C

Guilherme P. Telles

IC

30 de março de 2023

Funções

Função

- Em C uma função pode ser vista como um tipo especial de bloco.
- As regras de escopo de nomes se aplicam a funções da mesma forma que a blocos:
  - Um nome definido em um bloco é visível no bloco e em todos os blocos internos.
  - Um nome continua existindo enquanto o fluxo de execução estiver dentro do bloco em que ele foi definido, em algum bloco interno ou em alguma função chamada dentro do bloco.
  - Logo, variáveis definidas dentro de uma função deixam de existir quando a função termina.

Definição

- A definição de uma função tem a forma

```
[tipo|void] id(lista-de-declarações) {
  declarações e sentenças
}
```

- O tipo informa ao compilador como o valor retornado pela função vai ser convertido se for preciso.
  - A palavra `void` é usada para dizer que a função não retorna nenhum valor.
  - Se nada for especificado o retorno é `int`.

- A lista de declarações pode ser vazia, representada por `f(void)`.
  - `f()` é uma função com um número variável de parâmetros declarada na forma tradicional.
  - O mais correto é escrever `f(void)` para uma função que não recebe parâmetros.

Declaração

- A declaração de uma função tem a forma

```
[tipo|void] id(lista-de-declarações);
```

- As declarações são chamadas de *protótipos*.
- Os nomes de parâmetros podem ser omitidos na declaração.

Arquivos `.h` e `.c`

- Declarações tipicamente são colocadas em arquivos `.h`, que são incluídos por outros arquivos.
- Definições tipicamente são colocadas em arquivos `.c`.
- P.ex. permutations.c e permutations.h.

Chamada

- Uma chamada de função tem a forma

```
id(lista-de-expressões)
```

- A lista de expressões correspondem aos parâmetros na mesma ordem da declaração da função.
- Funções devem ser declaradas ou definidas antes de serem chamadas.

Retorno

- O retorno de função tem sintaxe

```
return;
return sentença;
return (sentença);
```

- Pode haver zero ou mais `return` em uma função.
- A cláusula `return` faz a execução retornar para o ponto em que a função foi chamada.
- Se não houver `return`, o retorno acontece quando o fim do corpo da função é alcançado.

- O valor retornado, se houver, é convertido para o tipo da função se for necessário.
- O valor retornado não precisa ser usado.
- Se uma função que retorna valor termina sem retornar um valor então o valor retornado é indefinido.
  - Exceção é a função Main, que retornará zero nesse caso.
- Funções não podem retornar vetores ou funções.
- Funções podem retornar apontadores mas não faz sentido retornar o endereço de uma variável local.

```c
float celsius(float F) {

  return (F-32)/1.8;
}
```

```c
void die(char* message, int code) {

  printf("terminated: %s\n",message);
  exit(code);
}
```

```c
void die2(void) {
  printf("terminated.\n");
  exit(1);
}
```

```c
int is_vowel(char c) {

  if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
    return 1;
  else
    return 0;
}
```

```c
int is_vowel(char c) {

  return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
```

Parâmetros

- Todos os parâmetros para uma função são passados por valor.
- Parâmetros não podem ser inicializados.
- O mecanismo de passagem por referência é obtido com o uso de apontadores como parâmetros e endereços de variáveis nas chamadas.

exm-par-1.c

```c
#include <stdio.h>

void swap_bad(int a, int b) {

  int x = a;
  a = b;
  b = x;
}


void swap_good(int* p, int* q) {

  int x = *q;
  *q = *p;
  *p = x;
}
```

```c
int main() {

  int x = 51, y = 42;

  printf("x=%d y=%d\n",x,y);

  swap_bad(x,y);
  printf("bad  x=%d y=%d\n",x,y);

  swap_good(&x,&y);
  printf("good x=%d y=%d\n",x,y);

  return 0;
}
```

- No exemplo, `p` e `q` são dois apontadores passados por valor.
- Se modificarmos o valor deles (isto é, o endereço armazenado neles) nada acontece com `x` e `y`.
- A troca acontece porque modificamos o *valor apontado* por eles.

Vetores como parâmetros de função

- Um parâmetro declarado como um vetor é um apontador.

```
int f(int V[])
```

é equivalente a

```
int f(int* V)
```

- O endereço do primeiro elemento do vetor é passado por valor.
- O tamanho de um vetor passado para uma função não pode ser recuperado por `sizeof`.
- Então quase sempre precisamos passar o tamanho do vetor para a função também.

exm-par-2a.c

```c
#include <stdio.h>

#define n 51

void func(int array[n]) {

  for (int j=0; j<n; j++)
    array[j] = j;

  printf("sizeof dentro da funcao %ld\n", sizeof(array));
}


int main() {

  int x[n];

  printf("sizeof fora de funcao %ld\n", sizeof(x));

  func(x);
}
```

- Alternativas:

```
void f(int array[13])

void f(int n, int array[n])

void f(int n, int array[])

void f(int array[], int n)

void f(int array*, int n)

void f(int n, int array*)
```

- Em qualquer dessas formas podemos usar o operador [] dentro da função:

```
array[i]
array[11]
```

```c
// Funcao para somar n elementos de um vetor.
int somar(int V[], int n) {   // alt: somar(int* V, int n)

  int soma = 0;

  for (n -= 1; n >= 0; n--)
    soma += V[n];

  return soma;
}


int main(void) {

  int A[10], i;

  for (i=0; i<10; i++)
    A[i] = i;

  printf("A[0..5]: %d\n",somar(A,5));
  printf("A[3..5]: %d\n",somar(A+3,3));
  printf("A[7..7]: %d\n",somar(A+7,1));
}
```

Strings

- Strings são vetores de char. Então as mesmas regras se aplicam.
- Uma particularidade é que como a string é terminada com `\0` podemos usar `strlen` para recuperar o tamanho da string e nem sempre é importante saber o tamanho do array.
- Se o tamanho do array for importante dentro da função, ele tem que ser passado como um parâmetro.

Arrays multi-dimensionais como parâmetros

- Um parâmetro declarado como um array multi-dimensional também é um apontador.
- Precisamos passar todas as dimensões na declaração do array como parâmetro, mas podemos omitir a primeira.

```c
void f(int M[13][23])
void f(int M[][23])

void f(int m, int n, int M[m][n])
void f(int m, int n, int M[][n])

void f(C[13][23][7])
void f(C[][23][7])

void f(int s, int m, int n, int C[s][m][n])
void f(int s, int m, int n, int C[][m][n])
```

Parênteses: estilo tradicional

- A forma tradicional de definição é

```
tipo id (lista-de-identificadores)
  definições-dos-parâmetros
{...}
```

- Na forma tradicional o tipo dos parâmetros deve ser assegurado pelo programador, não há conversão.

```
int f (a,b,c)
  int a,b;
  double c;
{
  return a+b*c;
}
```