

Fundamentos de Programação

Funções - passagem de parâmetros por valor

(Parte 1)

Dainf - UTFPR

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

Sobre programas longos

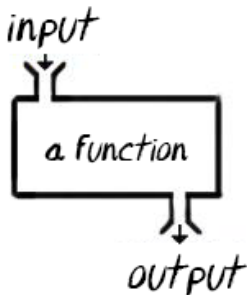
- Até o momento, temos trabalhado (“brincado”) com problemas curtos e diretos. Mas o que acontece quando os problemas se tornam grandes e complexos?
 - Sistemas reais costumam ter milhares de linhas de código.
 - É muito difícil pensar em cada linha de um sistema desses de uma só vez.
 - Depois de um tempo, até mesmo quem programou o sistema fica “perdido” no código.



Descomposição de problemas

Ao resolver um problema complexo, um ponto importante é conseguir “quebrá-lo” em subproblemas (decomposição). Com isso, são definidas partes menores, mais fáceis de entender e administrar. Este processo é conhecido como **modularização**!

Na linguagem C, utilizamos **funções** para implementar estes subproblemas! Princípio básico:



Funções

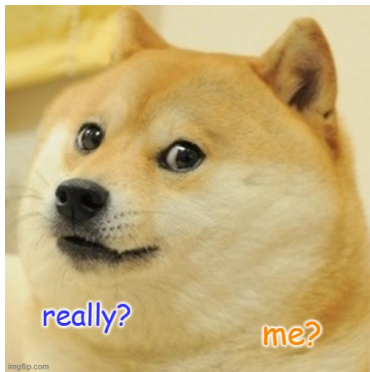
Funções

São estruturas que agrupam um conjunto de comandos, os quais são executados quando a função é chamada/invocada.



Vamos aprender a criar e usar funções em C.

Você já está usou funções nas listas de exercício



Por exemplo:

- `printf`: dada uma mensagem de entrada, mostra na tela.
- `scanf`: usada para entrada de dados.
- `main`: **SPOILER**, o `main` é uma função (mais detalhes depois).

Checkpoint

Por que utilizar funções?

Liste vantagens do uso de funções nos programas (pense em pelo menos uma antes de ver as respostas sugeridas, ok?).

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.

Checkpoint

Por que utilizar funções?

Liste vantagens do uso de funções nos programas (pense em pelo menos uma antes de ver as respostas sugeridas, ok?).

- Evitar que os blocos do programa fiquem grandes demais e, por consequência, mais difíceis de ler e entender.
- Separar o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa, minimizando erros e facilitando alterações.

Sobre repetição de tarefas

Exemplo

Digamos que o seu programa precisa localizar várias vezes o máximo entre variáveis.

```
1    (...)
2    if (var1 > var2)
3        max1_2 = var1;
4    else
5        max1_2 = var2;
6
7    if (var3 > var4)
8        max3_4 = var3;
9    else
10       max3_4 = var4;
11
12   if (var5 > var6)
13       max5_6 = var5;
14   else
15       max5_6 = var6;
16   (...)
```


Sobre repetição de tarefas

Não seria bom se pudéssemos abstrair a solução para subproblemas já resolvidos?

```
1 (...)  
2 max1_2 = max (var1, var2);  
3 max3_4 = max (var3, var4);  
4 max5_6 = max (var5, var6);  
5 (...)
```

Ou quem sabe ir ainda mais longe...

```
max1234 = max (var1, max (var2, max (var3, var4)))
```

Então...

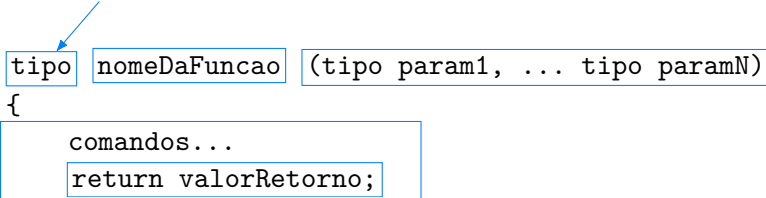
Na verdade, nós podemos fazer isso. Ou melhor, nós **devemos** fazer isso. Este é o princípio da modularização.

Definindo funções

Definindo uma função

Uma função é declarada/definida da seguinte forma:

Toda função deve ter um
tipo, o qual corresponde ao tipo
de seu valor de retorno
(int, float, etc.).



```
1 tipo nomeDaFuncao (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     return valorRetorno;
5 }
```

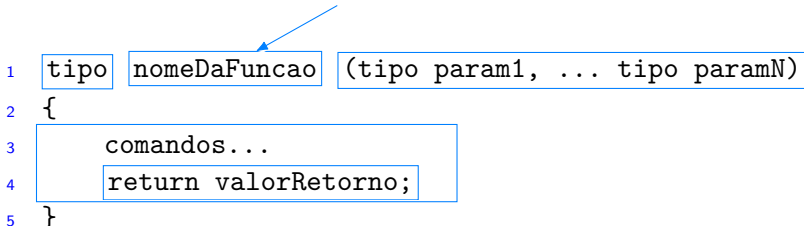
The diagram illustrates the syntax of a function definition. It consists of five lines of code. The first line is the function signature, which includes the return type, the function name, and the parameter list. The second line is an opening curly brace. The third line is the function body, which contains the code to be executed. The fourth line is the return statement, which returns the result of the function. The fifth line is a closing curly brace. Blue boxes highlight the following parts: 'tipo' on line 1, 'nomeDaFuncao' on line 1, '(tipo param1, ... tipo paramN)' on line 1, 'comandos...' on line 3, and 'return valorRetorno;' on line 4. A blue arrow points from the text 'tipo' in the explanatory paragraph above to the 'tipo' box on line 1.

Definindo uma função

Uma função é declarada/definida da seguinte forma:

É por meio deste nome que você irá invocar (chamar) a função - portanto, escolha um que seja significativo. Valem as mesmas regras dos identificadores de variáveis. Convenciona-se escrever:

nomeDaFuncao ou nome_da_funcao



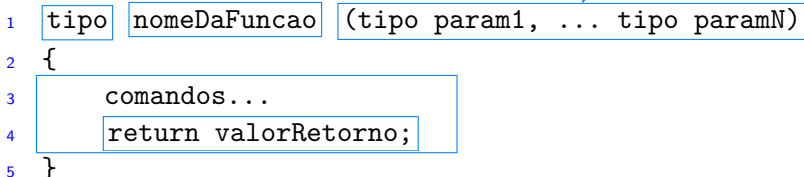
```
1  tipo  nomeDaFuncao  (tipo param1, ... tipo paramN)
2  {
3      comandos...
4      return valorRetorno;
5  }
```

The diagram illustrates the syntax for defining a function. It shows a function signature on line 1: `tipo nomeDaFuncao (tipo param1, ... tipo paramN)`. The components are highlighted with blue boxes: `tipo`, `nomeDaFuncao`, and the parameter list `(tipo param1, ... tipo paramN)`. A blue arrow points from the text 'nomeDaFuncao ou nome_da_funcao' above to the `nomeDaFuncao` box. The function body is enclosed in curly braces on lines 2 and 5. Inside the body, on line 3, is the code `comandos...`, and on line 4, is the return statement `return valorRetorno;`, which is also highlighted with a blue box.

Definindo uma função

Uma função é declarada/definida da seguinte forma:

Parâmetros formais - cada um tem seu tipo e se comporta como uma variável que só existe para a função, a qual é inicializada com os valores passados no momento que a função é invocada.



The diagram illustrates the syntax for defining a function in a programming language. It consists of five lines of code, each with a number on the left. The code is as follows:

```
1 tipo nomeDaFuncao (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     return valorRetorno;
5 }
```

Annotations (blue boxes) highlight specific parts of the syntax:

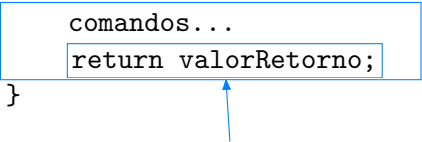
- Line 1: `tipo`, `nomeDaFuncao`, and the entire parameter list `(tipo param1, ... tipo paramN)` are boxed.
- Line 2: The opening curly brace `{` is boxed.
- Line 3: The text `comandos...` is boxed.
- Line 4: The text `return valorRetorno;` is boxed.
- Line 5: The closing curly brace `}` is boxed.

An arrow points from the text "Parâmetros formais" to the parameter list in line 1.

Definindo uma função

Uma função é declarada/definida da seguinte forma:

```
1 tipo nomeDaFuncao (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     return valorRetorno;
5 }
```



Retorna o conteúdo da variável `valorRetorno` para o trecho de código que chamou a função.

Definindo uma função

Uma função é declarada/definida da seguinte forma:

```
1 tipo nomeDaFuncao (tipo param1, ... tipo paramN)
2 {
3     comandos...
4     return valorRetorno;
5 }
```

Bloco de comandos da função (delimitado por chaves).

Exemplo: função soma()

A função soma() abaixo recebe dois inteiros como parâmetros, faz sua soma e retorna o resultado.¹

A função é do tipo `int`, pois retorna um valor inteiro

Procure escolher nomes significativos para funções.

```
1  int soma (int a, int b)
2  {
3      int c;
4
5      c = a + b;
6
7      return c;
8  }
```

Parâmetros de entrada:
note que cada parâmetro
é acompanhado do seu tipo.

Retorna o resultado
da soma (ou seja, o
conteúdo da variável c).

¹Esta função é simples demais e nunca seria feita no mundo real, mas releve isso por enquanto!

Exemplo: função calculaIMC()

Considere uma função que, dados como entrada peso e altura, calcula e retorna o IMC ($\text{peso}/\text{altura}^2$).

A função é do tipo float, pois retorna um valor float.

Procure escolher nomes significativos para funções.

```
1 float calculaIMC (float peso, float altura)
2 {
3     float imc;
4
5     imc = peso / (altura*altura);
6
7     return imc;
8 }
```

Parâmetros: altura e peso são do tipo float. Os parâmetros **não precisam** ter sempre todos o mesmo tipo!

Retorna o IMC calculado.

Sobre o return

A função pode retornar qualquer expressão cujo resultado tenha o tipo esperado.

```
1 float calculaIMC (float peso, float altura)
2 {
3     return peso / (altura*altura);
4 }
```

É possível retornar diretamente o resultado de uma expressão, ou seja, não é obrigatório criar uma variável.

Sobre o return

A função não precisa ter um único ponto de retorno.

```
1 int max (int a, int b)
2 {
3     if (a > b)
4         return (a);
5     else
6         return (b);
7
8     printf("Tchau");
9 }
```

Neste caso, a função retorna assim que encontrar o primeiro return. A mensagem Tchau **já** será impressa.

Sobre o return

Atenção: Retornar **não tem nada a ver** com “mostrar na tela”!



Revelations

Chamando (invocando) funções



**"Why is
my function
not returning
anything?"**

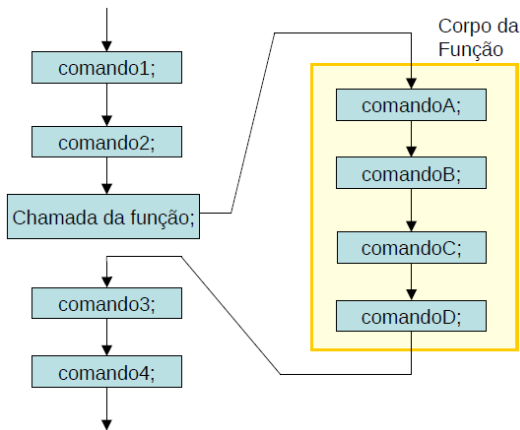


**"Oh, I never
called it"**

Para que as funções definidas sejam de fato executadas, precisamos chamá-las (invocá-las)!

Chamando (invocando) uma função

Quando invocamos uma função, o fluxo de execução é desviado de forma a executar as suas instruções. Quando a função termina, retorna o valor para quem a chamou e o fluxo inicial é retomado do ponto onde havia parado — os dois fluxos não seguem em paralelo!



Chamando (invocando) uma função

Depois que a função for declarada, ela pode ser chamada a partir de um ponto do programa. Para fazer isso com a função `calculaIMC()`, uma possível forma seria:

```
1 int main()  
2 {  
3     float peso, alt,  
4         resposta;  
5  
6     scanf("%f %f", &peso, &alt);  
7  
8     resposta = calculaIMC(peso, alt);  
9  
10    printf("O IMC eh: %f", resposta);  
11 }
```

Observe que a função `calculaIMC()` está sendo chamada (ou invocada) de “dentro” da `main()`.

Chamando (invocando) uma função

Ao chamar (invocar) uma função, precisamos estar atentos aos seguintes pontos:

```
1 resposta = calculaIMC(peso, alt);
```

O valor retornado pela função precisa ser utilizado de forma adequada.

Os argumentos passados como entrada precisam ser compatíveis com o que está definido na função.

Chamando (invocando) uma função

```
1 resposta = calculaIMC(peso, alt);
```

2
3 Neste exemplo, se o
4 retorno for armazenado
5 em uma variável, ela
6 deve ser do tipo float.

Neste exemplo, como a função `calculaIMC()`
espera receber dois valores do tipo `float`, as
variáveis `alt` e `peso` **precisam** ser `float`.
Note que os parâmetros “casam” pela **ordem**,
não pelo nome das variáveis!

```
7  
8 float calculaIMC (float peso, float altura)
```

Chamando (invocando) uma função

```
1 resposta = calculaIMC(1.78, 75.5);
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8 float calculaIMC (float peso, float altura)
```

Como os parâmetros “casam” pela ordem,
e não pelo nome, opcionalmente,
podemos passar valores diretamente
como parâmetro de entrada

Neste exemplo específico, o importante é passar como parâmetros quaisquer dois valores do tipo float, que é o exigido pela função!

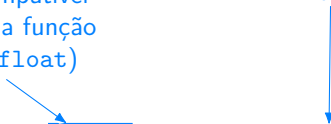
Chamando (invocando) uma função

Não necessariamente o retorno precisa ser armazenado em uma variável. Por exemplo...

Lembre-se que o especificador de formato deve ser compatível com o tipo do retorno da função (no caso do exemplo, `float`)

... é possível imprimir diretamente o valor retornado pela função.

```
1 printf("O IMC eh: ", calculaIMC(peso, alt));
```



Mas se você pretende reutilizar o valor retornado, pode ser interessante guardá-lo em uma variável. Do contrário, você precisará invocar a função várias vezes, e a cada chamada todo o processo será repetido!

Call a function



Importante!

Ao passar os argumentos para a função, assegure-se que:

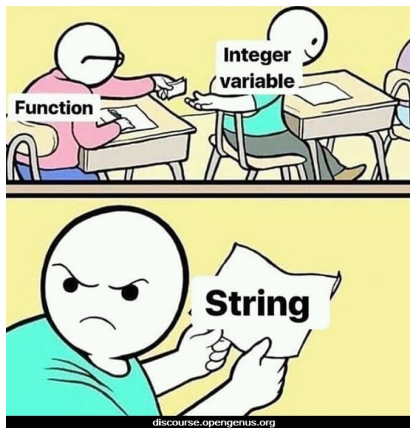
- eles são do tipo correto e
- estão na mesma ordem da declaração estabelecida no protótipo da função.

Além disso, manipule corretamente o valor retornado pela função:

- armazene-o em uma variável cujo tipo seja compatível com o tipo da função ...
- ... ou imprima o valor retornado usando o especificador de formato correto, por exemplo.

Atenção

É importante que você sempre verifique se o retorno da função está sendo adequadamente processado! Isso é um motivo muito comum de inconsistências nos programas!



Como/Onde colocar no código
as funções que eu fiz?

Pode ser **antes** da função main()...

Neste caso, toda a implementação da função ocorre antes da main(), como ilustra o exemplo abaixo:

```
1  #include<stdio.h>
2
3  float calculaIMC (float altura, float peso){
4      return peso / (altura*altura);
5  } //final da função...
6
7  int main(){
8      float alt, peso, resposta;
9
10     printf("Digite a altura e o peso: ");
11     scanf("%f %f", &alt, &peso);
12     resposta = calculaIMC(alt, peso);
13     printf("O IMC eh: %f", resposta);
14
15     return 0;
16 }
```


... ou **depois** da função main()

Neste caso, toda a implementação da função ocorre após a main(), mas é necessário declarar o *protótipo* da função antes!

```
1  #include<stdio.h>
2
3  //neste caso, é necessário colocar o protótipo aqui
4  float calculaIMC (float altura, float peso); //finalizado por ;
5
6  int main(){
7      float alt, peso, resposta;
8
9      printf("Digite a altura e o peso: ");
10     scanf("%f %f", &alt, &peso);
11     resposta = calculaIMC(alt, peso);
12     printf("O IMC eh: %f", resposta);
13     return 0;
14 } //final da main()
15
16 float calculaIMC (float altura, float peso){ //função
17     return peso / (altura*altura);
18 }
```

Protótipo??

Na lista de exercícios, você notará que as funções são descritas por um “protótipo”, que equivale à declaração da função, com o tipo de retorno e os parâmetros (mas sem o código).

- Diferente da implementação, o protótipo é seguido por um ;
- Falaremos mais sobre isso em breve.
- Por enquanto, siga o exemplo:

```
1 float calculaIMC (float altura, float peso); // Protótipo.
2
3 (...)
4
5 float calculaIMC (float altura, float peso) // Função.
6 {
7     return peso / (altura*altura);
8 }
```

Lista de exercícios

Agora é com você: faça os exercícios sugeridos! É comum que ocorram “bugs” (faz parte do aprendizado aprender a corrigí-los).

