ALGORITMOS E ESTRUTURAS DE DADOS I

UNIDADE 4

MODULARIZAÇÃO

PROF. NAÍSSES ZÓIA LIMA

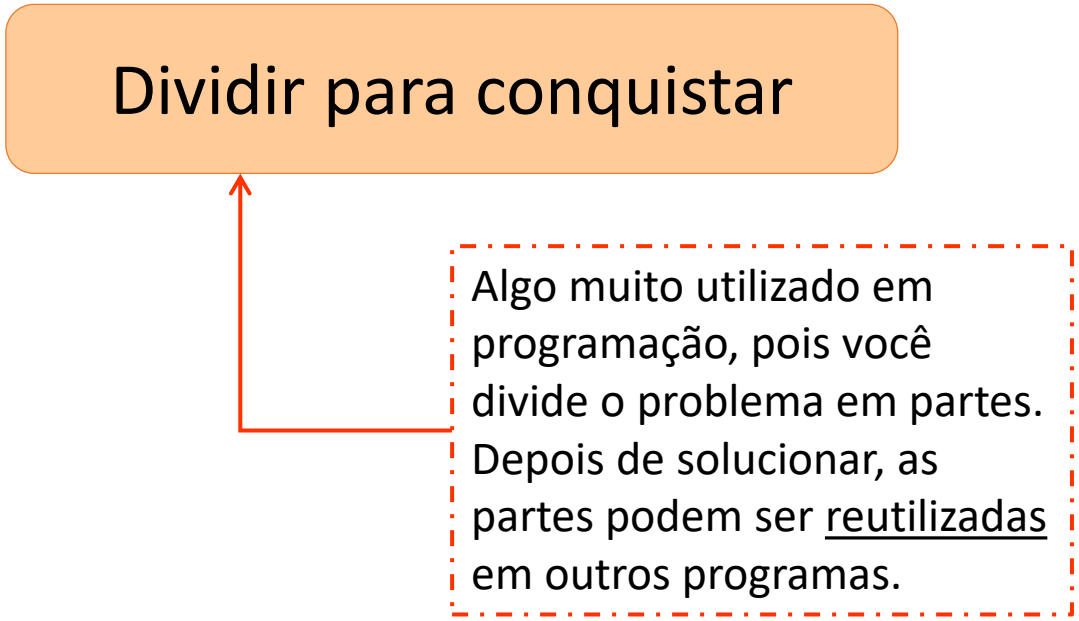
BASEADO NO MATERIAL DA PROFA. IVRE MARJORIE

Funções

Introdução

Razão principal para usar funções:

- Dividir a tarefa original em pequenas tarefas que simplificam e organizam o programa como um todo



- Promover reutilização de código, uma vez que o programa pode utilizar a função, que foi definida apenas uma vez, em vários locais.

Introdução

- **Outra razão**: reduzir o tamanho do programa
- Qualquer sequência de instruções que apareça no programa mais de uma vez é candidata a ser uma função
- O código de uma função é agregado ao programa uma única vez e pode ser executado muitas vezes no decorrer do programa
  - Para isso, basta fazer uma chamada da função. (reutilização!)
- Facilita a manutenção – uma alteração na função será feita somente em um ponto do código.

Função x Procedimento

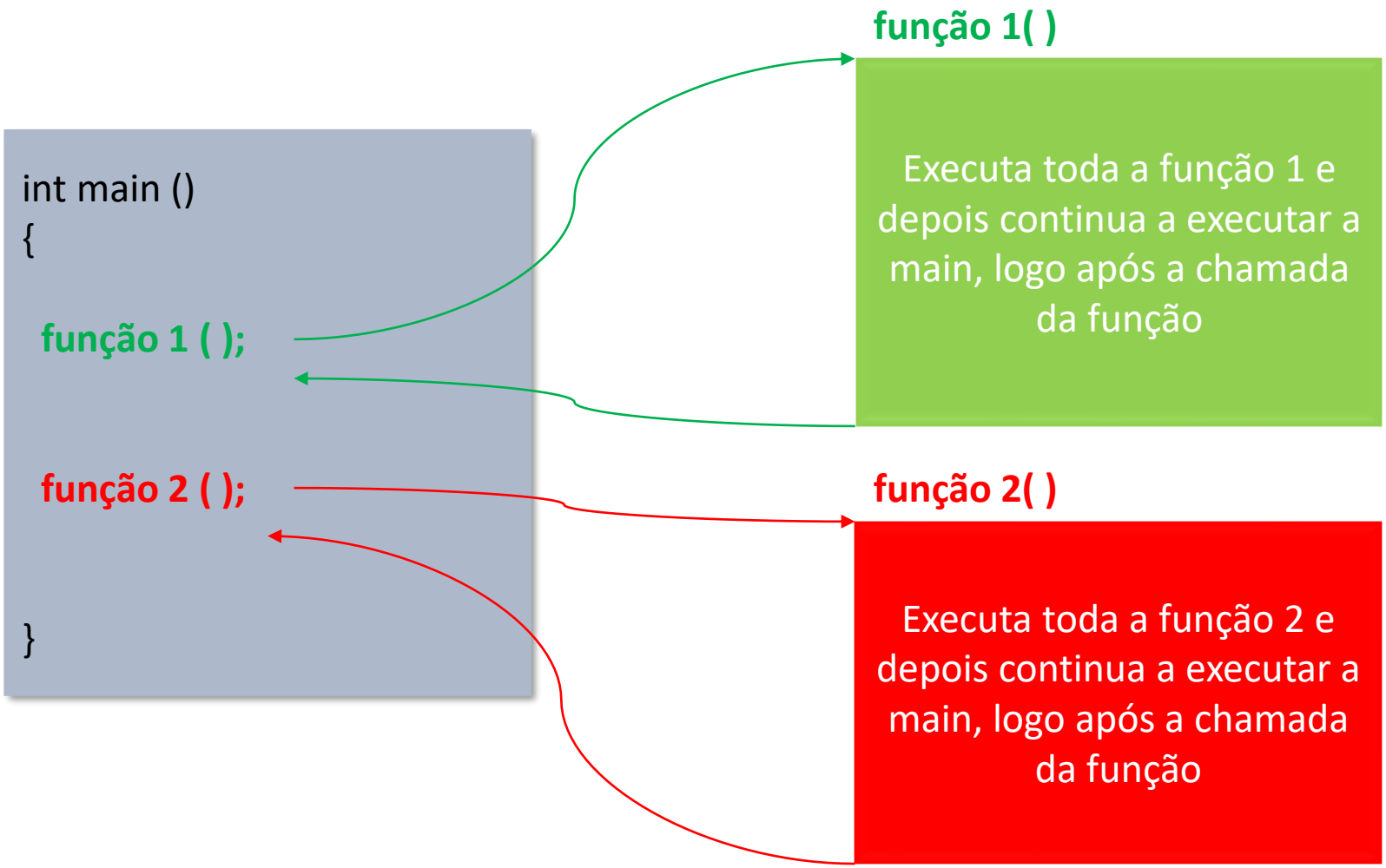
- **Função:** retorna algum resultado
  - Será de algum tipo
    - int, float, double, string, char, etc
  - deverá conter no bloco de comando o comando **return**

- **Procedimento (função sem retorno):** não retorna nenhum valor
  - Será do tipo **void** e não terá nenhum retorno

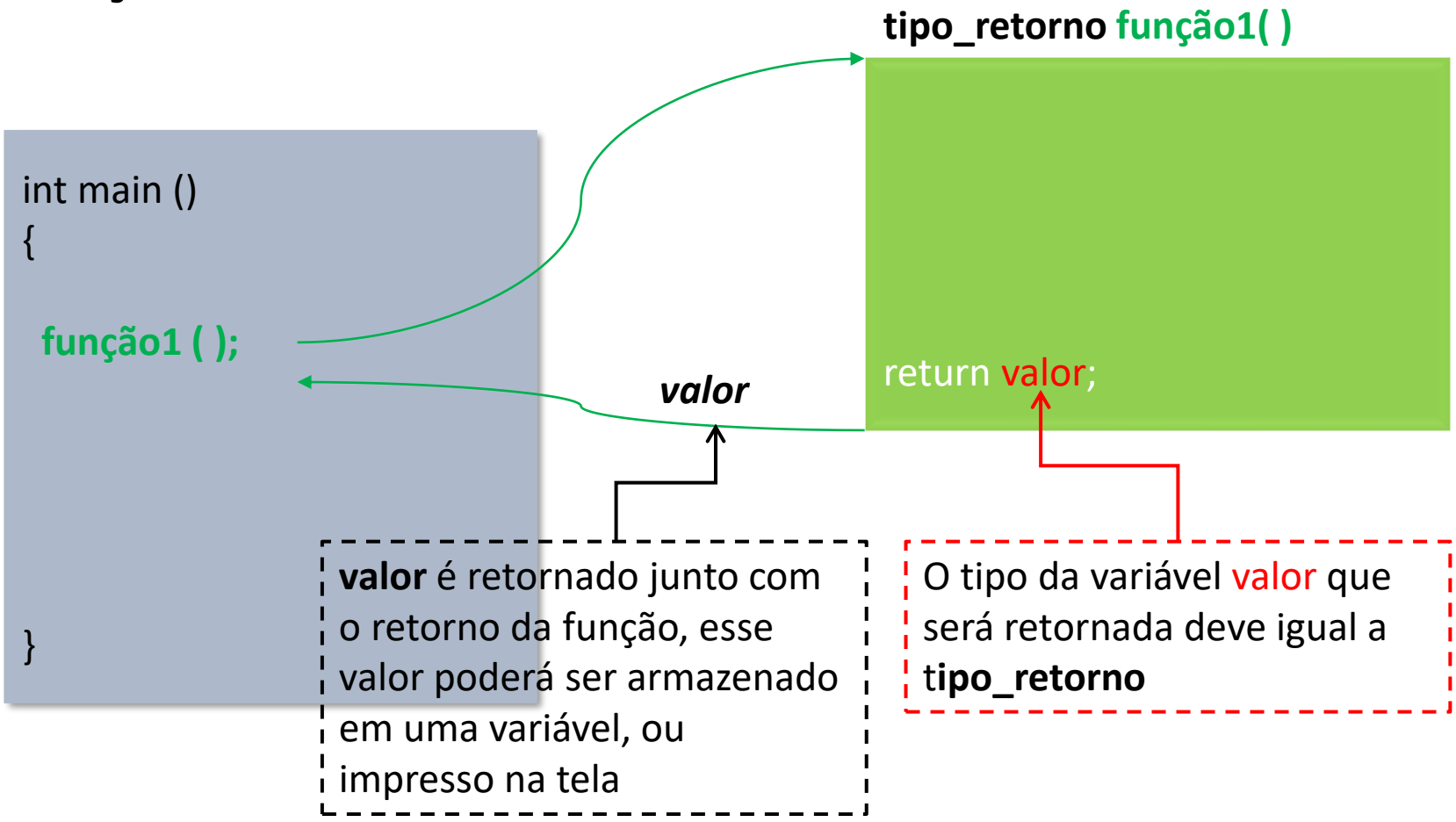
Chamando uma função

- Um programa pode conter uma ou mais funções, das quais uma delas deve ser a função **main (Função Principal)**
- A execução do programa sempre começa na função **main** e,
  - quando o fluxo de controle do programa encontra uma instrução que inclui o nome de uma função, a função é chamada;
  - o fluxo de controle executa as intruções da função;
  - depois que a função é executada, o fluxo de controle retorna para o ponto onde ocorreu a chama da função. Neste caso, retorna para a função main;
- A figura a seguir busca representar a chamada de uma função
- É possível termos chamadas de função dentro de outras funções, e o processo de execução será o mesmo

Chamando uma função



Chamando uma função:
Função com retorno



Protótipo de Função

- Uma função não pode ser chamada sem antes ter sido declarada
- A declaração de uma função é dita ***protótipo da função***
  - Deve ser colocada no início do programa, antes da main(), e estabelece seu nome, os parâmetros que ela recebe e o retorno
  - Ou podemos agrupar em um arquivo *header* (.h) para criar um conjunto de funções próprias (podemos montar nossas próprias bibliotecas!)
- O protótipo é exatamente a primeira linha da criação da função, com ponto e vírgula no final

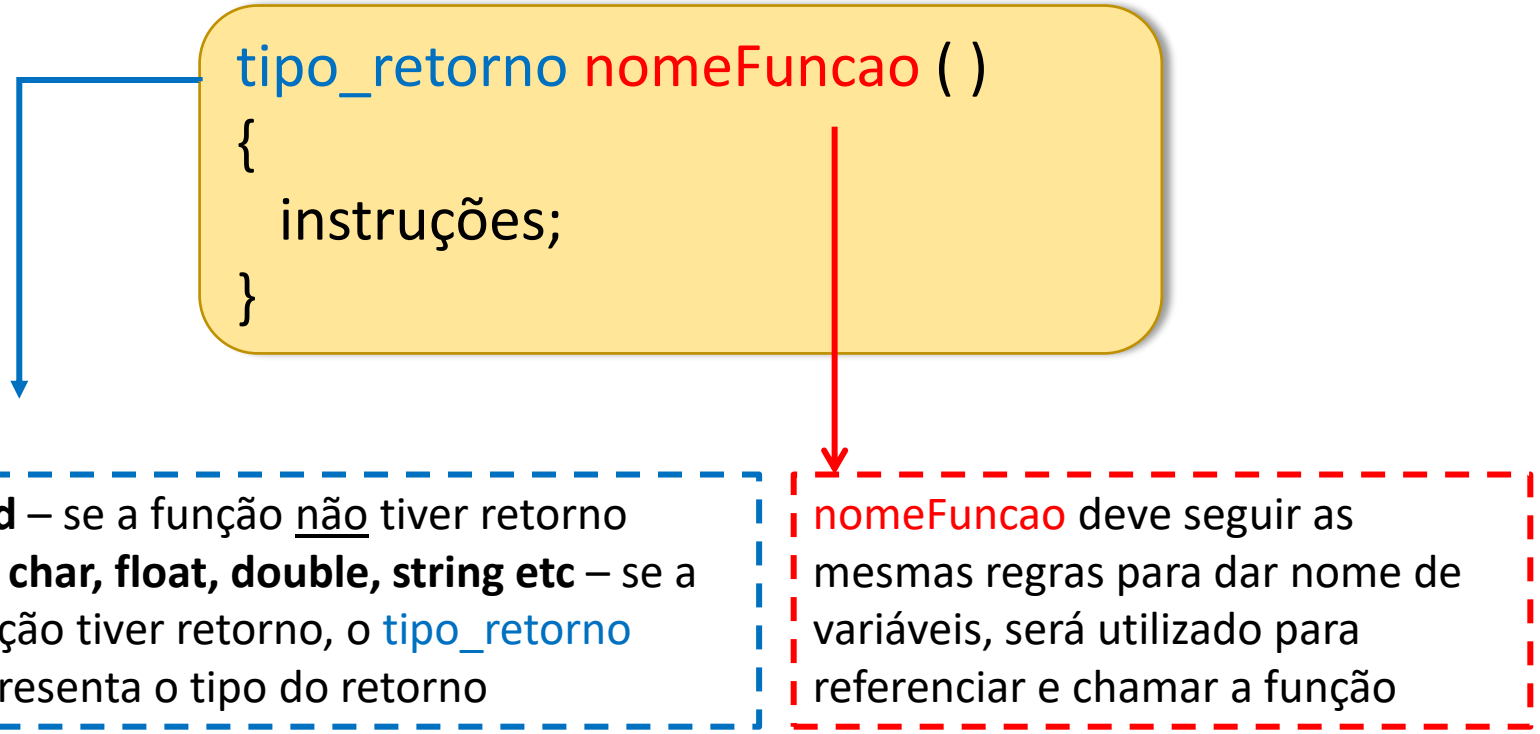
```
tipo_retorno nomeFuncao ( );
```

- ou

```
tipo_retorno nomeFuncao ( parametros );
```

Criação da Função

- As funções devem ser definidas antes ou depois da função principal, a main( )
- Ou podem ser definidas em arquivo .c em separado



Tipos de Funções

- As funções podem ou não ter parâmetros, bem como podem ou não ter retorno.
- Exemplos de combinações:
  1. Função com retorno e sem parâmetro
  2. Função com retorno e com parâmetro
  3. Função (procedimento) sem retorno e sem parâmetro
  4. Função (procedimento) sem retorno e com parâmetro

1- Função com retorno e sem parâmetro

- Essa função não recebe nenhum argumento (valor), mas retorna um valor (que geralmente, foi calculado dentro da função)
- Exemplo:

tipo da função indica que o retorno deverá ser float

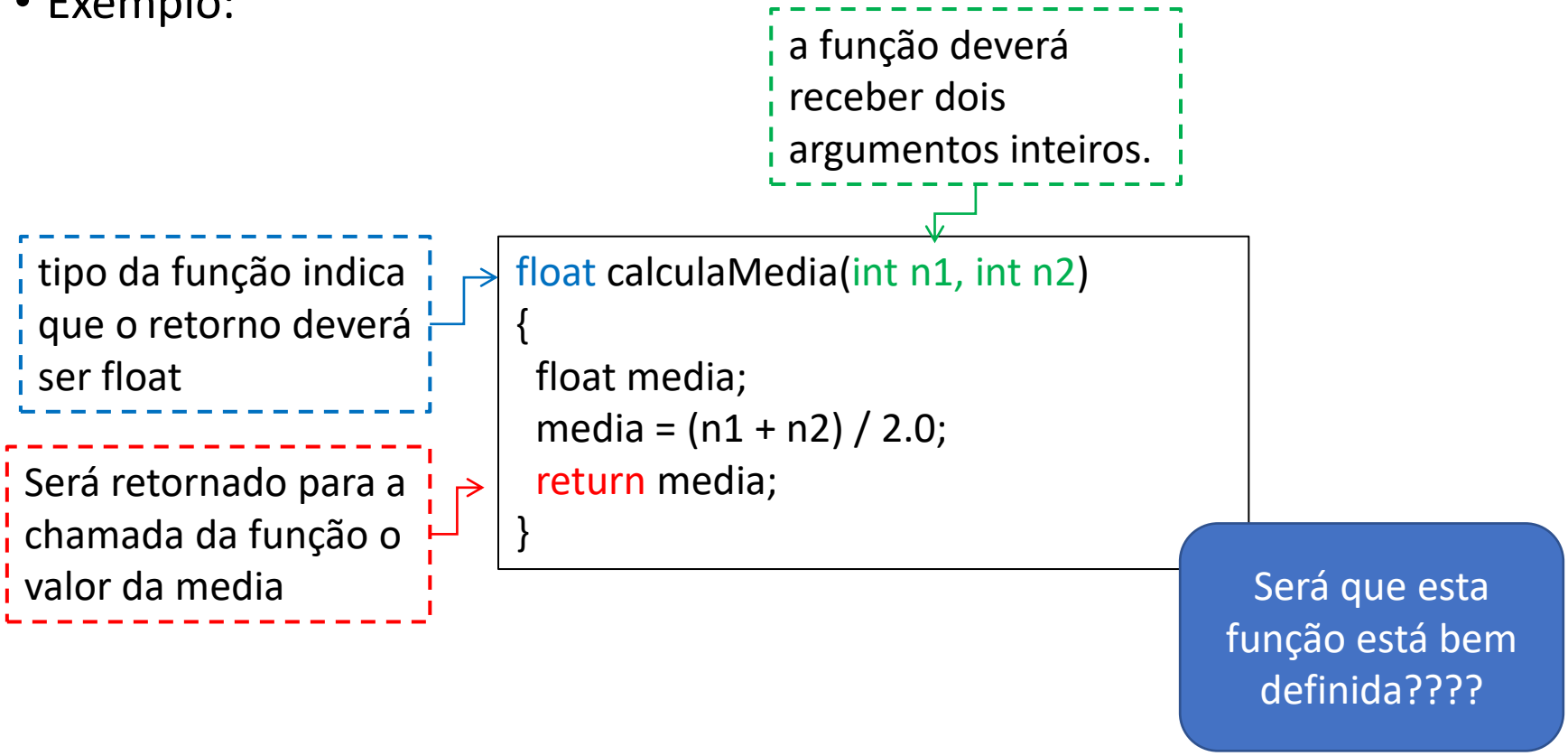
```
float calculaMedia()
{
  float num1, num2, media;
  printf("Digite o primeiro numero:");
  scanf("%f", &num1);
  printf("Digite o segundo numero:");
  scanf("%f", &num2);
  media = (num1 + num2)/2;
  return media;
}
```

Será retornado para a chamada da função o valor da media

Será que esta função está bem definida????

2- Função com retorno e com parâmetro

- Essa função recebe um ou mais parâmetros/argumentos e retorna um valor (é necessário o uso da palavra chave **return**)
- Exemplo:



3- Função sem retorno e sem parâmetro

- Essa função não recebe nenhum argumento (valor), e não retorna um valor (será sempre do tipo **void**)
- Exemplo:

O tipo void indica que a função não irá retornar um valor

```
void calculaMedia()
{
  float num1, num2, media;
  printf("Digite o primeiro numero:");
  scanf("%f", &num1);
  printf("Digite o segundo numero:");
  scanf("%f", &num2);
  media = (num1 + num2)/2;
  printf(”A media e: %.2f”, media);
}
```

Será que esta função está bem definida????

4- Função sem retorno e com parâmetro

- Essa função recebe um ou mais parâmetros/argumentos, mas não retorna um valor (será do tipo void)
- Exemplo:

a função possui dois parâmetros e irá receber dois argumentos inteiros

O tipo void indica que a função não irá retornar um valor, portanto, o valor calculado deverá ser impresso aqui mesmo

```
void calculaMedia(int n1, int n2)
{
  float media;
  media = (n1 + n2) / 2.0;
   printf("A media e: %.2f", media);
}
```

Será que esta função está bem definida????

Passagem de parâmetros

- Uma função pode receber um ou mais valores, chamados de parâmetros
- O valores que são passados a cada um dos parâmetros são chamados de argumentos
- As variáveis na declaração da função são chamados parâmetros formais
- Os valores passados a cada parâmetros da função na chamada são chamados parâmetros reais (ou argumentos)
- Na criação da função é necessário indicar o tipo de cada parâmetro
- Na chamada da função, não é necessário indicar o tipo de dado, apenas passar o valor e/ou a variável
- A passagem de parâmetro pode ser:
  - Por valor
  - Por referência

Parâmetros por Valor

- Passar uma variável como parâmetro para uma função por valor significa passar uma cópia da variável para a função

- Quaisquer alterações que ocorrem dentro da função para o parâmetro não tem nenhum efeito nos dados originais armazenados na variável

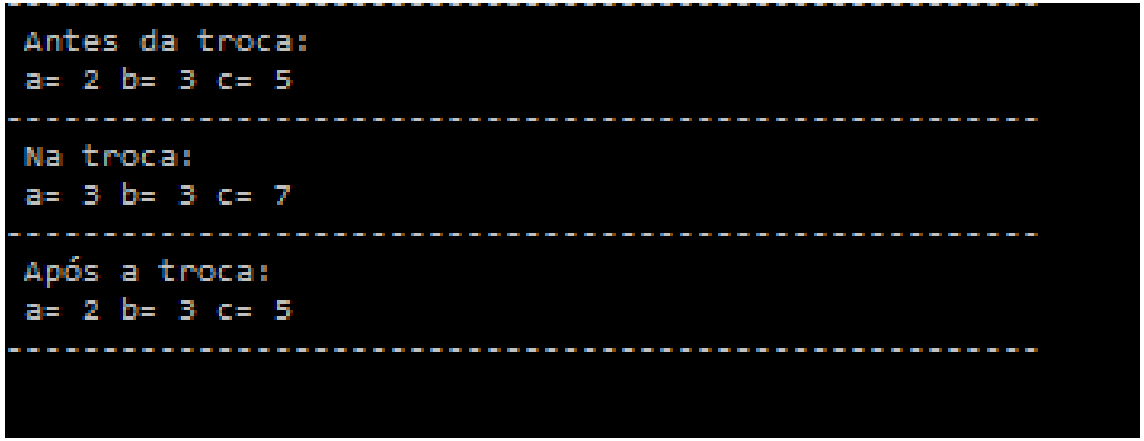
- Portanto, as alterações dentro da função não são refletidas no ambiente da sua chamada

Exemplo

```
void troca(int a, int b, int c);
int main()
{
  int a = 2, b = 3, c = 5;
  printf("\n");
  printf("-------------------------------------------------");
  printf("\n Antes da troca: \n a= %d b= %d c= %d ", a, b, c);
  printf("\n");
  printf("-------------------------------------------------");
  troca(a, b, c);
  printf("\n Apos a troca: \n a= %d b= %d c= %d ", a, b, c);
  printf("\n");
  printf("-------------------------------------------------");
  return 0;
}
void troca(int a, int b, int c)
{
  a = 3;
  c = a + 4;
  printf("\n Na troca: \n a= %d b= %d c= %d ", a, b, c);
  printf("\n");
  printf("-------------------------------------------------");
}
```

Exemplo

- Resultado:



Observe que o valor de todas as variáveis a, b e c são os mesmos antes e depois da função troca

Parâmetros por Referência

- Passagem por referência permite que a função altere o valor dos parâmetros e essa alteração persista no ambiente de chamada

- Para passar um parâmetro por referência na linguagem C, o parâmetro deve ser um ponteiro, ou seja:
  - Na **criação da função** use o operador ***** (é um ponteiro - indicar o tipo de dado e o operador * para as variáveis que serão referência)
  - Na **chamada da função** use o operador **&**

Parâmetros por Referência

- Exemplo:

```
void calculaMedia(int, int, float * );
int main()
{
  int num1, num2; float media;
  printf("Digite o primeiro valor:")
  scanf("%d", &num1);
  printf("Digite o segundo valor:")
  scanf("%d", &num2);
  calculaMedia(num1, num2,  &media);
  printf("A media e:  %.1f", media);
  return 0;
}
void calculaMedia(int n1, int n2, float *m)
{
   *m = (n1+ n2 ) / 2.0;
}
```

Nesse exemplo, a média não é retornada na função calculaMedia(), mas como o parâmetro é por referência, qualquer alteração no valor da média dentro da função é refletido no ambiente da chamada. Observe que no protótipo apenas indicamos com tipo + * a variável que será parâmetro por referência

Na chamada da função usamos o operador &, observe que apenas a variável média foi passada por referência

Na criação da função usamos o operador *

Vetor e Matriz como parâmetros

- O vetor e matriz já são considerados referências e **não** precisam do operador **&** na passagem do parâmetro – lembrem-se que são ponteiros para o primeiro elemento!
- Com isso, qualquer alteração em um vetor e matriz dentro de uma função será refletida no ambiente que chamou a função
- Na chamada da função indicamos apenas o nome do vetor e/ou matriz
- Já na criação da função indicamos que o parâmetro é vetor ou matriz com o uso dos colchetes
  - No caso dos vetores não é necessário colocar o tamanho
  - Já no caso das matrizes, é necessário indicar o tamanho da segunda dimensão (ou seja, das colunas)

Exemplo

```
#define N 6
void entradavetor(int v[ ]);
void somavetor(int v[ ], int * );
void imprimevetor(int v[ ]);

int main()
{
  int vet[N];
  int soma = 0;
  entradavetor(vet);
  somavetor(vet, &soma);
  printf("A soma dos valores é: %d",soma);
  imprimevetor(vet);
  return 0;
}
```

Observe:

- ✓ a soma teve que ser referenciada (&) e (*) mas o vetor não.
- ✓ o vetor é criado (declarado) dentro da função principal (main)

```
void entradavetor(int v[ ])
{
  int i;
  for(i=0; i<N; i++)
   {
     printf("Digite um numero: ");
     scanf("%d", &v[ i ]);
   }
}
void somavetor(int v[ ], int *s)
{
  int i;
  for(i = 0; i < N; i++)
   {
     *s = *s + v[ i ];
   }
}
void imprimevetor(int v[ ])
{
  int i;
  printf("Os valores do vetor são: ");
  for(i = 0; i < N; i++)
   {
     printf(" %d | ",v[ i ]);
   }
}
```

Vetor e Matriz como parâmetros

Matrizes:

- Se as dimensões estiverem disponíveis globalmente, podemos usá-las na declaração da função.

Vetor e Matriz como parâmetros

```
#include <stdio.h>
#define M 3;
#define N 3;

void print(int arr[M][N])
{
   int i, j;
   for (i = 0; i < M; i++) {
     for (j = 0; j < N; j++)
       printf("%d ", arr[i][j]);
     printf(“\n");
   }
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   print(arr);
   return 0;
}
```

Vetor e Matriz como parâmetros

Matrizes:

- Se a segunda dimensão estiver disponível globalmente, podemos usá-la na declaração da função.

Vetor e Matriz como parâmetros

```
#include <stdio.h>
#define N 3;

void print(int arr[][N], int m)
{
   int i, j;
   for (i = 0; i < m; i++) {
     for (j = 0; j < N; j++)
       printf("%d ", arr[i][j]);
     printf(“\n");
   }
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   print(arr, 3);
   return 0;
}
```

Vetor e Matriz como parâmetros

Matrizes:

- Se declaramos as dimensões como parâmetros da função antes do parâmetro da matriz, podemos usar os próprios parâmetros. Essa forma usa os aspectos do VLA.

Vetor e Matriz como parâmetros

```
#include <stdio.h>

void print(int m, int n, int arr[][n])
{
   int i, j;
   for (i = 0; i < m; i++) {
     for (j = 0; j < n; j++)
       printf("%d ", arr[i][j]);
      printf(“\n");
   }
}

int main()
{
   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int m = 3, n = 3;
   print(m, n, arr);
   return 0;
}
```

Vetor e Matriz como parâmetros

Matrizes:

- Podemos trabalhar com a matriz como sendo um ponteiro. Nesse caso, devemos explorar a aritmética de ponteiros e realizar a conversão de tipo na passagem de parâmetros.

Vetor e Matriz como parâmetros

```
#include <stdio.h>
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
      printf("\n");
    }
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;
    print((int *)arr, m, n);
    return 0;
}
```

Vetor e Matriz como parâmetros

Matrizes:

- Podemos trabalhar com a matriz como sendo, também:
  - um ponteiro para ponteiro
  - um array de ponteiros

Arquivos Header

- Podemos organizar os protótipos de funções em um arquivo para que possam ser utilizadas por programas.
- Para isso, declaramos as funções nos arquivos header (.h) e definimos as funções nos arquivos .c.

Arquivos Header

- Por exemplo, suponha que queiramos criar um conjunto de funções Podemos organizar os protótipos de funções em um arquivo para que possam ser utilizadas por programas.
- Vamos criar os arquivos array_util.h e array_util.c

Arquivos Header

- array_util.h

```
#ifndef ARRAY_UTIL_H_INCLUDED
#define ARRAY_UTIL_H_INCLUDED

void printArray(int n, int v[]);

#endif // ARRAY_UTIL_H_INCLUDED
```

- array_util.c

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

void printArray(int n, int v[]){
    for(int i = 0; i < n; i++) {
        printf("%d ", v[i]);
    }
}
```

Arquivos Header

- main.c

```
#include "array_util.h"

int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    printArray(5, arr);
    return 0;
}
```

Recursividade

Recursividade

- Considere o problema que envolve a soma dos números de 1 a 10. Podemos resolver este problema de forma **iterativa** usando estrutura de repetição:

```
#include <stdio.h>

int somaIterativo(int max) {
    int soma = 0;
    for(int i = 1; i <= max; i++)
        soma = soma + i;
    return soma;
}

int main()
{
    int soma = somaIterativo(10);
    printf("%d\n", soma);
    return 0;
}
```

Recursividade

- De maneira similar, ainda de forma **iterativa** usando estrutura de repetição:

```
#include <stdio.h>

int somaIterativo(int max) {
    int soma = 0;
    for(int i = max; i > 0; i--)
        soma = soma + i;
    return soma;
}

int main()
{
    int soma = somaIterativo(10);
    printf("%d\n", soma);
    return 0;
}
```

Recursividade

- Podemos resolver o mesmo problema utilizando uma abordagem diferente: de forma **recursiva**!

- Recursão é a técnica de fazer com que uma função chame (invoque) a si própria.

- A técnica de recursão proporciona uma forma de resolver problemas dividindo-o em subproblemas menores, que são mais fáceis de resolver (e entender).

Recursividade

- Considere o problema que envolve a soma dos números de 1 a 10. Vamos resolvê-lo de forma **recursiva**:

```
#include <stdio.h>

int somaRecursivo(int max) {
    if (max > 0)
        return max + somaRecursivo(max - 1);
    else
        return 0;
}

int main()
{
    int soma = somaRecursivo(10);
    printf("%d\n", soma);
    return 0;
}
```

Recursividade

```
int somaRecursivo(int max) {
    if (max > 0)
        return max + somaRecursivo(max - 1);
    else
        return 0;
}
```

- Quando a função somaRecursivo() é chamada, ela adiciona o valor max à soma de todos os números menores que max (chamando a si própria) e retorna o resultado.
- Quando max é zero (critério de parada), a função não chama a si própria e retorna 0. Veja a execução a seguir:

10 + somaRecursivo (9)

10 + (9 + somaRecursivo (8))

10 + (9 + (8 + somaRecursivo (7)))

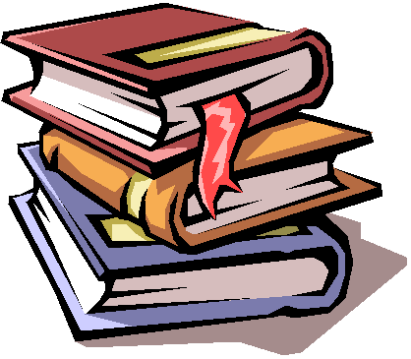
...

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + somaRecursivo (0)

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Recursividade

- Ao trabalharmos com recursão, devemos ter cuidado para garantir que a função irá terminar em algum ponto, isto é, de acordo com alguma condição (critério de parada).
- Caso contrário, a função irá chamar a si mesmo indefinidamente!
- Outro ponto importante é que a recursão que envolve um grande número de chamadas recursivas e com grau elevado de processamento pode requisitar uma quantidade excessiva de memória e processamento.
- Ainda assim, quando escrita corretamente, funções recursivas podem ser eficientes e uma abordagem elegante de programação.



Referência Bibliográfica

- MIZRAHI, Victorine Viviane. **Treinamento em linguagem C.** São Paulo:Pearson Prentice Hall, 2008. 2ª edição. Curso Completo. Capítulos 5, 7 e 9.

- ASCENCIO, Ana Fernanda Gomes e CAMPOS, Edilene A. Veneruchi. **Fundamentos da Programação de Computadores – Algoritmos, Pascal, C/C++ e Java**. São Paulo: Pearson Prentice Hall, 2012. 3ª Edição.