



**TOTVS**



***ADVPL 1***

<b>OBJETIVOS DO CURSO .....</b>	<b>6</b>
<b>MÓDULO 01: INTRODUÇÃO À PROGRAMAÇÃO .....</b>	<b>7</b>
<b>1. LÓGICA DE PROGRAMAÇÃO E ALGORITMOS .....</b>	<b>7</b>
1.1. Lógica de Programação .....	7
1.2. Desenvolvendo algoritmos .....	9
1.2.1. Estudando algoritmos .....	10
1.2.2. Teste de mesa.....	13
<b>2. ESTRUTURAS DE PROGRAMAÇÃO.....</b>	<b>15</b>
2.1. Diagrama de bloco .....	15
2.2. Estruturas de decisão e repetição.....	20
2.2.1. Estruturas de decisão .....	20
2.2.2. Estruturas de repetição .....	25
<b>MÓDULO 02: A LINGUAGEM ADVPL .....</b>	<b>29</b>
<b>3. ESTRUTURA DE UM PROGRAMA ADVPL.....</b>	<b>32</b>
3.1. Áreas de um Programa ADVPL.....	36
<b>4. DECLARAÇÃO E ATRIBUIÇÃO DE VARIÁVEIS .....</b>	<b>42</b>
4.1. Tipo de Dados .....	42
4.2. Declaração de variáveis .....	44
4.3. Escopo de variáveis.....	46
4.4. Entendendo a influência do escopo das variáveis.....	55
4.5. Operações com Variáveis .....	57
4.5.1. Atribuição de variáveis .....	57
4.5.2. Operadores da linguagem ADVPL.....	59
4.5.3. Operação de Macro Substituição .....	69
4.5.4. Funções de manipulação de variáveis.....	71
<b>5. ESTRUTURAS BÁSICAS DE PROGRAMAÇÃO .....</b>	<b>80</b>
5.1. Estruturas de repetição .....	80
5.1.1. Influenciando o fluxo de repetição.....	85
5.2. Estruturas de decisão .....	87

<b>6. ARRAYS E BLOCOS DE CÓDIGO .....</b>	<b>96</b>
6.1. Arrays .....	96
6.1.1. Inicializando arrays .....	100
6.1.2. Funções de manipulação de arrays .....	103
6.1.3. Cópia de arrays .....	106
6.2. Listas de Expressões e Blocos de Código .....	110
6.2.1. Premissas para utilização de Blocos de Código .....	110
6.2.2. Lista de expressões.....	112
6.2.3. Blocos de Código.....	116
6.2.4. Funções para manipulação de blocos de código.....	119
<b>7. FUNÇÕES.....</b>	<b>121</b>
7.1. Tipos e escopos de funções .....	123
7.2. Passagem de parâmetros entre funções.....	128
<b>8. DIRETIVAS DE COMPILAÇÃO .....</b>	<b>138</b>
<b>MÓDULO 03: DESENVOLVENDO PEQUENAS CUSTOMIZAÇÕES .....</b>	<b>151</b>
<b>9. ADVPL E O ERP MICROSIGA PROTHEUS .....</b>	<b>151</b>
9.1. O Ambiente Protheus .....	152
9.2. Organização e configuração inicial do ambiente Protheus .....	157
9.3. O Configurador do Protheus.....	166
9.3.1. Funcionalidades Abordadas .....	166
9.3.2. Estruturas básicas da aplicação ERP Protheus .....	166
9.3.3. Acessando o módulo Configurador.....	174
9.4. Funcionalidades do Configurador .....	176
9.4.1. Dicionário de Dados da aplicação ERP .....	178
9.4.2. Adição de tabelas ao Dicionário de Dados .....	179
9.4.3. Adição de campos às tabelas do Dicionário de Dados .....	183
9.4.4. Adição de índices para as tabelas do Dicionário de Dados .....	190
9.4.5. Adição de gatilhos para os campos das tabelas do Sistema .....	195
9.4.6. Criação de Tabelas Genéricas .....	199
9.4.7. Criação de Parâmetros.....	201
<b>10. TOTVS DEVELOPMENT STUDIO .....</b>	<b>203</b>
DESENVOLVIMENTO DE PEQUENAS CUSTOMIZAÇÕES .....	207
<b>11. ACESSO E MANIPULAÇÃO DE BASES DE DADOS EM ADVPL .....</b>	<b>207</b>
11.1. Diferenças e compatibilizações entre bases de dados .....	209
11.2. Funções de acesso e manipulação de dados .....	211

11.3.	Diferenciação entre variáveis e os nomes de campos.....	219
11.4.	Controle de numeração sequencial.....	221
<b>12.</b>	<b>CUSTOMIZAÇÕES PARA A APLICAÇÃO ERP .....</b>	<b>223</b>
12.1.	Customização de campos – Dicionário de Dados.....	223
12.1.1.	Validações de campos e perguntas.....	224
12.1.2.	Pictures de formação disponíveis.....	227
12.2.	Customização de gatilhos – Configurador.....	231
12.3.	Customização de parâmetros – Configurador .....	232
12.3.1.	Funções para a manipulação de parâmetros .....	233
12.3.2.	Cuidados na utilização de um parâmetro.....	235
12.4.	Pontos de Entrada – Conceitos, Premissas e Regras .....	236
<b>13.</b>	<b>INTERFACES VISUAIS .....</b>	<b>238</b>
13.1.	Sintaxe e componentes das interfaces visuais .....	238
13.2.	Interfaces padrões para atualizações de dados .....	242
13.2.1.	AxCadastro() .....	242
13.2.2.	MBrowse().....	244
13.2.3.	AxFunctions().....	250
<b>APÊNDICES .....</b>	<b>254</b>	
<b>BOAS PRÁTICAS DE PROGRAMAÇÃO .....</b>	<b>254</b>	
<b>14. UTILIZAÇÃO DE IDENTIFICAÇÃO .....</b>	<b>254</b>	
<b>15. CAPITULAÇÃO DE PALAVRAS-CHAVE.....</b>	<b>256</b>	
15.1.	Palavras em maiúsculo.....	257
<b>16. UTILIZAÇÃO DA NOTAÇÃO HÚNGARA.....</b>	<b>257</b>	
<b>17. PALAVRAS RESERVADAS .....</b>	<b>259</b>	
<b>GUIA DE REFERÊNCIA RÁPIDA: Funções e Comandos ADVPL .....</b>	<b>261</b>	
Conversão entre tipos de dados .....	261	
Verificação de tipos de variáveis.....	267	
Manipulação de arrays.....	270	
Manipulação de blocos de código .....	282	
Manipulação de strings .....	287	

Manipulação de variáveis numéricas.....	294
Manipulação de arquivos.....	299
Controle de numeração sequencial.....	319
Validação.....	320
Parâmetros .....	323
Componentes da interface visual .....	327
Interfaces de cadastro .....	333
Funções visuais para aplicações .....	340
Funções ADVPL para aplicações.....	345
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>347</b>

## OBJETIVOS DO CURSO

### Objetivos específicos do curso:

---

Ao final do curso o treinando deverá ter desenvolvido os seguintes conceitos, habilidades e atitudes:

#### a) Conceitos a serem aprendidos

- Fundamentos e técnicas de programação;
- Princípios básicos da linguagem ADVPL;
- Comandos e funções específicas da Microsiga.

#### b) Habilidades e técnicas a serem aprendidas

- Resolução de algoritmos através de sintaxes orientadas a linguagem ADVPL;
- Análise de fontes de baixa complexidade da aplicação ERP Protheus;
- Desenvolvimento de pequenas customizações para o ERP Protheus.

#### c) Atitudes a serem desenvolvidas

- Adquirir conhecimentos através da análise das funcionalidades disponíveis no ERP Protheus;
- Embasar a realização de outros cursos relativos à linguagem ADVPL.

# MÓDULO 01: INTRODUÇÃO À PROGRAMAÇÃO

## 1. LÓGICA DE PROGRAMAÇÃO E ALGORITMOS

No aprendizado de qualquer linguagem de programação é essencial desenvolver os conceitos relacionados à lógica e à técnica da escrita de um programa.

Com foco nesta necessidade, este tópico descreverá resumidamente os conceitos envolvidos no processo de desenvolvimento de um programa, através dos conceitos relacionados a:

- Lógica de programação
- Algoritmos
- Diagramas de blocos

### 1.1. Lógica de Programação

#### Lógica

---

A lógica de programação é necessária para pessoas que desejam trabalhar com desenvolvimento de sistemas e programas. Ela permite definir a sequência lógica para o desenvolvimento. Então o que é lógica?

**Lógica de programação é a técnica de encadear pensamentos para atingir determinado objetivo.**

#### Sequência Lógica

---

Estes pensamentos podem ser descritos como uma sequência de instruções, que devem ser seguidas para que seja cumprida determinada tarefa.

**Sequência Lógica são passos executados até atingir um objetivo, ou solução de um problema.**

## **Instruções**

---

Na linguagem comum entende-se por instruções como “um conjunto de regras, ou normas definidas para a realização, ou emprego de algo”. Em informática, porém, instrução é a informação que indica a um computador uma ação elementar a executar. Convém ressaltar que uma ordem isolada não permite realizar o processo completo, para isso é necessário um conjunto de instruções colocadas em ordem seqüencial lógica.

Por exemplo, se quisermos fazer uma omelete de batatas, precisaremos colocar em prática uma série de instruções: descascar as batatas, bater os ovos, fritar as batatas, etc. É evidente que essas instruções devem ser executadas em uma ordem adequada – não se podem descascar as batatas depois de fritá-las.

Dessa maneira, uma instrução tomada isoladamente não tem muito sentido para obtermos o resultado. Precisamos colocar em prática o conjunto de todas as instruções, e na ordem correta.

**Instruções é um conjunto de regras, ou normas definidas para a realização ou emprego de algo. Em informática, é o que indica a um computador uma ação elementar a executar.**

## **Algoritmo**

---

Um algoritmo é formalmente uma seqüência finita de passos que levam à execução de uma tarefa. Podemos pensar em algoritmo como uma receita, uma seqüência de instruções que dão cabo de uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser claras e precisas.

Como exemplos de algoritmos é possível citar os algoritmos das operações básicas (adição, multiplicação, divisão e subtração) de números reais decimais. Outros exemplos seriam os

manuais de aparelhos eletrônicos que explicam passo-a-passo como, por exemplo, gravar um evento.

Até mesmo as coisas mais simples podem ser descritas por sequências lógicas, tais como:

- ☑ "Chupar uma bala"
  1. Pegar a bala;
  2. Retirar o papel;
  3. Chupar a bala;
  4. Jogar o papel no lixo.
  
- ☑ "Somar dois números quaisquer"
  1. Escreva o primeiro número no retângulo A;
  2. Escreva o segundo número no retângulo B;
  3. Some o número do retângulo A com número do retângulo B e coloque o resultado no retângulo C.

## 1.2. Desenvolvendo algoritmos

---

### Pseudocódigo

---

Os algoritmos são descritos em uma linguagem chamada pseudocódigo. Este nome é uma alusão à posterior implementação em uma linguagem de programação, ou seja, quando for utilizada a linguagem de programação propriamente dita como, por exemplo, ADVPL.

Por isso os algoritmos são independentes das linguagens de programação, sendo que ao contrário de uma linguagem de programação, não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação.

## Regras para a construção do Algoritmo

---

Para escrever um algoritmo precisamos descrever a sequência de instruções, de maneira simples e objetiva. Para isso utilizaremos algumas técnicas:

1. Usar somente um verbo por frase;
2. Imaginar que você está desenvolvendo um algoritmo para pessoas que não trabalham com Informática;
3. Usar frases curtas e simples;
4. Ser objetivo;
5. Procurar usar palavras que não tenham duplo sentido.

## Fases

---

Para implementar um algoritmo de simples interpretação e codificação é necessário inicialmente dividir o problema apresentado em três fases fundamentais:

- ☑ **ENTRADA:** são os dados de entrada do algoritmo;
- ☑ **PROCESSAMENTO:** são os procedimentos utilizados para chegar ao resultado final;
- ☑ **SAÍDA:** são os dados já processados.

### 1.2.1. Estudando algoritmos

Neste tópico serão demonstrados alguns algoritmos do cotidiano que foram implementados utilizando os princípios descritos nos tópicos anteriores.

-  **Mascar um chiclete**
-  **Utilizar um telefone público – cartão**
-  **Fritar um ovo**
-  **Trocar lâmpadas**
-  **Descascar batatas**
-  **Jogar o jogo da forca**
-  **Calcular a média de notas**
-  **Jogar o jogo da velha – contra o algoritmo**

### **Mascar um chiclete**

---

1. Pegar o chiclete.
2. Retirar o papel.
3. Mastigar.
4. Jogar o papel no lixo.

### **Utilizar um telefone público - cartão**

---

1. Retirar o telefone do gancho.
2. Esperar o sinal.
3. Colocar o cartão.
4. Discar o número.
5. Falar no telefone.
6. Colocar o telefone no gancho.

### **Fritar um ovo**

---

1. Pegar frigideira, ovo, óleo e sal.
2. Colocar óleo na frigideira.
3. Ascender o fogo.
4. Colocar a frigideira no fogo.
5. Esperar o óleo esquentar.
6. Quebrar o ovo na frigideira.
7. Jogar a casca no lixo.
8. Retirar a frigideira do fogo quando o ovo estiver no ponto.
9. Desligar o fogo.
10. Colocar sal a gosto.

### **Trocar lâmpadas**

---

1. Se a lâmpada estiver fora do alcance, pegar uma escada.
2. Pegar a lâmpada nova.
3. Se a lâmpada queimada estiver quente, pegar um pano.
4. Tirar a lâmpada queimada.
5. Colocar a lâmpada nova.

## **Descascar batatas**

---

1. Pegar faca, bacia e batatas.
2. Colocar a água na bacia.
3. Descascar todas as batatas.
  - 3.1. Colocar as batatas descascadas na bacia.

## **Jogar o jogo da forca**

---

1. Escolher a palavra.
2. Montar o diagrama do jogo.
3. Enquanto houver lacunas vazias e o corpo estiver incompleto:
  - 3.1. Se acertar a letra: escrever na lacuna correspondente.
  - 3.2. Se errar a letra: desenhar uma parte do corpo na forca.

## **Calcular a média de notas**

---

1. Enquanto houver notas a serem recebidas:
  - 1.1. Receber a nota.
2. Some todas as notas recebidas.
3. Divida o total obtido pela quantidade de notas recebidas.
4. Exiba a média das notas.

## **Jogar o jogo da velha – contra o algoritmo**

---

1. Enquanto existir um quadrado livre e ninguém ganhou ou perdeu o jogo:
  - 1.1. Espere a jogada do adversário, continue depois.
  - 1.2. Se o centro estiver livre: jogue no centro.
  - 1.3. Senão, se o adversário possuir dois quadrados em linha com um quadrado livre jogue neste quadrado.
  - 1.4. Senão, se há algum canto livre, jogue neste canto.

## 1.2.2. Teste de mesa

Todo algoritmo desenvolvido, deve ser testado. Este teste é chamado "**TESTE DE MESA**", que significa seguir as instruções do algoritmo de maneira precisa para verificar se o procedimento utilizado está correto ou não.

Para avaliar a aplicação do teste de mesa, utilizaremos o algoritmo de calcular a média de notas:

- Algoritmo: Calcular a média de notas

1. Enquanto houver notas a serem recebidas:
  - a. Receber a nota.
2. Some todas as notas recebidas.
3. Divida o total obtido pela quantidade de notas recebidas.
4. Exiba a média das notas.

### Teste de mesa:

---

1. Para cada nota informada, receber e registrar na tabela abaixo:

ID	Nota
----	------

2. Ao término das notas, a tabela deverá conter todas as notas informadas, como abaixo:

ID	Nota
1	8.0
2	7.0
3	8.0

4	8.0
5	7.0
6	7.0

3. Some todas as notas: 45

4. Divida a soma das notas, pelo total de notas informado:  $45/6 \rightarrow 7.5$

5. Exiba a média obtida: Mensagem (Média: 7.5)



**Anotações**

---

---

---

---

---



**Exercícios**

Exercício 01

Vamos aprimorar os seguintes algoritmos descritos na apostila:

- 1) Usar telefone público – cartão.
- 2) Fritar um ovo.
- 3) Mascar um chiclete
- 4) Trocar lâmpadas.
- 5) Descascar batatas.
- 6) Usar telefone público – cartão
- 7) Jogar o “Jogo da Força”

## 2. ESTRUTURAS DE PROGRAMAÇÃO

### 2.1. Diagrama de bloco

O diagrama de blocos é uma padronização que representa os passos lógicos de um determinado processamento.

Com o diagrama definimos uma sequência de símbolos com significado bem definido. Portanto, sua principal função é a de facilitar a visualização dos passos de um processamento.

#### Simbologia

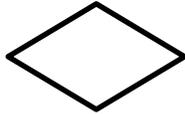
Existem diversos símbolos em um diagrama de bloco. No quadro abaixo estão representados alguns dos símbolos mais utilizados:

Símbolo	Função
	<b>Indica o início e o fim de um processamento.</b>
Terminador	
	<b>Processamento em geral.</b>
Processamento	



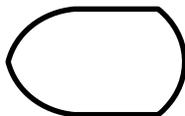
**Indica a entrada de dados através do teclado.**

Entrada  
Manual



**Indica um ponto no qual deverá ser efetuada uma escolha entre duas situações possíveis.**

Decisão



**Mostra os resultados obtidos com um processamento.**

Exibição



***Indica um documento utilizado pelo processamento, seja para entrada de informações ou para exibição dos dados disponíveis após um processamento.***

Documento



***Fique atento***

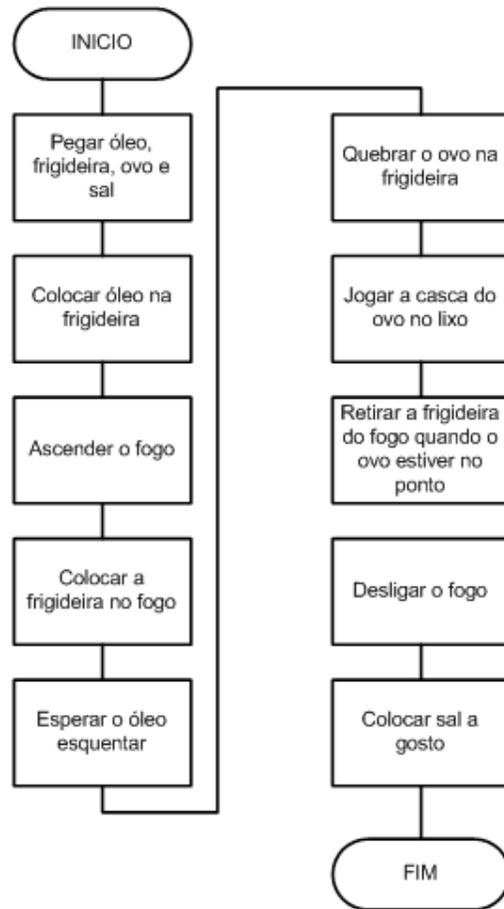
***Cada símbolo irá conter uma descrição pertinente à forma com o qual o mesmo foi utilizado no fluxo, indicando o processamento ou a informação que o mesmo representa.***

## **Representação de algoritmos através de diagramas de bloco**

---

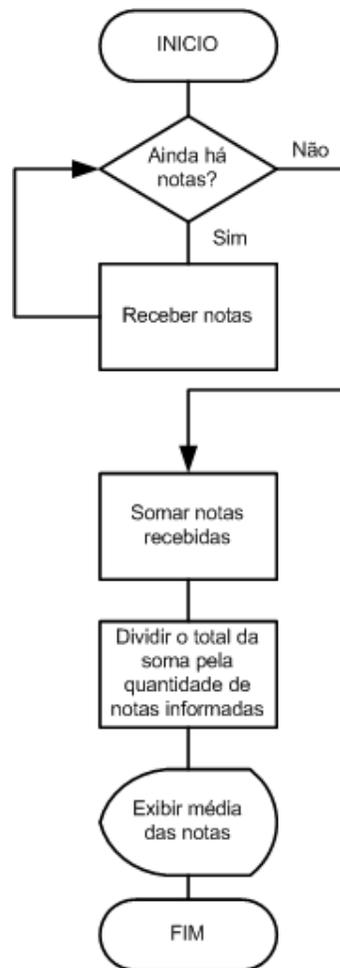
### **Algoritmo 01: Fritar um ovo**

1. Pegar frigideira, ovo, óleo e sal.
2. Colocar óleo na frigideira.
3. Acender o fogo.
4. Colocar a frigideira no fogo.
5. Esperar o óleo esquentar.
6. Quebrar o ovo na frigideira.
7. Jogar a casca no lixo.
8. Retirar a frigideira do fogo quando o ovo estiver no ponto.
9. Desligar o fogo.
10. Colocar sal a gosto.



### Algoritmo 02: Calcular a média de notas

1. Enquanto houver notas a serem recebidas:
  - a. Receber a nota.
2. Some todas as notas recebidas.
3. Divida o total obtido pela quantidade de notas recebidas.
4. Exiba a média das notas.



## 2.2. Estruturas de decisão e repetição

A utilização de estruturas de decisão e repetição em um algoritmo permite a realização de ações relacionadas às situações que influenciam na execução e na solução do problema.

Como foco na utilização da linguagem ADVPL, serão ilustradas as seguintes estruturas:

- Estruturas de decisão**
  - IF...ELSE
  - DO CASE ... CASE
  
- Estruturas de repetição**
  - WHILE...END
  - FOR...NEXT

### 2.2.1. Estruturas de decisão

Os comandos de decisão são utilizados em algoritmos cuja solução não é obtida através da utilização de ações meramente sequenciais, permitindo que estes comandos de decisão avaliem as condições necessárias para optar por uma ou outra maneira de continuar seu fluxo.

As estruturas de decisão que serão analisadas são:

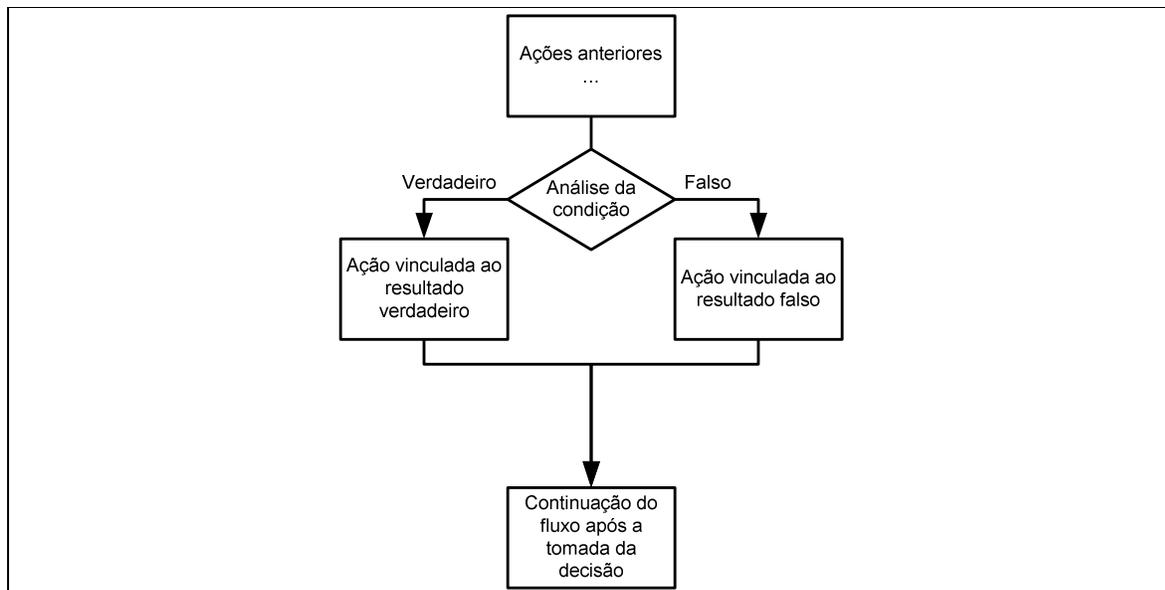
- IF...ELSE**
- DO CASE ... CASE**

#### **IF...ELSE**

---

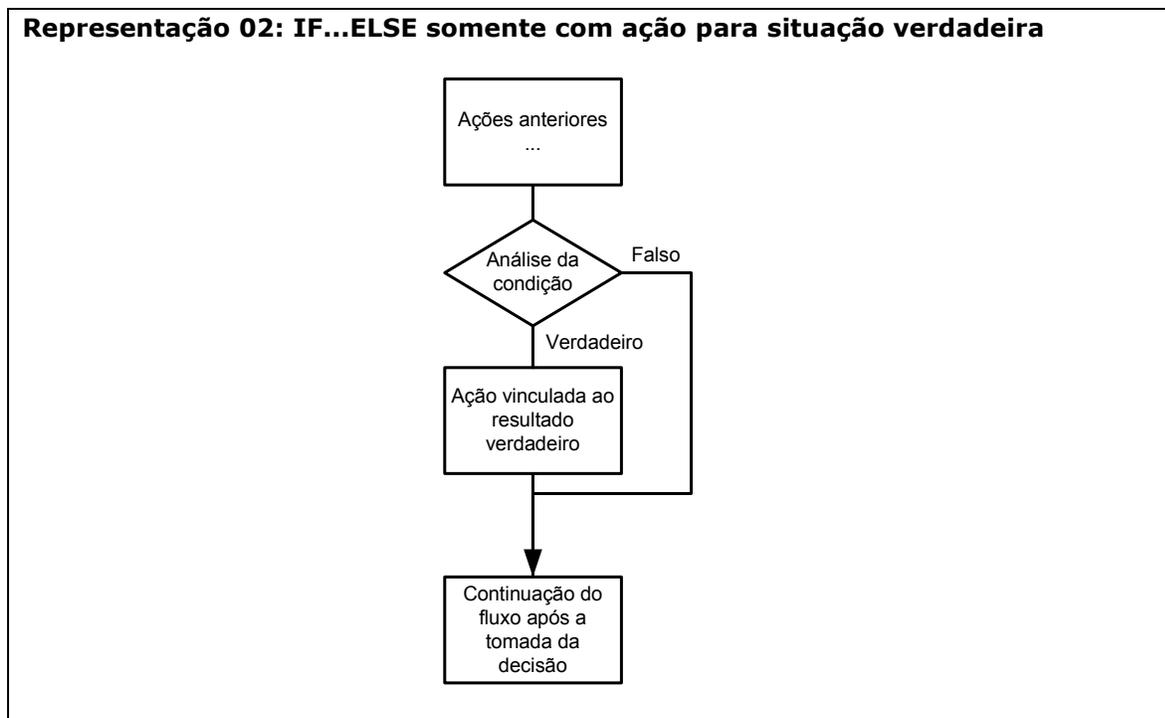
A estrutura IF...ELSE (Se/Senão) permite a análise de uma condição e a partir da qual ser executada uma de duas ações possíveis: Se a análise da condição resultar em um valor verdadeiro ou, se a análise da condição resultar em um valor falso.

**Representação 01: IF...ELSE com ações para ambas as situações**



Esta estrutura permite ainda que seja executada apenas uma ação, na situação em que a análise da condição resultar em um valor verdadeiro.

**Representação 02: IF...ELSE somente com ação para situação verdadeira**



Fique atento

***Apesar das linguagens de programação possuírem variações para a estrutura IF...ELSE, conceitualmente todas as representações podem ser descritas com base no modelo apresentado.***



**Dica**

***A linguagem ADVPL possui uma variação para a estrutura IF...ELSE, descrita como IF...ELSEIF...ELSE.***

***Com esta estrutura é possível realizar a análise de diversas condições em sequência, para as quais será avaliada somente a ação da primeira expressão cujo análise resultar em um valor verdadeiro.***



**Anotações**

---

---

---

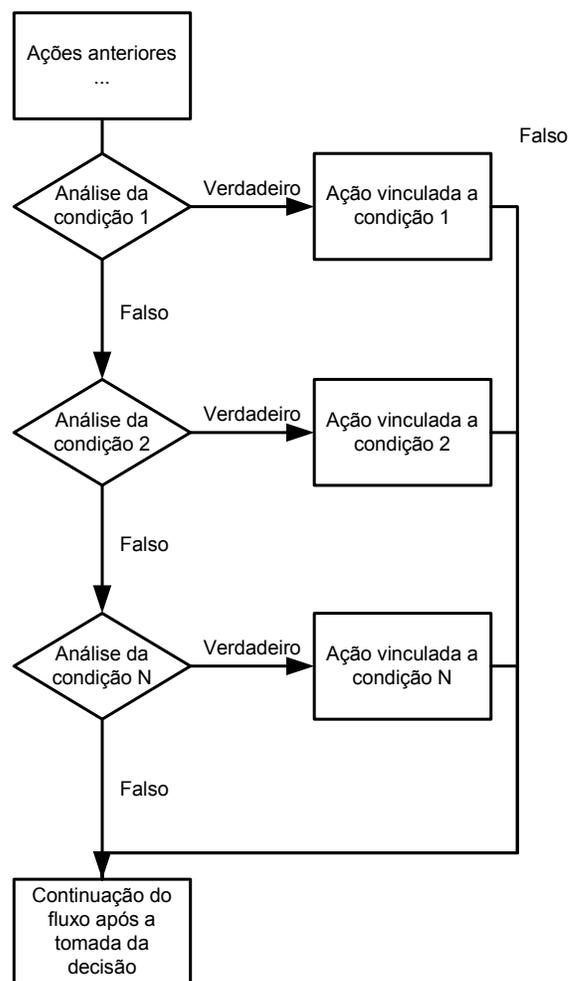
---

## DO CASE...CASE

---

A estrutura DO CASE...ENDCASE (Caso) permite a análise de diversas condições consecutivas, para as quais somente a condição para a primeira condição verdadeira será sua ação vinculada e executada.

O recurso de análise de múltiplas condições é necessário para solução de problemas mais complexos, nos quais as possibilidades de solução superam a mera análise de um único resultado verdadeiro, ou falso.





Fique  
atento

***Apesar das linguagens de programação possuírem variações para a estrutura DO CASE...CASE, conceitualmente todas as representações podem ser descritas com base no modelo apresentado.***

## 2.2.2. Estruturas de repetição

Os comandos de repetição são utilizados em algoritmos nas situações em que é necessário realizar uma determinada ação, ou um conjunto de ações para um número definido ou indefinido de vezes, ou ainda enquanto uma determinada condição for verdadeira.

As estruturas de decisão que serão analisadas são:

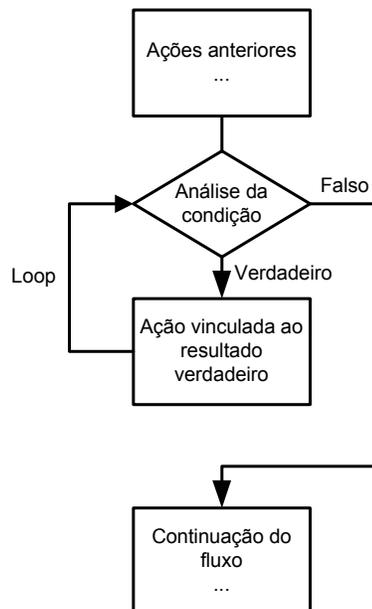
- ☑ **WHILE...END**
- ☑ **FOR...TO...NEXT**

### WHILE...END

---

Nessa estrutura, o conjunto de ações é executado enquanto a análise de uma condição de referência resultar em um valor verdadeiro. É importante verificar que o bloco somente será executado, inclusive se na primeira análise a condição resultar em um valor verdadeiro.

#### Representação: WHILE...END





Fique  
atento

***Existem diversas variações para a estrutura WHILE...END, na qual existe a possibilidade da primeira execução ser realizada sem a análise da condição, a qual valerá apenas a partir da segunda execução.***

***A linguagem ADVPL aceita a sintaxe DO WHILE...ENDDO que em outras linguagens representa a situação descrita anteriormente (análise da condição somente a partir da segunda execução), mas em ADVPL esta sintaxe tem o mesmo efeito do WHILE...END.***

## FOR...TO...NEXT

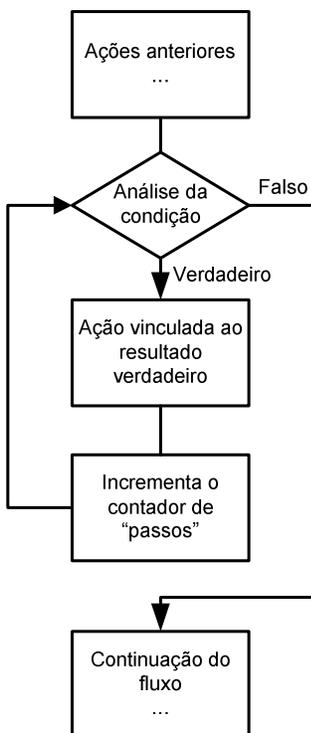
---

Nessa estrutura, o conjunto de ações é executado em uma quantidade de vezes definida, normalmente referenciada como "passo".

Para cada "passo" realizado pela estrutura FOR...TO...NEXT, é avaliada uma condição que verifica se foi atingido o número de execuções previamente definido. Dessa forma, a estrutura compreende um controle de número de "passos" executados e incrementado na análise da expressão NEXT.

Semelhante a estrutura WHILE...END, a primeira ação somente será realizada mediante um resultado verdadeiro na análise da condição.

### Representação: FOR...TO...NEXT





Fique atento

**A estrutura FOR...TO...NEXT, dependendo da linguagem de programação, permite a realização de um incremento simples a cada execução da instrução NEXT, ou a adição de outro valor ao contador que deverá especificado de acordo com a sintaxe da linguagem.**

**Em ADVPL pode ser utilizada a instrução "STEPS" para alterar o valor a ser adicionado no contador de passos a cada execução da instrução NEXT, sendo que este valor poderá ser até negativo, viabilizando uma contagem decrescente.**



### Exercícios

#### Exercício 02

Vamos montar os diagramas de blocos para os algoritmos desenvolvidos no exercício anterior:

1. Usar telefone público – cartão.
2. Fritar um ovo.
3. Mascar um chiclete.
4. Trocar lâmpadas.
5. Descascar batatas.
6. Jogar o "Jogo da Forca"

## MÓDULO 02: A LINGUAGEM ADVPL

A Linguagem ADVPL teve seu início em 1994, sendo na verdade uma evolução na utilização de linguagens no padrão xBase pela Microsiga Software S.A. (Clipper, Visual Objects e depois FiveWin). Com a criação da tecnologia Protheus era necessário criar uma linguagem que suportasse o padrão xBase para a manutenção de todo o código existente do sistema de ERP Siga Advanced. Foi então criada a linguagem chamada *Advanced Protheus Language*.

O ADVPL é uma extensão do padrão xBase de comandos e funções, operadores, estruturas de controle de fluxo e palavras reservadas, contando também com funções e comandos disponibilizados pela Microsiga que a torna uma linguagem completa para a criação de aplicações ERP prontas para a Internet. Também é uma linguagem orientada a objetos e eventos, permitindo ao programador desenvolver aplicações visuais e criar suas próprias classes de objetos.

Quando compilados, todos os arquivos de código tornam-se unidades de inteligência básicas, chamados APO's (de *Advanced Protheus Objects*). Tais APO's são mantidos em um repositório e carregados dinamicamente pelo PROTHEUS Server para a execução. Como não existe a "linkedição", ou união física do código compilado a um determinado módulo ou aplicação, funções criadas em ADVPL podem ser executadas em qualquer ponto do Ambiente Advanced Protheus.

O compilador e o interpretador da linguagem ADVPL é o próprio servidor PROTHEUS (PROTHEUS Server), e existe um Ambiente visual para o desenvolvimento integrado (PROTHEUSIDE), em que o código pode ser criado, compilado e depurado.

Os programas em ADVPL podem conter comandos, ou funções de interface com o usuário. De acordo com tal característica, tais programas são subdivididos nas seguintes categorias:

### **Programação Com Interface Própria com o Usuário**

---

Nessa categoria, são classificados os programas desenvolvidos para execução através do terminal remoto do Protheus, o Protheus Remote. O Protheus Remote é a aplicação encarregada da interface e da interação com o usuário, sendo que todo o processamento do código em ADVPL, o acesso ao banco de dados e o gerenciamento de conexões é efetuado no Protheus Server. O Protheus Remote é o principal meio de acesso à execução de rotinas escritas em ADVPL no Protheus Server. Por isso permite executar qualquer tipo de código, tenha ele interface com o usuário, ou não. Porém, nessa categoria são considerados apenas os programas que realizem algum tipo de interface remota, utilizando o protocolo de comunicação do Protheus.

Poderão ser criadas rotinas para a customização do sistema ERP Microsiga Protheus, desde processos adicionais até mesmo relatórios. A grande vantagem é aproveitar todo o Ambiente montado pelos módulos do ERP Microsiga Protheus. Porém, com o ADVPL é possível até mesmo criar toda uma aplicação, ou módulo, do começo.

Todo o código do sistema ERP Microsiga Protheus é escrito em ADVPL.

### **Programação Sem Interface Própria com o Usuário**

---

As rotinas criadas sem interface são consideradas nesta categoria porque geralmente têm uma utilização mais específica do que um processo adicional, ou um relatório novo. Tais rotinas não têm interface com o usuário através do Protheus Remote, e qualquer tentativa nesse sentido (como a criação de uma janela padrão) ocasionará uma exceção em tempo de execução. Essas rotinas são apenas processos, ou Jobs executados no Protheus Server. Algumas vezes, a interface dessas rotinas fica a cargo de aplicações externas, desenvolvidas em outras linguagens que são responsáveis por iniciar os processos no servidor Protheus, por meio dos meios disponíveis de integração e conectividade no Protheus.

De acordo com a utilização e com o meio de conectividade utilizado, estas rotinas são subcategorizadas assim:

#### **Programação por Processos**

Rotinas escritas em ADVPL podem ser iniciadas como processos individuais (sem interface), no Protheus Server através de duas maneiras: Iniciadas por outra rotina ADVPL por meio da chamada de funções como StartJob() ou CallProc(), ou iniciadas automaticamente, na inicialização do Protheus Server (quando propriamente configurado).

#### **Programação de RPC**

Com a utilização de uma biblioteca de funções disponível no Protheus (uma API de comunicação) são executadas as rotinas escritas em ADVPL diretamente no Protheus Server, por aplicações externas escritas em outras linguagens. Assim, é o que se chama de *RPC* (de *Remote Procedure Call*, ou *Chamada de Procedimentos Remota*).

O servidor Protheus também pode executar rotinas em ADVPL, em outros servidores Protheus, através de conexão TCP/IP direta, utilizando o conceito de *RPC*. Do mesmo modo,

as aplicações externas podem requisitar a execução de rotinas escritas em ADVPL, pela conexão TCP/IP direta.

### **Programação Web**

O Protheus Server pode também ser executado como um servidor Web, respondendo a requisições HTTP. No momento dessas requisições, pode executar rotinas escritas em ADVPL como processos individuais, enviando o resultado das funções como retorno das requisições para o cliente HTTP (como por exemplo, um Browser de Internet). Qualquer rotina escrita em ADVPL que não contenha comandos de interface, pode ser executada através de requisições HTTP. O Protheus permite a compilação de arquivos HTML, contendo código ADVPL embutido. São os chamados arquivos ADVPL ASP para a criação de páginas dinâmicas.

### **Programação TelNet**

TelNet é parte da gama de protocolos TCP/IP que permite a conexão a um computador remoto, utilizando uma aplicação cliente desse protocolo. O PROTHEUS Server pode emular um terminal pela execução de rotinas escritas em ADVPL. Ou seja, é possível escrever rotinas ADVPL cuja interface final será um terminal TelNet, ou um coletor de dados móvel.

### 3. ESTRUTURA DE UM PROGRAMA ADVPL

---

Um programa de computador nada mais é do que um grupo de comandos logicamente dispostos, com o objetivo de executar determinada tarefa. Esses comandos são gravados em um arquivo texto que é transformado em uma linguagem executável por um computador, utilizando o processo de *compilação*. A compilação substitui os comandos de alto nível (que os humanos compreendem) por instruções de baixo nível (compreendida pelo sistema operacional em execução no computador). No caso do ADVPL, não é o sistema operacional de um computador que executará o código compilado, mas sim o Protheus Server.

Dentro de um programa, os comandos e funções utilizados devem seguir as regras de sintaxe da linguagem utilizada, pois caso contrário, o programa será interrompido por erros. Os erros podem ser de compilação, ou de execução.

Erros de compilação são aqueles encontrados na sintaxe que não permitem que o arquivo de código do programa seja compilado. Podem ser comandos especificados de forma errônea, utilização inválida de operadores, etc.

Erros de execução são aqueles que acontecem depois da compilação, quando o programa está sendo executado. Podem ocorrer por inúmeras razões, mas geralmente se referem às funções não existentes, ou variáveis não criadas, ou inicializadas etc.

#### **Linhas de Programa**

---

As linhas existentes dentro de um arquivo texto de código de programa podem ser linhas de comando, linhas de comentário, ou linhas mistas.

#### **Linhas de Comando**

Linhas de comando possuem os comandos, ou instruções que serão executadas. Por exemplo:

```
Local nCnt  
Local nSoma := 0  
For nCnt := 1 To 10  
nSoma += nCnt  
Next nCnt
```

**Linhas de Comentário**

Linhas de comentário possuem um texto qualquer, mas não são executadas. Servem apenas para a documentação e para tornar mais fácil o entendimento do programa. Existem três formas de se comentar linhas de texto. A primeira delas é utilizar o sinal de \* (asterisco) no começo da linha:

\* Programa para cálculo do total

\* Autor: Microsiga Software S.A.

\* Data: 2 de outubro de 2001

Todas as linhas iniciadas com um sinal de asterisco são consideradas como comentário. É possível utilizar a palavra NOTE, ou dois símbolos da letra "e" comercial (&&) para realizar a função do sinal de asterisco. Porém todas essas formas de comentário de linhas são obsoletas e existem apenas para a compatibilização com o padrão xBase. A melhor maneira de comentar linhas em ADVPL é utilizar duas barras transversais:

```
// Programa para cálculo do total
// Autor: Microsiga Software S.A.
// Data: 2 de outubro de 2001
```

Outra forma de documentar textos é utilizando as barras transversais juntamente com o asterisco. É possível comentar todo um bloco de texto sem precisar comentar linha a linha:

```
/*
Programa para cálculo do total
Autor: Microsiga Software S.A.
Data: 2 de outubro de 2001
*/
```

Todo o texto encontrado entre a abertura (indicada pelos caracteres /\*) e o fechamento (indicada pelos caracteres \*/) é considerado como comentário.

#### **Linhas Mistas**

O ADVPL também permite que existam linhas de comando com comentário. Isto é possível adicionando-se as duas barras transversais (//) ao final da linha de comando e adicionando-se o texto do comentário:

```
Local nCnt
Local nSoma := 0 // Inicializa a variável com zero para a soma.
For nCnt := 1 To 10
nSoma += nCnt
Next nCnt
```

**Tamanho da Linha**

Assim como a linha física, delimitada pela quantidade de caracteres que pode ser digitado no editor de textos utilizado, existe uma linha considerada linha lógica. A linha lógica é aquela considerada para a compilação como uma única linha de comando.

A princípio, cada linha digitada no arquivo texto é diferenciada após o pressionamento da tecla <Enter>. Ou seja, a linha lógica é a linha física no arquivo. Porém, algumas vezes por limitação física do editor de texto, ou por estética, é possível "quebrar" a linha lógica em mais de uma linha física no arquivo texto. Isso é efetuado utilizando-se o sinal de ponto-e-vírgula (;).

```
If !Empty(cNome) .And. !Empty(cEnd) .And. ; <enter>
!Empty(cTel) .And. !Empty(cFax) .And. ; <enter>
!Empty(cEmail)

GravaDados(cNome,cEnd,cTel,cFax,cEmail)

Endif
```

Nesse exemplo existe uma linha de comando para a checagem das variáveis utilizadas. Como a linha torna-se muito grande, é possível dividi-la em mais de uma linha física, utilizando o sinal de ponto-e-vírgula. Se um sinal de ponto-e-vírgula for esquecido nas duas primeiras linhas, durante a execução do programa ocorrerá um erro, pois a segunda linha física será considerada como uma segunda linha de comando na compilação. E durante a execução esta linha não terá sentido.

### 3.1. Áreas de um Programa ADVPL

Apesar de não ser uma linguagem de padrões rígidos com relação à estrutura do programa, é importante identificar algumas de suas partes. Considere o programa de exemplo abaixo:

```
#include protheus.ch

/*
+=====+
| Programa: Cálculo do Fatorial      |
| Autor  : Microsiga Software S.A.   |
| Data   : 02 de outubro de 2001    |
+=====+
*/

User Function CalcFator()

Local nCnt
Local nResultado := 1 // Resultado do fatorial
Local nFator    := 5 // Número para o cálculo

// Cálculo do fatorial
For nCnt := nFator To 1 Step -1
nResultado *= nCnt
Next nCnt

// Exibe o resultado na tela, através da função alert
Alert("O fatorial de " + cValToChar(nFator) + ;
      " é " + cValToChar(nResultado))

// Termina o programa
Return
```

A estrutura de um programa ADVPL é composta pelas seguintes áreas:

- Área de Identificação
  - ✓ Declaração dos includes
  - ✓ Declaração da função
  - ✓ Identificação do programa
  
- Área de Ajustes Iniciais
  - ✓ Declaração das variáveis
  
- Corpo do Programa
  - ✓ Preparação para o processamento
  - ✓ Processamento
  
- Área de Encerramento

### **Área de Identificação**

---

Esta é uma área que não é obrigatória e é dedicada a documentação do programa. Quando existente, contém apenas comentários explicando a sua finalidade, data de criação, autor, etc., e aparece no começo do programa, antes de qualquer linha de comando.

O formato para esta área não é definido. É possível escrever qualquer tipo de informação desejada e ainda, escolher a formatação apropriada.

```

#include "protheus.ch"

/*
+=====+
| Programa: Cálculo do Fatorial      |
| Autor  : Microsiga Software S.A.  |
| Data   : 02 de outubro de 2001    |
+=====+
*/

User Function CalcFator()

```

Opcionalmente, é possível incluir definições de constantes utilizadas no programa ou inclusão de arquivos de cabeçalho nesta área.

### Área de Ajustes Iniciais

---

Nessa área geralmente são feitos os ajustes iniciais, importantes para o correto funcionamento do programa. Entre os ajustes se encontram declarações de variáveis, inicializações, abertura de arquivos, etc. Apesar do ADVPL não ser uma linguagem rígida e as variáveis poderem ser declaradas em qualquer lugar do programa, é aconselhável fazê-lo nessa área. Assim, é possível tornar o código legível e facilitar a identificação de variáveis não utilizadas.

```

Local nCnt
Local nResultado := 0 // Resultado do fatorial
Local nFator    := 10 // Número para o cálculo

```

### Corpo do Programa

---

Nessa área encontram-se as linhas de código do programa. Nela é realizada a tarefa necessária por meio da organização lógica dessas linhas de comando. Espera-se que as linhas de comando estejam organizadas de tal modo que no final da mesma, o resultado esperado seja

obtido, seja ele armazenado em um arquivo, ou em variáveis de memória, pronto para ser exibido ao usuário através de um relatório ou na tela.

```
// Cálculo do fatorial  
For nCnt := nFator To 1 Step -1  
nResultado *= nCnt  
Next nCnt
```

A preparação para o processamento é formada pelo conjunto de validações e processamentos necessários antes da realização do processamento em si.

Avaliando o processamento do cálculo do fatorial descrito anteriormente, é possível definir que a validação inicial a ser realizada é o conteúdo da variável nFator, pois a mesma determinará a correta execução do código.

```
// Cálculo do fatorial
nFator := GetFator()
// GetFator – função ilustrativa na qual a variável recebe a informação do usuário.

If nFator <= 0
    Alert("Informação inválida")
    Return
Endif

For nCnt := nFator To 1 Step -1
nResultado *= nCnt
Next nCnt
```

## Área de Encerramento

---

É nesta área onde as finalizações são efetuadas. É onde os arquivos abertos são fechados, e o resultado da execução do programa é utilizado. É possível exibir o resultado armazenado em uma variável, ou em um arquivo. Ainda, finalizar, se a tarefa já tiver sido toda completada no corpo do programa. É nessa área que se encontra o encerramento do programa. Todo programa em ADVPL deve sempre terminar com a palavra chave return.

```
// Exibe o resultado na tela, através da função alert
Alert("O fatorial de " + cValToChar(nFator) + ;
      " é " + cValToChar(nResultado))

// Termina o programa
Return
```



*Anotações*

---

---

---

---

---

## 4. DECLARAÇÃO E ATRIBUIÇÃO DE VARIÁVEIS

### 4.1. Tipo de Dados

O ADVPL não é uma linguagem de tipos rígidos (strongly typed), o que significa que variáveis de memória podem receber diferentes tipos de dados durante a execução do programa.

As variáveis podem também conter objetos, mas os tipos primários da linguagem são:

#### Numérico

O ADVPL não diferencia valores inteiros de valores com ponto flutuante. Portanto, é possível criar variáveis numéricas com qualquer valor dentro do intervalo permitido. Os seguintes elementos são do tipo de dado numérico:

```
2
43.53
0.5
0.00001
1000000
```

Uma variável do tipo de dado numérico pode conter um número de dezoito dígitos, incluindo o ponto flutuante, no intervalo de  $2.2250738585072014 \text{ E}-308$  até  $1.7976931348623158 \text{ E}+308$ .

#### Lógico

Valores lógicos em ADVPL são identificados através de .T. ou .Y. para verdadeiro e .F. ou .N. para falso (independentemente se os caracteres estiverem em maiúsculo, ou minúsculo).

## Caractere

---

Strings ou cadeias de caracteres são identificadas em ADVPL por blocos de texto entre aspas duplas (") ou aspas simples ('):

"Olá mundo!"

'Esta é uma string'

"Esta é 'outra' string"

Uma variável do tipo caractere pode conter strings com no máximo 1 MB, ou seja, 1048576 caracteres.

## Data

---

O ADVPL tem um tipo de dados específico para datas. Internamente as variáveis desse tipo de dado são armazenadas como um número correspondente à **data Juliana**.

Variáveis do tipo de dados Data não podem ser declaradas diretamente, e sim com a utilização de funções específicas como, por exemplo, CTOD() que converte uma string para data.

## **Array**

---

O Array é um tipo de dado especial. É a disposição de outros elementos em colunas e linhas. O ADVPL suporta arrays unidimensionais (vetores) ou multidimensionais (matrizes). Os elementos de um array são acessados através de índices numéricos iniciados em 1, identificando a linha e coluna para quantas dimensões existirem.

Arrays devem ser utilizadas com cautela, pois se forem muito grandes podem exaurir a memória do servidor.

## **Bloco de Código**

---

O bloco de código é um tipo de dado especial. É utilizado para armazenar instruções escritas em ADVPL que poderão ser executadas posteriormente.

### **4.2. Declaração de variáveis**

---

Variáveis de memória são um dos recursos mais importantes de uma linguagem. São áreas de memória criadas para armazenar informações utilizadas por um programa para a execução de tarefas. Por exemplo, quando o usuário digita uma informação qualquer, como o nome de um produto, em uma tela de um programa esta informação é armazenada em uma variável de memória para posteriormente ser gravada, ou impressa.

A partir do momento que uma variável é criada, não é mais necessário referenciar ao seu conteúdo, e sim, seu nome.

O nome de uma variável é um identificador único que deve respeitar um número **máximo de 10 caracteres**. O ADVPL não impede a criação de uma variável de memória cujo nome contenha mais de 10 caracteres, **porém apenas os 10 primeiros serão considerados** para a localização do conteúdo armazenado. Portanto, se forem criadas duas variáveis cujos 10 primeiros caracteres forem iguais, como nTotalGeralAnual e nTotalGeralMensal, as referências a qualquer uma delas no programa resultarão o mesmo, ou seja, serão a mesma variável:

```
nTotalGeralMensal := 100
nTotalGeralAnual := 300
Alert("Valor mensal: " + cValToChar(nTotalGeralMensal))
```

Quando o conteúdo da variável nTotalGeralMensal é exibido, o seu valor será de 300. Isso acontece porque no momento em que esse valor foi atribuído à variável nTotalGeralAnual, o ADVPL considerou apenas os 10 primeiros caracteres (assim como o faz quando deve exibir o valor da variável nTotalGeralMensal). Ou seja, considerou-as como a mesma variável. Assim o valor original de 100 foi substituído pelo de 300.

### 4.3. Escopo de variáveis

---

O ADVPL não é uma linguagem de tipos rígidos para variáveis, ou seja, não é necessário informar o tipo de dados que determinada variável irá conter no momento de sua declaração, e o seu valor pode mudar durante a execução do programa.

Também não há necessidade de declarar variáveis em uma seção específica do seu código fonte, embora seja aconselhável declarar todas as variáveis necessárias no começo, tornando a manutenção mais fácil, e evitando a declaração de variáveis desnecessárias.

Para declarar uma variável é necessário utilizar um *identificador de escopo*. Um identificador de escopo é uma palavra chave que indica a que contexto do programa a variável declarada pertence. O contexto de variáveis pode ser *local* (visualizadas apenas dentro do programa atual), *público* (visualizadas por qualquer outro programa), entre outros.

#### O Contexto de Variáveis dentro de um Programa

---

As variáveis declaradas em um programa ou função são visíveis de acordo com o escopo onde são definidas. Como também do escopo depende o tempo de existência das variáveis. A definição do escopo de uma variável é efetuada no momento de sua declaração.

```
Local nNúmero := 10
```

Esta linha de código declara uma variável chamada nNúmero, indicando que aonde pertence seu escopo é local.

Os identificadores de escopo são:

- Local
- Static
- Private
- Public

O ADVPL não é rígido em relação à declaração de variáveis no começo do programa. A inclusão de um identificador de escopo não é necessário para a declaração de uma variável, contanto que um valor lhe seja atribuído.

```
nNumero2 := 15
```

Quando um valor é atribuído a uma variável em um programa ou função, o ADVPL criará a variável se ela não tiver sido declarada anteriormente. A variável então é criada como se tivesse sido declarada como Private.

Devido a essa característica, quando se pretende fazer uma atribuição a uma variável declarada previamente, mas escreve-se o nome da variável de forma incorreta, o ADVPL não gerará nenhum erro de compilação ou de execução. Assim, compreenderá o nome da variável escrito de forma incorreta como se fosse a criação de uma nova variável. Dessa forma, será alterada a lógica do programa, e é um erro muitas vezes difícil de identificar.

## Variáveis de escopo local

---

Variáveis de escopo local são pertencentes apenas ao escopo da função onde foram declaradas e devem ser explicitamente declaradas com o identificador LOCAL, como no exemplo:

```
Function Pai()  
Local nVar := 10, aMatriz := {0,1,2,3}  
.  
<comandos>  
.  
Filha()  
.  
<mais comandos>  
.  
Return(.T.)
```

Neste exemplo, a variável nVar foi declarada como local e atribuída com o valor 10. Quando a função Filha é executada, nVar ainda existe mas não pode ser acessada. Quando a execução da função Pai terminar, a variável nVar é destruída. Qualquer variável, com o mesmo nome no programa que chamou a função Pai, não é afetada.

Variáveis de escopo local são criadas automaticamente, cada vez que a função onde forem declaradas for ativada. Elas continuam a existir e mantêm seu valor até o fim da ativação da função (ou seja, até que a função retorne o controle para o código que a executou). Se uma função é chamada recursivamente (por exemplo, chama a si mesma), cada chamada em recursão cria um novo conjunto de variáveis locais.

A visibilidade de variáveis de escopo locais é idêntica ao escopo de sua declaração, ou seja, a variável é visível em qualquer lugar do código fonte em que foi declarada. Se uma função é chamada recursivamente, apenas as variáveis de escopo local criadas na mais recente ativação são visíveis.

## **Variáveis de escopo static**

---

Variáveis de escopo static funcionam basicamente como as variáveis de escopo local, mas mantêm seu valor através da execução e devem ser declaradas explicitamente no código, com o identificador STATIC.

O escopo das variáveis static depende de onde são declaradas. Se forem declaradas dentro do corpo de uma função ou procedimento, seu escopo será limitado àquela rotina. Se forem declaradas fora do corpo de qualquer rotina, seu escopo afeta a todas as funções declaradas no fonte.

Neste exemplo, a variável nVar é declarada como static e inicializada com o valor 10:

**Function Pai()**

Static nVar := 10

.

<comandos>

.

Filha()

.

<mais comandos>

.

**Return(.T.)**

Quando a função Filha é executada, nVar ainda existe mas não pode ser acessada. Diferente de variáveis declaradas como LOCAL ou PRIVATE, nVar continua a existir e mantém seu valor atual quando a execução da função Pai termina. Entretanto, somente pode ser acessada por execuções subseqüentes da função Pai.

### **Variáveis de escopo private**

---

A declaração é opcional para variáveis privadas. Mas podem ser declaradas explicitamente com o identificador PRIVATE.

Adicionalmente, a atribuição de valor a uma variável não criada anteriormente, de forma automática cria-se a variável como privada. Uma vez criada, uma variável privada continua a existir e mantém seu valor até que o programa ou função onde foi criada termine (ou seja, até que a função onde foi feita retorne para o código que a executou). Neste momento, é automaticamente destruída.

É possível criar uma nova variável privada com o mesmo nome de uma variável já existente. Entretanto, a nova (duplicada) variável pode apenas ser criada em um nível de ativação inferior ao nível onde a variável foi declarada pela primeira vez (ou seja, apenas em uma função chamada pela função onde a variável já havia sido criada). A nova variável privada *esconderá* qualquer outra variável privada ou pública (veja a documentação sobre variáveis públicas) com o mesmo nome enquanto existir.

Uma vez criada, uma variável privada é visível em todo o programa, enquanto não for destruída automaticamente. Quando a rotina que a criou terminar, ou uma outra variável privada com o mesmo nome for criada em uma subfunção chamada (neste caso, a variável existente torna-se inacessível até que a nova variável privada seja destruída).

Em termos mais simples, uma variável privada é visível dentro da função de criação e todas as funções chamadas por esta, a menos que uma função chamada crie sua própria variável privada com o mesmo nome.

**Por exemplo:**

**Function Pai()**

Private nVar := 10

<comandos>

.

Filha()

<mais comandos>

.

**Return(.T.)**

Neste exemplo, a variável nVar é criada com escopo private e inicializada com o valor 10. Quando a função Filha é executada, nVar ainda existe e, diferente de uma variável de escopo local, pode ser acessada pela função Filha. Quando a função Pai terminar, nVar será destruída e qualquer declaração de nVar anterior se tornará acessível novamente.



Fique atento

**No ambiente ERP Protheus existe uma convenção adicional que fala sobre as variáveis, em uso pela aplicação, não sejam incorretamente manipuladas. Por esta convenção deve ser adicionado o caractere "\_" antes do nome das variáveis PRIVATE e PUBLIC. Para maiores informações avaliar o tópico: "Boas Práticas de Programação".**

**Exemplo: Private \_dData**

## Variáveis de escopo public

---

É possível criar variáveis de escopo public dinamicamente, no código com o identificador PUBLIC. As variáveis desse escopo continuam a existir e mantêm seu valor até o fim da execução da thread (conexão). Também é possível criar uma variável de escopo private com o mesmo nome de uma variável de escopo public existente, entretanto, não é permitido criar uma variável de escopo public com o mesmo nome de uma variável de escopo private existente.

Uma vez criada, uma variável de escopo public é visível em todo o programa em que foi declarada, até que seja *escondida* por uma variável de escopo private, criada com o mesmo nome. A nova variável de escopo private criada *esconde* a variável de escopo public existente, e esta se tornará inacessível até que a nova variável private seja destruída. Por exemplo:

### Function Pai()

```
Public nVar := 10
```

```
<comandos>
```

```
.
```

```
Filha()
```

```
<mais comandos>
```

```
.
```

### Return(.T.)

Nesse exemplo, nVar é criada como public e inicializada com o valor 10. Quando a função Filha é executada, nVar ainda existe e pode ser acessada. Diferente de variáveis locais ou privadas, nVar ainda existe após o término da execução da função Pai.

Diferentemente dos outros identificadores de escopo, quando uma variável é declarada como pública sem ser inicializada, o valor assumido é falso (.F.) e não nulo (nil).



Fique  
atento

**No ambiente ERP Protheus, existe uma convenção adicional que deve ser respeitada. Essa convenção diz que as variáveis em uso pela aplicação não sejam incorretamente manipuladas. Por essa convenção deve ser adicionado o caractere “\_” antes do nome de variáveis PRIVATE e PUBLIC. Para Maiores informações avalier o tópico: Boas Práticas de Programação.**

Exemplo: **Public\_cRotina**

#### 4.4. Entendendo a influência do escopo das variáveis

Considere as linhas de código de exemplo:

```
nResultado := 250 * (1 + (nPercentual / 100))
```

Se esta linha for executada em um programa ADVPL ocorrerá um erro de execução com a mensagem "variable does not exist: nPercentual", pois esta variável está sendo utilizada em uma expressão de cálculo sem ter sido declarada. Para solucionar esse erro, deve-se declarar a variável previamente:

```
Local nPercentual, nResultado  
nResultado := 250 * (1 + (nPercentual / 100))
```

No exemplo acima, as variáveis são declaradas previamente, utilizando o identificador de escopo *local*. Quando a linha de cálculo for executada, o erro de variável não existente não mais ocorrerá. Porém as variáveis não inicializadas têm sempre o valor default nulo (Nil) e este valor não pode ser utilizado em um cálculo, pois também gerará erros de execução (nulo não pode ser dividido por 100). A resolução deste problema é efetuada inicializando-se a variável através de uma das formas:

```
Local nPercentual, nResultado  
nPercentual := 10  
nResultado := 250 * (1 + (nPercentual / 100))  
  
ou  
  
Local nPercentual := 10, nResultado  
nResultado := 250 * (1 + (nPercentual / 100))
```

A diferença, entre o último exemplo e os dois anteriores, é que a variável é inicializada no momento da declaração. Nos dois exemplos, a variável é primeiro declarada e então inicializada em uma outra linha de código.

É aconselhável optar pelo operador de atribuição composto de dois pontos e sinal de igual, pois o operador de atribuição, utilizando somente o sinal de igual, pode ser facilmente confundido com o operador relacional (para comparação), durante a criação do programa.

## 4.5. Operações com Variáveis

### 4.5.1. Atribuição de variáveis

Uma vez que um valor lhe seja atribuído, o tipo de dado de uma variável é igual ao tipo de dado do valor atribuído. Ou seja, uma variável passa a ser numérica se um número lhe é atribuído, passa a ser caractere se uma string de texto lhe for atribuída etc. Porém, mesmo que uma variável seja de determinado tipo de dado, é possível mudar o tipo de variável atribuindo outro tipo a ela:

```
01 Local xVariavel // Declara a variável inicialmente com valor nulo
02
03 xVariavel := "Agora a variável é caractere..."
04 Alert("Valor do Texto: " + xVariavel)
05
06 xVariavel := 22 // Agora a variável é numérica
07 Alert(cValToChar(xVariavel))
08
09 xVariavel := .T. // Agora a variável é lógica
10 If xVariavel
11   Alert("A variável tem valor verdadeiro...")
12 Else
13   Alert("A variável tem valor falso...")
14 Endif
15
16 xVariavel := Date() // Agora a variável é data
17 Alert("Hoje é: " + DtoC(xVariavel))
18
19 xVariavel := nil // Nulo novamente
20 Alert("Valor nulo: " + xVariavel)
21
22 Return
```

No programa de exemplo anterior, a variável xVariavel é utilizada para armazenar diversos tipos de dados. A letra "x", em minúsculo no começo do nome, é utilizada para indicar uma variável que pode conter diversos tipos de dados, segundo a Notação Húngara (consulte documentação específica para detalhes). Este programa troca os valores da variável e exibe seu conteúdo para o usuário através da função ALERT(). Essa função recebe um parâmetro que deve ser do tipo string de caractere. Por isso, dependendo do tipo de dado da variável xVariavel, é necessário fazer uma conversão antes.

Apesar dessa flexibilidade de utilização de variáveis, devem-se tomar cuidados na passagem de parâmetros para funções ou comandos, e na concatenação (ou soma) de valores. Note a linha 20 do programa de exemplo. Quando esta linha é executada, a variável xVariavel contém o valor nulo. A tentativa de soma de tipos de dados diferentes gera erro de execução do programa. Nesta linha do exemplo, ocorrerá um erro com a mensagem "type mismatch on +".

Excetuando-se o caso do valor nulo, para os demais devem ser utilizadas funções de conversão, quando é necessário concatenar tipos de dados diferentes (por exemplo, nas linhas 07 e 17).

Note também que quando uma variável é do tipo de dado lógico, ela pode ser utilizada diretamente para checagem (linha 10):

```
If xVariavel
    é o mesmo que
If xVariavel = .T.
```

## 4.5.2. Operadores da linguagem ADVPL

### Operadores comuns

---

Na documentação sobre variáveis há uma breve demonstração de como atribuir valores a uma variável da forma mais simples. O ADVPL amplia significativamente a utilização de variáveis, através do uso de expressões e funções.

Uma expressão é um conjunto de operadores e operandos, cujo resultado pode ser atribuído a uma variável ou então analisado para a tomada de decisões. Por exemplo:

```
Local nSalario := 1000, nDesconto := 0.10
Local nAumento, nSalLiquido
nAumento := nSalario * 1.20
nSalLiquido := nAumento * (1-nDesconto)
```

Nesse exemplo são utilizadas algumas expressões para calcular o salário líquido após um aumento. Os operandos de uma expressão podem ser uma variável, uma constante, um campo de arquivo, ou uma função.

## Operadores Matemáticos

---

Os operadores utilizados em ADVPL para cálculos matemáticos são:

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
** ou ^	Exponenciação
%	Módulo (Resto da Divisão)

## Operadores de String

---

Os operadores utilizados em ADVPL para tratamento de caracteres são:

+	Concatenação de strings (união).
-	Concatenação de strings com eliminação dos brancos finais das strings intermediárias.
\$	Comparação de Substrings (contido em).

## Operadores Relacionais

---

Os operadores utilizados em ADVPL para operações e avaliações relacionais são:

<	Comparação Menor
>	Comparação Maior
=	Comparação Igual
==	Comparação Exatamente Igual (para caracteres)
<=	Comparação Menor ou Igual
>=	Comparação Maior ou Igual
<> ou # ou !=	Comparação Diferente

## Operadores Lógicos

---

Os operadores utilizados em ADVPL para operações e avaliações lógicas são:

.And.	E lógico
.Or.	OU lógico
.Not. ou !	NÃO lógico

## Operadores de Atribuição

---

Os operadores utilizados em ADVPL para atribuição de valores a variáveis de memória são:

<code>:=</code>	Atribuição Simples
<code>+=</code>	Adição e Atribuição em Linha
<code>-=</code>	Subtração e Atribuição em Linha
<code>*=</code>	Multiplicação e Atribuição em Linha
<code>/=</code>	Divisão e Atribuição em Linha
<code>**= ou ^=</code>	Exponenciação e Atribuição em Linha
<code>%=</code>	Módulo (resto da divisão) e Atribuição em Linha

### **Atribuição Simples**

O sinal de igualdade é utilizado para atribuir valor a uma variável de memória.

nVariavel := 10

☑ **Atribuição em Linha**

O operador de atribuição em linha é caracterizado por dois pontos e o sinal de igualdade. Tem a mesma função do sinal de igualdade sozinho, porém aplica a atribuição às variáveis. Com ele é possível atribuir mais de uma variável ao mesmo tempo.

```
nVar1 := nVar2 := nVar3 := 0
```

Quando diversas variáveis são inicializadas em uma mesma linha, a atribuição começa da direita para a esquerda, ou seja, nVar3 recebe o valor zero inicialmente, nVar2 recebe o conteúdo de nVar3 e nVar1 recebe o conteúdo de nVar2 por final.

Com o operador de atribuição em linha, é possível substituir as inicializações individuais de cada variável por uma inicialização apenas:

```
Local nVar1 := 0, nVar2 := 0, nVar3 := 0
```

por

```
Local nVar1 := nVar2 := nVar3 := 0
```

O operador de atribuição em linha também pode ser utilizado para substituir valores de campos em um banco de dados.

## ☑ Atribuição Composta

Os operadores de atribuição composta são uma facilidade da linguagem ADVPL para expressões de cálculo e atribuição. Dessa forma é possível economizar digitação:

Operador	Exemplo	Equivalente a
<code>+=</code>	<code>X += Y</code>	<code>X = X + Y</code>
<code>-=</code>	<code>X -= Y</code>	<code>X = X - Y</code>
<code>*=</code>	<code>X *= Y</code>	<code>X = X * Y</code>
<code>/=</code>	<code>X /= Y</code>	<code>X = X / Y</code>
<code>**=</code> ou <code>^=</code>	<code>X **= Y</code>	<code>X = X ** Y</code>
<code>%=</code>	<code>X %= Y</code>	<code>X = X % Y</code>

## Operadores de Incremento/Decremento

---

A linguagem ADVPL possui operadores para realizar incremento ou decremento de variáveis. Entende-se por incremento aumentar o valor de uma variável numérica em 1 e entende-se por decremento diminuir o valor da variável em 1. Os operadores são:

<code>++</code>	Incremento Pós ou Pré-fixado
<code>--</code>	Decremento Pós ou Pré-fixado

Os operadores de decremento/incremento podem ser colocados tanto antes (pré-fixado) como depois (pós-fixado) do nome da variável. Dentro de uma expressão, a ordem do operador é muito importante, podendo alterar o resultado da expressão. Os operadores incrementais são executados da esquerda para a direita dentro de uma expressão.

```
Local nA := 10
Local nB := nA++ + nA
```

O valor da variável nB resulta em 21, pois a primeira referência a nA (antes do ++) continha o valor 10 que foi considerado e imediatamente aumentado em 1. Na segunda referência a nA, este já possuía o valor 11. O que foi efetuado foi a soma de 10 mais 11, igual a 21. O resultado final após a execução dessas duas linhas é a variável nB contendo 21 e a variável nA contendo 11.

No entanto:

```
Local nA := 10
Local nB := ++nA + nA
```

Resulta em 22 pois o operador incremental aumentou o valor da primeira nA antes que seu valor fosse considerado.

### **Operadores Especiais**

---

Além dos operadores comuns, o ADVPL possui alguns outros operadores, ou identificadores. Estas são suas finalidades:

()	Agrupamento ou Função
[]	Elemento de Matriz
{}	Definição de Matriz, Constante ou Bloco de Código
->	Identificador de Apelido
&	Macro substituição
@	Passagem de parâmetro por referência
	Passagem de parâmetro por valor

✓ Os parênteses são utilizados para agrupar elementos em uma expressão, mudando a ordem de precedência da avaliação da expressão (segundo as regras matemáticas, por exemplo). Também servem para envolver os argumentos de uma função.

✓ Os colchetes são utilizados para especificar um elemento específico de uma matriz. Por exemplo, A[3,2] refere-se ao elemento da matriz A na linha 3, coluna 2.

- ✓ As chaves são utilizadas para a especificação de matrizes literais, ou blocos de código. Por exemplo, `A:={10,20,30}` cria uma matriz chamada A com três elementos.
  
- ✓ O símbolo `->` identifica um campo de um arquivo, diferenciando-o de uma variável. Por exemplo, `FUNC->nome` refere-se ao campo nome do arquivo FUNC. Mesmo que exista uma variável chamada nome, é o campo nome que será acessado.
  
- ✓ O símbolo `&` identifica uma avaliação de expressão através de macro e é visto em detalhes na documentação sobre **macro substituição**.
  
- ✓ O símbolo `@` é utilizado para indicar que durante a passagem de uma variável para uma função, ou procedimento ela seja tomada como uma referência e não como valor.
  
- ✓ O símbolo `||` é utilizado para indicar que durante a passagem de uma variável para uma função ou procedimento, ela seja tomada como um e valor não como referência.

## **Ordem de Precedência dos Operadores**

---

Dependendo do tipo de operador, existe uma ordem de precedência para a avaliação dos operandos. Em princípio, todas as operações, com os operadores, são realizadas da esquerda para a direita, se eles tiverem o mesmo nível de prioridade.

A ordem de precedência, ou nível de prioridade de execução dos operadores em ADVPL é:

1. Operadores de Incremento/Decremento pré-fixado
2. Operadores de String
3. Operadores Matemáticos
4. Operadores Relacionais
5. Operadores Lógicos
6. Operadores de Atribuição
7. Operadores de Incremento/Decremento pós-fixado

Em expressões complexas com diferentes tipos de operadores, a avaliação seguirá essa sequência. Caso exista mais de um operador do mesmo tipo (ou seja, de mesmo nível), a avaliação se dá da esquerda para direita. Para os operadores matemáticos entretanto, há uma precedência a seguir:

1. Exponenciação
2. Multiplicação e Divisão
3. Adição e Subtração

Considere o exemplo:

Local nResultado := 2+10/2+5\*3+2^3

O resultado desta expressão é 30, pois primeiramente é calculada a exponenciação  $2^3(=8)$ , então são calculadas as multiplicações e divisões  $10/2(=5)$  e  $5*3(=15)$ , e finalmente as adições resultando em  $2+5+15+8(=30)$ .

### **Alteração da Precedência**

A utilização de parênteses dentro de uma expressão altera a ordem de precedência dos operadores. Operandos entre parênteses são analisados antes dos que se encontram

fora dos parênteses. Se existirem mais de um conjunto de parênteses não-aninhados, o grupo mais a esquerda será avaliado primeiro e assim sucessivamente.

Local nResultado := (2+10)/(2+5)*3+2^3
--

No exemplo acima, primeiro será calculada a exponenciação  $2^3(=8)$ . Em seguida  $2+10(=12)$  será calculado,  $2+5(=7)$  calculado, e finalmente a divisão e a multiplicação serão efetuadas, o que resulta em  $12/7*3+8(=13.14)$ .

Se existirem vários parênteses aninhados, ou seja, colocados um dentro do outro, a avaliação ocorrerá do parênteses mais interno em direção ao mais externo.

### 4.5.3. Operação de Macro Substituição

O operador de macro substituição, simbolizado pelo "e" comercial (&), é utilizado para a avaliação de expressões em tempo de execução. Funciona como se uma expressão armazenada fosse compilada em tempo de execução, antes de ser de fato executada.

Considere o exemplo:

```
01 X := 10
02 Y := "X + 1"
03 B := &Y // O conteúdo de B será 11
```

A variável X é atribuída com o valor 10, enquanto a variável Y é atribuída com a string de caracteres contendo "X + 1".

A terceira linha utiliza o operador de macro. Esta linha faz com que o número 11 seja atribuído à variável B. É possível perceber que esse é o valor resultante da expressão em formato de caractere contida na variável Y.

Utilizando-se uma técnica matemática elementar, a substituição, temos que na segunda linha, Y é definido como "X + 1", então é possível substituir Y na terceira linha:

```
03 B := &"X + 1"
O operador de macro cancela as aspas:
03 B := X + 1
```

O operador de macro remove as aspas, o que deixa uma parte do código para ser executado. Deve-se ter em mente que tudo isso acontece em tempo de execução, o que torna tudo muito dinâmico. Uma utilização interessante é criar um tipo de calculadora ou, um avaliador de fórmulas, que determina o resultado de algo que o usuário digita.

O operador de macro tem uma limitação: variáveis referenciadas dentro da string de caracteres (X nos exemplos anteriores) não podem ser locais.



**Anotações**

---

---

---

---

---

#### 4.5.4. Funções de manipulação de variáveis

Além de atribuir, controlar o escopo e a macro executar o conteúdo das variáveis, é necessário manipular seu conteúdo através de funções específicas da linguagem para cada situação.

As operações de manipulação de conteúdo mais comuns em programação são:

- Conversões entre tipos de variáveis.
- Manipulação de strings.
- Manipulação de variáveis numéricas.
- Verificação de tipos de variáveis.
- Manipulação de arrays.
- Execução de blocos de código.

Neste tópico serão abordadas as conversões entre tipos de variáveis e as funções de manipulação de strings e variáveis numéricas.

#### Conversões entre tipos de variáveis

As funções mais utilizadas nas operações entre conversão entre tipos de variáveis são:

- CTOD()**
- CVALTOCHAR()**
- DTOC()**
- DTOS()**
- STOD()**
- STR()**
- STRZERO()**
- VAL()**

## CTOD()

---

<b>Sintaxe</b>	<b>CTOD(cData)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo caractere no formato "DD/MM/AAAA", para uma variável do tipo data.

## CVALTOCHAR()

---

<b>Sintaxe</b>	<b>CVALTOCHAR(nValor)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo numérico em uma string, sem a adição de espaços a informação.

## DTOC()

---

<b>Sintaxe</b>	<b>DTOC(dData)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo data para em caractere, sendo o resultado no formato "DD/MM/AAAA".

## DTOS()

---

<b>Sintaxe</b>	<b>DTOS(dData)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo data em um caractere, sendo o resultado no formato "AAAAMMDD".

## STOD()

---

<b>Sintaxe</b>	<b>STOD(sData)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo caractere, com conteúdo no formato "AAAAMMDD", em data.

## STR()

---

<b>Sintaxe</b>	<b>STR(nValor)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo numérico em uma string, adicionando espaços à direita.

## STRZERO()

---

<b>Sintaxe</b>	<b>STRZERO(nValor, nTamanho)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo numérico em uma string, adicionando zeros à esquerda do número convertido, de forma que a string gerada tenha o tamanho especificado no parâmetro.

## VAL()

---

<b>Sintaxe</b>	<b>VAL(cValor)</b>
<b>Descrição</b>	Realiza a conversão de uma informação do tipo caractere em numérica.

## Manipulação de strings

---

As funções mais utilizadas nas operações de manipulação do conteúdo de strings são:

- ☐ **ALLTRIM()**
- ☐ **ASC()**
- ☐ **AT()**
- ☐ **CHR()**
- ☐ **CSTUFF()**
- ☐ **LEN()**
- ☐ **RAT()**
- ☐ **SUBSTR()**

## ALLTRIM()

---

<b>Sintaxe</b>	<b>ALLTRIM(cString)</b>
<b>Descrição</b>	Retorna uma string sem os espaços à direita e à esquerda, referente ao conteúdo informado como parâmetro.  A função ALLTRIM() implementa as ações das funções RTRIM ("right trim") e LTRIM ("left trim").

## ASC()

---

<b>Sintaxe</b>	<b>ASC(cCaractere)</b>
<b>Descrição</b>	Converte uma informação caractere em seu valor, de acordo com a tabela ASCII.

## AT()

---

<b>Sintaxe</b>	<b>AT(cCaractere, cString )</b>
<b>Descrição</b>	Retorna a primeira posição de um caractere ou string, dentro de outra string especificada.

## CHR()

---

<b>Sintaxe</b>	<b>CHR(nASCII)</b>
<b>Descrição</b>	Converte um valor número referente a uma informação da tabela ASCII, no caractere que esta informação representa.

## LEN()

---

<b>Sintaxe</b>	<b>LEN(cString)</b>
<b>Descrição</b>	Retorna o tamanho da string especificada no parâmetro.

## LOWER()

---

<b>Sintaxe</b>	<b>LOWER(cString)</b>
<b>Descrição</b>	Retorna uma string com todos os caracteres minúsculos, tendo como base a string passada como parâmetro.

## RAT()

---

<b>Sintaxe</b>	<b>RAT(cCaractere, cString)</b>
<b>Descrição</b>	Retorna a última posição de um caractere ou string, dentro de outra string especificada.

## STUFF()

---

<b>Sintaxe</b>	<b>STUFF(cString, nPosInicial, nExcluir, cAdicao)</b>
<b>Descrição</b>	Permite substituir um conteúdo caractere em uma string já existente, especificando a posição inicial para esta adição e o número de caracteres que serão substituídos.

## SUBSTR()

---

<b>Sintaxe</b>	<b>SUBSTR(cString, nPosInicial, nCaracteres)</b>
<b>Descrição</b>	Retorna parte do conteúdo de uma string especificada, de acordo com a posição inicial deste conteúdo na string e a quantidade de caracteres que deverá ser retornada a partir daquele ponto (inclusive).

## UPPER()

---

<b>Sintaxe</b>	<b>UPPER(cString)</b>
<b>Descrição</b>	Retorna uma string com todos os caracteres maiúsculos, tendo como base a string passada como parâmetro.

## Manipulação de variáveis numéricas

---

As funções mais utilizadas nas operações de manipulação do conteúdo de strings são:

- ☐ **ABS()**
- ☐ **INT()**
- ☐ **NOROUND()**
- ☐ **ROUND()**

## ABS()

---

<b>Sintaxe</b>	<b>ABS(nValor)</b>
<b>Descrição</b>	Retorna um valor absoluto (independente do sinal) com base no valor especificado no parâmetro.

## INT()

---

<b>Sintaxe</b>	<b>INT(nValor)</b>
<b>Descrição</b>	Retorna a parte inteira de um valor especificado no parâmetro.

## NOROUND()

---

<b>Sintaxe</b>	<b>NOROUND(nValor, nCasas)</b>
<b>Descrição</b>	Retorna um valor, truncando a parte decimal do valor especificado no parâmetro de acordo com a quantidade de casas decimais solicitadas.

## ROUND()

---

<b>Sintaxe</b>	<b>ROUND(nValor, nCasas)</b>
<b>Descrição</b>	Retorna um valor, arredondando a parte decimal do valor especificado no parâmetro, de acordo com as quantidades de casas decimais solicitadas, utilizando o critério matemático.

## Verificação de tipos de variáveis

As funções de verificação permitem a consulta ao tipo do conteúdo da variável, durante a execução do programa.

 **TYPE()**

 **VALTYPE()**

### TYPE()

---

<b>Sintaxe</b>	<b>TYPE("cVariavel")</b>
<b>Descrição</b>	Determina o tipo do conteúdo de uma variável que não foi definida na função em execução.

### VALTYPE()

---

<b>Sintaxe</b>	<b>VALTYPE(cVariável)</b>
<b>Descrição</b>	Determina o tipo do conteúdo de uma variável que foi definida na função em execução.



*Anotações* \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## 5. ESTRUTURAS BÁSICAS DE PROGRAMAÇÃO

O ADVPL suporta várias estruturas de controle que permitem mudar a sequência de fluxo de execução de um programa. Essas estruturas permitem a execução de código baseado em condições lógicas e a repetição da execução de pedaços de código em qualquer número de vezes.

Em ADVPL, todas as estruturas de controle podem ser "aninhadas" dentro de todas as demais estruturas, contanto que estejam aninhadas propriamente. Estruturas de controle têm um identificador de início e um de fim, e qualquer estrutura aninhada deve se encontrar entre estes identificadores.

Também existem estruturas de controle para determinar que elementos, comandos, etc. em um programa serão compilados. Estas são as diretivas do pré-processador `#ifdef...#endif` e `#ifndef...#endif`. Consulte a documentação sobre o pré-processador para maiores detalhes.

As estruturas de controle em ADVPL estão divididas em:

- Estruturas de repetição.
- Estruturas de decisão.

### 5.1. Estruturas de repetição

Estruturas de repetição são designadas para executar uma seção de código mais de uma vez. Por exemplo, imaginando-se a existência de uma função para imprimir um relatório, digamos que é preciso imprimi-lo quatro vezes. É possível simplesmente chamar a função de impressão quatro vezes em sequência, mas isto se tornaria pouco profissional e não resolveria o problema se o número de relatórios fosse variável.

Em ADVPL existem dois comandos para a repetição de seções de código, que são os comandos **FOR...NEXT** e o comando **WHILE...ENDDO**.

#### O Comando **FOR...NEXT**

A estrutura de controle **FOR...NEXT**, ou simplesmente o loop **FOR**, repete uma seção de código em um número determinado de vezes.

**Sintaxe**

FOR Variável := nValorInicial TO nValorFinal [STEP nIncremento] Comandos... [EXIT] [LOOP] NEXT
--

**Parâmetros**

<b>Variável</b>	Especifica uma variável, ou um elemento de uma matriz para atuar como um contador. A variável, ou o elemento da matriz não precisa ter sido declarado antes da execução do comando FOR...NEXT. Se a variável não existir, será criada como uma <b><u>variável privada</u></b> .
<b>nValorInicial TO nValorFinal</b>	nValorInicial é o valor inicial para o contador; nValorFinal é o valor final para o contador. É possível utilizar valores numéricos literais, variáveis ou expressões, contanto que o resultado seja do tipo dado numérico.
<b>STEP nIncremento</b>	nIncremento é a quantidade que será incrementada, ou decrementada no contador após cada execução da seção de comandos. Se o valor de nIncremento for negativo, o contador será decrementado. Se a cláusula STEP for omitida, o contador será incrementado em 1. É possível utilizar os valores numéricos literais, variáveis ou expressões, contanto que o resultado seja do tipo de dado numérico.
<b>Comandos</b>	Especifica uma, ou mais instruções de comando ADVPL que serão executadas.
<b>EXIT</b>	Transfere o controle de dentro do comando FOR...NEXT para o comando imediatamente seguinte ao NEXT. Ou seja, finaliza a repetição da seção de comandos imediatamente. É possível adicionar o comando EXIT em qualquer lugar entre o FOR e o NEXT.
<b>LOOP</b>	Retorna o controle diretamente para a cláusula FOR sem executar o restante dos comandos entre o LOOP e o NEXT. O contador é incrementado ou decrementado normalmente, como se o NEXT tivesse sido alcançado. É possível posicionar o comando LOOP em qualquer lugar, entre o FOR e o NEXT.



Fique atento

**Uma variável, ou um elemento de uma matriz é utilizado como um contador para especificar quantas vezes os comandos ADVPL dentro da estrutura FOR...NEXT são executados.**

**Os comandos ADVPL depois do FOR são executados até que o NEXT seja alcançado. O contador (Variável) é então incrementado, ou decrementado com o valor em nIncremento (se a cláusula STEP for omitida, o contador é incrementado em 1). Então, o contador é comparado com o valor em nValorFinal. Se for menor ou igual ao valor em nValorFinal, os comandos seguintes ao FOR são executados novamente.**

**Se o valor for maior que o contido em nValorFinal, a estrutura FOR...NEXT é terminada e o programa continua a execução no primeiro comando após o NEXT.**

**Os valores de nValorInicial, nValorFinal e nIncremento são apenas considerados inicialmente. Entretanto, mudar o valor da variável utilizada como contador dentro da estrutura afetará o número de vezes que a repetição será executada. Se o valor de nIncremento é negativo e o valor de nValorInicial é maior que o de nValorFinal, o contador será decrementado a cada repetição.**

**Exemplo:**

```
Local nCnt
Local nSomaPar := 0
For nCnt := 0 To 100 Step 2
nSomaPar += nCnt
Next
Alert( "A soma dos 100 primeiros números pares é: " + ;
      cValToChar(nSomaPar) )
Return
```

Esse exemplo imprime a soma dos 100 primeiros números pares. A soma é obtida através da repetição do cálculo, utilizando a própria variável de contador. Como a cláusula STEP está sendo utilizada, a variável nCnt será sempre incrementada em 2. E como o contador começa com 0, seu valor sempre será um número par.

## O Comando WHILE...ENDDO

---

A estrutura de controle WHILE...ENDDO, ou simplesmente o loop WHILE, repete uma seção de código enquanto uma determinada expressão resultar em verdadeiro (.T.).

### Sintaxe

```
WHILE lExpressao
  Comandos...
[EXIT]
[LOOP]
ENDDO
```

### Parâmetros

<b>lExpressao</b>	Especifica uma expressão lógica cujo valor determina quando os comandos entre o WHILE e o ENDDO são executados. Enquanto o resultado de lExpressao for avaliado como verdadeiro (.T.), o conjunto de comandos são executados.
<b>Comandos</b>	Especifica uma ou mais instruções de comando ADVPL, que serão executadas enquanto lExpressao for avaliado como verdadeiro (.T.).
<b>EXIT</b>	Transfere o controle de dentro do comando WHILE...ENDDO para o comando imediatamente seguinte ao ENDDO, ou seja, finaliza a repetição da seção de comandos imediatamente. É possível posicionar o comando EXIT em qualquer lugar entre o WHILE e o ENDDO.
<b>LOOP</b>	Retorna o controle diretamente para a cláusula WHILE sem executar o restante dos comandos entre o LOOP e o ENDDO. A expressão em lExpressao é reavaliada para a decisão se os comandos continuarem sendo executados.

**Exemplo :**

```
Local nNumber := nAux := 350
nAux := Int(nAux / 2)
While nAux > 0
nSomaPar += nCnt
Next
Alert( "A soma dos 100 primeiros números pares é: " + ;
      cValToChar(nSomaPar) )
Return
```



Fique  
atento

***Os comandos entre o WHILE e o ENDDO são executados enquanto o resultado da avaliação da expressão em IExpressao permanecer verdadeiro (.T.). Cada palavra chave WHILE deve ter uma palavra chave ENDDO correspondente.***

### 5.1.1. Influenciando o fluxo de repetição

A linguagem ADVPL permite a utilização de comandos que influem diretamente em um processo de repetição, sendo eles:

- ☑ **LOOP**
- ☑ **EXIT**

#### LOOP

---

A instrução LOOP é utilizada para forçar um desvio no fluxo do programa de volta a análise da condição de repetição. Dessa forma, todas as operações, que seriam realizadas dentro da estrutura de repetição após o LOOP, serão desconsideradas.

#### Exemplo:

```
aItens:= ListaProdutos() // função ilustrativa que retorna um array com dados dos produtos.  
nQuantidade := Len(aItens)  
nItens := 0
```

```
While nItens < nQuantidade
```

```
    nItens++
```

```
    IF BLOQUEADO(aItens [nItens]) // função ilustrativa que verifica se o produto está
```

```
        LOOP                // bloqueado.
```

```
    ENDIF
```

```
    IMPRIME() // função ilustrativa que realiza a impressão de um item liberado para uso.
```

```
End
```

**// Se o produto estiver bloqueado, o mesmo não será impresso, pois a execução da // instrução LOOP fará o fluxo do programa retornar a partir da análise da condição.**

## EXIT

---

A instrução EXIT é utilizada para forçar o término de uma estrutura de repetição. Dessa forma, todas as operações que seriam realizadas dentro da estrutura de repetição após o EXIT serão desconsideradas, e o programa continuará a execução a partir da próxima instrução posterior ao término da estrutura (END ou NEXT).

### Exemplo:

While .T.

```
IF MSGYESNO("Deseja jogar o jogo da forca?")
```

```
    JFORCA() // Função ilustrativa que implementa o algoritmo do jogo da forca.
```

```
ELSE
```

```
    EXIT
```

```
ENDIF
```

End

```
MSGINFO("Final de Jogo")
```

**// Enquanto não for respondido "Não" para a pergunta: "Deseja jogar o jogo da forca", será executada a função do jogo da forca.**

**// Caso seja selecionada a opção "Não", será executada a instrução EXIT que provocará o término do LOOP, permitindo a execução da mensagem de "Final de Jogo".**



Anotações \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## 5.2. Estruturas de decisão

As Estruturas de desvio são designadas para executar uma seção de código se determinada condição lógica resultar em verdadeiro (.T.).

Em ADVPL existem dois comandos para execução de seções de código, de acordo com as avaliações lógicas, que são os comandos **IF...ELSE...ENDIF** e o comando **DO CASE...ENDCASE**.

### O Comando IF...ELSE...ENDIF

Executa um conjunto de comandos baseado no valor de uma expressão lógica.

**Sintaxe**

IF *lExpressao*

Comandos

[ELSE

Comandos...]

ENDIF

**Parâmetros**

<b>LExpressao</b>	Especifica uma expressão lógica que é avaliada. Se lExpressao resultar em verdadeiro (.T.), qualquer comando seguinte ao IF e antecedente ao ELSE ou ENDIF (o que ocorrer primeiro) será executado.  Se lExpressao resultar em falso (.F.) e a cláusula ELSE for definida, qualquer comando após essa cláusula e anterior ao ENDIF será executada. Se a cláusula ELSE não for definida, todos os comandos entre o IF e o ENDIF serão ignorados. Nesse caso, a execução do programa continua com o primeiro comando seguinte ao ENDIF.
<b>Comandos</b>	Conjunto de comandos ADVPL que serão executados dependendo da avaliação da expressão lógica em lExpressao.



Fique  
atento

*É possível aninhar um bloco de comando IF...ELSE...ENDIF dentro de outro bloco de comando IF...ELSE...ENDIF. Porém, para a avaliação de mais de uma expressão lógica, deve-se utilizar o comando DO CASE...ENDCASE, ou a versão estendida da expressão IF...ELSE...ENDIF denominada IF...ELSEIF...ELSE...ENDIF.*

**Exemplo:**

```
Local dVencto := CTOD("31/12/01")
If Date() > dVencto
Alert("Vencimento ultrapassado!")
Endif
Return
```

## O Comando IF...ELSEIF...ELSE...ENDIF

---

Executa o primeiro conjunto de comandos cuja expressão condicional resulta em verdadeiro (.T.).

**Sintaxe**

```
IF IExpressao1
Comandos
[ELSEIF IExpressaoX
Comandos]
[ELSE
Comandos...]
ENDIF
```

**Parâmetros**

<b>IExpressao1</b>	<p>Especifica uma expressão lógica que é avaliada. Se IExpressao resultar em verdadeiro (.T.), executará os comandos compreendidos entre o IF e a próxima expressão da estrutura (ELSEIF ou IF).</p> <p>Se IExpressao resultar em falso (.F.), será avaliada a próxima expressão lógica vinculada ao comando ELSEIF, ou se o mesmo não existir, será executada a ação definida no comando ELSE.</p>
<b>IExpressaoX</b>	<p>Especifica uma expressão lógica que será avaliada para cada comando ELSEIF. Esta expressão somente será avaliada se a expressão lógica, especificada no comando IF, resultar em falso (.F.).</p> <p>Se a IExpressaoX avaliada resulte em falso (.F.) será avaliada a próxima expressão IExpressaoX, vinculada ao próximo comando ELSEIF, ou se o mesmo não existir será executada a ação definida para o comando ELSE.</p>
<b>Comandos</b>	<p>Conjunto de comandos ADVPL que serão executados, dependendo da avaliação da expressão lógica em IExpressao.</p>



*Fique  
atento*

***O campo IF...ELSE...ELSEIF...ENDIF possui a mesma estruturação de decisão que pode ser obtida com a utilização do comando DO CASE...ENDCASE.***

**Exemplo:**

```
Local dVencto := CTOD("31/12/01")
If Date() > dVencto
Alert("Vencimento ultrapassado!")
Elseif Date() == dVencto
Alert("Vencimento na data!")
Else
Alert("Vencimento dentro do prazo!")
Endif
Return
```

**O Comando DO CASE...ENDCASE**

---

Executa o primeiro conjunto de comandos cuja expressão condicional resulta em verdadeiro (.T.).

 **Sintaxe**

```
DO CASE
CASE IExpressao1
  Comandos
[CASE IExpressao2
  Comandos
...
CASE IExpressaoN
  Comandos]
[OTHERWISE
  Comandos]
ENDCASE
```

**Parâmetros**

<b>CASE</b> <b>!Expressao1</b> <b>Comandos...</b>	<p>Quando a primeira expressão CASE, resultante em verdadeiro (.T.), for encontrada, o conjunto de comandos seguinte é executado. A execução do conjunto de comandos continua até que a próxima cláusula CASE, OTHERWISE ou ENDCASE for encontrada. Ao terminar de executar esse conjunto de comandos, a execução continua com o primeiro comando seguinte ao ENDCASE.</p> <p>Se uma expressão CASE resultar em falso (.F.), o conjunto de comandos seguinte a esta até a próxima cláusula é ignorado.</p> <p>Apenas um conjunto de comandos é executado. Estes são os primeiros comandos cuja expressão CASE é avaliada como verdadeiro (.T.). Após a execução, qualquer outra expressão CASE posterior é ignorada (mesmo que sua avaliação resultasse em verdadeiro).</p>
<b>OTHERWISE</b> <b>Comandos</b>	<p>Se todas as expressões CASE forem avaliadas como falso (.F.), a cláusula OTHERWISE determina se um conjunto adicional de comandos deve ser executado. Se essa cláusula for incluída, os comandos seguintes serão executados e então o programa continuará com o primeiro comando seguinte ao ENDCASE. Se a cláusula OTHERWISE for omitida, a execução continuará normalmente após a cláusula ENDCASE.</p>



Fique  
atento

**O Comando DO CASE...ENDCASE é utilizado no lugar do comando IF...ENDIF, quando um número maior do que uma expressão deve ser avaliada, substituindo a necessidade de mais de um comando IF...ENDIF aninhados.**

**Exemplo:**

```
Local nMes := Month(Date())
Local cPeriodo := ""

DO CASE
CASE nMes <= 3
cPeriodo := "Primeiro Trimestre"
CASE nMes >= 4 .And. nMes <= 6
cPeriodo := "Segundo Trimestre"
CASE nMes >= 7 .And. nMes <= 9
cPeriodo := "Terceiro Trimestre"
OTHERWISE
cPeriodo := "Quarto Trimestre"
ENDCASE

Return
```



### **Exercícios**

#### Exercício 04

Desenvolver um programa que implemente o algoritmo de descascar batatas, utilizando a estrutura de repetição FOR, demonstrando quantas batatas foram descascadas:



### **Exercícios**

#### Exercício 05

Desenvolver um programa que implemente o algoritmo de descascar batatas, utilizando a estrutura de repetição FOR, demonstrando quantas batatas faltam para serem descascadas:



### **Exercícios**

#### Exercício 06

Desenvolver um programa que implemente o algoritmo do Jogo da Forca:



### **Exercícios**

#### Exercício 07

Desenvolver um programa que implemente o algoritmo do Jogo da Velha:



**Anotações**

---

---

---

---

---

## 6. ARRAYS E BLOCOS DE CÓDIGO

### 6.1. Arrays

Arrays , ou matrizes são coleções de valores, semelhantes a uma lista. Uma matriz pode ser criada através de diferentes maneiras. Cada item em um array é referenciado pela indicação de sua posição numérica na lista, iniciando pelo número 1.

O exemplo a seguir declara uma variável, atribui um array de três elementos a ela, e então exibe um dos elementos e o tamanho do array:

```
Local aLetras      // Declaração da variável
aLetras := {"A", "B", "C"} // Atribuição do array a variável
Alert(aLetras[2])  // Exibe o segundo elemento do array
Alert(cValToChar(Len(aLetras))) // Exibe o tamanho do array
```

O ADVPL permite a manipulação de arrays facilmente. Enquanto que em outras linguagens como C ou Pascal é necessário alocar memória para cada elemento de um array (o que tornaria a utilização de "ponteiros" necessária), o ADVPL se encarrega de gerenciar a memória e torna simples adicionar elementos a um array, utilizando a função AADD():

```
AADD(aLetras,"D") // Adiciona o quarto elemento ao final do array.
Alert(aLetras[4]) // Exibe o quarto elemento.
Alert(aLetras[5]) // Erro! Não há um quinto elemento no array.
```

### Arrays como Estruturas

Uma característica interessante do ADVPL é que um array pode conter qualquer tipo de dado: números, datas, lógicos, caracteres, objetos, etc., e ao mesmo tempo. Em outras palavras, os elementos de um array não precisam ser necessariamente do mesmo tipo de dado, em contraste com outras linguagens como C e Pascal.

```
aFunct1 := {"Pedro",32,.T.}
```

Este array contém uma string, um número e um valor lógico. Em outras linguagens como C ou Pascal, este "pacote" de informações pode ser chamado como um "struct" (estrutura em C, por exemplo) ou um "record" (registro em Pascal, por exemplo). Como se fosse na verdade um registro de um banco de dados, um pacote de informações construído com diversos campos. Cada campo tendo uma parte diferente de um dado.

Suponha que no exemplo anterior, o array aFunc1 contenha informações sobre o nome de uma pessoa, sua idade e sua situação matrimonial. Os seguintes #defines podem ser criados para indicar cada posição dos valores dentro de um array:

```
#define FUNCT_NOME 1  
#define FUNCT_IDADE 2  
#define FUNCT_CASADO 3
```

E considere mais alguns arrays para representar mais pessoas:

```
aFunct2 := {"Maria" , 22, .T.}
aFunct3 := {"Antônio", 42, .F.}
```

Os nomes podem ser impressos assim:

```
Alert(aFunct1[FUNCT_NOME])
Alert(aFunct2[FUNCT_NOME])
Alert(aFunct3[FUNCT_NOME])
```

Agora, ao invés de trabalhar com variáveis individuais, é possível agrupá-las em um outro array, do mesmo modo que muitos registros são agrupados em uma tabela de banco de dados:

```
aFuncs := {aFunct1, aFunct2, aFunct3}
```

Que é equivalente a isso:

```
aFuncs := { {"Pedro" , 32, .T.}, ;
            {"Maria" , 22, .T.}, ;
            {"Antônio", 42, .F.} }
```

*aFuncs* é um array com 3 linhas por 3 colunas. Uma vez que as variáveis separadas foram combinadas em um array, os nomes podem ser exibidos assim:

```
Local nCount
For nCount := 1 To Len(aFuncs)
Alert(aFuncs[nCount, FUNCT_NOME])
// O acesso a elementos de um array multidimensional.
// pode ser realizado também desta forma:
// aFuncs[nCount][FUNCT_NOME]
Next nCount
```

A variável nCount seleciona que funcionário (ou que linha) é de interesse. Então a constante FUNCT\_NOME seleciona a primeira coluna daquela linha.

## **Cuidados com Arrays**

---

Arrays são listas de elementos, portanto memória é necessária para armazenar estas informações. Como estes arrays podem ser multidimensionais, a memória necessária será a multiplicação do número de itens em cada dimensão do array, considerando-se o tamanho do conteúdo de cada elemento contido nesta. Portanto o tamanho de um array pode variar muito.

A facilidade da utilização de arrays, mesmo que para armazenar informações em pacotes como descrito anteriormente, não é compensada pela utilização em memória quando o número de itens em um array for muito grande. Quando o número de elementos for muito grande deve-se procurar outras soluções, como a utilização de um arquivo de banco de dados temporário.

### 6.1.1. Inicializando arrays

Algumas vezes o tamanho da matriz é conhecido previamente. Outras vezes, o tamanho do array somente será conhecido em tempo de execução.

#### Se o tamanho do array é conhecido

---

Se o tamanho do array é conhecido no momento que o programa é escrito, há diversas maneiras de implementar o código:

```
01 Local nCnt
02 Local aX[10]
03 Local aY := Array(10)
04 Local aZ := {0,0,0,0,0,0,0,0,0,0}
05
06 For nCnt := 1 To 10
07   aX[nCnt] := nCnt * nCnt
08 Next nCnt
```

Este código preenche o array com uma tabela de quadrados. Os valores serão 1, 4, 9, 16 ... 81, 100. Note que a linha 07 se refere à variável aX, mas poderia também trabalhar com aY ou aZ.

O objetivo deste exemplo é demonstrar três modos de criar um array de tamanho conhecido, no momento da criação do código.

1. Na linha 02 o array é criado usando aX[10]. Isto indica ao ADVPL para alocar espaço para 10 elementos no array. Os colchetes [ e ] são utilizados para indicar o tamanho necessário.

2. Na linha 03 é utilizada a função array com o parâmetro 10 para criar o array, e o retorno desta função é atribuído à variável aY. Na linha 03 é efetuado o que se chama "desenhar a imagen do array". Como se pode notar, existem dez 0's na lista encerrada entre chaves ({}). Claramente, este método não é o utilizado para criar uma matriz de 1000 elementos.

3. O terceiro método difere dos anteriores porque inicializa a matriz com os valores definitivos. Nos dois primeiros métodos, cada posição da matriz contém um valor nulo (Nil) e deve ser inicializado posteriormente.

4. A linha 07 demonstra como um valor pode ser atribuído para uma posição existente em uma matriz, especificando o índice entre colchetes.

## Se o tamanho do array não é conhecido

---

Se o tamanho do array não é conhecido até o momento da execução do programa, há algumas maneiras de criar um array e adicionar elementos a ele. O exemplo a seguir ilustra a idéia da criação de um array vazio (sem nenhum elemento), e adição de elementos dinamicamente.

```
01 Local nCnt
02 Local aX[0]
03 Local aY := Array(0)
04 Local aZ := {}
05
06 For nCnt := 1 To nSize
07   AADD(aX, nCnt*nCnt)
08 Next nCnt
```

1. A linha 02 utiliza os colchetes para criar um array vazio. Apesar de não ter nenhum elemento, seu tipo de dado é array.
2. Na linha 03 a chamada da função array cria uma matriz sem nenhum elemento.
3. Na linha 04 está declarada a representação de um array vazio em ADVPL. Mais uma vez, estão sendo utilizadas as chaves para indicar que o tipo de dados da variável é array. Note que {} é um array vazio (tem o tamanho 0), enquanto {Nil} é um array com um único elemento nulo (tem tamanho 1).

Porque cada uma destes arrays não contém elementos, a linha 07 utiliza a função AADD() para adicionar elementos sucessivamente até o tamanho necessário (especificado por exemplo na variável nSize).

## 6.1.2. Funções de manipulação de arrays

A linguagem ADVPL possui diversas funções que auxiliam na manipulação de arrays, dentre as quais podemos citar as mais utilizadas:

- ARRAY()**
- AADD()**
- ACLONE()**
- ADEL()**
- ASIZE()**
- AINS()**
- ASORT()**
- ASCAN()**

## ARRAY()

---

<b>Sintaxe</b>	<b>ARRAY(nLinhas, nColunas)</b>
<b>Descrição</b>	A função Array() é utilizada na definição de variáveis de tipo array, como uma opção a sintaxe, utilizando chaves ("{}").

## AADD()

---

<b>Sintaxe</b>	<b>AADD(aArray, xItem)</b>
<b>Descrição</b>	A função AADD() permite a inserção de um item em um array já existente, sendo que este item pode ser um elemento simples, um objeto, ou outro array.

## ACLONE()

---

<b>Sintaxe</b>	<b>AADD(aArray)</b>
<b>Descrição</b>	A função ACLONE() realiza a cópia dos elementos de um array para outro array integralmente.

## ADEL()

---

<b>Sintaxe</b>	<b>ADEL(aArray, nPosição)</b>
<b>Descrição</b>	A função ADEL() permite a exclusão de um elemento do array. Ao efetuar a exclusão de um elemento, todos os demais são reorganizados de forma que a última posição do array passará a ser nula.

## ASIZE()

---

<b>Sintaxe</b>	<b>ASIZE(aArray, nTamanho)</b>
<b>Descrição</b>	A função ASIZE permite a redefinição da estrutura de um array pré-existente,

	adicionando ou removendo itens do mesmo.
--	--

## **ASORT()**

---

<b>Sintaxe</b>	<b>ASORT(aArray, nInicio, nItens, bOrdem)</b>
<b>Descrição</b>	A função ASORT() permite que os itens de um array sejam ordenados, a partir de um critério pré-estabelecido.

## ASCAN()

---

<b>Sintaxe</b>	<b>ASCAN(aArray, bSeek)</b>
<b>Descrição</b>	A função ASCAN() permite que seja identificada a posição do array que contém uma determinada informação, através da análise de uma expressão descrita em um bloco de código.

## AINS()

---

<b>Sintaxe</b>	<b>AINS(aArray, nPosicao)</b>
<b>Descrição</b>	A função AINS() permite a inserção de um elemento, no array especificado, em qualquer ponto da estrutura do mesmo, diferindo desta forma da função AADD(), a qual sempre insere um novo elemento ao final da estrutura já existente.

### 6.1.3. Cópia de arrays

Conforme comentado anteriormente, um array é uma área na memória, o qual possui uma estrutura que permite as informações serem armazenadas e organizadas das mais diversas formas.

Com base nesse conceito, o array pode ser considerado apenas como um “mapa” ou um “guia” de como as informações estão organizadas e de como elas podem ser armazenadas ou consultadas. Para se copiar um array deve-se levar este conceito em consideração, pois caso contrário o resultado esperado não será obtido na execução da “cópia”.

Para "copiar" o conteúdo de uma variável, utiliza-se o operador de atribuição ":=", conforme abaixo:

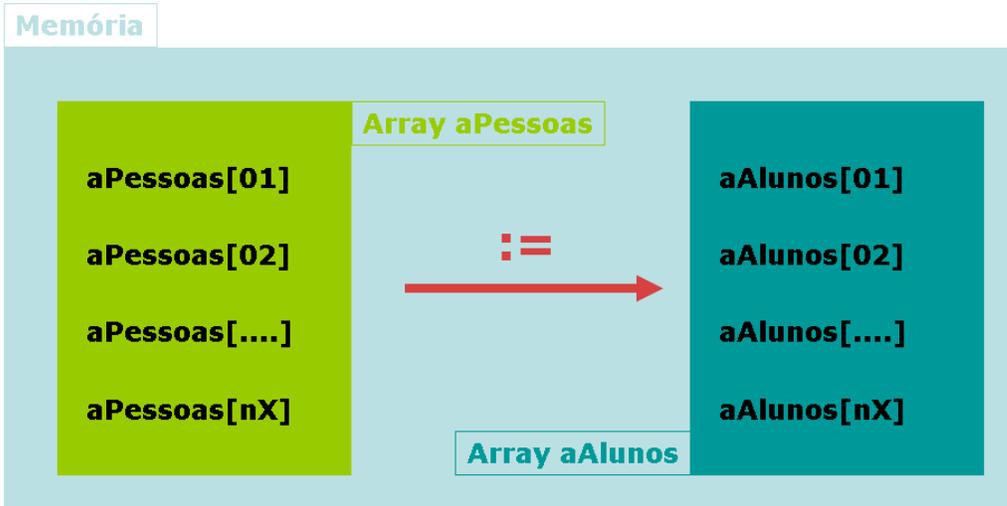
```
nPessoas := 10  
nAlunos := nPessoas
```

Ao executar a atribuição de nAlunos com o conteúdo de nPessoas, o conteúdo de nPessoas é atribuído a variável nAlunos, causando o efeito de cópia do conteúdo de uma variável para outra.

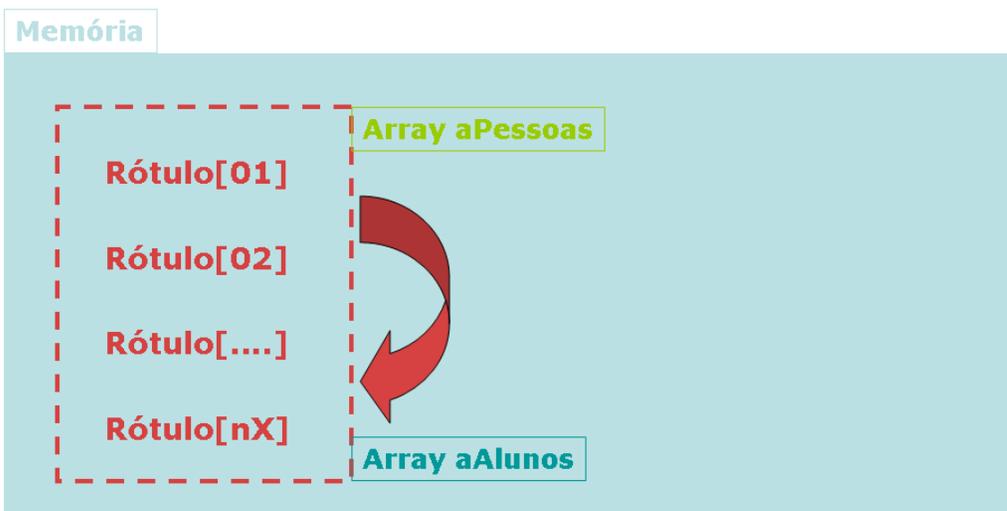
Isto porque o comando de atribuição copia o conteúdo da área de memória, representada pelo nome "nPessoas" para a área de memória representada pelo nome "nAlunos". Mas ao utilizar o operador de atribuição ":=", da mesma forma que utilizado em variáveis simples, para se copiar um array o efeito é diferente:

```
aPessoas := {"Ricardo", "Cristiane", "André", "Camila"}  
aAlunos := aPessoas
```

A variável aPessoas representa uma área de memória que contém a estrutura de um array ("mapa"), não as informações do array, pois cada informação está em sua própria área de memória.



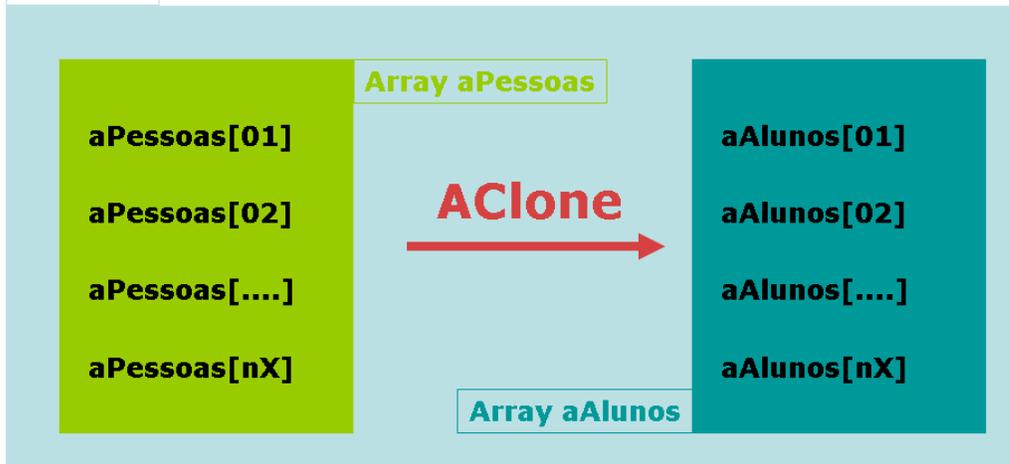
Desta forma, ao atribuir o conteúdo representado pela variável `aPessoas`, a variável `aAlunos` não está se “copiando” as informações e sim o “mapa” das áreas de memória, em que as informações estão realmente armazenadas.



Como foi copiado o “mapa” e não as informações, qualquer ação utilizando o rótulo `aAlunos` afetará as informações do rótulo `aPessoas`. Com isso ao invés de se obter dois arrays distintos, tem-se o mesmo array com duas formas de acesso (rótulos) diferentes.

Por esta razão deve ser utilizado o comando `ACLONE()`, quando se deseja obter um array com a mesma estrutura e informações que compõe outro array já existente.

Memória



## 6.2. Listas de Expressões e Blocos de Código

Bloco de código é um conceito existente há muito tempo em linguagens xBase. Não como algo que apareceu da noite para o dia, e sim uma evolução progressiva utilizando a combinação de muitos conceitos da linguagem para a sua implementação.

### 6.2.1. Premissas para utilização de Blocos de Código

#### Primeira premissa

O ADVPL é uma linguagem baseada em funções. Funções têm um valor de retorno. Assim como o operador de atribuição :=.

#### Assim, ao invés de escrever:

```
x := 10 // Atribui o valor 10 à variável chamada X
Alert("Valor de x: " + cValToChar(x))
```

#### Escreve-se:

```
// Atribui e então exibe o valor da variável X
Alert("Valor de x: " + cValtoChar(X := 10))
```

A expressão `x:=10` é avaliada primeiro, e então seu resultado (o valor de X, que agora é 10) é passada para a função `cvaltochar` para a conversão para caractere, e em seguida para a função `alert` para a exibição. Por causa dessa regra de precedência é possível atribuir um valor a mais de uma variável ao mesmo tempo:

```
Z := Y := X := 0
```

Por causa dessa regra, essa expressão é avaliada como se fosse escrita assim:

```
Z := ( Y := ( X := 0 ) )
```

Apesar de o ADVPL avaliar expressões da esquerda para a direita, no caso de atribuições, acontece ao contrário, ou seja, da direita para a esquerda. O valor é atribuído à variável X, que retorna o valor para ser atribuído à variável Y e assim sucessivamente. O zero foi "propagado através da expressão".

## Segunda premissa

Em ADVPL junta-se diversas linhas de código em uma única linha física de comando. Por exemplo, o código:

```
If IAchou
Alert("Cliente encontrado!")
Endif
pode ser escrito assim:
If IAchou ; Alert("Cliente encontrado!") ;
Endif
```

O ponto-e-vírgula indica ao ADVPL que a nova linha de código está para começar. É possível então colocar diversas linhas lógicas de código na mesma linha física, através do editor de texto utilizado.

Apesar da possibilidade de se escrever todo o programa assim, em uma única linha física, isto não é recomendado, pois dificulta a legibilidade do programa e, conseqüentemente, a sua manutenção.

## 6.2.2. Lista de expressões

A evolução dos blocos de código começa com as listas de expressões. Nos exemplos a seguir, o símbolo ==> indicará o retorno da expressão após a sua avaliação (seja para atribuir em uma variável, exibir para o usuário ou imprimir em um relatório), que será impressa em um relatório, por exemplo.

### Duas Linhas de Código

```
@00,00 PSAY x := 10 ==> 10
@00,00 PSAY y := 20 ==> 20
```

Cada uma das linhas terá a expressão avaliada, e o valor da variável será então impresso.

### Duas linhas de código em uma , utilizando -se ponto-e-vírgula

Este é o mesmo código que o anterior, apenas escrito em uma única linha:

```
Alert( cValToChar( x := 10 ; y := 20 ) ) ==> 10
```

Apesar desse código se encontrar em uma única linha física, existem duas linhas lógicas separadas pelo ponto e vírgula. Ou seja, esse código é equivalente a:

```
Alert( cValToChar( x := 10 ) )
y := 20
```

Portanto, apenas o valor 10 da variável x será passado para as funções cvaltochar e alert para ser exibido. E o valor 20 apenas será atribuído à variável y.

## **Convertendo para uma lista de expressões**

---

Quando parênteses são colocados ao redor do código e o sinal de ponto-e-vírgula substituído por uma vírgula apenas, o código torna-se uma lista de expressões:

```
Alert( cValToChar ( ( X := 10 , Y := 20 ) ) ) ==> 20
```

O valor de retorno resultante de uma lista de expressões é o valor resultante da última expressão ou elemento da lista. Funciona como se fosse um pequeno programa ou função, que retorna o resultado de sua última avaliação (efetuadas da esquerda para a direita).

Nesse exemplo, a expressão `x := 10` é avaliada, e então a expressão `y := 20`, cujo valor resultante é passado para a função `alert` e `cvaltochar`, e então exibido. Depois que essa linha de código é executada, o valor de `X` é igual a 10 e o de `y` igual a 20, e 20 será exibido.

Teoricamente, não há limitação para o número de expressões que podem ser combinadas em uma lista. Na prática, o número máximo é por volta de 500 símbolos.

Debugar listas de expressões é difícil porque as expressões não estão divididas em linhas de código fonte, o que torna todas as expressões associadas a uma mesma linha de código. Isto pode tornar muito difícil determinar onde um erro ocorreu.

## **Onde utilizar uma lista de expressões?**

---

O propósito principal de uma lista de expressões é agrupá-las em uma única unidade. Em qualquer lugar do código ADVPL, em que uma expressão simples pode ser utilizada, pode-se utilizar uma lista de expressões. E ainda, fazer com que várias coisas aconteçam onde normalmente apenas uma aconteceria.

```
X := 10 ; Y := 20
If X > Y
Alert("X")
Z := 1
Else
Alert("Y")
Z := -1
Endif
```

Aqui temos o mesmo conceito, escrito utilizando listas de expressões na função IIF():

```
X := 10 ; Y := 20
iif( X > Y , ;
( Alert("X"), Z := 1 ) , ;
( Alert("Y"), Z := -1 ) )
```

### **De listas de expressões para blocos de código**

Considere a seguinte lista de expressões:

```
Alert( cValToChar( ( x := 10, y := 20 ) ) ) ==> 20
```

O ADVPL permite criar funções, que são pequenas partes de um código, como se fosse um pequeno programa, utilizados para diminuir em partes de tarefas mais complexas e reaproveitar código em mais de um lugar num programa. Para maiores detalhes consulte a documentação sobre a criação de funções em ADVPL. Porém, a idéia neste momento é que a lista de expressões, utilizada na linha anterior, pode ser criada como uma função:

```
Function Lista()
X := 10
Y := 20
Return Y
```

E a linha de exemplo, com a lista de expressões, pode ser substituída, tendo o mesmo resultado, por:

```
Alert( cValToChar( Lista() ) ) ==> 20
```

Como mencionado anteriormente, uma lista de expressões é como um pequeno programa ou função. Com poucas mudanças, uma lista de expressões pode se tornar um bloco de código:

```
( X := 10 , Y := 20 ) // Lista de Expressões
{|| X := 10 , Y := 20 } // Bloco de Código
```

Note as chaves {} utilizadas no bloco de código. Ou seja, um bloco de código é uma matriz. Porém na verdade, não é uma lista de dados, e sim uma lista de comandos, uma lista de código.

```
// Isto é uma matriz de dados  
A := {10, 20, 30}  
// Isto é um bloco de código, porém funciona como // se fosse uma matriz de comandos  
B := {|| x := 10, y := 20}
```

### 6.2.3. Blocos de Código

Diferentemente de uma matriz, não se pode acessar elementos de um bloco de código, através de um índice numérico. Porém blocos de código são semelhantes a uma lista de expressões, e a uma pequena função.

Ou seja, podem ser executados. Para a execução, ou avaliação de um bloco de código, deve-se utilizar a função Eval():

```
nRes := Eval(B) ==> 20
```

Essa função recebe como parâmetro um bloco de código e avalia todas as expressões contidas neste bloco de código, retornando o resultado da última expressão avaliada.

#### Passando Parâmetros

---

Já que os blocos de código são como pequenas funções, também é possível a passagem de parâmetros para um bloco de código. Os parâmetros devem ser informados entre as barras verticais (|), separados por vírgulas, assim como em uma função.

```
B := {| N | X := 10, Y := 20 + N}
```

Porém deve-se notar que já que o bloco de código recebe um parâmetro, um valor deve ser passado quando o bloco de código for avaliado.

```
C := Eval(B, 1) ==> 21
```

#### Utilizando Blocos de Código

---

Blocos de código podem ser utilizados em diversas situações. Geralmente são utilizados para executar tarefas quando os eventos de objetos são acionados ou, para modificar o comportamento padrão de algumas funções. Como no exemplo, considere a matriz abaixo:

```
A := {"GARY HALL", "FRED SMITH", "TIM JONES"}
```

Esta matriz pode ser ordenada pelo primeiro nome, utilizando-se a chamada da função `asort(A)`, resultado na matriz com os elementos ordenados dessa forma:

```
 {"FRED SMITH", "GARY HALL", "TIM JONES"}
```

A ordem padrão para a função `asort` é ascendente. Este comportamento pode ser modificado através da informação de um bloco de código que ordena a matriz de forma descendente:

```
 B := { |X, Y| X > Y }  
 aSort(A, B)
```

O bloco de código (de acordo com a documentação da função `asort`) deve ser escrito para aceitar dois parâmetros que são os dois elementos da matriz para comparação. Note que o bloco de código não conhece que elementos estão comparando - a função `asort` seleciona os elementos (talvez utilizando o algoritmo *QuickSort*) e passa-os para o bloco de código. O bloco de código compara-os e retorna verdadeiro (.T.) se encontram na ordem correta, ou falso (.F.) se não. Se o valor de retorno for falso, a função `asort` então trocará os valores de lugar e seguirá comparando o próximo par de valores.

Então, no bloco de código anterior, a comparação  $X > Y$  é verdadeira se os elementos estão em ordem descendente, o que significa que o primeiro valor é maior que o segundo.

Para ordenar a mesma matriz pelo último nome, também em ordem descendente, é possível utilizar o seguinte bloco de código:

```
 B := { |X, Y| SUBSTR(X, At(" ",X)+1) > SUBSTR(Y, At(" ",Y)+1) }
```

Note que este bloco de código procura e compara as partes dos caracteres, imediatamente seguinte a um espaço em branco. Depois de utilizar esse bloco de código para a função `asort`, a matriz conterá:

```
 {"GARY HALL", "TIM JONES", "FRED SMITH"}
```

Finalmente, para ordenar um sub-elemento (coluna) de uma matriz por exemplo, utiliza-se o seguinte bloco de código:

```
B := { |X, Y| X[1] > Y[1] }
```

## 6.2.4. Funções para manipulação de blocos de código

A linguagem ADVPL possui diversas funções que auxiliam na manipulação de blocos de código, dentre as quais podemos citar as mais utilizadas:

- EVAL()
- DBEVAL()
- AEVAL()

### EVAL()

---

<b>Sintaxe</b>	<b>EVAL(bBloco, xParam1, xParam2, xParamZ)</b>
<b>Descrição</b>	A função EVAL() é utilizada para a avaliação direta de um bloco de código, utilizando as informações disponíveis no momento de sua execução. Esta função permite a definição e passagem de diversos parâmetros que serão considerados na interpretação do bloco de código.

### DBEVAL()

---

<b>Sintaxe</b>	<b>Array(bBloco, bFor, bWhile)</b>
<b>Descrição</b>	A função DBEval() permite que todos os registros, de uma determinada tabela, sejam analisados e para cada registro será executado o bloco de código definido.

### AEVAL()

---

<b>Sintaxe</b>	<b>AEVAL(aArray, bBloco, nInicio, nFim)</b>
<b>Descrição</b>	A função AEVAL() permite que todos os elementos de um determinada array sejam analisados e para cada elemento será executado o bloco de código definido.



**Anotações**

---

---

---

---

---

## 7. FUNÇÕES

A maior parte das rotinas escritas em programas são compostas de um conjunto de comandos. Essas rotinas se repetem ao longo de todo o desenvolvimento. Uma função nada mais é do que um conjunto de comandos que para ser utilizada, basta chamá-la pelo seu nome.

Para tornar uma função mais flexível, ao chamá-la é possível a passagem de parâmetros, que contêm os dados e as informações que definem o processamento da função.

Os parâmetros das funções descritas utilizando a linguagem ADVPL são posicionais, ou seja, na sua passagem não importa o nome da variável e sim a sua posição dentro da lista de parâmetros, que permite executar uma função escrevendo:

```
Calcula(parA, parB, parC) // Chamada da função em uma rotina
```

**E a função deve estar escrita assim:**

```
User Function Calcula(x, y, z)
```

```
... Comandos da Função
```

```
Return ...
```

Neste caso, x assume o valor de parA, y de parB e z de parC.

A função também tem a faculdade de retornar uma variável, podendo inclusive ser um Array. Para tal, encerra-se a função com:

**Return(campo)**

Assim `A := Calcula(parA,parB,parC)` atribui a A o conteúdo do retorno da função Calcula.

No ADVPL existem milhares de funções escritas pela equipe de Tecnologia Microsiga, pelos analistas de suporte e pelos próprios usuários.



**Fique  
atento**

***Existe um ditado que diz:***

***“Vale mais um programador que conhece todas as funções disponíveis em uma linguagem do que aquele que, mesmo sendo gênio, reinventa a roda a cada novo programa”.***

No TDN (TOTVS Developer Network) mais de 500 estão documentadas, e este número tende a aumentar exponencialmente com os novos processos de documentação que estão em implantação na Tecnologia e Inteligência Protheus.

O objetivo do curso é apresentar, demonstrar e fixar a utilização das principais funções, sintaxes e estruturas utilizadas em ADVPL.

No ADVPL até os programas chamados pelo menu são funções, sendo que em um repositório não pode haver funções com o mesmo nome e para permitir que os usuários e analistas possam desenvolver suas próprias funções sem que as mesmas conflitem com as já disponíveis no ambiente ERP, foi implementada pela Tecnologia Microsiga um tipo especial de função denominado “User Function”.

Nos tópicos a seguir serão detalhados os tipos de funções disponíveis na linguagem ADVPL, suas formas de utilização e respectivas diferenças.

## 7.1. Tipos e escopos de funções

Em ADVPL podem ser utilizados os seguintes tipos de funções:

- Function()**
- User Function()**
- Static Function()**
- Main Function()**

### Function()

---

Funções ADVPL convencionais, restritas ao desenvolvimento da área de Inteligência Protheus da Microsiga.

O interpretador ADVPL distingue os nomes de funções do tipo Function() com até dez caracteres. A partir do décimo carácter, apesar do compilador não indicar quaisquer tipos de erros, o interpretador ignorará os demais caracteres.

#### Exemplo:

```
// Fonte MATA100INCL.PRW
#include "protheus.ch"

Function MATA100INCL01()

ALERT("01")
Return

Function MATA100INCL02()

ALERT("02")
Return
```

Ao executar a função MATA100INCL01() será exibida a mensagem "01", mas ao executar a função MATA100INCL02() também será exibida a mensagem "01", pois o interpretador considera o nome da função como "MATA100INC".



**Fique atento**

***Funções do tipo Function() somente podem ser executadas através dos módulos do ERP.***

***Somente poderão ser compiladas funções do tipo Function() se o MP-IDE possuir uma autorização especial fornecida pela Microsiga.***

***Funções do tipo Function() são acessíveis por quaisquer outras funções em uso pela aplicação.***

## **User Function()**

---

As "User Defined Functions" ou funções definidas pelos usuários, são tipos especiais de funções implementados pelo ADVPL para garantir que desenvolvimentos específicos não realizados pela Inteligência Protheus da Microsiga sobreponham as funções padrões desenvolvidas para o ERP.

O interpretador ADVPL considera que o nome de uma User Function é composto pelo nome definido para a função, precedido dos caracteres "U\_". Desta forma a User Function XMAT100I será tratada pelo interpretador como "U\_XMAT100I".



**Fique atento**

***Como ocorre o acréscimo dos caracteres "U\_" no nome da função e o interpretador considera apenas os dez primeiros caracteres da função, para sua diferenciação é recomendado que os nomes das User Functions tenham apenas oito caracteres, evitando***

**resultados indesejados durante a execução da aplicação.**

**Funções do tipo User Function são acessíveis por quaisquer outras funções em uso pela aplicação, desde que em sua chamada sejam utilizados os caracteres "U\_", em conjunto com o nome da função.**



*Dica*

**As User Functions podem ser executadas a partir da tela inicial do client do ERP (Microsiga Protheus Remote), mas as aplicações que pretendem disponibilizar esta opção devem possuir um preparo adicional de ambiente.**

**Para maiores informações consulte no TDN o tópico sobre preparação de ambiente e a documentação sobre a função RpcSetEnv().**

## Static Function()

---

Funções ADVPL tradicionais, cuja visibilidade está restrita às funções descritas no mesmo arquivo de código fonte no qual estão definidas.

### Exemplo:

```
//Fonte FINA010.PRW

Function FINA010()

CriaSx1("FIN010")
Return

Static Function CRIASX1()
//Fonte FINA020.PRW
Function FINA020()

CriaSx1("FIN020")
Return

Static Function CRIASX1()
```

No exemplo acima, existem duas funções denominadas CRIASX1(), definidas em arquivos de código fonte distintos: FINA010.PRW e FINA020.PRW.

A função FINA010() terá visibilidade apenas da função CRIASX1(), definida no arquivo de código fonte FINA010.PRW, sendo que o mesmo ocorre com a função FINA020().

Este recurso permite isolar funções de uso exclusivo de um arquivo de código fonte, evitando a sobreposição ou a duplicação de funções na aplicação.

Neste contexto as Static Functions() são utilizadas para:

1. Padronizar o nome de uma determinada função, que possui a mesma finalidade, mas que sua implementação pode variar de acordo com a necessidade da função principal / aplicação.
2. Redefinir uma função padrão da aplicação, adequando-a às necessidades específicas de uma função principal / aplicação.
3. Proteger as funções de uso específico de um arquivo de código fonte / função principal.



Fique atento

***O Ambiente de desenvolvimento utilizado na aplicação ERP (MP-IDE) valida se existem Functions(), Main Functions() ou User Functions() com o mesmo nome, mas em arquivos de códigos fontes distintos, evitando a duplicidade ou sobreposição de funções.***

## **Main Function()**

---

Main Function() é outro tipo de função especial do ADVPL incorporado para permitir tratamentos diferenciados na aplicação ERP.

Uma Main Function() tem a característica de poder ser executada através da tela inicial de parâmetros do client do ERP (Microsig Protheus Remote), da mesma forma que uma User Function, com a diferença que as Main Functions somente podem ser desenvolvidas com o uso da autorização de compilação, tornando sua utilização restrita à Inteligência Protheus da Totvs.

Na aplicação ERP é comum o uso das Main Functions(), nas seguintes situações:

1. Definição dos módulos da aplicação ERP: Main Function Sigaadv()
2. Definição de atualizações e updates: AP710TOMP811()
3. Atualizações específicas de módulos da aplicação ERP: UpdateATF()

## 7.2. Passagem de parâmetros entre funções

Como mencionado anteriormente os parâmetros das funções descritas, utilizando a linguagem ADVPL são posicionais, ou seja, na sua passagem não importa o nome da variável e sim a sua posição dentro da lista de parâmetros.

Complementando esta definição, podem ser utilizadas duas formas distintas de passagens de parâmetros para funções descritas na linguagem ADVPL:

- Passagem de parâmetros por conteúdo.
- Passagem de parâmetros por referência.

### Passagem de parâmetros por conteúdo

A passagem de parâmetros por conteúdo é a forma convencional de definição dos parâmetros recebidos pela função chamada, na qual a função recebe os conteúdos passados pela função chamadora, na ordem com os quais são informados.

```
User Function CalcFator(nFator)

Local nCnt
Local nResultado := 0

For nCnt := nFator To 1 Step -1
nResultado *= nCnt
Next nCnt

Alert("O fatorial de " + cValToChar(nFator) + ;
      " é " + cValToChar(nResultado))

Return
```

Avaliando a função CalcFator() descrita anteriormente, podemos verificar que a mesma recebe como parâmetro para a sua execução a variável nFator.

Com base nesta função, podemos descrever duas formas de passagem de parâmetros por conteúdo:

- Passagem de conteúdos diretos.**
- Passagem de variáveis como conteúdos.**

#### **Exemplo 01 – Passagem de conteúdos diretos**

```
User Function DirFator()
```

```
Local nResultado := 0
```

```
nResultado := CalcFator(5)
```

A passagem de conteúdos diretos implica na definição explícita do valor do parâmetro, na execução da chamada da função. Nesse caso foi informado o conteúdo 5 (numérico) como conteúdo para o primeiro parâmetro da função CalcFator.

Como a linguagem ADVPL trata os parâmetros de forma posicional, o conteúdo 5 será atribuído diretamente à variável, definida como primeiro parâmetro da função chamada. No nosso caso, nFator.

Por ser uma atribuição de parâmetros por conteúdo, o interpretador da linguagem basicamente executa uma operação de atribuição normal, ou seja, nFator := 5.



Fique atento

***Duas características da linguagem ADVPL tornam necessária uma atenção especial na chamada de funções:***

***A linguagem ADVPL não é uma linguagem “tipada”, de forma que as variáveis não tem um tipo previamente definido, aceitando o conteúdo que lhes for imposto por meio de uma atribuição.***

***Os parâmetros de uma função são atribuídos de acordo com a ordem em que tais parâmetros são definidos na chamada desta ordem. Não é realizado nenhum tipo de consistência em relação aos tipos dos conteúdos, e a obrigatoriedade de parâmetros nesta ação.***



Dica

***Os parâmetros de uma função são caracterizados como variáveis de escopo LOCAL para efeito de execução.***

***Dessa forma os mesmos não devem ser definidos novamente como LOCAL, na área de definição e inicialização de variáveis, pois se isso ocorrer haverá a perda dos valores recebidos pela redefinição das variáveis na função.***

***Se for necessário garantir um conteúdo padrão para um determinado parâmetro, deve ser utilizado o identificador DEFAULT, conforme detalhamento no tópico “Tratamento de valores padrões para parâmetros de funções”.***

**Exemplo 02 – Passagem de variáveis como conteúdos**

```
User Function DirFator()

Local nResultado := 0
Local nFatorUser := 0

nFatorUser := GetFator() // Função ilustrativa na qual o usuário informa o fator a ser utilizado.

nResultado := CalcFator(nFatorUser)
```

A passagem de conteúdos como variáveis implica na utilização de variáveis de apoio para executar a chamada de uma função. Nesse caso foi informada a variável `nFatorUser`, a qual será definida pelo usuário com a utilização da função ilustrativa `GetFator()`. O uso de variáveis de apoio flexibiliza a chamada de outras funções, pois elas serão parametrizadas de acordo com as necessidades daquele processamento específico, no qual se encontra a função chamadora.

Como a linguagem ADVPL trata os parâmetros de forma posicional, o conteúdo da variável `nFatorUser` será atribuído diretamente à variável definida como primeiro parâmetro da função chamada, no caso `nFator`.

Por ser uma atribuição de parâmetros por conteúdo, o interpretador da linguagem basicamente executa uma operação de atribuição normal, ou seja, `nFator := nFatorUser`.



Dica

***A passagem de parâmetros não necessita que as variáveis informadas na função chamadora tenham os mesmos nomes das variáveis utilizadas na definição de parâmetros da função chamada.***

***Dessa forma podemos ter:***

***User Function DirFator()***

***Local nFatorUser := GetFator()***

***nResultado := CalcFator(nFatorUser)***

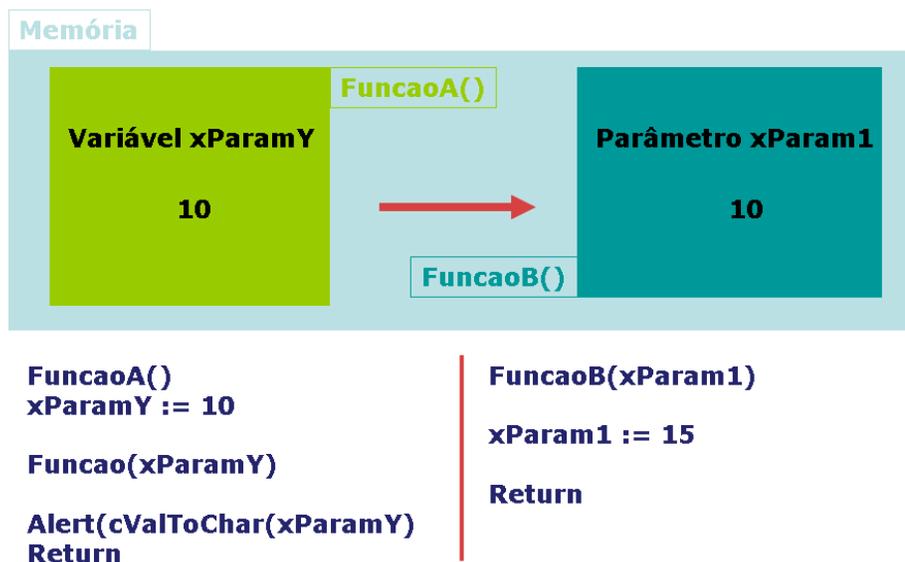
***Function CalcFator(nFator)***

***As variáveis nFatorUser e nFator podem ter nomes diferentes pois o interpretador fará a atribuição de conteúdo com base na ordem dos parâmetros e não pelo nome das variáveis.***

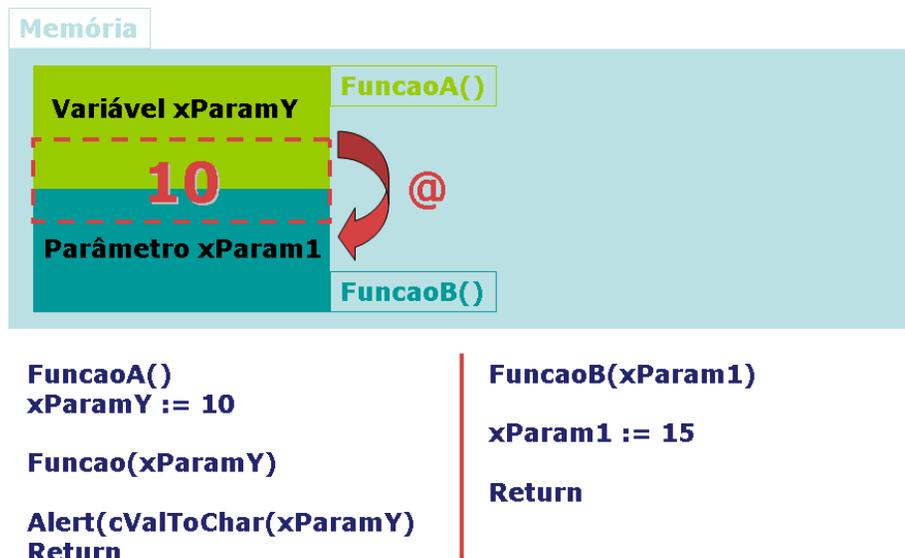
## Passagem de parâmetros por referência

A passagem de parâmetros por referência é uma técnica muito comum nas linguagens de programação, pois permite que variáveis de escopo LOCAL tenham o seu conteúdo manipulado por funções específicas, mantendo o controle dessas variáveis restritas à função que as definiu e às funções desejadas pela aplicação.

A passagem de parâmetros por referência utiliza o conceito de que uma variável é uma área de memória e, portanto, passar um parâmetro por referência nada mais é do que, ao invés de passar o conteúdo para a função chamada, passar qual a área de memória utilizada pela variável passada.



### Passagem de parâmetros tradicional – Duas variáveis x Duas áreas de memória



**Passagem de parâmetros por referência – Duas variáveis x uma única área de memória**



**Anotações**

---

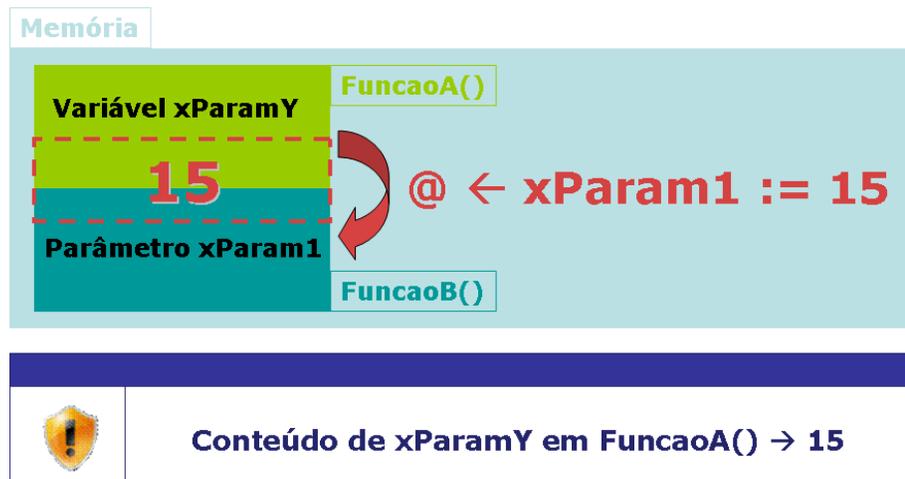
---

---

---

---

Dessa forma, a função chamada tem acesso não apenas ao conteúdo, mas à variável em si, pois a área de memória é a variável e qualquer alteração será visível à função chamadora quando tiver o retorno desta função.



### Tratamento de conteúdos padrões para os parâmetros de funções

O tratamento de conteúdos padrão para parâmetros de funções é muito utilizado nas funções padrões da aplicação ERP, de forma a garantir a correta execução dessas funções por qualquer função chamadora, evitando situações de ocorrências de erros pela falta da definição de parâmetros necessários à correta utilização da função.



Dica

***A linguagem ADVPL não obriga a passagem de todos os parâmetros descritos na definição da função, sendo que os parâmetros não informados serão considerados com conteúdo nulo.***

Dessa forma, o uso do identificador DEFAULT permite ao desenvolvedor a garantia de que, na utilização da função, determinados parâmetros terão o valor com um tipo adequado à função.

#### Exemplo:

```
User Function CalcFator(nFator)
```

```
Local nCnt
```

```
Local nResultado := 0
```

```
Default nFator := 1
```

```
For nCnt := nFator To 1 Step -1
```

```
    nResultado *= nCnt
```

```
Next nCnt
```

```
Return nResultado
```

No exemplo descrito, se o parâmetro nFator não for informado na função chamadora, o mesmo terá seu conteúdo definido como 1.

Se esse tratamento não fosse realizado e com isso o parâmetro nFator não fosse informado, ocorreria o seguinte evento de erro:

**Exemplo:**

```
User Function CalcFator(nFator)
```

```
Local nCnt
```

```
Local nResultado := 0
```

```
For nCnt := nFator To 1 Step -1 // nFator está como Nulo, portando nCnt é nulo
```

```
    nResultado *= nCnt
```

```
Next nCnt // Ao efetuar o Next, o interpretador realiza a ação nCnt += 1.
```

```
Return nResultado
```

Como o interpretador realizará a ação  $nCnt += 1$ , e o conteúdo da variável  $nCnt$  é nulo, ocorrerá o erro de "type mismatch on +=, expected N  $\rightarrow$  U", pois os tipos das variáveis envolvidos na operação são diferentes:  $nCnt \rightarrow$  nulo (U) e  $1 \rightarrow$  numérico (N).



Dica

***Se o parâmetro que possui a opção DEFAULT, descrita no fonte, for informado, a linha de DEFAULT não será executada, mantendo-se dessa forma o conteúdo passado pela função chamadora.***



**Anotações**

---

---

---

---

---

## 8. DIRETIVAS DE COMPILAÇÃO

O compilador ADVPL possui uma funcionalidade denominada pré-processador. O pré-processador é um programa que examina o programa fonte escrito em ADVPL e executa certas modificações nele, baseadas nas Diretivas de Compilação.

As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, o qual é executado pelo compilador antes da execução do processo de compilação propriamente dito. Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado de acordo com as diretivas de compilação. Estas são iniciadas pelo caractere "#".

As diretivas podem ser colocadas em qualquer parte do programa, sendo que as implementadas pela linguagem ADVPL são:

- #INCLUDE**
- #DEFINE**
- #IFDEF**
- #IFNDEF**
- #ELSE**
- #ENDIF**
- #COMMAND**



Lem  
bre-  
se

*As diretivas de compilação também são conhecidas como UDC – User Defined Commands.*

### Diretiva: **#INCLUDE**

---

A diretiva **#INCLUDE** indica em que arquivo de extensão ".CH" (padrão ADVPL) estão os UDCs a serem utilizados pelo pré-processador.

A aplicação ERP possui diversos includes que devem ser utilizados segundo a aplicação que será desenvolvida. Dessa forma, será permitida a utilização de recursos adicionais definidos para a linguagem, implementados pela área de Tecnologia da Totvs.

Os includes mais utilizados nas aplicações ADVPL, desenvolvidas para o ERP são:

- PROTHEUS.CH:** Diretivas de compilação como padrões para a linguagem. Contém a especificação da maioria das sintaxes utilizadas nos fontes, inclusive permitindo a compatibilidade da sintaxe tradicional do Clipper para os novos recursos implementados no ADVPL.

O include PROTHEUS.CH ainda contém a referência a outros includes utilizados pela linguagem ADVPL que complementam essa funcionalidade de compatibilidade com a sintaxe Clipper, tais como:

- **DIALOG.CH**
- **FONT.CH**
- **INI.CH**
- **PTMENU.CH**
- **PRINT.CH**



Fiqu  
atento

***A utilização do include "protheus.ch", nos fontes desenvolvidos para a aplicação ERP Protheus, é obrigatória e necessária ao correto funcionamento das aplicações***

**AP5MAIL.CH:** Permite a utilização da sintaxe tradicional na definição das seguintes funções de envio e recebimento de e-mail:

- CONNECT SMTP SERVER
- CONNECT POP SERVER
- DISCONNECT SMTP SERVER
- DISCONNECT POP SERVER
- POP MESSAGE COUNT
- SEND MAIL FROM
- GET MAIL ERROR
- RECEIVE MAIL MESSAGE

**TOPCONN.CH:** Permite a utilização da sintaxe tradicional na definição das seguintes funções de integração com a ferramenta TOPCONNECT (MP10 – DbAccess):

- TCQUERY

**TBICONN.CH:** Permite a utilização da sintaxe tradicional na definição de conexões, com a aplicação Server do ambiente ERP, através das seguintes sintaxes:

- CREATE RPCCONN
- CLOSE RPCCONN
- PREPARE ENVIRONMENT
- RESET ENVIRONMENT
- OPEN REMOTE TRANSACTION
- CLOSE REMOTE TRANSACTION
- CALLPROC IN
- OPEN REMOTE TABLES

**XMLXFUN.CH:** Permite a utilização da sintaxe tradicional, na manipulação de arquivos e strings no padrão XML, através das seguintes sintaxes:

- CREATE XMLSTRING
- CREATE XMLFILE
- SAVE XMLSTRING
- SAVE XMLFILE
- ADDITEM TAG
- ADDNODE NODE
- DELETENODE



*Dica*

*Os recursos de tratamentos de e-mails, integração com a ferramenta TOPCONNECT (DbAcess), preparação de ambientes e manipulação de arquivos e strings do padrão XML, serão abordados no curso de ADVPL Avançado.*



Dica

**O diretório de includes deve ser especificado no Ambiente de desenvolvimento do ERP Protheus (MP-IDE), para cada configuração de compilação disponível.**

**Se o diretório de includes não estiver informado, ou esteja informado incorretamente, será exibida uma mensagem de erro informando:**

*"Não foi possível criar o arquivo <camino\nome> .ERX"*



Dica

**As funções desenvolvidas para a aplicação ERP costumam utilizar includes, para definir o conteúdo de strings e as variáveis diversas, utilizadas pela aplicação em diferentes idiomas. Dessa forma é normal verificar que um fonte possui um arquivo ".CH" com o mesmo nome, o que caracteriza esse tipo de include.**

## **Diretiva: #DEFINE**

---

A diretiva #DEFINE permite que o desenvolvedor crie novos termos para serem utilizados no código fonte. Este termo tem o efeito de uma variável de escopo PUBLIC, mas que afeta somente o fonte na qual o #DEFINE está definido com a característica de não permitir a alteração de seu conteúdo.

Dessa forma, um termo definido através da diretiva #DEFINE pode ser considerado como uma constante.



Dica

**Os arquivos de include definidos para os fontes da aplicação ERP contêm diretivas #DEFINE para as strings de textos de mensagens exibidas para os usuários nos três idiomas com os quais a aplicação é distribuída:**

**Português, Inglês e Espanhol.**

***Por essa razão a aplicação ERP possui três repositórios distintos para cada uma das bases de dados homologadas pela Microsiga, pois cada compilação utiliza uma diretiva referente ao seu idioma.***

## **Diretivas: #IFDEF, #IFDEF, #ELSE e #ENDIF**

---

As diretivas #IFDEF, #IFDEF, #ELSE e #ENDIF permitem ao desenvolvedor criar fontes flexíveis e sensíveis a determinadas configurações da aplicação ERP.

Com essas diretivas, podem ser verificados parâmetros do Sistema, tais como o idioma com o qual está parametrizado e a base de dados utilizada para armazenar e gerenciar as informações do ERP. Assim, ao invés de escrever dois, ou mais códigos fontes que realizam a mesma função, mas utilizando recursos distintos para cada base de dados, ou exibindo mensagem para cada um dos idiomas tratados pela aplicação. O desenvolvedor pode preparar seu código fonte para ser avaliado pelo pré-processador, que irá gerar um código compilado de acordo com a análise dos parâmetros de ambiente.

Essas diretivas de compilação estão normalmente associadas às seguintes verificações de ambiente:

- Idioma: verifica as variáveis SPANISH e ENGLISH, disponibilizadas pela aplicação. O idioma português é determinado pela exceção:

```
#IFDEF SPANISH
    #DEFINE STR0001 "Hola !!!"
#ELSE

    #IFDEF ENGLISH
        #DEFINE STR0001 "Hello !!!"
    #ELSE
        #DEFINE STR0001 "Olá !!!"
    #ENDIF
#ENDIF
```



Fique atento

**Apesar da estrutura semelhante ao IF-ELSE-ELSEIF-ENDIF, não existe a diretiva de compilação #ELSEIF, o que torna necessário o uso de diversos #IFDEFs para a montagem de uma estrutura que seria facilmente solucionada com IF-ELSE-ELSEIF-ENDIF.**



Dica

**A aplicação ERP disponibiliza para a variável de escopo PUBLIC - `_LANGUAGE`, que contém uma string que identifica o idioma em uso pelo Sistema, cujos conteúdos possíveis são:**

**`"PORTUGUESE"`**

**`"SPANISH"`**

**`"ENGLISH"`**

- Banco de Dados: Verifica as variáveis AXS e TOP para determinar se o banco de dados em uso pela aplicação está no formato ISAM (DBF, ADS, CTREE, etc.) ou, se está utilizando a ferramenta TOPCONNECT (DbAccess).

```
#IFDEF TOP
```

```
    cQuery := "SELECT * FROM "+RETSQLNAME("SA1")
```

```
    dbUseArea(.T., "TOPCONN", TcGenQry(,cQuery), "SA1QRY",.T.,.T.)
```

```
#ELSE
```

```
    DbSelectArea("SA1")
```

```
#ENDIF
```



Fique atento

**Os bancos de dados padrão AS400 não permitem a execução de queries no formato SQLANSI, através da ferramenta TOPCONNECT (DbAccess).**

**Dessa forma é necessário realizar uma verificação adicional ao #IFDEF TOP antes de executar uma query, que no caso é realizada através do uso da função TcSrvType(), a qual retorna a string "AS/400", quando este for o banco em uso.**

**Para esses bancos deve ser utilizada a sintaxe ADVPL tradicional.**

## Diretiva: #COMMAND

A diretiva #COMMAND é utilizada principalmente nos includes da linguagem ADVPL para efetuar a tradução de comandos em sintaxe CLIPPER, para as funções implementadas pela Tecnologia Microsiga.

Essa diretiva permite que o desenvolvedor defina para o compilador como uma expressão deve ser interpretada.

### Trecho do arquivo PROTHEUS.CH

```
#xcommand @ <nRow>, <nCol> SAY [ <oSay> <label: PROMPT,VAR > ] <cText> ;  
  [ PICTURE <cPict> ] ; [ <dlg: OF,WINDOW,DIALOG > <oWnd> ] ;  
  [ FONT <oFont> ] ; [ <ICenter: CENTERED, CENTER > ] ;  
  [ <IRight: RIGHT > ] ; [ <IBorder: BORDER > ] ;  
  [ <IPixel: PIXEL, PIXELS > ] ; [ <color: COLOR,COLORS > <nClrText> [,<nClrBack> ] ] ;  
  [ SIZE <nWidth>, <nHeight> ] ; [ <design: DESIGN > ] ;  
  [ <update: UPDATE > ] ; [ <IShaded: SHADED, SHADOW > ] ;  
  [ <IBox: BOX > ] ; [ <IRaised: RAISED > ] ;  
=> ;
```

```
[ <oSay> := ] TSay():New( <nRow>, <nCol>, <{cText}>;  
[<oWnd>], [<cPict>], <oFont>, <.lCenter.>, <.lRight.>, <.lBorder.>;  
<.lPixel.>, <nClrText>, <nClrBack>, <nWidth>, <nHeight>;  
<.design.>, <.update.>, <.lShaded.>, <.lBox.>, <.lRaised.> )
```

Através da diretiva #COMMAND, o desenvolvedor determinou as regras para que a sintaxe tradicional da linguagem CLIPPER, para o comando SAY, fosse convertida na especificação de um objeto TSAY() do ADVPL.



### Exercícios

#### Exercício 08

Vamos Desenvolver um programa que permita ao usuário pesquisar um cliente, informando seu CNPJ e se o mesmo existir na base, exibir suas principais informações.



### Exercícios

#### Exercício 09

Utilizando a interface visual desenvolvida para o exercício anterior, desenvolver a função genérica `GetTexto()`, para ser utilizada nas aplicações do Jogo da Velha e Jogo da Forca.



### Exercícios

#### Exercício 10

Utilizando a interface visual desenvolvida para o exercício anterior, desenvolver a função genérica `GetTexto()`, para ser utilizada nas aplicações do Jogo da Velha e Jogo da Forca.



### Exercícios

#### Exercício 11

Utilizando a função `AVISO()`, desenvolver um programa que permita ao usuário selecionar a opção de busca de CNPJ por cliente ou fornecedor, e se encontrar, exiba os seus dados principais.



## Exercícios

### Exercício 12

Desenvolver uma rotina que capture vários CNPJs de clientes informados pelo usuário, e verifique para cada um deles se o mesmo existe ou não na base de dados. Ao final informar quais CNPJs foram informados, e de acordo com a seleção do usuário, exibir os dados principais de um destes clientes.



### Exercícios

#### Exercício 13

Utilizando a função FORMBATCH(), desenvolver uma rotina que verifique se para cada item de uma nota fiscal de entrada existe o respectivo cabeçalho, e se for encontrado algum item inconsistente, comunique a ocorrência ao usuário que está realizando o processamento.



### Exercícios

#### Exercício 14

Vamos Desenvolver uma rotina que, através do uso de um bloco de código, converta a estrutura da tabela SA1, obtida com a função DBSTRUCT(), em uma string denominada cCampo.

## **MÓDULO 03: DESENVOLVENDO PEQUENAS CUSTOMIZAÇÕES**

### **9. ADVPL E O ERP MICROSIGA PROTHEUS**

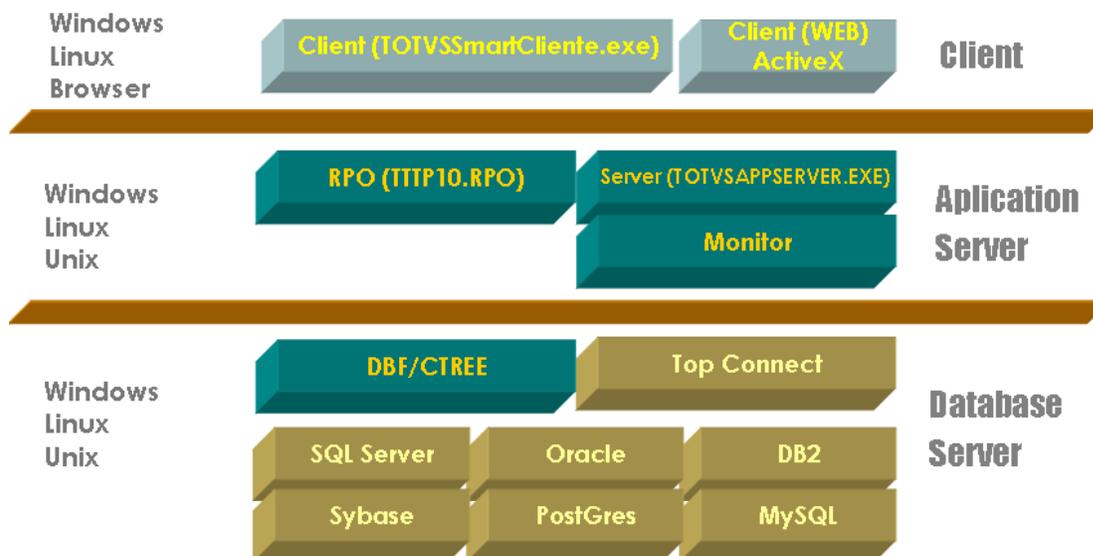
O ADVPL (Advanced Protheus Language) é uma linguagem de programação desenvolvida pela Totvs e que contém todas as instruções e funções necessárias ao desenvolvimento de um sistema, independentemente de sua complexidade.

O Sistema Protheus, por outro lado, é uma plataforma tecnológica que engloba um Servidor de Aplicação, um Dicionário de Dados e as Interfaces para a conexão com o usuário. É o Protheus que executa o código ADVPL, assim como o devido acesso à base de dados.

O Protheus é composto pelo ERP (que engloba, além das funcionalidades descritas nos capítulos anteriores, mais de trinta verticais aplicadas às áreas específicas de negócios) e pelo Configurador (programa que permite customizar o Sistema às necessidades do usuário de forma fácil).

## 9.1. O Ambiente Protheus

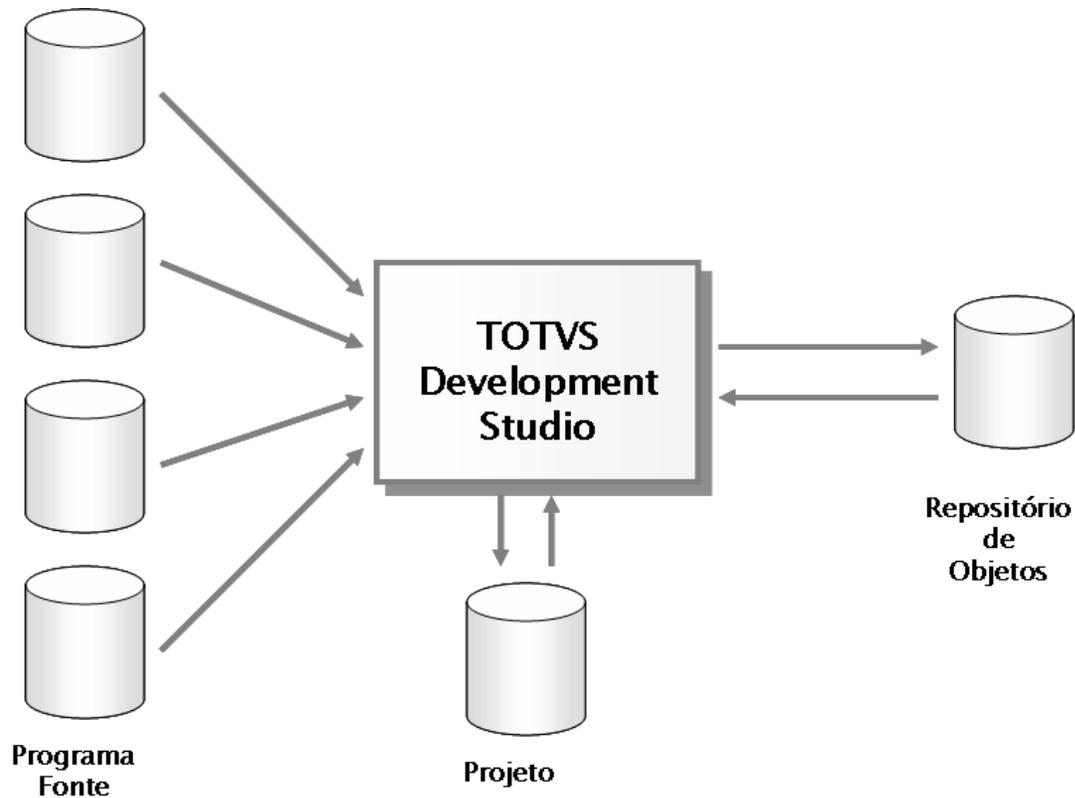
O Sistema Protheus é constituído de um conjunto de Softwares que compõem as camadas de funcionalidades básicas aos serviços de aplicação, interface, banco de dados e repositório, conforme o diagrama da figura abaixo:



**Figura: Camadas básicas do Ambiente Protheus**

Para executar um programa desenvolvido em ADVPL, é preciso antes de mais nada escrevê-lo e compilá-lo. Este procedimento é feito através da ferramenta TOTVS DevStudio do Protheus (Totvs Development Studio).

O objetivo do TOTVS DevStudio é facilitar a tarefa de escrever programas: através de cores, indica se a palavra escrita é uma instrução, uma variável, ou um comentário; organiza a biblioteca de programas em projetos e administra o repositório de objetos; aponta erros de sintaxe; permite o debug (execução passo a passo do programa, verificando o conteúdo das variáveis) e fornece assistentes (modelos) de programas.



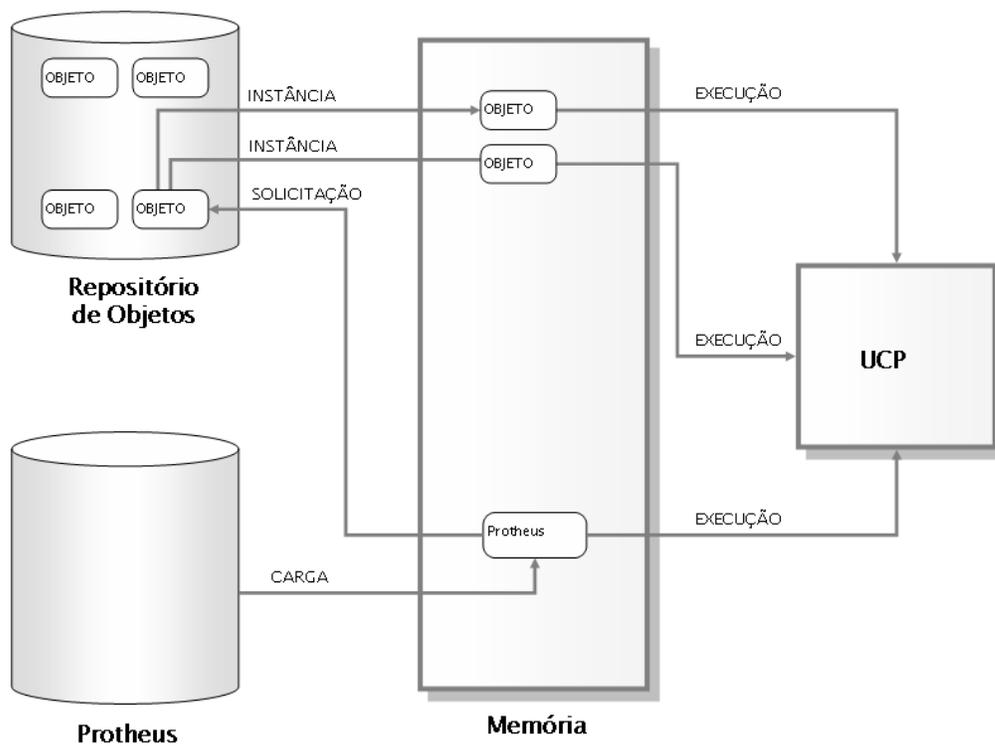
**Figura: Manutenção no repositório de objetos**

Após compilar o programa, o resultado é um objeto que é carregado na memória ficando disponível para sua execução através da aplicação PROTHEUS.

O objeto não é um executável, ou seja, não está convertido para a linguagem nativa do equipamento. Quem faz esse trabalho é o Protheus Server em tempo de execução. Por isso, o Protheus Server está sempre presente na memória em tempo de execução, permitindo:

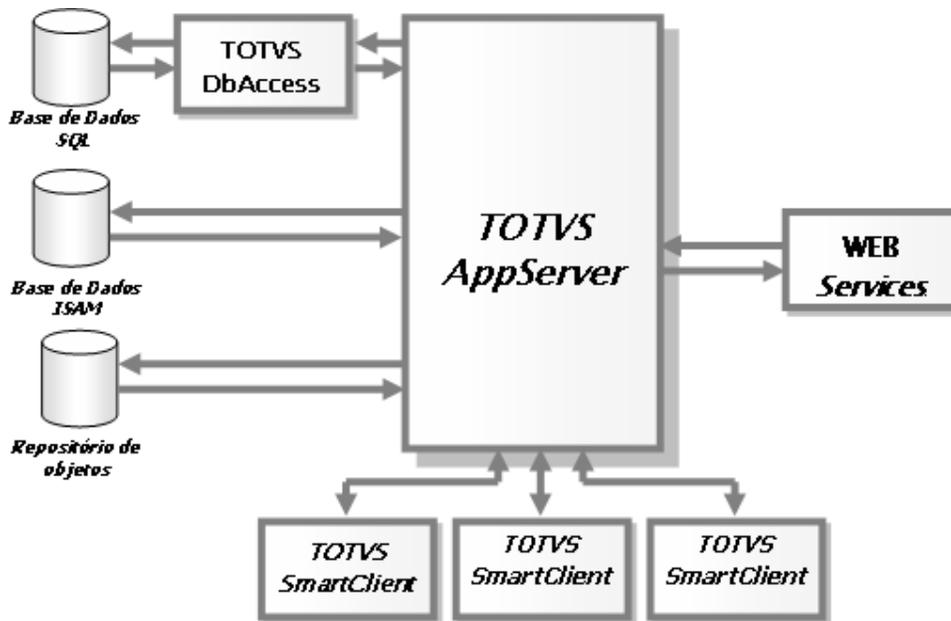
- Proteger o programa fonte, evitando que seja alterado indevidamente, pois somente os objetos são distribuídos com uma execução mais rápida em função da compilação no DEV-Studio;
- Flexibilização à plataforma de trabalho. Assim, um mesmo programa pode rodar em Ambientes Windows, Linux,

- ☑ ou mesmo em um Hand Held, ficando a tarefa de adequação para o Servidor Protheus;
- ☑ Que o Sistema cresça de forma ilimitada, pois os objetos ficam fora do executável;
- ☑ O uso de macro substituições, ou seja, o uso de rotinas exteriores ao Sistema, armazenadas em arquivos e que podem facilmente ser alteradas pelo usuário, pois o Server também interpreta o código fonte em tempo de execução.



**Figura: Diagrama esquemático de objetos Protheus**

O Repositório de Objetos é a biblioteca de objetos de todo o Ambiente Protheus, incluindo tanto os objetos implementados para as funcionalidades básicas do ERP como aqueles gerados pelos usuários. A figura abaixo demonstra a estrutura e a interconexão entre as várias camadas.



**Figura: Estrutura de interconexão do Protheus**

Ela demonstra também que os dados a serem processados podem estar armazenados em bases ISAM ou em Bancos de Dados padrão SQL. No primeiro caso, o server comunica-se diretamente com os dados. Em Bancos SQL é a interface TOPCONNECT / DBACCESS que converte os comandos de entrada e saída, adequando-os ao SQL utilizado (SQL Server Microsoft, Oracle, DB2, etc.).

Uma vez terminado o processamento do objeto chamado, ele é descartado da memória, ou seja, o Protheus é um Sistema que pode crescer de forma ilimitada pois os objetos, armazenados em um repositório, praticamente não ocupam espaço no HD (Hard Disk).

O Protheus é uma plataforma multicamada. Entre as diversas camadas, temos a interface de apresentação ao usuário (Remote), o tratamento dado para as regras de negócio implementadas (Server), o acesso aos objetos do repositório (Server), o acesso aos dados disponíveis no Banco de Dados (Server ou TOPCONNECT / DBACCESS ) e ao gerenciamento de serviços WEB (Server). Nesse processo, o Protheus possui, basicamente, quatro aplicativos utilizados com diferentes finalidades:

- Protheus Server / TOTVS AppServer:** Responsável pela comunicação entre o cliente, o banco de dados e o RPO. O nome do executável depende da versão do Sistema (TOTVSAPPSERVER.EXE) sendo que as plataformas ISAM suportadas pelo Protheus Server são DBF e CTREE.

- Protheus Remote / TOTVS SmartClient:** Instalado no Server ou na estação. O nome também depende da versão do Sistema (TOTVSSMARTCLIENT.EXE).
  
- TopConnect / DbAccess:** Responsável pela conversão dos comandos de banco de dados, adequando-os ao SQL utilizado.
  
- Protheus Monitor / TOTVS Monitor:** Programa de análise que verifica quem está usando o Sistema e possibilita o envio de mensagens, ou mesmo derrubar conexões (TOTVSMONITOR.EXE).

Alguns nomes referem-se a um conjunto de programas para facilitar a sua identificação:

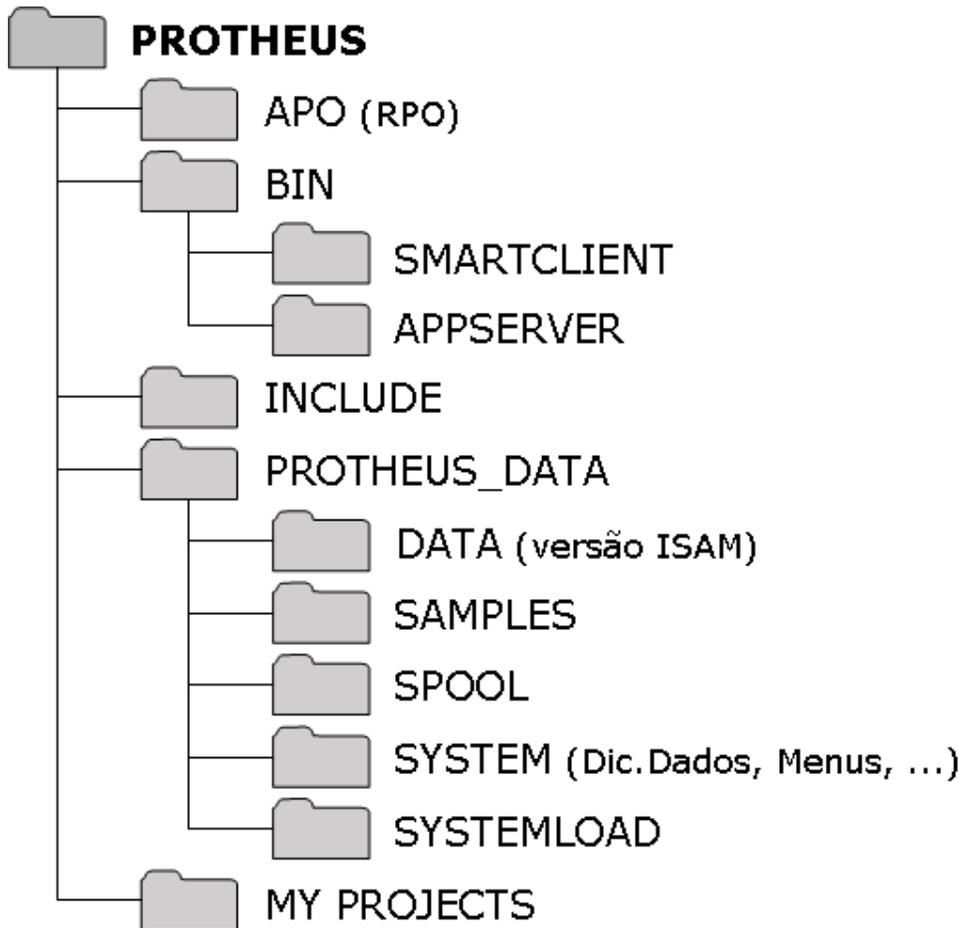
- RPO:** É o arquivo binário do APO (Advanced Protheus Objects), ou seja, os objetos.
  
- Build:** Executáveis, DLLs e o RPO completo.
  
- Patch:** Atualizações pontuais do RPO, aplicadas por meio do IDE.
  
- Update:** Pacote de atualização para o repositório (RPO), liberado periodicamente contendo todas as adequações e melhorias disponibilizadas para o Sistema em um determinado período, sendo não cumulativo, aplicadas por meio do DEV-Studio.

A interface de apresentação é realizada pelo SmartClient que processa a parte da estação, basicamente, tela e teclado. Pode estar gravado no Server e ser carregado via rede para a memória da estação. Ou, de preferência, deve ficar armazenado no HD da estação. Pode também ser carregado pelo Internet Explorer, rodando dentro do próprio browser com o SmartClient ActiveX e permitindo o acesso ao Protheus Server pela Internet, com as mesmas funcionalidades do SmartClient, sendo que o browser precisa suportar o uso da tecnologia ActiveX.

Caso exista algum Firewall ou Proxy entre o WEB Server e o Browser que vai acessar o SmartClient ActiveX, eles deverão ser configurados para permitir o seu download.

## 9.2. Organização e configuração inicial do ambiente Protheus

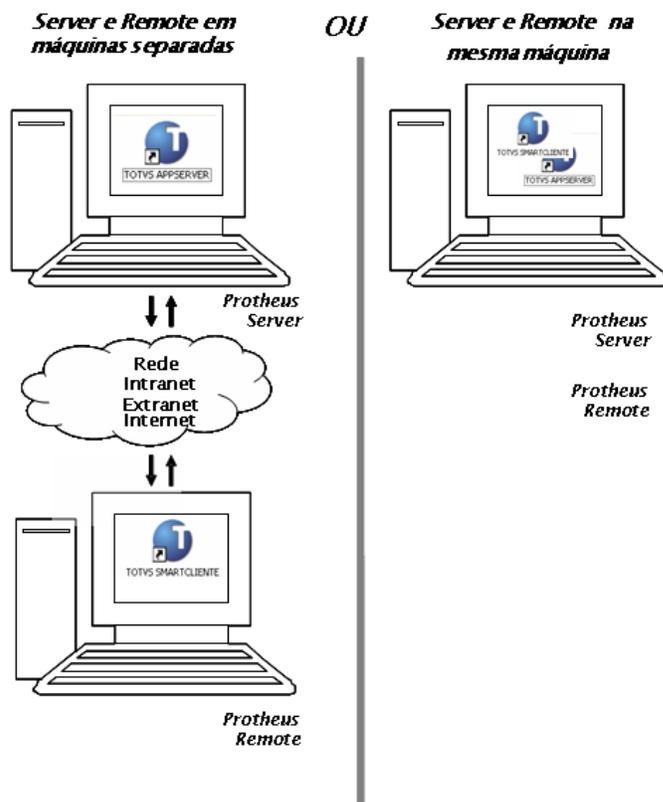
O Protheus ocupa uma pasta que tem a seguinte estrutura:



### **Figura: Estrutura básica das pastas do Protheus**

- 📁 **APO:** Contém o arquivo RPO, repositório de objetos do Protheus.
- 📁 **SMARTCLIENT:** Reúne um conjunto de arquivos executáveis, dll's e arquivos de configuração do Sistema, para possibilitar o acesso ao servidor.
- 📁 **APPSERVER:** Reúne um conjunto de executáveis, dll's e arquivos de configuração do Sistema que compõem o servidor.
- 📁 **INCLUDE:** Contém as bibliotecas necessárias para a compilação de programas Protheus.
- 📁 **DATA:** Contém a base de dados no caso de versão ISAM.
- 📁 **SAMPLES:** Oferece um conjunto de programas exemplo e arquivos ADVPL padrões da Microsiga.
- 📁 **SPOOL:** Nesta pasta são gravados os relatórios gerados em disco pelo Sistema Protheus.
- 📁 **SYSTEM:** Contém os arquivos de menus, os arquivos de configurações e os arquivos de customizações (SXs) do sistema Protheus.
- 📁 **SYSTEMLOAD:** Contém o dicionário de dados em formato TXT. É neste arquivo que estão todos os padrões e formatos para a geração dos arquivos de configurações e de customizações (SXs), conforme a localização de país definida pelo usuário, na entrada do Sistema.
- 📁 **MY PROJECTS:** Sugere-se a criação desta pasta para armazenar projetos e fontes das customizações realizadas pelo usuário.
- 📁 **UPDATES:** Sugere-se esta pasta para o armazenamento das atualizações a serem aplicadas no Sistema Protheus.

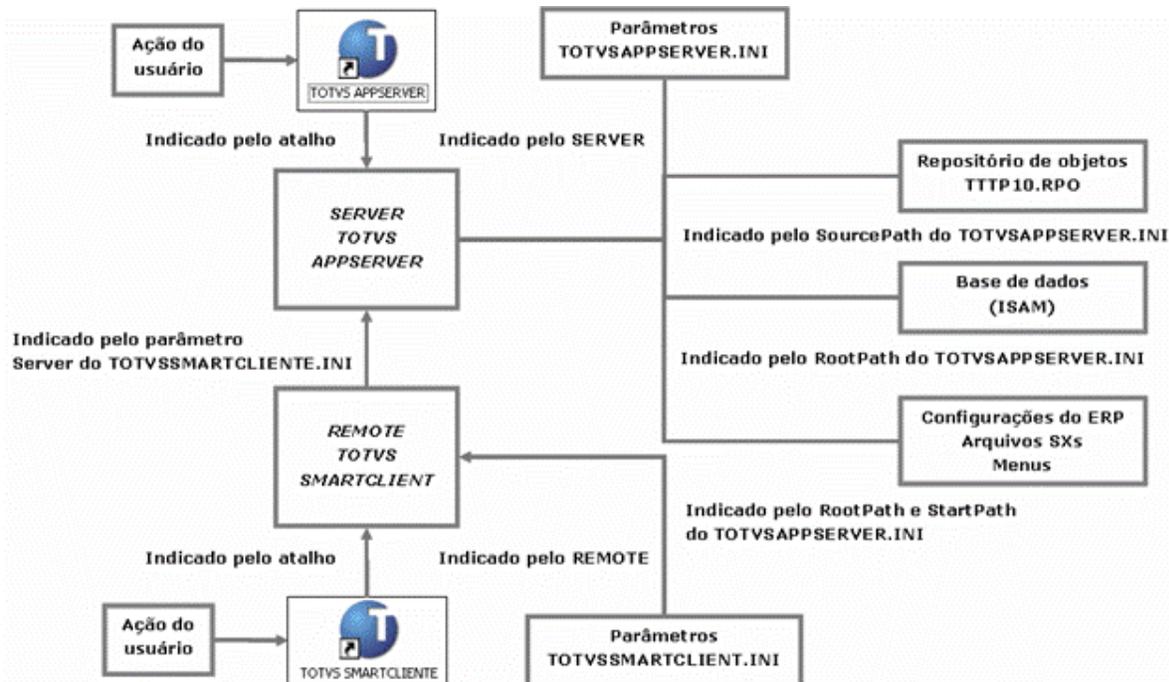
Apesar da estrutura ilustrada anteriormente, indicar que as pastas estão subordinadas à pasta PROTHEUS, é possível que algumas delas possam estar em máquinas diferentes, ou até mesmo em Ambientes computacionais diferentes.



**Figura: Formas de instalação e uso do Protheus**

Para isso, é necessário configurar, ou seja, informar ao Protheus onde está cada uma delas. Esse tipo de informação consta nos arquivos de parâmetros de configuração do Sistema (TOTVSAPPSERVER.INI e TOTVSSMARTCLIENT.INI) existentes nas respectivas pastas APPSERVER e SMARTCLIENT.

Os parâmetros do TOTVSAPPSERVER.INI são lidos pelo programa TOTVSAPPSERVER.EXE, logo no início de sua execução. O mesmo procedimento ocorre em relação aos parâmetros do TOTVSSMARTCLIENT.INI pelo programa TOTVSSMARTCLIENT.EXE. A execução desses dois programas é feita por meio de ação do usuário, facilitada pelos atalhos TOTVS APPSERVER e TOTVS SMARTCLIENT.



**Figura: Links dos parâmetros de configuração**

Para que o TOTVS AppServer e o TOTVS SmartClient sejam executados, os arquivos TOTVSAPPSERVER.INI e TOTVSSMARTCLIENTE.INI devem estar disponíveis nas respectivas pastas APPSERVER e SMARTCLIENT, pois são eles que indicam o endereço das demais pastas, conforme a ilustração da figura anterior.

O detalhe de preenchimento das propriedades dos respectivos atalhos TOTVS AppServer e o TOTVS SmartClient é demonstrado a seguir. No atalho do TOTV SAppServer, é necessário que seja informado o parâmetro "-debug" ou "-console".

## Propriedades dos atalhos

---



Destino: **c:\protheus\bin\appserver\totvsap  
- console**

Iniciar em: **c:\protheus\bin\appserver**

**-Console ou -Debug**

Executado como uma JanelaConsole, as informações recebidas das conexões com o TOTVS Application Server conectados são exibidas diretamente na tela do console do TOTVS Application Server, bem como informações de Não Conformidades.

**-Install**

Se o TOTVS Application Server, não for instalado como um Serviço do NT, durante a Instalação, isto pode ser realizado, executando-o com a opção de Linha de Comando.

**-Remove**

Para removê-lo da Lista de Serviços do NT, é possível executá-lo com a opção de Linha de Comando.



Destino: **c:\protheus\bin\smartclient\totvs  
-M**

Iniciar em: **c:\protheus\bin\smartclient**

**-Q (Quiet)**

Indica que o TOTVS Smart Client não deverá mostrar o Splash (Imagem de Apresentação) e a tela de identificação de Parâmetros Iniciais, necessita ser acompanhada da (Cláusula -P).

**-P (Main Program)**

Identifica o Programa (APO) Inicial.

**-E (Environment)**

Nome da Seção de Environment, no (Ini do Server) que será utilizada, para definições gerais.

**-C (Connection)**

Nome da Seção de Conexão que será utilizada para a conexão ao TOTVS Application Server.

**-M (AllowMultiSession)**

Permite as múltiplas instâncias (Cópias) do TOTVS Smart Client, na mesma máquina, o que por Default não é permitido.

Os parâmetros que configuram o local do RPO, o Banco de Dados (ISAM ou SQL), os arquivos de menus, configurações e customizações do sistema no arquivo INI são:

- SourcePath:** Indica o local de origem dos objetos. É o endereço do Repositório de Objetos (Exemplo: SourcePath=C:\PROTHEUS\APO).
  
- RootPath:** Aponta para a pasta raiz (inicial), a partir da qual serão localizados os dados (no caso de ISAM), bem como o próprio Dicionário de Dados (Exemplo: RootPath=C:\PROTHEUS\PROTHEUS\_DATA).
  
- StartPath:** Indica qual é a pasta dentro da pasta raiz (informada no parâmetro RootPath) que contém os arquivos de menus, os arquivos de configurações e os arquivos de customizações (SXs) do Sistema Protheus (Exemplo: StartPath=\SYSTEM\).

Não há necessidade de que os parâmetros estejam em ordem nos arquivos de configuração (.ini). Além dos parâmetros já detalhados, existem outros que podem indicar a versão do Sistema, o tipo de banco de dados, a linguagem do país em que está sendo utilizado e as máscaras de edição e formatação.



[ENVIRONMENT]

**SOURCEPATH=C:\PROTHEUS\APO**

**ROOTPATH=C:\MP811\PROTHEUS\_DATA**

**STARTPATH=\PROTHEUS\**

**RPODB=TOP**

**RPOLANGUAGE=PORTUGUESE**

**RPOVERSION=101**

**LOCALFILES=ADS**

**TRACE=0**

**LOCALDBEXTENSION=.DBF**

**PICTFORMAT=DEFAULT**

**DATEFORMAT=DEFAULT**

[DRIVERS]

**ACTIVE=TCP**

[TCP]

**TYPE=TCPIP**

**PORT=1234**

**Figura: Exemplo de um ambiente em um arquivo de parâmetros**

No exemplo da figura anterior, o rótulo [environment] descreve um conjunto de parâmetros que serão inicializados no Sistema. Os rótulos [Drivers] e [TCP] identificam a comunicação que pode ser estabelecida entre o Protheus Server e o Protheus Remote. Outros ambientes podem ser configurados no mesmo arquivo (TOTVSAPPSERVER.INI).

Já o arquivo de parâmetros do Protheus Remote (TOTVSSMARTCLIENT.INI) contém apenas as

configurações locais, basicamente as informações necessárias para a inicialização e a comunicação com o Protheus Server, conforme o exemplo da figura a seguir.



[CONFIG]

**LASTMAINPROG=SIGA  
CFG**

**LANGUAGE=1**

[DRIVERS]

**ACTIVE=TCP**

[TCP]

**SERVER=172.16.72.41**

**PORT=1234**

**Figura: Exemplo de um arquivo de configuração do remote**

- Active:** Indica qual é a forma de comunicação.
  
- Port:** Indica o número da porta a ser utilizada para a comunicação entre o Protheus Server e o Protheus Remote. É necessário que a porta utilizada na comunicação seja a mesma em ambos (no TOTVSAPPSEVER.INI e no TOTVSMARTCLIENT.INI). Vale ressaltar que a porta 80 é reservada para a Internet e pode causar conflitos caso seja utilizada na comunicação do Protheus.
  
- Server:** Aponta para o endereço do servidor que pode ser a própria máquina (localhost) ou o nome da máquina (Server= Servidor\_01) ou mesmo um endereço IP (exemplo Server=172.16.72.41).

**Exemplo:**

O parâmetro Server=172.16.72.41 no arquivo TOTVSMARTCLIENT.INI indica ao Protheus Remote o endereço da máquina na qual está funcionando o Protheus Server.

## 9.3. O Configurador do Protheus

### 9.3.1. Funcionalidades Abordadas

O objetivo deste tópico não é abranger toda a estrutura e recursos do módulo Configurador da aplicação ERP, mas permitir a realização de tarefas de configuração simples que serão necessárias no desenvolvimento de pequenas customizações.

Com foco neste objetivo, serão detalhadas as seguintes operações:

- Configuração e criação de novas tabelas no Dicionário de Dados.
- Atualização das estruturas do Dicionário de Dados.
  - o tabelas do sistema;
  - o validações de campos;
  - o índices de tabelas;
  - o gatilhos de campos.

Para contextualizar a estrutura da aplicação ERP no tópico a seguir é detalhada a forma como as tabelas de dados do Sistema estão divididas entre os diversos módulos que compõe o Sistema Protheus.

### 9.3.2. Estruturas básicas da aplicação ERP Protheus

#### Arquivos de configuração do sistema

Arquivo	Descrição
SIGAMAT	Cadastro de empresas e filiais do sistema
SIGAPSS	Arquivo de usuários, grupos e senhas do sistema
SIX	Índices dos arquivos
SX1	Perguntas e respostas
SX2	Mapeamento de tabelas
SX3	Dicionário de Dados

SX4	Agenda do Schedule de processos
SX5	Tabelas
SX6	Parâmetros
SX7	Gatilhos de Interface
SX8	Fora de uso
SX9	Relacionamentos entre tabelas
SXA	Pastas cadastrais apontadas no SX3
SXB	Consulta por meio da tecla F3 (Consulta Padrão)
SXD	Controle do Schedule de processos
SXE	Sequência de documentos (+1)
SXF	Sequência de documentos (Próximo)
SXG	Tamanho padrão para campos apontado pelo SX3
SXK	Resposta de Perguntas (SX1) por usuários
SXO	Controle de LOGs por tabela
SXP	Histórico de Logs cadastrados no SXO
SXQ	Cadastro de filtros inteligentes da mbrowse (contém as informações necessárias para a criação do filtro).
SXR	Cadastro de relacionamento entre programa x filtro (utilizada internamente pelo Protheus para verificar em
SXS	Cadastro de programas (utilizado na validação para mostrar/inibir os filtros na execução da mbrowse).
SXT	Tabela de usuários (contém as informações dos usuários que poderão utilizar os filtros da mbrowse).
SXOffice	Cadastro de relacionamento entre as entidades (tabelas) e as consultas TOII.

## Ambientes e tabelas

---

Na aplicação PROTHEUS as tabelas de dados podem ter uma estrutura mais simples e econômica, com tabelas em DBF/ADS do fabricante Extended System ou CTREE do fabricante FAIRCOM ou uma estrutura mais robusta e complexa, em bases SQL (SQLSERVER da Microsoft, ORACLE, DB II da IBM, SYBASE, MYSQL, POSTGREE, etc.).

No caso do SQL o acesso é feito através do TOPCONNECT / DBACCESS, que converte os comandos do ADVPL para este ambiente.

Para permitir uma utilização adequada das tabelas de dados do Sistema por cada um dos Ambientes da aplicação ERP, as tabelas foram divididas em grupos denominados "famílias". Cada módulo pode utilizar uma ou mais famílias de tabelas específicas para suas atividades, e ainda compartilhar informações com outros módulos, através de famílias comuns a todas as operações realizadas no Sistema.

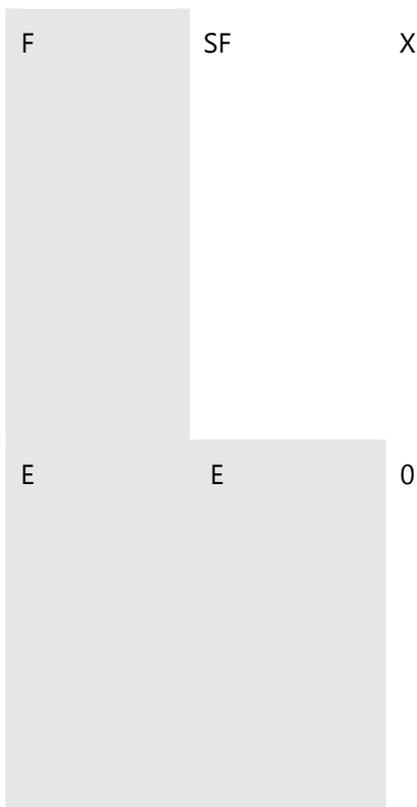
A tabela a seguir demonstra alguns dos módulos que compõe a aplicação ERP PROTHEUS atualmente:

<b>Ambiente</b>	<b>Identificação</b>
<b>SIGAATF</b>	ATIVO FIXO
<b>SIGACOM</b>	COMPRAS
<b>SIGACON</b>	CONTABILIDADE
<b>SIGAEST</b>	ESTOQUE E CUSTOS
<b>SIGAFAT</b>	FATURAMENTO
<b>SIGAFIN</b>	FINANCEIRO
<b>SIGAFIS</b>	LIVROS FISCAIS
<b>SIGAPCP</b>	PLANEJAMENTO E CONTROLE DA PRODUÇÃO
<b>SIGAGPE</b>	GESTÃO DE PESSOAL
<b>SIGAFAS</b>	FATURAMENTO DE SERVIÇOS
<b>SIGAVEI</b>	VEÍCULOS
<b>SIGALOJA</b>	CONTROLE DE LOJAS/AUTOMAÇÃO COMERCIAL
<b>SIGATMK</b>	CALL CENTER
<b>SIGAOFI</b>	OFICINAS
<b>SIGAPON</b>	PONTO ELETRÔNICO
<b>SIGAEIC</b>	EASY IMPORT CONTROL
<b>SIGATCF</b>	TERMINAL
<b>SIGAMNT</b>	MANUTENÇÃO DE ATIVOS
<b>SIGARSP</b>	RECRUTAMENTO E SELEÇÃO DE PESSOAL
<b>SIGAQIE</b>	INSPEÇÃO DE ENTRADA – QUALIDADE
<b>SIGAQMT</b>	METODOLOGIA – QUALIDADE

O nome de cada tabela no Protheus é constituído de seis dígitos que são utilizados para formar a seguinte representação:

F            X            X            E            E            0

Onde:



**Primeiro dígito representa a família, o segundo dígito pode ser utilizado para detalhar ainda mais a família especificada no primeiro nível (subfamília), e o terceiro dígito é a numeração sequencial das tabelas da família, iniciando em "0" e finalizando em "Z".**

**Os dois primeiros dígitos identificam a que empresa as tabelas estão vinculadas, lembrando que a informação de filial está contida nos dados da tabela.**

**O último dígito é fixo em "0".**

A tabela a seguir demonstra algumas das principais famílias de tabelas utilizadas pela aplicação ERP Protheus:

Família	Descrição
S -	<b>Tabelas pertencentes ao sistema básico, também chamado Classic</b>
S A	<b>Cadastros de entidades compartilhadas entre os Ambientes (Clientes, Fornecedores, Bancos entre outros)</b>

Família		Descrição
S	B	<b>Cadastros dos Ambientes de Materiais (Produtos, Saldos entre outros)</b>
S	C	<b>Arquivos de movimentações diversas, utilizados pelos ambientes de Materiais (Solicitação ao Almoxarifado, Solicitação de Compras, Pedido de Compras, Pedido de Vendas, Ordens de Produção entre outros)</b>
S	D	<b>Arquivos de movimentações de estoque (Itens de notas fiscais de entrada e saída, movimentos internos de estoque entre outros)</b>
S	E	<b>Cadastros e movimentações do ambiente Financeiro</b>
S	F	<b>Cadastros e movimentações Fiscais (Cabeçalhos das notas fiscais de entrada e saída, cadastro de tipos de entrada e saída, livros fiscais, entre outros)</b>
S	G	<b>Cadastros do Ambiente de Planejamento e Controle de Produção</b>
S	H	<b>Movimentos do Ambiente de Planejamento e Controle de Produção</b>
S	I	<b>Cadastros e movimentos do Ambiente Contábil (descontinuado)</b>
S	N	<b>Cadastros e movimentos do Ambiente Ativo Fixo</b>
S	R	<b>Cadastros e movimentos do Ambiente Gestão de Pessoal</b>
S	X	<b>Tabelas de configuração do Sistema</b>
S	Z	<b>Tabelas livres para utilização e projetos específicos em clientes</b>
A	-	<b>Gestão de Projetos</b>
C	-	<b>Contabilidade Gerencial</b>

Família		Descrição
C	T	<b>Contabilidade Gerencial</b>
C	V	<b>Contabilidade Gerencial</b>
C	W	<b>Contabilidade Gerencial</b>
D	-	<b>Transportadoras e derivados</b>
E	-	<b>Comércio exterior e derivados</b>
G	-	<b>Gestão Hospitalar</b>
J	-	<b>Gestão Educacional</b>
N	-	<b>Serviços Públicos</b>
P	-	<b>Reservado para projetos da fábrica de software</b>
Q	-	<b>Qualidade e derivados</b>
R	-	<b>Recursos Humanos e derivados</b>
T	-	<b>Plano de Saúde</b>
W	-	<b>Workflow</b>
Z	-	<b>Tabelas livres para utilização e projetos específicos em clientes em adição à família SZ</b>

## Índices

---

Cada tabela do Sistema possui seus índices definidos no arquivo de configuração SIX, o qual pode ser atualizado com a utilização do módulo Configurador.

Os arquivos de índices das tabelas de Sistema serão criados de acordo com o banco de dados utilizado (ISAM ou conexão via TOPCONNECT).

Para os bancos de dados ISAM serão gerados arquivos com a mesma nomenclatura da tabela de dados, mas com uma extensão diferenciada (atualmente .CDX). No caso da utilização de um banco de dados, cada índice será uma numeração sequencial em função do nome da tabela original.

As especificações das chaves de índices de cada um das tabelas está disponível no arquivo de sistema SIX, e a chave única da tabela utilizada para banco de dados está descrita na tabela SX2.

## Menus

---

Cada módulo da aplicação ERP possui um menu padrão com todas as funcionalidades disponíveis para o Ambiente, menu este definido através de sintaxe XML (arquivos .XNU).

Os menus possuem uma estrutura padrão que permite ao usuário localizar e identificar facilmente cada uma das funcionalidades oferecidas pelo Ambiente.

### 9.3.3. Acessando o módulo Configurador

Para executar o módulo Configurador é necessário que a aplicação Protheus Server esteja em execução e através da aplicação Protheus Remote deverá ser informada como programa inicial a opção SIGACFG.



**Figura: Parâmetros de inicialização do Sistema**

Após a confirmação, a validação do acesso é feita conforme tela ilustrada a seguir:

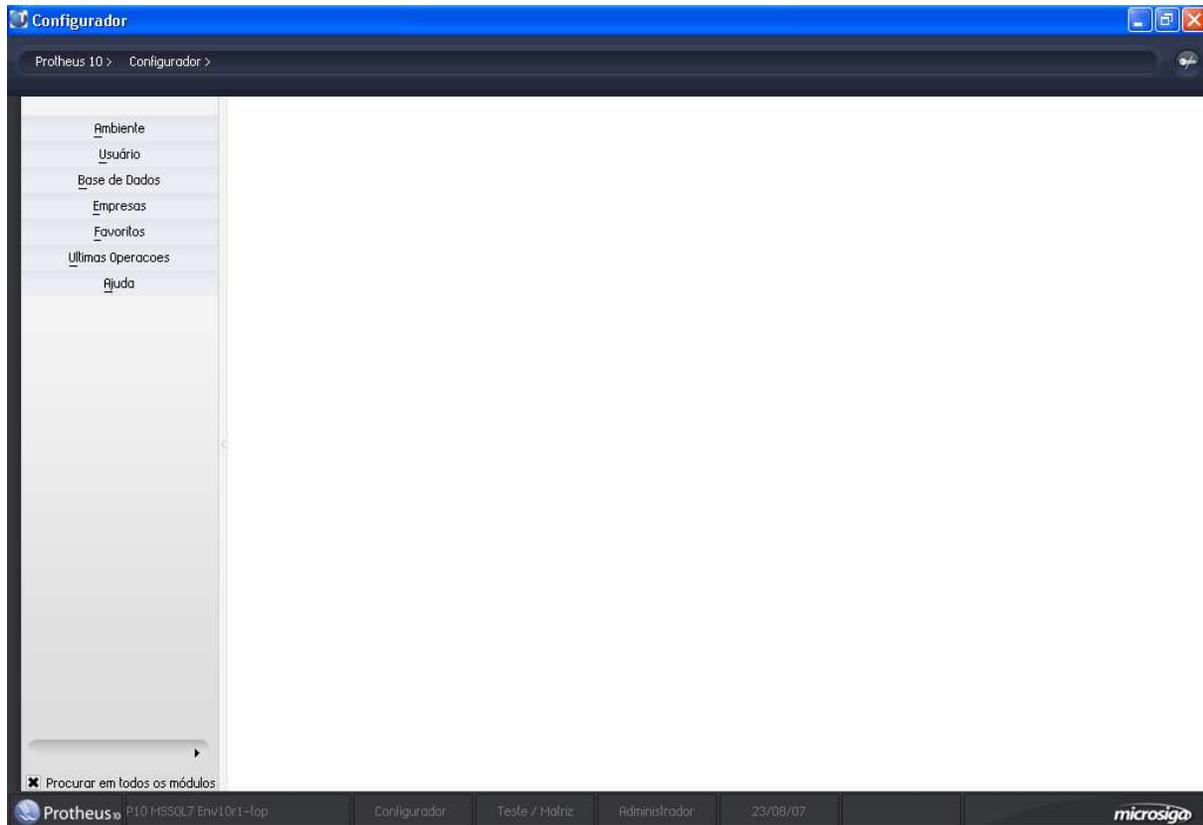


**Figura: Validação de acesso**



### **Figura: Confirmação do acesso ao módulo Configurador**

Logo após a sua confirmação do usuário e senha com direito de administrador, será apresentada a tela inicial do configurador, conforme mostra a figura a seguir:



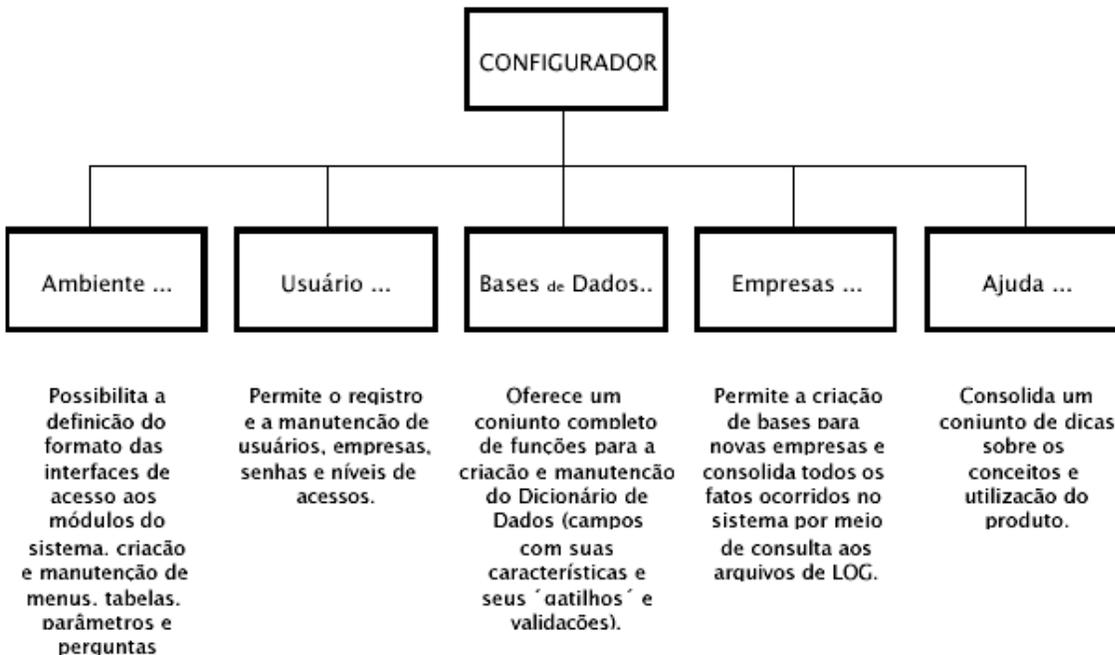
### **Figura: Interface principal do módulo Configurador**

## **9.4. funcionalidades do Configurador**

A customização de um Sistema como o Protheus consiste em adaptar o Sistema de forma a melhor atender as necessidades do cliente.

A flexibilidade de um Sistema, ou seja, sua capacidade de adaptar-se (polimorfismo, aquele que assume várias formas) é uma das mais importantes características de uma solução ERP.

As funcionalidades tratadas pelo Configurador definem a flexibilidade do ERP Protheus. Flexibilizar sem despadronizar, ou seja, tudo que foi customizado permanece válido, mesmo com o desenvolvimento de novas versões.



**Figura: Principais funcionalidades do módulo Configurador**

O Configurador é o programa básico para o processo de customização do Protheus, através da alteração das tabelas da família SX. Neles, o usuário ou o analista de suporte responsável pela implantação configura as informações que serão utilizadas pelos demais ambientes do Sistema.

Essas informações vão de simples parâmetros até complexas expressões e comandos que são interpretados em tempo de execução.

Nos próximos tópicos serão abordadas as funcionalidades de customização disponíveis no Ambiente Configurador, relevantes ao objetivo de desenvolvimento de pequenas customizações para a aplicação ERP.

### 9.4.1. Dicionário de Dados da aplicação ERP

A idéia do Dicionário de Dados é permitir que o usuário possa incluir ou inibir campos, ou mesmo alterar as propriedades dos campos existentes. Pode, ainda, criar novas tabelas. Ou seja, os programas ao invés de terem os campos definidos em seu código original, lêem o Dicionário em tempo de execução, montando arrays com as propriedades de cada um. A partir daí, sua utilização é normal, através do uso de funções do ADVPL que tornam o trabalho do desenvolvedor transparente a esta arquitetura.

O objetivo do Dicionário de Dados é permitir que o próprio usuário crie novas tabelas ou altere os campos nas tabelas existentes quanto ao seu uso, sua ordem de apresentação, legenda (nos três idiomas), validação, help, obrigatoriedade de preenchimento, inicialização etc.



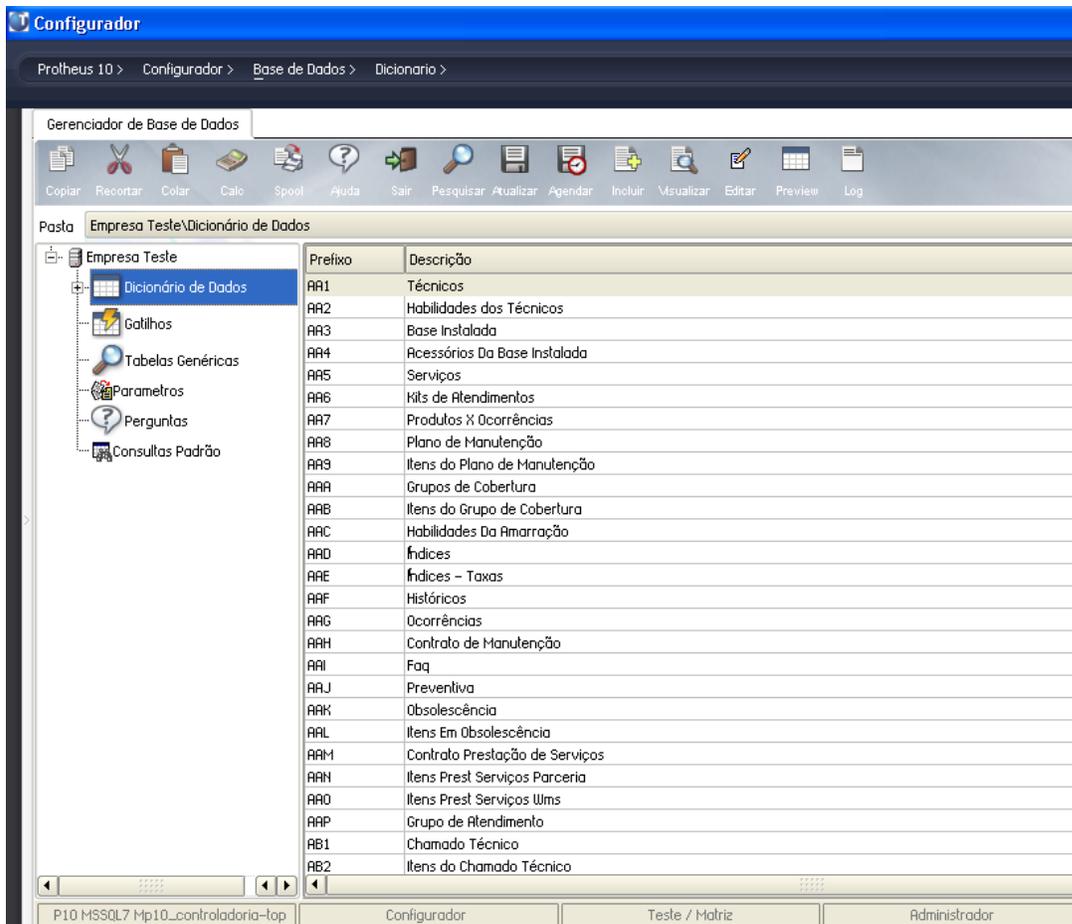
**Figura: Conjunto de pacotes que compõe o Dicionário de Dados**

## 9.4.2. Adição de tabelas ao Dicionário de Dados

### Procedimento

---

1. Para adicionar uma tabela ao dicionário de dados de uma empresa, selecione a opção Dicionário de Dados abaixo da empresa que será atualizada. (árvore de opções da parte esquerda da interface visual do Gerenciador de Bases de Dados).
2. Após a seleção da opção Dicionário de Dados serão exibidas as tabelas já cadastradas no arquivo de sistema SX2.

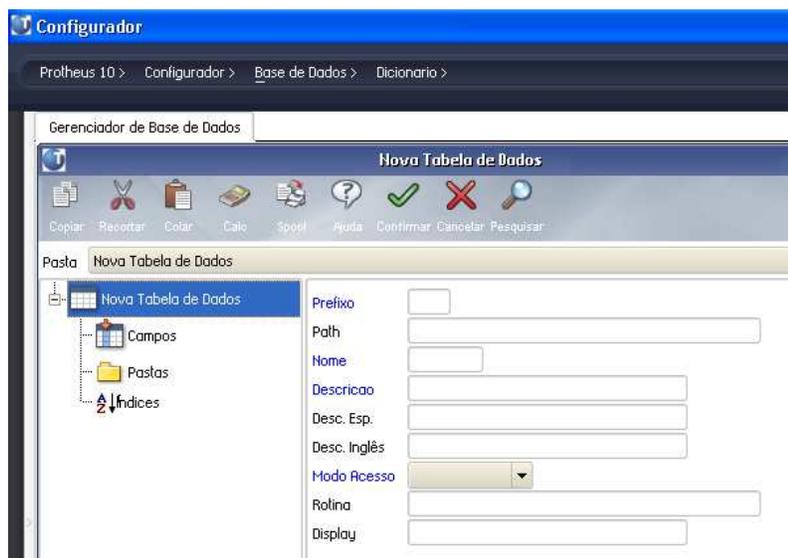


**Figura: Conjunto de tabelas já cadastradas no SX2**

3. Após a visualização das tabelas já cadastradas no SX2 da empresa selecionada, utilize



o botão Incluir ( ). Ao utilizar esta opção será exibida a tela para definição dos dados referentes à nova tabela que será criada:



**Figura: Cadastro de uma nova tabela**

4. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento da nova tabela com o botão Confirmar



### **Orientações para o cadastramento de uma nova tabela**

---

- O domínio SZ1 até SZZ (considerando todos os número e todas as letras no último byte) é reservado para dados exclusivos do usuário, pois esse intervalo não será utilizado pelo Sistema. Caso seja necessário o domínio Z00 a ZZZ também pode ser empregado para desenvolvimentos específicos do cliente.



Fique  
atento

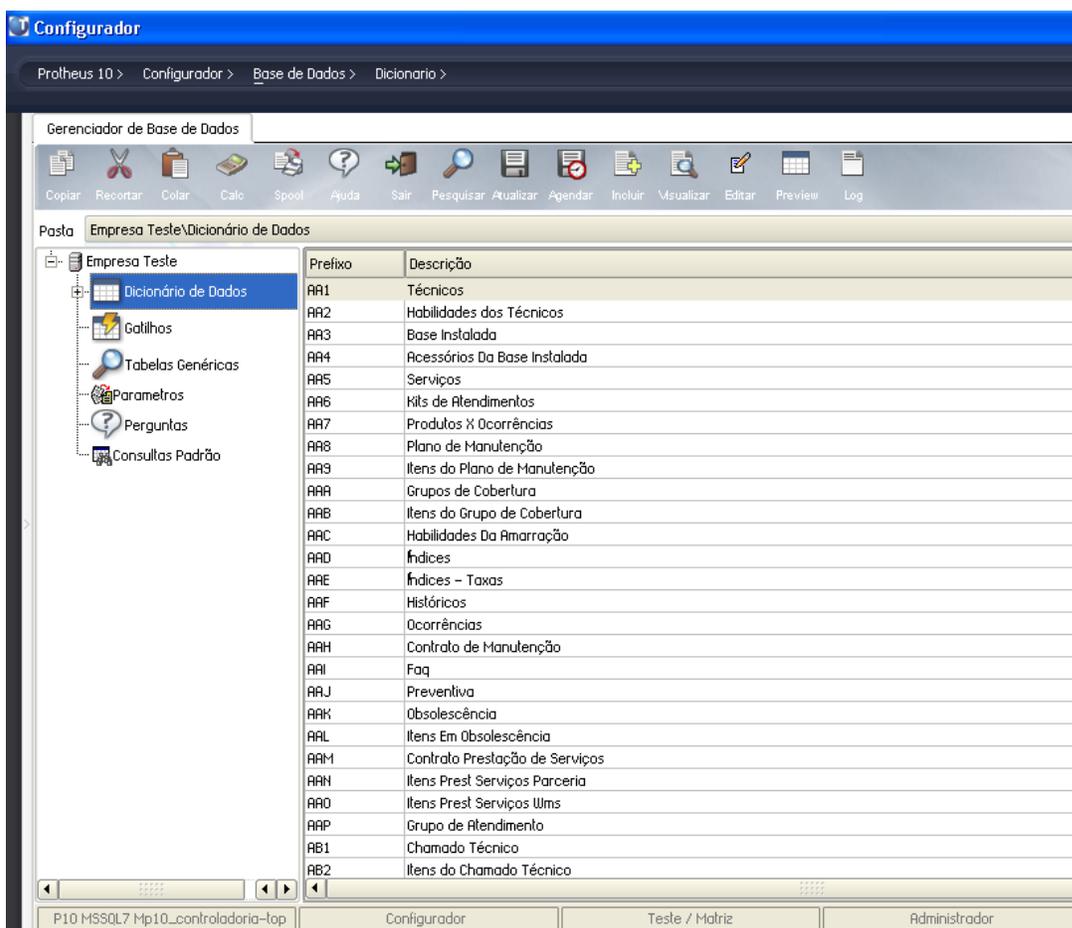
***Não devem ser criadas tabelas específicas de clientes com quaisquer outras nomenclaturas, o que pode afetar diretamente um processo de atualização futuro.***

- O nome da tabela é preenchido automaticamente, adicionando 990. Esse dado refere-se à empresa 99 (Teste Matriz) a qual está sendo adicionado à tabela.
- O Path refere-se à pasta que conterá efetivamente os dados das tabelas, quando ISAM, As versões com banco de dados relacional não são utilizadas. Essa pasta será criada dentro da pasta indicada na configuração do Sistema como ROOTPATH.
- O modo de acesso compartilhado indica que o Sistema possibilitará o uso simultâneo da tabela por duas ou mais filiais. Se for compartilhado o campo Filial fica em branco. Se for exclusivo, grava-se o código da filial ativa e somente ela tem acesso ao registro.
- Após a confirmação, a tabela criada passa a fazer parte do cadastro do Dicionário de Dados, contendo somente o campo FILIAL, o qual é criado como padrão pela funcionalidade do módulo.

### 9.4.3. Adição de campos às tabelas do Dicionário de Dados

#### Procedimento

1. Para adicionar um campo a uma tabela do dicionário de dados de uma empresa, selecione a opção Dicionário de Dados abaixo da empresa que será atualizada. (árvore de opções da parte esquerda da interface visual do Gerenciador de Bases de Dados).
2. Após a seleção da opção Dicionário de Dados serão exibidas as tabelas já cadastradas no arquivo de sistema SX2.

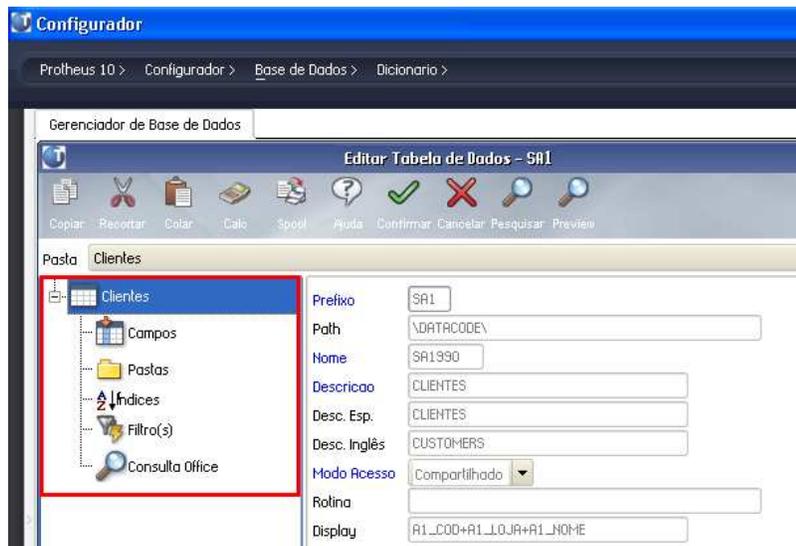


**Figura: Conjunto de tabelas já cadastradas no SX2**

3. Após a visualização das tabelas já cadastradas no SX2 da empresa selecionada,

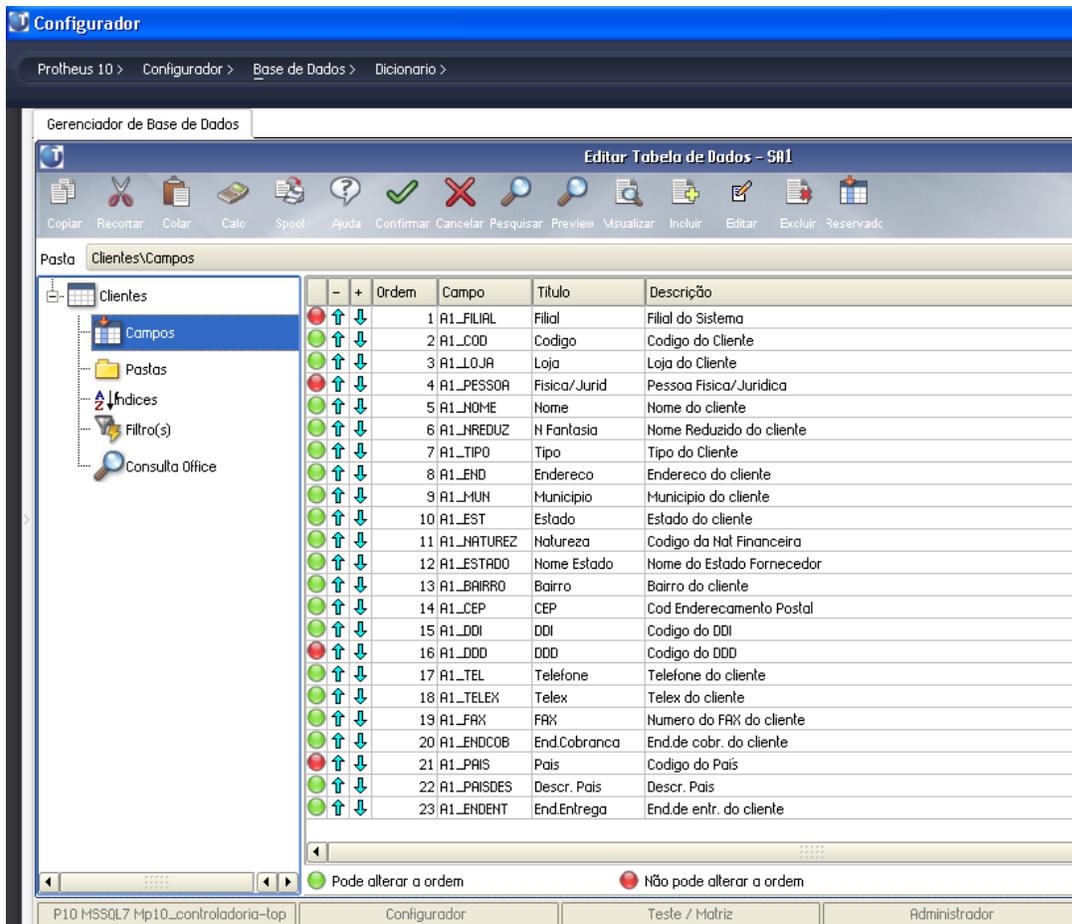


localize e selecione a tabela que será atualizada, e utilize o botão Editar (  ). Ao utilizar esta opção será exibida a tela de manutenção de campos da tabela selecionada:



**Figura: Estrutura de uma tabela já cadastrada no Sistema**

4. Selecione a opção Campos (  Campos ), para que sejam exibidos os campos disponíveis para a tabela no arquivo de sistema SX3.



**Figura: Estrutura de campos de uma tabela já cadastrada no Sistema**

5. Após a visualização dos campos já cadastrados no SX3 da tabela selecionada, utilize a



opção Incluir ( ). Ao utilizar esta opção será exibida a tela para definição dos dados referentes ao novo campo que será criado:



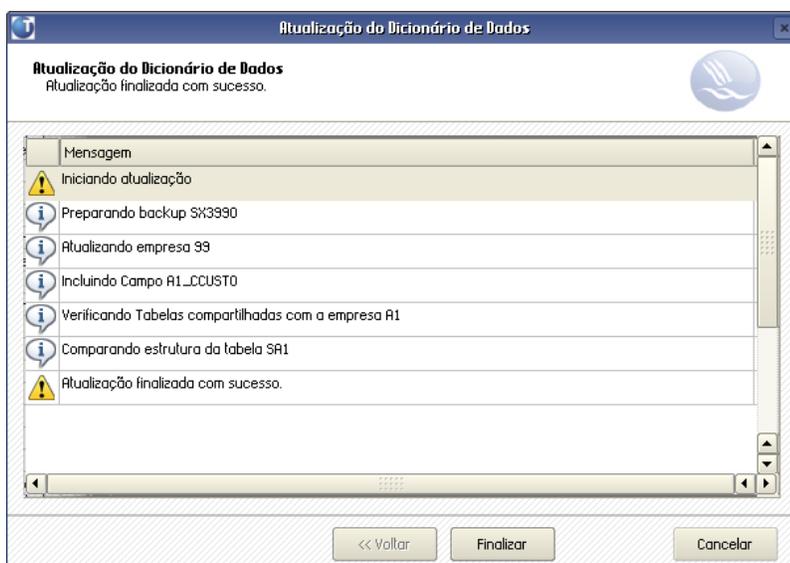
### **Figura: Dados para parametrização de um novo campo no Sistema**

5. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento do novo campo para a tabela com o

botão Confirmar ()

6. Confirme as atualizações para a tabela selecionada com o botão Confirmar ()

7. Atualize as configurações do Sistema com o botão Atualizar ()



**Figura: Atualização dos dados do Sistema**

### **Orientações para o cadastramento de um novo campo**

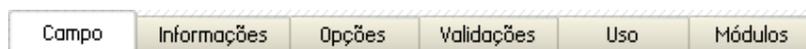
---

1. As propriedades definidas no Dicionário de Dados (SX3) são as seguintes:

- Nome do campo:** Todos os campos têm como prefixo o próprio nome da tabela, sendo que para tabelas da família "S", o prefixo do campo é composto apenas pelos próximos dois dígitos. No caso das demais tabelas, o prefixo do campo serão os três dígitos identificadores da tabela.

- Tipo do campo:** Indica se é caractere, numérico, lógico, data ou memo. É claro que a mudança do tipo de campo deve ser feita com muito cuidado, pois, se tivermos um campo numérico usado em cálculos e ele for alterado para caractere, certamente teremos um erro.
- 
- Tamanho do campo:** Também aqui é necessário certo cuidado ao alterá-lo, pois poderemos ter truncamentos em relatórios e consultas em que há espaço para conteúdos maiores que o original.
  
- Formato de edição:** Define como o campo aparece nas telas e nos relatórios.
  
- Contexto:** Pode ser real ou virtual. O contexto virtual cria o campo somente na memória e não na tabela armazenada no disco. Isso é necessário porque os programas de cadastramento e de consulta genérica apresentam somente uma tabela de cada vez. Assim, se quisermos apresentar um campo de uma outra tabela, ou mesmo o resultado de um cálculo, sem que tal informação ocupe espaço físico no HD, utilizamos o contexto virtual. Campos virtuais normalmente são alimentados por gatilhos.
  
- Propriedade:** Indica se um campo pode ou não ser alterado pelo usuário. Exemplo: saldos normalmente não podem, pois quem cuida dessa tarefa são os programas.

2. Demais características que devem ser observadas na configuração do campo:



### **Guia: Campo**

- O campo Decimal será solicitado somente para os campos de tipo numérico.
  
- O formato "!" indica que o caractere será sempre maiúsculo, independente da ação do usuário. O formato "@!" indica que essa característica estende-se por todo o campo.
  
- O contexto real indica que o campo existirá efetivamente no banco de dados e o contexto virtual significa que o campo existirá apenas no dicionário de dados e não fisicamente.

- A propriedade alterar indica que o campo pode ser alterado.
  
- Nesta janela, os dados estão classificados em seis pastas com objetivos de preenchimento bem específicos:

### **Guia: Informações**

- Contém as informações a respeito dos títulos.
  
- Título:** É a legenda que aparece nas telas/relatórios. Há inclusive três campos para esta finalidade: em português, espanhol e inglês. Esta propriedade pode ser alterada à vontade, pois não interfere em nenhum processamento.
  
- Descrição e Help:** São propriedades que objetivam documentar o campo.

### **Guia: Opções**

- Contém os dados que facilitam a digitação.

### **Guia: Validações**

- Representam as regras de validação do campo.
  
- Validações:** Nesta propriedade, escreve-se uma função de validação do campo que está sendo digitado. Existe um conjunto de funções disponíveis no ADVPL apropriadas para esse caso.
  
- Todas as validações informadas serão executadas no momento do preenchimento do próprio campo. Uma validação pode ser uma expressão lógica ou uma função de usuário que retorna um valor lógico Verdadeiro ou Falso. O sistema só permitirá o avanço para o próximo campo quando o respectivo preenchimento resultar Verdadeiro, seja na expressão ou no retorno da função.

### **Guia: Uso**

- Descreve a forma de utilização do campo.

**Guia: Módulos**

- Relaciona todos os módulos em que o campo será utilizado.



**Anotações**

---

---

---

---

---

#### **9.4.4. Adição de índices para as tabelas do Dicionário de Dados**

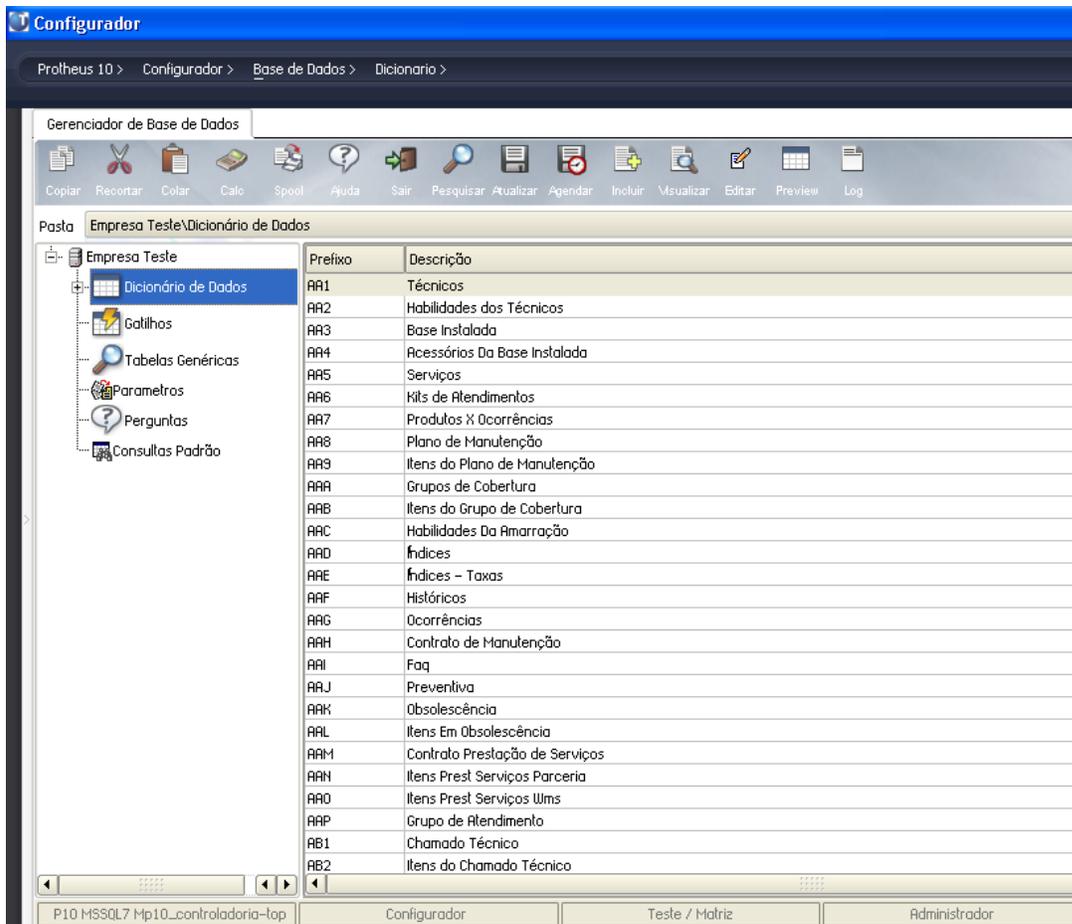
Conforme mencionado anteriormente, no Ambiente Protheus uma tabela pode ter vários índices, os quais serão gerados de acordo com o banco de dados configurado para o Sistema.

Os índices do Sistema auxiliam na seleção e obtenção de informações da base de dados, além de determinar a ordem de apresentação dos registros de uma tabela em consultas e relatórios.

##### **Procedimento**

---

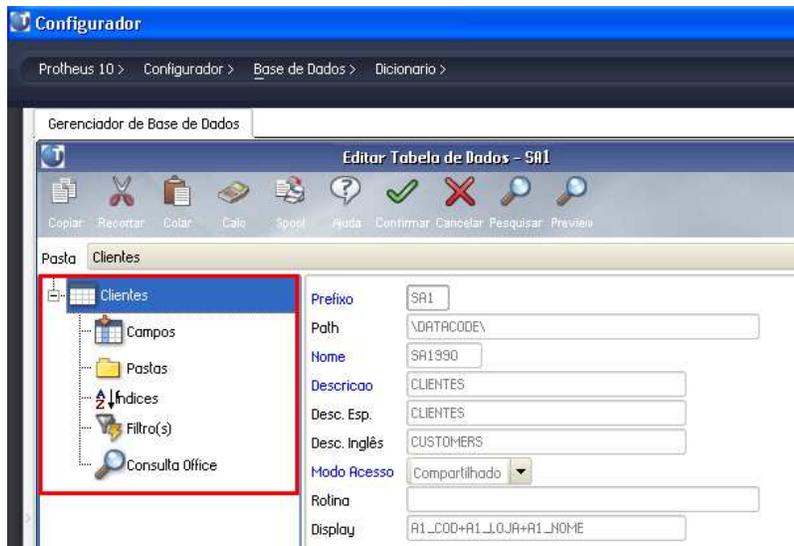
1. Para adicionar um índice a uma tabela do dicionário de dados de uma empresa, selecione a opção Dicionário de Dados abaixo da empresa que será atualizada. (árvore de opções da parte esquerda da interface visual do Gerenciador de Bases de Dados).
2. Após a seleção da opção Dicionário de Dados serão exibidas as tabelas já cadastradas no arquivo de sistema SX2.



**Figura: Conjunto de tabelas já cadastradas no SX2**

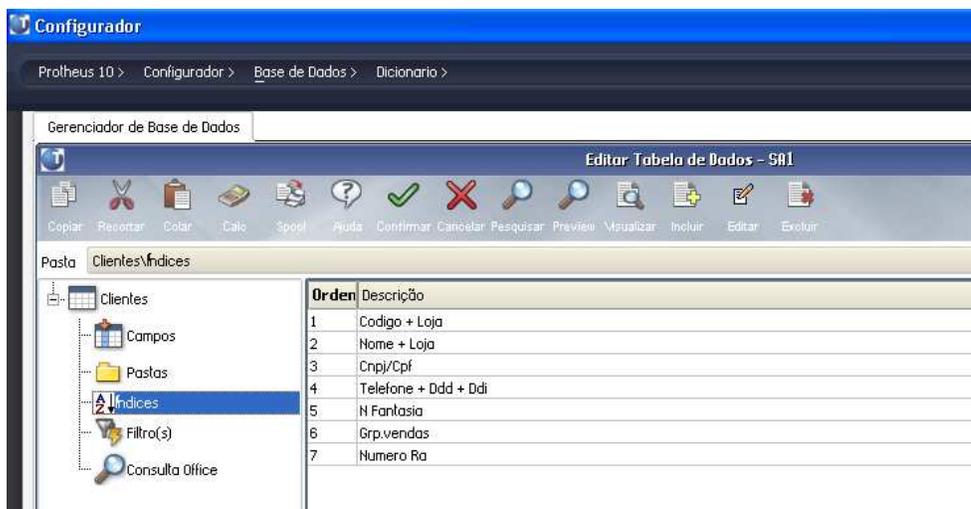
3. Após a visualização das tabelas já cadastradas no SX2 da empresa selecionada,

localize e selecione a tabela que será atualizada, e utilize o botão Editar (). Ao utilizar esta opção será exibida a tela de manutenção de campos da tabela selecionada:



**Figura: Estrutura de uma tabela já cadastrada no Sistema**

4. Selecione a opção índices (  ), para que sejam exibidos os índices disponíveis para a tabela no arquivo de sistema SIX.



**Figura: Índices disponíveis para a tabela no arquivo de sistema SIX**

5. Após a visualização dos índices já cadastrados no SIX para a tabela selecionada, utilize



a opção Incluir (  ). Ao utilizar esta opção, será exibida a tela para definição dos dados referentes ao novo índice que será criado:

**Figura: Adição de um índice para uma tabela**

6. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento do novo índice para a tabela com o botão



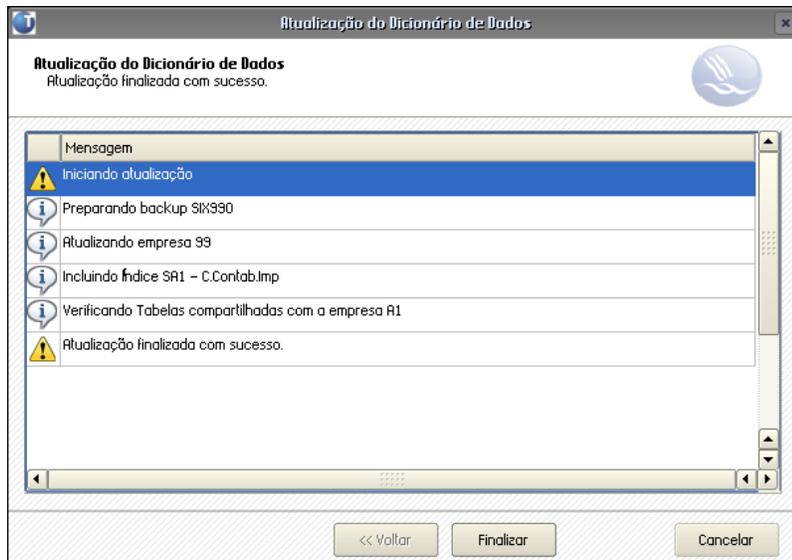
Confirmar (  ).

7. Confirme as atualizações para a tabela selecionada com o botão Confirmar (  ).



8. Atualize as configurações do Sistema com o botão Atualizar (  ).





**Figura: Atualização dos dados do Sistema**

## Orientações para o cadastramento de um novo índice

---

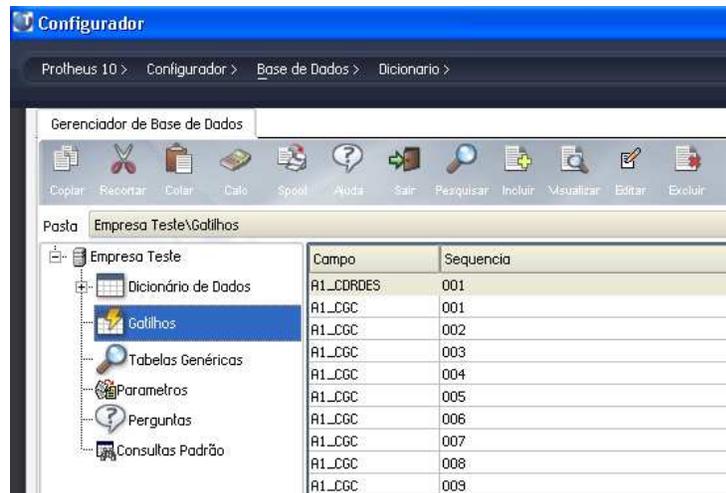
- ☑ O **Nickname** é uma identificação complementar do índice o qual pode ser utilizada para auxiliar o desenvolvedor na utilização desta ordem em uma aplicação, a qual pode ser padrão do Sistema ou específica de um cliente.
  
- ☑ Para selecionar os campos já cadastrados na tabela, pode ser utilizado o botão  Campos (  ). Esta facilidade preenche, automaticamente, os campos de descrição.
  
- ☑ O campo relativo à filial sempre faz parte dos índices, com exceção do SM2, para que os registros nas tabelas estejam agrupados por filiais, independente desta tabela ser compartilhada entre as filiais.
  
- ☑ Uma tabela poderá ter vários índices cadastrados no Dicionário de Dados. Em determinado momento, porém, apenas um deles oferecerá acesso ao registro. Essa ordem pode ser alterada em tempo de execução pelos programas da aplicação, através do comando DBSetOrder(), ou através da definição de uma ordem específica na utilização de queries para acesso aos dados diretamente em bancos de dados de Ambientes TOPCONNECT (DbAccess).

### 9.4.5. Adição de gatilhos para os campos das tabelas do Sistema

#### Procedimento

---

1. Para adicionar um gatilho a um campo de uma tabela do dicionário de dados de uma empresa, selecione a opção Gatilho abaixo da empresa que será atualizada. (árvore de opções da parte esquerda da interface visual do Gerenciador de Bases de Dados).
  
2. Após a seleção da opção Gatilhos serão exibidos os itens já cadastradas no arquivo de sistema SX7.



**Figura: Conjunto de gatilhos já cadastros no SX7**

3. Após a visualização dos gatilhos já cadastrados no SX7 da empresa selecionada, utilize



o botão Incluir ( ) para realizar o cadastro de um novo gatilho no Sistema:

**Figura: Dados para o cadastro de um novo gatilho no Sistema**

4. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento do novo gatilho de sistema com o



botão Confirmar ( ).

### **Orientações para o cadastramento de um novo gatilho**

---

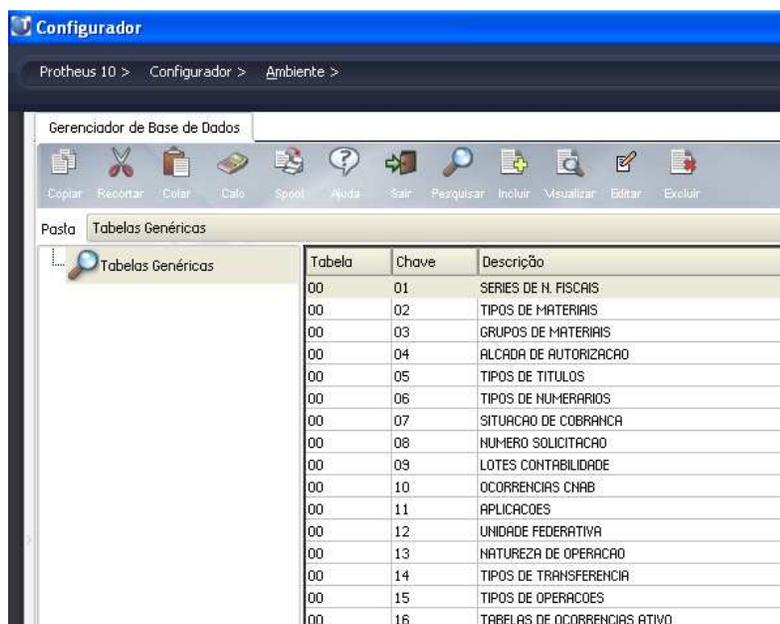
- Pode haver vários gatilhos para o mesmo campo. A ordem de execução é determinada pelo campo Sequência.
- Os tipos do Gatilho Primário, Estrangeiro e de Posicionamento definem se o Contra Domínio é um campo da mesma tabela, de outra tabela ou se o gatilho deve realizar um posicionamento, respectivamente.
- A regra pode ser uma expressão que resulta em um valor a ser preenchido no Contra Domínio.

- O posicionamento igual a Sim indica que será executado um comando de busca do registro de acordo com a chave indicada.
  
- O Alias, a Ordem e a Chave descrevem a tabela envolvida no gatilho, seu índice e a chave para que a funcionalidade se posicione no registro adequado.

## 9.4.6. Criação de Tabelas Genéricas

### Procedimento

1. Para adicionar uma tabela genérica, selecione os menus Ambiente, Cadastros, Tabelas.
2. Após a seleção da opção Tabelas serão exibidos os itens já cadastradas no arquivo de sistema SX5.



The screenshot shows the 'Configurador' window in Protheus 10. The breadcrumb path is 'Protheus 10 > Configurador > Ambiente > Gerenciador de Base de Dados'. The 'Pastas' (Folders) pane on the left shows 'Tabelas Genéricas' selected. The main area displays a table with the following data:

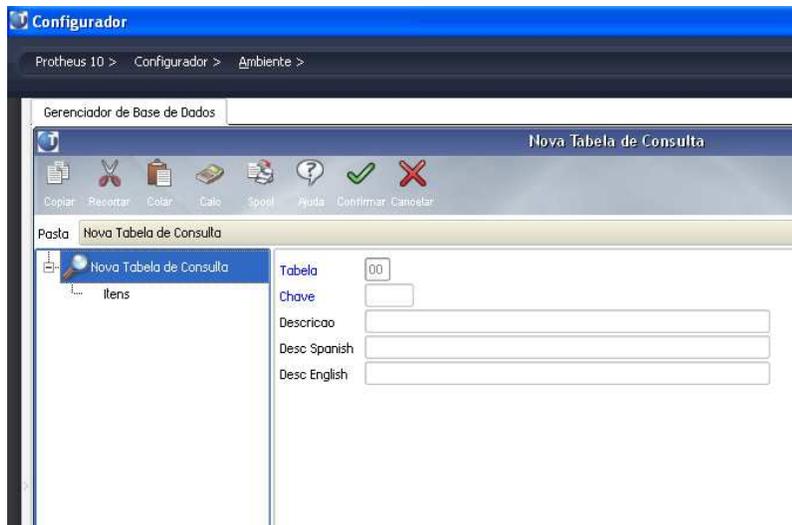
Tabela	Chave	Descrição
00	01	SERIES DE N. FISCAIS
00	02	TIPOS DE MATERIAIS
00	03	GRUPOS DE MATERIAIS
00	04	ALCADA DE AUTORIZACAO
00	05	TIPOS DE TITULOS
00	06	TIPOS DE NUMERARIOS
00	07	SITUACAO DE COBRANCA
00	08	NUMERO SOLICITACAO
00	09	LOTES CONTABILIDADE
00	10	OCORRENCIAS CNAB
00	11	APLICACOES
00	12	UNIDADE FEDERATIVA
00	13	NATUREZA DE OPERACAO
00	14	TIPOS DE TRANSFERENCIA
00	15	TIPOS DE OPERACOES
00	16	TABELAS DE OCORRENCIAS ATIVO

**Figura: Conjunto de Tabelas já cadastradas no SX5**

3. Após a visualização das tabelas já cadastrados no SX5 da empresa selecionada, utilize



o botão Incluir ( ) para realizar o cadastro de uma nova tabela no Sistema:



**Figura: Dados para o cadastro de uma nova tabela no Sistema**

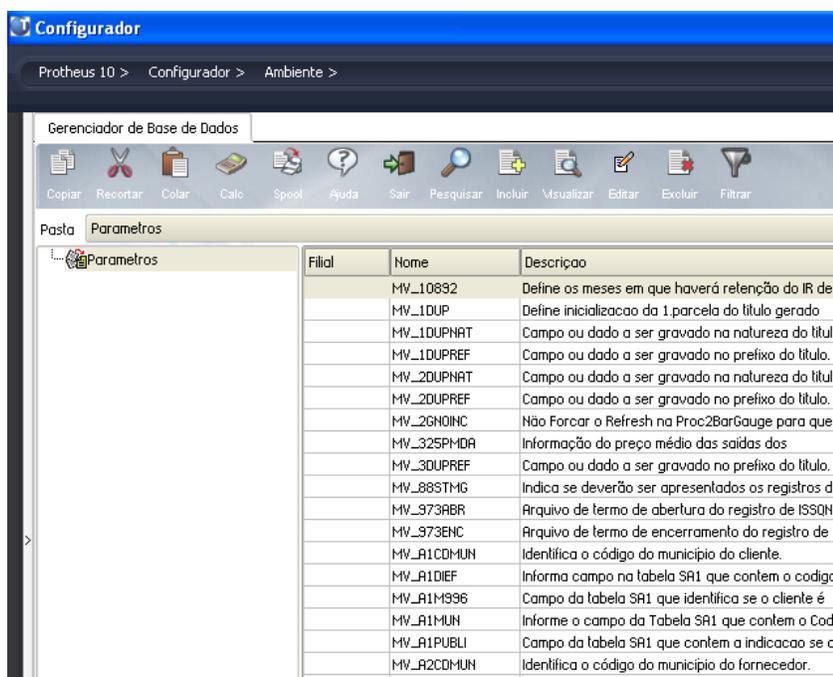
4. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento da nova tabela do sistema com o

botão Confirmar ().

## 9.4.7. Criação de Parâmetros

### Procedimento

1. Para adicionar um Parâmetro, selecione os menus Ambiente, Cadastros, Parâmetros.
2. Após a seleção da opção Tabelas serão exibidos os itens já cadastradas no arquivo de sistema SX6.



Filial	Nome	Descrição
	MV_10832	Define os meses em que haverá retenção do IR de
	MV_1DUP	Define inicializacao da 1.parcela do titulo gerado
	MV_1DUPNAT	Campo ou dado a ser gravado na natureza do titulo
	MV_1DUPREF	Campo ou dado a ser gravado no prefixo do titulo.
	MV_2DUPNAT	Campo ou dado a ser gravado na natureza do titulo
	MV_2DUPREF	Campo ou dado a ser gravado no prefixo do titulo.
	MV_2GNOINC	Não Forçar o Refresh na Proc2BarGauge para que
	MV_325PMDA	Informação do preço médio das saidas das
	MV_3DUPREF	Campo ou dado a ser gravado no prefixo do titulo.
	MV_88STMG	Indica se deverão ser apresentados os registros de
	MV_973ABR	Arquivo de termo de abertura do registro de ISSQN
	MV_973ENC	Arquivo de termo de encerramento do registro de
	MV_A1CDMUN	Identifica o código do municipio do cliente.
	MV_A1DIEF	Informa campo na tabela SA1 que contem o codigo
	MV_A1M996	Campo da tabela SA1 que identifica se o cliente é
	MV_A1MUN	Informe o campo da Tabela SA1 que contem o Codi
	MV_A1PUBLI	Campo da tabela SA1 que contem a indicacao se o
	MV_A2CDMUN	Identifica o código do municipio do fornecedor.

**Figura: Conjunto de Parâmetros já cadastrados no SX6**

3. Após a visualização dos Parâmetros já cadastrados no SX6 da empresa selecionada,



utilize o botão Incluir ( ) para realizar o cadastro de uma nova tabela no sistema:

The image shows a software dialog box titled "Novo Parametro". At the top, there is a menu bar with icons and labels for "Copiar", "Reortar", "Colar", "Calc", "Spool", "Ajuda", "OK", and "Cancelar". Below the menu bar, there are two tabs: "Informacoes" (selected) and "Descricao". The "Informacoes" tab contains several input fields: "Filial" with a dropdown arrow, "Nome da Var." with a text box, "Tipo" with a dropdown arrow, "Cont. Por" with a text box, "Cont. Ing" with a text box, and "Cont. Esp" with a text box.

**Figura: Dados para o cadastro de um novo parâmetro no Sistema**

4. Realize o preenchimento das informações solicitadas de acordo com as orientações a seguir, e ao término confirme o cadastramento do novo Parâmetro do Sistema com o

botão Confirmar ().

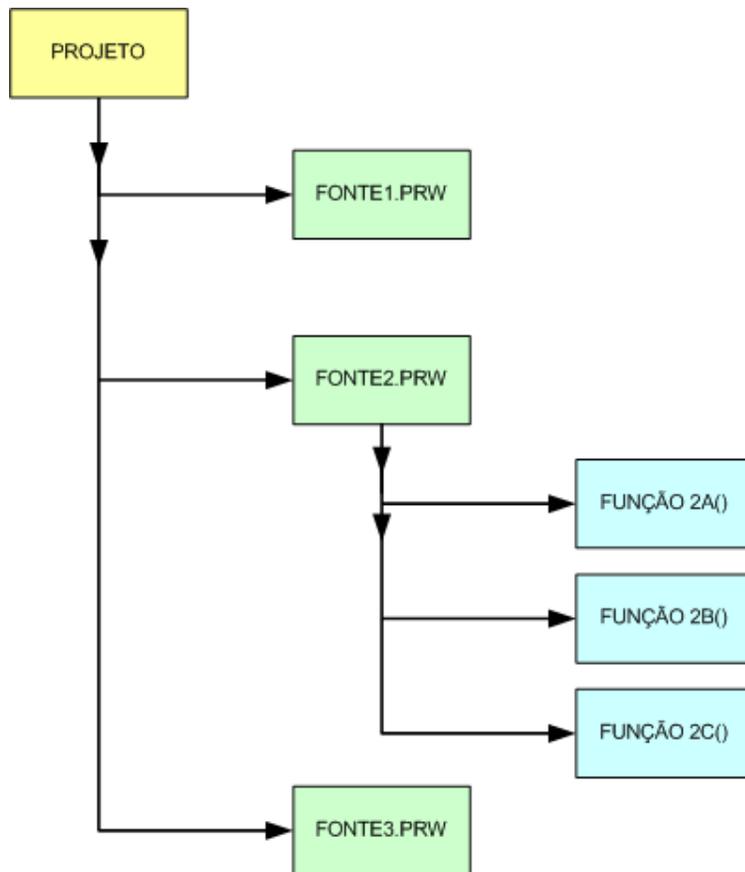
## 10. TOTVS DEVELOPMENT STUDIO

A ferramenta TOTVS Development Studio é um programa que faz parte do Protheus e permite o trabalho de edição, compilação e depuração de programas escritos em ADVPL.

### Projeto

Um programa para ser compilado deve ser vinculado a um projeto. Normalmente, programas que fazem parte de um determinado módulo ou ambiente estão em um mesmo projeto.

A vinculação dos programas a um projeto é feita por meio dos arquivos do tipo PRW. Na verdade, um projeto pode ser constituído de um ou mais arquivos deste tipo, que por sua vez, podem ter uma ou mais funções, conforme ilustra o diagrama a seguir:



**Figura: Representação da estrutura de um projeto no DEV-Studio**

## Compilação

---

Uma vez adicionado a um projeto e compilado sem incidências de erros de código, o objeto resultante será cadastrado no RPO (Repositório de Objetos) e poderá ser utilizado pela aplicação ERP.

A compilação dos itens de um projeto pode ser realizada individualmente, por grupo de fontes (pastas) ou ainda selecionando-o inteiramente. Cada um dos fontes será processado e compilado separadamente, permitindo a visualização do progresso da operação e das mensagens de aviso (warnings) ou erros (critical errors) na guia Mensagens.

## Execução

---

Para que os objetos compilados e disponíveis n RPO sejam utilizados, devem ser observadas as seguintes regras:

- Se o programa não manipula arquivos, pode-se chamá-lo diretamente do DEV-Studio (nome no lado direito da barra de ferramentas);
  
- Se o programa manipula tabelas existem duas opções:
  - Adicionar o programa no menu de um dos Ambientes e executá-lo através do Remote.
  
  - Realizar a preparação do Ambiente na própria rotina, permitindo sua execução diretamente pelo DEV-Studio.
  
- Não se pode compilar um programa com o Remote e o Monitor abertos, tenha este finalizado ou não por erro.

## Análise e depuração de erros

---

Para identificar as causas de erros, a ferramenta DEV-Studio possui diversos recursos que auxiliam o DEBUG.

A ação de DEBUG necessita que o programa seja executado a partir do DEV-Studio, sendo necessário observar as seguintes regras:

- Definir e marcar os pontos de parada mais adequados a análise do fonte;
- Executar a rotina através do DEV-Studio, selecionando seu nome diretamente, ou o módulo que contém a opção ou a ação que a executará;
- A partir do momento em que o DEV-Studio pausar o processamento em um dos pontos de parada especificados previamente, podem ser utilizadas as janelas de visualização, disponíveis no DEV-Studio, que são:

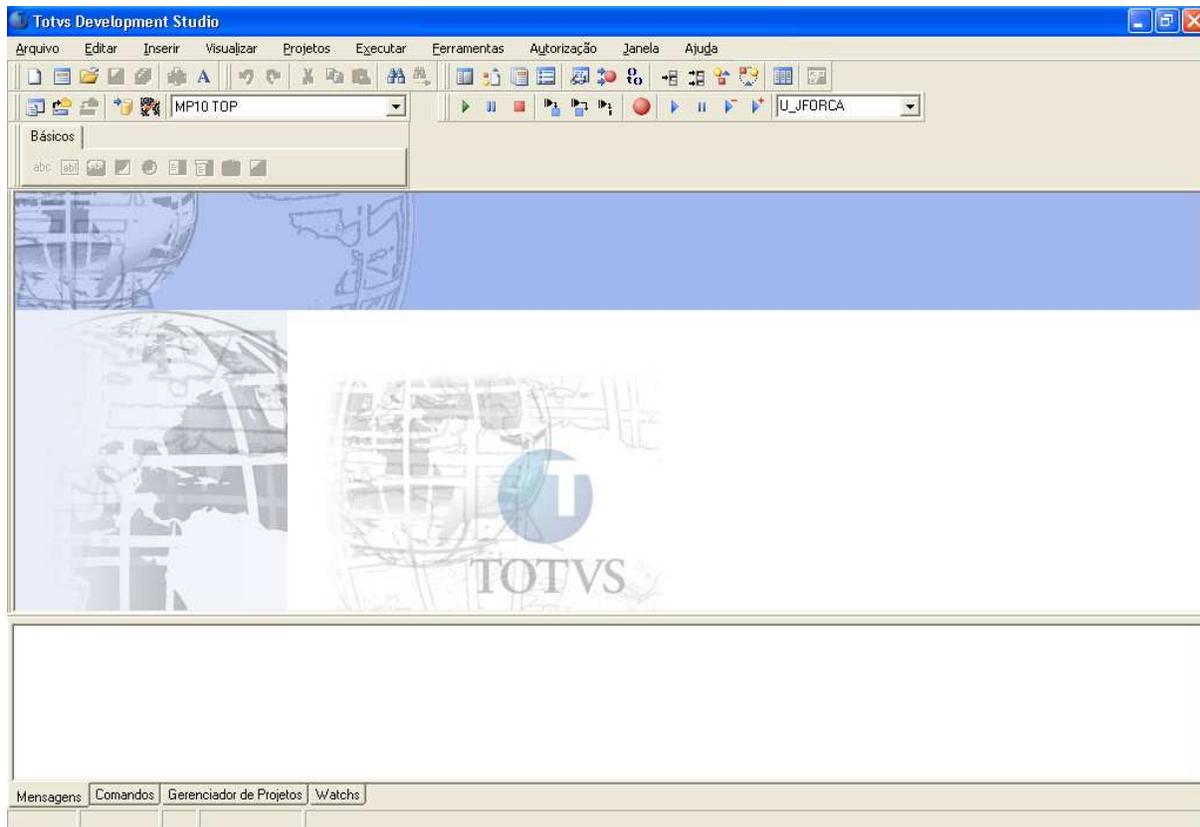
-  Variáveis Locais
-  Variáveis Privates
-  Variáveis Public's
-  Variáveis Static's
-  Janela da Watch's
-  Janela de Tabelas e Campos
-  Pilha de Chamadas

- Através da Janela de Watch's é possível determinar quais variáveis devem ser exibidas;
- Na pilha de chamadas, verifica-se a sequência de chamadas das funções;
- Na pasta de Comandos, pode-se, enquanto o programa estiver pausado, escrever qualquer comando e ao dar Enter, ele é executado, permitindo pesquisar palavras e expressões no próprio fonte ou em qualquer fonte armazenado no HD;
- Ao parar pode-se ou continuar o programa até o próximo ponto de parada, caso haja um outro definido, ou executar linha a linha da rotina.

## **Interface da aplicação**

---

Por ser um ambiente integrado de desenvolvimento, o DEV-Studio proporciona todas essas facilidades, por meio de interface única como ilustra a figura a seguir:



**Figura: Interface principal do TOTVS Development Studio**

- O DEV-Studio apresenta, no topo da tela, um conjunto de opções de menu e uma série de botões que facilitam a sua manipulação.
  
- Na tela central, é apresentado o código das funções em ADVPL. Na parte inferior são exibidas algumas pastas que facilitam a execução de comandos, exibição de conteúdos de variáveis e mensagens, bem como dados sobre o projeto.

## DESENVOLVIMENTO DE PEQUENAS CUSTOMIZAÇÕES

### 11. ACESSO E MANIPULAÇÃO DE BASES DE DADOS EM ADVPL

Como a linguagem ADVPL é utilizada no desenvolvimento de aplicação para o sistema ERP Protheus, ela deve possuir recursos que permitam o acesso e a manipulação de informações, independentemente da base de dados para o qual o Sistema foi configurado.

Desta forma a linguagem possui dois grupos de funções distintos para atuar com os bancos de dados:

- Funções de manipulação de dados genéricos;
- Funções de manipulação de dados específicas para ambientes TOPCONNECT / DBACCESS.

#### Funções de manipulação de dados genéricos

As funções de manipulação de dados, ditos como genéricos, permitem que uma aplicação ADVPL seja escrita da mesma forma, independente se a base de dados configurada para o sistema ERP for do tipo ISAM ou padrão SQL.

Muitas destas funções foram inicialmente herdadas da linguagem CLIPPER, e mediante novas implementações da área de Tecnologia da Microsiga foram melhoradas e adequadas às necessidades do ERP. Por esta razão é possível encontrar em documentações da linguagem CLIPPER informações sobre funções de manipulação de dados utilizados na ferramenta ERP.

Dentre as melhorias implementadas pela área de Tecnologia da Microsiga, podemos mencionar o desenvolvimento de novas funções como, por exemplo, a função MsSeek() - versão da Microsiga para a função DbSeek(), e a integração entre a sintaxe ADVPL convencional e a ferramenta de acesso a bancos de dados no padrão SQL – TOPCONNECT (DbAcess).

A integração, entre a aplicação ERP e a ferramenta TOPCONNECT, permite que as funções de acesso e manipulação de dados escritos em ADVPL sejam interpretados e convertidos para uma sintaxe compatível com o padrão SQL ANSI e desta forma aplicados aos SGDBs (Sistemas Gerenciadores de Bancos de Dados) com sua sintaxe nativa.

## **Funções de manipulação de dados para Ambientes TOPCONNECT / DBACCESS**

---

Para implementar um acesso mais otimizado e disponibilizar no Ambiente ERP funcionalidades que utilizem de forma mais adequada os recursos dos SGDBs homologados para o Sistema, foram implementadas funções de acesso e manipulação de dados específicas para Ambientes TOPCONNECT/DBACCESS.

Estas funções permitem que o desenvolvedor ADVPL execute comandos em sintaxe SQL, diretamente de um código fonte da aplicação, disponibilizando recursos como execução de queries de consulta, chamadas de procedures e comunicação com outros bancos de dados através de ODBCs.



**Fique  
atento**

***As funções específicas para Ambientes TOPCONNECT serão abordadas no material de ADVPL Avançado.***

## 11.1. Diferenças e compatibilizações entre bases de dados

---

Como a aplicação ERP pode ser configurada para utilizar diferentes tipos de bases de dados é importante mencionar as principais diferenças entre estes recursos, o que pode determinar a forma como o desenvolvedor optará por escrever sua aplicação.

### Acesso a dados e índices

---

No acesso a informações, em bases de dados do padrão ISAM, são sempre lidos os registros inteiros, enquanto no SQL pode-se ler apenas os campos necessários naquele processamento.

O acesso direto é feito através de índices que são tabelas paralelas às tabelas de dados e que contêm a chave e o endereço do registro, de forma análoga ao índice de um livro. Para cada chave, é criado um índice próprio.

Nas bases de dados padrão ISAM os índices são armazenados em um único arquivo do tipo CDX, já nos bancos de dados padrão SQL cada índice é criado com uma numeração sequencial, tendo como base o nome da tabela a qual ele está relacionado.

A cada inclusão ou alteração de um registro todos os índices são atualizados, tornando necessário planejar adequadamente quais e quantos índices serão definidos para uma tabela, pois uma quantidade excessiva pode comprometer o desempenho dessas operações.

Deve ser considerada a possibilidade da utilização de índices temporários para os processos específicos, os quais serão criados em tempo de execução da rotina. Esse fator deve levar em consideração o "esforço" do Ambiente a cada execução da rotina e a periodicidade com a qual é executada.

### Estrutura dos registros (informações)

---

Nas bases de dados padrão ISAM cada registro possui um identificador nativo ou ID sequencial e ascendente que funciona como o endereço base daquela informação.

Este ID, mais conhecido como RECNO ou RECNUMBER é gerado no momento de inclusão do registro na tabela e somente será alterado se a estrutura dos dados dessa tabela sofrer alguma manutenção. Dentre as manutenções que uma tabela de dados ISAM pode sofrer, é possível citar a utilização do comando PACK, o qual apagará fisicamente os registros deletados da tabela, forçando uma renumeração dos identificadores de todos os registros. Essa

situação também torna necessária a recriação de todos os índices vinculados àquela tabela.

Isso ocorre nas bases de dados ISAM devido ao conceito de exclusão lógica de registros que as mesmas possuem. Já os bancos de dados padrão SQL nativamente utilizam apenas o conceito de exclusão física de registros, o que para outras aplicações seria transparente, mas não é o caso do ERP Protheus.

Para manter a compatibilidade das aplicações desenvolvidas para as bases de dados padrão ISAM, a área de Tecnologia e Banco de Dados da Microsiga implementou, nos bancos de dados padrão SQL, o conceito de exclusão lógica de registros existente nas bases de dados ISAM, por meio da criação de campos de controle específicos: R\_E\_C\_N\_O\_, D\_E\_L\_E\_T\_ e R\_E\_C\_D\_E\_L\_.

Esses campos permitem que a aplicação ERP gerencie as informações do banco de dados, da mesma forma que as informações em bases de dados ISAM.

Com isso o campo R\_E\_C\_N\_O\_ será um identificador único do registro dentro da tabela, funcionando como o ID ou RECNUMBER de uma tabela ISAM, mas utilizando um recurso adicional disponível nos bancos de dados relacionais conhecido com **Chave Primária**.

Para a aplicação ERP Protheus o campo de controle R\_E\_C\_N\_O\_ é definido em todas as tabelas como sendo sua chave primária, o que transfere o controle de sua numeração sequencial ao banco de dados.

O campo D\_E\_L\_E\_T\_ é tratado internamente pela aplicação ERP como um "flag" ou marca de exclusão. Dessa forma, os registros que estiverem com este campo marcado serão considerados como excluídos logicamente. A execução do comando PACK, em uma tabela de um banco de dados padrão SQL, visa excluir fisicamente os registros com o campo D\_E\_L\_E\_T\_ marcado, mas não causará o efeito de renumeração de RECNO (no caso R\_E\_C\_N\_O\_) que ocorre na tabela de bases de dados ISAM.

## 11.2. Funções de acesso e manipulação de dados

As funções de acesso e manipulação de dados, descritas neste tópico, são as classificadas anteriormente como funções genéricas da linguagem ADVPL, permitindo que as mesmas sejam utilizadas independentemente da base de dados para a qual a aplicação ERP está configurada.

As funções de acesso e manipulação de dados definem basicamente:

- Tabela que está sendo tratada;
- Campos que deverão ser lidos ou atualizados;
- Método de acesso direto às informações (registros e campos).

Dentre as funções ADVPL disponíveis para acesso e manipulação de informações, este material detalhará as seguintes opções:

- SELECT()**
- DBSELECTAREA()**
- DBSETORDER()**
- DBSEEK() E MSSEEK()**
- DBSKIP()**
- DBGOTO()**
- DBGOTOP()**
- DBGOBOTTON()**
- DBSETFILTER()**
- RECLOCK()**
- SOFTLOCK()**
- MSUNLOCK()**
- DBDELETE()**
- DBUSEAREA()**
- DBCLOSEAREA()**

**DBRLOCK()**

---

<b>Sintaxe</b>	<b>DBRLOCK(xIdentificador)</b>
<b>Descrição</b>	<p>Função de base de dados, que efetua o lock (travamento) do registro identificado pelo parâmetro xIdentificador. Esse parâmetro pode ser o Recno() para tabelas em formato ISAM, ou a chave primária para bancos de dados relacionais.</p> <p>Se o parâmetro xIdentificador não for especificado, todos os locks da área de trabalho serão liberados e o registro posicionado será travado e adicionado em uma lista de registros bloqueados.</p>

### **DBCLOSEAREA()**

---

<b>Sintaxe</b>	<b>DbCloseArea()</b>
<b>Descrição</b>	<p>Permite que um "alias" presente na conexão seja fechado, o que viabiliza novamente seu uso em outra operação. Este comando tem efeito apenas no alias ativo na conexão, sendo necessária sua utilização em conjunto com o comando DbSelectArea().</p>

### **DBCOMMIT()**

---

<b>Sintaxe</b>	<b>DBCOMMIT()</b>
<b>Descrição</b>	Efetua todas as atualizações pendentes na área de trabalho ativa.

### **DBCOMMITALL()**

---

<b>Sintaxe</b>	<b>DBCOMMITALL()</b>
<b>Descrição</b>	Efetua todas as atualizações pendentes em toda a área de trabalho, em uso pela thread (conexão) ativa.

### **DBDELETE()**

---

<b>Sintaxe</b>	<b>DbDelete()</b>
----------------	-------------------

<b>Descrição</b>	Efetua a exclusão lógica do registro posicionado na área de trabalho ativa, sendo necessária sua utilização em conjunto com as funções RecLock() e MsUnLock().
------------------	--

## **DBGOTO()**

---

<b>Sintaxe</b>	<b>DbGoto(nRecno)</b>
<b>Descrição</b>	Movimenta o cursor da área de trabalho ativa para o record number (recno) especificado, realizando um posicionamento direto, sem a necessidade de uma busca (seek) prévia.

## DBGOTOP()

---

<b>Sintaxe</b>	<b>DbGoTop()</b>
<b>Descrição</b>	Move o cursor da área de trabalho ativa para o primeiro registro lógico.

## DBGOBOTTON()

---

<b>Sintaxe</b>	<b>DbGoBotton()</b>
<b>Descrição</b>	Move o cursor da área de trabalho ativa para o último registro lógico.

## DBRLOCKLIST()

---

<b>Sintaxe</b>	<b>DBRLOCKLIST()</b>
<b>Descrição</b>	Retorna um array contendo o record number (recno) de todos os registros, travados da área de trabalho ativa.

## DBSEEK() E MSSEEK()

---

<b>Sintaxe</b>	<b>DbSeek(cChave, ISoftSeek, ILast)</b>
<b>Descrição</b>	<p><b>DbSeek:</b> Permite posicionar o cursor da área de trabalho ativo no registro com as informações especificadas na chave de busca, fornecendo um retorno lógico indicando se o posicionamento foi efetuado com sucesso, ou seja, se a informação especificada na chave de busca foi localizada na área de trabalho.</p> <p><b>MsSeek():</b> Função desenvolvida pela área de Tecnologia da Microsiga, a qual possui as mesmas funcionalidades básicas da função DbSeek(), com a vantagem de não necessitar acessar novamente a base de dados para localizar uma informação já utilizada pela thread (conexão) ativa.</p>

## DBSKIP()

---

<b>Sintaxe</b>	<b>DbSkip(nRegistros)</b>
<b>Descrição</b>	Movê o cursor do registro posicionado para o próximo (ou anterior, dependendo do parâmetro), em função da ordem ativa para a área de trabalho.

## **DBSELECTAREA()**

---

<b>Sintaxe</b>	<b>DbSelectArea(nArea   cArea)</b>
<b>Descrição</b>	Define a área de trabalho especificada como sendo a área ativa. Todas as operações subsequentes que fizerem referência a uma área de trabalho para utilização, a menos que a área desejada seja informada explicitamente.

## DBSETFILTER()

---

<b>Sintaxe</b>	<b>DbSetFilter(bCondicao, cCondicao)</b>
<b>Descrição</b>	Define um filtro para a área de trabalho ativa, o qual pode ser descrito na forma de um bloco de código ou através de uma expressão simples.

## DBSETORDER()

---

<b>Sintaxe</b>	<b>DbSetOrder(nOrdem)</b>
<b>Descrição</b>	Define qual índice será utilizado pela área de trabalho ativa, ou seja, pela área previamente selecionada através do comando DbSelectArea(). As ordens disponíveis no Ambiente Protheus são aquelas definidas no SINDEIX /SIX, ou as ordens disponibilizadas por meio de índices temporários.

## DBORDERNICKNAME()

---

<b>Sintaxe</b>	<b>DbOrderNickName(NickName)</b>
<b>Descrição</b>	Define qual índice criado pelo usuário será utilizado. O usuário pode incluir os seus próprios índices e no momento da inclusão deve criar o NICKNAME para o mesmo.

## DBUNLOCK()

---

<b>Sintaxe</b>	<b>DBUNLOCK()</b>
<b>Descrição</b>	Mesma funcionalidade da função UNLOCK(), só que recomendada para ambientes de rede nos quais os arquivos são compartilhados.  Libera o travamento do registro posicionado na área de trabalho ativa e confirma as atualizações efetuadas naquele registro.

## DBUNLOCKALL()

---

<b>Sintaxe</b>	<b>DBUNLOCKALL()</b>
<b>Descrição</b>	Libera o travamento de todos os registros de todas as áreas de trabalho disponíveis na thread (conexão) ativa.

### **DBUSEAREA()**

---

<b>Sintaxe</b>	<b>DbUseArea(INovo, cDriver, cArquivo, cAlias, ICompartilhado,; ISoLeitura)</b>
<b>Descrição</b>	Define um arquivo de base de dados como uma área de trabalho disponível na aplicação.

### **MSUNLOCK()**

---

<b>Sintaxe</b>	<b>MsUnLock()</b>
<b>Descrição</b>	Libera o travamento (lock) do registro posicionado, confirmando as atualizações efetuadas neste registro.

### **RECLOCK()**

---

<b>Sintaxe</b>	<b>RecLock(cAlias,IInclui)</b>
<b>Descrição</b>	Efetua o travamento do registro posicionado na área de trabalho ativa, permitindo a inclusão ou alteração das informações do mesmo.

### **RLOCK()**

<b>Sintaxe</b>	<b>RLOCK() → ISucesso</b>
<b>Descrição</b>	Efetua o travamento do registro posicionado na área de trabalho ativa.

### **SELECT()**

---

<b>Sintaxe</b>	<b>Select(cArea)</b>
<b>Descrição</b>	Determina o número de referência de um determinado alias em um ambiente de trabalho. Caso o alias especificado não esteja em uso no Ambiente, será retornado o valor 0 (zero).

## **SOFTLOCK()**

---

<b>Sintaxe</b>	<b>SoftLock(cAlias)</b>
<b>Descrição</b>	<p>Permite a reserva do registro posicionado na área de trabalho ativa de forma que outras operações, com exceção da atual, não possam atualizar este registro. Difere da função RecLock() pois não gera uma obrigação de atualização, e pode ser sucedido por ele.</p> <p>Na aplicação ERP Protheus, o SoftLock() é utilizado nos browses, antes da confirmação da operação de alteração e exclusão, pois neste momento a mesma ainda não foi efetivada, mas outras conexões não podem acessar aquele registro pois o mesmo está em manutenção, o que implementa a integridade da informação.</p>

## **UNLOCK()**

---

<b>Sintaxe</b>	<b>UNLOCK()</b>
<b>Descrição</b>	Libera o travamento do registro posicionado na área de trabalho ativa e confirma as atualizações efetuadas naquele registro.

### 11.3. Diferenciação entre variáveis e os nomes de campos

Muitas vezes uma variável pode ter o mesmo nome que um campo de um arquivo ou de uma tabela aberta no momento. Neste caso, o ADVPL privilegiará o campo, de forma que uma referência a um nome, que identifique tanto uma variável como um campo, resultará no conteúdo do campo.

Para especificar qual deve ser o elemento referenciado, deve-se utilizar o operador de identificação de apelido (->) e um dos dois identificadores de referência, MEMVAR ou FIELD.

```
cRes := MEMVAR->NOME
```

Essa linha de comando identifica que o valor atribuído à variável cRes deve ser o valor da variável de memória chamada NOME.

```
cRes := FIELD->NOME
```

Nesse caso, o valor atribuído à variável cRes será o valor do campo NOME, existente no arquivo, ou tabela aberto na área atual.

O identificador FIELD pode ser substituído pelo apelido de um arquivo ou tabela abertos, para evitar a necessidade de selecionar a área antes de acessar o conteúdo de determinado campo.

```
cRes := CLIENTES->NOME
```

As tabelas de dados, utilizadas pela aplicação ERP, recebem automaticamente do Sistema o apelido ou ALIAS, especificado para as mesmas no arquivo de sistema SX2. Assim, se o campo NOME pertence a uma tabela da aplicação PROTHEUS, o mesmo poderá ser referenciado com a indicação do ALIAS pré-definido dessa tabela.

```
cRes := SA1->NOME // SA1 – Cadastro de Clientes
```



**Dica**

***Para maiores detalhes sobre abertura de arquivos com a atribuição de apelidos, consulte a documentação sobre o acesso ao banco de dados, ou à documentação da função dbUseArea().***

***Os alias das tabelas da aplicação ERP são padronizados em três letras, que correspondem às iniciais da tabela. As configurações de cada ALIAS, utilizadas pelo Sistema podem ser visualizadas através do módulo Configurador -> Bases de Dados -> Dicionários -> Bases de Dados.***

## 11.4. Controle de numeração sequencial

Alguns campos de numeração do Protheus são fornecidos pelo Sistema em ordem ascendente. É o caso, por exemplo, do número do pedido de venda e outros que servem como identificador das informações das tabelas. É preciso ter um controle do fornecimento desses números, em especial quando vários usuários estão trabalhando simultaneamente.



Fique atento

***Os campos que recebem o tratamento de numeração sequencial pela aplicação ERP não devem ser considerados como chave primária das tabelas à qual estão vinculados.***

***No caso específico da aplicação ERP Protheus, a chave primária em Ambientes TOPCONNECT será o campo R\_E\_C\_N\_O\_, e para bases de dados padrão ISAM, o conceito de chave primária é implementado pela regra de negócio do sistema, pois este padrão de dados não possui o conceito de unicidade de dados.***

### Semáforos

Para definir o conceito do que é um semáforo de numeração deve-se avaliar a seguinte sequência de eventos no sistema:

- Ao ser fornecido um número, ele permanece reservado até a conclusão da operação que o solicitou;
- Se esta operação for confirmada o número é indisponibilizado, mas se a operação for cancelada, o número voltará a estar disponível mesmo que naquele momento números maiores já tenham sido oferecidos e utilizados.

Com isso, mesmo que tenhamos vários processos solicitando as numerações sequenciais para uma mesma tabela, como por exemplo inclusões simultâneas de pedidos de vendas, teremos para cada pedido um número exclusivo e sem o intervalos e numerações não utilizadas.

## Funções de controle de semáforos e numeração sequencial

---

A linguagem ADVPL permite a utilização das seguintes funções para o controle das numerações sequenciais utilizadas nas tabelas da aplicação ERP:

- ☐ **GETSXENUM()**
- ☐ **CONFIRMSXE()**
- ☐ **ROLLBACKSXE()**

### GETSXENUM()

---

<b>Sintaxe</b>	<b>GETSXENUM(cAlias, cCampo, cAliasSXE, nOrdem)</b>
<b>Descrição</b>	Obtém o número sequência do alias especificado no parâmetro através da referência aos arquivos de sistema SXE/SXF, ou ao servidor de numeração, quando esta configuração está habilitada no Ambiente Protheus.

### CONFIRMSXE()

---

<b>Sintaxe</b>	<b>CONFIRMSXE(IVerifica)</b>
<b>Descrição</b>	Confirma o número alocado através do último comando GETSXENUM().

### ROLLBACKSXE()

---

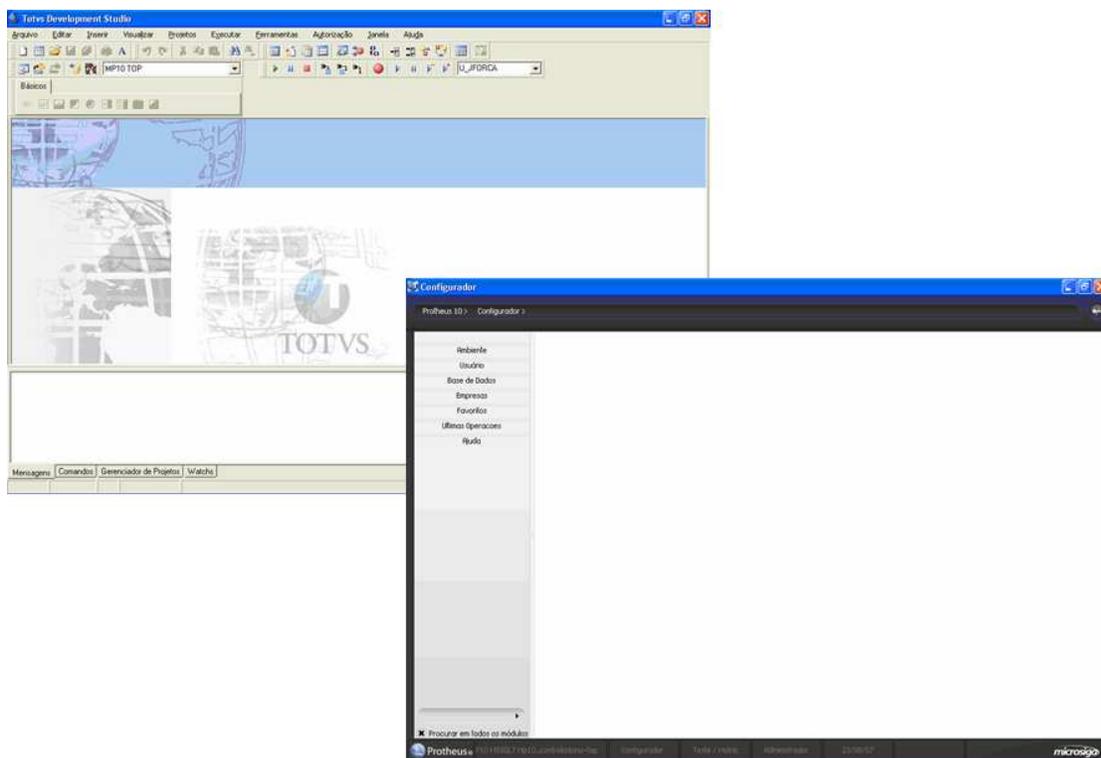
<b>Sintaxe</b>	<b>ROLLBACKSXE()</b>
<b>Descrição</b>	Descarta o número fornecido pelo último comando GETSXENUM(), retornando a numeração disponível para as outras conexões.

## 12. CUSTOMIZAÇÕES PARA A APLICAÇÃO ERP

Nesse tópico serão abordadas as formas pelas quais a aplicação ERP Protheus pode ser customizada, com a utilização da linguagem ADVPL.

Pelos recursos de configuração da aplicação ERP, disponíveis no módulo Configurador, é possível implementar as seguintes customizações:

- validações de campos e perguntas do Sistema e de usuários;
- inclusão de gatilhos em campos de Sistemas e de usuários;
- inclusão de regras em parâmetros de Sistemas e de usuários;
- Desenvolvimento de pontos de entrada para interagir com funções padrões do Sistema.



### 12.1. Customização de campos – Dicionário de Dados

## 12.1.1. Validações de campos e perguntas

As funções de validação têm como característica fundamental um retorno do tipo lógico, ou seja, um conteúdo .T. – Verdadeiro, ou .F. – Falso.

Com base nessa premissa, a utilização de validações, no Dicionário de Dados (SX3), ou nas Perguntas de Processos e Relatórios (SX1), deverá estar focada sempre na utilização de funções, ou expressões que resultem em um retorno lógico.

Com a utilização do módulo Configurador é possível alterar as propriedades de um campo, ou de uma pergunta de forma a incluir as regras de validação para as seguintes situações:

- SX3 – validação de usuário (X3\_VLDUSER);**
- SX1 – validação da pergunta (X1\_VALID).**

Dentre as funções que a linguagem ADVPL, em conjunto com os recursos desenvolvidos pela aplicação ERP, para validação de campos e perguntas serão detalhadas:

- VAZIO()**
- NAOVAZIO()**
- EXISTCPO()**
- EXISTCHAV()**
- PERTENCE()**
- POSITIVO()**
- NEGATIVO()**
- TEXTO()**

### EXISTCHAV()

---

<b>Sintaxe</b>	<b>ExistChav(cAlias, cConteudo, nIndice)</b>
<b>Descrição</b>	Retorna .T. ,ou .F. se o conteúdo especificado existe no alias especificado. Se existir, será exibido um help de Sistema com um aviso informando a ocorrência.  Esta função é utilizada normalmente para verificar se um determinado código de cadastro já existe na tabela na qual a informação será inserida, como por exemplo, o CNPJ no cadastro de clientes, ou fornecedores.

## EXISTCPO()

---

<b>Sintaxe</b>	<b>ExistCpo(cAlias, cConteúdo, nIndice)</b>
<b>Descrição</b>	Retorna .T., ou .F. se o conteúdo especificado não existir no alias especificado. Se não existir, será exibido um help de Sistema com um aviso informando a ocorrência.  Essa Função é utilizada normalmente para verificar se a informação digitada em um campo, que é dependente de outra tabela, realmente existe na mesma. Como por exemplo, o código de um cliente em um pedido de venda.

## NAOVAZIO()

---

<b>Sintaxe</b>	<b>NaoVazio()</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo do campo posicionado no momento não está vazio.

## NEGATIVO()

---

<b>Sintaxe</b>	<b>Negativo()</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo digitado para o campo é negativo.

## PERTENCE()

---

<b>Sintaxe</b>	<b>Pertence(cString)</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo digitado para o campo está contido na string, definida como o parâmetro da função. Normalmente utilizada em campos com a opção de combo, pois caso contrário seria utilizada a função ExistCpo().

## POSITIVO()

---

<b>Sintaxe</b>	<b>Positivo()</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo digitado para o campo é positivo.

## TEXTO()

---

<b>Sintaxe</b>	<b>Texto()</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo digitado para o campo contém apenas números, ou alfanuméricos.

## VAZIO()

---

<b>Sintaxe</b>	<b>Vazio()</b>
<b>Descrição</b>	Retorna .T. ou .F. se o conteúdo do campo posicionado no momento está vazio.

## 12.1.2. Pictures de formação disponíveis

Com base na documentação disponível no TDN, a linguagem ADVPL e a aplicação ERP Protheus admitem as seguintes pictures:

### Dicionário de Dados (SX3) e GET

---

Funções	
Conteúdo	Funcionalidade
<b>A</b>	Permite apenas caracteres alfabéticos
<b>C</b>	Exibe CR depois de números positivos
<b>E</b>	Exibe numérico com o ponto e vírgula invertidos (formato Europeu)
<b>R</b>	Inserir caracteres diferentes dos caracteres de template na exibição, mas não os inserir na variável do GET
<b>S&lt;n&gt;</b>	Permite o rolamento horizontal do texto dentro do GET, <n>. É um número inteiro que identifica o tamanho da região
<b>X</b>	Exibe DB depois de números negativos
<b>Z</b>	Exibe zeros como brancos
<b>(</b>	Exibe números negativos entre parênteses com os espaços em branco iniciais
<b>)</b>	Exibe números negativos entre parênteses sem os espaços em branco iniciais
<b>!</b>	Converte caracteres alfabéticos para maiúsculos

Templates	
Conteúdo	Funcionalidade
<b>X</b>	Permite qualquer caractere
<b>9</b>	Permite apenas dígitos para qualquer tipo de dado, incluindo o sinal para numéricos

#	Permite dígitos, sinais e espaços em branco para qualquer tipo de dado
!	Converte caracteres alfabéticos para maiúsculo
*	Exibe um asterisco no lugar dos espaços em branco iniciais em números
.	Exibe o ponto decimal
,	Exibe a posição do milhar

**Exemplo 01 – Picture campo numérico**

CT2\_VALOR – Numérico – 17,2

Picture: @E 99,999,999,999,999.99

**Exemplo 02 – Picture campo texto, com digitação apenas em caixa alta**

A1\_NOME – Caracter - 40

Picture: @!

<b>Funções</b>	
<b>Conteúdo</b>	<b>Funcionalidade</b>
<b>C</b>	Exibe CR depois de números positivos
<b>E</b>	Exibe numérico com o ponto e a vírgula invertidos (formato Europeu)
<b>R</b>	Insere caracteres diferentes dos caracteres de template
<b>X</b>	Exibe DB depois de números negativos
<b>Z</b>	Exibe zeros como brancos
<b>(</b>	Envolve números negativos entre parênteses
<b>!</b>	Converte todos os caracteres alfabéticos para maiúsculo

<b>Templates</b>	
<b>Conteúdo</b>	<b>Funcionalidade</b>
<b>X</b>	Exibe dígitos para qualquer tipo de dado
<b>9</b>	Exibe dígitos para qualquer tipo de dado
<b>#</b>	Exibe dígitos para qualquer tipo de dado
<b>!</b>	Converte caracteres alfabéticos para maiúsculo
<b>*</b>	Exibe asterisco no lugar de espaços em branco e inicia em números
<b>.</b>	Exibe a posição do ponto decimal
<b>,</b>	Exibe a posição do milhar

**Exemplo 01 – Picture campo numérico**

CT2\_VALOR – Numérico – 17,2

Picture: @E 99,999,999,999,999.99



*Anotações*

---

---

---

---

## 12.2. Customização de gatilhos – Configurador

A aplicação ERP utiliza o recurso de gatilhos em campo com a finalidade de auxiliar o usuário no preenchimento de informações, durante a sua digitação.. As funções que podem ser utilizadas no gatilho estão diretamente relacionadas à definição da expressão de retorno, que será executada na avaliação do gatilho do campo.

As regras que devem ser observadas na montagem de um gatilho e configuração de seu retorno são:

- ☑ Na definição da chave de busca do gatilho deve ser avaliada qual filial deverá ser utilizada como parte da chave: a filial da tabela de origem do gatilho, ou a filial da tabela que será consultada. O que normalmente determina a filial que será utilizada, como parte da chave, é justamente a informação que será consultada, aonde:
  - As Consultas de informações, entre tabelas com estrutura de cabeçalho e itens, devem utilizar a filial da tabela de origem, pois ambas as tabelas devem possuir o mesmo tratamento de filial (compartilhado ou exclusivo).

### Exemplos:

Pedido de vendas -> SC5 x SC6  
Nota fiscal de entrada -> SF1 x SD1  
Ficha de imobilizado -> SN1 x SN3  
Orçamento contábil -> CV1 x CV2

- Consulta de informações de tabelas de cadastros devem utilizar a filial da tabela a ser consultada, pois o compartilhamento dos cadastros normalmente é independente em relação às movimentações e outros cadastros do Sistema.

### Exemplos:

Cadastro de clientes -> SA1 (compartilhado)  
Cadastro de fornecedores -> SA2 (compartilhado)  
Cadastro de vendedores -> SA3 (exclusivo)  
Cadastro de transportadoras -> SA4 (exclusivo)

- Consulta as informações de tabelas de movimentos que devem utilizar a filial da tabela a ser consultada, pois apesar das movimentações de um módulo seguirem um determinado padrão. A consulta pode ser realizada entre tabelas de módulos distintos, o que poderia gerar um retorno incorreto baseado nas diferentes parametrizações desses Ambientes.

### Exemplos:

Contas a pagar -> SE2 (compartilhado)  
Movimentos contábeis -> CT2 (exclusivo)  
Pedidos de compras -> SC7 (compartilhado)  
Itens da nota fiscal de entrada -> SD1 (exclusivo)

- ☑ Na definição da regra de retorno deve ser considerado o tipo do campo que será atualizado, pois é esse campo que determina qual o tipo do retorno será considerado válido para o gatilho.

### 12.3. Customização de parâmetros – Configurador

Os parâmetros de sistema utilizados pela aplicação ERP e definidos através do módulo configurador possuem as seguintes características fundamentais:

- ☑ Tipo de parâmetro: de forma similar a uma variável, um parâmetro terá um tipo de conteúdo pré-definido em seu cadastro.

Essa informação é utilizada pelas funções da aplicação ERP, na interpretação do conteúdo do parâmetro e no retorno dessa informação à rotina que o consultou.

- ☑ Interpretação do conteúdo do parâmetro: Diversos parâmetros do Sistema têm seu conteúdo macro executado durante a execução de uma rotina do ERP. Esses parâmetros macro executáveis tem como única característica em comum o seu tipo: caractere, mas não existe nenhum identificador explícito que permite a fácil visualização de quais parâmetros possuem um retorno simples e de quais parâmetros terão o seu conteúdo macro executado para determinar o retorno "real".

A única forma eficaz de avaliar como um parâmetro é tratado (simples retorno, ou macro execução) é através do código fonte da rotina, que deverá ser avaliado como é tratado o retorno de uma destas funções:

- **GETMV()**
- **SUPERGETMV()**
- **GETNEWPAR()**

Um retorno macro executado é determinado através do uso do operador "&" ou de uma das funções de execução de blocos de código, em conjunto com uma das funções citadas anteriormente.

### 12.3.1. Funções para a manipulação de parâmetros

A aplicação ERP disponibiliza as seguintes funções para consulta e atualização de parâmetros:

- GETMV()
- SUPERGETMV()
- GETNEWPAR()
- PUTMV()

#### GETMV()

---

<b>Sintaxe</b>	<b>GETMV(cParametro)</b>
<b>Descrição</b>	Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Se o parâmetro não existir, será exibido um help do sistema informando a ocorrência.

## GETNEWPAR()

---

<b>Sintaxe</b>	<b>GETNEWPAR(cParametro, cPadrao, cFilial)</b>
<b>Descrição</b>	<p>Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Se o parâmetro não existir, será exibido um help do Sistema informando a ocorrência.</p> <p>Difere do SuperGetMV() pois considera que o parâmetro pode não existir na versão atual do Sistema, e por consequência não será exibida a mensagem de help.</p>

## PUTMV()

---

<b>Sintaxe</b>	<b>PUTMV(cParametro, cConteudo)</b>
<b>Descrição</b>	<p>Atualiza o conteúdo do parâmetro especificado no arquivo SX6, de acordo com as parametrizações informadas.</p>

## SUPERGETMV()

---

<b>Sintaxe</b>	<b>SUPERGETMV(cParametro , IHelp , cPadrao , cFilial)</b>
<b>Descrição</b>	<p>Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Se o parâmetro não existir, será exibido um help do Sistema informando a ocorrência.</p> <p>Difere do GetMv() pois os parâmetros consultados são adicionados em uma área de memória que permite que em uma nova consulta não seja necessário acessar e pesquisar o parâmetro na base de dados.</p>

### 12.3.2. Cuidados na utilização de um parâmetro

Um parâmetro de sistema tem a finalidade de propiciar um retorno válido a um conteúdo previamente definido na configuração do módulo para uma rotina, processo, ou quaisquer outros tipos de funcionalidades disponíveis na aplicação.

Apesar de haver parâmetros que permitam a configuração de expressões, e por consequência a utilização de funções para definir o retorno que será obtido com a consulta desse parâmetro, **é expressamente proibido o uso de funções em parâmetros para manipular as informações da base de dados do Sistema.**

Se houver necessidade de ser implementado um tratamento adicional a um processo padrão do Sistema, o mesmo deverá utilizar o recurso de ponto de entrada.

A razão dessa restrição é simples:

- As rotinas da aplicação ERP não protegem a consulta de conteúdos de parâmetros, quanto às gravações realizadas dentro, ou fora de uma transação.
- Dessa forma, qualquer alteração na base realizada por uma rotina, configurada em um parâmetro, pode ocasionar a perda da integridade das informações do Sistema.

## 12.4. Pontos de Entrada – Conceitos, Premissas e Regras

### Conceitos

---

Um ponto de entrada é uma User Function desenvolvida com a finalidade de interagir com uma rotina padrão da aplicação ERP.

A User Function deverá ter um nome pré-estabelecido no desenvolvimento da rotina padrão do ERP, e de acordo com esta pré-disposição e o momento no qual o ponto de entrada é executado durante um processamento, ele poderá:

- Complementar uma validação realizada pela aplicação;
- Complementar as atualizações realizadas pelo processamento em tabelas padrões do ERP;
- Implementar a atualização de tabelas específicas durante o processamento de uma rotina padrão do ERP;
- Executar uma ação sem processos de atualizações, mas que necessite utilizar as informações atuais do Ambiente, durante o processamento da rotina padrão para determinar as características do processo;
- Substituir um processamento padrão do Sistema por uma regra específica do cliente, no qual o mesmo será implementado.

### Premissas e Regras

---

- Um ponto de entrada não deve ser utilizado para outras finalidades senão para as quais foi pré-definido, sob pena de causar a perda da integridade das informações da base de dados, ou provocar eventos de erro durante a execução da rotina padrão.
- Um ponto de entrada deve ser transparente para o processo padrão, de forma que todas as tabelas, acessadas pelo ponto de entrada e que sejam utilizadas pela rotina padrão, deverão ter a sua situação imediatamente anterior à execução do ponto

restaurado ao término do mesmo, e para isto recomenda-se o uso das funções GETAREA() e RESTAREA().

- ☑ Como um ponto de entrada não é executado da forma tradicional, ou seja, ele não é chamado como uma função, ele não recebe parâmetros. A aplicação ERP disponibiliza uma variável de Sistema denominada PARAMIXB, a qual recebe os parâmetros da função chamadora e os disponibiliza para serem utilizados pela rotina customizada.
  
- ☑ A variável PARAMIXB não possui um padrão de definição nos códigos fontes da aplicação ERP, desta forma seu tipo pode variar de um conteúdo simples (caractere, numérico, lógico e etc.) a um tipo complexo como um array ou um objeto. Assim, é necessário sempre avaliar a documentação sobre o ponto, bem como proteger a função customizada de tipos de PARAMIXB não tratados por ela.

## 13. INTERFACES VISUAIS

A linguagem ADVPL possui duas formas distintas para definição de interfaces visuais no Ambiente ERP: sintaxe convencional, nos padrões da linguagem CLIPPER e a sintaxe orientada a objetos.

Além das diferentes sintaxes disponíveis para definição das interfaces visuais, o ERP Protheus possui funcionalidades pré-definidas, as quais já contêm todos os tratamentos necessários para atender as necessidades básicas de acesso e manutenção das informações do Sistema.

Neste tópico serão abordadas as sintaxes convencionais para definição das interfaces visuais da linguagem ADVPL e as interfaces de manutenção, disponíveis no Ambiente ERP Protheus.

### 13.1. Sintaxe e componentes das interfaces visuais

A sintaxe convencional para definição de componentes visuais da linguagem ADVPL depende diretamente, em include especificado no cabeçalho, do fonte. Os dois includes, disponíveis para o Ambiente ADVPL Protheus, são:

- RWMAKE.CH: Permite a utilização da sintaxe CLIPPER na definição dos componentes visuais.
- PROTHEUS.CH: Permite a utilização da sintaxe ADVPL convencional, a qual é um aprimoramento da sintaxe CLIPPER, com a inclusão de novos atributos para os componentes visuais disponibilizados no ERP Protheus.

Para ilustrar a diferença na utilização destes dois includes, seguem as diferentes definições para o componentes Dialog e MsDialog:

#### Exemplo 01 – Include Rwmake.ch

```
#include "rwmake.ch"

@ 0,0 TO 400,600 DIALOG oDlg TITLE "Janela em sintaxe Clipper"
ACTIVATE DIALOG oDlg CENTERED
```

#### Exemplo 02 – Include Protheus.ch

```
#include "protheus.ch"

DEFINE MSDIALOG oDlg TITLE "Janela em sintaxe ADVPL "FROM 000,000 TO 400,600
PIXEL
ACTIVATE MSDIALOG oDlg CENTERED
```



***Ambas sintaxes produzirão o mesmo efeito quando compiladas e executadas no ambiente Protheus, mas deve ser utilizada sempre a sintaxe ADVPL, por meio do uso do include PROTHEUS.CH.***

Os componentes da interface visual que serão tratados neste tópico, utilizando a sintaxe ADVPL são:

- ☐ **MSDIALOG()**
- ☐ **MSGET()**
- ☐ **SAY()**
- ☐ **BUTTON()**
- ☐ **SBUTTON()**

#### **BUTTON()**

---

<b>Sintaxe</b>	<b>@ nLinha,nColuna BUTTON cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef ACTION AÇÃO</b>
<b>Descrição</b>	Define o componente visual Button, o qual permite a inclusão de botões de operação na tela da interface, os quais serão visualizados somente com um texto simples para sua identificação.

#### **MSDIALOG()**

---

<b>Sintaxe</b>	<b>DEFINE MSDIALOG oObjetoDLG TITLE cTitulo FROM nLinIni,nColIni TO nLiFim,nColFim OF oObjetoRef UNIDADE</b>
<b>Descrição</b>	Define o componente MSDIALOG(), que é utilizado como base para os demais componentes da interface visual, pois um componente MSDIALOG() é uma janela da aplicação.

## MSGET()

---

<b>Sintaxe</b>	<b>@ nLinha, nColuna MSGET VARIÁVEL SIZE nLargura,nAltura UNIDADE OF oObjetoRef F3 cF3 VALID VALID WHEN WHEN PICTURE cPicture</b>
<b>Descrição</b>	Define o componente visual MSGET, que é utilizado para captura de informações digitáveis na tela da interface.

## SAY()

---

<b>Sintaxe</b>	<b>@ nLinha, nColuna SAY cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef</b>
<b>Descrição</b>	Define o componente visual SAY, que é utilizado para a exibição de textos em uma tela de interface.

## SBUTTON()

---

<b>Sintaxe</b>	<b>DEFINE SBUTTON FROM nLinha, nColuna TYPE N ACTION AÇÃO STATUS OF oObjetoRet</b>
<b>Descrição</b>	Define o componente visual SButton, que permite a inclusão de botões de operação na tela da interface, que serão visualizados dependendo da interface do sistema ERP, utilizada somente com um texto simples para sua identificação, ou com uma imagem (BitMap) pré-definido.

## Interface visual completa

---

Abaixo, segue um código completo de interface, utilizando todos os elementos da interface visual descritos anteriormente:

```
DEFINE MSDIALOG oDlg TITLE cTitulo FROM 000,000 TO 080,300 PIXEL  
  
@ 001,001 TO 040, 150 OF oDlg PIXEL  
  
@ 010,010 SAY cTexto SIZE 55, 07 OF oDlg PIXEL
```

```
@ 010,050 MSGET cCGC      SIZE 55, 11 OF oDlg PIXEL PICTURE "@R
99.999.999/9999-99";
VALID !Vazio()

DEFINE SBUTTON FROM 010, 120 TYPE 1 ACTION (nOpca := 1,oDlg:End());
ENABLE OF oDlg

DEFINE SBUTTON FROM 020, 120 TYPE 2 ACTION (nOpca := 2,oDlg:End());
ENABLE OF oDlg

ACTIVATE MSDIALOG oDlg CENTERED
```



*Dica*

***O código demonstrado anteriormente é utilizado nos exercícios de fixação deste material e deverá produzir a seguinte interface:***

Busca por fornecedor

Informe o CNPJ:

Ok Cancelar

## 13.2. Interfaces padrões para atualizações de dados

Os programas de atualização de cadastros e digitação de movimentos seguem um padrão que se apóia no Dicionário de Dados.

Basicamente são duas as interfaces que permitem a visualização das informações e a manipulação dos dados do Sistema.

- AxCadastro**
- Mbrowse**

Ambos modelos utilizam como premissa que a estrutura da tabela a ser utilizada esteja definida no dicionário de dados do sistema (SX3).

### 13.2.1. AxCadastro()

O AxCadastro() é uma funcionalidade de cadastro simples, com poucas opções de customização, composta por:

- Browse padrão para visualização das informações da base de dados, de acordo com as configurações do SX3 – Dicionário de Dados (campo browse).
- Funções de pesquisa, visualização, inclusão, alteração e exclusão de padrões para visualização de registros simples, sem a opção de cabeçalho e itens.

- Sintaxe: AxCadastro(cAlias, cTitulo, cVldExc, cVldAlt)**

- Parâmetros**

<b>cAlias</b>	Alias padrão do sistema para utilização, o qual deve estar definido no dicionário de dados – SX3
<b>cTitulo</b>	Título da Janela
<b>cVldExc</b>	Validação para Exclusão
<b>cVldAlt</b>	Validação para Alteração

### Exemplo:

```
#include "protheus.ch"
```

```
User Function XCadSA2()
```

```
Local cAlias := "SA2"
```

```
Local cTitulo := "Cadastro de Fornecedores"
```

```
Local cVldExc := ".T."
```

```
Local cVldAlt := ".T."
```

```
dbSelectArea(cAlias)
```

```
dbSetOrder(1)
```

```
AxCadastro(cAlias,cTitulo,cVldExc,cVldAlt)
```

```
Return
```

Clientes - 99/01

Clientes - Incluir

Copiar Recortar Colar Calc Spell Ajuda Gas Credito OK Cancelar

Cadastrais Adm/Fin. Fecais Vendas Outros

Codigo  Leja

Fisica/Jurid Fisica

Nome

N Fantasia  Tipo

Endereco

Municipio  Estado

Bairro  CEP

DDD  DDD

Telefone  Telex

FAX  Pais

Descr. Pais  CNPJ/CPF

Conlato  Ins. Estad.

RG/Ced.Estr.  Ins. Municip.

DI.Rber/Nasc  E-Mail

Home-Page  Cod. CBO

Cod CNRE  Bloqueado Não

Insc.Rural

tro - EE/FF

Desqualificar

Desqualizar

Incluir

Alterar

Excluir

### 13.2.2. MBrowse()

A Mbrowse() é uma funcionalidade de cadastro que permite a utilização de recursos mais aprimorados na visualização e manipulação das informações do Sistema, possuindo os seguintes componentes:

- Browse padrão para visualização das informações da base de dados, de acordo com as configurações do SX3 – Dicionário de Dados (campo browse).
- Parametrização para as funções específicas para as ações de visualização, inclusão, alteração e exclusão de informações, o que viabiliza a manutenção de informações com estrutura de cabeçalhos e itens.
- Recursos adicionais como identificadores de status de registros, legendas e filtros para as informações.

**Sintaxe simplificada: MBrowse(nLin1, nCol1, nLin2, nCol2, cAlias)**

**Parâmetros**

<b>nLin1, nCol1, nLin2, nCol2</b>	Coordenadas dos cantos aonde o browse será exibido. Para seguir o padrão da AXCADASTRO() use 6,1,22,75 .
<b>cAlias</b>	Alias padrão do sistema para utilização. Deve estar definido no dicionário de dados – SX3.

☑ **Variáveis private adicionais**

<b>aRotina</b>	<p>Array contendo as funções que serão executadas pela Mbrowse.</p> <p>Este array pode ser parametrizado com as funções básicas da AxCadastro conforme abaixo:</p> <pre>AADD(aRotina,{"Pesquisar" ,"AxPesqui",0,1}) AADD(aRotina,{"Visualizar" ,"AxVisual",0,2}) AADD(aRotina,{"Incluir" ,"AxInclui" ,0,3}) AADD(aRotina,{"Alterar" ,"AxAltera",0,4}) AADD(aRotina,{"Excluir" ,"AxDeleta",0,5})</pre>
<b>cCadastro</b>	<p>Título do browse que será exibido.</p>

**Exemplo:**

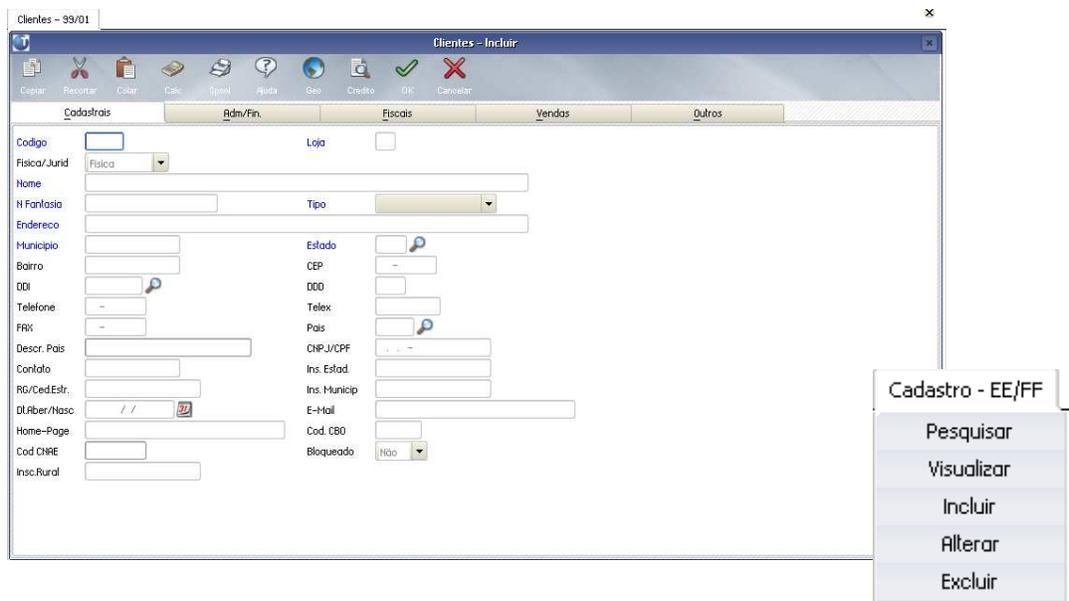
```
#include "protheus.ch"

User Function MBrwSA2()

Local cAlias := "SA2"
Private cCadastro := "Cadastro de Fornecedores"
Private aRotina := {}

AADD(aRotina,{"Pesquisar" ,"AxPesqui",0,1})
AADD(aRotina,{"Visualizar" ,"AxVisual",0,2})
AADD(aRotina,{"Incluir" ,"AxInclui",0,3})
AADD(aRotina,{"Alterar" ,"AxAltera",0,4})
AADD(aRotina,{"Excluir" ,"AxDeleta",0,5})
```

```
dbSelectArea(cAlias)
dbSetOrder(1)
mBrowse(6,1,22,75,cAlias)
Return
```



**Fique atento**

***Utilizando a parametrização exemplificada, o efeito obtido com o uso da Mbrowse() será o mesmo obtido com o uso da AxCadastro().***

***A posição das funções no array aRotina define o conteúdo de uma variável de controle que será repassada para as funções chamadas a partir da Mbrowse, convencionada como nOpc. Dessa forma, para manter o padrão da aplicação ERP a ordem a ser seguida na definição do aRotina é;***

***1 – Pesquisar;***

***2 – Visualizar;***

***3 – Incluir;***

**4 – Alterar;**

**5 – Excluir;**

**6 – Livre.**



**Dica**

**Ao definir as funções no array *aRotina*, se o nome da função não for especificado com “()”, a *Mbrowse* passará como parâmetros as seguintes variáveis de controle:**

***cAlias*: Alias ativo definido para a *Mbrowse*;**

***nRecno*: Record number (*recno*) do registro posicionado no alias ativo;**

***nOpc*: Posição da opção utilizada na *Mbrowse* de acordo com a ordem da função no array *a Rotina*.**

### Exemplo: Função BInclui() substituindo a função AxInclui()

```
#include "protheus.ch"

User Function MBrwSA2()

Local cAlias := "SA2"

Private cCadastro := "Cadastro de Fornecedores"
Private aRotina      := {}

AADD(aRotina,{"Pesquisar" ,"AxPesqui"  ,0,1})
AADD(aRotina,{"Visualizar" ,"AxVisual"  ,0,2})
AADD(aRotina,{"Incluir"    ,"U_BInclui" ,0,3})
AADD(aRotina,{"Alterar"    ,"AxAltera"  ,0,4})
AADD(aRotina,{"Excluir"    ,"AxDeleta"  ,0,5})

dbSelectArea(cAlias)
dbSetOrder(1)
mBrowse(6,1,22,75,cAlias)
Return

USER FUNCTION BInclui(cAlias, nReg, nOpc)

Local cTudoOk := "(Alert('OK'),.T.)"
AxInclui(cAlias,nReg,nOpc,,,,cTudoOk)

RETURN
```



**Anotações**

---

---

---

---

---

### 13.2.3. AxFunctions()

Conforme foi mencionado nos tópicos sobre as interfaces padrões AxCadastro() e Mbrowse(), existem funções padrões da aplicação ERP que permitem a visualização, inclusão, alteração e exclusão de dados em formato simples.

Essas funções são padrões na definição da interface AxCadastro() e podem ser utilizadas também da construção no array aRotina, utilizado pela Mbrowse(), as quais estão listadas a seguir:

- ☐ **AXPESQUI()**
- ☐ **AXVISUAL()**
- ☐ **AXINCLUI()**
- ☐ **AXALTERA()**
- ☐ **AXDELETA()**

#### AXALTERA()

---

<b>Sintaxe</b>	<b>AxAlterar(cAlias, nReg, nOpc, aAcho, cFunc, aCpos, cTudoOk, IF3,; cTransact, aButtons, aParam, aAuto, IVirtual, IMaximized)</b>
<b>Descrição</b>	Função de alteração padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

#### AXDELETA()

---

<b>Sintaxe</b>	<b>AXDELETA(cAlias, nReg, nOpc, cTransact, aCpos, aButtons, aParam,; aAuto, IMaximized)</b>
<b>Descrição</b>	Função de exclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

#### AXINCLUI()

---

<b>Sintaxe</b>	<b>AxInclui(cAlias, nReg, nOpc, aAcho, cFunc, aCpos, cTudoOk, IF3,; cTransact, aButtons, aParam, aAuto, IVirtual, IMaximized)</b>
<b>Descrição</b>	Função de inclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

## **AXPESQUI()**

---

<b>Sintaxe</b>	<b>AXPESQUI()</b>
<b>Descrição</b>	Função de pesquisa padrão em registros exibidos pelos browses do Sistema, que posiciona o browse no registro pesquisado. Exibe uma tela que permite a seleção do índice a ser utilizado na pesquisa e a digitação das informações que compõe a chave de busca.

## AXVISUAL()

---

<b>Sintaxe</b>	<b>AXVISUAL(cAlias, nReg, nOpc, aAcho, nColMens, cMensagem, cFunc,; aButtons, lMaximized )</b>
<b>Descrição</b>	Função de visualização padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().



### Exercícios

#### Exercício 15

Desenvolver uma validação para um campo específico do tipo caractere, cujo conteúdo esteja relacionado a outra tabela e que exiba uma mensagem de aviso caso o código informado não exista nessa tabela relacionada.



### Exercícios

#### Exercício 16

Desenvolver uma validação para um campo caractere existente na base, para que seja avaliado se aquele código já existe cadastrado, e se positivo, exiba uma mensagem de aviso alertando sobre a ocorrência.



### Exercícios

#### Exercício 17

Desenvolver um gatilho que retorne uma descrição complementar para um campo vinculado ao campo código utilizado nos exercícios anteriores.



### **Exercícios**

#### Exercício 18

Customizar o parâmetro que define o prefixo do título de contas a pagar, gerado pela integração COMPRAS -> FINANCEIRO através da inclusão de uma nota fiscal de entrada, de forma que o prefixo do título seja gerado com o código da filial corrente.



### **Exercícios**

#### Exercício 19

Proteger a rotina desenvolvida no exercício anterior, de forma a garantir que na utilização da filial como prefixo do título, não ocorrerá duplicidade de dados em contas a pagar do financeiro.



### **Exercícios**

#### Exercício 20

Vamos Implementar uma validação adicional no cadastro de clientes, com a utilização do ponto de entrada adequado, de forma que o campo CNPJ (A1\_CGC) seja obrigatório para todos os tipos de clientes, exceto os definidos como Exterior.

## APÊNDICES

## BOAS PRÁTICAS DE PROGRAMAÇÃO

### 14. UTILIZAÇÃO DE IDENTIFICAÇÃO

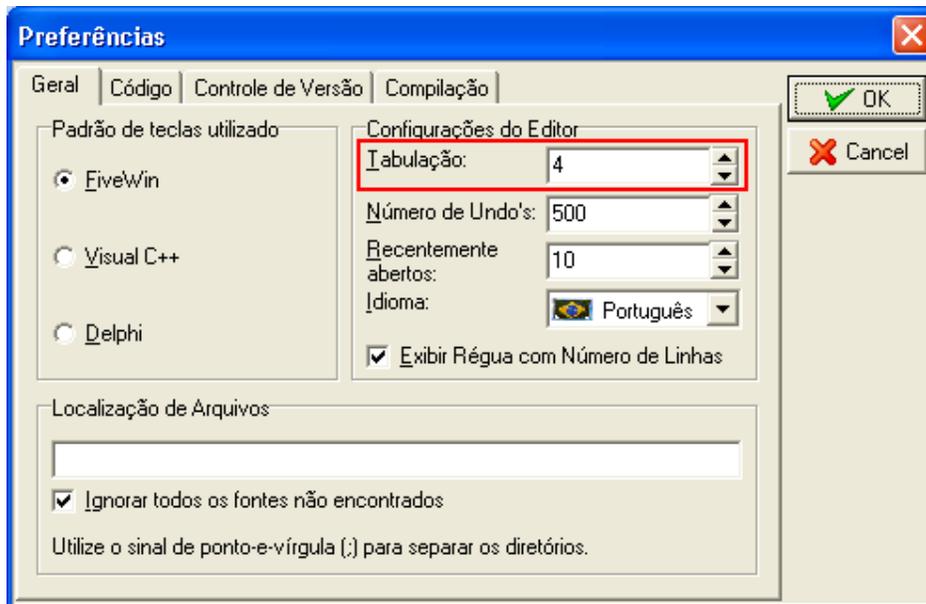
É obrigatória a utilização da indentação, pois torna o código muito mais legível. Veja os exemplos abaixo:

```
While !SB1->(Eof())
If mv_par01 = SB1->B1_COD
dbSkip()
Loop
Endif
Do Case
Case SB1->B1_LOCAL == "01" .Or. SB1->B1_LOCAL == "02"
TrataLocal(SB1->B1_COD,SB1->B1_LOCAL)
Case SB1->B1_LOCAL == "03"
TrataDefeito(SB1->B1_COD)
Otherwise
TrataCompra(SB1->B1_COD,SB1->B1_LOCAL)
EndCase
dbSkip()
EndDo
```

A utilização da indentação, seguindo as estruturas de controle de fluxo (while, if, caso etc), torna a compreensão do código muito mais fácil:

```
While !SB1->(Eof())
    If mv_par01 = SB1->B1_COD
        dbSkip()
    Loop
Endif
Do Case
Case SB1->B1_LOCAL == "01" .Or. SB1->B1_LOCAL == "02"
    TrataLocal(SB1->B1_COD,SB1->B1_LOCAL)
Case SB1->B1_LOCAL == "03"
    TrataDefeito(SB1->B1_COD)
Otherwise
    TrataCompra(SB1->B1_COD,SB1->B1_LOCAL)
EndCase
dbSkip()
EndDo
```

Para indentar o código utilize a tecla <TAB> e a ferramenta DEV-Studio, a qual pode ser configurada através da opção "Preferências":



## 15. CAPITULAÇÃO DE PALAVRAS-CHAVE

Uma convenção amplamente utilizada é a de capitular as palavras chaves, funções, variáveis e campos utilizando uma combinação de caracteres em maiúsculo e minúsculo, visando facilitar a leitura do código fonte. O código a seguir:

```
local ncnt while ( ncnt++ < 10 ) ntotal += ncnt * 2 enddo
```

Ficaria melhor com as palavras chaves e variáveis capituladas:

```
Local nCnt While ( nCnt++ < 10 ) nTotal += nCnt * 2 EndDo
```



Fique atento

***Para as funções de manipulação de dados que comecem por "db", a capitulação só será efetuada após o "db":***

*dbSeek()*

*dbSelectArea()*



Anotações

---

---

---

---

---

## 15.1. Palavras em maiúsculo

A regra é utilizar caracteres em maiúsculo para:

**Constantes:**

```
#define NUMLINES 60 #define NUMPAGES 1000
```

**Variáveis de memória:**

```
M-> CT2_CRCONV M->CT2_MCONVER := CriaVar("CT2_CONVER")
```

**Campos:**

```
SC6->C6_NUMPED
```

**Querys:**

```
SELECT * FROM...
```

## 16. UTILIZAÇÃO DA NOTAÇÃO HÚNGARA

A notação húngara consiste em adicionar os prefixos aos nomes de variáveis, de modo a facilmente identificar o seu tipo. Isso facilita a criação de códigos-fonte extensos, pois usando a Notação Húngara, você não precisa ficar o tempo todo voltando à definição de uma variável para se lembrar de qual é o tipo de dados que deve ser colocado nela. Variáveis devem ter um prefixo de Notação Húngara em minúsculas, seguido de um nome que identifique a função da variável, sendo que a inicial de cada palavra deve ser maiúscula.

É obrigatória a utilização dessa notação para nomear variáveis.

Notação	Tipo de dado	Exemplo
a	<b>Array</b>	<b>aValores</b>
b	<b>Bloco de código</b>	<b>bSeek</b>
c	<b>Caracter</b>	<b>cNome</b>
d	<b>Data</b>	<b>dDataBase</b>
l	<b>Lógico</b>	<b>lContinua</b>
n	<b>Numérico</b>	<b>nValor</b>

o	<b>Objeto</b>	<b>oMainWindow</b>
x	<b>Indefinido</b>	<b>xConteudo</b>



*Anotações*

---

---

---

---

---

## 17. PALAVRAS RESERVADAS

AADD	DTOS	INKEY	REPLICATE	VAL
ABS	ELSE	INT	RLOCK	VALTYPE
ASC	ELSEIF	LASTREC	ROUND	WHILE
AT	EMPTY	LEN	ROW	WORD
BOF	ENDCASE	LOCK	RTRIM	YEAR
BREAK	ENDDO	LOG	SECONDS	CDOW
ENDIF	LOWER	SELECT	CHR	EOF
LTRIM	SETPOS	CMONTH	EXP	MAX
SPACE	COL	FCOUNT	MIN	SQRT
CTOD	FIELDNAME	MONTH	STR	DATE
FILE	PCOL	SUBSTR	DAY	FLOCK
PCOUNT	TIME	DELETED	FOUND	PROCEDURE
TRANSFORM	DEVPOS	FUNCTION	PROW	TRIM
DOW	IF	RECCOUNT	TYPE	DTOC
IIF	RECNO	UPPER	TRY	AS
CATCH	THROW			



Fique  
atento

***As palavras reservadas não podem ser utilizadas para variáveis, procedimentos ou funções;***

***As Funções reservadas são pertencentes ao compilador e não podem ser redefinidas por uma aplicação;***

***Todos os identificadores, que comecem com dois ou mais caracteres "\_", são utilizados como identificadores internos e são reservados.***

***Os Identificadores de escopo PRIVATE ou PUBLIC utilizados em aplicações específicas, desenvolvidas por ou para clientes, devem ter sua identificação iniciada por um caractere "\_".***

## GUIA DE REFERÊNCIA RÁPIDA: Funções e Comandos ADVPL

Neste guia de referência rápida serão descritas as funções básicas da linguagem ADVPL, incluindo as funções herdadas da linguagem Clipper, necessárias ao desenvolvimento no Ambiente ERP.

### Conversão entre tipos de dados

#### CTOD()

Realiza a conversão de uma informação do tipo caracter no formato "DD/MM/AAAA", para uma variável do tipo data.

- Sintaxe: CTOD(cData)**
- Parâmetros**

<b>cData</b>	Caractere no formato "DD/MM/AAAA".
--------------	------------------------------------

#### Exemplo:

```
cData := "31/12/2006"
dData := CTOD(cData)

IF dDataBase >= dData
    MSGALERT("Data do sistema fora da competência")
ELSE
    MSGINFO("Data do sistema dentro da competência")
ENDIF
```

#### CVALTOCHAR()

Realiza a conversão de uma informação do tipo numérico em uma string, sem a adição de espaços a informação.

- ☑ **Sintaxe: CVALTOCHAR(nValor)**
- ☑ **Parâmetros**

<b>nValor</b>	Valor numérico que será convertido para caractere.
---------------	--

**Exemplo:**

```
FOR nPercorridos := 1 to 10
    MSGINFO("Passos percorridos: "+CvalToChar(nPercorridos))
NEXT nPercorridos
```

## **DTOC()**

---

Realiza a conversão de uma informação do tipo data para caractere, sendo o resultado no formato "DD/MM/AAAA".

- Sintaxe: DTOC(dData)**
- Parâmetros**

<b>dData</b>	Variável com conteúdo data.
--------------	-----------------------------

### **Exemplo:**

```
MSGINFO("Database do sistema: "+DTOC(dData)
```

## **DTOS()**

---

Realiza a conversão de uma informação do tipo data em um caractere, sendo o resultado no formato "AAAAMMDD".

- Sintaxe: DTOS(dData)**
- Parâmetros**

<b>dData</b>	Variável com conteúdo data.
--------------	-----------------------------

### **Exemplo:**

```
cQuery := "SELECT A1_COD, A1_LOJA, A1_NREDUZ FROM SA1010 WHERE "  
cQuery += "A1_DULTCOM >=" + DTOS(dDataIni) + ""
```

## **STOD()**

---

Realiza a conversão de uma informação do tipo caractere, com conteúdo no formato "AAAAMMDD" em data.

- Sintaxe: STOD(sData)**
- Parâmetros**

<b>sData</b>	String no formato "AAAAMMDD".
--------------	-------------------------------

### **Exemplo:**

```
sData := LERSTR(01,08) // Função que realiza a leitura de uma string de um txt previamente
                        // aberto
dData := STOD(sData)
```

## **STR()**

---

Realiza a conversão de uma informação do tipo numérico em uma string, adicionando espaços à direita.

- Sintaxe: STR(nValor)**
- Parâmetros**

<b>nValor</b>	Valor numérico que será convertido para caractere.
---------------	--

**Exemplo:**

```
FOR nPercorridos := 1 to 10
    MSGINFO("Passos percorridos: "+CvalToChar(nPercorridos))
NEXT nPercorridos
```

**STRZERO()**

---

Realiza a conversão de uma informação do tipo numérico em uma string, adicionando zeros à esquerda do número convertido, de forma que a string gerada tenha o tamanho especificado no parâmetro.

- Sintaxe: STRZERO(nValor, nTamanho)**
- Parâmetros**

<b>nValor</b>	Valor numérico que será convertido para caractere.
<b>nTamanho</b>	Tamanho total desejado para a string retornada.

**Exemplo:**

```
FOR nPercorridos := 1 to 10
    MSGINFO("Passos percorridos: "+CvalToChar(nPercorridos))
NEXT nPercorridos
```

## VAL()

---

Realiza a conversão de uma informação do tipo carácter em numérica.

- Sintaxe: VAL(cValor)**
- Parâmetros**

<b>cValor</b>	String que será convertida para numérico.
---------------	---

### Exemplo:

```
Static Function Modulo11(cData)
LOCAL L, D, P := 0
L := Len(cdata)
D := 0
P := 1
While L > 0
    P := P + 1
    D := D + (Val(SubStr(cData, L, 1)) * P)
    If P = 9
        P := 1
    End
    L := L - 1
End
D := 11 - (mod(D,11))
If (D == 0 .Or. D == 1 .Or. D == 10 .Or. D == 11)
    D := 1
End
Return(D)
```

## Verificação de tipos de variáveis

### TYPE()

Determina o tipo do conteúdo de uma variável, a qual não foi definida na função em execução.

- Sintaxe: TYPE("cVariavel")**
- Parâmetros**

<b>"cVariavel"</b>	Nome da variável que se deseja avaliar, entre aspas ("").
--------------------	---

#### Exemplo:

```
IF TYPE("dDataBase") == "D"  
    MSGINFO("Database do sistema: "+DTC("dDataBase"))  
ELSE  
    MSGINFO("Variável indefinida no momento")
```

## VALTYPE()

---

Determina o tipo do conteúdo de uma variável, a qual não foi definida na função em execução.

- Sintaxe: VALTYPE(cVariavel)**
- Parâmetros**

<b>cVariavel</b>	Nome da variável que se deseja avaliar.
------------------	---

### Exemplo:

```
STATIC FUNCTION GETTEXTO(nTamanho, cTitulo, cSay)

LOCAL cTexto := ""
LOCAL nColF      := 0
LOCAL nLargGet   := 0
PRIVATE oDlg

Default cTitulo:= "Tela para informar texto"
Default cSay   := "Informe o texto:"
Default nTamanho := 1

IF ValType(nTamanho) != "N" // Se o parâmetro foi passado incorretamente
    nTamanho := 1
ENDIF

cTexto      := Space(nTamanho)
nLargGet    := Round(nTamanho * 2.5,0)
nColF      := Round(195 + (nLargGet * 1.75) ,0)

DEFINE MSDIALOG oDlg TITLE cTitulo FROM 000,000 TO 120,nColF PIXEL
@ 005,005 TO 060, Round(nColF/2,0) OF oDlg PIXEL
```

```
@ 010,010 SAY cSay SIZE 55, 7 OF oDlg PIXEL
@ 010,065 MSGET cTexto SIZE nLargGet, 11 OF oDlg PIXEL ;
Picture "@" VALID !Empty(cTexto)
DEFINE SBUTTON FROM 030, 010 TYPE 1 ;
ACTION (nOpca := 1,oDlg:End()) ENABLE OF oDlg
DEFINE SBUTTON FROM 030, 040 TYPE 2 ;
ACTION (nOpca := 0,oDlg:End()) ENABLE OF oDlg
ACTIVATE MSDIALOG oDlg CENTERED

cTexto := IIF(nOpca=1,cTexto,"")

RETURN cTexto
```

## Manipulação de arrays

### Array()

A função Array() é utilizada na definição de variáveis de tipo array, como uma opção a sintaxe utilizando chaves ("{}").

- ☑ **Sintaxe: Array(nLinhas, nColunas)**
- ☑ **Parâmetros**

<b>nLinhas</b>	Determina o número de linhas com as quais o array será criado.
<b>nColunas</b>	Determina o número de colunas com as quais o array será criado.

#### Exemplo:

```
aDados := Array(3,3) // Cria um array de três linhas, cada qual com 3 colunas.
```



Fique atento

***O array definido pelo comando Array() apesar de já possuir a estrutura solicitada, não possui conteúdo em nenhum de seus elementos, ou seja:***

***aDados[1] -> array de três posições***

***aDados[1][1] -> posição válida, mas de conteúdo nulo.***

### AADD()

A função AADD() permite a inserção de um item em um array já existente, sendo que este item podem ser um elemento simples, um objeto ou outro array.

- ☑ **Sintaxe: AADD(aArray, xItem)**
- ☑ **Parâmetros**

<b>aArray</b>	Array pré-existente no qual será adicionado o item definido em xItem.
<b>xItem</b>	Item que será adicionado ao array.

**Exemplo:**

```
aDados := {} // Define que a variável aDados é um array, sem especificar suas dimensões.
```

```
aItem := {} // Define que a variável aItem é um array, sem especificar suas dimensões.
```

```
AADD(aItem, cVariavel1) // Adiciona um elemento no array aItem de acordo com o cVariavel1
```

```
AADD(aItem, cVariavel2) // Adiciona um elemento no array aItem de acordo com o cVariavel2
```

```
AADD(aItem, cVariavel3) // Adiciona um elemento no array aItem de acordo com o cVariavel3
```

```
// Neste ponto o array aItem possui 03 elementos os quais podem ser acessados com:
```

```
// aItem[1] -> corresponde ao conteúdo de cVariavel1
```

```
// aItem[2] -> corresponde ao conteúdo de cVariavel2
```

```
// aItem[3] -> corresponde ao conteúdo de cVariavel3
```

```
AADD(aDados,aItem) // Adiciona no array aDados o conteúdo do array aItem
```

**Exemplo (continuação):**

```
// Neste ponto, o array aDados possui apenas um elemento, que também é um array
```

```
// contendo 03 elementos:
```

```
// aDados [1][1] -> corresponde ao conteúdo de cVariavel1
```

```
// aDados [1][2] -> corresponde ao conteúdo de cVariavel2
```

```
// aDados [1][3] -> corresponde ao conteúdo de cVariavel3
```

```
AADD(aDados, aItem)
```

```
AADD(aDados, aItem)
```

```
// Neste ponto, o array aDados possui 03 elementos, aonde cada qual é um array com outros
```

```
// 03 elementos, sendo:
```

```
// aDados [1][1] -> corresponde ao conteúdo de cVariavel1
```

```
// aDados [1][2] -> corresponde ao conteúdo de cVariavel2
```

```
// aDados [1][3] -> corresponde ao conteúdo de cVariavel3
```

```
// aDados [2][1] -> corresponde ao conteúdo de cVariavel1
// aDados [2][2] -> corresponde ao conteúdo de cVariavel2
// aDados [2][3] -> corresponde ao conteúdo de cVariavel3

// aDados [3][1] -> corresponde ao conteúdo de cVariavel1
// aDados [3][2] -> corresponde ao conteúdo de cVariavel2
// aDados [3][3] -> corresponde ao conteúdo de cVariavel3

// Desta forma, o array aDados montando com uma estrutura de 03 linhas e 03 colunas, com
// o conteúdo definido por variáveis externas, mas com a mesma forma obtida com o uso do
// comando: aDados := ARRAY(3,3).
```

## **ACLONE()**

---

A função ACLONE() realiza a cópia dos elementos de um array para outro array integralmente.

- Sintaxe: AADD(aArray)**
- Parâmetros**

<b>aArray</b>	Array pré-existente que terá seu conteúdo copiado para o array especificado.
---------------	--

### **Exemplo:**

```
// Utilizando o array aDados, utilizado no exemplo da função AADD()
aItens := ACLONE(aDados).

// Neste ponto, o array aItens possui exatamente a mesma estrutura e informações do array
// aDados.
```



**Por ser uma estrutura de memória, um array não pode ser simplesmente copiado para outro array, com a utilização de uma atribuição simples (":=").**

**Para mais informações sobre a necessidade de utilizar o comando `ACLONE()`, verifique o tópico 6.1.3 – Cópia de Arrays.**

## **ADEL()**

---

A função `ADEL()` permite a exclusão de um elemento do array. Ao efetuar a exclusão de um elemento, todos os demais são reorganizados de forma que a última posição do array passará a ser nula.

- Sintaxe: `ADEL(aArray, nPosição)`**
- Parâmetros**

<b>aArray</b>	Array do qual deseja-se remover uma determinada posição.
<b>nPosição</b>	Posição do array que será removida.

### **Exemplo:**

```
// Utilizando o array aItens do exemplo da função ACLONE() temos:
```

```
ADEL(aItens,1) // Será removido o primeiro elemento do array aItens.
```

```
// Neste ponto, o array aItens continua com 03 elementos, onde:
```

```
// aItens[1] -> antigo aItens[2], o qual foi reordenado como efeito da exclusão do item 1.
```

```
// aItens[2] -> antigo aItens[3], o qual foi reordenado como efeito da exclusão do item 1.
```

```
// aItens[3] -> conteúdo nulo, por se tratar do item excluído.
```

## **ASIZE()**

---

A função ASIZE permite a redefinição da estrutura de um array pré-existente, adicionando ou removendo itens do mesmo.

- ☑ **Sintaxe: ASIZE(aArray, nTamanho)**
- ☑ **Parâmetros**

<b>aArray</b>	Array pré-existente que terá sua estrutura redimensionada.
<b>nTamanho</b>	Tamanho com o qual se deseja redefinir o array. Se o tamanho for menor do que o atual, serão removidos os elementos do final do array, já se o tamanho for maior do que o atual serão inseridos itens nulos ao final do array.

**Exemplo:**

```
// Utilizando o array aItens, o qual teve um elemento excluído pelo uso da função ADEL()

ASIZE(aItens,Len(aItens-1)).

// Nesse ponto o array aItens possui 02 elementos, ambos com conteúdos válidos.
```



Dica

**Utilizar a função ASIZE(), após o uso da função ADEL(), é uma prática recomendada e evita que seja acessada uma posição do array com um conteúdo inválido para a aplicação em uso.**

**ASORT()**

A função ASORT() permite que os itens de um array sejam ordenados a partir de um critério pré-estabelecido.

- Sintaxe: ASORT(aArray, nInicio, nItens, bOrdem)**
- Parâmetros**

<b>aArray</b>	Array pré-existente que terá seu conteúdo ordenado utilizando um critério estabelecido.
<b>nInicio</b>	Posição inicial do array para início da ordenação. Se não for informado, o array será ordenado a partir de seu primeiro elemento.
<b>nItens</b>	Quantos itens, a partir da posição inicial deverão ser ordenados. Se não for informado, serão ordenados todos os elementos do array.
<b>bOrdem</b>	O bloco de código que permite a definição do critério de ordenação do array. Se o bOrdem não for informado, será utilizado o critério ascendente.



Dica

**Um bloco de código é basicamente uma função escrita em linha. Dessa forma, sua estrutura deve "suportar" todos os requisitos de uma função, os quais são por meio da análise e interpretação de parâmetros recebidos, executar um processamento e fornecer um retorno.**

**Com base nesse requisito, é possível definir um bloco de código com a estrutura abaixo:**

**$bBloco := \{ |xPar1, xPar2, \dots, xParZ| A\grave{c}\tilde{a}o1, A\grave{c}\tilde{a}o2, A\grave{c}\tilde{a}oZ \}$ , aonde:**

**|| -> define o intervalo onde estão compreendidos os parâmetros**

**Ação Z-> expressão que será executadas pelo bloco de código**

**Ação1... AçãoZ -> intervalo de expressões que serão executadas pelo bloco de código, no formato de lista de expressões.**

**Retorno -> resultado da última ação executada pelo bloco de código, no caso**

**AçãoZ.**

**Para maiores detalhes sobre a estrutura e utilização de blocos de código, consulte o tópico 6.2 – Listas de Expressões e Blocos de código.**

### Exemplo 01 – Ordenação ascendente

```
aAlunos := { "Mauren", "Soraia", "Andréia" }
```

```
aSort(aAlunos)
```

```
// Nesse ponto, os elementos do array aAlunos serão { "Andréia", "Mauren", "Soraia" }
```

### Exemplo 02 – Ordenação descendente

```
aAlunos := { "Mauren", "Soraia", "Andréia"}
bOrdem := {|x,y| x > y }

// Durante a execução da função aSort(), a variável "x" receberá o conteúdo do item que está
// posicionado. Como o item que está posicionado é a posição aAlunos[x] e aAlunos[x] ->
// string contendo o nome de um aluno, é possível substituir "x" por cNomeAtu.
// A variável "y" receberá o conteúdo do próximo item a ser avaliado, e usando a mesma
// analogia de "x", é possível substituir "y" por cNomeProx. Dessa forma o bloco de código
// bOrdem pode ser re-escrito como:

bOrdem := {|cNomeAtu, cNomeProx| cNomeAtu > cNomeProx}

aSort(aAlunos,,bOrdem)

// Nesse ponto, os elementos do array aAlunos serão {"Soraia", "Mauren", "Andréia"}
```

### ASCAN()

---

A função ASCAN() permite que seja identificada a posição do array que contém uma determinada informação, através da análise de uma expressão descrita em um bloco de código.

- Sintaxe: ASCAN(aArray, bSeek)**
- Parâmetros**

<b>aArray</b>	Array pré-existente no qual deseja identificar a posição que contém a informação pesquisada.
<b>bSeek</b>	Bloco de código que configura os parâmetros da busca ser realizada.

**Exemplo:**

```
aAlunos := {"Márcio", "Denis", "Arnaldo", "Patrícia"}
```

```
bSeek := {x | x == "Denis"}
```

```
nPosAluno := aScan(aAlunos,bSeek) // retorno esperado → 2
```



*Dica*

*Durante a execução da função aScan, a variável "x" receberá o conteúdo do item que está posicionado no momento, no caso aAlunos[x]. Como aAlunos[x] é uma posição do array que contém o nome do aluno, "x" poderia ser renomeada para cNome, e a definição do bloco bSeek poderia ser re-escrita como:*

```
bSeek := {cNome | cNome == "Denis"}
```



Dica

***Na definição dos programas é sempre recomendável utilizar variáveis com nomes significativos, desta forma os blocos de código não são exceção.***

***Sempre opte por analisar como o bloco de código será utilizado e ao invés de "x", "y" e similares, defina os parâmetros com nomes que representem seu conteúdo. Será mais simples o seu entendimento e o entendimento de outros que forem analisar o código escrito.***

## **AINS()**

---

A função AINS() permite a inserção de um elemento no array especificado, em qualquer ponto da estrutura do mesmo, diferindo dessa forma da função AADD(), a qual sempre insere um novo elemento ao final da estrutura já existente.

- Sintaxe: AINS(aArray, nPosicao)**
- Parâmetros**

<b>aArray</b>	Array pré-existente no qual desejasse inserir um novo elemento.
<b>nPosicao</b>	Posição na qual o novo elemento será inserido.

### **Exemplo:**

```
aAlunos := {"Edson", "Robson", "Renato", "Tatiana"}
```

```
AINS(aAlunos,3)
```

```
// Nesse ponto o array aAlunos terá o seguinte conteúdo:
```

```
// {"Edson", "Robson", nulo, "Renato", "Tatiana"}
```



Fique  
atento

***Similar ao efeito da função ADEL(), o elemento inserido no array pela função AINS() terá um conteúdo nulo, sendo necessário tratá-lo após a realização deste comando.***

## Manipulação de blocos de código

### EVAL()

A função EVAL() é utilizada para avaliação direta de um bloco de código, utilizando as informações disponíveis no mesmo de sua execução. Esta função permite a definição e passagem de diversos parâmetros que serão considerados na interpretação do bloco de código.

- ☑ **Sintaxe: EVAL(bBloco, xParam1, xParam2, xParamZ)**
- ☑ **Parâmetros**

<b>bBloco</b>	Bloco de código que será interpretado.
<b>xParamZ</b>	Parâmetros que serão passados ao bloco de código. A partir da passagem do bloco, todos os demais parâmetros da função serão convertidos em parâmetros para a interpretação do código.

#### Exemplo:

```
nInt := 10
bBloco := {|N| x:= 10, y:= x*N, z:= y/(x*N)}

nValor := EVAL(bBloco, nInt)

// O retorno será dado pela avaliação da última ação da lista de expressões, no caso "z".
// Cada uma das variáveis definidas, em uma das ações da lista de expressões, fica disponível
// para a próxima ação.
// Desta forma temos:
// N → recebe nInt como parâmetro (10)
// X → tem atribuído o valor 10 (10)
// Y → resultado da multiplicação de X por N (100)
// Z → resultado da divisão de Y pela multiplicação de X por N ( 100 / 100) → 1
```

## **DBEVAL()**

---

A função DBEval() permite que todos os registros de uma determinada tabela sejam analisados, e para cada registro será executado o bloco de código definido.

- Sintaxe: Array(bBloco, bFor, bWhile)**
- Parâmetros**

<b>bBloco</b>	Bloco de código principal, contendo as expressões que serão avaliadas para cada registro do alias ativo.
<b>bFor</b>	Condição para continuação da análise dos registros, com o efeito de uma estrutura For ... Next.
<b>bWhile</b>	Condição para continuação da análise dos registros, com o efeito de uma estrutura While ... End.

### Exemplo 01

**Considerando o trecho de código abaixo:**

```
dbSelectArea("SX5")  
dbSetOrder(1)  
dbGotop()
```

```
While !Eof() .And. X5_FILIAL == xFilial("SX5") .And.; X5_TABELA <= mv_par02
```

```
    nCnt++
```

```
    dbSkip()
```

```
End
```

**O mesmo pode ser re-escrito com o uso da função DBEVAL():**

```
dbEval( { |x| nCnt++ },{|X5_FILIAL==xFilial("SX5") .And. X5_TABELA<=mv_par02})
```

### Exemplo 02

**Considerando o trecho de código abaixo:**

```
dbSelectArea("SX5")  
dbSetOrder(1)  
dbGotop()
```

```
While !Eof() .And. X5_TABELA == cTabela
```

```
    AADD(aTabela,{X5_CHAVE, Capital(X5_DESCRI)})
```

```
    dbSkip()
```

```
End
```

**Exemplo 02 (continuação):**

**O mesmo pode ser re-escrito com o uso da função DBEVAL():**

```
dbEval({|| aAdd(aTabela,{X5_CHAVE,Capital(X5_DESCR)}),,|| X5_TABELA==cTabela})
```



**Fique  
atento**

***Na utilização da função DBEVAL(), deve ser informado apenas um dos dois parâmetros: bFor ou bWhile.***

## AEVAL()

---

A função AEVAL() permite que todos os elementos de um determinada array sejam analisados e para cada elemento será executado o bloco de código definido.

- ☑ **Sintaxe: AEVAL(aArray, bBloco, nInicio, nFim)**
- ☑ **Parâmetros**

<b>aArray</b>	Array que será avaliado na execução da função
<b>bBloco</b>	Bloco de código principal, contendo as expressões que serão avaliadas para cada elemento do array informado
<b>nInicio</b>	Elemento inicial do array, a partir do qual serão avaliados os blocos de código
<b>nFim</b>	Elemento final do array, até o qual serão avaliados os blocos de código

### Exemplo 01:

#### Considerando o trecho de código abaixo:

```
AADD(aCampos,"A1_FILIAL")
AADD(aCampos,"A1_COD")
SX3->(dbSetOrder(2))
For nX:=1 To Len(aCampos)
  SX3->(dbSeek(aCampos[nX]))
  aAdd(aTitulos,AllTrim(SX3->X3_TITULO))
Next nX
```

#### O mesmo pode ser re-escrito com o uso da função AEVAL():

```
aEval(aCampos,{|x| SX3->(dbSeek(x)),IIF(Found(), AAdd(aTitulos,;
AllTrim(SX3->X3_TITULO))})})
```

## Manipulação de strings

### ALLTRIM()

Retorna uma string sem os espaços à direita e à esquerda, referente ao conteúdo informado como parâmetro.

A função ALLTRIM() implementa as ações das funções RTRIM ("right trim") e LTRIM ("left trim").

- Sintaxe: ALLTRIM(cString)**
- Parâmetros**

<b>cString</b>	String que será avaliada para remoção dos espaços à direita e à esquerda.
----------------	---

#### Exemplo:

```
cNome := ALLTRIM(SA1->A1_NOME)

MSGINFO("Dados do campo A1_NOME:" + CRLF
        "Tamanho:" + CVALTOCHAR(LEN(SA1->A1_NOME)) + CRLF
        "Texto:" + CVALTOCHAR(LEN(cNome)))
```

### ASC()

Converte uma informação caractere em seu valor, de acordo com a tabela ASCII.

- Sintaxe: ASC(cCaractere)**
- Parâmetros**

<b>cCaractere</b>	Caractere que será consultado na tabela ASCII.
-------------------	--

**Exemplo:**

```
USER FUNCTION NoAcento(Arg1)
Local nConta := 0
Local cLetra := ""
Local cRet := ""
Arg1 := Upper(Arg1)
For nConta:= 1 To Len(Arg1)
    cLetra := SubStr(Arg1, nConta, 1)
    Do Case
        Case (Asc(cLetra) > 191 .and. Asc(cLetra) < 198) .or.;
            (Asc(cLetra) > 223 .and. Asc(cLetra) < 230)
            cLetra := "A"
        Case (Asc(cLetra) > 199 .and. Asc(cLetra) < 204) .or.;
            (Asc(cLetra) > 231 .and. Asc(cLetra) < 236)
            cLetra := "E"

        Case (Asc(cLetra) > 204 .and. Asc(cLetra) < 207) .or.;
            (Asc(cLetra) > 235 .and. Asc(cLetra) < 240)
            cLetra := "I"

        Case (Asc(cLetra) > 209 .and. Asc(cLetra) < 215) .or.;
            (Asc(cLetra) == 240) .or. (Asc(cLetra) > 241 .and. Asc(cLetra) < 247)
            cLetra := "O"

        Case (Asc(cLetra) > 216 .and. Asc(cLetra) < 221) .or.;
            (Asc(cLetra) > 248 .and. Asc(cLetra) < 253)
            cLetra := "U"

        Case Asc(cLetra) == 199 .or. Asc(cLetra) == 231
            cLetra := "C"
```

```

EndCase

cRet := cRet+cLetra

Next
Return UPPER(cRet)

```

## **AT()**

---

Retorna à primeira posição de um caractere ou string dentro de outra string especificada.

- Sintaxe: AT(cCaractere, cString )**
- Parâmetros**

<b>cCaractere</b>	Caractere ou string que se deseja verificar.
<b>cString</b>	String na qual será verificada a existência do conteúdo de cCaractere.

### **Exemplo:**

```

STATIC FUNCTION NOMASCARA(cString,cMascara,nTamanho)

LOCAL cNoMascara := ""
LOCAL nX := 0
IF !Empty(cMascara) .AND. AT(cMascara,cString) > 0
    FOR nX := 1 TO Len(cString)
        IF !(SUBSTR(cString,nX,1) $ cMascara)
            cNoMascara += SUBSTR(cString,nX,1)
        ENDIF
    NEXT nX
    cNoMascara := PADR(ALLTRIM(cNoMascara),nTamanho)
ELSE
    cNoMascara := PADR(ALLTRIM(cString),nTamanho)
ENDIF
RETURN cNoMascara

```

## CHR()

---

Converte um valor número, referente a uma informação da tabela ASCII, no caractere que esta informação representa.

- Sintaxe: CHR(nASCII)**
- Parâmetros**

<b>nASCII</b>	Código ASCII do caractere.
---------------	----------------------------

### Exemplo:

```
#DEFINE CRLF CHR(13)+CHR(10) // FINAL DE LINHA
```

## LEN()

---

Retorna o tamanho da string especificada no parâmetro.

- Sintaxe: LEN(cString)**
- Parâmetros**

<b>cString</b>	String que será avaliada.
----------------	---------------------------

### Exemplo:

```
cNome := ALLTRIM(SA1->A1_NOME)
```

```
MSGINFO("Dados do campo A1_NOME:" + CRLF
```

```
    "Tamanho:" + CVALTOCHAR(LEN(SA1->A1_NOME)) + CRLF
```

```
    "Texto:" + CVALTOCHAR(LEN(cNome)))
```

## LOWER()

---

Retorna uma string com todos os caracteres minúsculos, tendo como base a string passada como parâmetro.

- Sintaxe: LOWER(cString)**
- Parâmetros**

<b>cString</b>	String que será convertida para caracteres minúsculos.
----------------	--

### Exemplo:

```
cTexto := "ADVPL"
```

```
MSGINFO("Texto:" + LOWER(cTexto))
```

## RAT()

---

Retorna a última posição de um caractere ou string dentro de outra string especificada.

- Sintaxe: RAT(cCaractere, cString)**
- Parâmetros**

<b>cCaractere</b>	Caractere ou string que se deseja verificar
<b>cString</b>	String na qual será verificada a existência do conteúdo de cCaractere

## STUFF()

---

Permite substituir um conteúdo caractere em uma string já existente, especificando a posição inicial para esta adição e o número de caracteres que serão substituídos.

- Sintaxe: STUFF(cString, nPosInicial, nExcluir, cAdicao)**
- Parâmetros**

--	--

### Exemplo:

```
cLin := Space(100)+cEOL // Cria a string base
```

```
cCpo := PADR(SA1->A1_FILIAL,02) // Informação que será armazenada na string
```

```
cLin := Stuff(cLin,01,02,cCpo) // Substitui o conteúdo de cCpo na string base
```

## SUBSTR()

---

Retorna parte do conteúdo de uma string especificada, de acordo com a posição inicial deste conteúdo na string e a quantidade de caracteres que deverá ser retornada a partir daquele ponto (inclusive).

- ☑ **Sintaxe: SUBSTR(cString, nPosInicial, nCaracteres)**
- ☑ **Parâmetros**

<b>cString</b>	String que se deseja verificar
<b>nPosInicial</b>	Posição inicial da informação que será extraída da string
<b>nCaracteres</b>	Quantidade de caracteres que deverão ser retornados a partir daquele ponto (inclusive)

**Exemplo:**

```
cCampo := "A1_NOME"
```

```
nPosUnder := AT(cCampo)
```

```
cPrefixo := SUBSTR(cCampo,1, nPosUnder) // → "A1_"
```

## UPPER()

---

Retorna uma string com todos os caracteres maiúsculos, tendo como base a string passada como parâmetro.

- Sintaxe: UPPER(cString)**
- Parâmetros**

<b>cString</b>	String que será convertida para caracteres maiúsculos.
----------------	--

### Exemplo:

```
cTexto := "advpl"
```

```
MSGINFO("Texto:" + LOWER(cTexto))
```

## Manipulação de variáveis numéricas

## ABS()

---

Retorna um valor absoluto (independente do sinal), com base no valor especificado no parâmetro.

- Sintaxe: ABS(nValor)**
- Parâmetros**

<b>nValor</b>	Valor que será avaliado
---------------	-------------------------

### Exemplo:

```
nPessoas := 20
```

```
nLugares := 18
```

```
IF nPessoas < nLugares
    MSGINFO("Existem "+CVALTOCHAR(nLugares- nPessoas)+" disponíveis")
ELSE
    MSGSTOP("Existem "+CVALTOCHAR(ABS(nLugares- nPessoas))+" faltando")
ENDIF
```

## INT()

---

Retorna a parte inteira de um valor especificado no parâmetro.

- Sintaxe: INT(nValor)**
- Parâmetros**

<b>nValor</b>	Valor que será avaliado
---------------	-------------------------

### Exemplo:

```
STATIC FUNCTION COMPRAR(nQuantidade)

LOCAL nDinheiro := 0.30
LOCAL nPrcUnit := 0.25

IF nDinheiro >= (nQuantidade*nPrcUnit)
    RETURN nQuantidade
ELSEIF nDinheiro > nPrcUnit
    nQuantidade := INT(nDinheiro / nPrcUnit)
ELSE
    nQuantidade := 0
ENDIF

RETURN nQuantidade
```

## NOROUND()

---

Retorna um valor, truncando a parte decimal do valor especificado no parâmetro, de acordo com a quantidade de casas decimais solicitadas.

- Sintaxe: NOROUND(nValor, nCasas)**
- Parâmetros**

<b>nValor</b>	Valor que será avaliado
<b>nCasas</b>	Número de casas decimais válidas. A partir da casa decimal especificada os valores serão desconsiderados

### Exemplo:

nBase := 2.985

nValor := NOROUND(nBase,2) → 2.98

## ROUND()

---

Retorna um valor, arredondando a parte decimal do valor especificado no parâmetro, de acordo com a quantidade de casas decimais solicitadas, utilizando o critério matemático.

- Sintaxe: ROUND(nValor, nCasas)**
- Parâmetros**

<b>nValor</b>	Valor que será avaliado
<b>nCasas</b>	Número de casas decimais válidas. As demais casas decimais sofrerão o arredondamento matemático, aonde:  Se $nX \leq 4 \rightarrow 0$ , senão +1 para a casa decimal superior

### Exemplo:

nBase := 2.985

nValor := ROUND(nBase,2)  $\rightarrow$  2.99



Anotações \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

## Manipulação de arquivos

### SELECT()

Determina o número de referência de um determinado alias em um ambiente de trabalho. Caso o alias especificado não esteja em uso no ambiente, será retornado o valor 0 (zero).

- Sintaxe: Select(cArea)**
- Parâmetros**

<b>cArea</b>	Nome de referência da área de trabalho a ser verificada
--------------	---

#### Exemplo:

```
nArea := Select("SA1")
```

```
ALERT("Referência do alias SA1: "+STRZERO(nArea,3)) // → 10 (proposto)
```

### DBGOTO()

Move o cursor da área de trabalho ativa para o record number (recno) especificado, realizando um posicionamento direto, sem a necessidade uma busca (seek) prévio.

- Sintaxe: DbGoto(nRecno)**
- Parâmetros**

<b>nRecno</b>	Record number do registro a ser posicionado
---------------	---

**Exemplo:**

```
DbSelectArea("SA1")
```

```
DbGoto(100) // Posiciona no registro 100
```

```
IF !EOF() // Se a área de trabalho não estiver em final de arquivo
```

```
    MsgInfo("Você está no cliente:"+A1_NOME)
```

```
ENDIF
```



*Anotações*

---

---

---

---

## DBGOTOP()

---

Move o cursor da área de trabalho ativa para o primeiro registro lógico.

- Sintaxe: DbGoTop()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

### Exemplo:

```
nCount := 0 // Variável para verificar quantos registros há no intervalo
DbSelectArea("SA1")
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
DbGoTop()

While !BOF() // Enquanto não for o início do arquivo
    nCount++ // Incrementa a variável de controle de registros no intervalo
    DbSkip(-1)
End

MsgInfo("Existem :"+STRZERO(nCount,6)+ "registros no intervalo").

// Retorno esperado :000001, pois o DbGoTop posiciona no primeiro registro.
```

## DBGOBOTTON()

---

Move o cursor da área de trabalho ativa para o último registro lógico.

- Sintaxe: DbGoBotton()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

### Exemplo:

```
nCount := 0 // Variável para verificar quantos registros há no intervalo
DbSelectArea("SA1")
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
DbGoBotton()

While !EOF() // Enquanto não for o início do arquivo
    nCount++ // Incrementa a variável de controle de registros no intervalo
    DbSkip(1)
End

MsgInfo("Existem :"+STRZERO(nCount,6)+ "registros no intervalo").

// Retorno esperado :000001, pois o DbGoBotton posiciona no último registro.
```

## DBSELECTAREA()

---

Define a área de trabalho especificada como sendo a área ativa. Todas as operações subsequentes que fizerem referência a uma área de trabalho para utilização, a menos que a área desejada seja informada explicitamente.

- ☑ **Sintaxe: DbSelectArea(nArea | cArea)**
- ☑ **Parâmetros**

<b>nArea</b>	Valor numérico que representa a área desejada, em função de todas as áreas já abertas pela aplicação, que pode ser utilizado ao invés do nome da área
<b>cArea</b>	Nome de referência da área de trabalho a ser selecionada

**Exemplo 01: DbselectArea(nArea)**

nArea := Select("SA1") // → 10 (proposto)

DbSelectArea(nArea) // De acordo com o retorno do comando Select().

ALERT("Nome do cliente: "+A1\_NOME) // Como o SA1 é o alias selecionado, os comandos  
// a partir da seleção do alias compreendem que ele  
// está implícito na expressão, o que causa o mesmo  
// efeito de SA1->A1\_NOME.

**Exemplo 01: DbselectArea(cArea)**

DbSelectArea("SA1") // Especificação direta do alias que deseja-se selecionar.

ALERT("Nome do cliente: "+A1\_NOME) // Como o SA1 é o alias selecionado, os comandos  
// a partir da seleção do alias compreendem que ele  
// está implícito na expressão, o que causa o mesmo  
// efeito de SA1->A1\_NOME.

## DBSETORDER()

---

Define qual índice será utilizado pela área de trabalho ativa, ou seja, pela área previamente selecionada através do comando DbSelectArea(). As ordens disponíveis no Ambiente Protheus são aquelas definidas no SINDEIX /SIX, ou as ordens disponibilizadas por meio de índices temporários.

- Sintaxe: DbSetOrder(nOrdem)**
- Parâmetros**

<b>nOrdem</b>	Número de referência da ordem que deseja ser definida como ordem ativa para a área de trabalho
---------------	--

### Exemplo:

```
DbSelectArea("SA1")
DbSetOrder(1) // De acordo com o arquivo SIX -> A1_FILIAL+A1_COD+A1_LOJA
```

## DBORDERNICKNAME()

---

Define qual índice criado pelo usuário será utilizado. O usuário pode incluir os seus próprios índices e no momento da inclusão deve criar o NICKNAME para o mesmo.

- Sintaxe: DbOrderNickName(NickName)**
- Parâmetros**

<b>NickName</b>	NickName atribuído ao índice criado pelo usuário.
-----------------	---

### Exemplo:

```
DbSelectArea("SA1")
DbOrderNickName("Tipo") // De acordo com o arquivo SIX -> A1_FILIAL+A1_TIPO
NickName: Tipo
```

## DBSEEK() E MSSEEK()

---

**DbSeek():** Permite posicionar o cursor da área de trabalho ativo no registro com as informações especificadas na chave de busca, fornecendo um retorno lógico e indicando se o posicionamento foi efetuado com sucesso, ou seja, se a informação especificada, na chave de busca, foi localizada na área de trabalho.

☑ **Sintaxe: DbSeek(cChave, ISoftSeek, ILast)**

☑ **Parâmetros**

<b>cChave</b>	Dados do registro que se deseja localizar, de acordo com a ordem de busca previamente especificada pelo comando DbSetOrder(), ou seja, de acordo com o índice ativo no momento para a área de trabalho.
<b>ISoftSeek</b>	Define se o cursor ficará posicionado no próximo registro válido, em relação à chave de busca especificada, ou em final de arquivo, caso não seja encontrada exatamente a informação da chave. Padrão → .F.
<b>ILast</b>	Define se o cursor será posicionado no primeiro ou no último registro de um intervalo, com as mesmas informações especificadas na chave. Padrão → .F.

### Exemplo 01 – Busca exata

```
DbSelectArea("SA1")
DbSetOrder(1) // acordo com o arquivo SIX -> A1_FILIAL+A1_COD+A1_LOJA

IF DbSeek("01" + "000001" + "02" ) // Filial: 01, Código: 000001, Loja: 02

    MsgInfo("Cliente localizado", "Consulta por cliente")

Else

    MsgAlert("Cliente não encontrado", "Consulta por cliente")

Endif
```

## Exemplo 02 – Busca aproximada

```
DbSelectArea("SA1")
```

```
DbSetOrder(1) // acordo com o arquivo SIX -> A1_FILIAL+A1_COD+A1_LOJA
```

```
DbSeek("01" + "000001" + "02", .T. ) // Filial: 01, Código: 000001, Loja: 02
```

```
// Exibe os dados do cliente localizado, o qual pode não ser o especificado na chave:
```

```
MsgInfo("Dados do cliente localizado: "+CRLF +;
```

```
    "Filial:"    + A1_FILIAL  + CRLF +;
```

```
    "Código:"   + A1_COD     + CRLF +;
```

```
    "Loja:"     + A1_LOJA    + CRLF +;
```

```
    "Nome:"     + A1_NOME    + CRLF, "Consulta por cliente")
```

**MsSeek():** Função desenvolvida pela área de Tecnologia da Microsiga, a qual possui as mesmas funcionalidades básicas da função DbSeek(), com a vantagem de não necessitar novamente do acesso da base de dados para localizar uma informação já utilizada pela thread (conexão) ativa.

Desta forma, a thread mantém em memória os dados necessários para reposicionar os registros já localizados através do comando DbSeek (no caso o Recno()), de forma que a aplicação pode simplesmente efetuar o posicionamento sem executar novamente a busca.

A diferença entre o DbSeek() e o MsSeek() é notada em aplicações com grande volume de posicionamentos, como relatórios, que necessitam referenciar diversas vezes o mesmo registro, durante uma execução.

## **DBSKIP()**

---

Move o cursor do registro posicionado para o próximo (ou anterior dependendo do parâmetro), em função da ordem ativa para a área de trabalho.

- Sintaxe: DbSkip(nRegistros)**
- Parâmetros**

<b>nRegistros</b>	Define em quantos registros o cursor será deslocado. Padrão → 1
-------------------	---

### **Exemplo 01 – Avançando registros**

```
DbSelectArea("SA1")
DbSetOrder(2) // A1_FILIAL + A1_NOME
DbGotop() // Posiciona o cursor no início da área de trabalho ativa.

While !EOF() // Enquanto o cursor da área de trabalho ativa não indicar fim de arquivo
    MsgInfo("Você está no cliente:" + A1_NOME)
    DbSkip()
End
```

### **Exemplo 02 – Retrocedendo registros**

```
DbSelectArea("SA1")
DbSetOrder(2) // A1_FILIAL + A1_NOME
DbGoBotton() // Posiciona o cursor no final da área de trabalho ativa.

While !BOF() // Enquanto o cursor da área de trabalho ativa não indicar início de arquivo
    MsgInfo("Você está no cliente:" + A1_NOME)
    DbSkip(-1)
End
```

## DBSETFILTER()

---

Define um filtro para a área de trabalho ativa, o qual pode ser descrito na forma de um bloco de código ou através de uma expressão simples.

- ☑ **Sintaxe: DbSetFilter(bCondicao, cCondicao)**
- ☑ **Parâmetros**

<b>bCondicao</b>	Bloco que expressa a condição de filtro em forma executável
<b>cCondicao</b>	Expressão de filtro simples na forma de string

### Exemplo 01 – Filtro com bloco de código

```
bCondicao := {|| A1_COD >= "000001" .AND. A1_COD <= "001000"}
DbSelectArea("SA1")
DbSetOrder(1)
DbSetFilter(bCondicao)
DbGoBotton()

While !EOF()
    MsgInfo("Você está no cliente:" + A1_COD)
    DbSkip()
End

// O último cliente visualizado deve ter o código menor do que "001000".
```

### Exemplo 02 – Filtro com expressão simples

```
cCondicao := "A1_COD >= '000001' .AND. A1_COD <= '001000'"
DbSelectArea("SA1")
DbSetOrder(1)
DbSetFilter(cCondicao)
DbGoBotton()

While !EOF()
    MsgInfo("Você está no cliente:" + A1_COD)
    DbSkip()
End

// O último cliente visualizado deve ter o código menor do que "001000".
```

## DBSTRUCT()

Retorna um array contendo a estrutura da área de trabalho (alias) ativo. A estrutura será um array bidimensional conforme abaixo:

I	No	Tip	Tama	Deci
D	me	o	nho	mais
*	ca	ca		
	mp	mp		
	o	o		

\*Índice do array

**Sintaxe: DbStruct()**

**Parâmetros**

Nenhum

### Exemplo:

```
cCampos := ""
DbSelectArea("SA1")
aStructSA1 := DbStruct()

FOR nX := 1 to Len(aStructSA1)

    cCampos += aStructSA1[nX][1] + "/"

NEXT nX

ALERT(cCampos)
```

## RECLOCK()

---

Efetua o travamento do registro posicionado na área de trabalho ativa, permitindo a inclusão ou alteração das informações do mesmo.

- Sintaxe: RecLock(cAlias,lInclui)**
- Parâmetros**

<b>cAlias</b>	Alias que identifica a área de trabalho que será manipulada.
<b>lInclui</b>	Define se a operação será uma inclusão (.T.) ou uma alteração (.F.).

### Exemplo 01 - Inclusão

```
DbSelectArea("SA1")
RecLock("SA1",.T.)
SA1->A1_FILIAL := xFilial("SA1") // Retorna a filial de acordo com as configurações do ERP.
SA1->A1_COD := "900001"
SA1->A1_LOJA := "01"
MsUnlock() // Confirma e finaliza a operação.
```

### Exemplo 02 - Alteração

```
DbSelectArea("SA1")
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
DbSeek("01" + "900001" + "01") // Busca exata

IF Found() // Avalia o retorno do último DbSeek realizado
RecLock("SA1",.F.)
SA1->A1_NOME := "CLIENTE CURSO ADVPL BÁSICO"
SA1->A1_NREDUZ := "ADVPL BÁSICO"
MsUnlock() // Confirma e finaliza a operação
```

ENDIF



Dica

A linguagem ADVPL possui variações da função **RecLock()**, as quais são:

- RLOCK()**
- DBRLOCK()**

A sintaxe e a descrição destas funções estão disponíveis no Guia de Referência Rápido ao final deste material.

## MSUNLOCK()

---

Libera o travamento (lock) do registro posicionado, confirmando as atualizações efetuadas neste registro.

- Sintaxe: MsUnLock()**
- Parâmetros**

Nenhum	.
--------	---

### Exemplo:

```
DbSelectArea("SA1")
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
DbSeek("01" + "900001" + "01") // Busca exata

IF Found() // Avalia o retorno do último DbSeek realizado
RecLock("SA1",.F.)
```

SA1->A1\_NOME := "CLIENTE CURSO ADVPL BÁSICO"

```
SA1->A1_NREDUZ := "ADVPL BÁSICO"  
MsUnlock() // Confirma e finaliza a operação  
ENDIF
```



Dica

***A linguagem ADVPL possui variações da função MsUnlock(), as quais são:***

***UNLOCK()***

***DBUNLOCK()***

***DBUNLOCKALL()***

***A sintaxe e a descrição destas funções estão disponíveis no Guia de Referência Rápido ao final deste material.***

## **SOFTLOCK()**

---

Permite a reserva do registro posicionado na área de trabalho ativa de forma que outras operações, com exceção da atual, não possam atualizar este registro. Difere da função RecLock(), pois não gera uma obrigação de atualização, e pode ser sucedido por ele.

Na aplicação ERP Protheus, o SoftLock() é utilizado nos browses, antes da confirmação da operação de alteração e exclusão, pois neste momento a mesma ainda não foi efetivada, mas outras conexões não podem acessar aquele registro, pois o mesmo está em manutenção, o que implementa a integridade da informação.

- Sintaxe: SoftLock(cAlias)**
- Parâmetros**

<b>cAlias</b>	Alias de referência da área de trabalho ativa, para o qual o registro posicionado será travado.
---------------	---

**Exemplo:**

```
cChave := GetCliente() // Função ilustrativa que retorna os dados de busca de um cliente.

DbSelectArea("SA1")
DbSetOrder(1)
DbSeek(cChave)

IF Found()
    SoftLock() // Reserva o registro localizado
    IConfirma := AlteraSA1() // Função ilustrativa que exibe os dados do registro
    // posicionado e permite a alteração dos mesmos.

    IF IConfirma
        RecLock("SA1",.F.)
        GravaSA1() // Função ilustrativa que altera os dados conforme a AlteraSA1().
        MsUnLock() // Liberado o RecLock() e o SoftLock() do registro.
    Endif
Endif
```

## DBDELETE()

---

Efetua a exclusão lógica do registro posicionado na área de trabalho ativa, sendo necessária sua utilização em conjunto com as funções RecLock() e MsUnLock().

- Sintaxe: DbDelete()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

### Exemplo:

```
DbSelectArea("SA1")
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
DbSeek("01" + "900001" + "01") // Busca exata

IF Found()
  RecLock("SA1",.F.) // Define que será realizada uma alteração no registro posicionado.
  DbDelete() // Efetua a exclusão lógica do registro posicionado.
  MsUnLock() // Confirma e finaliza a operação.
ENDIF
```

## DBUSEAREA()

---

Define um arquivo de base de dados como uma área de trabalho disponível na aplicação.

- Sintaxe: DbUseArea(INovo, cDriver, cArquivo, cAlias, ICompartilhado,; ISoLeitura)**

**Parâmetros**

<b>INovo</b>	Parâmetro opcional que permite, caso o cAlias especificado já esteja em uso, ser fechado antes da abertura do arquivo da base de dados.
<b>cDriver</b>	Driver que permita a aplicação manipular o arquivo de base de dados especificado. A aplicação ERP possui a variável __LOCALDRIVER, definida a partir das configurações do .ini do server da aplicação.  Algumas chaves válidas: "DBFCDX", "CTRECDX", "DBFCDXAX", "TOPCONN".
<b>cArquivo</b>	Nome do arquivo de base de dados que será aberto com o alias especificado.
<b>cAlias</b>	Alias para referência do arquivos de base de dados pela aplicação.
<b>ICompartilhado</b>	Se o arquivo poderá ser utilizado por outras conexões.
<b>ISoLeitura</b>	Se o arquivo poderá ser alterado pela conexão ativa.

**Exemplo:**

```
DbUserArea(.T., "DBFCDX", "\SA1010.DBF", "SA1DBF", .T., .F.)  
DbSelectArea("SA1DBF")  
MsgInfo("A tabela SA1010.DBF possui:" + STRZERO(RecCount(),6) + " registros.")  
DbCloseArea()
```

**DBCLOSEAREA()**

Permite que um alias presente na conexão seja fechado, o que viabiliza novamente seu uso em outro operação. Este comando tem efeito apenas no alias ativo na conexão, sendo necessária sua utilização em conjunto com o comando DbSelectArea().

**Sintaxe: DbCloseArea()**

**Parâmetros**

<b>Nenhum</b>	.
---------------	---

**Exemplo:**

```

DbUserArea(.T., "DBFCDX", "\SA1010.DBF", "SA1DBF", .T., .F.)
DbSelectArea("SA1DBF")
MsgInfo("A tabela SA1010.DBF possui:" + STRZERO(RecCount(),6) + " registros.")
DbCloseArea()

```

## Controle de numeração sequencial

### GETSXENUM()

Obtém o número sequência do alias especificado no parâmetro, através da referência aos arquivos de sistema SXE/SXF ou ao servidor de numeração, quando esta configuração está habilitada no ambiente Protheus.

- Sintaxe: GETSXENUM(cAlias, cCampo, cAliasSXE, nOrdem)**
- Parâmetros**

<b>cAlias</b>	Alias de referência da tabela para a qual será efetuado o controle da numeração sequencial.
<b>cCampo</b>	Nome do campo no qual está implementado o controle da numeração.
<b>cAliasSXE</b>	Parâmetro opcional, quando o nome do alias nos arquivos de controle de numeração não é o nome convencional do alias para o sistema ERP.
<b>nOrdem</b>	Número do índice para verificar qual a próxima ocorrência do número.

### CONFIRMSXE()

Confirma o número alocado através do último comando GETSXENUM().

- Sintaxe: CONFIRMSXE(IVerifica)**
- Parâmetros**

<b>I</b> verifica	Verifica se o número confirmado não foi alterado, e se por consequência já existe na base de dados.
-------------------	---

## **ROLLBACKSXE()**

---

Descarta o número fornecido pelo último comando GETSXENUM(), retornando a numeração disponível para outras conexões.

- Sintaxe: ROLLBACKSXE()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

## **Validação**

---

### **EXISTCHAV()**

---

Retorna .T. ou .F. se o conteúdo especificado existe no alias especificado. Se existir será exibido um help de sistema com um aviso informando da ocorrência.

A Função utilizada normalmente para verificar se um determinado código de cadastro já existe na tabela na qual a informação será inserida, como por exemplo o CNPJ no cadastro de clientes ou, fornecedores.

- Sintaxe: ExistChav(cAlias, cConteudo, nIndice)**
- Parâmetros**

<b>cAlias</b>	Alias de referência para a validação da informação
<b>cConteudo</b>	Chave a ser pesquisada, sem a filial

<b>nÍndice</b>	Índice de busca para consulta da chave
----------------	--

## **EXISTCPO()**

---

Retorna .T. ou .F. se o conteúdo especificado não existe no alias especificado. Caso não exista será exibido um help de sistema com um aviso informando da ocorrência.

Função utilizada normalmente para verificar se a informação digitada em um campo, a qual depende de outra tabela, realmente existe nesta outra tabela, como por exemplo o código de um cliente em um pedido de venda.

- Sintaxe: ExistCpo(cAlias, cConteudo, nÍndice)**
- Parâmetros**

<b>cAlias</b>	Alias de referência para a validação da informação
<b>cConteudo</b>	Chave a ser pesquisada, sem a filial
<b>nÍndice</b>	Índice de busca para consulta da chave

## **NAOVAZIO()**

---

Retorna .T. ou .F. se o conteúdo do campo posicionado no momento não está vazio.

- Sintaxe: NaoVazio()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

## **NEGATIVO()**

---

Retorna .T. ou .F. se o conteúdo digitado para o campo é negativo.

- Sintaxe: Negativo()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---

## **PERTENCE()**

---

Retorna .T. ou .F. se o conteúdo digitado para o campo está contido na string definida como parâmetro da função. Normalmente utilizada em campos com a opção de combo, pois caso contrário seria utilizada a função ExistCpo().

- Sintaxe: Pertence(cString)**
- Parâmetros**

<b>cString</b>	String contendo as informações válidas que podem ser digitadas para um campo.
----------------	---

## **POSITIVO()**

---

Retorna .T. ou .F. se o conteúdo digitado para o campo é positivo.

- Sintaxe: Positivo()**
- Parâmetros**

Nenhum	.
--------	---

### **TEXTO()**

---

Retorna .T. ou .F. se o conteúdo digitado para o campo contém apenas números ou alfanuméricos.

- Sintaxe: Texto()**
- Parâmetros**

Nenhum	.
--------	---

### **VAZIO()**

---

Retorna .T. ou .F. se o conteúdo do campo posicionado no momento está vazio.

- Sintaxe: Vazio()**
- Parâmetros**

Nenhum	.
--------	---

## **Parâmetros**

---

### **GETMV()**

---

Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Se o parâmetro não existir, será exibido um help do sistema informando a ocorrência.

- Sintaxe: GETMV(cParametro)**
- Parâmetros**

<b>cParametro</b>	Nome do parâmetro do sistema no SX6, sem a especificação da filial de sistema.
-------------------	--

### **GETNEWPAR()**

---

Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Se o parâmetro não existir, será exibido um help do sistema informando a ocorrência.

Difere do SuperGetMV() pois considera que o parâmetro pode não existir na versão atual do sistema, e por consequência não será exibida a mensagem de help.

- Sintaxe: GETNEWPAR(cParametro, cPadrao, cFilial)**
- Parâmetros**

<b>cParametro</b>	Nome do parâmetro do sistema no SX6, sem a especificação da filial de sistema.
<b>cPadrao</b>	Conteúdo padrão que será utilizado, caso o parâmetro não exista no SX6.
<b>cFilial</b>	Define para qual filial será efetuada a consulta do parâmetro. Padrão → filial corrente da conexão.

## **PUTMV()**

---

Atualiza o conteúdo do parâmetro especificado no arquivo SX6, de acordo com as parametrizações informadas.

- Sintaxe: PUTMV(cParametro, cConteudo)**
- Parâmetros**

<b>cParametro</b>	Nome do parâmetro do sistema no SX6, sem a especificação da filial de sistema
<b>cConteudo</b>	Conteúdo que será atribuído ao parâmetro no SX6

## **SUPERGETMV()**

---

Retorna o conteúdo do parâmetro especificado no arquivo SX6, considerando a filial parametrizada na conexão. Caso o parâmetro não exista, será exibido um help do sistema informando a ocorrência.

Difere do GetMv(), pois os parâmetros consultados são adicionados em uma área de memória, que permite que em uma nova consulta não seja necessário acessar e pesquisar o parâmetro na base de dados.

- Sintaxe: SUPERGETMV(cParametro , IHelp , cPadrao , cFilial)**
- Parâmetros**

<b>cParametro</b>	Nome do parâmetro do sistema no SX6, sem a especificação da filial de sistema.
<b>IHelp</b>	Se será exibida a mensagem de Help, caso o parâmetro não seja encontrado no SX6.
<b>cPadrao</b>	Conteúdo padrão que será utilizado caso o parâmetro não exista no SX6.

<b>cFilial</b>	Define para qual filial será efetuada a consulta do parâmetro. Padrão → filial corrente da conexão.
----------------	---

## Componentes da interface visual

### MSDIALOG()

Define o componente MSDIALOG() que é utilizado como base para os demais componentes da interface visual, pois um componente MSDIALOG() é uma janela da aplicação.

**Sintaxe:**

```
DEFINE MSDIALOG oObjetoDLG TITLE cTitulo FROM nLinIni,nColIni TO nLiFim,nColFim OF  
oObjetoRef UNIDADE
```

**Parâmetros**

oObjetoDLG	Posição do objeto Say em função da janela em que ele será definido
cTitulo	Título da janela de diálogo
nLinIni, nColIni	Posição inicial em linha / coluna da janela
nLiFim, nColFim	Posição final em linha / coluna da janela
oObjetoRef	Objeto diálogo no qual a janela será definida
UNIDADE	Unidade de medida das dimensões: PIXEL

**Exemplo:**

```
DEFINE MSDIALOG oDlg TITLE cTitulo FROM 000,000 TO 080,300 PIXEL  
ACTIVATE MSDIALOG oDlg CENTERED
```



**Anotações**

---

---

---

---

---

## MSGET()

---

Define o componente visual MSGET, o qual é utilizado para captura de informações digitáveis na tela da interface.

**Sintaxe:**

```
@ nLinha, nColuna MSGET VARIAVEL SIZE nLargura,nAltura UNIDADE OF oObjetoRef F3 cF3  
VALID VALID WHEN WHEN PICTURE cPicture
```

**Parâmetros**

nLinha, nColuna	Posição do objeto MsGet em função da janela em que ele será definido
VARIAVEL	Variável da aplicação que será vinculada ao objeto MsGet, que definirá suas características e na qual será armazenado o que for informado no campo
nLargura,nAltura	Dimensões do objeto MsGet para exibição do texto
UNIDADE	Unidade de medida das dimensões: PIXEL
oObjetoRef	Objeto dialog no qual o componente será definido
cF3	String que define a consulta padrão, a qual será vinculada ao campo
VALID	Função de validação para o campo.
WHEN	Condição para manipulação do campo, a qual pode ser diretamente .T. ou .F., ou uma variável ou uma chamada de função.
cPicture	String contendo a definição da Picture de digitação do campo.

**Exemplo:**

```
@ 010,050 MSGET cCGC SIZE 55, 11 OF oDIg PIXEL PICTURE "@R 99.999.999/9999-99";  
VALID !Vazio()
```



*Anotações*

---

---

---

---

## SAY()

---

Define o componente visual SAY, o qual é utilizado para exibição de textos em uma tela de interface.

**Sintaxe:**

```
@ nLinha, nColuna SAY cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef
```

**Parâmetros**

nLinha, nColuna	Posição do objeto Say em função da janela em que ele será definido
cTexto	Texto que será exibido pelo objeto Say
nLargura,nAltura	Dimensões do objeto Say para exibição do texto
UNIDADE	Unidade de medida das dimensões: PIXEL
oObjetoRef	Objeto dialog no qual o componente será definido

**Exemplo:**

```
@ 010,010 SAY cTexto SIZE 55, 07 OF oDlg PIXEL
```

## BUTTON()

---

Define o componente visual Button, o qual permite a inclusão de botões de operação na tela da interface, os quais serão visualizados somente com um texto simples para sua identificação.

**Sintaxe: BUTTON()**

```
@ nLinha,nColuna BUTTON cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef  
ACTION AÇÃO
```

**Parâmetros**

nLinha,nColuna	Posição do objeto Button em função da janela em que ele será definido
cTexto	String contendo o texto que será exibido no botão
nLargura,nAltura	Dimensões do objeto Button para exibição do texto
UNIDADE	Unidade de medida das dimensões: PIXEL
oObjetoRef	Objeto dialog no qual o componente será definido
AÇÃO	Função ou lista de expressões que define o comportamento do botão quando ele for utilizado

**Exemplo:**

```
010, 120 BUTTON "Confirmar" SIZE 080, 047 PIXEL OF oDlg;  
ACTION (nOpca := 1,oDlg:End())
```

**SBUTTON()**

Define o componente visual SButton, que permite a inclusão de botões de operação na tela da interface, os quais serão visualizados dependendo da interface do sistema ERP utilizada somente com um texto simples para sua identificação, ou com uma imagem (Bitmap) pré-definida.

**Sintaxe: SBUTTON()**

```
DEFINE SBUTTON FROM nLinha, nColuna TYPE N ACTION AÇÃO STATUS OF oObjetoRet
```

**Parâmetros**

nLinha, nColuna	Posição do objeto sButton em função da janela em que ele será definido
TYPE N	Número que indica o tipo do botão (imagem) pré-definida que será utilizada

AÇÃO	Função ou lista de expressões que define o comportamento do botão quando ele for utilizado
STATUS	Propriedade de uso do botão: ENABLE ou DISABLE
oObjetoRet	Objeto dialog no qual o componente será definido

**Exemplo:**

```
DEFINE SBUTTON FROM 020, 120 TYPE 2 ACTION (nOpca := 2,oDlg:End());
ENABLE OF oDlg
```

**Visual dos diferentes tipos de botões disponíveis:**



## Interfaces de cadastro

### AXCADASTRO()

<b>Sintaxe</b>	<b>AxCadastro(cAlias, cTitulo, cVIdExc, cVIdAlt)</b>
<b>Descrição</b>	O AxCadastro() é uma funcionalidade de cadastro simples, com poucas opções de customização.

## **MBROWSE()**

---

<b>Sintaxe</b>	<b>MBrowse(nLin1, nCol1, nLin2, nCol2, cAlias)</b>
<b>Descrição</b>	A Mbrowse() é uma funcionalidade de cadastro que permite a utilização de recursos mais aprimorados na visualização e manipulação das informações do sistema.

## **AXPESQUI()**

---

Função de pesquisa padrão em registros exibidos pelos browses do sistema, a qual posiciona o browse no registro pesquisado. Exibe uma tela que permite a seleção do índice a ser utilizado na pesquisa e a digitação das informações que compõe a chave de busca.

- Sintaxe: AXPESQUI()**
- Parâmetros**

<b>Nenhum</b>	.
---------------	---



*Anotações*

---

---

---

---

---

## **AXVISUAL()**

---

Função de visualização padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

**Sintaxe: AXVISUAL(cAlias, nReg, nOpc, aAcho, nColMens, cMensagem, cFunc,; aButtons, IMaximized )**

**Parâmetros**

<b>cAlias</b>	Tabela cadastrada no Dicionário de Tabelas (SX2) que será editada
<b>nReg</b>	Record number (recno) do registro posicionado no alias ativo
<b>nOpc</b>	Número da linha do aRotina que definirá o tipo de edição (Inclusão, Alteração, Exclusão, Visualização).
<b>aAcho</b>	Vetor com nome dos campos que serão exibidos. Os campos de usuário sempre serão exibidos se não existir no parâmetro um elemento com a expressão "NOUSER"
<b>nColMens</b>	Parâmetro não utilizado
<b>cMensagem</b>	Parâmetro não utilizado.
<b>cFunc</b>	Função que deverá ser utilizada para carregar as variáveis que serão utilizadas pela Enchoice. Neste caso o parâmetro IVirtual é definido internamente pela AxFunction() executada como .T.
<b>aButtons</b>	Botões adicionais para a EnchoiceBar, no formato: aArray[n][1] -> Imagem do botão; aArray[n][2] -> bloco de código contendo a ação do botão; aArray[n][3] -> título do botão.
<b>IMaximized</b>	Indica se a janela deverá ser ou não maximizada.

## **AXINCLUI()**

---

Função de inclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

**Sintaxe: AxInclui(cAlias, nReg, nOpc, aAcho, cFunc, aCpos, cTudoOk, IF3,; cTransact, aButtons, aParam, aAuto, IVirtual, IMaximized)**

**Parâmetros**

<b>cAlias</b>	Tabela cadastrada no Dicionário de Tabelas (SX2) que será editada
<b>nReg</b>	Record number (recno) do registro posicionado no alias ativo
<b>nOpc</b>	Número da linha do aRotina que definirá o tipo de edição (Inclusão, Alteração, Exclusão, Visualização)
<b>aAcho</b>	Vetor com nome dos campos que serão exibidos. Os campos de usuário sempre serão exibidos se não existir no parâmetro um elemento com a expressão "NOUSER"
<b>cFunc</b>	Função que deverá ser utilizada para carregar as variáveis que serão utilizadas pela Enchoice. Neste caso, o parâmetro IVirtual é definido internamente pela AxFunction(), executada como .T.
<b>aCpos</b>	Vetor com nome dos campos que poderão ser editados
<b>cTudoOk</b>	Função de validação de confirmação da tela. Não deve ser passada como Bloco de Código, mas pode ser passada como uma lista de expressões, desde que a última ação efetue um retorno lógico:  “(Func1(), Func2(), ...,FuncX(), .T. )”
<b>IF3</b>	Indica se a enchoice está sendo criada em uma consulta F3 para utilizar variáveis de memória
<b>cTransact</b>	Função que será executada dentro da transação da AxFunction()
<b>aButtons</b>	Botões adicionais para a EnchoiceBar, no formato:  aArray[n][1]           ->           Imagem           do           botão aArray[n][2]   ->   bloco de código contendo a ação do botão aArray[n][3]   ->   título do botão
<b>aParam</b>	Funções para execução em pontos pré-definidos da AxFunction(), conforme abaixo:  aParam[1] := Bloco de código que será processado antes da exibição da interface.  aParam[2] := Bloco de código para processamento na validação da confirmação.  aParam[3] := Bloco de código que será executado dentro da transação da

	AxFunction(). aParam[4] := Bloco de código que será executado fora da transação da AxFunction().
<b>aAuto</b>	Array no formato utilizado pela funcionalidade MsExecAuto(). Caso seja informado este array, não será exibida a tela de interface, e será executada a função EnchAuto(). aAuto[n][1] := Nome do campo aAuto[n][2] := Conteúdo do campo aAuto[n][3] := Validação que será utilizada em substituição as validações do SX3.
<b>IVirtual</b>	Indica se a Enchoice() chamada pela AxFunction() utilizará variáveis de memória ou os campos da tabela na edição
<b>IMaximized</b>	Indica se a janela deverá ser ou não maximizada

### **AXALTERA()**

---

Função de alteração padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

**Sintaxe: AXALTERA(cAlias, nReg, nOpc, aAcho, aCpos, nColMens, cMensagem,; cTudoOk, cTransact, cFunc, aButtons, aParam, aAuto, IVirtual, IMaximized)**

**Parâmetros**

➤ **Vide documentação de parâmetros da função AxInclui().**

## AXDELETA()

Função de exclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

☑ **Sintaxe: AXDELETA(cAlias, nReg, nOpc, cTransact, aCpos, aButtons, aParam,; aAuto, IMaximized)**

☑ **Parâmetros**

<b>cAlias</b>	Tabela cadastrada no Dicionário de Tabelas (SX2) que será editada
<b>nReg</b>	Record number (recno) do registro posicionado no alias ativo
<b>nOpc</b>	Número da linha do aRotina que definirá o tipo de edição (Inclusão, Alteração, Exclusão, Visualização)
<b>cTransact</b>	Função que será executada dentro da transação da AxFunction()
<b>aCpos</b>	Vetor com nome dos campos que poderão ser editados
<b>aButtons</b>	Botões adicionais para a EnchoiceBar, no formato: aArray[n][1] -> Imagem do botão aArray[n][2] -> bloco de código contendo a ação do botão aArray[n][3] -> título do botão
<b>aParam</b>	Funções para execução em pontos pré-definidos da AxFunction(), conforme abaixo: aParam[1] := Bloco de código que será processado antes da exibição da interface. aParam[2] := Bloco de código para processamento na validação da confirmação. aParam[3] := Bloco de código que será executado dentro da transação da AxFunction(). aParam[4] := Bloco de código que será executado fora da transação da AxFunction().
<b>aAuto</b>	Array no formato utilizado pela funcionalidade MsExecAuto(). Caso seja informado este array, não será exibida a tela de interface, e será executada a função EnchAuto(). aAuto[n][1] := Nome do campo aAuto[n][2] := Conteúdo do campo

	aAuto[n][3] := Validação que será utilizada em substituição as validações do SX3.
<b>IMaximized</b>	Indica se a janela deverá ser ou não maximizada

## Funções visuais para aplicações

### ALERT()

- Sintaxe: AVISO(cTexto)**
- Parâmetros**

<b>cTexto</b>	Texto a ser exibido
	

### AVISO()

- Sintaxe: AVISO(cTitulo, cTexto, aBotoes, nTamanho)**
- Retorno: numérico indicando o botão selecionado.**
  
- Parâmetros**

<b>cTitulo</b>	Título da janela
<b>cTexto</b>	Texto do aviso
<b>aBotoes</b>	Array simples (vetor) com os botões de opção
<b>nTamanho</b>	Tamanho (1,2 ou 3)



## FORMBATCH()

---

- ☑ **Sintaxe:** FORMBATCH(cTitulo, aTexto, aBotoes, bValid, nAltura, nLargura )
- ☑ **Parâmetros**

<b>cTitulo</b>	Título da janela
<b>aTexto</b>	Array simples (vetor) contendo cada uma das linhas de texto que serão exibidas no corpo da tela
<b>aBotoes</b>	Array com os botões do tipo SBUTTON(), com a seguinte estrutura:  <b>{nTipo, lEnable, {   Ação() }}</b>
<b>bValid</b>	(opcional) Bloco de validação do janela
<b>nAltura</b>	(opcional) Altura em pixels da janela
<b>nLargura</b>	(opcional) Largura em pixels da janela
	A diagrama de uma janela de diálogo criada com o FORMBATCH(). A janela tem um título "cTitulo" e uma barra de título azul com ícones de minimizar, maximizar e fechar. O corpo da janela contém um campo de texto rotulado "cTexto" e dois botões "Ok" e "Cancelar" na parte inferior direita.

## MSGFUNCTIONS()

---

- ☑ **Sintaxe: MSGALERT(cTexto, cTitulo)**
- ☑ **Sintaxe: MSGINFO(cTexto, cTitulo)**
- ☑ **Sintaxe: MSGSTOP(cTexto, cTitulo)**
- ☑ **Sintaxe: MSGYESNO(cTexto, cTitulo)**

☑ **Parâmetros**

<b>cTexto</b>	Texto a ser exibido como mensagem
<b>cTitulo</b>	Título da janela de mensagem
<b>MSGALERT</b>	 A screenshot of a Windows-style dialog box titled "cTitulo". On the left side, there is a yellow triangular warning icon with a black exclamation mark. To the right of the icon, the text "cTexto" is displayed. At the bottom right of the dialog, there is an "Ok" button.
<b>MSGINFO</b>	 A screenshot of a Windows-style dialog box titled "cTitulo". On the left side, there is a blue speech bubble icon containing a white lowercase letter "i". To the right of the icon, the text "cTexto" is displayed. At the bottom right of the dialog, there is an "Ok" button.
<b>MSGSTOP</b>	 A screenshot of a Windows-style dialog box titled "cTitulo". On the left side, there is a red circular icon with a white "X" inside. To the right of the icon, the text "cTexto" is displayed. At the bottom right of the dialog, there is an "Ok" button.

**MSGYESNO**



## Funções ADVPL para aplicações

### GETAREA()

Função utilizada para proteger o ambiente ativo no momento de algum processamento específico. Para salvar uma outra área de trabalho (alias) que não o ativo, a função GetArea() deve ser executada dentro do alias: ALIAS->(GetArea()).

- Sintaxe: GETAREA()**
- Retorno: Array contendo {Alias(),IndexOrd(),Recno()}**

- Parâmetros**

<b>Nenhum</b>	.
---------------	---

### RESTAREA()

Função utilizada para devolver a situação do ambiente salva, através do comando GETAREA(). Deve-se observar que a última área restaurada é a área que ficará ativa para a aplicação.

- Sintaxe: RESTAREA(aArea)**
- Parâmetros**

<b>aArea</b>	Array contendo: {cAlias, nOrdem, nRecno}, normalmente gerado pelo uso da função GetArea().
--------------	--

#### Exemplo:

```
// ALIAS ATIVO ANTES DA EXECUÇÃO DA ROTINA → SN3
User Function XATF001()

LOCAL cVar
LOCAL aArea := GetArea()
LOCAL IRet := .T.
```

```
cVar := &(ReadVar())
```

```
dbSelectArea("SX5")
```

```
IF !dbSeek(xFilial)+"Z1"+cVar
```

```
    cSTR0001 := "REAV - Tipo de Reavaliacao"
```

```
    cSTR0002 := "Informe um tipo de reavaliacao valido"
```

```
    cSTR0003 := "Continuar"
```

```
    Aviso(cSTR0001,cSTR0002,{cSTR0003},2)
```

```
    IRet := .F.
```

```
ENDIF
```

```
RestArea(aArea)
```

```
Return( IRet )
```

## REFERÊNCIAS BIBLIOGRÁFICAS

### Referências bibliográficas

---

Gestão empresarial com ERP

**Ernesto Haberkorn, 2006**

Lógica de Programação – A Construção de Algoritmos e Estruturas de Dados

**Forbellone, André Luiz Villar - MAKRON, 1993**

Introdução à Programação - 500 Algoritmos Resolvidos

**Anita Lopes, Guto Garcia – CAMPUS / ELSEVIER, 2002**

Apostila de Treinamento - ADVPL

**Educação corporativa**

Apostila de Treinamento – Introdução à programação

**Educação corporativa**

Apostila de Treinamento – Boas Práticas de Programação

**Inteligência Protheus e Fábrica de Software**

Curso Básico de Lógica de Programação

**Paulo Sérgio de Moraes – PUC Campinas**

**TDN – TOTVS Developer Network**

**Microsiga Software S.A.**

Materiais diversos de colaboradores Totvs

**Colaboradores Totvs**