

# Programação Estruturada

Funções

---

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



# Funções

---

- Um ponto chave na resolução de um problema complexo é conseguir “quebrá-lo” em subproblemas menores.
- Ao criarmos um programa para resolver um problema, é crítico quebrar um código grande em partes menores, fáceis de serem entendidas e administradas.
- Isto é conhecido como modularização, e é empregado em qualquer projeto de engenharia envolvendo a construção de um sistema complexo.

## Funções

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

- Vocês já usaram algumas funções como **scanf** e **printf**.
- Algumas funções podem devolver algum valor ao final de sua execução:

---

```
1 x = sqrt(4);
```

---

- Vamos aprender como criar e usar funções em C.

## Por que utilizar funções?

- 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.

## Definindo uma função

Uma função é definida da seguinte forma:

---

```
1 tipo_retorno nome(tipo parâmetro1,..., tipo parâmetroN) {  
2     comandos;  
3     return valor_de_retorno;  
4 }
```

---

- Toda função deve ter um tipo (**int**, **char**, **float**, **void**, etc). Esse tipo determina qual será o tipo de seu valor de retorno.
- Os **parâmetros** são variáveis, que são inicializadas com valores indicados durante a invocação da função.
- O comando **return** devolve para o invocador da função o resultado da execução desta.

## Definindo uma função: exemplo

A função abaixo recebe como parâmetro dois valores inteiros. Ela faz a soma destes valores e devolve o resultado.

---

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

---

- Note que o valor de retorno (variável **c**) é do mesmo tipo da função.
- Quando o comando **return** é executado, a função para de executar e retorna o valor indicado para quem fez a invocação (ou chamada) da função.

## Definindo uma função: exemplo

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

Qualquer função pode invocar esta função `soma`, passando como parâmetro dois valores inteiros, que serão atribuídos para as variáveis **a** e **b**, respectivamente.

```
1 int main() {  
2     int r;  
3     r = soma(12, 90);  
4     printf("r = %d\n", r);  
5     r = soma (-9, 45);  
6     printf("r = %d\n", r);  
7     return 0;  
8 }
```

## Definindo uma função: exemplo

---

```
1  #include <stdio.h>
2
3  int soma(int a, int b) {
4      int c;
5      c = a + b;
6      return c;
7  }
8
9  int main() {
10     int res, x1 = 4, x2 = -10;
11     res = soma(5, 6);
12     printf("Primeira soma: %d\n", res);
13     res = soma(x1, x2);
14     printf("Segunda soma: %d\n",res);
15     return 0;
16 }
```

---

## Execução de um programa com funções

- Qualquer programa começa executando os comandos da função **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 ou o fim da função seja alcançado.
- Depois disso, o fluxo de execução volta para o ponto onde a chamada da função ocorreu.

## Definindo uma função: exemplo

A lista de parâmetros de uma função pode ser vazia.

---

```
1 int leInteiro() {
2     int c;
3     printf("Digite um número: ");
4     scanf("%d", &c);
5     return c;
6 }
```

---

O retorno será usado pelo invocador da função:

---

```
1 int main() {
2     int r;
3     r = leInteiro();
4     printf("Numero digitado: %d\n", r);
5     return 0;
6 }
```

---

## Definindo uma função: exemplo

---

```
1  #include <stdio.h>
2
3  int leInteiro() {
4      int c;
5      printf("Digite um numero: ");
6      scanf("%d", &c);
7      return c;
8  }
9
10 int main() {
11     int r;
12     r = leInteiro();
13     printf("Numero digitado: %d\n", r);
14     return 0;
15 }
```

## Exemplo de função

A expressão contida dentro do comando **return** é chamada de *valor de retorno* (é a resposta da função). Nada após esse comando será executado.

---

```
1  #include <stdio.h>
2
3  int leInteiro() {
4      int c;
5      printf("Digite um numero: ");
6      scanf("%d", &c);
7      return c;
8      printf("Bla bla bla!\n");
9  }
10
11 int soma (int a, int b) {
12     return a + b;
13 }
14
15 int main() {
16     int x1, x2, res;
17     x1 = leInteiro();
18     x2 = leInteiro();
19     res = soma(x1, x2);
20     printf("Soma eh: %d\n", res);
21     return 0;
22 }
```

## Invocando uma função

- Uma forma clássica de realizarmos a invocação (ou chamada) de uma função é atribuindo o seu valor a uma variável:

---

```
1 x = soma(4, 2);
```

---

- Na verdade, o resultado da chamada de uma função é uma expressão e pode ser usado em qualquer lugar que aceite uma expressão:

---

```
1 printf("Soma de a e b: %d\n", soma(a, b));
```

---

## Invocando uma função

Na chamada da função, para cada um dos parâmetros desta devemos fornecer um valor de mesmo tipo e na mesma ordem da declaração.

---

```
1  #include <stdio.h>
2
3  int somaComMensagem(int a, int b, char texto[100]) {
4      int c = a + b;
5      printf("%s %d\n", texto, c);
6      return c;
7  }
8
9  int main() {
10     somaComMensagem(4, 5, "Resultado da soma:");
11     return 0;
12 }
```

## Invocando uma função

A saída do programa anterior será:

---

```
1 Resultado da soma: 9
```

---

Já a chamada abaixo gerará um erro de compilação:

---

```
1 int main() {  
2     somaComMensagem(4, "Resultado da soma:", 5);  
3     return 0;  
4 }
```

---

## Invocando uma função

- Ao chamar uma função passando variáveis **simples** como parâmetros, estamos usando apenas os seus valores, que serão copiados para as variáveis parâmetros da função.
- Os valores das variáveis na chamada da função não são afetados por alterações dentro da função.

---

```
1  #include <stdio.h>
2
3  int incr(int x) {
4      x = x + 1;
5      return x;
6  }
7
8  int main() {
9      int a = 2, b;
10     b = incr(a);
11     printf("a = %d, b = %d\n", a, b);
12     return 0;
13 }
```

---

- O que será impresso? O valor de **a** é alterado pela função **incr**?

## O tipo void

---

- O “tipo” **void** é um tipo especial.
- Ele representa “nada”, ou seja, uma variável desse tipo armazena conteúdo indeterminado, e uma função desse tipo retorna um conteúdo indeterminado.
- Em geral, este tipo é utilizado para indicar que uma função não retorna nenhum valor.

## O tipo void

- Por exemplo, a função abaixo imprime o número que for passado para ela como parâmetro e não devolve nada.
- Neste caso não utilizamos o comando **return**.

---

```
1 void imprimeInteiro(int numero) {  
2     printf("Número %d\n", numero);  
3 }
```

---

## O tipo void

---

```
1  #include <stdio.h>
2
3  void imprimeInteiro(int numero) {
4      printf("Número %d\n", numero);
5  }
6
7  int main () {
8      imprimeInteiro(10);
9      imprimeInteiro(20);
10     return 0;
11 }
```

---

## A função main

---

## A função main

- O programa principal é uma função especial, que possui um tipo fixo (**int**) e é invocada automaticamente pelo sistema operacional quando este inicia a execução do programa.
- Quando utilizado, o comando **return** informa ao sistema operacional se o programa funcionou corretamente ou não. O padrão é que um programa retorne zero caso tenha funcionado corretamente ou qualquer outro valor caso contrário.

---

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Ola turma!\n");
5      return 0;
6  }
```

---

# Protótipo de funções

---

## Protótipo de funções: definindo funções depois da main

Até o momento, aprendemos que devemos definir as funções antes do programa principal. O que ocorreria se declarássemos depois?

---

```
1  #include <stdio.h>
2
3  int main () {
4      float a = 0, b = 5;
5      printf("%f\n", soma(a, b));
6      return 0;
7  }
8
9  float soma(float op1, float op2) {
10     return op1 + op2;
11 }
```

---

Dependendo do compilador, ocorre um erro de compilação!

## Protótipo de funções: declarando uma função sem defini-la

- Para organizar melhor um programa e podermos implementar funções em partes distintas do arquivo fonte, utilizamos **protótipos de funções**.
- Protótipos de funções correspondem a primeira linha da definição de uma função, contendo: tipo de retorno, nome da função, parâmetros e por fim **um ponto e vírgula**.

---

```
1 tipo_retorno nome(tipo parâmetro1, ..., tipo  
↪ parâmetroN);
```

---

- O protótipo de uma função deve aparecer antes do seu uso.
- Em geral coloca-se os protótipos de funções no início do seu arquivo do programa.

# Protótipo de funções: declarando uma função sem defini-la

Em geral o programa é organizado da seguinte forma:

---

```
1  #include <stdio.h>
2  #include <outras bibliotecas>
3
4  /* Protótipos de funções */
5  int fun1(lista de parâmetros);
6
7  int main() {
8      comandos;
9  }
10
11 /* Declarações de funções */
12 int fun1(lista de parâmetros) {
13     comandos;
14 }
```

---

# Protótipo de Funções: Exemplo 1

---

```
1  #include <stdio.h>
2
3  float soma(float op1, float op2);
4  float subtrai(float op1, float op2);
5
6  int main () {
7      float a = 0, b = 5;
8      printf("soma = %f\nsubtracao = %f\n", soma (a, b), subtrai(a, b));
9      return 0;
10 }
11
12 float soma(float op1, float op2) {
13     return op1 + op2;
14 }
15
16 float subtrai(float op1, float op2) {
17     return op1 - op2;
18 }
```

---

# Funções Podem Invocar Funções

---

## Funções Podem Invocar Funções

- Nos exemplos anteriores apenas a função **main** invocava funções por nós definidas.
- Isto não é uma regra. Qualquer função pode invocar outra função (exceto a **main**, que é invocada apenas pelo sistema operacional).
- Veja o exemplo no próximo slide.

# Funções Podem Invocar Funções

O que será impresso?

---

```
1  #include <stdio.h>
2
3  int fun1(int a);
4  int fun2(int b);
5
6  int main() {
7      int c = 5;
8      c = fun1(c);
9      printf("c = %d\n", c);
10     return 0;
11 }
12
13 int fun1(int a) {
14     a = a + 1;
15     a = fun2(a);
16     return a;
17 }
18
19 int fun2(int b) {
20     b = 2 * b;
21     return b;
22 }
```

---

# **Escopo de Variáveis: variáveis locais e globais**

---

## Variáveis locais e variáveis globais

- Uma variável é chamada **local** se ela foi declarada dentro de uma função. Nesse caso ela existe somente dentro da função e, após o término da execução desta, a variável deixa de existir. **Variáveis parâmetros também são variáveis locais.**
- Uma variável é chamada **global** se ela for declarada fora de qualquer função. Essa variável é visível em todas as funções. Qualquer função pode alterá-la e ela existe durante toda a execução do programa.

# Organização de um programa

Em geral um programa é organizado da seguinte forma:

---

```
1  #include <stdio.h>
2  #include <outras bibliotecas>
3
4  protótipos de funções
5
6  declaração de Variáveis Globais
7
8  int main() {
9      declaração de variáveis locais à main
10     comandos;
11 }
12
13 int fun1(parâmetros) { /* parâmetros também são variáveis locais */
14     declaração de variáveis locais à fun1
15     comandos;
16 }
17
18 ...
```

- O **escopo** de uma variável determina de quais partes do código ela pode ser acessada, ou seja, de quais partes do código a variável é visível.
- A regra de escopo em C é bem simples:
  - As variáveis globais são visíveis por todas as funções.
  - As variáveis locais são visíveis apenas na função onde foram declaradas.

# Escopo de variáveis

---

```
1  #include <stdio.h>
2
3  void fun1();
4  int fun2(int local_b);
5
6  int global;
7
8  int main() {
9      int local_main;
10     /* Neste ponto são visíveis global e local_main */
11 }
12
13 void fun1() {
14     int local_a;
15     /* Neste ponto são visíveis global e local_a */
16 }
17
18 int fun2(int local_b) {
19     int local_c;
20     /* Neste ponto são visíveis global, local_b e local_c */
21 }
```

---

# Escopo de variáveis

- É possível declarar variáveis locais com o mesmo nome de variáveis globais.
- Nesta situação, a variável local “esconde” a variável global.

---

```
1  #include <stdio.h>
2
3  void fun();
4
5  int nota = 10;
6
7  int main() {
8      nota = 20;
9      fun();
10     printf("%d\n", nota);
11     return 0;
12 }
13
14 void fun() {
15     int nota;
16     nota = 5;
17     /* Neste ponto nota é a variável local de fun. */
18 }
```

---

# Exemplo 1

---

```
1  #include <stdio.h>
2
3  void fun1();
4  void fun2();
5
6  int x;
7
8  int main() {
9      x = 1;
10     fun1();
11     fun2();
12     printf("main: %d\n", x);
13     return 0;
14 }
15
16 void fun1() {
17     x = x + 1;
18     printf("fun1: %d\n", x);
19 }
20
21 void fun2() {
22     int x = 3;
23     printf("fun2: %d\n", x);
24 }
```

---

O que será impresso?

## Exemplo 2

---

```
1  #include <stdio.h>
2
3  void fun1();
4  void fun2();
5
6  int x = 1;
7
8  int main() {
9      int x = 1;
10     fun1();
11     fun2();
12     printf("main: %d\n", x);
13     return 0;
14 }
15
16 void fun1() {
17     x = x + 1;
18     printf("fun1: %d\n", x);
19 }
20
21 void fun2() {
22     int x = 4;
23     printf("fun2: %d\n", x);
24 }
```

---

O que será impresso?

## Exemplo 3

---

```
1  #include <stdio.h>
2
3  void fun1();
4  void fun2(int x);
5
6  int x = 1;
7
8  int main() {
9      x = 2;
10     fun1();
11     fun2(x);
12     printf("main: %d\n", x);
13     return 0;
14 }
15
16 void fun1() {
17     x = x + 1;
18     printf("fun1: %d\n", x);
19 }
20
21 void fun2(int x) {
22     x = x + 1;
23     printf("fun2: %d\n", x);
24 }
```

---

O que será impresso?

## Variáveis locais e variáveis globais

- O uso de variáveis globais deve ser evitado pois é uma causa comum de erros:
  - Partes distintas e funções distintas podem alterar a variável global, causando uma grande interdependência entre estas partes distintas de código.
- A legibilidade do seu código também pode piorar com o uso de variáveis globais:
  - 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.

## Exemplo utilizando funções

---

## Exemplo utilizando funções

Em aulas anteriores vimos como testar se um número em **candidato** é primo:

---

```
1 divisor = 2;
2 eh_primo = 1;
3 while (divisor <= candidato/2) {
4     if (candidato % divisor == 0) {
5         eh_primo = 0;
6         break;
7     }
8     divisor++;
9 }
10 if (eh_primo)
11     printf("%d, ", candidato);
```

---

E usamos isso para imprimir os  $n$  primeiros números primos:

## Exemplo utilizando funções

---

```
1  int main() {
2      int divisor, n, eh_primo, candidato, primosImpressos;
3      scanf("%d", &n);
4      if (n >= 1) {
5          printf("2, ");
6          primosImpressos = 1;
7          candidato = 3;
8          while (primosImpressos < n) {
9              divisor = 2;
10             eh_primo = 1;
11             while (divisor <= candidato/2) {
12                 if (candidato % divisor == 0) {
13                     eh_primo = 0;
14                     break;
15                 }
16                 divisor++;
17             }
18             if (eh_primo) {
19                 printf("%d, ", candidato);
20                 primosImpressos++;
21             }
22             candidato = candidato+2;
23         }
24     }
25     return 0;
26 }
```

## Exemplo utilizando funções

- Podemos criar uma função que testa se um número é primo ou não (note que isto é exatamente um bloco logicamente bem definido).
- Depois fazemos chamadas para esta função.

---

```
1  int eh_primo(int candidato) {
2      int divisor = 2;
3
4      while (divisor <= candidato/2) {
5          if (candidato % divisor == 0) {
6              return 0;
7          }
8          divisor++;
9      }
10
11     return 1;
12 }
```

---

# Exemplo utilizando funções

---

```
1  #include <stdio.h>
2
3  int eh_primo(int candidato); /* retorna 1 se candidato é primo, e 0 caso contrário */
4
5  int main() {
6      int n, candidato, primosImpressos;
7
8      scanf("%d",&n);
9
10     if (n >= 1) {
11         printf("2, ");
12         primosImpressos = 1;
13         candidato = 3;
14         while (primosImpressos < n) {
15             if (eh_primo(candidato)) {
16                 printf("%d, ", candidato);
17                 primosImpressos++;
18             }
19             candidato = candidato + 2;
20         }
21     }
22
23     return 0;
24 }
```

---

## Exercícios

---

## Exercício

Escreva uma função que computa a potência  $a^b$  para valores  $a$  (double) e  $b$  (int) passados por parâmetro (não use bibliotecas como math.h). Sua função deve ter o seguinte protótipo:

---

```
1 double pot(double a, int b);
```

---

Use a função anterior e crie um programa que imprima todas as potências:

$$2^0, 2^1, \dots, 2^{10}, 3^0, \dots, 3^{10}, \dots, 10^{10}.$$

## Exercício

Escreva uma função que computa o fatorial de um número  $n$  passado por parâmetro. Sua função deve ter o seguinte protótipo:

---

```
1 long int fat(int n);
```

---

OBS: Caso  $n \leq 0$ , seu programa deve retornar 1.

Use a função anterior e crie um programa calcula o coeficiente binomial de dois números  $n$  e  $k$  lidos:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Pergunta: o que acontece se você define a função como sendo do tipo `int`?

## Exercício

Escreva uma função em C para computar a raiz quadrada de um número positivo. Use a idéia abaixo, baseada no método de aproximações sucessivas de Newton descrito abaixo.

Seja  $Y$  um número. Sua raiz quadrada é raiz da equação

$$f(x) = x^2 - Y .$$

A primeira aproximação é  $x_1 = Y/2$ . A  $(n + 1)$ -ésima aproximação é

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

A função deverá retornar o valor da vigésima aproximação.

## Exercício

Faça um programa que teste quais números de uma sequência fazem parte da sequência de Fibonacci.

O programa deve receber inicialmente um inteiro  $n$ . Em seguida, deve receber  $n$  números inteiros positivos.

A resposta consiste de uma única linha, com os números da sequência da entrada que fazem parte da sequência de Fibonacci separados por um espaço.

Por exemplo, se a entrada é “7 3 9 -7 4 1 0 3”, então a saída é “3 1 0 3”.

Escreva um programa que mostre os dois números primos mais próximos de um dado número  $n$ .

Por exemplo, se  $n = 24$ , os dois números primos mais próximos dele são 23 e 29. Se  $n = 47$ , então os dois números primos mais próximos dele são 47 e 53.