

Programação e Desenvolvimento de Software I

Funções

Aula de hoje

Funções

- Definição e estrutura básica
 - Declaração
 - Parâmetros
 - Corpo
 - Retorno
- Tipos de passagem de parâmetros
 - Por valor
 - Por referência
 - Arrays
 - Estruturas

Função

Blocos de código que podem ser nomeados e chamados de dentro de um programa

Sequência de comandos que recebe um nome

Pode ser chamada de qualquer parte do programa, quantas vezes forem necessárias

Ex:

- **printf():** função que escreve na tela
- **scanf():** função que lê valores do teclado

Função

Facilitam a estruturação e reutilização do código

- **Estruturação:** programas grandes e complexos são construídos bloco a bloco
- **Reutilização:** o uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros

Exemplo - motivação

Exemplo de programa:

Escrever um programa que imprima o seguinte:

Números entre 1 e 5

1

2

3

4

5

Exemplo - motivação


```
int main()
{
    int i;
    for (i=1; i<=20; i++)
        putchar('*');
    putchar('\n');
    printf("Números entre 1 e 5\n");
    for (i=1; i<=20; i++)
        putchar('*');
    putchar('\n');
    for (i=1; i<=5; i++)
        printf("%d\n", i);
    for (i=1; i<=20; i++)
        putchar('*');
    putchar('\n');
}
```

código
repetido

O ideal seria não precisar repetir o código e ainda assim obter o mesmo efeito!


Exemplo - motivação

```
void linha()  
{  
    int i;  
    for (i=1; i<=20; i++)  
        putchar('*');  
        putchar('\n');  
}
```



definição do
procedimento
linha()

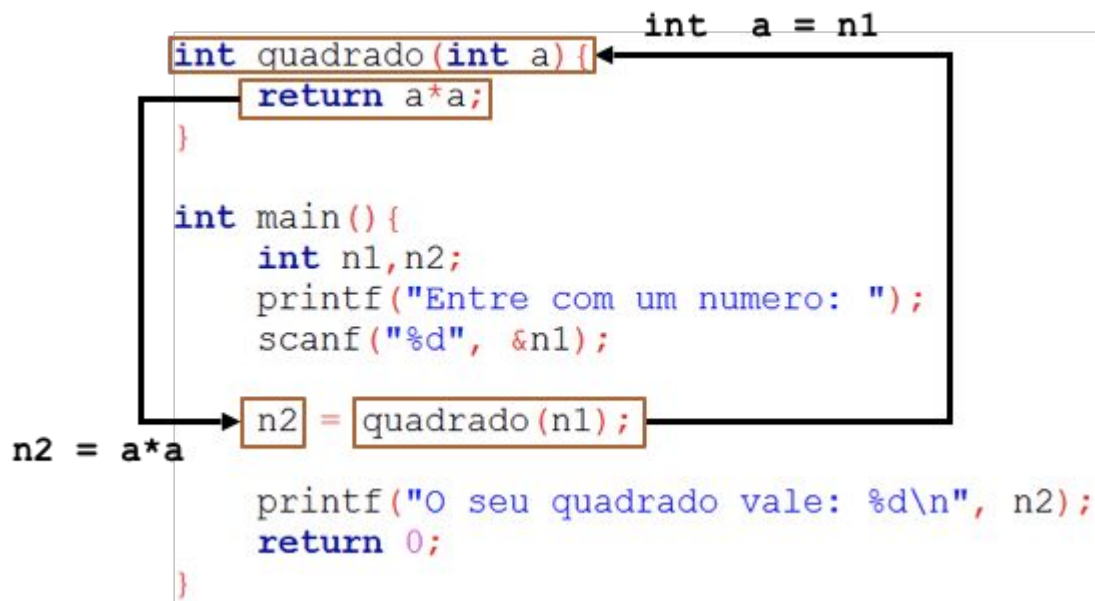
```
int main(){  
    int i;  
    linha();  
    printf("Números entre 1 e 5\n");  
    linha();  
    for (i=1; i<=5; i++)  
        printf("%d\n",i);  
    linha();  
}
```



chamada do
procedimento
linha ()

Função - ordem de execução

Ao chamar uma função, o programa que a chamou é pausado até que a função termine 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 cláusula **main**

- O protótipo apenas indica a existência da função
- Desse modo ela pode ser declarada após a função main()

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

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;
}
```

Declaração de funções

```
Especificador_de_tipo nome_da_função ( lista de parâmetros ) {  
    corpo da função  
}
```

- ⇒ Especificador_de_tipo - especifica o tipo de valor que o comando return da função devolve, podendo ser qualquer tipo válido
- ⇒ nome_da_função - é um identificador escolhido pelo programador que não se deve repetir; segue as mesmas regras para definição de identificadores
- ⇒ lista de parâmetros - é uma lista de nomes e tipos de variáveis separadas por vírgulas, que recebem os valores dos argumentos quando a função é chamada. Todos os parâmetros da função devem incluir o tipo e o nome da variável

```
int valor_absoluto (int x) {  
    if (x<0)  
        return -x;  
    else  
        return x;  
}
```

Declaração de funções

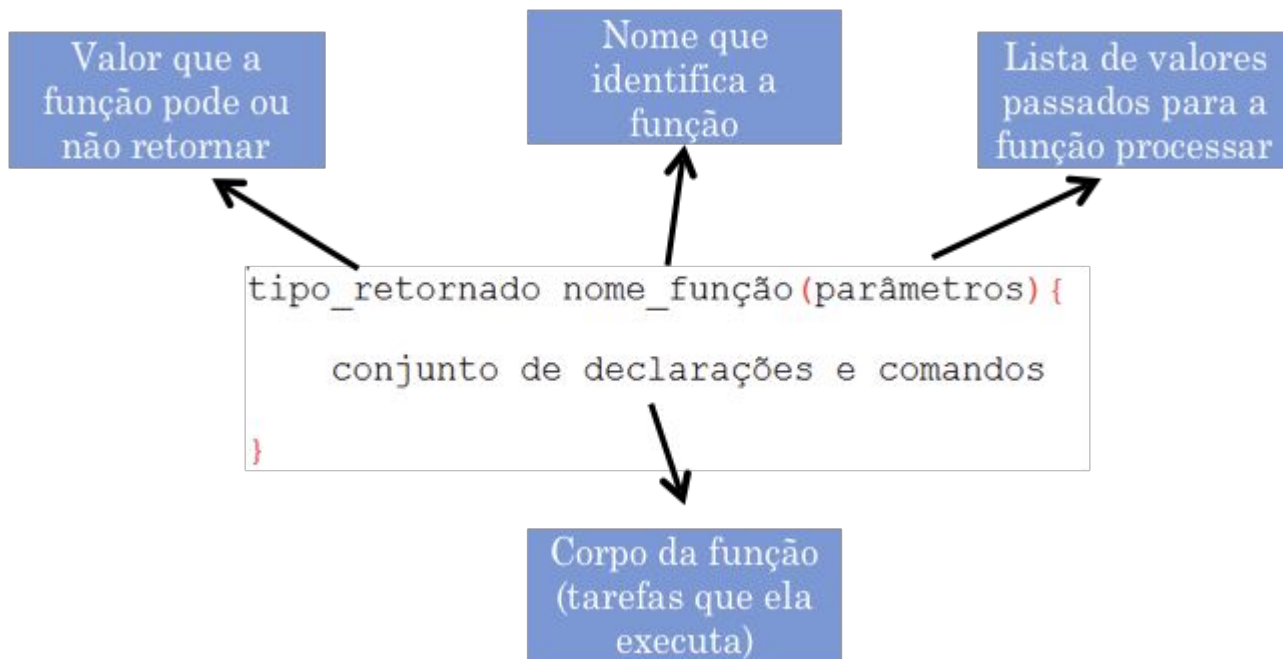
```
Especificador_de_tipo nome_da_função ( lista de parâmetros ) {  
    corpo da função  
}
```

- ↳ Especificador_de_tipo - especifica o tipo de valor que o comando return da função devolve, podendo ser qualquer tipo válido
- ↳ nome_da_função - é um identificador escolhido pelo programador que não se deve repetir; segue as mesmas regras para definição de identificadores
- ↳ lista de parâmetros - é uma lista de nomes e tipos de variáveis separadas por vírgulas, que recebem os valores dos argumentos quando a função é chamada. Todos os parâmetros da função devem incluir o tipo e o nome da variável

```
float potencia (float base, int expoente) {  
    int i;  
    float resultado = 1;  
    if (expoente == 0)  
        return 1;  
    for (i = 1; i <= expoente; i++)  
        resultado = resultado * base;  
    return resultado;  
}
```

Função - estrutura

Assim, a forma geral de uma função é:



Função - corpo

O corpo da função é o trecho de código que contém a lógica de funcionamento da função

- É formado pelos comandos que a função deve executar
- Ele processa os parâmetros (se houver), realiza outras tarefas e gera saídas (se necessário)
- Por exemplo:

```
int main() {  
    //conjunto de declarações e comandos  
    return 0;  
}
```

Função - corpo


De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função

- Uma função é construída com o intuito de realizar uma única tarefa, específica e bem-definida
- As operações de entrada/saída de dados (e.g. usando **scanf()** e **printf()**) devem ser feitas antes da chamada à função (e.g. na **main()**)
- Isso assegura que a função construída possa ser utilizada nas mais diversas aplicações, garantindo 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 achar necessários



```
//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 (i.e. de quem o 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

Podemos retornar qualquer valor válido em C

- Tipos pré-definidos: int, char, float, double
- Tipos definidos pelo usuário: struct

Função - retorno

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

Função - retorno

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

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

Função - retorno

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");  
}
```

Escopo

Funções também estão sujeitas ao escopo das variáveis

O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa

- Variáveis locais
- Parâmetros formais
- Variáveis globais

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:, na função abaixo

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

x é um **parâmetro** formal da função **quadrado**

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, terá preferência a variável local em relação à variável global

- **EVITE O USO DE VARIÁVEIS GLOBAIS!**

Passagem de parâmetros

Funções - passagem de parâmetros

Na linguagem C, os parâmetros de uma função são sempre passados por **valor**

- Assim, é feita uma cópia do valor do parâmetro e essa cópia é passada para a função

Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função

Funções - 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

Funções - passagem por referência

Quando se quer que o valor da variável mude dentro da função:

- Usa-se a passagem de parâmetros por referência

Neste tipo de chamada, não se passa para a função o valor da variável, mas sua **referência** (seu endereço de memória)

Funções - passagem por referência

Ex: função **scanf()**

- Sempre que desejamos ler algo do teclado, passamos para a função **scanf()** o nome da variável onde o dado será armazenado
- Essa variável tem seu valor modificado dentro da função **scanf()**, e seu valor pode ser acessado no programa principal

```
int main(){  
    int x = 5;  
    printf("Antes do scanf: x = %d\n",x);  
    printf("Digite um numero: ");  
    scanf("%d",&x);  
    printf("Depois do scanf: x = %d\n",x);  
  
    return 0;  
}
```


Funções - passagem por referência

- Para passar um parâmetro por referência, coloca-se um asterisco "*" na frente do nome do parâmetro na declaração da função

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

- 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);
```

Funções - 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;
}
```

Funções - passagem por referência

```
int n = x;

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

Funções - exercício

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

```
void Troca (int*a,int*b) {  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Funções - vetores como parâmetros

Para utilizar vetores como parâmetros de funções alguns cuidados simples são necessários

Vetores são sempre passados por referência para uma função

- A passagem de arrays **por referência** evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função
 - Essa cópia de dados afeta o desempenho do programa

Funções - vetores como parâmetros

É necessário declarar um segundo parâmetro (em geral uma variável inteira)

- O conteúdo dessa variável é o tamanho do vetor

Quando passados um vetor por parâmetro, independente do seu tipo, o que é de fato passado é o endereço do primeiro elemento do vetor

Funções - vetores como parâmetros

Na passagem de um vetor como parâmetro de uma função podemos declarar a função de diferentes maneiras

No exemplo abaixo, todas as três formas são equivalentes:

```
void imprime(int *m, int n);  
void imprime(int m[], int n);  
void imprime(int m[5], int n);
```

Funções - vetores como parâmetros

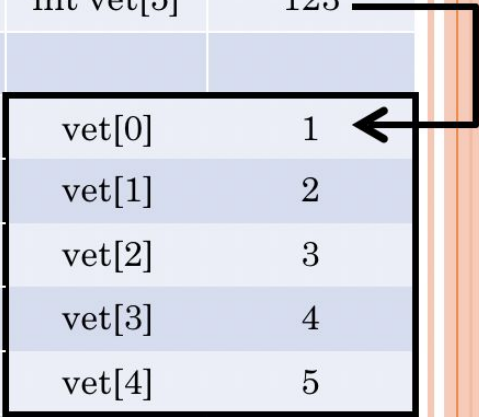
Exemplo de uma função que imprime um vetor:

```
void imprime(int *m, int n){
    int i;
    for (i=0; i< n;i++)
        printf ("%d \n", m[i]);
}

int main (){
    int vet[5] = {1,2,3,4,5};
    imprime(vet,5);

    return 0;
}
```

Memória		
posição	variável	conteúdo
119		
120		
121	int vet[5]	123
122		
123	vet[0]	1
124	vet[1]	2
125	vet[2]	3
126	vet[3]	4
127	vet[4]	5
128		



Funções - vetores como parâmetros

Vimos que para vetores, não é necessário especificar o número de elementos para a função

```
void imprime (int*m, int n);  
void imprime (int m[], int n);
```

No entanto, para vetores com mais que uma dimensão, é necessário especificar o tamanho de todas as dimensões, exceto a primeira

```
void imprime (int m[][5], int n);
```

Funções - vetores como parâmetros

Na passagem de um vetor para uma função, o compilador precisa saber o tamanho de cada elemento, não o número de elementos

Uma matriz pode ser interpretada como um vetor de vetores

- **int m[4][5]**: vetor de 4 elementos onde cada elemento é um vetor de 5 posições inteiras

Funções - vetores como parâmetros

Logo, o compilador precisa saber o tamanho de cada elemento do array

```
int m[4][5]  
  
void imprime (int m[][5], int n);
```

No exemplo acima, informamos ao compilador que estamos passando um vetor, onde cada elemento dele é outro vetor de 5 posições inteiras

Funções - vetores como parâmetros

Isso é necessário para que o programa saiba que o vetor possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão

As notações abaixo funcionam para vetores com mais de uma dimensão. Mas o vetor é tratado como se tivesse apenas uma dimensão dentro da função

```
void imprime (int*m, int n);  
void imprime (int m[], int n);
```

Funções - struct como parâmetro

Podemos passar uma struct por valor ou por referência

Temos duas possibilidades:

- Passar por parâmetro toda a struct
- Passar por parâmetro apenas um campo específico da struct

Funções - struct como parâmetro

Passar por parâmetro apenas **um campo específico da struct**:

- Valem as mesmas regras vistas até o momento
- Cada campo da struct é como uma variável independente. Ela pode, portanto, ser passada individualmente por **valor** ou por **referência**

Funções - struct como parâmetro

Passar por parâmetro **toda a struct**

- **Passagem por valor**
 - Valem as mesmas regras vistas até o momento
 - A struct é tratada como uma variável qualquer e seu valor é copiado para dentro da função
- **Passagem por referência**
 - Valem as regras de uso do asterisco "*" e operador de endereço "&"
 - Devemos acessar o conteúdo da struct para somente depois acessar os seus campos e modificá-los
 - Uma alternativa é usar o **operador seta "->"**

Funções - struct como parâmetro

Usando “*”

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p) {  
    (*p).x = 10;  
    (*p).y = 20;  
}  
  
struct ponto p1;  
atribui(&p1);
```

Usando “->”

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p) {  
    p->x = 10;  
    p->y = 20;  
}  
  
struct ponto p1;  
atribui(&p1);
```


Resumo da aula

Resumo - funções (Capítulo 9)

Reduz a complexidade de um programa

- Elaborar a solução em partes pequenas e bem definidas
- Uma tarefa complexa é dividida em funções específicas

Evita a repetição de código ao longo do programa

- Diminui o tamanho do código
- Menos erros, menor custo, menor tempo de programação

Permite reutilização de código

- Código já testado, sem erros => custo menor de programação e maior confiabilidade