Equipe 6 – Funções e modularização

Alunos: Daniel, Italo, Samuel, Fabricio

Função

▼ O Que é?

No universo da programação, uma função pode ser entendida como uma "caixinha" ou um bloco de código que agrupa um conjunto específico de instruções. O principal objetivo de uma função é executar uma tarefa bem definida e, idealmente, isolada do restante do programa.

Características Principais de uma Função:

1. **Nome:** Toda função possui um nome único que a identifica. É através desse nome que você "chama" ou "invoca" a função para que ela execute suas instruções.
2. **Parâmetros (Entradas de Dados):** Uma função pode receber informações externas para realizar sua tarefa. Pense neles como os "ingredientes" que você dá à sua "caixinha" para que ela possa trabalhar.
3. **Retorno (Saída de Dados):** Após executar suas instruções, uma função pode devolver um resultado ou não. Esse resultado é conhecido como valor de retorno.

```
int nome(int parametros) {
   return "Saída de dados";
}
```

▼ Vantagens X Desvantagens

Vantagens	Desvantagens
Fácil Organização: Simples de organizar em programas pequenos.	Difícil de manter em projetos grandes.

Poucos Arquivos: Não precisa de múltiplos arquivos.	Sem reaproveitamento em outros programas.
	Arquivo único pode ficar confuso.

▼ Exemplo Prático:

▼ C(Destaque)

Sem Função:

```
// Sem função
int a = 2;
int b = 3;
printf("Soma: %d\n", a + b);

int x = 5;
int y = 7;
printf("Multiplicação: %d\n", x * y);

int base = 2;
int expoente = 8;
int pot = 1;
for(int i=0;i<expoente;i++) pot *= base;
printf("Potência: %d\n", pot);

int n = -10;
if(n < 0) n = -n;
printf("Valor absoluto: %d\n", n);

int num = 16;
int raiz = 0;
for(int i=0;i<=num;i++) if(i*i==num) raiz = i;
printf("Raiz inteira: %d\n", raiz);

int maior = 10;
int menor = 3;
if(menor > maior) maior = menor;
printf("Maior: %d\n", maior);
```

Saida:

```
Resultado: 5
Multiplicação: 35
```

Com Função:

```
// Com função
int soma(int a, int b) { return a + b; }
int multiplica(int x, int y) { return x * y; }
int pot(int base, int exp) { int r=1; for(int i=0;i<exp;i++) r*=base; retur
n r; }
int absoluto(int n) { return n<0?-n:n; }
int raizint(int n) { int i=0; while(i*i<n) i++; return i*i==n?i:0; }
int maior(int a, int b) { return a>b?a:b; }
printf("Soma: %d\n", soma(2, 3));
printf("Multiplicação: %d\n", multiplica(5, 7));
printf("Potência: %d\n", pot(2,8));
printf("Valor absoluto: %d\n", absoluto(-10));
printf("Raiz inteira: %d\n", raizint(16));
printf("Maior: %d\n", maior(10,3));
```

Saida:

```
Resultado: 5
Multiplicação: 35
```

▼ JavaScript

Sem Função:

```
// Sem função
const a = 2;
const b = 3;
const resultado = a + b;
console.log('Resultado:', resultado);
```

Saida:

```
Resultado: 5
```

Com Função:

```
// Com função
function soma(a, b) {
  return a + b;
}
const resultado = soma(2, 3);
console.log('Resultado:', resultado);
```

Saida:

```
Resultado: 5
```

▼ SQL

Sem Função:

```
-- Sem função
SELECT 2 + 3 AS resultado;
```

Com Função:

```
-- Com função
CREATE FUNCTION soma(a INT, b INT) RETURNS INT
BEGIN
  RETURN a + b;
END;
-- Uso:
SELECT soma(2, 3) AS resultado;
```

Modularização

▼ O Que é?

1. **Definição:** É a prática de dividir um programa em módulos menores (arquivos, pacotes, bibliotecas, funções) para melhorar organização,

reutilização e manutenção.

2. **Diferença de apenas usar funções:**
    a. **Função isolada:** Só organiza um pedaço de código.
    b. **Modularização:** organiza o programa inteiro em partes independentes que podem ser importadas, reutilizadas e testadas separadamente.
    c. **Em C:** Isso geralmente significa dividir o programa em múltiplos arquivos `.c` e `.h`, cada um representando um módulo.

▼ Vantagens X Desvantagens

Vantagens	Desvantagens
Organização do código: mais limpo e compreensível.	**Exige mais arquivos:** Isso pode ser um problema para projetos pequenos.
Reutilização: um módulo pode ser usado em outros projetos.	**Mais difícil para iniciantes entenderem a estrutura:** Isso pode ser um problema para projetos pequenos.
Manutenção facilitada: corrigir ou melhorar só um módulo, sem mexer no resto.	
Trabalho em equipe: cada pessoa pode trabalhar em um módulo diferente.	
Escalável para projetos grandes: pode ser facilmente expandido.	
Facilidade de teste: cada módulo pode ser testado separadamente.	
Facilidade de depuração: erros podem ser encontrados e corrigidos mais rapidamente.	
Redução de custos: menos trabalho duplicado.	

▼ Exemplo Prático:

▼ C(Destaque)

Como extruturar:

```
/meu_programa
   ├── main.c
   ├── matematica.h
   ├── matematica.c
```

Arquivo `matematica.h` (cabeçalho - interface do módulo)

```
#ifndef MATEMATICA_H
#define MATEMATICA_H

int somar(int a, int b);
int subtrair(int a, int b);
int multiplicar(int a, int b);
float dividir(int a, int b);

#endif/
```

Arquivo `matematica.c` (cabeçalho - implementação - módulo em si)

```
#include "matematica.h"

int somar(int a, int b) {
   return a + b;
}

int subtrair(int a, int b) {
   return a - b;
}

int multiplicar(int a, int b) {
   return a * b;
}

float dividir(int a, int b) {
   if (b == 0) return 0; // evitar divisão por zero
   return (float) a / b;
}
```

Arquivo `main.c` (programa principal)

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

int main() {
    int x = 10, y = 5;

    printf("Soma: %d
", somar(x, y));
    printf("Subtração: %d
", subtrair(x, y));
    printf("Multiplicação: %d
", multiplicar(x, y));
    printf("Divisão: %.2f
", dividir(x, y));

    return 0;
}
```

Saída:

```
Soma: 15
Subtração: 5
Multiplicação: 50
Divisão: 2
```

▼ JavaScript

Exemplo em JavaScript (NestJS/Angular)

1. Em **NestJS:** Cada `@Module` agrupa controllers, services, providers relacionados.
2. Em **Angular:** Cada `NgModule` organiza componentes, diretivas, serviços.

NestJS(Framework Back-End)

```
@Module({
  controllers: [UserController],
  providers: [UserService],
  exports: [UserService]
})
export class UserModule {}
```

Angular(Framework Front-End

```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Conclusão:

- **Em C, modularização:** dividir em múltiplos arquivos `.c` + `.h`.
- **Em JS/TS (Nest/Angular), modularização:** Já é nativa, cada módulo organiza parte do sistema.
- **Modularizar ≠ apenas usar funções:** É organizar todo o sistema em partes independentes.