

Linguagem C: Funções

Parte 1*

Delano Beder

*Baseado no livro: Linguagem C Completa e Descomplicada, de André Backes

Função

- Funções são blocos de código que podem ser nomeados e chamados de dentro de um programa.
- Já usamos diversas funções
 - **printf()**: função que escreve na tela
 - **scanf()**: função que lê o teclado

Por que utilizar?

- Evitar que os blocos do programa fiquem grandes demais e mais difíceis de entender
- Facilitar a leitura do programa-fonte
- Separar o programa em partes(blocos) que possam ser logicamente compreendidos de forma isolada

Por que utilizar?

- Facilitar a estruturação e reutilização do código.
 - Permitem o reaproveitamento de código já construído
 - Evitam a repetição desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros

Função - estrutura

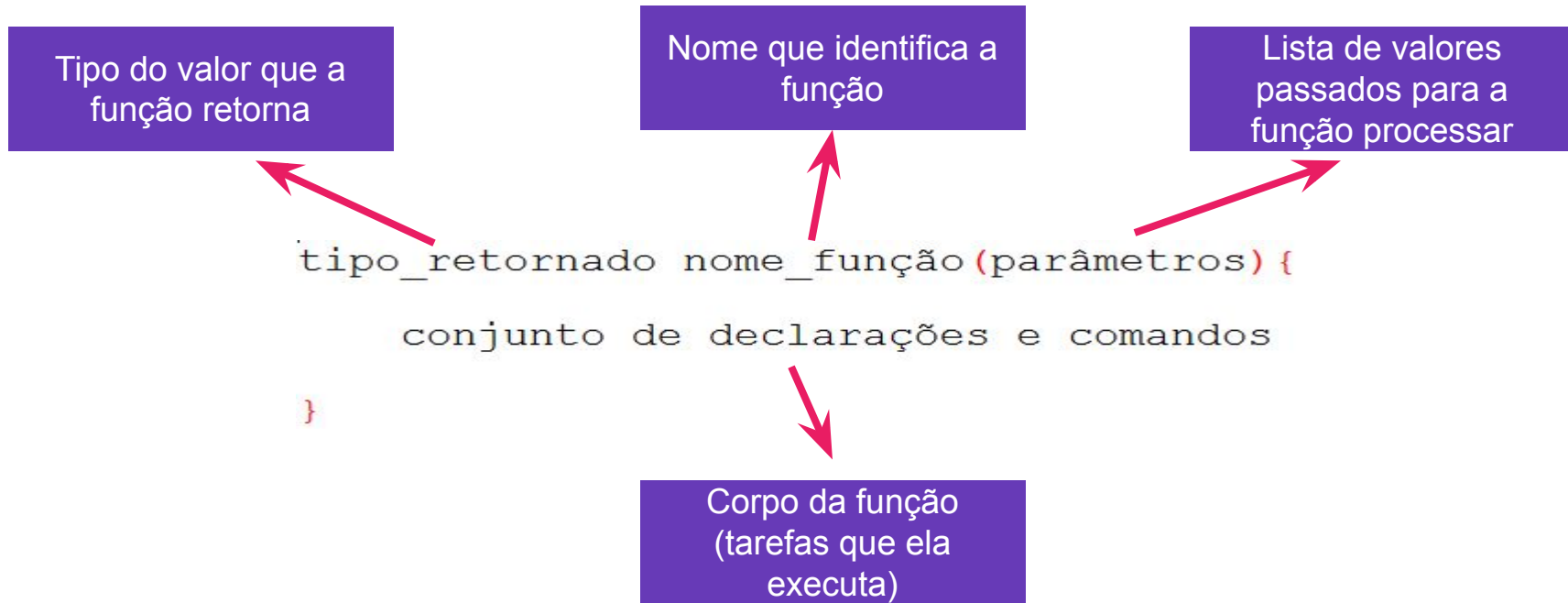
Uma função possui o seguinte formato:

```
tipo_retornado nome_da_funcao (tipo <parametro1>,  
tipo <parametro2>, ..., tipo <parametro n>) {  
  
    Comandos;  
  
    return(valor de retorno);  
  
}
```

Toda função tem um tipo, que determina o tipo do valor que será retornado.

Função - estrutura

- Forma geral de uma função:



Função - Corpo

- Formado pelos comandos que a função deve executar
- Processa os parâmetros (se houver), realiza outras tarefas e gera saídas (se necessário)

```
int soma(int x, int y) {  
    return x + y;  
}
```

Função - Corpo

- Evita-se fazer operações de leitura e escrita dentro de uma função.
 - Uma função é construída com o intuito de realizar uma tarefa específica e bem definida
 - As operações de entrada e saída de dados (funções **scanf()** e **printf()**) geralmente são feitas em quem chamou a função (por exemplo, na **main()**)
 - Isso assegura que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade

Função - Parâmetros

- A declaração de parâmetros é uma lista de variáveis juntamente com seus tipos:
 - *tipo1 nome1, tipo2 nome2, ... , tipoN nomeN*
 - Pode-se definir quantos parâmetros forem necessários, separados por vírgula

```
//Declaração CORRETA de parâmetros
int soma(int x, int y) {
    return x + y;
}
```



```
//Declaração ERRADA de parâmetros
int soma(int x, y) {
    return x + y;
}
```


Função - Parâmetros

- É por meio dos parâmetros que uma função recebe informação do programa principal (isto é, de quem a chamou)
 - Não é preciso fazer a leitura das variáveis dos parâmetros dentro da função

```
int x = 2;  
int y = 3;
```

```
int soma(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = soma(2, 3);  
  
    return 0;  
}
```

```
int soma(int x, int y) {  
    scanf("%d", &x);  
    scanf("%d", &y);  
  
    return x + y;  
}
```



Função - Parâmetros

- Podemos criar uma função que não recebe nenhum parâmetro de entrada
- Isso pode ser feito de duas formas
 - Podemos deixar a lista de parâmetros vazia
 - Podemos colocar **void** entre os parênteses

```
void imprime() {  
    printf("Teste\n");  
}
```

```
void imprime(void) {  
    printf("Teste\n");  
}
```

Função - Retorno

- Uma função pode ou não retornar um valor
 - Se ela retornar um valor, alguém deverá receber este valor
 - Uma função que retorna nada é definida colocando-se o tipo **void** como valor retornado. **Funções que não retornam nada são conhecidos como Procedimentos.**
- Podemos retornar qualquer valor válido em C
 - tipos pré-definidos: int, char, float e double
 - tipos definidos pelo usuário: struct
 - Uma função não pode retornar um vetor
 - A linguagem C não suporta a atribuição de um vetor a outro

Comando return

- O valor retornado pela função é dado pelo comando **return**
- Forma geral:
 - **return** *valor ou expressão*;
 - **return**;
 - Usada para terminar uma função que não retorna valor
- É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Comando return

Função com retorno de valor

```
int soma(int x, int y){  
    return x + y;  
}  
  
int main(){  
    int z = soma(2,3);  
  
    return 0;  
}
```

Função sem retorno de valor (Procedimento)

```
void imprime(){  
    printf("Teste\n");  
}  
  
int main(){  
    imprime();  
  
    return 0;  
}
```

Comando return

- Uma função pode ter mais de uma declaração **return**.
 - Quando o comando **return** é executado, a função termina imediatamente.
 - Todos os comandos restantes são **ignorados**.

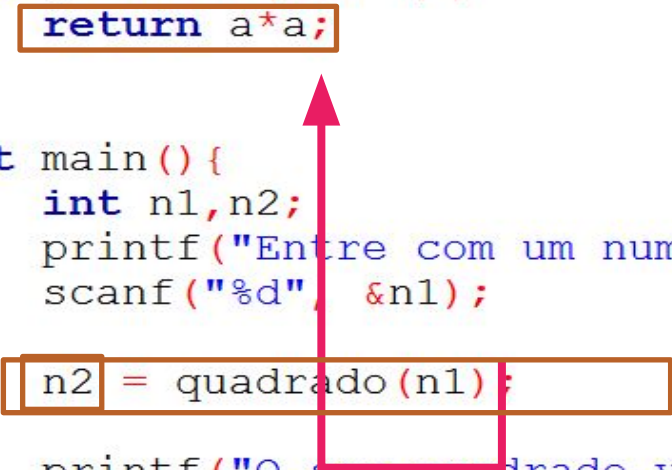
```
int maior(int x, int y){  
    if(x > y)  
        return x;  
    else  
        return y;  
    printf("Esse texto nao sera impresso\n");  
}
```

→ ignorado

Função – Ordem de Execução

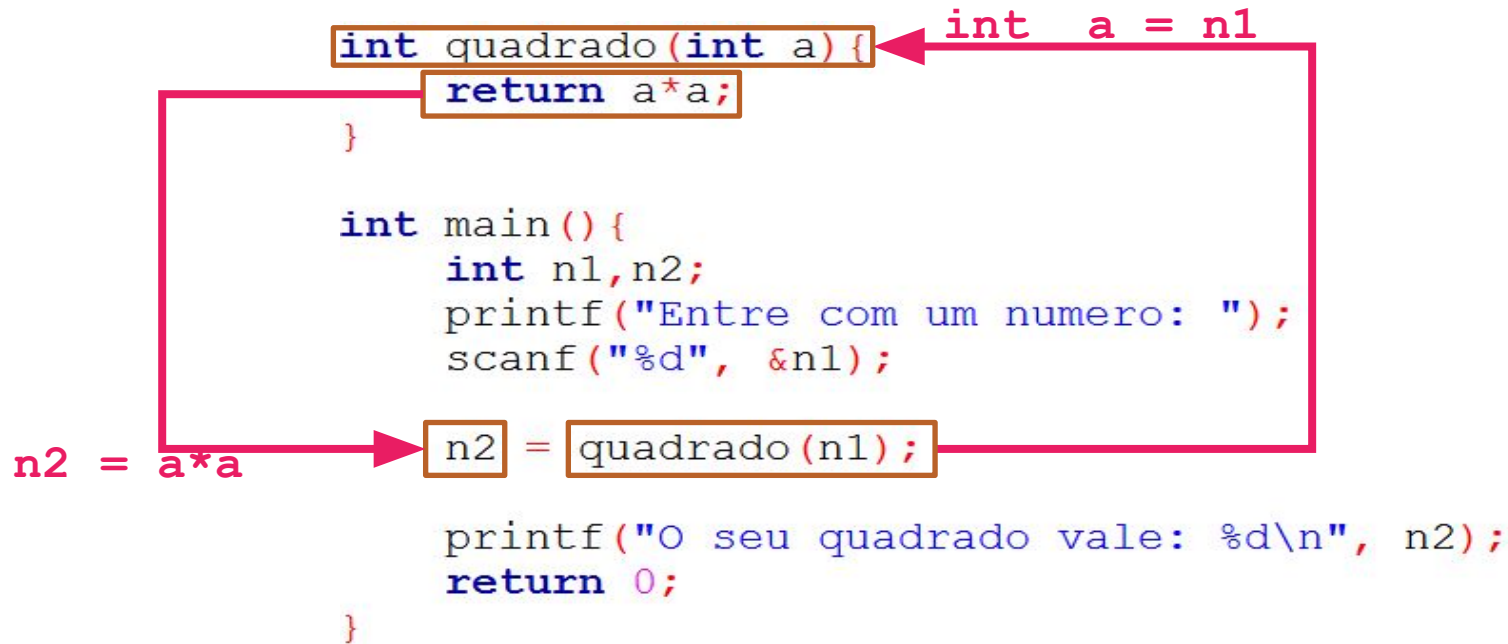
- Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução

```
int quadrado(int a) {  
    return a*a;  
}  
  
int main() {  
    int n1, n2;  
    printf("Entre com um numero: ");  
    scanf("%d", &n1);  
    n2 = quadrado(n1);  
    printf("O seu quadrado vale: %d\n", n2);  
    return 0;  
}
```

A red arrow originates from the function call `n2 = quadrado(n1);` in the `main` function and points upwards to the `return a*a;` statement in the `quadrado` function. Both the function call and the return statement are enclosed in orange rectangular boxes, illustrating the flow of control and data during a function call.

Função – Ordem de Execução

- Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução



Declaração de Funções

- Funções devem ser declaradas antes de serem utilizadas, ou seja, antes da função **main**.
 - Uma função criada pelo programador pode utilizar qualquer outra função, inclusive as que foram criadas

```
int quadrado(int a) {  
    return a*a;  
}
```

```
int main() {  
    int n1, n2;  
    printf("Entre com um numero: ");  
    scanf("%d", &n1);  
  
    n2 = quadrado(n1);  
  
    printf("O seu quadrado vale: %d\n", n2);  
    return 0;  
}
```

Declaração de Funções

- Podemos definir apenas o **protótipo da função** antes da função **main**.
 - O protótipo apenas indica a existência da função
 - Desse modo ela pode ser declarada após a função `main()`.

```
tipo_retornado nome_função(parâmetros);
```

Os nomes dos parâmetros não são obrigatórios.
Apenas os tipos. Protótipos equivalentes.

```
int soma(int a, int b);  
int soma(int, int);
```

Declaração de Funções

- Exemplo de protótipo

```
int quadrado(int a);
```

```
int main() {  
    int n1, n2;  
    printf("Entre com um numero: ");  
    scanf("%d", &n1);  
  
    n2 = quadrado(n1);  
  
    printf("O seu quadrado vale: %d\n", n2);  
    return 0;  
}  
  
int quadrado(int a) {  
    return a*a;  
}
```

Exercícios 1: conceitos básicos - AVA

Exercícios 2: prática de desenvolvimento - AVA

Escopo

- O escopo determina a validade de variáveis nas diversas partes do programa.
 - Variáveis Locais
 - Variáveis Globais
 - Parâmetros formais

Escopo

- Variáveis locais são aquelas que só têm validade dentro do bloco no qual são declaradas.
 - Um bloco começa quando abrimos uma chave e termina quando fechamos a chave.
 - Ex.: variáveis declaradas dentro da função.

```
int fatorial (int n) {  
    if (n == 0)  
        return 1;  
    else {  
        int i;  
        int f = 1;  
        for (i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

Escopo

- **Parâmetros formais** são declarados como sendo as entradas de uma função.
 - O **parâmetro formal** é uma **variável local** da função.
 - Ex.:
 - x é um parâmetro formal

```
float quadrado(float x);
```


Escopo

- Variáveis globais são declaradas fora de todas as funções do programa.
- Elas são conhecidas e podem ser alteradas por todas as funções do programa.
 - Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local.
- ***Evite variáveis globais!***

Exemplo

```
#include <stdio.h>
```

```
int x, y; → globais
```

Parâmetros formais são
variáveis locais

```
int soma (int x, int y) {
```

```
    x += 1;
```

```
    y -= 1;
```

```
    printf ("Soma de %d e %d: %d\n", x, y, x + y);
```

```
    return (x + y);
```

```
}
```

Quais variáveis
foram alteradas?

```
int main () {
```

```
    printf ("Digite o valor de x: ");
```

```
    scanf ("%d", &x);
```

```
    printf ("Digite o valor de y: ");
```

```
    scanf ("%d", &y);
```

```
    printf ("Soma de %d e %d: %d\n", x, y, soma(x, y));
```

```
    printf ("x: %d\n", x);
```

```
    printf ("y: %d\n", y);
```

```
    return 0;
```

```
}
```

Qual a saída?

```
Digite o valor de x: 14
Digite o valor de y: 18
Soma de 15 e 17: 32
Soma de 14 e 18: 32
x: 14
y: 18
```

Exercícios 3: escopo - AVA

Exercícios 4: prática de desenvolvimento - AVA

Passagem de Parâmetros

- Na linguagem C, os parâmetros de uma função geralmente são passados por **valor**, ou seja, uma cópia do valor do parâmetro é feita e passada para a função.
- Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função.

Passagem por valor

```
void incrementa(int n) {  
    n = n + 1;  
  
    printf("Dentro da funcao: x = %d\n", n);  
}  
  
int main() {  
    int x = 5;  
    printf("Antes da funcao: x = %d\n", x);  
  
    incrementa(x);  
  
    printf("Depois da funcao: x = %d\n", x);  
    return 0;  
}
```

Saída:

Antes da funcao: x = 5

Dentro da funcao: x = 6

Depois da funcao: x = 5

Passagem por referência

- Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros **por referência**.
- Neste tipo de chamada, não se passa para a função o valor da variável, mas a sua **referência** (seu endereço na memória).

Passagem por referência

- Utilizando o endereço da variável, qualquer alteração que a variável sofra dentro da função será refletida fora da função.
- Ex: função **scanf()**

Passagem por referência

- Para passar um parâmetro por referência, utiliza-se um ponteiro:

```
//passagem de parâmetro por valor  
void incrementa(int n);
```

```
//passagem de parâmetro por referência  
void incrementa(int *n);
```


Passagem por referência

- Ao se chamar a função, é necessário agora utilizar o operador “&”, como é feito com a função **scanf()**:

```
//passagem de parâmetro por valor
```

```
int x = 10;  
incrementa(x);
```

```
//passagem de parâmetro por referência
```

```
int x = 10;  
incrementa(&x);
```

Passagem por referência

- No corpo da função, é necessário usar um asterisco “*” sempre que se desejar acessar o conteúdo do parâmetro passado por referência.

```
//passagem de parâmetro por valor
```

```
void incrementa(int n) {  
    n = n + 1;  
}
```

```
//passagem de parâmetro por referência
```

```
void incrementa(int *n) {  
    *n = *n + 1;  
}
```

Passagem por referência

```
1 include <stdio.h>
2 include <stdlib.h>
3
4 void soma_mais_um(int *n){
5     *n = *n + 1;
6     printf("Dentro da funcao: x = %d\n",*n);
7 }
8
9 int main (){
10     int x = 5;
11     printf("Antes da funcao: x = %d\n",x);
12     soma_mais_um(&x);
13     printf("Depois da funcao: x = %d\n",x);
14
15     return 0;
16 }
```

Saída

Antes da funcao: x = 5
Dentro da funcao: x = 6
Depois da funcao: x = 6

Exercício

- Crie uma função que troque o valor de dois números inteiros passados por referência.

Por valor	Por referência
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void Troca(int a,int b){ 5 int temp; 6 temp = a; 7 a = b; 8 b = temp; 9 printf(''Dentro: %d e %d\n'',a,b); 10 } 11 12 int main() { 13 int x = 2; 14 int y = 3; 15 printf(''Antes: %d e % d\n'',x,y); 16 Troca(x,y); 17 printf(''Depois: %d e %d\n'',x,y); 18 19 return 0; 20 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void Troca(int*a,int*b){ 5 int temp; 6 temp = *a; 7 *a = *b; 8 *b = temp; 9 printf(''Dentro: %d e %d\n'',*a,*b); 10 } 11 12 int main() { 13 int x = 2; 14 int y = 3; 15 printf(''Antes: %d e % d\n'',x,y); 16 Troca(&x,&y); 17 printf(''Depois: %d e %d\n'',x,y); 18 19 return 0; 20 }</pre>

Por valor

Saída

Antes: 2 e 3
Dentro: 3 e 2
Depois: 2 e 3

Por referência

Saída

Antes: 2 e 3
Dentro: 3 e 2
Depois: 3 e 2

**Exercícios: passagem por valor e por
referência – AVA**

Referências

— — —

- BACKES, André. **Linguagem C: completa e descomplicada**. Rio de Janeiro: Elsevier, 2013. 371 p. ISBN 978-85-352-6855-3. Disponível na Biblioteca.