

# Fundamentos de Programação

## Funções - passagem de parâmetros por valor

### (Parte 2)

Dainf - UTFPR

Profa. Leyza B. Dorini  
Prof. Bogdan T. Nassu

## Escopo local de variáveis

Cada função tem o seu escopo, delimitado pelas chaves. Uma variável **local** é aquela declarada dentro do escopo de uma função. Nesse caso, ela pode ser acessada somente dentro daquela função e deixa de “existir” após o término da execução da mesma. Se a função for chamada várias vezes, a variável será “recriada” a cada chamada.

### Atenção!


Variáveis locais com mesmo nome declaradas em funções diferentes são variáveis distintas.

# Escopo local de variáveis

Vamos fazer um teste de mesa para analisar de forma detalhada o seguinte exemplo:

```
1  int dobro(int x){
2
3      int tmp;
4      tmp = x*2;
5      return tmp;
6  }
7
8  int main()
9  {
10     int x = 5, tmp = 3,
11         aux;
12
13     aux = dobro(x) + tmp;
14     return 0;
15 }
```

A execução do programa  
começa pela função main().



# Escopo local de variáveis

```
1  int dobro(int x){  
2  
3      int tmp;  
4      tmp = x*2;  
5      return tmp;  
6  }  
7  
8  int main()  
9  {  
10     int x = 5, tmp = 3,  
11     aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

As variáveis x e tmp  
são declaradas e inicializadas.

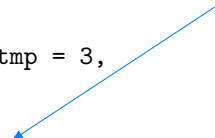
A variável aux  
é declarada.

Variáveis do escopo da main()
x = 5
tmp = 3
aux = ?

# Escopo local de variáveis

```
1  int dobro(int x){  
2  
3      int tmp;  
4      tmp = x*2;  
5      return tmp;  
6  }  
7  
8  int main()  
9  {  
10     int x = 5, tmp = 3,  
11         aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

Encontra a chamada  
para a função dobro  
(passando o conteúdo  
de x como parâmetro).



Variáveis do escopo da main()
x = 5
tmp = 3
aux = ?

# Escopo local de variáveis

```
1  int dobro(int x){  
2  
3      int tmp;  
4      tmp = x*2;  
5      return tmp;  
6  }  
7  
8  int main()  
9  {  
10     int x = 5, tmp = 3,  
11         aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

A variável `x` é declarada e inicializada com o valor recebido por parâmetro.

Variáveis do escopo da função

`x = 5`

Observe que essa variável `x` é diferente daquela declarada na `main()`.

Variáveis do escopo da `main()`

`x = 5`

`tmp = 3`

`aux = ?`

# Escopo local de variáveis

```
1  int dobro(int x){  
2  
3      int tmp;  
4      tmp = x*2;  
5      return tmp;  
6  }  
7  
8  int main()  
9  {  
10     int x = 5, tmp = 3,  
11         aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

A variável tmp é declarada —  
note que ela é diferente daquela  
declarada na main().

## Variáveis do escopo da função

x = 5  
tmp = ?


## Variáveis do escopo da main()

x = 5  
tmp = 3  
aux = ?

# Escopo local de variáveis

```
1 int dobro(int x){  
2  
3     int tmp;  
4     tmp = x*2;  
5     return tmp;  
6 }
```

As instruções da função  
são executadas.



```
7  
8 int main()  
9 {  
10     int x = 5, tmp = 3,  
11     aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

## Variáveis do escopo da função

x = 5  
tmp = 10

## Variáveis do escopo da main()


x = 5  
tmp = 3  
aux = ?



# Escopo local de variáveis

```
1  int dobro(int x){
2
3      int tmp;
4      tmp = x*2;
5      return tmp;
6  }
7
8  int main()
9  {
10     int x = 5, tmp = 3,
11         aux;
12
13     aux = dobro(x) + tmp;
14     return 0;
15 }
```

A função retorna o conteúdo da variável tmp para quem a chamou (no caso deste exemplo, retorna para a main()).



## Variáveis do escopo da função

x = 5  
tmp = 10

## Variáveis do escopo da main()

x = 5  
tmp = 3  
aux = ?

# Escopo local de variáveis

```
1 int dobro(int x){  
2  
3     int tmp;  
4     tmp = x*2;  
5     return tmp;  
6 }
```

```
7  
8 int main()  
9 {  
10     int x = 5, tmp = 3,  
11         aux;  
12  
13     aux = dobro(x) + tmp;  
14     return 0;  
15 }
```

O conteúdo retornado pela função dobro()  
é somado ao conteúdo de tmp e o  
resultado é armazenado na variável aux...

Variáveis do escopo da função

x = 5

tmp = 10

Variáveis do escopo da main()

x = 5

tmp = 3

aux = 13

# Escopo local de variáveis

```
1  int dobro(int x){
2
3      int tmp;
4      tmp = x*2;
5      return tmp;
6  }
7
8  int main()
9  {
10     int x = 5, tmp = 3;
11     aux;
12
13     aux = dobro(x) + tmp;
14     return 0;
15 }
```

... e as variáveis locais  
da função são  
“destruídas” (desalocadas).

## Variáveis do escopo da função

~~x = 5~~

~~tmp = 10~~

## Variáveis do escopo da main()

x = 5

tmp = 3

aux = 13

## Exercício de fixação

---

# Exercício de fixação

```
1  int teste(int x){
2      int tmp;
3
4      x = x * 2;
5      tmp = x * 10;
6
7      return tmp;
8  }
9  int main(){
10     int x = 5,
11     aux;
12
13     printf("Antes: x=%d\n", x);
14     aux = teste(x);
15     printf("Depois: x=%d, aux=%d\n", x, aux);
16
17     return 0;
18 }
```

Variáveis do escopo da função

x =

tmp =

Variáveis do escopo da main()

x =

aux =

O que será  
impresso aqui?

# Exercício de fixação

Resposta:

```
1  int teste(int x){
2      int tmp;
3
4      x = x * 2;
5      tmp = x * 10;
6
7      return tmp;
8  }
9  int main(){
10     int x = 5,
11     aux;
12
13     printf("Antes: x=%d\n", x);
14     aux = teste(x);
15     printf("Depois: x=%d, aux=%d\n", x, aux);
16
17     return 0;
18 }
```

Variáveis do escopo da função

x = 5 10

tmp = 100

Note que a variável x  
que tem o valor alterado  
é a do escopo da função.

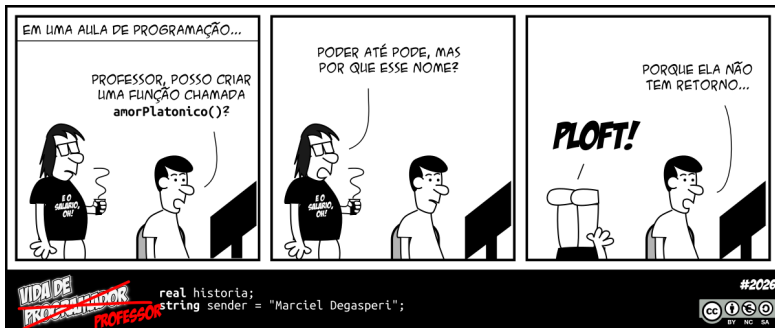
Variáveis do escopo da main()

x = 5

aux = 100

# O tipo void


Ele representa o “tipo não especificado” e pode ser utilizado para definir procedimentos, que são basicamente funções que não retornam nenhum valor (em C usa-se apenas a nomenclatura “função”).



# Definindo uma função sem retorno (procedimento)

Um procedimento é definido da seguinte forma:

O tipo `void` é  
utilizado para indicar que a função  
não retornará nenhum valor.



```
1 void nomeProced (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     //nao tem retorno.
5 }
```



# Definindo uma função sem retorno (procedimento)

Um procedimento é definido da seguinte forma:

É por meio deste nome que  
você irá invocar (chamar)  
a função.

```
1 void nomeProced (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     //nao tem retorno.
5 }
```

Parâmetros de entrada -  
cada um tem seu tipo, e se comporta  
como uma variável declarada e  
inicializada quando a  
função é chamada.

Bloco de comandos do  
procedimento (delimitado por chaves).

# Definindo uma função sem retorno (procedimento)

Um procedimento é definido da seguinte forma:

```
1 void nomeProced (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     //nao tem retorno.
5 }
```

O ponto mais importante  
aqui é: **não tem** return.

## Exemplo 01 - definindo um procedimento

O procedimento abaixo imprime o número que for passado para ele como parâmetro uma determinada quantidade de vezes.

```
1 void imprime(int numero, int vezes)
2 {
3     int i;
4     for(i=0; i<vezes; i++)
5         printf("%d ", numero);
6 }
```

**Importante:** observe que o comando `return` **não** é utilizado.

Como/Onde colocar no código  
os procedimentos que eu fiz?

# Funciona da mesma forma que para as funções!

A implementação do procedimento pode estar:

- ① antes da função `main()` ou
- ② depois dela - neste caso, o protótipo do procedimento deve estar definido antes da `main()`.

## Siga um padrão!

Nos seus programas, você pode optar por colocar procedimentos/funções implementados antes ou depois da `main()`. O importante é seguir um padrão (ou seja, não coloque alguns procedimentos/funções antes e outros depois).

# Chamando (invocando) um procedimento

Para invocarmos um procedimento, basta chamar passando os parâmetros de entrada necessários. Para o caso do exemplo anterior:

```
imprime (num, vezes);  
imprime (10, 5);  
imprime (numero*4, qtde);
```

## Importante:

Como um procedimento não tem retorno, não podemos atribuir sua chamada à uma variável, como fazemos com funções:

```
x = imprime (num, vezes);    //ERRO!!!
```

# Chamando (invocando) um procedimento

Para invocarmos um procedimento, basta chamar passando os parâmetros de entrada necessários. Para o caso do exemplo anterior:

```
imprime (num, vezes);  
imprime (10, 5);  
imprime (numero*4, qtde);
```

## Importante:

Como um procedimento não tem retorno, não podemos atribuir sua chamada à uma variável, como fazemos com funções:

```
x = imprime (num, vezes);    //ERRO!!!
```

Lembre-se que:

- Qualquer programa começa executando os comandos da `main()`.
- Quando se encontra a chamada para uma função, o fluxo de execução passa para ela e são executados os comandos até que um `return` seja encontrado.
- Depois disso, o fluxo de execução volta para o ponto onde a chamada da função ocorreu.



## O que colocar em uma função?

Quando criar uma função, pense nela como uma unidade de comportamento bem definido e limitado. Leve também em consideração a possibilidade de reutilização!

# O que colocar em uma função?

```
1 void raizQuadrada () {  
2     double x, raiz_calculada:  
3     scanf("%lf ", &x);  
4     /* Algoritmo para calcular  
5        a raiz quadrada. */  
6     (...)  
7     printf("%lf ", raiz_calculada);  
8 }
```

e se a informação for lida  
a partir de um arquivo?

e se for necessário  
usar essa informação  
(e não apenas imprimi-la)?

# O que colocar em uma função?

```
1 void raizQuadrada () {  
2     double x, raiz_calculada;  
3     scanf("%lf ", &x);  
4     /* Algoritmo para calcular  
5        a raiz quadrada. */  
6     (...)  
7     printf("%lf ", raiz_calculada);  
8 }  
9  
10 double raizQuadrada (double x) {  
11     double raiz_calculada;  
12     /* Algoritmo para calcular a raiz quadrada. */  
13     (...)  
14     return (raiz_calculada);  
15 }
```

essa solução atende a uma  
quantidade maior de possibilidades

# O que **NÃO** colocar em uma função?



Escopo de variáveis: local × global

# Variáveis locais e variáveis globais

As variáveis podem ser classificadas de duas formas:

- Uma variável **local** é aquela declarada dentro de uma função. Nesse caso, ela pode ser acessada somente dentro daquela função e deixa de “existir” após o término da execução da mesma. Lembre-se: variáveis locais com mesmo nome declaradas em funções diferentes são variáveis distintas.
- Uma variável **global** é aquela declarada fora de qualquer função, tornando-se “visível” em todas as funções. Portanto, é possível acessá-la em qualquer ponto do programa. Ela só deixa de “existir” após o término da execução do programa.

# Variáveis globais

## Atenção

A princípio, você não poderá usar variáveis globais nos programas desenvolvidos durante a disciplina (a não ser que devidamente justificadas). ;-)



**Richard**  
@zzaaho



Q: What is the best prefix for  
global variables?

A: //

2:11 AM · 21 Jan 19 · [Twitter Web Client](#)

# Variáveis globais

O uso de variáveis globais deve ser evitado pois é uma causa comum de erros:

- Partes diferentes do código, inclusive em funções distintas, podem alterar a variável global, causando uma grande interdependência entre estas partes distintas de código.
- A legibilidade do código pode piorar com o uso de variáveis globais, dado que ao ler uma função que usa uma variável global é difícil inferir seu valor inicial e, portanto, qual o resultado da função sobre a variável global.



# Professor(a), mas variáveis globais....



Mais algumas coisas...

# A função `main()`

Você já parou para pensar que a `main()` é uma função? Sim, e é especial, por ser o “ponto de entrada” do programa.



# A função `main()`

O padrão é ter um `int` como valor de retorno. O valor de retorno é um código de status do programa. Por convenção, se o programa executou sem erros, é 0.

A lista de parâmetros costuma ser uma das opções abaixo:

- `int main ()`: sem parâmetros.
- `int main (int argc, char* argv [])`: recebe argumentos da linha de comando. Não vamos usar esta versão.

# Funções podem invocar funções

---

Uma função pode chamar outra função...

- ... que pode chamar outra função, que pode chamar outra função...
- Isso é uma das coisas que permite que o grau de abstração dos programas suba. Começamos com funções “amplas”, que vão chamando funções cada vez mais específicas.
- As chamadas sucessivas formam uma “pilha de chamadas”.

# Funções podem invocar funções

```
1  int fun1(int a);
2  int fun2(int b);
3
4  int main(){
5      int c = 5;
6      c = fun1(c);
7      printf("c = %d\n", c);
8      return 0;
9  }
10
11 int fun1(int a){
12     a = a + 1;
13     a = fun2(a);
14     return a;
15 }
16
17 int fun2(int b){
18     b = 2 * b;
19     return b;
20 }
```

# Agora é com você!

---

- Hoje, vimos mais uma das estruturas fundamentais da programação: **as funções**.
- Quando fizer os exercícios, procure escrever diretamente as funções, não tente colocar tudo no `main` para depois arrumar.
- Fazer desta forma pode parecer mais difícil agora, mas:
  - Ajudará no desenvolvimento do pensamento mais abstrato.
  - No longo prazo, reduzirá o trabalho e a quantidade de bugs!
- Para testar as funções que você fizer nos exercícios, crie uma `main()` simples, que invoca a função algumas vezes com parâmetros diferentes.

# Agora é com você!

Faça os exercícios da lista! Quanto mais você praticar, mais vai ter dúvidas e aprender...

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```