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> ;
  }
```

A função abaixo calcula o fatorial de um inteiro:

```
#include <stdio.h>

int fatorial(int n)  // retorna um int, argumento é de tipo int
{
   int i, fat ;

   if (n <= 1)       // por definição
      return (1) ;

   fat = 1 ;
   for (i=2; i <= n; i++)
      fat *= i ;

   return (fat) ;
}

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

A função "fatorial" também pode ser definida de forma recursiva:

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

Ou ainda:

```
int 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*.

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

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 que ela seja definida:

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

// 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 "interface" da função, porque define o nome, o número e

tipos dos argumentos de entrada e o tipo do valor de retorno. Essas informações são suficientes para a compilação do código que use a função. Por isso, protótipos são declarados em arquivos de cabeçalho (".h").

Parâmetros ou Argumentos

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 a pilha (*stack*) de onde são acessados pelo código da função. Em consequência, alterações nos parâmetros efetuadas dentro das funções não têm impacto fora dela.

Por exemplo:

```
#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

Para que as ações de uma função sobre seus parâmetros sejam perceptíveis 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:

```
#include <stdio.h>

void inc (int *n)   // o argumento é o endereço da variável n
{
  (*n)++ ;          // incrementa o valor apontado
}

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

E a execução resulta em:

a vale 0 a vale 1

Deve-se observar que a variável "a" é transferida para a função "inc" por referência, mas o ponteiro "*n" recebe uma **cópia** do endereço de "a". A transferência de parâmetros é sempre feita por cópia.

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

void clean_vect (int n, int v[])  // ou "int v[SIZE]"
{
   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) ;
   ...
}
```

Por que a declaração "int m[][MAXCOL]" é suficiente, ao invés da versão completa "int m[MAXLIN][MAXCOL]"?

Exercícios

Escreva funções em C para:

- calcular o valor de a^b (a elevado à b-ésima potência), com *b* inteiro, o argumento *a* e o valor de retorno de tipo *double*;
- trocar os conteúdos de duas variáveis inteiras entre si;
- comparar dois números inteiros *a* e *b*; a função deve retornar -1 se $a<b$, 0 se a=b e +1 se $a>b$;
- retornar o maior valor em um vetor de inteiros.
- Escreva um programa em C para somar dois vetores de inteiros. Escreva funções separadas para (a) ler um vetor; (b) somar dois vetores; (c) imprimir um vetor.
- Escreva sua própria versão das funções de manipulação de strings "strlen", "strcpy" e "strcat". Depois, compare o desempenho de sua implementação em relação às funções originais da LibC (sugestão: meça o tempo necessário para ativar cada função um milhão de vezes).