

Todos os direitos autorais reservados pela **TOTVS S.A.**

Proibida a reprodução total ou parcial, bem como a armazenagem em sistema de recuperação e a transmissão, de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros, sem prévia autorização por escrito da proprietária.

O desrespeito a essa proibição configura em apropriação indevida dos direitos autorais e patrimoniais da TOTVS.

Conforme artigos 122 e 130 da LEI no. 5.988 de 14 de Dezembro de 1973.

ADPVL Fundamental

Protheus – Versão 12



Sumário

1. Objetivo do Curso	5
2. A Linguagem ADVPL	5
2.1. Programação Com Interface Própria com o Usuário	5
2.2. Programação Sem Interface Própria com o Usuário.....	6
3. Estrutura de um programa ADVPL	6
3.1. Áreas de um Programa ADVPL	8
3.2. Área de Identificação	9
3.3. Área de Ajustes Iniciais.....	10
3.4. Corpo do Programa	10
3.5. Área de Encerramento	11
4. Declaração E Atribuição De Variáveis	11
4.1. Tipo de Dados	11
4.2. Numérico	11
4.3. Lógico	11
4.4. Caractere	11
4.5. Data	12
4.6. Array	12
4.7. Bloco de Código	12
4.8. Declaração de variáveis.....	12
5. Escopo de variáveis.....	13
5.1. O Contexto de Variáveis dentro de um Programa	13
5.2. Variáveis de escopo local	13
5.3. Variáveis de escopo static	14
5.4. Variáveis de escopo private	14
5.5. Variáveis de escopo public	15
5.6. Entendendo a influência do escopo das variáveis	16
5.7. Operações com Variáveis.....	17
6. Operadores da linguagem ADVPL.....	18
6.1. Operadores comuns	18
6.1.1. Operadores Matemáticos	18
6.1.2. Operadores de String.....	18
6.1.3. Operadores Relacionais.....	18
6.1.4. Operadores Lógicos.....	19
6.1.5. Operadores de Atribuição	19
6.1.6. Atribuição em Linha.....	19
6.1.7. Atribuição Composta	20
6.1.8. Operadores de Incremento/Decremento	20
6.1.9. Operadores Especiais	21

6.1.10. Ordem de Precedência dos Operadores.....	21
7. Operação de Macro Substituição.....	22
8. Funções de manipulação de variáveis.....	23
9. Conversões entre tipos de variáveis.....	23
9.1. Manipulação de strings.....	25
9.2. Manipulação de variáveis numéricas.....	27
9.3. Verificação de tipos de variáveis.....	28
10. Estruturas Básicas De Programação.....	29
11. Estruturas de repetição.....	29
11.1. FOR...NEXT.....	29
11.2. WHILE...ENDDO.....	30
11.2.1. Influenciando o fluxo de repetição.....	31
12. Estruturas de decisão.....	32
12.1. IF...ELSE...ENDIF.....	33
12.2. IF...ELSEIF...ELSE...ENDIF.....	33
12.3. DO CASE...ENDCASE.....	34
13. Arrays E Blocos De Código.....	35
13.1. Arrays.....	35
13.2. Arrays como Estruturas.....	36
13.3. Funções de manipulação de arrays.....	38
13.4. Cópia de arrays.....	40
14. Blocos de Código.....	42
15. FUNÇÕES.....	43
15.1. Tipos e escopos de funções.....	44
16. Passagem de parâmetros entre funções.....	45
16.1. Passagem de parâmetros por conteúdo.....	46
16.2. Passagem de parâmetros por referência.....	47
16.2.1. Tratamento de conteúdos padrões para parâmetros de funções.....	49
18. diretivas De Compilação.....	50
19. Diretiva: #INCLUDE.....	50
20. Diretiva: #DEFINE.....	51
21. Diretivas: #IFDEF, #ifndef, #ELSE e #ENDIF.....	51
22. Diretiva: #COMMAND.....	52
23. Desenvolvimento De Pequenas Customizações.....	53
23.1. Acesso E Manipulação De Bases De Dados Em Advpl.....	53
23.2. Funções de acesso e manipulação de dados.....	55
24. Diferenciação entre variáveis e nomes de campos.....	58
25. Controle de numeração sequencial.....	59
26. Semáforos.....	59
26.1. Funções de controle de semáforos e numeração sequencial.....	59

27. Customizações Para A Aplicação Erp	60
27.1. Customização de campos – Dicionário de Dados.....	60
27.2. Pictures de formação disponíveis	62
27.3. Customização de gatilhos – Configurador	63
27.4. Customização de parâmetros – Configurador	64
27.4.1. Cuidados na utilização de um parâmetro	65
28. Pontos de Entrada – Conceitos, Premissas e Regras	66
28.1. Premissas e Regras.....	66
29. Interfaces Visuais	67
29.1. Sintaxe e componentes das interfaces visuais	67
29.2. Interface visual completa	68
29.3. MBrowse()	70
29.4. AxFUNCTIONS().....	72
30. apêndices	73
30.1. Boas Práticas De Programação.....	73
30.1.1. Utilização De Identação	73
30.2. Capitulação De Palavras-Chaves	75
30.2.1. Palavras em maiúsculo	75
1.1.1. Utilização Da Notação Húngara	75
1.1.2. Palavras Reservadas	76
31. Guia De Referência Rápida: Funções E Comandos Advpl.....	76
1.2. Conversão entre tipos de dados	77
31.1. Verificação de tipos de variáveis.....	79
31.2. Manipulação de arrays.....	80
31.3. Manipulação de blocos de código.....	85
31.4. Manipulação de strings.....	87
31.5. Manipulação de variáveis numéricas.....	91
31.6. Manipulação de arquivos	92
32. Funções visuais para aplicações	101

1. Objetivo 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 TOTVS.

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.

2. 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 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

2.1. Programação Com Interface Própria com o Usuário

Nesta categoria, entram os programas desenvolvidos para serem executados 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 a execução de rotinas escritas em ADVPL no Protheus Server, e por isso permite executar qualquer tipo de código, tenha ele interface com o usuário ou não.

Porém, nesta categoria são considerados apenas os programas que realizem algum tipo de interface remota, utilizando o protocolo de comunicação do Protheus. Podem-se criar 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.

2.2. 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. Estas rotinas são apenas processos, ou Jobs, executados no Protheus Server. Algumas vezes, a interface destas 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:** Através de uma biblioteca de funções disponível no Protheus (uma API de comunicação), podem-se executar rotinas escritas em ADVPL diretamente no Protheus Server, por aplicações externas escritas em outras linguagens. Isto 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, 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 destas 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, através de uma aplicação cliente deste protocolo. O PROTHEUS Server pode emular um terminal pela execução de rotinas escritas em ADVPL. Ou seja, pode-se 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, através de um processo chamado 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 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 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. Pode-se utilizar a palavra NOTE ou dois símbolos da letra "e" comercial (&&) para realizar a função do sinal de asterisco. Porém todas estas formas de comentário de linhas são obsoletas e existem apenas para 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 utilizar as barras transversais juntamente com o asterisco, podendo-se 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, pode-se "quebrar" a linha lógica em mais de uma linha física no arquivo texto. Isto 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
```

Neste exemplo existe uma linha de comando para a checagem das variáveis utilizadas. Como a linha torna-se muito grande, pode-se 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
MsgAlert("O fatorial de " + cValToChar(nFator) + ;
        " é " + cValToChar(nResultado))

// Termina o programa
Return( NIL )
```

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

3.2. 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. Pode-se colocar qualquer tipo de informação desejada e 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 podem-se incluir definições de constantes utilizadas no programa ou inclusão de arquivos de cabeçalho nesta área.

3.3. Área de Ajustes Iniciais

Nesta área geralmente se fazem 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 nesta área visando tornar o código mais legível e facilitar a identificação de variáveis não utilizadas.

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

3.4. Corpo do Programa

É nesta área que se encontram as linhas de código do programa. É onde se realiza a tarefa necessária por meio da organização lógica destas linhas de comando. Espera-se que as linhas de comando estejam organizadas de tal modo que no final desta área 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, pode-se 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
```

3.5. 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. Pode-se exibir o resultado armazenado em uma variável ou em um arquivo ou simplesmente finalizar, caso a tarefa já tenha sido toda completada no corpo do programa. É nesta 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
MsgAlert("O fatorial de " + cValToChar(nFator) + ;
        " é " + cValToChar(nResultado))
// Termina a função
Return( )
```

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:

4.2. Numérico

O ADVPL não diferencia valores inteiros de valores com ponto flutuante, portanto podem-se 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 E-308 até 1.7976931348623158 E+308.

4.3. 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).

4.4. Caractere

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

```
"Olá !!!"
```

'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.

4.5. Data

O ADVPL tem um tipo de dados específico para datas. Internamente as variáveis deste 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 através da utilização de funções específicas como, por exemplo, CTOD() que converte uma string para data.

4.6. 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.

4.7. 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.

4.8. 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 necessário mais se referenciar ao seu conteúdo, e sim ao seu nome. O nome de uma variável é um identificador único o qual deve respeitar um 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 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.

5. 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 deve-se 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.

5.1. 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 caso ela não tenha 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. Pois compreenderá o nome da variável escrito de forma incorreta como se fosse a criação de uma nova variável. Isto alterará a lógica do programa, e é um erro muitas vezes difícil de identificar.

5.2. 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:

```
User 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.

5.3. 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:

```
User 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 mantêm seu valor atual quando a execução da função Pai termina. Entretanto, somente pode ser acessada por execuções subseqüentes da função Pai.

5.4. 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 mantêm 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.

5.5. Variáveis de escopo public

Podem-se criar variáveis de escopo public dinamicamente, no código com o identificador PUBLIC. As variáveis deste escopo continuam a existir e mantêm seu valor até o fim da execução da thread (conexão).

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:

```
User Function Pai()  
Private nVar := 10  
  
<comandos>  
Filha()  
<mais comandos>  
  
Return(.T.)
```


Neste 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).

No ambiente ERP Protheus, existe uma convenção adicional a qual deve ser respeitada que variáveis em uso pela aplicação não sejam incorretamente manipuladas. Por esta convenção deve ser adicionado o caractere “_” antes do nome de variáveis PRIVATE e PUBLIC. Maiores informações avaliar o tópico: Boas Práticas de Programação.

Exemplo: Public _cRotina

5.6. 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 este erro, deve-se declarar a variável previamente:

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

Neste exemplo, 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 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

```
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. Em ambos os 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.

5.7. Operações com 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, pode-se mudar o tipo da 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.
```

6. Operadores da linguagem ADVPL

6.1. 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)
```

Neste 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.

6.1.1. Operadores Matemáticos

Os operadores utilizados em ADVPL para cálculos matemáticos são:

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

6.1.2. 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).

6.1.3. 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 !=	Comparação Diferente

6.1.4. Operadores Lógicos

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

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

6.1.5. Operadores de Atribuição

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

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

6.1.6. 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 pode-se 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, pode-se 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.

6.1.7. Atribuição Composta

Os operadores de atribuição composta são uma facilidade da linguagem ADVPL para expressões de cálculo e atribuição. Com eles pode-se economizar digitação:

Operador	Exemplo	Equivalente a
+=	$X += Y$	$X = X + Y$
-=	$X -= Y$	$X = X - Y$
*=	$X *= Y$	$X = X * Y$
/=	$X /= Y$	$X = X / Y$
**= ou ^=	$X **= Y$	$X = X ** Y$
%=	$X \% = Y$	$X = X \% Y$

6.1.8. 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:

++	Incremento Pós ou Pré-fixado
--	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 destas 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.

6.1.9. 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.

6.1.10. 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:

- Operadores de Incremento/Decremento pré-fixado
- Operadores de String
- Operadores Matemáticos
- Operadores Relacionais

Operadores Lógicos
Operadores de Atribuição
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:

Exponenciação
Multiplicação e Divisão
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 dos parênteses mais interno em direção ao mais externo.

7. Operação de Macro Substituição

O operador de macro substituição, simbolizado pelo "&" 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. Pode-se 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 pode-se substituir Y na terceira linha:

```
03 B := &"X + 1"
```

O operador de macro cancela as aspas:

03 B := X + 1

Pode-se perceber que o operador de macro remove as aspas, o que deixa um pedaço de 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 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.

8. Funções de manipulação de variáveis

Além de atribuir, controlar o escopo e 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.

9. Conversões entre tipos de variáveis

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

CTOD()

Sintaxe	CTOD(cData)
Descrição	Realiza a conversão de uma informação do tipo caractere no formato "DD/MM/AAAA", para uma variável do tipo data.
Exemplo	CtoD("01/12/16") + 5 -> 06/12/2016

DTOD()

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

Exemplo	DtoC(Date()) -> data da maquina em formato "DD/MM/AA" caracter.
----------------	---

DTOS()

Sintaxe	DTOS(dData)
Descrição	Realiza a conversão de uma informação do tipo data em um caractere, sendo o resultado no formato "AAAAMMDD".
Exemplo	Converte uma data , no formato string "AAAAMMDD" DtoS (Date()) -> "20161212"

STOD()

Sintaxe	STOD(sData)
Descrição	Realiza a conversão de uma informação do tipo caractere "AAAAMMDD", com conteúdo no formato em data
Exemplo	Converte data em string, no formato Stod (Date()) -> ""AAAAMMDD"

CVALTOCHAR()

Sintaxe	CVALTOCHAR(nValor)
Descrição	Realiza a conversão de uma informação do tipo numérico em uma string, sem a adição de espaços a informação.
Exemplo	cValTochar(50) -> "50"

STR()

Sintaxe	STR(nNumero, nTamanho, nDecimal)
Descrição	Realiza a conversão de uma informação do tipo numérico em uma string, adicionando espaços à direita.
Exemplo	Str(700, 8, 2) -> " 700.00"

STRZERO()

Sintaxe	STRZERO(nNumero, nTamanho, nDecimal)
Descrição	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.
Exemplo	StrZero(60, 10) -> "0000000060"

VAL()

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

Exemplo	Val("1234") -> 1234
----------------	---------------------

9.1. Manipulação de strings

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

ALLTRIM()

Sintaxe	ALLTRIM(cString)
Descrição	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").
Exemplo	AllTrim(" ADVPL ") -> "ADVPL"

ASC()

Sintaxe	ASC(cCaractere)
Descrição	Converte uma informação caractere em seu valor, de acordo com a tabela ASCII. Essa função é útil em expressões que precisam executar cálculos numéricos baseados no valor ASCII de um caractere.
Exemplo	Asc("A") -> 65

AT()

Sintaxe	AT(cProcura, cString, nApos)
Descrição	Retorna a primeira posição de um caractere ou string, dentro de outra string especificada.
Exemplo	At("a", "abcde") -> 1 At("f", "abcde") -> 0

RAT()

Sintaxe	RAT(cCaractere, cString)
Descrição	Retorna a última posição de um caractere ou string, dentro de outra string especificada.
Exemplo	RAt("A", "MARIANO") -> 5

CHR()

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

Exemplo	Chr(65)-> "A" Chr(13)-> Tecla Enter
----------------	--

LEN()

Sintaxe	LEN(cString)
Descrição	Retorna o tamanho da string ou estrutura especificada.
Exemplo	Len("ABCDEF") -> 6 // Len({"Jorge", 34, .T.}) -> 3 Elementos

LOWER()

Sintaxe	LOWER(cString)
Descrição	Retorna uma string com todos os caracteres minúsculos, tendo como base a string passada como parâmetro.
Exemplo	Lower("AbCdE") -> "abcde"

STUFF()

Sintaxe	STUFF(cString, nPosInicial, nExcluir, cAdicao)
Descrição	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.
Exemplo	Stuff("ABCDE", 3, 2, "123") -> "AB123E" substitui "CD" por "123"

SUBSTR()

Sintaxe	SUBSTR(cString, nPosInicial, nCaracteres)
Descrição	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).
Exemplo	Substr("ADVPL", 3, 2) -> "VL"

UPPER()

Sintaxe	UPPER(cString)
Descrição	Retorna uma string com todos os caracteres maiúsculos, tendo como base a string passada como parâmetro.
Exemplo	UPPER ("AbCdE") -> "ABCDE"

PadR()

Sintaxe	Padr(cString)
Descrição	Adiciona caracteres de preenchimento à direita do conteúdo de uma variável.

Exemplo	PadR("ABC", 10, "**") -> "ABC*****"
----------------	-------------------------------------

PadC()

Sintaxe	PadC(cString)
Descrição	Adiciona caracteres de preenchimento centralizando o conteúdo de uma variável.
Exemplo	PadC("ABC", 10, "**") -> " ***ABC*****"

PadL()

Sintaxe	PadL(cString)
Descrição	Adiciona caracteres de preenchimento a esquerda do conteúdo de uma variável.
Exemplo	PadL("ABC", 10, "**") -> "ABC*****"

Replicate()

Sintaxe	Replicate(cString, nCount)
Descrição	Gera uma string repetida a partir de outra.
Exemplo	Replicate("1", 5) -> "*****"

Replicate()

Sintaxe	Replicate(cString, nCount)
Descrição	Gera uma string repetida a partir de outra.
Exemplo	Replicate("1", 5) -> "*****"

StrTran()

Sintaxe	StrTran(cString , cSearch, cReplace, nStart , nCount)
Descrição	Pesquisa e substitui um conjunto de caracteres de uma string.
Exemplo	StrTran("ABCABC", "B", "1") -> A1CA1C

9.2. Manipulação de variáveis numéricas

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

ABS()

Sintaxe	ABS(nValor)
----------------	-------------

Descrição	Retorna um valor absoluto (independente do sinal) com base no valor especificado no parâmetro.
Exemplo	ABS(10 – 90)) -> 80

INT()

Sintaxe	INT(nValor)
Descrição	Retorna a parte inteira de um valor específico no parâmetro.
Exemplo	INT(999.10) -> 999

NOROUND()

Sintaxe	NOROUND(nValor, nCasas)
Descrição	Retorna um valor, truncando a parte decimal do valor especificado no parâmetro de acordo com a quantidade de casas decimais solicitadas.
Exemplo	NOROUND(2.985, 2) -> 2.98

ROUND()

Sintaxe	ROUND(nValor, nCasas)
Descrição	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.
Exemplo	ROUND(2.986, 2) -> 2.99

9.3. Verificação de tipos de variáveis

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

TYPE()

Sintaxe	TYPE("cVariavel")
Descrição	Determina o tipo do conteúdo de uma variável, a qual não foi definida na função em execução.
Exemplo	TYPE("DATE()") -> "D".

VALTYPE()

Sintaxe	VALTYPE(cVariável)
Descrição	Determina o tipo do conteúdo de uma variável, a qual foi definida na função em execução.
Exemplo	VALTYPE("nValor / 2") -> "N"

10. Estruturas Básicas De Programação

O ADVPL suporta várias estruturas de controle que permitem mudar a seqüência de fluxo de execução de um programa. Estas 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.

11. 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, pode-se desejar imprimi-lo quatro vezes. Claro, pode-se simplesmente chamar a função de impressão quatro vezes em seqüência, 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`.

11.1. 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 Variavel := nValorInicial TO nValorFinal [STEP nIncremento]`

Comandos...

`[EXIT]`

`[LOOP]`

`NEXT`

Parâmetros

Variável	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 <code>FOR...NEXT</code> . Se a variável não existir, será criada como uma variável privada .
nValorInicial TO nValorFinal	nValorInicial é o valor inicial para o contador; nValorFinal é o valor final para o contador. Pode-se utilizar valores numéricos literais, variáveis ou expressões, contanto que o resultado seja do tipo de dado numérico.
STEP nIncremento	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 <code>STEP</code> for omitida, o contador será incrementado em 1. Pode-se utilizar valores numéricos literais, variáveis ou

	expressões, contanto que o resultado seja do tipo de dado numérico.
Comandos	Especifica um ou mais instruções de comando ADVPL que serão executadas.
EXIT	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. Pode-se colocar o comando EXIT em qualquer lugar entre o FOR e o NEXT.
LOOP	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. Pode-se colocar o comando LOOP em qualquer lugar entre o FOR e o NEXT.

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
```

Este 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.

11.2. 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 IExpressao

Comandos...

[EXIT]

[LOOP]

ENDDO

Parâmetros:

IExpressao	Especifica uma expressão lógica cujo valor determina quando os comandos entre o WHILE e o ENDDO são executados. Enquanto o resultado de IExpressao for avaliado como verdadeiro (.T.), o conjunto de comandos são executados.
Comandos	Especifica um ou mais instruções de comando ADVPL, que serão executadas enquanto IExpressao for avaliado como verdadeiro (.T.).
EXIT	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. Pode-se colocar o comando EXIT em qualquer lugar entre o WHILE e o ENDDO.
LOOP	Retorna o controle diretamente para a cláusula WHILE sem executar o restante dos comandos entre o LOOP e o ENDDO. A expressão em IExpressao é reavaliada para a decisão se os comandos continuarão 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

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.

11.2.1. Influenciando o fluxo de repetição

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

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

Exemplo:

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

While nltens < nQuantidade
  nltens++
  IF BLOQUEADO(altens [nltens]) // função ilustrativa que
                                //verifica se o produto está bloqueado.
    LOOP
  ENDIF
  IMPRIME() // função ilustrativa que realiza a impressão de um item
            //liberado para uso.
EndDo
```

Caso o produto esteja 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. Desta 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 lWhile
  IF MSGYESNO("Deseja jogar o jogo da forca?")
    JFORCA() // Função ilustrativa que implementa o algoritmo do
              //jogo da forca.
  ELSE
    EXIT
  ENDIF
EndDo
```

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".

12. Estruturas de decisão

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 avaliações lógicas, que são os comandos **IF...ELSE...ENDIF** e o comando **DO CASE...ENDCASE**.

12.1. IF...ELSE...ENDIF

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

Sintaxe:

```
IF IExpressao
Comandos
[ELSE
Comandos...]
ENDIF
Parâmetros
```

LExpressao	<p>Especifica uma expressão lógica que é avaliada. Se IExpressao resultar em verdadeiro (.T.), qualquer comando seguinte ao IF e antecedente ao ELSE ou ENDIF (o que ocorrer primeiro) será executado.</p> <p>Se IExpressao 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 são ignorados. Neste caso, a execução do programa continua com o primeiro comando seguinte ao ENDIF.</p>
Comandos	Conjunto de comandos ADVPL que serão executados dependendo da avaliação da expressão lógica em IExpressao.

Pode-se 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 dVento := CTOD("31/12/16")
```

```
If Date() > dVento
Alert("Vencimento ultrapassado!")
Endif
```

```
Return
```

12.2. IF...ELSEIF...ELSE...ENDIF

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

Sintaxe

```
IF IExpressao1
Comandos
[ELSEIF ExpressaoX
Comandos
[ELSE
Comandos...
ENDIF
```

Parâmetros

IExpressao1	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). 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.
IExpressaoX	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.). Caso a IExpressaoX avaliada resulte em falso (.F.) será avaliada a próxima expressão IExpressaoX, vinculada ao próximo comando ELSEIF, ou caso o mesmo não exista será executada a ação definida para o comando ELSE.
Comandos	Conjunto de comandos ADVPL que serão executados, dependendo da avaliação da expressão lógica em IExpressao.

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 dVenc to := CTOD("31/12/16")
```

```
If Date() > dVenc to
Alert("Vencimento ultrapassado!")
Elseif Date() == dVenc to
Alert("Vencimento na data!")
Else
Alert("Vencimento dentro do prazo!")
Endif
Return()
```

12.3. 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

CASE !Expressao1 Comandos...	<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 seja 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>
OTHERWISE Comandos	<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>

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( NIL )
```

13. Arrays E Blocos De Código

13.1. 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.
```

13.2. 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.

```
aFunc1 := {"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 um pedaço diferente de 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:

```
aFunc2 := {"Maria" , 22, .T.}
aFunc3 := {"Antônio", 42, .F.}
```

Os nomes podem ser impressos assim:

```
MsgAlert(aFunc1[FUNCT_NOME])
MsgAlert(aFunc2[FUNCT_NOME])
MsgAlert(aFunc3[FUNCT_NOME])
```

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

```
aFuncs := {aFunc1, aFunc2, aFunc3}
```


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)

MsgInfo(aFuncs[nCount, FUNCT_NOME])

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.
- **Inicializando arrays:** Algumas vezes o tamanho da matriz é conhecido previamente. Outras vezes o tamanho do array somente será conhecido em tempo de execução.
- **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. 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. 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 imagem 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. 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. A linha 07 demonstra como um valor pode ser atribuído para uma posição existente em uma matriz, especificando o índice entre colchetes.

- **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

```

A linha 02 utiliza os colchetes para criar um array vazio. Apesar de não ter nenhum elemento, seu tipo de dado é array. Na linha 03 a chamada da função array cria uma matriz sem nenhum elemento. 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).

13.3. 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()

Sintaxe	ARRAY(nLinhas, nColunas)
Descrição	A função Array() é utilizada na definição de variáveis de tipo array, como uma opção a sintaxe, utilizando chaves ("{}").
Exemplo	<pre> aVetor := Array(3) -> aVetor[1] aVetor[2] aVetor[3] aMatriz := Array(3,2) -> aMatriz[1,1] aMatriz[1,2] aMatriz[2,1] aMatriz[2,2] aMatriz[3,1] aMatriz[3,2] </pre>

AADD()

Sintaxe	AADD(aArray, xItem)
Descrição	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.

Exemplo

```
Local aVetor := {}
AADD(aVetor, "Maria" ) // Adiciona um elemento no array
AADD(aVetor, "Jose" ) // Adiciona um elemento no array
AADD(aVetor, "Marcio") // Adiciona um elemento no array

Local aMatriz := {}
AADD(aMatriz, { "Maria" ,29,"F"} ) // Adiciona um elemento no array
AADD(aMatriz, { "Jose" ,42,"M"} ) // Adiciona um elemento no array
AADD(aMatriz, { "Marcio" ,53,"M"} ) // Adiciona um elemento no array
```

ACLONE()

Sintaxe	AADD(aArray)
Descrição	A função ACLONE() realiza a cópia dos elementos de um array para outro array integralmente.
Exemplo	altens := ACLONE(aDados)

ADEL()

Sintaxe	ADEL(aArray, nPosição)
Descrição	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, sendo necessário informar um novo valor pela função aSize() ou Ains()
Exemplo	<pre>Local aVetor := {"1","2","3","4","5","6","7","8","9"} ADEL(aVetor,1) // Será removido o primeiro elemento do array. -> aVetor{"2","3","4","5","6","7","8","9",NIL}</pre>

ASIZE()

Sintaxe	ASIZE(aArray, nTamanho)
Descrição	A função ASIZE permite a redefinição da estrutura de um array pré-existente, adicionando ou removendo itens do mesmo.
Exemplo	<pre>Local aVetor := {"1","2","3","4","5","6","7","8","9"} aSize(aVetor,8) // Será reestruturado o array com 8 elementos. -> aVetor{"1","2","3","4","5","6","7","8" }</pre>

AINS()

Sintaxe	AINS(aArray, nPosicao)
Descrição	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.

Exemplo	Local aVetor := {"1","2","3","4","5","6","7","8","9"}
	alns(aVetor,1) // Será reestruturado o array com 8 elementos. ->
	aVetor{Nil,"1","2","3","4","5","6","7","8" }

ASORT()

Sintaxe	ASORT(aArray, nInicio, nItens, bOrdem)
Descrição	A função ASORT() permite que os itens de um array sejam ordenados, a partir de um critério pré-estabelecido.
Exemplo	LOCAL aVetor := { "A", "D", "C", "B" } ASORT(aVetor) -> { "A", "B", "C", "D" } ASORT(aVetor,,, { x, y x > y }) -> { "D", "C", "B", "A" }
	Local aMatriz := { { "Fipe", 9.3 }, { "IPC", 8.7 }, { "DIEESE", 12.3 } } ASORT(aMatriz, , , { x,y x[2] > y[2] }) // Com base na ordenação acima, a ordem fica: DIEESE 12.3 Fipe 9.3 IPC 8.7

ASCAN()

Sintaxe	ASCAN(aArray, bSeek)
Descrição	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.
Exemplo	LOCAL aVetor := { "Java", "AdvPL", "C++" } nPos := ASCAN(aVetor, "AdvPL") -> 2 posição no array
	Local aMatriz := { { "Fipe", 9.3 }, { "IPC", 8.7 }, { "DIEESE", 12.3 } } nPos := ASCAN(aVetor,{ x x[1] == "AdvPL" } -> 3 posição no array

13.4. 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:

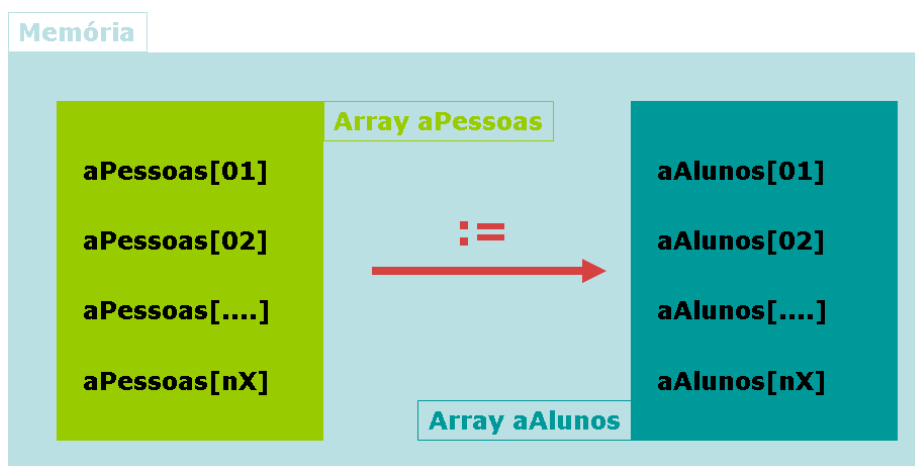
```
nPessoas := 10
nAlunos := nPessoas
```

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 deseja-se obter um array com a mesma estrutura e informações que compõe outro array já existente.

Memória



14. 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.

Funções para manipulação de blocos de código

EVAL()

Sintaxe	EVAL(bBloco, xParam1, xParam2, xParamZ)
Descrição	A função EVAL() é utilizada para 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.
Exemplo:	<pre>nInt := 10 bBloco := { N x:= 10, y:= x*N, z:= y/(x*N)} nValor := EVAL(bBloco, nInt)</pre> <p>O retorno será dado pela avaliação da ultima 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.</p> <p>Desta forma temos:</p> <p>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</p>

DBEVAL()

Sintaxe	Array(bBloco, bFor, bWhile)
Descrição	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.
Exemplo	<pre>dbSelectArea("SX5") dbSetOrder(1) dbGotopt() While !Eof() .And. X5_FILIAL == xFilial("SX5") .And.; X5_TABELA <=cTab nCnt++ dbSkip() EndDo</pre> <p>O mesmo pode ser re-escrito com o uso da função DBEVAL():</p> <pre>dbEval({ x nCnt++ }, { X5_FILIAL==xFilial("SX5") .And. X5_TABELA<= cTab }</pre>

AEVAL()

Sintaxe	AEVAL(aArray, bBloco, nInicio, nFim)
Descrição	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.
Exemplo	<p>Considerando o trecho de código abaixo:</p> <pre>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</pre> <p>O mesmo pode ser re-escrito com o uso da função AEVAL():</p> <pre>aEval(aCampos,{ x SX3->(dbSeek(x)),IIF(Found(), AAdd(aTitulos,AllTrim(SX3->X3_TITULO)))}</pre>

15. Funções

A maior parte das rotinas que queremos escrever em programas são compostas de um conjunto de comandos, rotinas estas que 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 pode-se passar parâmetros, os quais contém os dados e 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, o que permite executar uma função escrevendo:

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

E a função deve estar escrita:

```
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.

Existe um ditado que diz que:

“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 ADVPL, até os programas chamados do 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.

15.1. Tipos e escopos de funções

Em ADVPL podem ser utilizados os seguintes tipos de funções:

- **Function():** Funções ADVPL convencionais, restritas ao desenvolvimento da área de Inteligência Protheus da Microsiga. O interpretador ADVPL distingue nomes de funções do tipo Function() com até dez caracteres. A partir do décimo caracter, apesar do compilador não indicar quaisquer tipos de erros, o interpretador ignorará os demais caracteres.
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”.

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.

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

Neste contexto as Static Functions() são utilizadas para: 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. Redefinir uma função padrão da aplicação, adequando-a as necessidades específicas de uma função principal / aplicação. Proteger funções de uso específico de um arquivo de código fonte / função principal.

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 código 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 (Microsiga 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 a Inteligência Protheus da Microsiga. Na aplicação ERP é comum o uso das Main Functions(), nas seguintes situações:

Definição dos módulos da aplicação ERP: Main Function Sigaadv()

Definição de atualizações e updates: AP710TOMP811()

Atualizações específicas de módulos da aplicação ERP: UpdateATF()

16. 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:

16.1. 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 " + ; ValToChar(nResultado))
```

```
Return( NIL )
```

Avaliando a função CalcFator() descrita anteriormente, podemos verificar que a mesma recebe como parâmetro para 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. Neste 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.

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 realizada nenhum tipo de consistência, em relação aos tipos dos conteúdos, e obrigatoriedade de parâmetros nesta ação. Os parâmetros de uma função são caracterizados como variáveis de escopo LOCAL para efeito de execução. Desta forma os mesmos não devem ser definidos novamente como LOCAL, na área de definição e inicialização de variáveis, pois caso isto ocorra haverá a perda dos valores recebidos pela redefinição das variáveis na função.

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. Neste caso foi informada a variável `nFatorUser`, a qual será definida pelo usuário através 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 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 := nFatorUser`.

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.

Desta forma podemos ter:

```
User Function DirFator()
```

```
Local nFatorUser := GetFator()
```

```
nResultado := CalcFator(nFatorUser)
```

```
...
```

```
Function CalcFator(nFator)
```

```
...
```

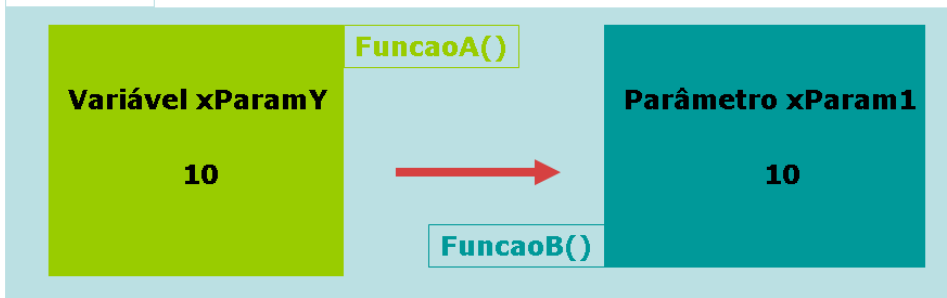
As variáveis `nFatorUser` e `nFator` podem ter nomes diferentes, pois o interpretador fará a atribuição de conteúdo com base na ordem dos parâmetros e não pelo nome das variáveis.

16.2. Passagem de parâmetros por referência

A passagem de parâmetros por referência é uma técnica muito comum nas linguagens de programação, a qual permite que variáveis de escopo LOCAL tenham seu conteúdo manipulado por funções específicas, mantendo o controle destas variáveis restrito à função que as definiu e as 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.

Memória



```

FuncaoA()
xParamY := 10

Funcao(xParamY)

Alert(cValToChar(xParamY))
Return

```

```

FuncaoB(xParam1)

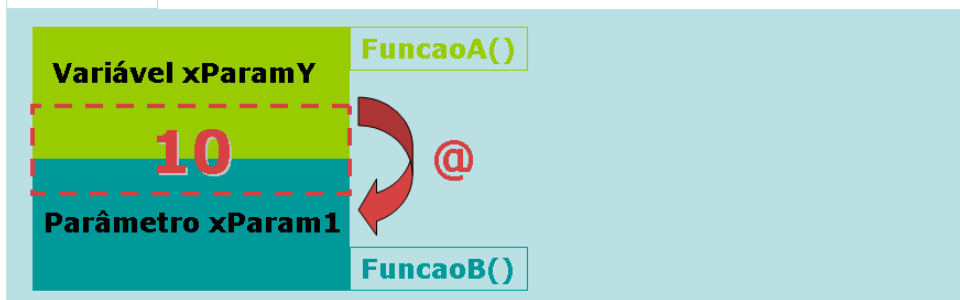
xParam1 := 15

Return

```

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

Memória



```

FuncaoA()
xParamY := 10

Funcao(xParamY)

Alert(cValToChar(xParamY))
Return

```

```

FuncaoB(xParam1)

xParam1 := 15

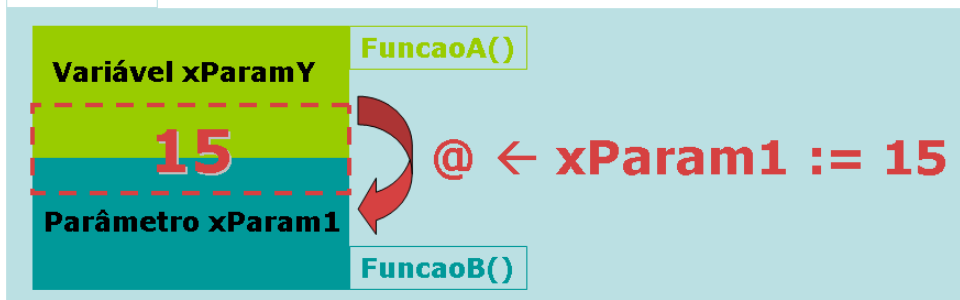
Return

```

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

Desta 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 nesta será visível à função chamadora quando tiver o retorno desta função.

Memória



Conteúdo de xParamY em FuncaoA() → 15

16.2.1. Tratamento de conteúdos padrões para parâmetros de funções

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

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.

Desta forma o uso do identificador DEFAULT permite ao desenvolvedor garantir 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, caso o parâmetro nFator não seja informado na função chamadora, o mesmo terá seu conteúdo definido como 1.

Se este 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
// nFator está como Nulo, portando nCnt é nulo

For nCnt := nFator To 1 Step -1
    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 → U”, pois os tipos das variáveis envolvidos na operação são diferentes: `nCnt` → nulo (U) e `1` → numérico (N).

Pâmetro que possui a opção `DEFAULT`, descrita no fonte, seja informado, a linha de `DEFAULT` não será executada, mantendo desta forma o conteúdo passado pela função chamadora.

18. diretivas De Compilação

O compilador ADVPL possui uma funcionalidade denominada pré-processador, o qual nada mais é do que 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

```

19. 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, os quais devem ser utilizados segundo a aplicação que será desenvolvida, o que permitirá a utilização de recursos adicionais definidos para a linguagem, implementados pela área de Tecnologia da Microsiga. 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 esta funcionalidade de compatibilidade com a sintaxe Clipper, tais como:

DIALOG.CH
FONT.CH
INI.CH
PTMENU.CH
PRINT.CH

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

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

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

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

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

Caso o diretório de includes não esteja informado, ou esteja informado incorretamente, será exibida uma mensagem de erro

20. Diretiva: #DEFINE

A diretiva #DEFINE permite que o desenvolvedor crie novos termos para serem utilizadas 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.

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

Os arquivos de include definidos para os fontes da aplicação ERP contém 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 esta 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.

21. Diretivas: #IFDEF, #ifndef, #else e #endif

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

Através destas 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.

Desta forma, 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, o qual irá gerar um código compilado de acordo com a análise dos parâmetros de ambiente.

Estas diretivas de compilação estão normalmente associadas as 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
```

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.

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

#IFDEF TOP

```
cQuery := "SELECT * FROM "+RETSQNAME("SA1")
dbUseArea(.T., "TOPCONN", TcGenQry(,cQuery), "SA1QRY",.T.,.T.)
```

#ELSE

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

#ENDIF

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

Desta 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 estes bancos deve ser utilizada a sintaxe ADVPL tradicional.

22. 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.

Esta diretiva permite que o desenvolvedor defina para o compilador como uma expressão deverá 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>, <.ICenter.>, <.IRight.>, <.IBorder.>;,
<.IPixel.>, <nClrText>, <nClrBack>, <nWidth>, <nHeight>;,
<.design.>, <.update.>, <.IShaded.>, <.IBox.>, <.IRaised.> )
```

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.

23. Desenvolvimento De Pequenas Customizações

23.1. 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 (DbAccess).

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 ODBC.

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 seqüencial, tendo como base o nome da tabela ao 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 destas operações.

Deve ser considerada a possibilidade de utilização de índices temporários para processos específicos, os quais serão criados em tempo de execução da rotina. Este 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 seqüencial e ascendente que funciona como o endereço base daquela informação.

Este ID, mas conhecido como RECNO ou RECNUMBER é gerado no momento de inclusão do registro na tabela e somente será alterado caso a estrutura dos dados desta tabela sofra 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. Esta situação também torna necessária a recriação de todos os índices vinculados àquela tabela.

Isto 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 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.

Estes 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 seqüencial ao banco de dados.

O campo D_E_L_E_T_ é tratado internamente pela aplicação ERP como um “flag” ou marca de exclusão. Desta 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.

23.2. 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.

DBRLOCK()

Sintaxe	DBRLOCK(xIdentificador)
Descrição	Função de base de dados, que efetua o lock (travamento) do registro identificado pelo parâmetro xIdentificador. Este parâmetro pode ser o Recno() para tabelas em formato ISAM, ou a chave primária para bancos de dados relacionais. 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.

DBCLOSEAREA()

Sintaxe	DbCloseArea()
Descrição	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().

DBCOMMIT()

Sintaxe	DBCOMMIT()
Descrição	Efetua todas as atualizações pendentes na área de trabalho ativa.

DBCOMMITALL()

Sintaxe	DBCOMMITALL()
Descrição	Efetua todas as atualizações pendentes em todas as área de trabalho, em uso pela thread (conexão) ativa.

DBDELETE()

Sintaxe	DbDelete()
Descrição	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()

Sintaxe	DbGoto(nRecno)
Descrição	Move 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()

Sintaxe	DbGoTop()
Descrição	Move o cursor da área de trabalho ativa para o primeiro registro lógico.

DBGOBOTTON()

Sintaxe	DbGoBotton()
Descrição	Move o cursor da área de trabalho ativa para o último registro lógico.

DBRLOCKLIST()

Sintaxe	DBRLOCKLIST()
Descrição	Retorna um array contendo o record number (recno) de todos os registros, travados da área de trabalho ativa.

DBSEEK() E MSSEEK()

Sintaxe	DbSeek(cChave, lSoftSeek, lLast)
Descrição	<p>DbSeek: 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>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 acessar novamente a base de dados para localizar uma informação já utilizada pela thread (conexão) ativa.</p>

DBSKIP()

Sintaxe	DbSkip(nRegistros)
Descrição	Move o cursor do registro posicionado para o próximo (ou anterior, dependendo do parâmetro), em função da ordem ativa para a área de trabalho.

DBSELECTAREA()

Sintaxe	DbSelectArea(nArea cArea)
Descrição	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()

Sintaxe	DbSetFilter(bCondicao, cCondicao)
----------------	-----------------------------------

Descrição	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()

Sintaxe	DbSetOrder(nOrdem)
Descrição	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()

Sintaxe	DbOrderNickName(NickName)
Descrição	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()

Sintaxe	DBUNLOCK()
Descrição	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()

Sintaxe	DBUNLOCKALL()
Descrição	Libera o travamento de todos os registros de todas as áreas de trabalho disponíveis na thread (conexão) ativa.

DBUSEAREA()

Sintaxe	DbUseArea(INovo, cDriver, cArquivo, cAlias, IComparilhado,; ISoLeitura)
Descrição	Define um arquivo de base de dados como uma área de trabalho disponível na aplicação.

MSUNLOCK()

Sintaxe	MsUnLock()
Descrição	Libera o travamento (lock) do registro posicionado, confirmando as atualizações efetuadas neste registro.

RECLOCK()

Sintaxe	RecLock(cAlias,IInclui)
Descrição	Efetua o travamento do registro posicionado na área de trabalho ativa, permitindo a inclusão ou

alteração das informações do mesmo.

RLOCK()

Sintaxe	RLOCK() → ISucesso
Descrição	Efetua o travamento do registro posicionado na área de trabalho ativa.

SELECT()

Sintaxe	Select(cArea)
Descrição	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()

Sintaxe	SoftLock(cAlias)
Descrição	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.

UNLOCK()

Sintaxe	UNLOCK()
Descrição	Libera o travamento do registro posicionado na área de trabalho ativa e confirma as atualizações efetuadas naquele registro.

24. Diferenciação entre variáveis e 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

Esta linha de comando identifica que o valor atribuído à variável cRes deve ser o valor da variável de memória chamada NOME.

cRes := FIELD->NOME

Neste 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 desta tabela.

cRes := SA1->A1_NOME // SA1 – Cadastro de Clientes

Para maiores detalhes sobre abertura de arquivos com atribuição de apelidos, consulte a documentação sobre acesso a banco de dados ou a 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, utilizado pelo Sistema, podem ser visualizadas através do módulo Configurador -> Bases de Dados -> Dicionários -> Bases de Dados.

25. 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.

Os campos, que recebem o tratamento de numeração sequencial pela aplicação ERP, não devem ser considerados como chave primária das tabelas aos quais 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.

26. 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 ser disponível mesmo que naquele momento números maiores já tenham sido oferecidos e utilizados.

Com isso, mesmo que tenhamos vários processos solicitando numerações sequenciais para uma mesma tabela, como por exemplo inclusões simultâneas de pedidos de vendas, teremos para cada pedido um número exclusivos e sem o intervalos e numerações não utilizadas.

26.1. 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()

Sintaxe	GETSXENUM(cAlias, cCampo, cAliasSXE, nOrdem)
Descrição	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()

Sintaxe	CONFIRMSXE(IVerifica)
Descrição	Confirma o número alocado através do último comando GETSXENUM().

ROLLBACKSXE()

Sintaxe	ROLLBACKSXE()
Descrição	Descarta o número fornecido pelo último comando GETSXENUM(), retornando a numeração disponível para outras conexões.

27. Customizações Para A Aplicação Erp

Neste 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

27.1. Customização de campos – Dicionário de Dados

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 nesta premissa, a utilização de validações, no Dicionário de Dados (SX3) ou nas Perguntas de Processos e Relatórios (SX1), deverá focar sempre na utilização de funções ou expressões que resultem em um retorno lógico.

Através do módulo Configurador é possível alterar as propriedades de um campo ou de uma pergunta, de forma a incluir 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:

EXISTCHAV()

Sintaxe	ExistChav(cAlias, cConteudo, nIndice)
Descrição	Retorna .T. ou .F. se o conteúdo especificado existe no alias especificado. Caso exista será exibido um help de Sistema com um aviso informando da ocorrência. 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.

EXISTCPO()

Sintaxe	ExistCpo(cAlias, cConteudo, nIndice)
Descrição	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.

NAOVAZIO()

Sintaxe	NaoVazio()
Descrição	Retorna .T. ou .F. se o conteúdo do campo posicionado no momento não está vazio.

NEGATIVO()

Sintaxe	Negativo()
Descrição	Retorna .T. ou .F. se o conteúdo digitado para o campo é negativo.

PERTENCE()

Sintaxe	Pertence(cString)
Descrição	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().

POSITIVO()

Sintaxe	Positivo()
Descrição	Retorna .T. ou .F. se o conteúdo digitado para o campo é positivo.

TEXTO()

Sintaxe	Texto()
Descrição	Retorna .T. ou .F. se o conteúdo digitado para o campo contém apenas números ou alfanuméricos.

VAZIO()

Sintaxe	Vazio()
Descrição	Retorna .T. ou .F. se o conteúdo do campo posicionado no momento está vazio.

27.2. Pictures de formação disponíveis**Dicionário de Dados (SX3) e GET**

Funções	
Conteúdo	Funcionalidade
A	Permite apenas caracteres alfabéticos.
C	Exibe CR depois de números positivos.
E	Exibe numérico com o ponto e vírgula invertidos (formato Europeu).
R	Insere caracteres diferentes dos caracteres de template na exibição, mas não os insere na variável do GET.
S<n>	Permite rolamento horizontal do texto dentro do GET, <n> é um número inteiro que identifica o tamanho da região.
X	Exibe DB depois de números negativos.
Z	Exibe zeros como brancos.
(Exibe números negativos entre parênteses com os espaços em branco iniciais.
)	Exibe números negativos entre parênteses sem os espaços em branco iniciais.
!	Converte caracteres alfabéticos para maiúsculo.

Templates	
Conteúdo	Funcionalidade
X	Permite qualquer caractere.
9	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: @!

SAY e PSAY

Funções	
Conteúdo	Funcionalidade
C	Exibe CR depois de números positivos.
E	Exibe numérico com o ponto e a vírgula invertidos (formato Europeu).
R	Inserir caracteres diferentes dos caracteres de template.
X	Exibe DB depois de números negativos.
Z	Exibe zeros como brancos.
(Envolve números negativos entre parênteses.
!	Converte todos os caracteres alfabéticos para maiúsculo.

Templates	
Conteúdo	Funcionalidade
X	Exibe dígitos para qualquer tipo de dado.
9	Exibe dígitos para qualquer tipo de dado.
#	Exibe dígitos para qualquer tipo de dado.
!	Converte caracteres alfabéticos para maiúsculo.
*	Exibe asterisco no lugar de espaços em branco e iniciais em números.
.	Exibe a posição do 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

27.3. 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 digitação de informações. 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:

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 destes 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 este campo que determina qual tipo do retorno será considerado válido para o gatilho.

27.4. Customização de parâmetros – Configurador

Os parâmetros de sistema utilizados pela aplicação ERP e definidos através do módulo configurador possuem as seguintes características fundamentais:

Tipo do parâmetro: de forma similar a uma variável, um parâmetro terá um tipo de conteúdo pré-definido em seu cadastro.

Esta informação é utilizada pelas funções da aplicação ERP, na interpretação do conteúdo do parâmetro e retorno desta 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. Estes parâmetros macro executáveis tem como única característica em comum 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 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, no qual 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.

Funções para manipulação de parâmetros

A aplicação ERP disponibiliza as seguintes funções para consulta e atualização de parâmetros:

GETMV()

Sintaxe	GETMV(cParametro)
Descrição	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.

GETNEWPAR()

Sintaxe	GETNEWPAR(cParametro, cPadrao, cFilial)
Descrição	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 SuperGetMV() pois considera que o parâmetro pode não existir na versão atual do Sistema, e por consequência não será exibida a mensagem de help.

PUTMV()

Sintaxe	PUTMV(cParametro, cConteudo)
Descrição	Atualiza o conteúdo do parâmetro especificado no arquivo SX6, de acordo com as parametrizações informadas.

SUPERGETMV()

Sintaxe	SUPERGETMV(cParametro , lHelp , cPadrao , cFilial)
Descrição	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.

27.4.1. 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 deste parâmetro, é expressamente proibido o uso de funções em parâmetros para manipular informações da base de dados do Sistema.

Caso haja a 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 desta restrição é simples:

As rotinas da aplicação ERP não protegem a consulta de conteúdos de parâmetros, quanto a gravações realizadas dentro ou fora de uma transação.

Desta forma, qualquer alteração na base realizada por uma rotina, configurada em um parâmetro, pode ocasionar a perda da integridade das informações do Sistema.

28. Pontos de Entrada – Conceitos, Premissas e Regras

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.

28.1. Premissas e Regras

Um ponto de entrada não deve ser utilizado para outras finalidades senão para as quais o mesmo 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 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().

- **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()).

Retorno: Array contendo {Alias(),IndexOrd(),Recno()}

- **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)

Retorno: aArea Array contendo {cAlias, nOrdem, nRecno}, normalmente gerado pelo uso da função GetArea().

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.

29. 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 a 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.

29.1. 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, segue abaixo as diferentes definições para os 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 as 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:

BUTTON()

Sintaxe	@ nLinha,nColuna BUTTON cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef ACTION AÇÃO
Descrição	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()

Sintaxe	DEFINE MSDIALOG oObjetoDLG TITLE cTitulo FROM nLinIni,nColIni TO nLiFim,nColFim OF oObjetoRef UNIDADE
Descrição	Define o componente MSDIALOG(), o qual é utilizado como base para os demais componentes da interface visual, pois um componente MSDIALOG() é uma janela da aplicação.

MSGET()

Sintaxe	@ nLinha, nColuna MSGET VARIABEL SIZE nLargura,nAltura UNIDADE OF oObjetoRef F3 cF3 VALID VALID WHEN WHEN PICTURE cPicture
Descrição	Define o componente visual MSGET, o qual é utilizado para captura de informações digitáveis na tela da interface.

SAY()

Sintaxe	@ nLinha, nColuna SAY cTexto SIZE nLargura,nAltura UNIDADE OF oObjetoRef
Descrição	Define o componente visual SAY, o qual é utilizado para exibição de textos em uma tela de interface.

SBUTTON()

Sintaxe	DEFINE SBUTTON FROM nLinha, nColuna TYPE N ACTION AÇÃO STATUS OF oObjetoRet
Descrição	Define o componente visual SButton, o qual 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-definido.

29.2. Interface visual completa

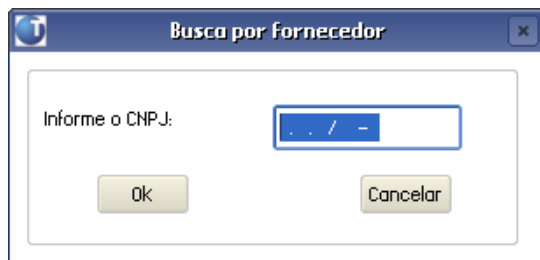
Abaixo segue um código completo de interface, utilizando todos os elementos da interface visual descritos anteriormente:

```

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

```

O código demonstrado anteriormente é utilizado nos exercícios de fixação deste material e deverá produzir a seguinte interface:



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(): O AxCadastro() é uma funcionalidade de cadastro simples, com poucas opções de customização, a qual é composta de:
- mBrowse padrão para visualização das informações da base de dados, de acordo com as configurações do SX3 – Dicionário de Dados (campo browse).

Ambos os modelos utilizam como premissa que a estrutura da tabela a ser utilizada esteja definida no dicionário de dados do sistema (SX3).

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:

cAlias	Alias padrão do sistema para utilização, o qual deve estar definido no dicionário de dados – SX3.
cTitulo	Título da Janela.
cVldExc	Validação para Exclusão.
VldAlt	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
```

29.3. 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 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:

nLin1, nCol1, nLin2, nCol2	Coordenadas dos cantos aonde o browse será exibido.
cAlias	Alias padrão do sistema para utilização, o qual deve estar definido no dicionário de dados.

Variáveis private adicionais

aRotina	<p>Array contendo as funções que serão executadas pela Mbrowse. 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>
----------------	---

cCadastro

Título do browse que será exibido.

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()

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. Desta 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

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( NIL )
```

```
//-----
User Function BInclui(cAlias, nReg, nOpc)
```

```
Local cTudoOk := "(Alert('OK'),.T.)"
AxInclui(cAlias,nReg,nOpc,,,cTudoOk)
```

```
Return()
```

29.4. AxFunctions()

Conforme 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. Estas 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:

AXALTERA()

Sintaxe	AxAltera(cAlias, nReg, nOpc, aAcho, cFunc, aCpos, cTudoOk, IF3,; cTransact, aButtons, aParam, aAuto, IVirtual, IMaximized)
Descrição	Função de alteração padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

AXDELETA()

Sintaxe	AXDELETA(cAlias, nReg, nOpc, cTransact, aCpos, aButtons, aParam,; aAuto, IMaximized)
Descrição	Função de exclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

AXINCLUI()

Sintaxe	AxInclui(cAlias, nReg, nOpc, aAcho, cFunc, aCpos, cTudoOk, IF3,; cTransact, aButtons, aParam, aAuto, IVirtual, IMaximized)
Descrição	Função de inclusão padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

AXPESQUI()

Sintaxe	AXPESQUI()
Descrição	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.

AXVISUAL()

Sintaxe	AXVISUAL(cAlias, nReg, nOpc, aAcho, nColMens, cMensagem, cFunc, aButtons, lMaximized)
Descrição	Função de visualização padrão das informações de um registro, no formato Enchoice, conforme demonstrado no tópico sobre a interface AxCadastro().

Exercício

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 nesta tabela relacionada.

Exercício

Desenvolver uma validação para um campo caractere existente na base, para que seja avaliado se aquele código já existe cadastrado, e caso positivo exiba uma mensagem de aviso alertando desta ocorrência.

Exercício

Desenvolver um gatilho que retorne uma descrição complementar para um campo vinculado ao campo código utilizado nos exercícios anteriores.

30. apêndices**30.1. Boas Práticas De Programação****30.1.1. Utilização De Identação**

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
```

30.2. Capitulação De Palavras-Chaves

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
```

Para funções de manipulação de dados que comecem por "db", a capitulação só será efetuada após o "db":

- dbSeek()
- dbSelectArea()

30.2.1. 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...
```

1.1.1. Utilização Da Notação Húngara

A notação húngara consiste em adicionar os prefixos aos nomes de variáveis, de modo a facilmente se identificar seu tipo. Isto facilita na 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 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 desta notação para nomear variáveis.

Notação	Tipo de dado	Exemplo
a	Array	aValores
b	Bloco de código	bSeek
c	Caracter	cNome
d	Data	dDataBase
l	Lógico	lContinua
n	Numérico	nValor
o	Objeto	oMainWindow
x	Indefinido	xConteudo

1.1.2. 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			

Palavras reservadas não podem ser utilizadas para variáveis, procedimentos ou funções;

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.

Identificadores de escopo PRIVATE ou PUBLIC utilizados em aplicações específicas, desenvolvidas por ou para clientes, devem ter sua identificação iniciada por um caractere “_”.

31. 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.

1.2. Conversão entre tipos de dados

CTOD()

Realiza a conversão de uma informação do tipo character no formato "DD/MM/AAAA", para uma variável do tipo data.

Sintaxe: CTOD(cData)

Parâmetros:

cData Caractere no formato "DD/MM/AAAA".

Exemplo:

```
cData := "31/12/2016"
```

```
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:

nValor 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:

dData 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:

dData 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:

sData 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:

nValor 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:

nValor Valor numérico que será convertido para caractere.

nTamanho 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 caracter em numérica.

Sintaxe: VAL(cValor)

Parâmetros:

cValor 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)

```

31.1. 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:

"cVariavel" 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")
EndIf

```

VALTYPE()

Determina o tipo do conteúdo de uma variável, a qual não foi definida na função em execução.

Sintaxe: VALTYPE(cVariavel)

Parâmetros:

cVariavel 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 errado
    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 MGET 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

```

31.2. 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:

nLinhas Determina o número de linhas com as quais o array será criado.

nColunas 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.
```

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:

aArray Array pré-existente no qual será adicionado o item definido em xItem.

xItem Item que será adicionado ao array.

Exemplo:

```
aDados := {} // Define que a variável aDados é um array, sem especificar suas dimensões.
```

```
altem := {} // Define que a variável altem é um array, sem especificar suas dimensões.
```

```
AADD(altem, cVariavel1) // Adiciona um elemento no array altem de acordo com o cVariavel1
```

```
AADD(altem, cVariavel2) // Adiciona um elemento no array altem de acordo com o cVariavel2
```

```
AADD(altem, cVariavel3) // Adiciona um elemento no array altem de acordo com o cVariavel3
```

```
// Neste ponto o array aItem possui 03 elementos os quais podem ser acessados com:
```

```
// altem[1] -> corresponde ao conteúdo de cVariavel1
```

```
// altem[2] -> corresponde ao conteúdo de cVariavel2
```

```
// altem[3] -> corresponde ao conteúdo de cVariavel3
```

```
AADD(aDados, altem) // Adiciona no array aDados o conteúdo do array altem
```

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, altem)
```

```
AADD(aDados, altem)
```

```
// 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:

aArray Array pré-existente que terá seu conteúdo copiado para o array especificado.

Exemplo:

Utilizando o array aDados, utilizado no exemplo da função AADD()

```
altens := ACLONE(aDados).
```

Neste ponto, o array altens 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, através 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:

aArray Array do qual deseja-se remover uma determinada posição.

nPosição Posição do array que será removida.

Exemplo:

Utilizando o array altens do exemplo da função ACLONE() temos:

```
ADEL(altens,1) // Será removido o primeiro elemento do array altens.
```

Neste ponto, o array altens continua com 03 elementos, aonde:

altens[1] -> antigo altens[2], o qual foi reordenado como efeito da exclusão do item 1.

altens[2] -> antigo altens[3], o qual foi reordenado como efeito da exclusão do item 1.

altens[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:

aArray	Array pré-existente que terá sua estrutura redimensionada.
nTamanho	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 altens, o qual teve um elemento excluído pelo uso da função ADEL()

ASIZE(altens, Len(altens)-1)).

Neste ponto o array altens possui 02 elementos, ambos com conteúdos válidos.

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:

aArray	Array pré-existente que terá seu conteúdo ordenado através de um critério estabelecido.
nInicio	Posição inicial do array para início da ordenação. Caso não seja informado, o array será ordenado a partir de seu primeiro elemento.
nItens	Quantos itens, a partir da posição inicial deverão ser ordenados. Caso não seja informado, serão ordenados todos os elementos do array.
bOrdem	Bloco de código que permite a definição do critério de ordenação do array. Caso bOrdem não seja informado, será utilizado o critério ascendente.

Um bloco de código é basicamente uma função escrita em linha. Desta 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, pode-se definir um bloco de código com a estrutura abaixo:

bBloco := { [xPar1, xPar2, ... xParZ] Ação1, Ação2, AçãoZ } ,

- || -> 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.

Exemplo 01 – Ordenação ascendente

```
aAlunos := { "Mauren", "Soraia", "Andréia" }
```

```
aSort(aAlunos)
```

Neste 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, pode-se substituir "x" por cNomeAtu.

A variável "y" receberá o conteúdo do próximo item a ser avaliado, e usando a mesma analogia de "x", pode-se substituir "y" por cNomeProx. Desta forma o bloco de código bOrdem pode ser re-escrito como:

```
bOrdem := { |cNomeAtu, cNomeProx| cNomeAtu > cNomeProx }
```

```
aSort(aAlunos,,bOrdem)
```

Neste 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:

aArray Array pré-existente no qual desejasse identificar a posição que contém a informação pesquisada.

bSeek Bloco de código que configura os parâmetros da busca a ser realizada.

Exemplo:

```
aAlunos := { "Márcio", "Denis", "Arnaldo", "Patrícia" }
```

```
bSeek := { |x| x == "Denis" }
```

```
nPosAluno := aScan(aAlunos,bSeek) // retorno esperado → 2
```

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" }
```

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. Serão 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 desta forma da função AADD(), a qual sempre insere um novo elemento ao final da estrutura já existente.

Sintaxe: AINS(aArray, nPosicao)

Parâmetros:

aArray Array pré-existente no qual desejasse inserir um novo elemento.

nPosicao Posição na qual o novo elemento será inserido.

Exemplo:

```
aAlunos := {"Edson", "Robson", "Renato", "Tatiana"}
```

```
AINS(aAlunos,3)
```

Neste ponto o array aAlunos terá o seguinte conteúdo:

```
{"Edson", "Robson", nulo, "Renato", "Tatiana"}
```

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.

31.3. 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:

bBloco Bloco de código que será interpretado.

xParamZ 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)

$Z \rightarrow$ resultado da divisão de Y pela multiplicação de X por N ($100 / 100$) $\rightarrow 1$

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:

bBloco	Bloco de código principal, contendo as expressões que serão avaliadas para cada registro do alias ativo.
bFor	Condição para continuação da análise dos registros, com o efeito de uma estrutura For ... Next.
bWhile	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_DESCR)})
    dbSkip()
End
```

O mesmo pode ser re-escrito com o uso da função DBEVAL():

```
dbEval({|aAdd(aTabela,{X5_CHAVE,Capital(X5_DESCR)})|}, ;
{|X5_TABELA==cTabela})
```

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:

aArray	Array que será avaliado na execução da função.
bBloco	Bloco de código principal, contendo as expressões que serão avaliadas para cada elemento do array informado.
nInicio	Elemento inicial do array, a partir do qual serão avaliados os blocos de código.
nFim	Elemento final do array, até o qual serão avaliados os blocos de código.

Exemplo 01:

```
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)))})
```

31.4. 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:

cString	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:

cCaractere:	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:

cCaractere	Caractere ou string que se deseja verificar.
cString	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:

nASCII 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:

cString 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:

cString 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:

cCaractere Caractere ou string que se deseja verificar.

cString 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:

cString String que se deseja verificar.

nPosInicial Posição inicial da informação que será extraída da string.

nCaracteres 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:

cString String que será convertida para caracteres maiúsculos.

Exemplo:

cTexto := "advpl"

MSGINFO("Texto:" + LOWER(cTexto))

31.5. Manipulação de variáveis numéricas

ABS()

Retorna um valor absoluto (independente do sinal), com base no valor especificado no parâmetro.

Sintaxe: ABS(nValor)

Parâmetros:

nValor 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:

nValor 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:

nValor Valor que será avaliado.

nCasas 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 quantidades de casas decimais solicitadas, utilizando o critério matemático.

Sintaxe: ROUND(nValor, nCasas)

Parâmetros

nValor	Valor que será avaliado.
nCasas	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) → 2.99

31.6. 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:

cArea 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:

nRecno 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
```

```
DBGOTOP()
```

Move o cursor da área de trabalho ativa para o primeiro registro lógico.

Sintaxe: DbGoTop()

Parâmetros

Nenhum

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)
```

```
EndDo
```

```
MsgInfo("Existem :"+STRZERO(nCount,6)+ "registros no intervalo").
```

```
// Retorno esperado :000001, pois o DbGoTop posiciona no primeiro
```

```
// registro.
```

DBGOBOTTOM()

Move o cursor da área de trabalho ativa para o último registro lógico.

Sintaxe: DbGoBotton()

Parâmetros:

Nenhum

Exemplo:

```
nCount := 0 // Variável para verificar quantos registros há no intervalo
```

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

```
DbSetOrder(1) // A1_FILIAL + A1_COD + A1_LOJA
```

```
DbGoBottom()
```

```
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 DbGoBottom posiciona no último registro.
```

DBSELECTAREA()

Define a área de trabalho especificada como sendo a área ativa. Todas as operações subseqüentes que fizerem referência a uma área de trabalho para utilização, a menos que a área desejada seja informada explicitamente.

Sintaxe: DbSelectArea(nArea | cArea)

Parâmetros:

nArea 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.

cArea 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 utilizada 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:

nOrdem	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:

NickName 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:

cChave 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.

ISoftSeek 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.

ILast 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:

nRegistros 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
DbGoBottom() // 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:

bCondicao Bloco que expressa a condição de filtro em forma executável.

cCondicao 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)
DbGoBottom()
```

```
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)
DbGoBottom()
```

```
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

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:

cAlias Alias que identifica a área de trabalho que será manipulada.

lInclui 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 - 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
```

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.)
```




```
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, IComparilhado,;
ISoLeitura)

Parâmetros

INovo	Parâmetro opcional que permite, caso o cAlias especificado já esteja em uso, ser fechado antes da abertura do arquivo da base de dados.
cDriver	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", "CTREECDX", "DBFCDXAX", "TOPCONN".
cArquivo	Nome do arquivo de base de dados que será aberto com o alias especificado.
cAlias	Alias para referência do arquivos de base de dados pela aplicação.
IComparilhado	Se o arquivo poderá ser utilizado por outras conexões.
ISoLeitura	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:

Nenhum

Exemplo:


```
DbUserArea(.T., "DBFCDX", "\SA1010.DBF", "SA1DBF", .T., .F.)
DbSelectArea("SA1DBF")
MsgInfo("A tabela SA1010.DBF possui:" + STRZERO(RecCount(),6) + " registros.")
DbCloseArea()
```

32. Funções visuais para aplicações

ALERT()

Sintaxe: Alert(cTexto)

Parâmetros


cTexto	Texto a ser exibido.
	

AVISO()

Sintaxe: AVISO(cTitulo, cTexto, aBotoes, nTamanho)

Retorno: numérico indicando o botão selecionado.


Parâmetros

cTitulo	Título da janela.
cTexto	Texto do aviso.
aBotoes	Array simples (vetor) com os botões de opção.
nTamanho	Tamanho (1,2 ou 3).
	

FORMBATCH()

Sintaxe: FORMBATCH(cTitulo, aTexto, aBotoes, bValid, nAltura, nLargura)

Parâmetros

cTitulo	Título da janela.
aTexto	Array simples (vetor) contendo cada uma das linhas de texto que serão exibidas no corpo da tela.
aBotoes	Array com os botões do tipo SBUTTON(), com a seguinte estrutura: {nTipo, lEnable, { Ação() }}
bValid	(opcional) Bloco de validação do janela.
nAltura	(opcional) Altura em pixels da janela.
nLargura	(opcional) Largura em pixels da janela.
	

MSGFUNCTIONS()


Sintaxe: MSGALERT(cTexto, cTitulo)



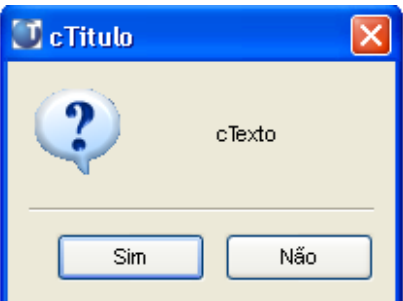
Sintaxe: MSGINFO(cTexto, cTitulo)

Sintaxe: MSGSTOP(cTexto, cTitulo)

Sintaxe: MSGYESNO(cTexto, cTitulo)

Parâmetros:

cTexto	Texto a ser exibido como mensagem.
cTitulo	Título da janela de mensagem.
MSGALERT	

MSGINFO	
MSGSTOP	
MSGYESNO	

REFERÊNCIAS BIBLIOGRÁFICAS

Gestão empresarial com ERP
Ernesto Haberkorn, 2006

Apostila de Treinamento – ADVPL
Educação corporativa

Apostila de Treinamento – Introdução á programação
Educação corporativa

Apostila de Treinamento – Boas Práticas de Programação
Inteligência Protheus e Fábrica de Software

Materiais diversos de colaboradores Microsiga Protheus
Colaboradores Totvs