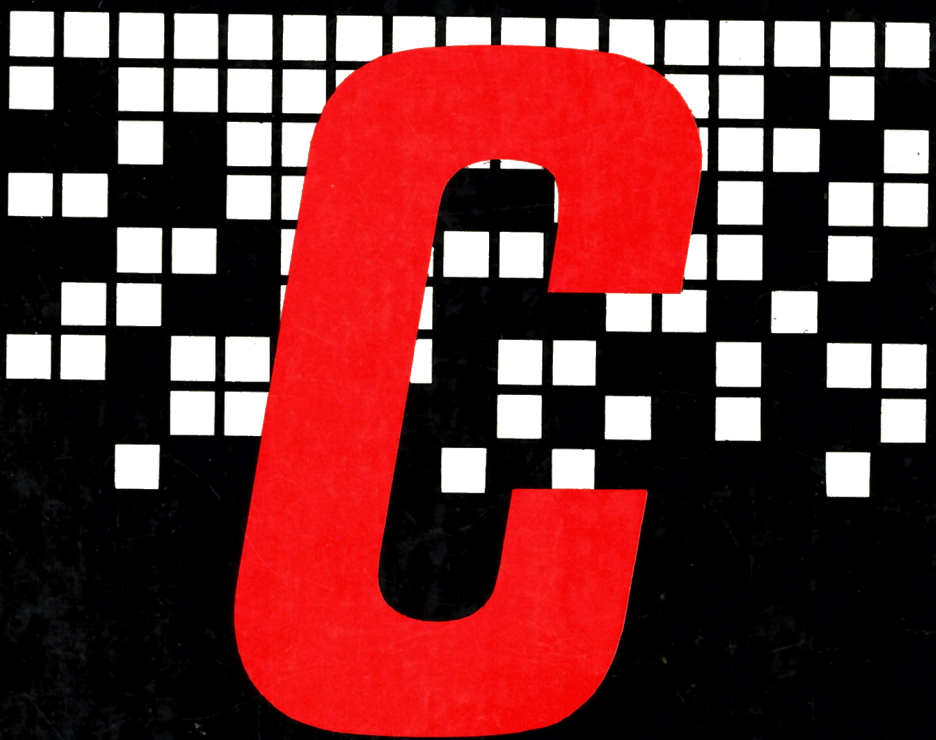
INTEGRANDO CLIPPER COM LINGUAGEM C



Roberto Carlos Mayer

Roberto Carlos Mayer

INTEGRANDO CLIPPER COM LINGUAGEM C

INTEGRANDO CLIPPER COM LINGUAGEM C

INTEGRANDO CLIPPER COM LINGUAGEM C

Roberto Carlos Mayer

MAKRON *Books* do Brasil Editora Ltda.
Editora McGraw-Hill Ltda.
São Paulo
Rua Tabapuã, 1105, Itaim-Bibi
CEP 04533
(011) 829-8604 e (011) 820-8528

Rio de Janeiro • *Lisboa* • *Porto* • *Bogotá* • *Buenos Aires* • *Guatemala* • *Madrid* • *México* • *New York* • *Panamá* • *San Juan* • *Santiago*

Auckland • Hamburg • Kuala Lumpur • London • Milan • Montreal • New Delhi • Paris • Singapore • Sydney • Tokyo • Toronto

Integrando Clipper com Linguagem C

EDITOR: MILTON MIRA DE ASSUMPÇÃO FILHO

Supervisora de Produção Editorial: Daisy Pereira Daniel
Supervisor de Produção Gráfica: José Rodrigues
Capa: layout: Douglas Lucas

Editoração Eletrônica: ALGORITMO SISTEMAS LTDA.

Dados de Catalogação na Publicação (CIP) Internacional
(Câmara Brasileira do Livro, SP, Brasil)

Mayer, Roberto Carlos.
Integrando CLIPPER com linguagem C / Roberto Carlos Mayer. -- São Paulo : McGraw-Hill, 1990.
1. C (Linguagem de programação para computadores) 2. CLIPPER (Linguagem de programação para computadores) 3. CLIPPER (Programa de computador) I. Título.

CDD-001.6425
89-2133 -001.6424

Índices para catálogo sistemático:

1. C : Linguagem de programação : Computadores : Processamento de dados 001.6424
2. CLIPPER : Computadores : Programas : Processamento de dados 001.6425
3. CLIPPER : Liguagem de programação : Computadores : Processamento de dados 001.6424

Sumário

Prefácio

Este livro é destinado a todos os usuários do Clipper que desejam ampliar os recursos a sua disposição através do uso da linguagem C. Ao longo dos cinco capítulos do texto, é descrito por completo o processo de integração de rotinas escritas em linguagem C com programas escritos em Clipper. Esta descrição inclui muitas dicas não disponíveis no manual do Clipper, como também diversas rotinas em C já prontas, a título de exemplos. Estas podem ser integradas de imediato com programas escritos em Clipper.

Todo o primeiro capítulo apresenta uma análise comparativa entre a linguagem C e o Clipper, incluindo uma comparação dos tipos de variáveis, mecanismos de passagem de parâmetros e dos recursos para a estruturação de programas grandes existentes em ambas as linguagens. O capítulo é encerrado por uma análise das conseqüências destas diferenças sobre as possibilidades de integração.

O capítulo 2 explica como escrever rotinas em C que possam ser chamadas posteriormente desde programas escritos em Clipper. Para facilitar o entendimento, desenvolvemos uma rotina exemplo paralelamente às explicações.

No terceiro capítulo é descrito em detalhes o processo de compilação que deve ser seguido para produzir programas executáveis a partir de programas escritos em Clipper, que contenham chamadas a

rotinas escritas em C. Este processo é ilustrado através de um programa que utiliza a rotina exemplo desenvolvida ao longo do capítulo anterior.

O capítulo 4 apresenta um conjunto de dicas a respeito de que tipo de rotinas escrever e deixar de escrever em C para uso com o Clipper.

Finalmente, o capítulo 5 apresenta diversas rotinas já prontas, com o código fonte completo, a título de exemplos. Estas rotinas podem ser incorporadas de imediato aos seus programas Clipper. Além disso, o capítulo contém uma seção com idéias a respeito de rotinas que podem ser desenvolvidas convenientemente em C.

Para facilitar o seu trabalho diário, você deve consultar o apêndice, que contém a descrição das funções da biblioteca do Clipper que podem ser usadas na criação de rotinas em C.

Agradeço a todos os que tornaram possível a existência deste livro, em particular à equipe da editora McGraw-Hill, pelo seu esforço em tornar o texto disponível, e a minha esposa Célia, pela colaboração prestada.

Roberto Carlos Mayer

São Paulo, agosto de 1989.

Capítulo 1

Uma comparação entre Clipper e C

Introdução

O Clipper é conhecido pela maioria das pessoas como um compilador para programas escritos utilizando a linguagem do dBase, um gerenciador de banco de dados desenvolvido pela Ashton-Tate, dos Estados Unidos. Porém, o Clipper, desenvolvido pela Nantucket, também dos Estados Unidos, possui uma quantidade de recursos adicionais que o tornam uma linguagem de programação mais rica. Esta linguagem se constitui num superconjunto da linguagem do dBase.

Entre muitos outros recursos que o Clipper possui a mais do que o dBase está a possibilidade de integrar rotinas escritas em outras linguagens, especificamente C e Assembly, de forma relativamente simples (como se fossem funções primitivas da linguagem).

Este texto pretende dar-lhe uma visão completa do desenvolvimento de programas integrando a linguagem do Clipper com a linguagem C. Para tanto, consideraremos que você já conhece ambas as linguagens, isto é, já sabe escrever programas em cada uma dessas linguagens separadamente.

Neste primeiro capítulo, apresentaremos uma análise comparativa entre o Clipper e C, discutindo essencialmente os aspectos necessários para a integração de ambos.

Filosofia de projeto das linguagens: adequação de acordo com o tipo de aplicação

Clipper foi projetado inicialmente como um compilador para o dBase. Nas versões mais recentes, principalmente as versões Autumn '86 e Summer '87, foi incorporada uma quantidade de recursos adicionais, que o transformaram numa linguagem única, mas que preserva os objetivos originais: o Clipper é uma linguagem destinada ao desenvolvimento de aplicações voltadas para bancos de dados, embora seja possível escrever programas em Clipper que não manipulem bancos de dados. Todos os recursos adicionais da linguagem visam facilitar a construção de aplicações de bancos de dados com uma aparência moderna (os recursos para salvar e restaurar telas, criar menus por exemplo, etc.).

Trata-se de uma linguagem de banco de dados, já que os comandos de acesso ao banco de dados fazem parte da própria linguagem, mas Clipper não pode ser considerado uma linguagem de 4ª geração, já que mantém muitas das características das linguagens tradicionalmente usadas nesse tipo de aplicações, como o Cobol. Por exemplo, a maioria absoluta das operações de arquivo efetuadas pelos comandos da linguagem operam com um registro por vez, o que obriga a escrever programas executando repetições para implementar operações que envolvem conjuntos de registros de dados.

Apesar disto, os projetistas do Clipper não perderam de vista o objetivo original da linguagem: todos os recursos denominados de "baixo nível", como por exemplo o acesso a arquivos que não possuem o formato dos arquivos de banco de dados, foram introduzidos como funções, e não como comandos novos da linguagem.

A própria linguagem do Clipper permite a abstração de código através da definição de rotinas por parte do programador. Assim, este pode definir as suas próprias funções usando os comandos do Clipper, mas estará sempre limitado a usar os recursos da linguagem.

Se você precisar de algum recurso que a linguagem não forneça, como um novo tipo de dados ou o acesso a recursos físicos do computador, não há alternativa: a linguagem do Clipper não permite. Neste caso é que devemos utilizar outra linguagem, mais apropriada para tarefas de "baixo nível".

O próprio compilador Clipper é um programa (que traduz os nossos programas em Clipper para arquivos contendo código objeto, posteriormente link-editados para produzir código executável). Ele foi escrito utilizando a linguagem C. Esta linguagem é uma linguagem de alto nível, mas permite o acesso a todos os recursos de baixo nível do computador, de forma que não existe nada que possa ser feito com um computador que não possa ser programado em C.

Assim, resulta bastante natural que os projetistas do Clipper nos fornecessem uma "porta" para integrar rotinas em C com programas Clipper. Ao fazer isto, praticamente estamos adicionando um novo recurso ao Clipper.

Cabe aqui observar que poderíamos abandonar o Clipper e escrever as nossas aplicações de bancos de dados completamente em C. Neste caso, porém, teríamos de desenvolver todos os recursos que o Clipper já nos fornece prontos, inclusive o gerenciamento do banco de dados. Mas, como você já deve ter percebido, isto seria extremamente antieconômico, já que cada programador estaria codificando novamente os mesmos recursos.

É por isso que podemos afirmar que o Clipper é uma linguagem de nível mais alto do que C. Quanto mais recursos necessários a uma classe de programadores já estiverem à disposição em uma linguagem, mais alto o seu nível. Já a linguagem C tem como uma das regras usadas no seu projeto a não inclusão na linguagem de nenhum recurso que possa ser construído utilizando outros recursos mais simples já existentes na linguagem.

Com esta perspectiva sobre as duas linguagens, analisaremos alguns recursos que existem em ambas as linguagens, como por exemplo a possibilidade de usar variáveis ou de escrever funções.

Tipos de variáveis em Clipper e em C

Em Clipper existem basicamente dois tipos de variáveis: as variáveis simples e as matrizes. Embora as variáveis simples possam ser usadas sem nenhuma declaração prévia deste fato, elas podem ser declaradas através do comando DECLARE. Este comando introduz o nome da variável. No caso das matrizes, a declaração é obrigatória e deve conter a indicação da quantidade de elementos que possuirá.

Por exemplo, estas são declarações válidas em Clipper:

```
DECLARE     VAR         && Declarações opcionais (não obrigatórias)
DECLARE     VAR1, VAR2, VAR3
DECLARE     MATRIZ[30]      && Declaração obrigatória para a matriz
```

É importante notar que nada pode ser dito a respeito dos tipos dos valores que estas variáveis podem assumir. Não existe, por exemplo, a possibilidade de indicar que uma certa variável será usada apenas para conter valores lógicos (**.T.** e **.F.**), de forma que o compilador possa emitir uma mensagem de erro no caso de atribuirmos um valor de outro tipo a esta variável.

Já em C, toda variável deve ser declarada (caso contrário, ela nem pode ser usada), indicando não só o seu nome, mas também o tipo da variável. Isto permite que o compilador efetue as checagens do tipo citado acima. Ainda permite determinar com exatidão a quantidade de memória necessária para a execução do programa sem precisar executá-lo. Por exemplo, em C declaramos

```
char   VAR;             /*Declarações sempre obrigatórias */
int    I, J, K;         /*Todas incluem um nome de tipo */
double X[80];
```

Obviamente, qualquer integração entre rotinas escritas em Clipper e C terá de fornecer um meio para compatibilizar os dois conceitos de variáveis. Veremos como isto é feito no capítulo seguinte.

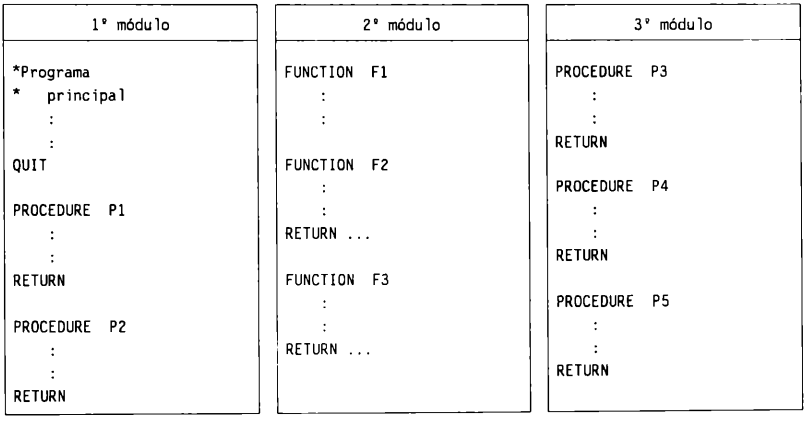
Estruturação de programas grandes em Clipper e em C

O segundo aspecto a ser considerado para a integração de rotinas escritas em C e em Clipper é o mecanismo pelo qual as rotinas acabam sendo parte dos programas. Neste aspecto, o Clipper e a linguagem C usam modelos bastante parecidos: um programa é composto por um número indeterminado de arquivos fonte, cada um denominado *módulo* do programa. Cada módulo contém um determinado número de rotinas.

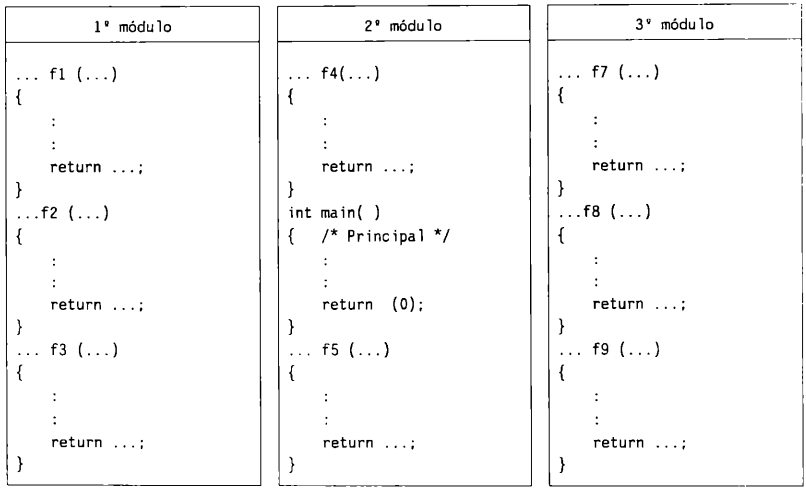
A única diferença é a forma pela qual é escolhida a rotina que será a primeira a ser executada. No Clipper, a execução inicia pelo módulo que, ao invés de conter uma rotina no início, contém diretamente comandos. Já em C, a execução inicia pela rotina

denominada **main**. Esta diferença é apenas uma questão de nomenclatura, já que nenhuma das duas formas permite efetuar alguma coisa a mais do que a outra.

Esquematicamente, um programa Clipper composto de um determinado número de módulos pode ser representado como



enquanto um programa C tem esta estrutura:



1º módulo

```
... f1 (...)
{
    :
    :
    return ...;
}
...f2 (...)
{
    :
    :
    return ...;
}
... f3 (...)
{
    :
    :
    return ...;
}
```

2º módulo

```
... f4(...)
{
    :
    :
    return ...;
}
int main( )
{   /* Principal */
    :
    :
    return  (0);
}
... f5 (...)
{
    :
    :
    return ...;
}
```

3º módulo

```
... f7 (...)
{
    :
    :
    return ...;
}
...f8 (...)
{
    :
    :
    return ...;
}
... f9 (...)
{
    :
    :
    return ...;
}
```

As rotinas que compõem cada módulo são, no caso do Clipper, funções e procedimentos. As funções produzem um valor como resultado enquanto os procedimentos não produzem nenhum valor. Esta distinção é mantida quando da utilização das rotinas, isto é, a sua chamada desde outras rotinas: no caso das funções, a chamada é simplesmente o nome da rotina seguido dos parâmetros apropriados entre parênteses, enquanto uma chamada de procedimento deve ser efetuada usando o comando **DO** *procedimento* **WITH** *parâmetros* (Se não houver parâmetros, **WITH** não precisa ser escrito.)

Em C não existe a distinção entre função e procedimento. Este último é considerado como uma função que retorna um valor de tipo vazio (**void**). Assim, a sintaxe de chamada das rotinas é uma só, e coincide com a sintaxe de chamada de funções do Clipper.

Portanto, não existem diferenças importantes entre o tratamento das rotinas em ambas as linguagens, que precisem de uma solução especial para compatibilizá-las de forma a permitir programas mistos.

A passagem de parâmetros em Clipper e em C

Apesar da correspondência quase perfeita entre os conceitos de rotina em Clipper e em C, existe uma diferença importante decorrente das diferenças entre os dois sistemas de variáveis.

Qualquer rotina de utilidade genérica vale-se de parâmetros para poder ser usada na maior quantidade possível de situações diferentes. Os parâmetros são valores usados na rotina, mas que serão, ou ao menos poderão ser, diferentes a cada execução da rotina. Sendo valores, os parâmetros são armazenados em ambas as linguagens seguindo as mesmas regras das variáveis.

A declaração de parâmetros de rotinas em Clipper é efetuada usando o comando **PARAMETERS**, que deve ser seguido dos nomes dos parâmetros (novamente não existe a possibilidade de indicar o tipo dos valores). Ainda, a declaração dos parâmetros não os torna obrigatórios: a rotina pode determinar o número de parâmetros efetivamente recebidos através da função **PCOUNT()**. Por exemplo, este procedimento imprime os parâmetros que receber (no máximo três):

```
PROCEDURE EXEMPLO
PARAMETERS P1, P2, P3

IF PCOUNT() = 0 .OR. PCOUNT() > 3
      ? "Este procedimento deve receber de 1 a 3 parâmetros"
ELSE
      IF PCOUNT() = 1
            ? P1
      ELSE
            IF PCOUNT() = 2
                  ? P1, P2
            ELSE
                  ? P1, P2, P3
            ENDIF
      ENDIF
ENDIF

RETURN
```

Já em C, a quantidade de parâmetros e o tipo de cada um precisa ser fixado ao escrever o programa, na declaração dos parâmetros:

```
void exemplo(int P1, int P2, int P3)
{
      (void) printf("%d %d %d\n", P1, P2, P3);
      return;
}
```

Assim, não é possível efetuar a passagem de parâmetros diretamente de uma rotina em Clipper para uma rotina em C (ou vice-versa). Para conseguirmos isto será necessário usar algum recurso que permita compatibilizar estas duas situações.

Quanto ao mecanismo de passagem de parâmetros, ambas as linguagens permitem que a passagem de parâmetros seja efetuada copiando valores de uma rotina para outra (passagem *por valor*), ou copiando o endereço da variável usada como parâmetro para a rotina chamada (passagem por *referência*).

No caso da passagem de parâmetros por valor, a alteração do valor de um parâmetro dentro da rotina chamada não afeta o valor indicado como parâmetro na chamada (já que foi efetuada uma cópia deste).

Já no caso da passagem de parâmetros por referência, a alteração do valor do parâmetro dentro da rotina chamada implica a alteração da variável indicada como parâmetro na chamada, porque a rotina chamada recebe, internamente, o endereço do parâmetro.

Para determinar qual dos dois mecanismos de passagem de parâmetros utilizar, o Clipper usa as seguintes regras:

- **Procedimentos:** Variáveis simples e matrizes são passadas por referência. Elementos de matrizes, expressões e nomes de variáveis ou campos entre parênteses são passados por valor.
- **Funções:** Apenas as matrizes são passadas por referência. Todos os demais tipos de parâmetros são passados por valor.

Clipper permite que você indique, na chamada de um procedimento ou função, que um determinado parâmetro deve ser passado utilizando o mecanismo de passagem por referência. Para tanto, você deve, ao efetuar a chamada, preceder o parâmetro com o sinal @. Por exemplo, se você deseja passar uma variável por referência, o nome da variável deve ser escrito *@variável* na chamada da função.

No caso deste programa Clipper

```
DECLARE X, Y, T

X = 10
Y = 20
T = TesteMecanismos(X, @Y)
? T, X, Y
RETURN

FUNCTION TesteMecanismos
PARAMETERS A, B
A = A + 10
B = B + 10
RETURN A + B
```

a saída impressa será

```
50      10      30
```

ou seja, o valor de X, passado por valor, não é alterado pela função, mas o de Y, passado por referência, o é.

Já em C, somente existe a passagem de parâmetros por valor. A passagem de parâmetros por referência é simulada através da utilização explícita de variáveis dos tipos ponteiros (que contém endereços de memória).

O programa exemplo acima seria escrito em C como

```
int main()
{
      int X, Y, T;

      X = 10;
      Y = 20;
      T = TesteMecanismos(X, &Y);
      printf("%10d%10d%10d", T, X, Y);
      return 0;
}

int TesteMecanismos(int A, int *B)
{
      A += 10;
      *B += 10;
      return A + *B;
}
```

A saída produzida é a mesma do que a do programa em Clipper.

Possibilidades de integração entre Clipper e C

A princípio, a única limitação séria para integrar rotinas escritas em Clipper e em C é o fato de que o conceito de variáveis é incompatível entre ambas as linguagens.

Podemos imaginar dois tipos de integração entre rotinas escritas nas duas linguagens: uma rotina em Clipper chamando uma rotina escrita em C, e uma rotina escrita em C chamando uma rotina escrita em Clipper.

Como o sistema de variáveis do Clipper é mais complicado (pela ausência de tipos durante a compilação), os projetistas do Clipper optaram pela solução mais simples para eles: a execução inicia por uma rotina escrita em Clipper, que pode chamar uma rotina escrita em C (esta precisa de recursos especiais para acessar os parâmetros, mas estes são os mesmos usados internamente quando uma rotina escrita em Clipper chama outra rotina escrita em Clipper).

Já a possibilidade contrária, uma rotina escrita em C chamando uma rotina escrita em Clipper, implicaria a existência de recursos para converter as variáveis de tipo fixo de C para as variáveis do Clipper, o que não acontece com programas Clipper simples (note que apesar do próprio Clipper ser escrito em C, não existe nenhuma interação entre as variáveis internas ao compilador e as variáveis do seu programa Clipper).

Portanto, a única integração possível entre rotinas escritas em C e rotinas escritas em Clipper é que estas efetuem chamadas às primeiras. O caminho inverso não é possível.

Capítulo 2

Escrevendo rotinas em C para o Clipper

Neste capítulo introduzimos a maneira de escrever rotinas em C que serão integradas com programas escritos em Clipper. Para tanto, mostramos as diferenças que devem ser levadas em conta ao escrever as rotinas em C em relação à forma em que estas seriam escritas caso fossem utilizadas em programas escritos em C.

Para facilitar a assimilação dos conceitos apresentados, desenvolveremos paralelamente às explicações uma rotina em C, que poderá ser chamada desde programas em Clipper. Esta rotina efetua a leitura do nome de discos (definido quando estes são formatados ou através do comando LABEL do MS-DOS), motivo pelo qual a denominaremos **getlabel()**.

A função **getlabel()** receberá como parâmetro a letra de identificação do disco de interesse, retornando um string contendo o rótulo do disco. Caso a função seja chamada sem parâmetros, retornará o rótulo do disco em uso correntemente.

Escrevendo funções em C junto com o Clipper

Toda função que você escrever em C para ser chamada de programas escritos em Clipper deve obedecer aos quesitos relacionados nesta seção.

Em primeiro lugar, é necessário incluir no fonte da rotina os arquivos de nome **nandef.h** e **extend.h**, fornecidos junto com o Clipper, usando os comandos **#include <nandef.h>** e **#include <extend.h>** no início do fonte em C (você pode incluir o diretório junto com o nome do arquivo, por exemplo **#include "\clipper\nandef.h"**). Estes arquivos contêm um conjunto de definições necessárias para poder estabelecer a ligação entre as rotinas escritas em C e as rotinas em Clipper que irão chamá-las.

Analisaremos o conteúdo do arquivo **extend.h** ao longo de todo este capítulo. Para tanto, apresentamos abaixo sua listagem (ligeiramente abreviada) contendo todos os elementos essenciais (observe os comentários traduzidos para o português).

```
/*    extend.h
 *
 *    Definições e variáveis da interface do Clipper com funções em C
 *
 *    Modelo de memória grande (Large) do compilador Microsoft C 5.x
 *
 *    Copyright (c) Nantucket Corporation 1987
 */

#define CLIPPER   void pascal

/*    máscaras para os valores da função _parinfo() */
#define UNDEF               0
#define CHARACTER       1
#define NUMERIC         2
#define LOGICAL         4
#define DATE            8
#define ALIAS               16
#define MPTR            32
          /*    somado ao tipo se passagem por referência */
#define MEMO            65
#define WORD            128
#define ARRAY               512


/*    protótipos das funções de acesso aos dados dos parâmetros */
extern int _parinfo(int num_parm);
extern int _parinfa(int num_parm, int indice);


/*    protótipos das funções de acesso aos valores dos parâmetros */
```

```
extern char * _parc(   int num_parm, ...);
extern int    _parcsiz(int num_parm, ...);
extern int    _parclen(int num_parm, ...);
extern int    _parni(  int num_parm, ...);
extern long   _parnl(  int num_parm, ...);
extern double _parnd(  int num_parm, ...);
extern int    _parl(   int num_parm, ...);
extern char * _pards(  int num_parm, ...);


/*     uma e somente uma das funções _ret... deve ser usada */

/*     funções para o retorno de valores às rotinas em Clipper */
extern void _retc(   char *string);
extern void _retclen(char *string, int tamanho);
extern void _retni(  int numero);
extern void _retnl(  long numero);
extern void _retnd(  double numero);
extern void _retl(   int flag);
extern void _retds(  char *string);

/*     retorno ao Clipper para chamadas efetuadas com DO */
extern void _ret(void);


/*     alocação de memória */
/*     parâmetro:  tamanho solicitado em bytes */
/*     retorna:    endereço da área ou NULL */
extern unsigned char *_exmgrab(unsigned tamanho);

/*     libera memória alocada */
/*     parâmetros: endereço de _exmgrab()
                   mesmo tamanho dado quando chamou _exmgrab() */
extern void _exmback(unsigned char *area, unsigned tamanho);


/*     macros para a checagem dos parâmetros */
#define PCOUNT          (_parinfo(0))
#define ISCHAR(n)       (_parinfo(n) & CHARACTER)
#define ISNUM(n)        (_parinfo(n) & NUMERIC)
#define ISLOG(n)        (_parinfo(n) & LOGICAL)
#define ISDATE(n)       (_parinfo(n) & DATE)
#define ISMEMO(n)       (_parinfo(n) & MEMO)
#define ISBYREF(n)      (_parinfo(n) & MPTR)
#define ISARRAY(n)      (_parinfo(n) & ARRAY)
#define ALENGTH(n)      (_parinfa(n, 0))
```

Portanto, toda função que você for escrever em C, para poder ser chamada pelo Clipper, deve ter o seguinte formato:

```
#include "\clipper\nandef.h"
#include "\clipper\extend.h"

CLIPPER <nome da função> ()
{

      /*    implementação da função */

}
```

Note o uso da definição **CLIPPER**, fornecida no **extend.h**. Ela é equivalente a escrever **void pascal**, ou seja, o compilador C deve assumir que a função não retorna nenhum valor (mesmo que retorne um valor para o Clipper) e usar as convenções de chamada internas do Pascal e não do C. Apesar disto, é melhor escrever **CLIPPER**, já que desta forma uma possível mudança destas características em versões futuras do Clipper não implicará alteração do código fonte das nossas rotinas em C.

É possível escrever várias funções em seqüência dentro de um único arquivo fonte em C, mas isto tem algumas conseqüências se quisermos usar estas funções em bibliotecas (veja a seção correspondente).

Para concluir esta seção apresentamos o esboço da função **getlabel()**, de acordo com as explicações acima:

```
#include "\clipper\nandef.h"
#include "\clipper\extend.h"

/*    Protótipo da função (opcional) */
CLIPPER getlabel(void);

/*    Definição da função */
CLIPPER getlabel()
/*    Recebe caractere indicando drive a ser utilizado
      Retorna string contendo o label do disco ("" se não há label)
      Retorna label do disco corrente se não é fornecido parâmetro
 */
{
      /*    Aqui escreveremos o código da função */
}
```

Note que, diferentemente das funções escritas em C e chamadas desde programas em C, as funções a serem chamadas desde programas escritos em Clipper não incluem nenhuma informação a respeito do(s) parâmetro(s) e do eventual valor de retorno.

Acesso a parâmetros fornecidos por programas em Clipper

Como você pôde observar na seção anterior, as funções escritas em C não declaram os parâmetros que irão receber. O acesso aos parâmetros deve ser efetuado através de um conjunto de funções, fornecidas como parte integrante da biblioteca de funções do Clipper. Analisaremos estas funções a seguir.

Para descomplicar o uso das funções, a Nantucket introduziu um conjunto de definições (macros), que estão perto do fim do arquivo **extend.h**, que permitem verificar os tipos dos parâmetros recebidos.

PCOUNT equivale ao número de parâmetros recebidos. As macros **IS...()** valem verdadeiro ou falso, conforme o parâmetro seja do tipo testado:

```
ISCHAR(num_parm)  é do tipo string do Clipper?
ISNUM(num_parm)   é do tipo número do Clipper?
ISLOG(num_parm)   é do tipo lógico do Clipper?
ISDATE(num_parm)  é do tipo data   do Clipper?
ISMEMO(num_parm)  é do tipo memo   do Clipper?
ISARRAY(num_parm) é do tipo matriz do Clipper?
ISBYREF(num_parm) foi passado  por referência?
```

O *num_parm* é o número de ordem do parâmetro sendo testado. Por exemplo, para testar se o primeiro e o último parâmetro são do tipo numérico, podemos escrever (em C)

```
ISNUM(1) && ISNUM(PCOUNT)
```

No caso das matrizes, a função **_parinfa()** fornece as informações necessárias (compare com **extend.h**):

```
_parinfa(num_parm, 0)      número de elementos da matriz
_parinfa(num_parm, 1)      tipo do 1° elemento da matriz
_parinfa(num_parm, 2)      tipo do 2° elemento da matriz
          :
          :
```

```
_parinfa(num_parm, _parinfa(num_parm, 0))
                                    tipo do último elemento da matriz
```

Note que todas estas funções servem apenas para informar a quantidade e os tipos dos parâmetros. Isto é necessário para que a nossa função em C saiba que operações aplicar aos parâmetros. No caso de variáveis string podemos usar a função _parclen() para saber o tamanho corrente do string e a função _parcsiz() para saber quantos caracteres foram reservados para esse valor na memória.

Para acessar os valores dos parâmetros usaremos as funções par...() (veja os seus protótipos no extend.h). Note a correspondência entre os tipos do Clipper e os tipos de C:

string	char *
data	char *
lógico	int
número	int, long ou double

Todas as funções **_par...()** têm como primeiro parâmetro o número do parâmetro e como segundo parâmetro, quando for uma matriz, o índice do elemento sendo acessado. Discutiremos agora estas funções individualmente.

As funções **_parc()** e **_pards()** retornam o endereço onde o Clipper armazenou a variável correspondente (as datas são armazenadas como strings no formato **AAAAMMDD**). Note que são apenas os endereços que são retornados. Você deve copiar os valores em variáveis locais se quiser usar os valores dos parâmetros.

A função **_parl()** retorna 1 ou 0, conforme o valor do parâmetro do Clipper seja **.T.** ou **.F.**, respectivamente.

As funções **_parni()**, **_parnl()** e **_parnd()** convertem o parâmetro numérico em um valor dos tipos **int**, **long** e **double** de C, respectivamente.

Note que todas estas possibilidades precisam ser analisadas pela função que você escrever antes de tomar qualquer iniciativa. Se as variáveis do Clipper fossem declaradas com tipo, nada disto seria necessário. Estas funções todas servem de ponte entre os dois sistemas de variáveis.

Por exemplo, para varrer todos os parâmetros e determinar os seus tipos, poderíamos usar a estrutura de função seguinte:

```
#include "\clipper\nandef.h"
#include "\clipper\extend.h"

CLIPPER <nome da função> ()
{
      register int i, j;

      for (i = 1; i <= PCOUNT; i++)
            if (ISARRAY(i))
                  for (j = 1; j <= ALENGTH(i); j++)
                        /*    processa cada elemento da matriz */
            else if (ISCHAR(i)) {
                  char *string = _parc(i);
                  /*    processa o string */
            }
            else if (IS...
                  /*    casos restantes
                        :
                        :
                  */
            else
                  /*    trata a chamada com parâmetro de tipo errado */
}
```

No caso da função **getlabel()**, desejamos verificar que a função tenha sido chamada com um único parâmetro, do tipo string, contendo um único caractere (a letra de identificação do disco a ser utilizado). Assim, podemos expandir o código de **getlabel()** para

```
#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "\clipper\nandef.h"
#include "\clipper\extend.h"

/*    Protótipo da função (opcional) */
CLIPPER getlabel(void);

/*    Definição da função */
CLIPPER getlabel()
/*    Recebe caractere indicando drive a ser utilizado
      Retorna string contendo o label do disco ("" se não há label)
```

```
        Retorna label do disco corrente se não é fornecido parâmetro
 */
{
        char DRIVE;    /* Letra do disco a ser usado */

        /*    Se número de parâmetros é incorreto */
        if (PCOUNT != 1
        /*    ou o parâmetro não é um string */
        || !ISCHAR(1)
        /*    ou o string contém mais de um caractere */
        || strlen(_parc(1)) > 1
        /*    ou o caractere contido não é uma letra */
        || !isalpha(_parc(1)[0])) {
                unsigned DRIVE_NUM;

                /*    Detecta disco corrente com a função _dos_getdrive
                      da biblioteca do compilador C Microsoft */
                _dos_getdrive(&DRIVE_NUM);
                /*    Converte o número obtido na letra correspondente */
                DRIVE = (char) (64u + DRIVE_NUM);
        }
        else
                /*    Use como disco o primeiro (e único) caractere contido
                      no string recebido como parâmetro */
                DRIVE = _parc(1)[0];

        /*    Aqui escreveremos o resto do código da função */

}
```

Note a inclusão dos arquivos de cabeçalho correspondentes às funções da biblioteca do compilador C, chamadas pelo código de **getlabel()**.

Retorno de valores para programas em Clipper

Analogamente à passagem de parâmetros, o retorno de valores da função em C para o programa Clipper deve ser efetuado através de funções fornecidas na biblioteca de funções do Clipper. Neste caso, usaremos a família de funções **_ret...()** (veja os seus protótipos no **extend.h**).

As funções **_retc()** e **_retds()** retornam ao Clipper o endereço onde você armazenou a variável correspondente (as datas devem estar

armazenadas como strings no formato **AAAAMMDD**). Note que as funções retornam apenas os endereços, portanto as strings a serem retornadas devem ser declaradas como variáveis de classe **static** em C, de forma a garantir que os seus valores continuem a existir mesmo após o encerramento da execução da função que os gerou.

Alternativamente, você pode usar a função _retclen(), que permite passar ao programa Clipper o endereço do string e o seu tamanho (a função **_retc()** considera o terminador **'\0'**).

A função **_retl()** retorna um valor **.T.** ou **.F.** para o programa Clipper, conforme o valor do parâmetro seja 1 ou 0, respectivamente.

As funções **_retni()**, **_retnl()** e **_retnd()** retornam o parâmetro numérico do tipo int, long e double de C, respectivamente, em um valor numérico do Clipper.

Note que todas estas funções apenas retornam o valor para o programa Clipper. A execução retorna ao programa Clipper através do return normal do C (que pode ser omitido se for o último comando da função, já que as rotinas escritas em C são declaradas como sendo do tipo **void**).

Podemos agora completar a primeira versão operacional da função **getlabel()**:

```
#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "\clipper\nandef.h"
#include "\clipper\extend.h"

/*     Protótipo da função (opcional) */
CLIPPER getlabel(void);

/*     Definição da função */
CLIPPER getlabel()
/*     Recebe caractere indicando drive a ser utilizado
       Retorna string contendo o label do disco ("" se não há label)
       Retorna label do disco corrente se não é fornecido parâmetro
 */
{
       /*   Variáveis para pesquisa no diretório */
```

```
        static struct find_t BUFFER;     /*    Buffer de pesquisa */
        static char PATH[] = "C:\\*.*";  /*    Chave de pesquisa */

        char DRIVE;    /* Letra do disco a ser usado */

        /*    Se número de parâmetros é incorreto */
        if (PCOUNT != 1
        /*    ou o parâmetro não é um string */
        || !ISCHAR(1)
        /*    ou o string contém mais de um caractere */
        || strlen(_parc(1)) > 1
        /*    ou o caractere contido não é uma letra */
        || !isalpha(_parc(1)[0])) {
             unsigned DRIVE_NUM;

             /*    Detecta disco corrente com a função _dos_getdrive
                  da biblioteca do compilador C Microsoft */
             _dos_getdrive(&DRIVE_NUM);
             /*    Converte o número obtido na letra correspondente */
             DRIVE = (char) (64u + DRIVE_NUM);
        }
        else
             /*    Use como disco o primeiro (e único) caractere contido
                  na string recebida como parâmetro */
             DRIVE = _parc(1)[0];

        /*    Transforma letra do disco em maiúscula */
        PATH[0] = toupper(DRIVE);

        /*    Se não achar rótulo do disco no diretório raiz */
        if (_dos_findfirst(PATH, _A_VOLID, &BUFFER))
             /*    O nome do rótulo será o string vazio */
             BUFFER.name[0] = '\0';
        /*    Se achar o rótulo e tiver mais de 8 caracteres */
        else if (BUFFER.name[8] == '.')
             /*    Elimine o ponto que precede o nono caractere */
             strcpy(&BUFFER.name[8], &BUFFER.name[9]);

        /*    Devolve o rótulo detectado para o programa em Clipper */
        _retc(BUFFER.name);
        return;
}
```

No capítulo seguinte, onde explicamos como compilar esta função, suporemos que ela está gravada no arquivo fonte de nome **getlabel.c**.

Observe que as variáveis cujos valores são retornados ao programa Clipper têm que ser declaradas com a classe estática, para que os valores continuem a existir após o encerramento da execução da função.

Chamada de funções C desde programas Clipper

A chamada das funções escritas em C desde programas escritos em Clipper é efetuada da mesma forma como se as funções fossem escritas em Clipper, com o nome e os parâmetros entre parênteses. Caso a função em C não retorne valor algum, ela deve ser chamada como se fosse um procedimento em Clipper, usando o comando **DO/WITH**.

Por exemplo, este programa Clipper, que supomos estar gravado no arquivo fonte de nome **getlaclp.prg**, chama a função **getlabel()**

```
* Programa de teste da rotina getlabel
PARAMETERS DRIVE

DECLARE LABEL

IF PCOUNT() = 1
      LABEL = GETLABEL(DRIVE)
ELSE
      LABEL = GETLABEL()
ENDIF

? "Label do disco :" + LABEL + ":"

QUIT
```

Note o uso do nome do disco como parâmetro de linha de comando.

Chamada de funções C desde funções C chamadas do Clipper

Se as funções que você for escrever para serem chamadas de programas Clipper forem complexas, você pode decompô-las em várias subfunções. A função chamada do Clipper usará a interface com o Clipper detalhada nas seções anteriores, enquanto as demais funções,

chamadas por outras funções C e retornando para funções C, devem ser escritas obedecendo à sintaxe normal de C, inclusive no que diz respeito à passagem de parâmetros e retorno de valores.

A mesma observação vale para o uso das funções da biblioteca do compilador C (mas nem toda função pode ser usada; veja o Capítulo 4 para maiores informações).

A função **getlabel()** inclui várias chamadas a funções escritas em C, da biblioteca do compilador, como **strlen()**, **strcpy()**, **isalpha()**, **toupper()**, **_dos_getdrive()** e **_dos_findfirst()**.

Alocação dinâmica de memória em rotinas em C

Caso você deseje alocar variáveis dinamicamente nas suas funções em C que serão chamadas de programas Clipper, você não pode usar as funções de alocação dinâmica do tipo de **malloc()** e **free()**, porque o controle da memória está a cargo do Clipper e não do compilador C, durante a execução do programa.

Entretanto, para permitir escrever funções que utilizem áreas de memória alocadas dinamicamente, o Clipper fornece duas funções em sua biblioteca, chamadas **_exmgrab()** e **_exmback()** (veja os protótipos na listagem de **extend.h**).

A função **_exmgrab()** é a encarregada de alocar memória. Ela deve ser chamada com um número inteiro sem sinal como parâmetro, que indica o tamanho da área de memória desejada. Se a função conseguir alocar memória, retorna um ponteiro para a região alocada, caso contrário retorna o valor **NULL**.

Para liberar memória previamente alocada com **_exmgrab()** e que não será mais necessária, você deve usar a função **_exmback()**. A chamada deve ter dois parâmetros: o endereço da área a ser liberada e o seu tamanho (note que estes valores correspondem ao valor de retorno e ao parâmetro de **_exmgrab()**, respectivamente). A função **_exmback()** não retorna nenhum valor.

Por exemplo, na função **getlabel()**, poderíamos ter alocado a variável **PATH** dinamicamente com uma declaração:

```
char *PATH = _exmgrab(13);
```

Antes de usar a variável, deveríamos verificar que o ponteiro PATH não tivesse **NULL** como valor. Ao fim da função deveríamos liberar a memória alocada através de uma chamada à função **exmback()**:

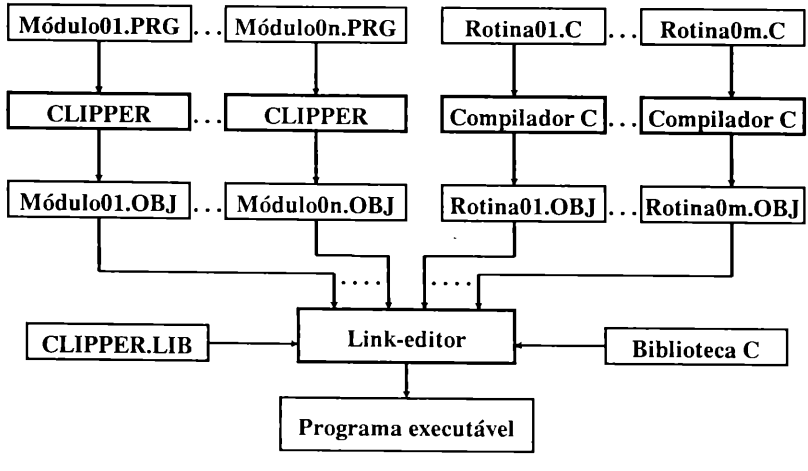
```
_exmback(PATH, 13);
```

Note que a ausência deste comando implicaria a criação de mais uma área de memória a cada vez que a função **getlabel()** fosse chamada.

Capítulo 3

Compilando e ligando rotinas em Clipper e C

O processo de criação de programas Clipper contendo rotinas em C é semelhante ao processo para programas escritos nestas linguagens isoladamente: cada módulo é compilado separadamente e depois é efetuada a link-edição.

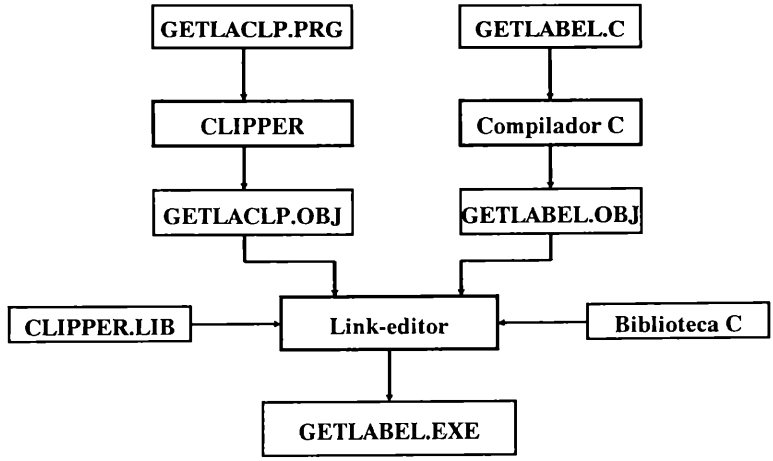


Na figura acima, o processo é representado para programas escritos numa combinação de Clipper e C. As caixas traçadas com linhas

simples representam arquivos utilizados no processo, enquanto as traçadas com linhas duplas representam programas necessários.

Neste capítulo continuaremos exemplificando os conceitos apresentados através da implementação da função **getlabel()**. Lembre-se de que supomos que os arquivos **getlaclp.prg** e **getlabel.c** contêm respectivamente o código fonte do programa de teste em Clipper e da função em C.

O processo de geração do programa executável **getlabel.exe**, contendo ambos, pode ser representado graficamente por



O restante deste capítulo descreve detalhadamente cada passo deste processo.

O processo de compilação de rotinas em C

Como o Clipper, a partir da versão Summer '87, foi desenvolvido com o compilador C da Microsoft Corp, na sua versão 5.0, devemos usar este mesmo compilador (ou uma versão mais recente compatível com a 5.0) para compilar as funções que escrevemos em C.

Se a função a ser compilada estiver contida no arquivo de nome **funcao.c**, o comando de compilação deverá ser

```
cl -c -AL -Zl -Oalt -FPa -Gs funcao.c
```

onde as opções especificadas têm o seguinte significado:

-c indica ao compilador C que somente deve compilar a função (caso contrário ele iria chamar o link-editor e tentar criar o arquivo **funcao.exe**);

-AL indica que deve ser usado o modelo de memória Large (grande), no qual tanto os programas como os dados podem ocupar qualquer quantidade de memória. A única limitação é que nenhuma matriz pode, ela individualmente, ocupar mais de 64 Kb de memória, o que é bastante razoável;

-Zl informa ao compilador que o arquivo **funcao.obj** a ser gerado não deve conter referência a nenhuma biblioteca de funções (elas terão de ser especificadas manualmente na hora da link-edição);

-Oalt indica ao compilador o tipo de otimização desejada (**a** significa que cada variável do programa é acessada através de um único nome, **l** significa otimização de repetições – loops –, e **t** indica que deve ser favorecida a velocidade de execução em relação ao tamanho do código gerado durante a otimização);

-FPa significa que será usada a biblioteca matemática alternativa da Microsoft, mais rápida do que o emulador do 8087, porém com precisão menor (para ser compatível com as variáveis do Clipper);

-Gs indica que não deve ser gerado código para verificar a integridade da pilha. Como esta é usada pelo Clipper de forma diferente do compilador C, a ausência desta opção produziria mensagens de erro toda vez que a pilha fosse verificada pela rotina do compilador C (ao entrar em uma função escrita em C).

Para evitar digitar todas as opções necessárias na compilação de cada rotina, você pode criar um arquivo de comandos (**.BAT**) da seguinte forma:

```
cl -c -AL -Zl -Oalt -FPa -Gs %1
```

Se o arquivo for criado com o nome **CLCLIPER.BAT**, então você pode compilar as suas rotinas com o comando

```
clcliper funcao.c
```

onde **funcao.c** deve ser substituído pelo nome do arquivo contendo a função que você deseja compilar.

Por exemplo, o arquivo **getlabel.c** pode ser compilado com o comando

```
cl -c -AL -Zl -Oalt -FPa -Gs getlabel.c
```

ou, supondo que você criou o arquivo **CLCLIPER.BAT**, com o comando

```
clcliper getlabel.c
```

Em ambos os casos é executado o compilador C, gerando o arquivo objeto de nome **getlabel.obj**.

O processo de compilação de rotinas em Clipper

Após a compilação da função em C, como explicado na seção anterior, devemos compilar o programa Clipper que chama a função em C. Se este programa chamar **programa.prg**, o Clipper criará um arquivo chamado **programa.obj** através do comando:

```
clipper programa.prg -m
```

onde a opção **-m** indica ao Clipper para não procurar e compilar automaticamente outros módulos chamados por aquele sendo compilado. Isto evita, entre outras coisas, que o Clipper tente compilar a função escrita em C.

No caso do nosso exemplo, é necessário compilar o programa Clipper **GETLACLP.PRG**. Isto pode ser efetuado com o comando

```
clipper getlaclp.prg -m
```

gerando o arquivo objeto **GETLACLP.OBJ**.

Ligando rotinas C e Clipper com as bibliotecas

O passo final para a criação do programa executável contendo as rotinas escritas em C e Clipper é a ligação destes com as bibliotecas de ambos os compiladores.

Se você usar o link-editor da Microsoft, LINK, ele deve ser chamado com um comando como o seguinte, para criar o programa executável **programa.exe**:

```
link programa funcao,programa,nul,llibca clipper /NOE
```

onde o primeiro parâmetro do link-editor são os arquivos de extensão **.obj**, o segundo é o nome a ser dado ao arquivo executável (extensão **.exe**), **nul** descarta o mapa de variáveis e o último parâmetro são as bibliotecas de funções do Clipper e do C.

Note que você deve indicar também o diretório das bibliotecas, a não ser que você tenha usado o comando **SET LIB** do DOS para definir os diretórios onde as bibliotecas devem ser procuradas. Ainda, note que o(s) nome(s) exato(s) da(s) biblioteca(s) do compilador C depende(m) de como você instalou o seu compilador (se você não chamar nenhuma função da biblioteca do C, pode omiti-la).

A opção **/NOE**, abreviatura de **/NOEXTDICTIONARY**, evita que o link-editor considere como erro a existência de símbolos com o mesmo nome em vários dos módulos sendo ligados (esta opção é necessária porque a biblioteca do Clipper contém símbolos definidos também na biblioteca do compilador C; você obterá eventualmente algumas mensagens de erro indicando isto, mas o programa executável será gerado corretamente, se você usar a opção).

Para gerar o programa executável **getlabel.exe** a partir dos arquivos objeto criados nas duas seções anteriores com o LINK da Microsoft, você deve usar o comando

```
link getlabel getlaclp,getlabel,nul,llibca clipper /NOE
```

ou, alternativamente,

```
link getlabel+getlaclp,getlabel,nul,llibca+clipper /NOE
```

A ordem em que os arquivos objeto são relacionados pode ser modificada, mas não é recomendável alterar a ordem das bibliotecas de funções.

Caso você utilize o link-editor da Borland, o Turbo Linker (que tem a vantagem de executar mais rapidamente), você deve usar um comando como este para efetuar a link-edição:

```
tlink programa funcao,programa,nul,llibca clipper
```

Note a ausência da opção **/NOE**. O Turbo Linker só verifica a existência de símbolos com nomes duplicados entre os módulos sendo ligados caso seja explicitamente solicitado (através do uso da opção **/d**).*

O comando correspondente para o nosso programa exemplo seria

```
tlink getlabel getaclp,getlabel,nul,llibca clipper
```

Eventualmente, dependendo do tamanho do(s) seu(s) programa(s), você precisará indicar ao link-editor o número de segmentos de memória necessários. Caso você esteja usando o LINK da Microsoft, o número de segmentos assumido como default é 128 (você pode usar a opção **/SEGMENTS:***número* para indicar um número maior de segmentos, onde *número* deve indicar a quantidade desejada de segmentos). No Turbo Linker não existe nada análogo: se o seu programa não pode ser ligado por causa do número excessivo de segmentos, você receberá simplesmente uma mensagem de erro (a solução será usar o LINK da Microsoft).

No LINK da Microsoft você também pode indicar a quantidade de memória necessária para a pilha do seu programa (o default é 2048 bytes, use a opção **/STACK:***número* para solicitar o uso de uma área de memória de tamanho diferente para a pilha).

A criação de bibliotecas de funções

Na medida em que for criando mais funções C para serem chamadas de programas Clipper, será cada vez mais difícil você lembrar quais funções realmente usou em cada um de seus programas escritos em Clipper.

Se você quiser deixar o trabalho de procurar quais módulos objeto são necessários para a link-edição ao próprio link-editor, você

* Se você optar pelo uso do Turbo Link, deve saber que este possui um pequeno problema: em certas situações, o arquivo executável gerado não contém as instruções para que as áreas de memória correspondentes às variáveis da classe **static** não inicializadas explicitamente nas funções escritas em C sejam inicializadas com zeros em binário durante a carga do programa executável na memória (efetuada pelo interpretador de comandos do MS-DOS quando é solicitada a execução do programa). De acordo com o manual do Turbo C, isto é devido ao fato de que o compilador da Microsoft utiliza certos formatos no arquivo objeto que não foram documentados pela Microsoft.

deverá criar uma biblioteca contendo as funções. Para tanto, você precisará dispor de um utilitário de gerenciamento de bibliotecas, como o **LIB** (se você tem o compilador C da Microsoft, **LIB** faz parte do disco denominado *Utilities*) ou o **TLIB**, que acompanha as linguagens da Borland. Nesta seção descrevemos apenas o uso do LIB.

Para incluir um arquivo objeto em uma biblioteca, use o comando

```
LIB biblioteca +arquivo;
```

Você pode eliminar módulos objeto da biblioteca indicando como a operação na frente do nome do arquivo, ou substituir um módulo já contido na biblioteca por uma nova versão com a operação +.

LIB cria ou usa arquivos de biblioteca de extensão **.lib**. Por exemplo, para criar a biblioteca **cclipper.lib** com a função dada pelo módulo **getlabel.obj**, use o comando

```
LIB cclipper +getlabel;
```

Note que o link-editor incluirá nos programas executáveis sempre módulos objeto completos. Portanto, se o seu módulo objeto contém várias funções e você chama apenas uma delas, o código de todas será incluído no arquivo executável. Para evitar este problema, você deve criar cada função como um arquivo fonte e objeto separado. Por outro lado, isto aumentará o tempo necessário para a link-edição.

A praxe mais comum é usar uma função por arquivo, exceto quando forem funções que não podem ser chamadas isoladamente, isto é, se todo programa que chama a função **A()** tem que chamar a função **B()**, direta ou indiretamente, você não perderá nada colocando as duas funções em um único módulo.

Automatizando o processo de compilação e ligação

Junto com o compilador C, a Microsoft fornece um utilitário denominado **MAKE**, que é uma versão para MS-DOS de um utilitário do mesmo nome, existente no sistema operacional Unix.

MAKE pode ser usado para automatizar a compilação e ligação de programas. Para tanto é necessário criar um arquivo especial, que

chamaremos de "arquivo de make", contendo blocos organizados desta forma:

```
dependente: depêndencias
      comando
```

onde *dependente* é um nome de arquivo e *dependências* é uma lista de nomes de arquivos. Caso o arquivo *dependente* seja mais antigo do que algum dos arquivos da lista, o *comando* será executado sob o controle de **MAKE** (caso contrário o bloco é ignorado, evitando a recompilação ou religação quando desnecessárias). Por exemplo, se você cria um arquivo de nome **TESTE.** (sem extensão) contendo

```
rotinac.obj: rotinac.c
      cl -c -AL -Zl -Oalt -FPa -Gs rotinac.c

teste.obj: teste.prg
      clipper teste.prg -m

rotinac.exe: rotinac.obj teste.obj
      link teste+rotinac,teste,nul,\msc\lib\llibca+\clipper\clipper /NOE
```

e solicitar a sua execução através do comando

```
MAKE TESTE
```

o utilitário **MAKE** compilará, na seqüência, a rotina em C, o programa de teste e ligará ambos, somente se necessário. Por exemplo, após uma alteração da rotina em C, só será necessário recompilar esta rotina e ligar novamente com o programa **teste**.

Denominamos o arquivo passado ao utilitário **MAKE** para controlar o processo de compilação de **arquivo de make**.

O arquivo de **make** correspondente ao programa **getlabel**, contido no arquivo **GETLABEL**, contém as seguintes regras de dependência:

```
#getlabel - arquivo de make para getlabel.exe

getlabel.obj: getlabel.c
      cl -c -AL -W3 -Zl -Oalt -FPa -Gs getlabel.c

getlaclp.obj: getlaclp.prg

      clipper getlaclp -m
```

```
getlabel.exe: getlabel.obj getlaclp.obj
      link @getlabel.lnk
```

onde **getlabel.lnk** é um arquivo contendo os parâmetros de LINK:

```
getlabel getlaclp
getlabel
nul
\msc\lib\llibca \clipper\clipper /NOE
```

A primeira linha contém os nomes dos arquivos objeto, a segunda o nome do programa executável, a terceira o destino do mapa de memória do programa executável (**nul** descarta esta saída) e a última os nomes das bibliotecas do compilador C e do Clipper.

Quando você for criar programas maiores, envolvendo um número maior de arquivos fonte, você pode automatizar o processo ainda mais, incluindo no arquivo de **make** regras de inferência. Estas regras indicam a **MAKE** o comando a ser executado em função das extensões do arquivo dependente e suas dependências. Por exemplo, a regra

```
.c.obj:
      cl -c -AL -W3 -Zl -Oalt -FPa -Gs $*.c
```

define que toda vez que um arquivo de extensão **.obj** precise ser criado a partir de um arquivo fonte de extensão **.c**, deve ser executado o comando indicado. Observe que **$*** é um símbolo especial, substituído por **MAKE** pelo nome do arquivo dependente (sem a extensão).

Definindo regras analogamente para a compilação com o Clipper e o uso do LINK, o arquivo de **make** pode ser escrito

```
#getlabel - arquivo de make para getlabel.exe

#regras de inferência

.c.obj:        # de C   para OBJ
      cl -c -AL -W3 -Zl -Oalt -FPa -Gs $*.c

.prg.obj:      # de PRG para OBJ
      clipper $*.prg -m

.obj.exe:      # de OBJ para EXE
      link @$*.lnk
```

```
# regras de dependência

getlabel.obj: getlabel.c

getlaclp.obj: getlaclp.prg

getlabel.exe: getlabel.obj getlaclp.obj
```

Quando uma relação de dependência não é seguida de nenhum comando, **MAKE** procura utilizar uma das regras de inferência.

Depurando rotinas em C para uso com o Clipper

Caso você programe em C habitualmente, deve estar familiarizado com as ferramentas de depuração existentes, como por exemplo o **CodeView** da Microsoft e o **Turbo Debugger** da Borland. Infelizmente não é possível usar estas ferramentas com programas que contenham rotinas em C e em Clipper, porque este último não gera arquivos objeto no formato necessário.

Assim, a depuração de rotinas em C pode tornar-se extremamente difícil quando estas já estiverem ligadas com programas escritos em Clipper. A única chance de usar as ferramentas de depuração é se depurarmos as rotinas em C como se fossem programas autônomos, **antes** de sua inclusão em programas Clipper.

Para tanto não é necessário escrever duas versões da rotina. Você deve usar a compilação condicional do pré-processador da linguagem C (os comandos **#if**, **#else** e **#endif**) para incluir no mesmo arquivo fonte a interface normal de C e a interface especial do Clipper, além de uma função principal **main()** para poder testar a rotina de forma isolada (veja por exemplo a listagem da rotina **extenso.c** no Capítulo 5). Use um símbolo do pré-processador (no exemplo é usado o símbolo **COM_CLIPPER**) para controlar o tipo de compilação.

No caso da função **getlabel()**, podemos aplicar esta estratégia da seguinte forma (observe os comentários na listagem):

```
#include <dos.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

```
/*    Se COM_CLIPPER existe, inclui os arquivos do Clipper */
#if defined(COM_CLIPPER)
      #include "\clipper\nandef.h"
      #include "\clipper\extend.h"

      /*    Protótipo para uso com o Clipper */
      CLIPER getlabel(void);
#else
      /*    Protótipos para uso como programa C autônomo */
      char *getlabel(char DRIVE);
      int main(void);
#endif


#if defined(COM_CLIPPER)
      /*    Cabeçalho para uso com o Clipper */
      CLIPPER getlabel()
#else
      /*    Cabeçalho para uso como função normal em C */
      char *getlabel(char DRIVE)
#endif
/*    Recebe caractere indicando drive a ser utilizado
      Retorna string contendo o label do disco ("" se não há label)
      Retorna label do disco corrente se não é fornecido parâmetro
 */
{
      static struct find_t BUFFER;
      static char PATH[] = "C:\\*.*";

/*    Se uso com o Clipper, defina os valores dos parâmetros
      através de chamadas às funções da interface Clipper-C
 */
#if defined(COM_CLIPPER)
      char DRIVE;

      /*    Se número de parâmetros é incorreto */
      if (PCOUNT != 1
      /*    ou o parâmetro não é um string */
      || !ISCHAR(1)
      /*    ou o string contém mais de um caractere */
      || strlen(_parc(1)) > 1
      /*    ou o caractere contido não é uma letra */
      || !isalpha(_parc(1)[0])) {
            unsigned DRIVE_NUM;
```

```
            /*    Detecta disco corrente com a função _dos_getdrive
                  da biblioteca do compilador C Microsoft */
            _dos_getdrive(&DRIVE_NUM);
            /*    Converte o número obtido na letra correspondente */
            DRIVE = (char) (64u + DRIVE_NUM);
      }
      else
            /*    Use como disco o primeiro (e único) caractere contido
                  na string recebida como parâmetro */
            DRIVE = _parc(1)[0];
#endif
/*    Caso contrário os parâmetros já estão definidos pelo
      cabeçalho da função, como qualquer função em C
 */


      /*    Transforma letra do disco em maiúscula */
      PATH[0] = toupper(DRIVE);

      /*    Se não achar rótulo do disco no diretório raiz */
      if (_dos_findfirst(PATH, _A_VOLID, &BUFFER))
            /*    O nome do rótulo será a string vazia */
            BUFFER.name[0] = '\0';
      /*    Se achar o rótulo e tiver mais de 8 caracteres */
      else if (BUFFER.name[8] == '.')
            /*    Elimine o ponto que precede o nono caractere */
            strcpy(&BUFFER.name[8], &BUFFER.name[9]);

#if defined(COM_CLIPPER)
      /*    Devolve o rótulo detectado para o programa em Clipper */
      _retc(BUFFER.name);
      return;
#else
      /*    Devolve o rótulo detectado para o programa em C */
      return BUFFER.name;
#endif
}


/*    Quando COM_CLIPPER está ausente, isto é um programa! */
#if !defined(COM_CLIPPER)
int main()
{
      char DISCO[12];

      do {
            fputs("Digite letra de drive: ", stdout);
```

```
            gets(DISCO);
            /*    Se não é digitado 'Enter' sem nenhum caractere antes */
            if (*DISCO)
                  /*    Chama a função getlabel() */
                  printf("Label do disco %c tem %d posições: %s\n",
                              DISCO[0],
                              strlen(getlabel(*DISCO)), getlabel(*DISCO));
      } while (*DISCO);

      return 0;
}
#endif
```

Como apresentado, o arquivo **getlabel.c** contém um programa completo escrito em C. Este programa deve ser depurado, utilizando todas as ferramentas disponíveis para programas C.

Uma vez que a função em C esteja depurada, a sua inclusão em programas Clipper é mais simples, pois qualquer erro que apareça na função terá que ser um erro de interface, e não um erro de lógica interna da função. Para tanto, é suficiente definir o símbolo **COM_CLIPPER** no início do arquivo fonte:

```
#define COM_CLIPPER
```

Não há necessidade do símbolo possuir um valor.

Capítulo 4

O que pode e deve ser escrito em C

Analisaremos neste capítulo as restrições práticas decorrentes da integração de rotinas em C com programas Clipper e as suas conseqüências sobre as atividades que as rotinas em C podem realizar.

Através desta análise você tomará conhecimento do tipo de rotinas que é mais conveniente escrever em C em vez de usar o Clipper, e quais os recursos habituais do C que podem ser usados no desenvolvimento destas rotinas.

O que pode ser acessado por rotinas C, em relação à biblioteca de funções, periféricos e arquivos

Já vimos, como no caso das funções de alocação dinâmica de memória, que existem restrições quanto aos recursos de C que devem ou podem ser usados pelas funções escritas em C para serem integradas com programas desenvolvidos em Clipper. Vamos analisar agora os motivos disto e apresentar algumas orientações a respeito.

Ao criar uma função em C para ser chamada por programas escritos em Clipper, você estará criando, através do uso do compilador C, código objeto que será executado sob o controle do Clipper, e não sob controle do compilador C. Normalmente, o compilador C introduz uma seqüência de atividades de inicialização que são executadas antes de iniciar a execução do programa C pela função **main()** (como a conversão

da linha de comando em matriz de strings, o eventual redirecionamento da entrada e saída, a inicialização do sistema de alocação dinâmica de memória, a detecção e inicialização de um coprocessador numérico, a inicialização de variáveis globais da biblioteca de funções e a abertura dos periféricos **stdin**, **stdout**, **stderr**, **stdaux** e **stdprn**, entre outros). Ao usar as funções em C com o Clipper, nenhuma destas atividades de inicialização é realizada.

Portanto, ao escrever as funções em C, elas não poderão usar as funções da biblioteca do compilador C que dependem dos processos descritos. Entre estas funções, estão todas as funções do grupo de entrada e saída de alto nível, as funções de classificação de caracteres (**is...()**) e as funções de alocação dinâmica de memória.

Como você pode perceber, esta é uma restrição séria ao que pode ser desenvolvido em C. A Nantucket recomenda que toda a entrada e saída dos programas que estejam escritos parcialmente em Clipper seja efetuada através de comandos da própria linguagem do Clipper e não por funções em C, mesmo que o compilador C forneça funções que não precisam de inicialização para poder ser usadas. Estas funções poderiam, por exemplo, acessar diretamente a memória de vídeo, mas desta forma estariam alterando o conteúdo da tela sem o conhecimento das rotinas do Clipper, o que pode levar estas a comportar-se de forma inesperada ou errada em chamadas subseqüentes a elas.

Uma outra possibilidade em que você poderia estar pensando seria chamar, desde as funções em C, diretamente as funções da biblioteca do Clipper. Isto permitiria não só o uso da entrada e saída como também a manipulação do próprio banco de dados desde funções escritas em C. Desafortunadamente, a Nantucket não fornece a documentação necessária para que possamos fazer isto. (Suspeitamos que o pessoal da Nantucket esteja com o receio de que, se publicarem as convenções de chamada dessas funções da biblioteca do Clipper, muitos programadores deixem de escrever programas em Clipper e passem a escrever os seus programas exclusivamente em C, usando apenas a biblioteca de funções do Clipper.)

Se toda a entrada e saída for efetuada com os comandos da linguagem do Clipper, e as rotinas em C não podem chamar nenhuma função da biblioteca de funções do Clipper além das funções de interface já discutidas nos capítulos anteriores, então o tipo de tarefas que

podemos programar em C para usar junto com o Clipper fica claramente delimitado: podemos usar as funções em C somente para fornecer recursos que possam ser implementados de forma autônoma, como por exemplo, funções que efetuem cálculos ou conversões de dados e funções que acessem recursos de baixo nível não utilizados pelo Clipper. Discutiremos brevemente estas duas categorias de funções.

Acesso aos recursos internos do PC

Este tipo de funções pode acessar recursos internos do MS-DOS que não estejam disponíveis dentro do Clipper, como por exemplo a manipulação de diretórios (embora possamos usar estes recursos do MS-DOS indiretamente na linguagem do Clipper através do comando **RUN**).

Outro tipo de funções pode ser o de funções que acessem os recursos da BIOS do IBM PC. Através deste tipo de funções podemos, por exemplo, descobrir a configuração de hardware na qual o programa está rodando.

Finalmente, é possível escrever funções em C que acessem diretamente o hardware, através de acesso direto à memória ou às portas de entrada e saída.

Criando rotinas mais eficientes

Este é o segundo tipo de funções que podemos escrever em C. Trata-se basicamente de efetuar tarefas que demandam um número significativo de operações, o que as tornaria muito lentas se programadas em Clipper, através do uso de funções escritas em C.

Dentro deste tipo de rotinas podemos citar funções que geram números e datas por extenso, funções que manipulem números contidos em strings, funções para operar com datas, funções que calculem valores correspondentes a funções matemáticas e/ou financeiras não fornecidas pelo Clipper etc.

Recomendamos que você dê ênfase maior ao desenvolvimento de funções em C para serem usadas com o Clipper dentro deste grupo de funções.

Capítulo 5

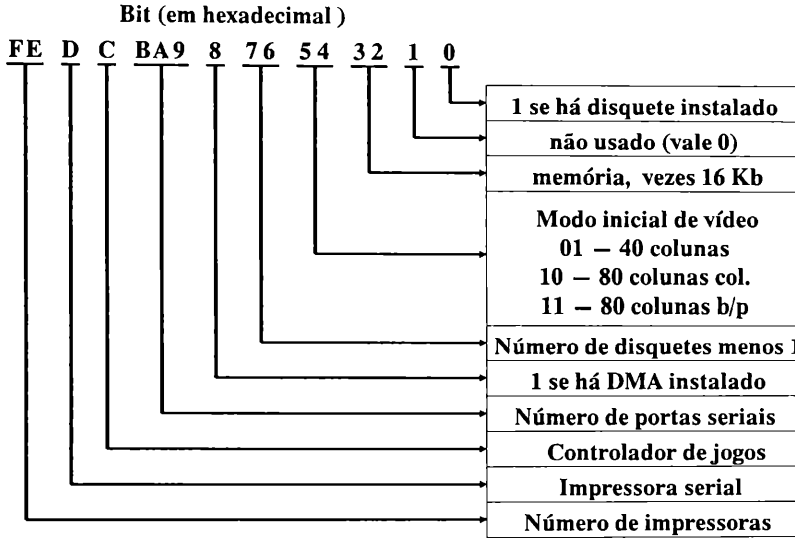
Algumas aplicações práticas

Este último capítulo apresentará a você algumas rotinas escritas em C, prontas para serem integradas com programas em Clipper. Estas rotinas servem não só como exemplos do tipo de rotinas que você poderá desenvolver, mas ao mesmo tempo demonstram as técnicas necessárias para construir estas rotinas. Para tanto, os exemplos são apresentados em ordem crescente de complexidade.

Além disso, todos os exemplos de rotinas apresentados constituem-se em rotinas de utilidade prática imediata, podendo ser usadas exatamente na forma apresentada ou introduzindo alterações necessárias para o seu ambiente de programação.

Determinando a configuração do hardware

Esta função é um exemplo de acesso aos recursos da BIOS.



Esta função não recebe nenhum parâmetro e retorna uma string de 16 casas contendo a representação binária de um número de 16 bits, com o significado detalhado pela figura acima. Por exemplo, a string "0101001001111101" indica um equipamento com uma impressora paralela, uma porta serial, dois disquetes e modo de vídeo inicial de 80 x 25 branco e preto.

Eis a função **bios_equip()**:

```
#include <bios.h>

#include "\clipper\nandef.h"
#include "\clipper\extend.h"

CLIPPER bios_equip()
/*     Retorna string contendo os bits da lista de equipamentos da BIOS
 */
{
       static char BIOS_STRING[17];
       unsigned BIOS_EQUIP = _bios_equiplist();
            /*    Chamada à BIOS para obter o valor numérico */
       register int I;
```

```
        /*   Constrói string contendo dígitos binários */
        for (I = 0; I < 16; I++) {
             BIOS_STRING[15 - I] = (BIOS_EQUIP & 1) + '0';
             BIOS_EQUIP >>= 1;
        }
        BIOS_STRING[16] = '\0';

        /*   Retorna o string para o Clipper */
        _retc(BIOS_STRING);
}
```

Eis aqui um pequeno programa (pequeno mesmo) em Clipper que usa a função acima:

```
* Imprime a lista de equipamentos de acordo com os valores da BIOS

? BIOS_EQUIP()
QUIT
```

Na prática, você iria isolar um trecho da string para determinar a informação necessária, como, por exemplo, verificar se existe uma impressora instalada.

A criação de diretórios

Este é um exemplo de uma função do MS-DOS que está disponível no Clipper apenas indiretamente, através do uso do comando **RUN**. Esta função recebe como parâmetro um nome de diretório a ser criado e retorna um valor lógico, onde **.T.** é usado para indicar que o diretório foi criado e **.F.** para o caso contrário.

Eis a função **mkdir()**:

```
#include <direct.h>
#include <string.h>

#include "\clipper\nandef.h"
#include "\clipper\extend.h"

CLIPPER make_dir()
/*     Cria diretório com nome contido no string recebido como parâmetro
       Retorna .T. ou .F., indicando sucesso ou não na criação
 */
{
```

```
        /*    Se não há um parâmetro ou ele não é uma string */
        if (PCOUNT != 1 || !ISCHAR(1))
              _retl(0);
        /*    Parâmetro correto: */
        else {
              static char DIRETORIO[128];

              /*    Armazena o nome em variável local */
              strcpy(DIRETORIO, _parc(1));

              /*    String tem comprimento maior ao permitido? */
              if (strlen(DIRETORIO) > 63)
                    /*    Retorna valor indicando erro */
                    _retl(0);
              else
                    /*    Retorna resultado da criação para o Clipper */
                    _retl(mkdir(DIRETORIO) + 1);
        }
}
```

Eis aqui um pequeno programa em Clipper que usa a função acima:

```
* Recebe string da linha de comando contendo nome de diretório
* a ser criado chamando a rotina make_dir escrita em C

PARAMETERS DIRETORIO

IF PCOUNT() = 1
      IF make_dir(DIRETORIO)
            ? DIRETORIO + " criado corretamente"
      ELSE
            ? "Nao é possível criar " + DIRETORIO
      ENDIF
ELSE
      ? "Chame o programa com o nome do diretório a ser criado na linha"
      ? "de comando. Em caso de erro, você será informado."
ENDIF

QUIT
```

Adição de valores monetários

A representação de números com casas decimais é efetuada pelo Clipper usando internamente um formato equivalente à notação

científica binária. Assim, os valores monetários não têm representação exata, o que leva a erros de arredondamento quando você soma um número grande de valores.

Vamos desenvolver aqui uma função em C que recebe como parâmetro duas strings, contendo os números a serem somados, e retorna uma string contendo o resultado da soma. Por exemplo, se a função recebe **"1234.04"** e **"417.11"**, deve retornar **"1651.15"**. Note que as strings devem conter números com duas casas decimais.

Eis a função **money_add()**:

```
#include <string.h>

#include "\clipper\nandef.h"
#include "\clipper\extend.h"

CLIPPER money_add()
/*     Retorna string com soma de duas strings contendo valores mone-
       tários (duas casas decimais precedidas por um ponto decimal)
       Retorna string vazia em caso de erro
 */
{
       static char RESULTADO[128];

       /*   Se não há dois parâmetros ou um deles não é uma string */
       if (PCOUNT != 2 || !ISCHAR(1) || !ISCHAR(2)) {
            /*   Retorna valor indicando erro */
            RESULTADO[0] = '\0';
            _retc(RESULTADO);
       }
       /*   Parâmetros corretos: */
       else {
            static char VALOR1[128], VALOR2[128];
            int LEN1, LEN2;
            int VAI_UM;
            register int I;
            char *PONTO;

            /*   Armazena 1º número em variável local */
            strcpy(VALOR1, _parc(1));
            /*   Armazena 2º número em variável local */
            strcpy(VALOR2, _parc(2));

            LEN1 = strlen(VALOR1);
            LEN2 = strlen(VALOR2);
```

```
/*    Uma das strings tem comprimento maior ao permitido
      ou uma delas não possui duas casas decimais ? */
if (LEN1 > 127 || VALOR1[LEN1 - 3] != '.'
||  LEN2 > 127 || VALOR2[LEN2 - 3] != '.') {
      /*    Retorna valor indicando erro */
      RESULTADO[0] = '\0';
      _retc(RESULTADO);
      return;
}

/*    Inverte as strings contendo ambos os números */
strrev(VALOR1);
strrev(VALOR2);

/*    Primeira casa não tem "vai um" da anterior */
VAI_UM = 0;
/*    Para cada posição em que há dígitos em um dos valores */
for (I = 0; I < LEN1 || I < LEN2; I++)
      /*    Posição do ponto decimal ? */
      if (I == 2)
            /*    Coloca ponto decimal no resultado */
            RESULTADO[I] = '.';
      /*    Posição contendo dígito(s) */
      else {
            /*    Determina dígito do primeiro valor */
            int DIG1 = I < LEN1 ? VALOR1[I] - '0' : 0;
            /*    Determina dígito do segundo valor */
            int DIG2 = I < LEN2 ? VALOR2[I] - '0' : 0;
            /*    Determina dígito do resultado */
            int DIG_RES = DIG1 + DIG2 + VAI_UM;
            /*    Haverá "vai um"? */
            if (VAI_UM = (DIG_RES > 9))
                  /*    Tira 10 do dígito do resultado */
                  DIG_RES -= 10;
            /*    Insere caractere para o dígito no resultado */
            RESULTADO[I] = '0' + DIG_RES;
      }

/*    Há "vai um" da última casa? */
if (VAI_UM)
      /*    Coloque um dígito um no número */
      RESULTADO[I++] = '1';
/*    Coloca terminador na string do resultado */
RESULTADO[I] = '\0';

/*    Inverte a string resultado para conter número normal */
```

```
            strrev(RESULTADO);

            /*   Retorna resultado para o Clipper */
            _retc(RESULTADO);
        }
}
```

Eis aqui um pequeno programa em Clipper que permite testar a função acima:

```
* Recebe dois números (valores monetários) da linha de comando
* Imprime a sua soma usando a rotina money_add escrita em C

PARAMETERS STR1, STR2

IF PCOUNT() = 2
      ? STR1 + "+" + STR2 + "=" + money_add(STR1, STR2)
ELSE
      ? "Chame o programa com 2 valores monetários na linha de comando"
      ? "Em caso de erro, o resultado será a string vazia"
ENDIF

QUIT
```

Geração de valores monetários por extenso

Esta função recebe uma string como parâmetro, que contém um valor monetário, e retorna uma string contendo a especificação do valor por extenso. Por exemplo, se a função recebe a string **"1651.15"**, ela deve retornar a string **"um mil, seiscentos e cinqüenta e um cruzados novos e quinze centavos"**.

Eis a função **extenso()**:

```
#define COM_CLIPPER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#if defined(COM_CLIPPER)
      #include "\clipper\nandef.h"
      #include "\clipper\extend.h"
#endif
```

```
static const char *const EXT_PAR[8][11] = {

      { "zero", "um", "dois", "três", "quatro", "cinco",
        "seis", "sete","oito", "nove", "" },

      { "dez", "onze", "doze", "treze", "quatorze", "quinze",
        "dezesseis","dezessete", "dezoito", "dezenove", "" },

      { "vinte", "vinte e um", "vinte e dois", "vinte e três",
        "vinte e quatro", "vinte e cinco", "vinte e seis",
        "vinte e sete","vinte e oito", "vinte e nove", "" },

      { "", "dez", "vinte", "trinta", "quarenta", "cinqüenta",
        "sessenta","setenta", "oitenta", "noventa", " e " },

      { "cem", "cento", "duzentos", "trezentos", "quatrocentos",
        "quinhentos", "seiscentos", "setecentos", "oitocentos",
        "novecentos", " e " },

      { "", " mil", " milhão", " bilhão", " trilhão",
        " quatrilhão", " quintilhão", " sextilhão", " septilhão",
        " octilhão", " nonilhão" },

      { "", " mil", " milhões", " bilhões", " trilhões",
        " quatrilhões", " quintilhões", " sextilhões", " septilhões",
        " octilhões", " nonilhões" },

      { ", ", " cruzado novo", " cruzados novos", " de cruzados novos",
        " centavo", " centavos" }
};

static int ORDEM_DIG[3] = { 2, 1, 0 };
      /*   Indica a ordem da composição dos dígitos:
                2 = centena;  1 = dezena;  0 = unidade
       */

static char EXTENSO[512];
      /*   String para conter valor por extenso */

int terna_extenso(char CENTENA, char DEZENA, char UNIDADE);

#if defined(COM_CLIPPER)
      CLIPPER extenso()
#else
      void extenso(char *VALOR_MON)
#endif
/*    Recebe string contendo um valor monetário
```

```
        como seqüência de dígitos
        Retorna string contendo o mesmo valor escrito por extenso
        Retorna string vazia em caso de erro
 */
{
        char CENTS[3];        /*    Centavos do valor */
        int NUM_TERNAS_NAO_NULAS = 0;
        register int I;
        int LEN, FATOR;
        char VALOR[35];

        EXTENSO[0] = '\0';

#if defined(COM_CLIPPER)

        /*    Não há um parâmetro ou não é uma string do tamanho certo? */
        if (PCOUNT != 1 || !ISCHAR(1) || strlen(_parc(1)) > 34) {
             /*    Retorna valor indicando erro */
             _retc(EXTENSO);
             return;
        }
        else {
             LEN = strlen(_parc(1));
             /*    Armazena o número em variável local */
             strcpy(VALOR, _parc(1));
        }
#else
        if (strlen(VALOR) > 34)
             return;
        else {
             strcpy(VALOR, VALOR_MON);
             LEN = strlen(VALOR);
        }
#endif

        /*    Parâmetro válido: */

        /*    Separa os centavos, se houver */
        CENTS[0] = '0';
        if (VALOR[LEN - 3] == '.') {
             CENTS[1] = VALOR[LEN - 2];
             CENTS[2] = VALOR[LEN - 1];
             VALOR[LEN -= 3] = '\0';
        }
        else
             CENTS[1] = CENTS[2] = '0';
```

```
/*   Completa a parte inteira para ter número exato de ternas */
strrev(VALOR);
switch (LEN % 3) {
     case 1:    VALOR[LEN++] = '0';
     case 2:    VALOR[LEN++] = '0';
                VALOR[LEN]   = '\0';
}
strrev(VALOR);

/*   Para cada terna de números */
for (I = 0, FATOR = LEN / 3 - 1; I < LEN; I += 3, FATOR--) {
     /*   Gera valor por extenso para a terna */
     int TERNA =
          terna_extenso(VALOR[I], VALOR[I + 1], VALOR[I + 2]);

     NUM_TERNAS_NAO_NULAS += TERNA > 0;
     if (TERNA) {
          char *PROX_TERNA = VALOR + I + 3;

          /*   Insere o fator */
          strcat(EXTENSO, EXT_PAR[TERNA == 1 ? 5 : 6][FATOR]);
          /*   Procura próxima terna não nula */
          while (*PROX_TERNA && !strncmp(PROX_TERNA, "000", 3))
               PROX_TERNA += 3;
          /*   Achou terna não nula? */
          if (*PROX_TERNA)
               /*   A próxima é a última (significativa),
                    e vale menos de 100? */
               if ((PROX_TERNA[3] == '\0'
                    || !strpbrk(PROX_TERNA + 3, "123456789"))
               && strncmp(PROX_TERNA, "100", 3) <= 0)
                    /*   Concatena " e " */
                    strcat(EXTENSO, EXT_PAR[4][10]);
               else
                    /*   Concatena ", " */
                    strcat(EXTENSO, EXT_PAR[7][0]);
     }
}

/*   Inclui unidade monetária da parte inteira */
if (LEN > 6 && !strcmp(VALOR + LEN - 6, "000000"))
     /*   De plural */
     strcat(EXTENSO, EXT_PAR[7][3]);
else if (NUM_TERNAS_NAO_NULAS == 1
&& !strcmp(VALOR + LEN - 3, "001"))
     /*   Singular */
     strcat(EXTENSO, EXT_PAR[7][1]);
```

```
        else if (NUM_TERNAS_NAO_NULAS)
             /*   Plural */
             strcat(EXTENSO, EXT_PAR[7][2]);

        /*   Se houver, inclui os centavos no valor por extenso */
        if (CENTS[2] > '0' || CENTS[1] > '0' || CENTS[0] > '0') {
             /*   Se houve parte inteira */
             if (EXTENSO[0])
                  /*   Inclui Separador */
                  if (CENTS[1] < '2' || CENTS[2] == '0')
                       /*   < 10, separa com " e " */
                       strcat(EXTENSO, EXT_PAR[4][10]);
                  else
                       /*   Separa com ", " */
                       strcat(EXTENSO, EXT_PAR[7][0]);
             /*   Gera centavos por extenso, um só ? */
             if (terna_extenso(CENTS[0], CENTS[1], CENTS[2]) == 1)
                  /*   Singular */
                  strcat(EXTENSO, EXT_PAR[7][4]);
             else
                  /*   Plural */
                  strcat(EXTENSO, EXT_PAR[7][5]);
        }

#if defined(COM_CLIPPER)
        /*   Retorna resultado para o Clipper */
        _retc(EXTENSO);
#else
        /*   Imprime resultado para testes */
        puts(EXTENSO);
#endif
}


int terna_extenso(char CENTENA, char DEZENA, char UNIDADE)
/*      Recebe os três algarismos de uma terna
        Concatena o valor da terna a string EXTENSO
        Exemplo:  CENTENA = '2'  DEZENA = '4'  UNIDADE = '7'
                          concatena      "duzentos e quarenta e sete"
        Retorna o valor da terna
 */
{
        char *TEXTOS[3][2];
        int  DIG1 = CENTENA - '0',
             DIG2 = DEZENA - '0',
             DIG3 = UNIDADE - '0';
        register int I, J;
```

```
        if (DIG1 == 1 && DIG2 == 0 && DIG3 == 0) {
             strcat(EXTENSO, EXT_PAR[4][0]);
             return 100;
        }

        TEXTOS[2][0] = DIG1 > 0 ?
                              EXT_PAR[4][DIG1] :
                              NULL;
        TEXTOS[2][1] = DIG1 > 0 && (DIG2 > 0 || DIG3 > 0) ?
                              EXT_PAR[4][10] :
                              NULL;

        TEXTOS[1][0] = DIG2 > 0 ?
                              (DIG2 < 3 ?
                                   EXT_PAR[DIG2][DIG3] :
                                   EXT_PAR[3][DIG2]
                              ) :
                              NULL;
        TEXTOS[1][1] = DIG2 > 0 ?
                              (DIG2 < 3 ?
                                   EXT_PAR[DIG2][10] :
                                   (DIG3 > 0 ?
                                        EXT_PAR[3][10] :
                                        NULL
                                   )
                              ) :
                              NULL;

        TEXTOS[0][0] = (DIG2 == 0 || DIG2 >= 3) && DIG3 > 0 ?
                              EXT_PAR[0][DIG3] :
                              NULL;
        TEXTOS[0][1] = NULL;

        for (I = 0; I < 3; I++)
             for (J = 0; J < 2; J++)
                  if (TEXTOS[ORDEM_DIG[I]][J])
                       strcat(EXTENSO, TEXTOS[ORDEM_DIG[I]][J]);

        return DIG1 * 100 + DIG2 * 10 + DIG3;
}


#if !defined(COM_CLIPPER)
int main()
{
        char S[48];
        do {
```

```
            fputs("Digite valor monetário: ", stdout);
            gets(S);
            if (*S)
                extenso(S);
        } while (*S);

        return 0;
}
#endif
```

Eis aqui um pequeno programa em Clipper que permite testar a função acima:

```
* Programa de demonstração da utilização da rotina extenso()

PARAMETERS VALOR

IF PCOUNT() = 0
        ? "Indique o valor monetário na linha de comando"
ELSE
        ? EXTENSO(VALOR)
ENDIF

QUIT
```

Pesquisa de diretórios

Nesta seção apresentamos um conjunto de funções escritas em C, que permitem acessar todas as informações do diretório. Este é um exemplo de uma aplicação onde não é possível trabalhar com uma única função em C.

Todas as funções necessárias estão dentro de um único arquivo fonte, já que não é possível utilizá-las separadamente. A seguir apresentamos todos os arquivos que compõem o código para a implementação e teste das funções de diretório.

O arquivo de **make** para **director.exe** é:

```
#director - arquivo de make para director.exe

#regras de inferência

.c.obj:         # de C   para OBJ
```

```
        cl -c -AL -W3 -Zl -Oalt -FPa -Gs -DCOM_CLIPPER $*.c
.prg.obj:       # de PRG para OBJ
        clipper $*.prg -m

.obj.exe:       # de OBJ para EXE
        link @$*.lnk

director.obj: director.c

direcclp.obj: direcclp.prg

director.exe: director.obj direcclp.obj
```

Note a definição do símbolo **COM_CLIPPER** nas opções de compilação. O arquivo **director.lnk** contém as opções para a link-edição:

```
director direcclp
director
nul
\msc\lib\llibca \clipper\clipper \clipper\extend /NOE
```

Observe que este programa precisa, além das bibliotecas habituais, da presença da biblioteca **extend.lib**. Você pode incluir qualquer número de bibliotecas e/ou arquivos objeto, desde que haja memória suficiente para a execução do programa gerado.

A seguir apresentamos o arquivo fonte **director.c**, que contém as definições das funções **fileCount()**, **fileFirst()**, **fileNext()**, **fileSize()**, **fileAttrib()**, **fileDate()** e **fileTime()**, que retornam, respectivamente, o número de arquivos que satisfazem uma determinada chave de pesquisa no diretório, o nome do primeiro arquivo que satisfaz uma determinada pesquisa, o nome do arquivo seguinte de uma pesquisa e o tamanho, atributos, data e hora da última alteração do arquivo, cujo nome é o último obtido através de uma chamada a **fileFirst()** ou **fileNext()**. Veja a listagem a seguir para obter os detalhes dos parâmetros e valores de retorno destas funções:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <dos.h>

#if defined(COM_CLIPPER)
```

```
        #include "\clipper\nandef.h"
        #include "\clipper\extend.h"

        CLIPPER fileCount(void);
        CLIPPER fileFirst(void);
        CLIPPER fileNext(void);
        CLIPPER fileAttrib(void);
        CLIPPER fileTime(void);
        CLIPPER fileDate(void);
        CLIPPER fileSize(void);
#else
        int   fileCount(char *PATH);
        char *fileFirst(char *PATH);
        char *fileNext(void);
        char *fileAttrib(void);
        char *fileTime(void);
        char *fileDate(void);
        long  fileSize(void);
        int main(void);
#endif

/*      Símbolo para detecção de arquivos, de quaisquer atributos */
#define _A_ALL _A_NORMAL | _A_RDONLY | _A_HIDDEN | _A_SYSTEM \
                        | _A_VOLID  | _A_SUBDIR | _A_ARCH

/*      Buffer para armazenamento dos dados dos arquivos achados */
static struct find_t Buffer;

#if defined(COM_CLIPPER)
        CLIPPER fileCount()
#else
        int fileCount(char *PATH)
#endif
/*      Recebe string indicando chave de pesquisa
        Retorna quantidade de arquivos que satisfazem a chave
 */
{
        static int CONTAGEM;

#if defined(COM_CLIPPER)
        char *PATH;

        /*    Obtém parâmetro, se correto */
        if (PCOUNT == 1 && ISCHAR(1))

             PATH = _parc(1);
        else
```

```
            PATH = NULL;
#endif

       CONTAGEM = 0;

       if (PATH)
            if (_dos_findfirst(PATH, _A_ALL, &Buffer) == 0)
                 do
                      CONTAGEM++;
                 while(_dos_findnext(&Buffer) == 0);

#if defined(COM_CLIPPER)
       _retni(CONTAGEM);
       return;
#else
       return CONTAGEM;
#endif
}


#if defined(COM_CLIPPER)
       CLIPPER fileFirst()
#else
       char *fileFirst(char *PATH)
#endif
/*
       Retorna string contendo nome do primeiro arquivo achado
       Retorna string vazia quando não existe arquivo
 */
{
#if defined(COM_CLIPPER)
       char *PATH;

       /*   Obtém parâmetro, se correto */
       if (PCOUNT == 1 && ISCHAR(1))
            PATH = _parc(1);
       else
            PATH = NULL;
#endif

       /*   Se não há parâmetro, ou não acha arquivo */
       if (!PATH || _dos_findfirst(PATH, _A_ALL, &Buffer) != 0)
            /*   Devolve string vazia */
            Buffer.name[0] = '\0';

       /*   Devolve nome do arquivo achado */
#if defined(COM_CLIPPER)
```

```
        _retc(Buffer.name);
        return;
#else
        return Buffer.name;
#endif
}


#if defined(COM_CLIPPER)
        CLIPPER fileNext()
#else
        char *fileNext()
#endif

/*
        Retorna string contendo nome do próximo arquivo achado
        Retorna string vazia quando não existem mais arquivos
 */

{
        /*    Se não acha arquivo */
        if (_dos_findnext(&Buffer) != 0)
             /*    Devolve string vazia */
             Buffer.name[0] = '\0';


        /*    Devolve nome do arquivo achado */
#if defined(COM_CLIPPER)
        _retc(Buffer.name);
        return;
#else
        return Buffer.name;
#endif
}


#if defined(COM_CLIPPER)
        CLIPPER fileAttrib()
#else
        char *fileAttrib()
#endif


/*
        Retorna string contendo iniciais dos atributos
```

```
        do último arquivo do qual foi solicitado o nome
             A   atributo de arquivo não arquivado com BACKUP
             D   atributo de arquivo contendo subdiretório
             S   atributo de arquivo do sistema operacional
             H   atributo de arquivo escondido
             R   atributo de arquivo só para leitura
             L   atributo de rótulo de disco
*/
{
        /*   String de identificação dos atributos */
        static char ATTRIBS[7];
        register int I = 0;

        /*   Se bit do atributo de arquivamento ligado */
        if (Buffer.attrib & _A_ARCH)
             /*   Inclui A na string de atributos */
             ATTRIBS[I++] = 'A';

        /*   Se bit do atributo de subdiretório ligado */
        if (Buffer.attrib & _A_SUBDIR)
             /*   Inclui D na string de atributos */
             ATTRIBS[I++] = 'D';

        /*   Se bit do atributo de arquivo de sistema ligado */
        if (Buffer.attrib & _A_SYSTEM)
             /*   Inclui S na string de atributos */
             ATTRIBS[I++] = 'S';

        /*   Se bit do atributo de arquivo escondido ligado */
        if (Buffer.attrib & _A_HIDDEN)
             /*   Inclui H na string de atributos */
             ATTRIBS[I++] = 'H';

        /*   Se bit do atributo de arquivo só para leitura ligado */
        if (Buffer.attrib & _A_RDONLY)
             /*   Inclui R na string de atributos */
             ATTRIBS[I++] = 'R';

        /*   Se bit do atributo de rótulo de disco ligado */
        if (Buffer.attrib & _A_VOLID)
             /*   Inclui L na string de atributos */
             ATTRIBS[I++] = 'L';

        /*   Encerra string de atributos */
        ATTRIBS[I] = '\0';

        /*   Devolve string de atributos do arquivo */
```

```
#if defined(COM_CLIPPER)
      _retc(ATTRIBS);
      return;
#else
      return ATTRIBS;
#endif
}


#if defined(COM_CLIPPER)
      CLIPPER fileTime()
#else
      char *fileTime()
#endif
/*
      Retorna string contendo o horário da última alteração do
      último arquivo cujo nome foi obtido, no formato HH:MM:SS
 */
{
      /*   String para armazenar a hora do arquivo */
      static char HORARIO[9];

      /*   Descompactação do horário */
      unsigned HORAS    = Buffer.wr_time >> 11;
      unsigned MINUTOS  = Buffer.wr_time >> 5 & 63;
      unsigned SEGUNDOS = (Buffer.wr_time & 31) * 2;

      /*   Inicializa string no formato HH:MM:SS */
      sprintf(HORARIO, "%02u:%02u:%02u", HORAS, MINUTOS, SEGUNDOS);

      /*   Devolve o horário */
#if defined(COM_CLIPPER)
      _retc(HORARIO);
      return;
#else
      return HORARIO;
#endif
}


#if defined(COM_CLIPPER)
      CLIPPER fileDate()
#else
      char *fileDate()
#endif
/*
      Retorna a data da última alteração do último arquivo cujo
```

```
      nome foi obtido, no formato HH:MM:SS
 */
{
      /*    String para armazenar a data */
      static char DATA[11];

      /*    Descompactação da data */
      unsigned ANO = (Buffer.wr_date >> 9) + 1980;
      unsigned MES = Buffer.wr_date >> 5 & 15;
      unsigned DIA = Buffer.wr_date & 31;

#if defined(COM_CLIPPER)
      /*    Inicializa string no formato AAAAMMDD */
      sprintf(DATA, "%04u%02u%02u", ANO, MES, DIA);
      /*    Devolve ao Clipper como tipo data */
      _retds(DATA);
      return;
#else
      /*    Inicializa string no formato DD/MM/AAAA */
      sprintf(DATA, "%02u/%02u/%04u", DIA, MES, ANO);
      /*    Devolve como string */
      return DATA;
#endif
}


#if defined(COM_CLIPPER)
      CLIPPER fileSize()
#else
      long fileSize()
#endif
/*
      Retorna número inteiro indicando o comprimento do arquivo em bytes
      do último arquivo cujo nome foi obtido
 */
{
      /*    Devolve tamanho do arquivo */
#if defined(COM_CLIPPER)
      _retnl(Buffer.size);
      return;
#else
      return Buffer.size;
#endif
}


#if !defined(COM_CLIPPER)
```

```
int main()
{
      char SEARCH_PATH[63];

      do {
            fputs("Digite chave de pesquisa no diretório: ", stdout);
            gets(SEARCH_PATH);
            if (*SEARCH_PATH) {
                  char *FILE_NAME;
                  register int I = fileCount(SEARCH_PATH), J;

                  if (I > 0) {
                        printf(
                                "   |Nome arquivo|Tamanho |Atribs|"
                                                  "Data/hora alteração\n"
                                "---+------------+--------+------+"
                                                  "-------------------\n");
                        FILE_NAME = fileFirst(SEARCH_PATH);
                        printf("  1|%-12.12s|%8ld|%-6.6s|%10.10s %8.8s\n",
                                    FILE_NAME, fileSize(), fileAttrib(),
                                    fileDate(), fileTime());
                  }
                  for (J = 2; J <= I; J++) {
                        FILE_NAME = fileNext();
                        printf("%3i|%-12.12s|%8ld|%-6.6s|%10.10s %8.8s\n",
                                    J, FILE_NAME, fileSize(), fileAttrib(),
                                    fileDate(), fileTime());
                  }
            }
      } while (*SEARCH_PATH);

      return 0;
}
#endif
```

Note que as funções **fileSize()**, **fileAttrib()**, **fileDate()** e **fileTime()** devolvem a informação correspondente ao arquivo cujo nome foi obtido por último através de uma chamada a **fileFirst()** ou **fileNext()**. Essas quatro funções e **fileNext()** devem ser chamadas sem nenhum parâmetro.

O programa Clipper **direcclp.prg**, para testar as funções de diretório, recebe como parâmetro da linha de comando a chave de pesquisa de diretório a ser passada às funções em C:

```
* Programa de teste das rotinas de diretório
PARAMETERS CHAVE
DECLARE QUANT, ARQUIVO, K

CLEAR SCREEN
SET DATE BRITISH

IF PCOUNT() < 1
      ? "Indique chave de pesquisa na linha de comando"
      QUIT
ENDIF

QUANT = fileCount(CHAVE)
? ""
? STR(QUANT,3) + " arquivo(s) de chave " + CHAVE
IF QUANT > 0
      ? "   |Nome arquivo|Tamanho |Atribs|Data/Hora ult.alt"
      ? "---+------------+--------+------+--------+--------"
      ARQUIVO = fileFirst(CHAVE)
      DO DISPLAY WITH 1, ARQUIVO, fileSize(), fileAttrib(), ;
                                                  fileDate(), fileTime()
      FOR K = 2 TO QUANT
            ARQUIVO = fileNext()
            DO DISPLAY WITH K, ARQUIVO, fileSize(), fileAttrib(), ;
                                                  fileDate(), fileTime()
            IF K % 18 == 0
                  DECLARE C

                  ? ""
                  ? "Aperte qualquer tecla para continuar..."
                  C = INKEY(0)
                  CLEAR SCREEN
                  ? ""
                  ? ""
                  ? "   |Nome arquivo|Tamanho |Atribs|Data/Hora ult.alt"
                  ? "---+------------+--------+------+--------+--------"
            ENDIF
      NEXT
ENDIF

QUIT


PROCEDURE DISPLAY
PARAMETERS N, ARQ, TAM, ATR, DATA, HORA

?  STR(N, 3)                    + "|"
```

```
?? LEFT(ARQ + "              ", 12) + "|"
?? STR(TAM, 8)                      + "|"
?? LEFT(ATR + "      ", 6)          + "|"
?? DTOC(DATA) + "|" + HORA

RETURN
```

Note a presença do procedimento **DISPLAY**, escrito em Clipper. Não existe nenhuma limitação quanto ao número de procedimentos e/ou funções escritas em Clipper que você pode incluir no seu programa que chama rotinas escritas em C.

Exercícios propostos

Nesta última seção propomos que você mesmo inicie o desenvolvimento de funções em C.

Como primeiro exercício, sugerimos que você modifique os exemplos dados acima, para torná-los mais robustos. Por exemplo, modifique a função **money_add()** de forma a somar corretamente os dois números, independentemente do número de casas decimais ou modifique a função **extenso()** de forma a operar corretamente quando receber de zero a três casas após o ponto decimal. Ainda você pode modificar estas funções de forma a produzirem a resposta correta mesmo no caso de serem chamadas com valores precedidos ou sucedidos por qualquer número de caracteres brancos.

Depois disto, você deve começar a desenvolver funções novas por conta própria. Para auxiliar você no caso da sua imaginação não ser muito fértil, relacionamos a seguir um número razoável de idéias, que você pode usar para este fim:

a) Desenvolva funções para subtrair, multiplicar, dividir, exponenciar e calcular percentuais sobre valores monetários, usando a mesma interface da função de adição mostrada no primeiro exemplo deste capítulo.

b) Desenvolva uma função que gera datas por extenso, isto é, receba como parâmetro uma variável do tipo data e retorne uma string com a data por extenso. Por exemplo, se a data recebida for **19890731**, a função deve retornar uma string contendo **31 de julho de 1989**. (Uma possível variação desta função é retornar a data por extenso

incluindo o nome do dia da semana, por exemplo **Segunda, 31 de julho de 1989**, ou ainda **aos trinta e um dias do mês de julho do ano de mil novecentos e oitenta e nove**.)

c) Desenvolva uma função que calcule a quantidade de dias entre duas datas, isto é, receba como parâmetro duas variáveis do tipo data e retorne um número inteiro indicando a quantidade de dias entre as duas datas. Por exemplo, se as datas recebidas forem **19890101 e 19910401**, a função deve retornar o inteiro **830**.

d) Desenvolva uma função que calcule datas a partir de prazos, isto é, receba como parâmetro uma variável do tipo data e um número inteiro e retorne a data correspondente ao prazo indicado pelo número de dias, a partir da data recebida. Por exemplo, se os dados recebidos forem **19890101** e **830**, a função deve retornar a data **19910401**. (Cuide para que a função também devolva a data correta caso o prazo seja negativo.)

e) Desenvolva uma função que verifique a validade de uma data, isto é, receba como parâmetro uma variável do tipo data como string, e retorne **.F.** ou **.T.**, conforme a data seja correta ou não. Por exemplo, a data **19890931** não é correta.

f) Desenvolva uma função que calcule a taxa interna de retorno de uma seqüência de operações financeiras, isto é, receba a data base do processo e duas matrizes contendo, respectivamente, a data e o valor de cada lançamento, e retorne a taxa interna de retorno do fluxo de caixa representado por esses valores (note que pode haver valores negativos).

g) Desenvolva uma função que crie uma string a ser usada com o comando **PICTURE** a partir dos dados de um campo de arquivo, isto é, receba como parâmetro o comprimento do campo, o seu tipo e o número de casas decimais, e retorne uma string pronta para ser usada como valor junto com o comando **PICTURE**. Por exemplo, se os dados recebidos forem 9, N e 2, a função deve retornar a string *#######.##*. (Uma variação desta função poderia produzir a string de PICTURE apenas para números usados como valores monetários, colocando os pontos a cada três casas e a vírgula decimal; por exemplo, com os dados acima a função retornaria a string @E9.999.999,99.)

h) Desenvolva um conjunto de funções para manipular horários. A representação dos horários pode ser feita através de strings no

formato "HH:MM:SS" ou "HH:MM:SS.CC", onde *H* indica horas, *M* minutos, *S* segundos e *C* centésimos de segundo. Uma primeira função poderia verificar a validade de um horário (para evitar "14:75:99" como horário válido). Depois você poderia implementar funções que permitissem efetuar operações aritméticas com os horários. Por exemplo, uma função poderia obter a soma de dois horários e outra a diferença entre dois horários. Ainda, você poderia implementar a multiplicação de um horário por um número (se forem só inteiros é mais fácil) retornando o horário correspondente àquele horário repetido o número de vezes dado, e a divisão de um horário por um número retornando a fração correspondente do horário.

Apêndice

Referência rápida às funções da interface

Descrição do formato de apresentação

Neste apêndice apresentamos todas as funções e definições parametrizadas (macros) que podem ser usadas para escrever rotinas em C para o Clipper, em ordem alfabética, obedecendo o seguinte formato:

```
Protótipo da função
```

Descrição do significado da função, seus parâmetros e valor de retorno

Não esqueça de que toda rotina escrita em C para ser chamada desde um programa escrito em Clipper deve conter estes dois **#includes**:

```
#include "\clipper\nandef.h"
#include "\clipper\extend.h"
```

onde você deve substituir **clipper** pelo nome do diretório onde você instalou o Clipper no seu equipamento (se não chamar **clipper**, ou se não estiver contido no diretório raiz).

Relação alfabética das funções

```
int ALENGTH(int ORDEM);
```

Macro que retorna um número inteiro, indicando o número de elementos da matriz passada como **ORDEM**-ésimo parâmetro. Você deve garantir antes que o parâmetro realmente seja uma matriz. Esta macro é equivalente a efetuar uma chamada à função **_parinfa()** com parâmetros **ORDEM** e zero.

```
void _exmback(void *AREA, unsigned int TAMANHO);
```

Função que deve ser utilizada para retornar para o sistema de gerenciamento da memória do Clipper uma área de memória alocada dinamicamente (antes) através do uso da função **_exmgrab()**. O endereço **AREA** e o **TAMANHO** devem ser os mesmos utilizados e/ou obtidos com a função **_exmgrab()**. Esta função deve ser usada no lugar da função **free()** da biblioteca do compilador C.

```
void *_exmgrab(unsigned int TAMANHO);
```

Função que deve ser usada para solicitar ao sistema de gerenciamento da memória do Clipper para tentar alocar dinamicamente uma área de memória de **TAMANHO** bytes. Se a função conseguir alocar uma área do **TAMANHO** solicitado, ela retorna o endereço dessa área, caso contrário ela retorna **NULL**. Esta função deve ser usada no lugar da função **malloc()** da biblioteca do compilador C.

```
int ISARRAY(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo matriz ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor obtido com a máscara **ARRAY**.

```
int ISBYREF(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução tenha sido passado por referência (usando o símbolo @ na frente do nome da variável usada como parâmetro) ou não,

respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor obtido com a máscara **MPTR**.

```
int ISCHAR(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo string ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor com a máscara **CHARACTER**.

```
int ISDATE(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo data ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor com a máscara **DATE**.

```
int ISLOG(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo lógico ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor com a máscara **LOGICAL**.

```
int ISMEMO(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo memo ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor com a máscara **MEMO**.

```
int ISNUM(int ORDEM);
```

Macro que retorna verdadeiro ou falso, conforme o **ORDEM**-ésimo parâmetro passado para a função escrita em C durante a sua execução seja do tipo número ou não, respectivamente. Esta macro é equivalente a efetuar uma chamada à função **_parinfo()**, com parâmetro **ORDEM** é testar o valor com a máscara **NUMERIC**.

```
char *_parc(int ORDEM, int INDICE);
```

Função que retorna o endereço do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo string (isto deve ter sido verificado previamente). Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é uma string, indique o índice da string na matriz como segundo parâmetro de **_parc()**. Note que caso o string acessado não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int _parclen(int ORDEM, int INDICE);
```

Função que retorna o comprimento do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo string (isto deve ter sido verificado previamente). O comprimento da string é obtido contando o número de caracteres que existem antes do caractere terminador **'\0'**. Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é uma string, indique o índice da string na matriz como segundo parâmetro de **_parclen()**. Note que caso a string acessada não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int  _parcsiz(int ORDEM, int INDICE);
```

Função que retorna o tamanho ocupado pelo **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo string (isto deve ter sido verificado previamente). O tamanho da string é a quantidade de memória efetivamente alocada para a sua armazenagem (a função retorna zero se o valor passado corresponde a uma constante). Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é uma string, indique o índice da string na matriz como segundo parâmetro de **_parcsiz()**. Note que caso a string acessada não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
char *pards(int ORDEM, int INDICE);
```

Função que retorna o endereço do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo data (isto deve ter sido verificado previamente). A string apontada tem o formato **AAAAMMDD**. Caso o

parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é uma string, indique o índice da data na matriz como segundo parâmetro de **_pards()**. Note que caso a data acessada não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int parinfa(int ORDEM, int INDICE);
```

Função que retorna o tipo do **INDICE**-ésimo elemento do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo matriz (isto deve ter sido verificado previamente). Estes valores podem ser comparados com as máscaras **UNDEF**, **CHARACTER**, **NUMERIC**, **LOGICAL**, **DATE** e **MEMO** para determinar o tipo do elemento da matriz durante a execução do programa. Note que se **INDICE** valer zero, **_parinfa()** devolve o número de elementos da matriz.

```
int parinfo(int ORDEM);
```

Função que retorna o tipo do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução. Estes valores podem ser comparados com as máscaras **UNDEF**, **CHARACTER**, **NUMERIC**, **LOGICAL**, **DATE**, **MEMO** e **ARRAY** para determinar o tipo do elemento da matriz durante a execução do programa. Note que se **ORDEM** valer zero, **_parinfo()** devolve o número de parâmetros efetivamente passados à rotina em C.

```
int parl(int ORDEM, int INDICE);
```

Função que retorna o valor do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo lógico (isto deve ter sido verificado previamente). O valor retornado é 1 (se foi passado **.T.**) ou 0 (se foi passado **.F.**). Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é um valor lógico, indique o índice do valor lógico na matriz como segundo parâmetro de **_parl()**. Note que caso o valor lógico acessado não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
double parnd(int ORDEM, int INDICE);
```

Função que retorna o valor do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo numérico (isto deve ter sido verificado

previamente). O valor é retornado como um valor do tipo **double** da linguagem C. Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é um valor numérico, indique o índice do número na matriz como segundo parâmetro de **_parnd()**. Note que caso o valor numérico acessado não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int parni(int ORDEM, int INDICE);
```

Função que retorna o valor do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo numérico (isto deve ter sido verificado previamente). O valor é retornado como um valor do tipo **int** da linguagem C. Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é um valor numérico, indique o índice do número na matriz como segundo parâmetro de **_parni()**. Note que caso o número acessado não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int parnl(int ORDEM, int INDICE);
```

Função que retorna o valor do **ORDEM**-ésimo parâmetro efetivamente passado à rotina escrita em C durante a execução, desde que este parâmetro seja do tipo numérico (isto deve ter sido verificado previamente). O valor é retornado como um valor do tipo **long** da linguagem C. Caso o parâmetro seja uma matriz, cujo **INDICE**-ésimo elemento é um valor numérico, indique o índice do número na matriz como segundo parâmetro de **_parni()**. Note que caso o número acessado não seja elemento de uma matriz, o segundo parâmetro deve ser omitido.

```
int PCOUNT;
```

Macro equivalente à chamada de função _parinfo(0). Retorna o número de parâmetros com que a função escrita em C foi efetivamente chamada.

```
void _ret(void);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper que não possui valor de retorno. Isto é útil para criar rotinas em C que possam ser chamadas de programas em Clipper usando o comando **DO/WITH**. Note que esta função não retorna o controle da

execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retc(char *STRING);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo string quando for usada a função **_retc()**). O valor retornado será o conjunto dos caracteres que precedem o terminador **'\0'**, a partir do endereço dado como parâmetro à função **_retc()**. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retclen(char *STRING, int TAMANHO);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo string quando for usada a função **_retclen()**). O valor retornado será formado pelos caracteres a partir do endereço dado como primeiro parâmetro passado a **_retclen()**, em quantidade determinada pelo parâmetro **TAMANHO**. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retds(char *STRING);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo data quando for usada a função **_retds()**). O valor retornado será formado a partir de uma string no formato **AAAAMMDD**, cujo endereço deve ser passado como primeiro parâmetro a **_retds()**. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retl(int FLAG);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo lógico quando for usada a função **_retl()**). O valor retornado será formado a partir de um número inteiro, onde 1 deve ser usado para retornar **.T.**, e 0 para retornar **.F.**. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retnd(double NUMERO);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo numérico quando for usada a função **_retnd()**). O valor retornado será formado a partir de um número do tipo double da linguagem C. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retni(int NUMERO);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo numérico quando for usada a função **_retni()**). O valor retornado será formado a partir de um número do tipo int da linguagem C. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

```
void _retnl(long NUMERO);
```

Função que deve ser usada pela rotina escrita em C para indicar ao Clipper qual é o seu valor de retorno (que será sempre do tipo numérico quando for usada a função **_retnl()**). O valor retornado será formado a partir de um número do tipo long da linguagem C. Note que esta função não retorna o controle da execução ao programa escrito em Clipper; este deve ser retornado através de um **return** normal de C, implícito ou não.

Índice Analítico

CADASTRO PARA MALA DIRETA

Favor preencher todos os campos

NOME (não abreviar):

END. RESIDENCIAL:

CIDADE: UF: CEP: FONE:

EMPRESA:

SEXO: F( ) M( ) NASCIMENTO:____/____.
MÊS ANO

1. Cargo: ( ) 1. Digitador ( ) 2. Programador ( ) 3. Analista ( ) 4. Supervisor ( ) 5. Gerente ( ) 6. Diretor ( ) 7. Outros (especificar):_____

2. Escolaridade: ( ) 1. 1º grau ( ) 2. 2º grau ( ) 3. 3º grau

3. Eu geralmente, compro livros: ( ) 1. Em livrarias ( ) 2. Por telefone ( ) 3. Em feiras ( ) 4. Por Reembolso Postal ( ) 5. Outros (especificar):_____

4. Costumo comprar em média, a seguinte quantidade de livros por ano: ( ) 1. 01 ( ) 2. 03 ( ) 3. 05 ( ) 4. mais

5. Estou interessado em livros das seguintes áreas: ( ) 1. Informática ( ) 2. Negócios ( ) 3. Controle Qualidade ( ) 4. Informática na Educação ( ) 5. Ensino Inglês ( ) 6. Psicologia ( ) 7. Sociologia ( ) 8. Metodologia ( ) 9. Medicina ( ) 10. Eng. Civil ( ) 11. Eng. Elétrica/ Eletrônica ( ) 12. Eng. Mecânica ( ) 13. Eng. Química ( ) 14. Física ( ) 15. Matemática ( ) 16. Outras (especificar):_____

6. Utiliza microcomputador: ( ) Sim ( ) Não

7. Qual? ( ) 1. PC ( ) 2. Apple ( ) 3. MSX ( ) 4. Outros (especificar):_____

8. Softwares mais utilizados: ( ) 1. Banco de Dados ( ) 2. Planilha ( ) 3. Proc. Texto ( ) 4. Outros (especificar):_____

9. Linguagens: ( ) 1. C ( ) 2. Cobol ( ) 3. Clipper/dBase ( ) 4. Assembler ( ) 5. Outras (especificar):_____

Autor: Roberto Carlos Mayer
Obra: Integrando Clipper com Liguagem C

MAKRON *Books*

MAKRON *Books* do Brasil Editora Ltda.
Nova denominação da
Editora McGRAW-HILL Ltda.
"Padrão em Livros de Qualidade"
Rua Tabapuã, 1105 – Itaim Bibi – São Paulo
04533 – Tels.: (011) 829-8604 / 820-8528

Impressão e Acabamento
GRÁFICA E EDITORA FCA
AV. HUMBERTO DE ALENCAR CASTELO BRANCO, 3972 · TEL. 419-0200
SÃO BERNARDO DO CAMPO · CEP 09700 · SP

ROBERTO CARLOS MAYER é professor do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP) e consultor independente na área de software. Tem 10 anos de experiência internacional, adquirida em empresas como Instituto Argentino de Microcomputação, Autom, Mercedes-Benz e Banco Real.

Formado em Matemática na USP, onde realiza atualmente estudos de pós-graduação em Ciência da Computação. Apresentou trabalhos em diversos congressos e simpósios, no Brasil e no exterior.

Dedica-se à formação de profissionais na área de software, através de cursos e seminários, e ao desenvolvimento de ferramentas de software.

Este livro é destinado a todos os usuários do Clipper que desejam ampliar os recursos a sua disposição através do uso da linguagem C. Ao longo dos cinco capítulos do texto, é descrito por completo o processo de integração de rotinas escritas em linguagem C com programas escritos em Clipper. Esta descrição inclui muitas dicas não disponíveis no manual do Clipper, como também diversas rotinas em C já prontas, a título de exemplos. Estas podem ser integradas de imediato com programas escritos em Clipper.

OUTROS LIVROS NA ÁREA

Jamsa	–	*C Library – Bibliotecas*
Mayer	–	*Linguagem C ANSI – Guia do Usuário*
Schildt	–	*Linguagem C – Guia do Usuário*
Schildt	–	*C Avançado – Guia do Usuário*
Schildt	–	*Linguagem C – Guia Prático e Interativo*
Schildt	–	*C – Guia de Referência Básica*
Schildt	–	*Inteligência Artificial Utilizando Linguagem C*
Schildt	–	*Turbo C – Guia do Usuário*
Schildt	–	*Turbo C – Guia de Referência Básica*
Schildt	–	*Turbo C Avançado – Guia do Usuário*
Silveira	–	*Linguagem C – Comandos Básicos – Guia do Operador*

0-07-460857-6