [elemento]
```

Parte	Sintaxe da declaração do loop For ... Each
elemento	Exigido. Variável usada para interagir através dos elementos da coleção ou 'array'. Para coleções, o elemento somente pode ser uma variável do tipo Variant, uma variável de objeto genérica, ou qualquer variável de objeto específico. Para 'arrays', o elemento só pode ser uma variável do tipo Variant.
grupo	Exigido. Nome de uma coleção de objetos ou 'array' (exceto um 'array' de tipos definidos pelo usuário).
declarações	Opcional. Um ou mais declarações que são executadas em cada item no grupo.