```
11    ? "Dentro da rotina inicializar"
12
13 RETURN
```

Note que a única diferença foi passar a rotina estática INICIALIZAR() como parâmetro, através de um bloco de código, para a rotina CONFIG01() que está em outro arquivo.

Agora altere o arquivo chamado bloco2.prg e dentro dele digite o seguinte:

Listing 18.14: bloco2.prg

```
1
2 PROCEDURE Config01( bBloco )
3
4    EVAL( bBloco )
5
6 RETURN
```

Dentro da rotina config, você deve receber o parâmetro (o bloco que é a chamada a rotina INICIALIZAR()) e executar o bloco através da função EVAL().

.:Resultado:.

```
Dentro da rotina inicializar
```

Você só precisa modificar a rotina que irá chamar (no nosso caso MAIN()) e a rotina que será chamada (CONFIG01()). A rotina que será executada (INICIALIZAR()) não precisa sofrer alteração alguma.

18.4.6 Blocos de código extendidos

O conceito de bloco de código, que já era poderoso na linguagem Clipper, foi estendido na linguagem Harbour. Esse tipo de bloco está representado através do exemplo a seguir:

Listing 18.15: Bloco de código extendido

```
1  FUNCTION Main
2  LOCAL b1, b2
3
4     b1 := {||
5               ? "POSSO OCUPAR"
6               ? "MUITAS LINHAS"
7               ? "E CONTER COMANDOS"
8               ? "NÄO PRECISO DE PONTO E ViRGULA PARA SEPARAR AS
                   LINHAS"
9               RETURN NIL
10           }
```

```
11
12    ? EVAL( b1 )
13
14
15 RETURN NIL
```

.:Resultado:.

```
POSSO OCUPAR
MUITAS LINHAS
E CONTER COMANDOS
NÃO PRECISO DE PONTO E ViRGULA PARA SEPARAR AS LINHAS
NIL
```

Agora, se você acompanhou tudo o que foi dito sobre blocos de código, você poderia perguntar: “Nas subseções anteriores foi dito que o bloco de código só aceita expressões compiladas na forma de funções e operadores, mas agora o bloco passa a aceitar comandos. Algo está errado na sua explicação!”.

A resposta a essa observação não é muito difícil. Temos **dois** formatos para a escrita de um bloco de código. A primeira delas é o formato original, que foi criado pela equipe do Clipper, e a segunda forma é o bloco extendido, que estamos vendo agora. Apesar de sintaticamente iguais, o bloco de código extendido é um outro tipo de bloco de código.

O que difere um bloco de código padrão Clipper de um bloco de código extendido do Harbour ?

A resposta é simples: um ENTER (uma linha em branco) no início do bloco. Acompanhe o exemplo a seguir:

Listing 18.16: Bloco de código extendido (errado)

```
1  FUNCTION Main
2  LOCAL b1
3
4     b1 := {||  SET DATE BRITISH }
5
6     EVAL( b1 )
7
8     ? DATE()
9
10 RETURN NIL
```

Esse código não vai sequer compilar, pois ele é um bloco de código no formato tradicional. Lembre-se: blocos de código nesse formato não aceitam comandos no seu interior.

.:Resultado:.

```
Compiling 'bloco05.prg'...
```

```
bloco05.prg(4) Error E0030  Syntax error "syntax error at 'DATE'"
```

Já esse código irá compilar e funcionar, pois usa um bloco de código estendido.

Listing 18.17: Bloco de código extendido (certo)

```
1  FUNCTION Main
2  LOCAL b1
3
4     b1 := {||
5              SET DATE BRITISH
6           }
7
8     EVAL( b1 )
9
10    ? DATE()
11
12 RETURN NIL
```

Funciona perfeitamente

.:Resultado:.

```

29/12/16
```

Note também um fato estranho: durante a fase de compilação o compilador emitiu o seguinte aviso :

.:Resultado:.

```
bloco06.prg(6) Warning W0007  Function '{||...}' does not end with
   RETURN statement
```

Isso acontece porque o bloco de código estendido comporta-se como uma função, que, como nós vimos no capítulo sobre funções, exige um retorno.

18.4.6.1 Variáveis locais ao bloco

Outro fato interessante sobre os blocos de código estendidos é a possibilidade de termos uma declaração LOCAL dentro deles. Isso irá gerar uma variável que somente será visível dentro do bloco e não irá interferir no ambiente externo.

Listing 18.18: Variáveis LOCAIS ao bloco

```
1  #define CONST_UNIVERSAL 8600
2
3  FUNCTION Main
```

```
4  LOCAL bChave, nVariavel
5  LOCAL nChaveFinal
6
7     bChave := {| nVar |
8                  LOCAL nToken := SECONDS()
9                  LOCAL nResult
10
11                    nResult := nVar * nToken + CONST_UNIVERSAL
12
13
14                 RETURN nResult
15              }
16
17    INPUT "Informe um número para a geração da chave : " TO
         nVariavel
18    nChaveFinal := EVAL( bChave , nVariavel )
19
20    ? "O valor da chave é " , nChaveFinal
21
22 RETURN NIL
```

.:Resultado:.

```
Informe um número para a geração da chave : 567
O valor da chave é    22019636.96
```

Dica 168

A programação funcional é uma nova forma de programar que está ganhando espaço entre as modernas linguagens de programação. Caso queira aprender mais sobre esse novo paradigma da programação, consulte: https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional

18.5 Macros x Blocos de código

Blocos de código são superiores as macros, mas existe um caso onde as macros devem ser usadas: na criação de uma expressão em tempo de execução. Por exemplo:

Listing 18.19: Macros x Blocos

```
1  FUNCTION Main
2  LOCAL cExpressao
3
4     ACCEPT "Informe a expressão a ser executada : " TO cExpressao
5     ? &cExpressao
```