Funções

este módulo apresenta o uso de funções na linguagem C. Funções são o principal elemento de modularização de código nesta linguagem, são muito versáteis e extensivamente usadas.

Declaração

Em C, uma função é declarada da seguinte forma geral:

```
<return type> function_name (<type parameter>, ...)
{
  <function body>
  return <value> ;
}
```

Um exemplo trivial: uma função para somar dois inteiros:

soma.c

```c
#include <stdio.h>

int soma (int a, int b)
{
  return (a + b) ;
}

int main ()
{
  printf ("soma (2, 5): %d\n", soma (2, 5)) ;
  return (0) ;
}
```

Um exemplo de função para calcular o fatorial de um número inteiro:

fatorial.c

```c
#include <stdio.h>

long fatorial (int n)
{
  int i ;
  long fat ;

  if (n <= 1)       // por definição
    fat = 1 ;
  else
  {
    fat = 1 ;
    for (i = 2; i <= n; i++)
      fat *= i ;
  }
```

```
  return (fat) ;
}

int main ()
{
  printf ("fat (10): %ld\n", fatorial (10)) ;
  return (0) ;
}
```

Obviamente, a função `fatorial` também pode ser definida de forma recursiva:

fatorial.c

```
long fatorial (int n)
{
  long fat ;

  if (n <= 1)
    fat = 1 ;
  else
    fat = n * fatorial (n - 1) ;

  return fat ;
}
```

Ou:

fatorial.c

```
long fatorial (int n)
{
  if (n <= 1)
    return 1 ;
  else
    return (n * fatorial (n - 1)) ;
}
```

Ou ainda:

fatorial.c

```
long fatorial (int n)
{
  return (n <= 1 ? 1 : n * fatorial (n - 1)) ;
}
```

A linguagem C **não aceita** funções aninhadas (definidas dentro de outras funções). Assim todas as funções são definidas no mesmo nível hierárquico. Observe que **isso não impede de chamar uma função dentro de outra função**.

Cabe observar que não é obrigatório usar o valor de retorno de uma função. Por exemplo, o código abaixo é perfeitamente válido (apesar de ser inútil):

```
fatorial (20) ;
```

O valor de retorno de uma função pode ser de qualquer tipo suportado pela linguagem C (tipos numéricos, ponteiros, *chars*, *structs*, etc), exceto *arrays* (vetores e matrizes) e funções. Todavia, essa limitação pode ser contornada através do uso de ponteiros para dados desses tipos. Funções que não retornam nenhum valor podem ser declaradas com o tipo `void`:

```
void hello ()
{
  printf ("Hello!\n") ;
}
```

Protótipos

Em princípio, toda função em C deve ser declarada antes de ser usada. Caso o compilador encontre uma função sendo usada que não tenha sido previamente declarada, ele pressupõe que essa função retorna um `int` e emite um aviso no terminal.

Em algumas situações não é possível respeitar essa regra. Por exemplo, se duas funções diferentes se chamam mutuamente, teremos:

```
char patati (int a, int b)
{
   ...
   c = patata (x, y, z) ; // precisa declarar "patata" antes de usar
   ...
}

char patata (int a, int b, int c)
{
   ...
   x = patati (n1, n2) ;
   ...
}
```

Para evitar problemas, é possível declarar um “protótipo” da função antes de sua definição completa:

```
// protótipos das funções "patati" e "patata"
char patati (int a, int b) ;
char patata (int a, int b, int c) ;

// implementação da função "patati" (deve respeitar o protótipo)
char patati (int a, int b)
{
   ...
   c = patata (x, y, z) ;    // ok, patata tem um protótipo definido
   ...
}

// implementação da função "patata" (deve respeitar o protótipo)
char patata (int a, int b, int c)
```

```
{
   ...
   x = patati (n1, n2) ;  // ok, patati tem um protótipo definido
   ...
}
```

Um protótipo de função pode ser visto como a sua "interface", porque define o **nome** da função, o número e tipos dos **parâmetros** de entrada e o tipo do valor de **retorno**. Essas informações são suficientes para a compilação de qualquer código que use essa função. Por isso, protótipos são muito usados em arquivos de cabeçalho (`.h`).

Parâmetros

Em C, os parâmetros das funções são transferidos sempre **por valor** (ou por cópia), pois os valores fornecidos na chamada da função são copiados para dentro da pilha (*stack*) e ficam à disposição do código interno da função. Em consequência, alterações nos parâmetros efetuadas dentro das funções não têm impacto fora dela.

Por exemplo:

paramcopia.c

```
#include <stdio.h>

void inc (int n)
{
  n++ ;
  printf ("n vale %d\n", n) ;
}

int main ()
{
  int a = 0 ;
  printf ("a vale %d\n", a) ;
  inc (a) ;
  printf ("a vale %d\n", a) ;
  return (0) ;
}
```

A execução deste código resulta em:

```
a vale 0
n vale 1
a vale 0
```

Parâmetros por referência

Para que as ações de uma função sobre seus parâmetros sejam visíveis fora da função, esses parâmetros devem ser informados **por referência**. C não suporta nativamente a passagem de parâmetros por referência, mas se considerarmos que um ponteiro é uma referência a uma variável, basta usar parâmetros de tipo

ponteiro para obter esse efeito:

paramref.c

```c
#include <stdio.h>

void inc (int *n)
{
  (*n)++ ;
}

int main ()
{
  int a = 0 ;
  printf ("a vale %d\n", a) ;
  inc (&a) ; // informar a referência (endereço) de a
  printf ("a vale %d\n", a) ;
  return (0) ;
}
```

E a execução fica:

```
a vale 0
a vale 1
```

Deve-se observar que o ponteiro *n recebe uma **cópia** do endereço de a, ou seja, a transferência de parâmetros propriamente dita continua sendo feita por cópia.

Um exemplo clássico de passagem de parâmetros por referência é a troca de valores entre duas variáveis. O código abaixo implementa essa função:

troca.c

```c
#include <stdio.h>

// troca os valores de dois inteiros entre si
void troca (int *a, int *b)
{
  int aux ;

  aux = *a ;
  *a  = *b ;
  *b  = aux ;
}

int main ()
{
  int i, j ;

  i = 21 ;
  j = 76 ;
  printf ("i: %d, j: %d\n", i, j) ;

  troca (&i, &j) ;
  printf ("i: %d, j: %d\n", i, j) ;
```

```
  return (0) ;
}
```

O que acontece no código acima se chamarmos `troca (&i, NULL)`? Como resolver isso?

Parâmetros vetoriais

A passagem de vetores e matrizes como parâmetros de funções tem algumas particularidades que devem ser observadas:

- Como o nome de um vetor representa o endereço de seu primeiro elemento, na prática vetores são passados à função **por referência**;
- Em consequência, o conteúdo de um vetor **pode ser alterado** em uma chamada de função.

Um exemplo simples de chamada de função com parâmetros vetoriais:

```
#define SIZE 100

// "limpa" um vetor de n posições com zeros
void clean_vect (int n, int v[])    // ou "int v[SIZE]" ou ainda "int *v"
{
  int i ;
  for (i = 0 ; i < n; i++)
    v[i] = 0 ;
}

int main ()
{
  int vector[SIZE] ;
  int num ;

  ...
  clean_vect (num, vector) ;
  ...
}
```

No caso de matrizes (vetores multidimensionais), deve-se informar ao compilador os tamanhos máximos das várias dimensões, para que ele possa calcular a posição onde cada elemento da matriz foi alocado na memória. Por exemplo:

```
#define MAXLIN 100
#define MAXCOL 50

void clean_mat (int l, int c, int m[][MAXCOL])  // ou "int m[MAXLIN][MAXCOL]"
{
  int i, j ;
  for (i = 0 ; i < l; i++)
    for (j = 0 ; j < c; j++)
      m[i][j] = 0 ;
}
```

```
int main ()
{
  int mat[MAXLIN][MAXCOL] ;
  int lin, col ;

  ...
  clean_mat (lin, col, mat) ;
  ...
}
```

Uso do return

A estrutura “ortodoxa” de código com a chamada a `return` somente no final da função pode levar a um código longo e cansativo de ler (e de programar). O exemplo a seguir apresenta uma função que compara dois inteiros e retorna -1 (se a<b), 0 (se a=b) ou +1 (se a>b):

```
int compara (int a, int b)
{
  int result ;

  if (a < b)
    result = -1 ;
  else
    if (a > b)
      result = 1 ;
    else
      result = 0 ;

  return result ;
}
```

Entretanto, é possível sair da função invocando `return` a qualquer instante, levando a um código mais conciso e fácil de ler:

```
int compara (int a, int b)
{
  if (a < b) return -1 ;
  if (a > b) return  1 ;
  return 0 ;
}
```

Exercícios

a) Passagem de parâmetros por valor:

1. Escrever funções em C para:
    1. calcular a^b, com *b* inteiro e *a* e o retorno de tipo *double*.
    2. trocar duas variáveis inteiras entre si.
    3. comparar dois números inteiros *a* e *b*; a função retorna -1 se *a*<*b*, 0 se *a*=*b* e +1 se *a*>*b*.
    4. retornar o maior valor em um vetor de inteiros.

b) Passagem de parâmetros vetoriais:

1. Escrever um programa em C para somar dois vetores de inteiros. Crie funções separadas para a) ler um vetor; b) somar dois vetores; c) imprimir um vetor.
2. Escreva um programa para ordenação de vetores, com as seguintes funções:
    - `le_vetor (vetor, N)`: ler um número N e um vetor de N inteiros;
    - `ordena_vetor (vetor, N)`: ordenar o vetor lido usando a técnica de **ordenação da bolha**;
    - `escreve_vetor (vetor, N)`: imprimir os elementos de um vetor com N elementos.
3. Escreva um programa para transpor matrizes, com as seguintes funções:
    - `le_matriz (matriz, M, N)`: ler uma matriz de MxN inteiros;
    - `transpoe_matriz (matriz, M, N)`: transpor uma matriz;
    - `escreve_matriz (matriz, M, N)`: imprimir uma matriz.

c) Passagem de parâmetros por referência:

1. Escreva uma função `int separa (float r, int *pi, float *pf)` que separa um número real `r` em suas partes inteira (`pi`) e fracionária (`pf`). Por exemplo: 37,543 ⇒ 37 e 0,543. A função retorna 1 se deu certo ou 0 se ocorreu algum erro.
2. Defina uma estrutura `struct data` com três campos: *dia*, *mês* e *ano*. Em seguida, escreva as seguintes funções:
    - `int data_set (int d, int m, int a, struct data *d)`: ajusta a data d com o dia/mês/ano recebidos; retorna 1 se a data é válida e 0 se não for ou se outro erro ocorreu (desconsidere anos bissextos).
    - `void data_print (struct data d)`: imprime a data informada em d, no formato "dd/mm/aaaa".
3. Escreva uma função `int max (int v[], int tam, int *maxval, int *maxpos)` onde:
    - `v[]`: vetor de inteiros desordenado
    - `tam`: tamanho do vetor (número de elementos)
    - `maxval`: maior valor encontrado no vetor
    - `maxpos`: posição do maior valor no vetor (a 1ª posição, se `max` se repetir)
    - retorno: 1 em sucesso ou 0 em erro