

```
1
2
3  Curso de Linguagem 'C' {
4
5      [Curso básico de Linguagem C]
6
7
8
9      { Alison Almeida }
10
11
12  }
13
14
```

Conteúdo 'Programático';

- Introdução a Linguagem C;
- Tipos de dados, variáveis e operadores;
- Entrada e Saída de Dados
- Estruturas de controle;
- Funções;
- Vetores e Matrizes;
- Strings;
- Ponteiros;
- Alocação Dinâmica de Memória;
- Struct's
- Unions e Enumeradores;
- Manipulação de Arquivos;
- Boas práticas e Projeto Final;

```
1 01 {
```

```
2  
3  
4  
5 [Introdução a Linguagem C]
```

```
6 { História e características de linguagem; }
```

```
7 { Compiladores e IDE's; }
```

```
8 { Estrutura básica de um programa em C; }
```

```
9  
10  
11  
12 }  
13  
14
```

```
1  historia_linguagem_c (void* args) {
```



```
5  Desenvolvida por Dennis Ritchie, nos laboratorios de  
6  Bell Labs no inicio dos anos 1970;
```



```
9  Surgiu como uma evolução da linguagem B, que por sua vez  
10 foi inspirada na linguagem BCPL;
```



```
12 Tinha o objetivo de criar uma linguagem de programação  
13 de alto nível que permitisse acesso de baixo nível à  
14 memória e hardware;
```

```
15 }
```

```
1  caracteristicas_c (void* args) {  
2  
3
```



```
5  Linguagem de proposito geral, pode ser usada para  
6  desenvolver desde sistemas operacionais até softwares  
7  embarcados;
```



```
9  Compilada, é executada em código de máquina, resultando  
10 em programas eficientes e de alto desempenho;
```



```
12 Os ponteiros são as características mas poderosas da  
13 linguagem C, permitindo a manipulação direta de endereço  
14 de memória;
```

```
15 }
```

```
1  compiladores_ide_c (void* args) {
```



```
5      GCC - Clang/LLVM - Microsoft Visual C++(MSCV);
```



```
8      Visual Studio Code - Visual Studio;
```



```
11      Code Blocks - Falcon C++ - Dev C++;
```

```
12 }  
13  
14
```

1 **estrutura_basica_c (void* args) {**
2

```
#include <stdio.h> // Biblioteca padrão de entrada e saída

int main() {
    // Declaração de variáveis
    int idade;

    // Exibição de mensagem e entrada de dados
    printf("Digite sua idade: ");
    scanf("%d", &idade);

    // Saída de dados
    printf("Você tem %d anos.\n", idade);

    return 0; // Indica que o programa terminou corretamente
}
```

13 }
14

1 02 {

2
3
4 [Entrada e saída de dados]

5 { Formatação de entrada e saída; }

6 { Funções: **printf()** e **scanf()**; }

7 { Leitura/escrita de caracteres: **getchar()** e

8
9
10 **putchar()**; }

11 }
12
13
14


```
1  formatacao_dados (void* args) {
```



A formatação de dados em C é feita com especificadores de formato, que definem como os valores são exibidos ou lidos



Esses especificadores são usados principalmente com **printf(entrada)** e **scanf(saida)**.

```
8  
9  
10  
11  
12  }  
13  
14
```

```
1 especificadores (void* args) {
```

Principais Especificadores de Formato

Especificador	Tipo de Dado	Exemplo (printf)	Saída
%d / %i	Inteiro decimal	printf("%d", 10);	10
%f	Ponto flutuante	printf("%.2f", 3.14159);	3.14
%c	Caractere	printf("%c", 'A');	A
%s	String (texto)	printf("%s", "Ola");	Ola
%x / %X	Inteiro hexadecimal	printf("%x", 255);	ff
%o	Inteiro octal	printf("%o", 10);	12
%p	Ponteiro (endereço)	printf("%p", ptr);	0x7ffe...
%u	Inteiro sem sinal	printf("%u", 3000);	3000

```
14 }
```

```
1  precisao_largura (void* args) {
```



Largura minima: Define o espaço reservado na saída

Ex: `printf("%5d", 42)` → saída " 42" (5 espaços)



Casas decimais em `float`: Controla a precisão

Ex: `printf("%.3f", 3.14159)` → saída "3.142"



Alinhamento a esquerda (-): garante que o valor fique à esquerda

Ex: `printf("%-5d", 42)` → saída "42 "

```
13 }
```

1 **estrutura_basica_c (void* args) {**
2

```
#include <stdio.h> // Biblioteca padrão de entrada e saída

int main() {
    // Declaração de variáveis
    int idade;

    // Exibição de mensagem e entrada de dados
    printf("Digite sua idade: ");
    scanf("%d", &idade);

    // Saída de dados
    printf("Você tem %d anos.\n", idade);

    return 0; // Indica que o programa terminou corretamente
}
```

14 **}**

```
1 printf_e_scanf (void* args) {
```



```
5 São funções de entrada/saída de dados presentes na lib  
6 <stdio.h>;
```



```
8 printf: exibe informações na tela. Utiliza especificadores  
9 de formato para definir o tipo de dado a ser exibido;
```



```
11 scanf: lê dados do usuário e armazena em variáveis.  
12 Também usa especificadores de formato;
```

```
13 }  
14
```

exercicio_2.1 (void* args) {

- Calculadora de Soma

Descrição: Peça ao usuário dois números e exiba a soma entre eles.

- Exemplos de entrada:

Digite o primeiro número: 5

Digite o segundo número: 7

- Exemplo de saída

A soma de 5 e 7 é 12.

}

exercicio_2.2 (void* args) {

- Conversor de Temperatura

Descrição: Solicite ao usuário uma temperatura em graus Celsius e converta para Fahrenheit usando a fórmula: $F = (C \times 9/5) + 32$

- Exemplos de entrada:

Digite a temperatura em Celsius: 25

- Exemplo de saída:

25.00°C equivalem a 77.00°F.

}

exercicio_2.3 (void* args) {

- Cadastro Simples

Descrição: Solicite ao usuário seu nome, idade e altura e exiba as informações formatadas.

- Exemplo de entrada:

Digite seu nome: Ana

Digite sua idade: 22

Digite sua altura (em metros): 1.68

- Exemplo de saída:

Cadastro realizado:

Nome: Ana

Idade: 22 anos

Altura: 1.68 metros

}


```
1  getchar_e_putchar (void* args) {
```



```
4  São funções de entrada/saída de dados presentes na lib  
5  <stdio.h>;
```



```
8  getchar: lê um único caractere da entrada padrão stdin. Ela  
9  retorna o valor do caractere lido como um inteiro (int)  
10 permitindo o uso do valor especial EOF para indicar o fim  
11 da entrada;
```



```
14 Putchar: captura apenas um único caractere na saída  
padrão stdout;
```

```
    int putchar(int c)
```

```
}
```

03 {

[Tipos de dados, variáveis e operadores]

{ Tipos de dados e modificadores; }

{ Variáveis; }

{ Operadores; }

}

header.h

content.h

```
1  definicao_memoria (void* args) {
```

1 byte \Leftrightarrow 8 bits

1 byte \Leftrightarrow 8 bits

1 kilobyte \Leftrightarrow 1024 bytes

1 megabyte \Leftrightarrow 1024 kilobyte

1 gigabyte \Leftrightarrow 1024 megabyte

1 terabyte \Leftrightarrow 1024 gigabyte



1 byte

```
14 }
```

```
1 tipos_dados_c (void* args) {
```

Tipo	Tamanho (aprox.)	Faixa de Valores	Exemplo
int	4 bytes	-2,147,483,648 a 2,147,483,647	int idade = 25;
float	4 bytes	$\pm 3.4 \times 10^{-38}$ a $\pm 3.4 \times 10^{38}$ (6-7 dígitos)	float altura = 1.75;
double	8 bytes	$\pm 1.7 \times 10^{-308}$ a $\pm 1.7 \times 10^{308}$ (15-16 dígitos)	double pi = 3.1415926535;
char	1 byte	0 a 255 (ASCII)	char letra = 'A';

```
14 }
```

```
1  modificadores_tipo (void* args) {  
2  
3
```

Modificador	Aplicável a	Efeito
<code>signed</code>	<code>char</code> , <code>int</code>	Permite números negativos e positivos (padrão para <code>int</code>)
<code>unsigned</code>	<code>char</code> , <code>int</code>	Apenas números positivos (0 até o dobro do máximo positivo do tipo)
<code>short</code>	<code>int</code>	Reduz o tamanho (normalmente 2 bytes)
<code>long</code>	<code>int</code> , <code>double</code>	Aumenta o tamanho (normalmente 8 bytes para <code>int</code> , 8 bytes para <code>double</code>)

```
10  unsigned int positivo = 3000000000;  
11  short int pequeno = 32767;  
12  long int grande = 9223372036854775807;  
13  long double precisao = 3.141592653589793238;  
14
```

```
}
```

```
1 operadores_aritméticos (void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```

Operador	Descrição	Exemplo	Resultado
+	Adição	5 + 3	8
-	Subtração	5 - 3	2
*	Multiplicação	5 * 3	15
/	Divisão	10 / 2	5
%	Módulo	10 % 3	1

```
1 operadores_relacionais (void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```

Operador	Significado	Exemplo (x = 5, y = 3)	Retorno
==	Igual	x == y	0 (falso)
!=	Diferente	x != y	1 (verdadeiro)
>	Maior que	x > y	1 (verdadeiro)
<	Menor que	x < y	0 (falso)
>=	Maior ou igual	x >= y	1
<=	Menor ou igual	x <= y	0

```
1 operadores_logicos (void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```

Operador	Descrição	Exemplo (x = 1, y = 0)	Resultado
&&	E lógico	x && y	0 (falso)
,		,	OU lógico
!	NÃO lógico	!x	0

exercicio_3.1 (void* args) {

- **Cálculo da Idade**

Descrição: Crie um programa que leia o ano de nascimento de uma pessoa e calcule a idade dela, assumindo que o ano atual seja 2025. Use o tipo **int** para armazenar o ano de nascimento e a idade

- **Exemplo de entrada:**

Digite seu ano de nascimento: 1990

- **Exemplo de saída:**

Sua idade em 2025 é: 35

}

```
1  exercicio_3.2 (void* args) {
2
3      • Calculadora Simples
4      Descrição: Crie um programa que leia dois números inteiros e
5      realize as operações de soma, subtração, multiplicação e divisão.
6      Mostre os resultados de todas as operações
7      • Exemplo de entrada:
8      Digite o primeiro número: 8
9      Digite o segundo número: 4
10     • Exemplo de saída:
11     Soma: 12
12     Subtração: 4
13     Multiplicação: 32
14     Divisão: 2
15 }
```

04 {

[Estruturas de Controle]

{ Estruturas Condicionais; }

{ Estruturas de Repetição; }

{ Palavras importantes; }

}

estruturas_condicionais (void* args)

```
{
```



As estruturas condicionais em C são usadas para tomar decisões e alterar o fluxo de execução do programa com base em condições lógicas.

```
}
```

```
1 else_if (void* args) {
```



2 O comando **if** executa um bloco de código **se uma condição**
3 **for verdadeira (true).**

```
4
```

```
5 c
```

Copy

Edit

```
6
```

```
7 if (condição) {
```

```
8     // Código executado se a condição for verdadeira
```

```
9 } else if (outra condição) {
```

```
10     // Código executado se a segunda condição for verdadeira
```

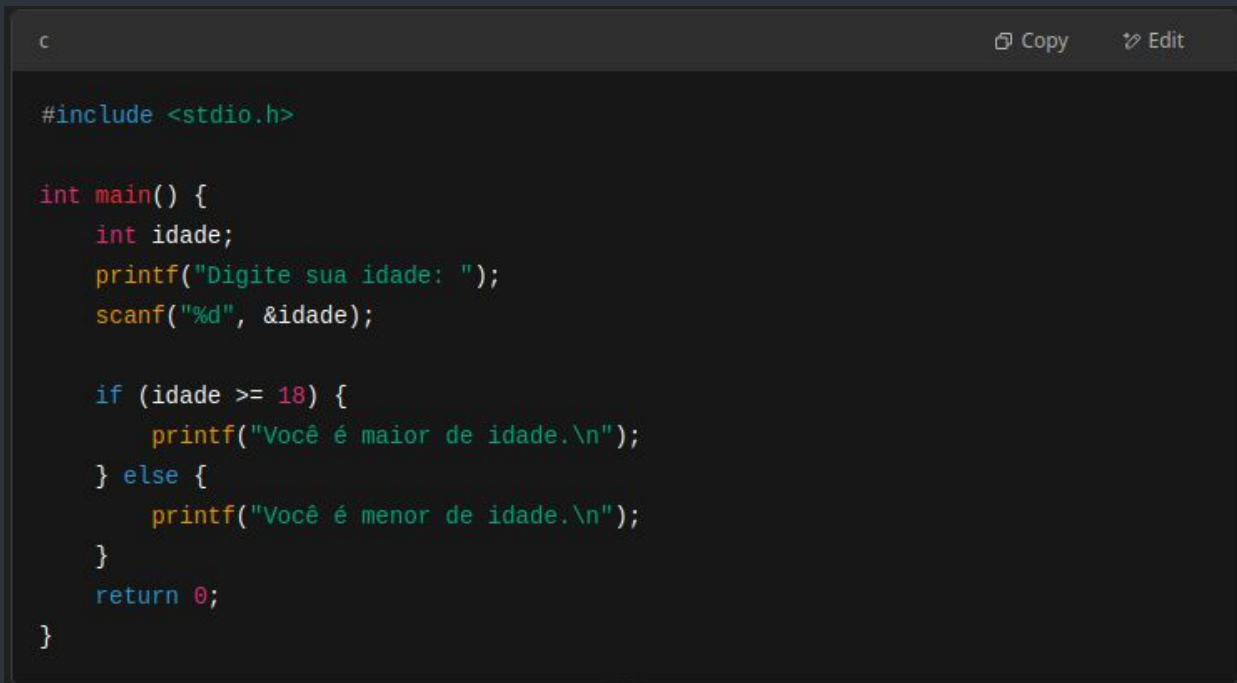
```
11 } else {
```

```
12     // Código executado se nenhuma condição for verdadeira
```

```
13 }
```

```
14 }
```

```
1 else_if_exemplo (void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```



```
c Copy Edit  
  
#include <stdio.h>  
  
int main() {  
    int idade;  
    printf("Digite sua idade: ");  
    scanf("%d", &idade);  
  
    if (idade >= 18) {  
        printf("Você é maior de idade.\n");  
    } else {  
        printf("Você é menor de idade.\n");  
    }  
    return 0;  
}
```

```
1 switch_case (void* args) {  
2
```



3 O **switch** é útil quando precisamos comparar um **valor fixo**
4 com múltiplas opções.

```
5  
6  
7 switch (variável) {  
8     case valor1:  
9         // Código para valor1  
10        break;  
11    case valor2:  
12        // Código para valor2  
13        break;  
14    default:  
15        // Código se nenhum valor corresponder  
16 }
```

```
17 }
```

```
1 switch_case_exemplo (void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

```
c Copy Edit  
  
#include <stdio.h>  
  
int main() {  
    int opcao;  
    printf("Escolha uma opção (1-3): ");  
    scanf("%d", &opcao);  
  
    switch (opcao) {  
        case 1:  
            printf("Opção 1 escolhida.\n");  
            break;  
        case 2:  
            printf("Opção 2 escolhida.\n");  
            break;  
        case 3:  
            printf("Opção 3 escolhida.\n");  
            break;  
        default:  
            printf("Opção inválida!\n");  
    }  
    return 0;  
}
```


exercicio_4.1 (void* args) {

- Média de Notas

Descrição: Crie um programa que leia 3 notas de um aluno, calcule a média e informe se o aluno foi aprovado ou reprovado. A média mínima para aprovação é 6. Utilize **float** para armazenar as notas e a média, e operadores relacionais para comparar a média.

- Exemplo de entrada:

Digite a primeira nota: 7.5

Digite a segunda nota: 6.0

Digite a terceira nota: 8.0

- Exemplo de saída:

A média do aluno é 7.50.

O aluno foi aprovado.

}

exercicio_4.2 (void* args) {

- **Determinando o Maior Número**

Descrição: Crie um programa que leia três números inteiros e determine qual é o maior. Use operadores relacionais para comparar os números.

- **Exemplo de entrada:**

Digite o primeiro número: 12

Digite o segundo número: 45

Digite o terceiro número: 23

- **Exemplo de saída:**

O maior número é 45.

}

```
1  estruturas_repeticao (void* args) {
```



As estruturas de repetição (loops) em C permitem executar um bloco de código várias vezes, enquanto uma condição for verdadeira. Elas são essências para automatizar tarefas repetitivas.

```
14 }
```

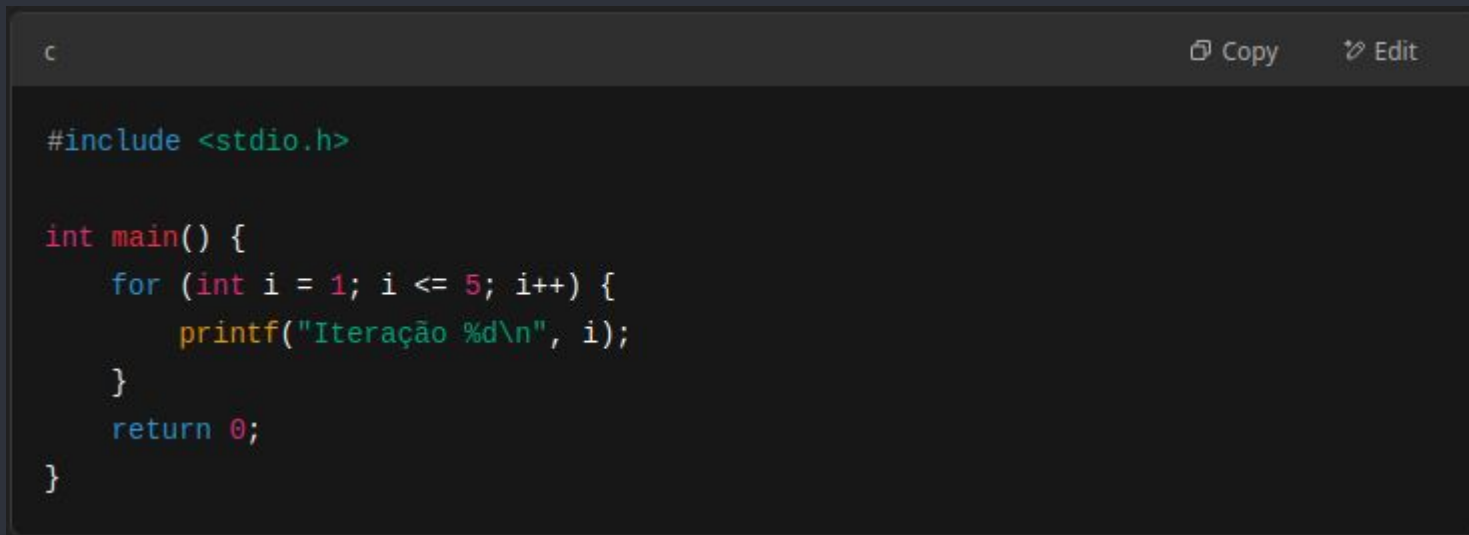
```
1 for (void* args) {
```



3 O **for** é usado quando **sabemos o número de repetições.**

```
8
9
10 for (inicialização; condição; incremento) {
11     // Código a ser repetido
12 }
13
14 }
```

```
1 for_exemplo (void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```



The screenshot shows a code editor window with a dark theme. The title bar at the top left says 'c'. On the top right, there are two icons: a copy icon and an edit icon, with the text 'Copy' and 'Edit' respectively. The code inside the editor is as follows:

```
#include <stdio.h>  
  
int main() {  
    for (int i = 1; i <= 5; i++) {  
        printf("Iteração %d\n", i);  
    }  
    return 0;  
}
```

```
1 while (void* args) {
```



3 O **while** executa um bloco **enquanto uma condição for verdadeira**

```
7 c
```

[Copy](#)[Edit](#)

```
9 while (condição) {  
10     // Código repetido  
11 }  
12  
13  
14 }
```

```
1 while_exemplo (void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

```
c Copy Edit  
  
#include <stdio.h>  
  
int main() {  
    int contador = 1;  
  
    while (contador <= 5) {  
        printf("Número: %d\n", contador);  
        contador++;  
    }  
    return 0;  
}
```

```
1 do_while (void* args) {
```



3 O **do-while** sempre executa o bloco pelo menos uma vez,
4 pois a condição é verificada **após a execução**.

```
5
```

Copy

Edit

```
6  
7  
8  
9 do {  
10     // Código repetido  
11 } while (condição);  
12  
13  
14 }
```



```
1 do_while_exemplo (void* args) {
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
}
```

```
c Copy Edit

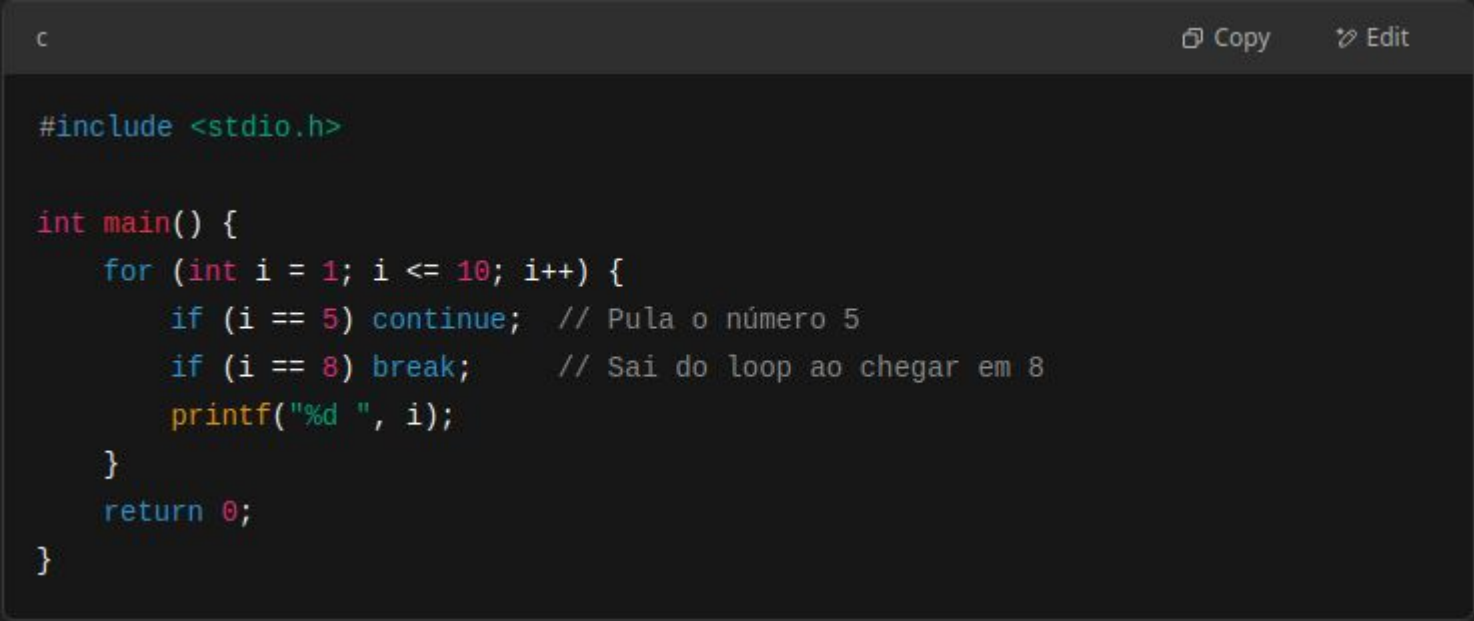
#include <stdio.h>

int main() {
    int num;
    do {
        printf("Digite um número positivo: ");
        scanf("%d", &num);
    } while (num <= 0);

    printf("Número digitado: %d\n", num);
    return 0;
}
```

```
1  palavras_importantes (void* args) {
2
3
4      [📁] break: Encerra o loop imediatamente
5
6
7
8
9
10     [📁] continue: Pula para a próxima iteração do loop
11
12
13
14 }
```

```
1 break_continue_example (void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```



The screenshot shows a code editor window with a dark theme. The title bar indicates the file is named 'c'. There are 'Copy' and 'Edit' buttons in the top right corner. The code inside the editor is as follows:

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) continue; // Pula o número 5
        if (i == 8) break;    // Sai do loop ao chegar em 8
        printf("%d ", i);
    }
    return 0;
}
```

exercicio_4.3 (void* args) {

- **Soma de numeros pares**

Descrição: Crie um programa que peça ao usuário um número inteiro **N** e calcule a soma de todos os números **pares** de 1 **até N** usando um laço **for**. Se o usuário digitar um valor menor que 1, peça um novo valor usando **do-while**.

- **Exemplo de entrada:**

Digite um numero inteiro positivo: -3

Digite um numero inteiro positivo: 10

- **Exemplo de saida:**

A soma dos numeros pares de 1 até 10 e: 30

}

exercicio_4.4 (void* args) {

• Adivinhe o numero

Descrição: Crie um jogo onde o usuário precisa adivinhar um número secreto entre 1 e 100. Use **while** para fornecer dicas.

- “Muito alto”, se o chute for maior.
- “Muito baixo”, se o chute for menor.

O número secreto pode ser **definido no código** ou gerado **aleatoriamente**. Adicione um **limite de tentativas** e, se o usuário não acertar, exiba o número secreto.

}

1
2
3
4
5
6
7
8
9
10
11
12
13
14

05 {

[Funções]

{ Declaração e Definição; }

{ Tipos; }

{ Passagens de Parametros; }

{ Recursão; }

}

```
1 funcoes_C (void* args) {
```



```
2  
3 Funções são blocos de código reutilizáveis que ajudam a  
4 organizar o programa, evitar repetições e facilitar a  
5 manutenção.  
6  
7  
8  
9  
10  
11  
12  
13  
14
```

```
}
```

```
1  declaracao_definicao (void* args) {
```



Uma função pode ser **declarada** antes do **main()** e **definida** depois.

```
6
7
8  tipo_retorno nome_funcao(parâmetros) {
9      // Corpo da função
10     return valor; // (Opcional se for void)
11 }
12
13
14 }
```



```
1 definicao_exemplo (void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```

c

Copy

Edit

#include <stdio.h>

// Declaração da função

int soma(int a, int b);

int main() {

int resultado = soma(5, 3);

printf("A soma é: %d\n", resultado);

return 0;

}

// Definição da função

int soma(int a, int b) {

return a + b;

}

```
1 tipos_funcoes (void* args) {
```



3 As funções podem ser classificadas pelo **tipo de retorno**
4 e **parâmetros**:

Tipo	Descrição
<code>void</code>	Não retorna valor.
Com retorno	Retorna um valor (<code>int</code> , <code>float</code> , etc.).
Sem parâmetros	Não recebe argumentos.
Com parâmetros	Recebe valores ao ser chamada.

```
14 }
```

```
1 tipos_funcoes_exemplo (void* args) {  
2  
3  
4  
5  
6 void mensagem() {  
7     printf("Olá, bem-vindo ao programa!\n");  
8 }  
9  
10 int main() {  
11     mensagem(); // Chamada da função  
12     return 0;  
13 }  
14 }
```

1 `passagem_parametros (void* args) {`



3 Passagem por valor (**padrão**). A função recebe **uma cópia**
4 do valor da variável original.

5 c

Copy

Edit

```
6 void altera(int x) {  
7     x = 10;  
8 }  
9
```

```
10 int main() {  
11     int a = 5;  
12     altera(a);  
13     printf("%d\n", a); // A variável original NÃO é alterada  
14 }
```

}

1 `passagem_parametros (void* args) {`



2 Passagem por referência (**usando ponteiros**). A função
3 pode **modificar** a variável original.

4

5 c

Copy

Edit

```
6 void altera(int *x) {  
7     *x = 10;  
8 }  
9
```

```
10 int main() {  
11     int a = 5;  
12     altera(&a);  
13     printf("%d\n", a); // A variável original é alterada  
14 }
```

}

```
1 recursao_funcoes (void* args) {
```



Uma função pode **chamar a si mesma** para resolver problemas repetitivos.

```
2
3
4
5
6
7 int fatorial(int n) {
8     if (n == 0) return 1;
9     return n * fatorial(n - 1);
10 }
11
12 int main() {
13     printf("Fatorial de 5: %d\n", fatorial(5));
14     return 0;
15 }
```

exercicio_5.1 (void* args) {

• **Calculo de fatorial**

Descrição: Crie uma função chamada fatorial, que recebe um numero inteiro positivo como parâmetro e retorne seu fatorial. No **main**, solicite ao usuário que insira um número, chame a função e exiba o resultado.

• **Exemplo de entrada e saída:**

Digite um numero inteiro: 5

O fatorial de 5 eh: 120

• **Observação:**

- O fatorial de **n** é o produto de $n * (n - 1) * (n - 2) * \dots * 1$
- Use um loop for ou while para calcular o fatorial.
- Trate o caso especial de $0! = 1$

}

exercicio_5.2 (void* args) {

• Verificação de Número Par ou Ímpar

Descrição: Crie uma função que receba um número inteiro como parâmetro e retorne **1** se for par e **0** se for ímpar. No **main** peça, ao usuário que insira um número, chame a função e verifique se é par ou ímpar.

• Exemplo de entrada/saída

Digite um numero: 7

0 numero 7 eh impar

}

1 06 {

2
3
4 [Vetores e Matrizes]

5
6
7 { Arrays Unidimensionais; }

8
9 { Arrays Multidimensionais; }

10 { Arrays e Funções; }

11
12 }
13
14

```
1 vetores_matrizes(void* args) {  
2
```



3 Em C, **vetores (arrays unidimensionais)** e **matrizes (arrays**
4 **multidimensionais)** são usadas para armazenar múltiplos
5 valores do mesmo tipo em única estrutura, facilitando a
6 manipulação de grandes quantidades de dados.

```
7  
8  
9  
10  
11  
12  
13  
14 }
```

```
1 vetores(void* args) {
```



Um **vetor** é uma sequência de elementos do mesmo tipo, acessados por um índice.

```
5 tipo nome[tamanho];
```

```
7 c
```

[Copy](#)[Edit](#)

```
9 #include <stdio.h>
```

```
11 int main() {
```

```
12     int numeros[5] = {10, 20, 30, 40, 50}; // Inicialização
```

```
13     printf("Terceiro elemento: %d\n", numeros[2]); // Índice começa em 0
```


```
14     return 0;
```

```
    }
```

```
}
```

```
1 percorrendo_vetor(void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

c

 Copy code

#include <stdio.h>

int main() {

int numeros[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {

printf("Elemento %d: %d\n", i, numeros[i]);

}

return 0;

}

exercicio_6.1 (void* args) {

- **Soma de Elementos de um Vetor**

Descrição: Crie um programa que solicite ao usuário que insira 5 números inteiros em um vetor. Em seguida, implemente uma função chamada **soma_vetor** que recebe o vetor e seu tamanho como parâmetros e retorna a soma dos elementos

- **Exemplo de entrada e saída:**

Digite 5 numeros: 1 2 3 4 5

A soma dos elementos eh: 30

}

```
1 matrizes(void* args) {
```



Matrizes são **tabelas de valores**, geralmente representados por linhas e colunas

```
5 tipo nome[linhas][colunas];
```

```
7 c
```

[Copy code](#)

```
8 #include <stdio.h>
```

```
10 int main() {
```

```
11     int matriz[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

```
12     printf("Elemento da linha 1, coluna 2: %d\n", matriz[1][2]);
```

```
13     return 0;
```

```
14 }
```

```
}
```

header.h

content.h

1 percorrendo_matriz(void* args) {

2
3 #include <stdio.h>

4
5 int main() {
6 int matriz[2][3] = { {1, 2, 3}, {4, 5, 6} };

7 for (int i = 0; i < 2; i++) {
8 for (int j = 0; j < 3; j++) {
9 printf("%d ", matriz[i][j]);
10 }
11 printf("\n");
12 }
13

14 }
15
16 return 0;
17 }
18

exercicio_6.2 (void* args) {

- **Multiplicação de Matriz por Escalar**

Descrição: Escreva um programa que leia uma **matriz 3×3** de números inteiros fornecidos pelo usuário. Em seguida, peça ao usuário um número escalar e multiplique todos os elementos da matriz por esse valor. Por fim, exiba a matriz resultante:

- A função de multiplicação deve ter o seguinte protótipo:

```
void multiplica_matriz(int matriz[3][3], int escalar);
```

}

1 arrays_funcoes(void* args) {



3 Podemos passar **vetores como argumento** para funções

```
4
5
6 void imprimir(int v[], int tamanho) {
7     for (int i = 0; i < tamanho; i++) {
8         printf("%d ", v[i]);
9     }
10    printf("\n");
11 }
12
13 int main() {
14     int numeros[] = {10, 20, 30, 40, 50};
15     imprimir(numeros, 5);
16     return 0;
17 }
```

Copy code





```
1 arrays_funcoes(void* args) {
```



2 Assim como vetores, **matrizes** também como argumento
3 para funções

```
4  
5  
6 void imprimirMatriz(int m[][3], int linhas) {  
7     for (int i = 0; i < linhas; i++) {  
8         for (int j = 0; j < 3; j++) {  
9             printf("%d ", m[i][j]);  
10        }  
11        printf("\n");  
12    }  
13 }  
14 }
```

```
int main() {  
    int matriz[2][3] = { {1, 2, 3}, {4, 5, 6} };  
    imprimirMatriz(matriz, 2);  
    return 0;  
}
```

```
1  resume_vetores_matrizes(void* args) {
2
3      [  Vetores armazenam vários valores do mesmo tipo em um única
4      dimensão.
5
6      [  Matrizes armazenam dados de mesmo tipo em múltiplas
7      dimensões.
8
9      [  Os índices começam em 0.
10
11     [  Arrays e matrizes podem ser passados para funções para
12     manipulação.
13
14 }
```

07 {

[Strings]

{ Declaração; }

{ Entrada e Saída de Strings; }

{ Principais funções; }

{ Diferenças entre vetores e ponteiros para

strings; }

}

```
1 strings(void* args) {
```



Em C, **strings** são representados como **arrays de caracteres** terminadas pelo caracter especial `'\0'` (nulo)



Diferente de linguagens com Python e Java, **não existe um tipo de dado específico para strings**, então elas são manipuladas como vetores de **char**.

```
14 }
```

```
1  declaracao_strings(void* args) {
```



Existem várias formas de declarar e inicializar strings em C

```
6  c
```

[Copy code](#)


```
7  char str1[] = "Olá"; // Inicialização implícita (adiciona '\0' automaticamente)
8  char str2[10] = "Mundo"; // Vetor com espaço extra
9  char str3[] = {'C', ' ', 'l', 'i', 'n', 'g', 'u', 'a', 'g', 'e', 'm', '\0'}; // Forma
10 char *str4 = "Teste"; // Ponteiro para string (armazenada na memória de leitura)
```



Toda string deve terminar com ‘\0’ para ser reconhecida corretamente.

```
14 }
```

```
1 entrada_strings(void* args) {
```

2  Para ler strings, não utilize `scanf("%s", str);` **sem um limite**, pois pode causar **estouro de buffer** (buffer overflow). Use `fgets` para maior segurança.

```
3  
4  
5  
6  
7 c
```

[Copy code](#)

```
8  
9 char nome[50];  
10 printf("Digite seu nome: ");  
11 fgets(nome, sizeof(nome), stdin); // Lê até 49 caracteres + '\0'
```

```
12  
13  
14 }
```

```
1  saida_strings(void* args) {
```



2 A função **printf** pode ser usada para imprimir strings

```
3  
4  
5  c
```

[Copy code](#)

```
6  
7  printf("%s\n", str1); // Saída: Olá
```

```
8  
9  
10  
11  
12  
13  
14 }
```



```
1  exercicio_7.1 (void* args) {
```

```
2
3
4      • Contar Caracteres em uma String
```

```
5      Descrição: Crie um programa que solicite ao usuário que insira uma
6      string e, em seguida, conte quantos caracteres existem nela (sem
7      considerar o caractere \0)
```

```
8
9
10     • Exemplo de entrada e saída:
```

```
11     Digite uma string: invertendo a logica
12     A string tem 19 caracteres.
```

```
13
14 }
```

principais_funcoes(void* args) {



A biblioteca `<string.h>` contém várias funções para manipulação de string:

Função	Descrição	Exemplo
<code>strlen(str)</code>	Retorna o tamanho da string (sem contar <code>\0</code>)	<code>strlen("C") → 1</code>
<code>strcpy(dest, src)</code>	Copia <code>src</code> para <code>dest</code> (cuidado com buffer overflow)	<code>strcpy(str1, str2);</code>
<code>strncpy(dest, src, n)</code>	Copia no máximo <code>n</code> caracteres de <code>src</code> para <code>dest</code>	<code>strncpy(dest, src, 5);</code>
<code>strcat(dest, src)</code>	Concatena <code>src</code> ao final de <code>dest</code>	<code>strcat(str1, "Mundo");</code>
<code>strcmp(str1, str2)</code>	Compara duas strings (0 se iguais)	<code>strcmp("a", "b") → -1</code>
<code>strchr(str, c)</code>	Retorna um ponteiro para a primeira ocorrência de <code>c</code>	<code>strchr("hello", 'e');</code>
<code>strstr(str1, str2)</code>	Retorna um ponteiro para a primeira ocorrência de <code>str2</code> em <code>str1</code>	<code>strstr("hello", "ll");</code>

1 exemplo_completo(void* args) {

```
c Copy code

#include <stdio.h>
#include <string.h>

int main() {
    char nome[50];



    printf("Digite seu nome: ");
    fgets(nome, sizeof(nome), stdin); // Lê string com segurança

    // Removendo a quebra de linha adicionada pelo fgets
    nome[strcspn(nome, "\n")] = '\0';

    printf("Olá, %s!\n", nome);

    return 0;
}
```

14 }

```
1 vet_ptr_x_strings(void* args) {
2
3     [  ] char nome[] = "Texto" → modificável, armazenado na
4     memória.
5
6
7     [  ] char *nome = "Texto" → Somente leitura (armazenada
8     na memória de leitura)
9
10
11
12
13
14 }
```

exercicio_7.2 (void* args) {

- **Verificar se uma String é um Palíndromo**

Descrição: Crie um programa que peça ao usuário para digitar uma palavra e verifique se ela é um **palíndromo** (ou seja, se pode ser lida da mesma forma de trás para frente).

- **Exemplo de entrada e saída:**

Digite uma palavra: radar

A palavra radar eh um palindromo

}

```
1 08 {
2
3   [Ponteiros]
4   { 0 que é; }
5
6   { Variáveis normais e ponteiros; }
7
8   { Declaração e inicialização; }
9
10  { Operadores; }
11
12  { Aritmética de Ponteiros; }
13 }
14
```

```
1 o_que_e_ponteiro(void* args) {
```



Ponteiro é uma variável que armazena o endereço de memória de outra variável.

```
5     tipo *nome_do_ponteiro;
```



```
7     c
```

Copy

Edit

```
8  
9     int a = 10;  
10    int *p = &a; // Ponteiro armazenando o endereço de 'a'  
  
11    printf("Valor de a: %d\n", a);  
12    printf("Endereço de a: %p\n", &a);  
13    printf("Valor armazenado no ponteiro p: %p\n", p);  
14    printf("Valor apontado por p: %d\n", *p);
```

```
}
```

```
1 operadores_ponteiros(void* args) {
2
3     [  Operador &: → o operador & (E comercial) é usado
4     para obter o endereço de memória de uma variável.
5
6
7     [  Operador * (desreferência) → O operador *
8     (asterisco) é usado para acessar ou modificar o
9     valor armazenado no endereço apontado pelo ponteiro.
10
11
12
13
14 }
```



```
1 obter_endereco_memoria(void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

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

```
int main() {
```

```
    int x = 10;
```

```
    int *p = &x; // 'p' armazena o endereço de 'x'
```

```
    printf("Endereço de x: %p\n", &x);
```

```
    printf("Endereço armazenado em p: %p\n", p);
```

```
    return 0;
```

```
}
```

```
Endereço de x: 0x7ffee7b548c4
```

```
Endereço armazenado em p: 0x7ffee7b548c4
```

```
1 obter_valor_ponteiro(void* args) {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

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

```
int main() {
```

```
    int x = 20;
```

```
    int *p = &x; // 'p' armazena o endereço de 'x'
```

```
    printf("Valor de x: %d\n", *p); // Acessa o valor de x através do ponteiro
```

```
    *p = 50; // Modifica o valor de x usando o ponteiro
```

```
    printf("Novo valor de x: %d\n", x);
```

```
    return 0;
```

```
}
```

```
Valor de x: 20
```

```
Novo valor de x: 50
```

header.h

content.h

```
1 o_que_e_ponteiro(void* args) {
```

```
2     char a = 10
3     char *ch = &a
```


a →

Endereço	Valor
----------	-------

0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

```
11
12     &a retorna o endereço de a, enquanto *ch acessa o valor
13     armazenado no endereço
```

```
14 }
```

```
1 ponteiros_e_arrays(void* args) {
2
3     [  ] Os arrays são manipulados com ponteiros porque o
4     nome do array é um ponteiro para o primeiro
5     elemento.
6
7
8     int arr[] = {1, 2, 3, 4};
9     int *p = arr; // 'p' aponta para o primeiro elemento
10
11     printf("%d\n", *p);    // Exibe 1
12     printf("%d\n", *(p+1)); // Exibe 2
13
14 }
```

```
1  aritmetica_ponteiros(void* args) {
2
3  Podemos realizar operações matemáticas com ponteiros:
4
5  • Incremento (++) → Avança para o próximo elemento do tipo do
6    ponteiro.
7
8  • Decremento (--) → Retrocede para o elemento anterior.
9
10 • Soma (+) → Avança n posições na memória.
11
12 • Subtração (-) → Retrocede n posições na memória.
13
14 • Subtração entre ponteiros → Determina a distância entre dois
    elementos em um array.
15 }
```

header.h

content.h

```
1 incremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch++;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

ch  a 

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 incremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch++;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

a →

ch →

Endereço

Valor

0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 incremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch++;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

a →

ch →

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 incremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch++;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

a →

ch →

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 decremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch--;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

a →

ch →

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 decremento(void* args) {
2
3     char a = 10;
4     char *ch = &a;
5
6     ch--;
7
8
9
10
11
12
13
14 }
```

a →

ch →

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 decremento(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch--;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

ch  a 

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 soma_ponteiros(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch = ch + 4;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

ch \Rightarrow a \Rightarrow

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 soma_ponteiros(void* args) {  
2  
3     char a = 10;  
4     char *ch = &a;  
5  
6     ch = ch + 4;  
7  
8  
9  
10  
11  
12  
13  
14 }
```

ch  a 

ch 

Endereço

Valor

0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

header.h

content.h

```
1 subtracao_ponteiros(void* args) {
2
3     char a = 10;
4     char *ch = &a;
5
6     ch = ch - 4;
7
8
9
10
11
12
13
14 }
```

a →

Endereço	Valor
0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

ch →

header.h

content.h

```
1 subtracao_ponteiros(void* args) {
2
3     char a = 10;
4     char *ch = &a;
5
6     ch = ch - 4;
7
8
9
10
11
12
13
14 }
```

ch → a →


Endereço

Valor

0x0000	10
0x0001	20
0x0002	30
0x0003	40
0x0004	50

ch →

diferenca_entre_ponteiros(void* args)

{  Podemos subtrair dois ponteiros para descobrir quantos elementos existem entre eles.

```
c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *p1 = &arr[1]; // Aponta para o segundo elemento
    int *p2 = &arr[4]; // Aponta para o quinto elemento

    int distancia = p2 - p1; // Número de elementos entre p1 e p2
    printf("Distância entre os ponteiros: %d\n", distancia);

    return 0;
}
```

Distância entre os ponteiros: 3

- Como cada **int** ocupa 4 bytes, a subtração considera a **quantidade de elementos**, não a diferença entre bytes.

```
1  comparar_ponteiros(void* args) {
```



```
2  Podemos comparar dois ponteiros usando operadores  
3  relacionais
```

Operador	Significado
==	Ponteiros apontam para o mesmo endereço
!=	Ponteiros apontam para endereços diferentes
>	Um ponteiro aponta para um endereço maior na memória
<	Um ponteiro aponta para um endereço menor

```
14 }
```

header.h

content.h

exemplo_comparacao(void* args) {

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

```
int main() {
```

```
    int a = 10, b = 20;
```

```
    int *p1 = &a, *p2 = &b;
```

```
    if (p1 != p2) {
```

```
        printf("Os ponteiros apontam para endereços diferentes.\n");
```

```
    }
```

```
    return 0;
```

```
}
```

lua

Copy

Edit

Os ponteiros apontam para endereços diferentes.

}

ponteiros_funcoes(void* args) {




Passar ponteiros para funções permite modificar diretamente os valores das variáveis

```
#include <stdio.h>

void modificar(int *p) {
    *p = 100; // Modifica o valor apontado
}

int main() {
    int x = 10;
    modificar(&x);
    printf("%d\n", x); // Exibe 100
    return 0;
}
```

}

```
1  ponteiro_para_ponteiro(void* args) {
2      [  ] Podemos ter ponteiros que armazenam o endereço de outros
3      ponteiros.
4
5
6
7
8
9
10
11
12
13
14 }
```

```
int a = 10;
int *p = &a; // Ponteiro para int
int **pp = &p; // Ponteiro para ponteiro

printf("%d\n", **pp); // Exibe 10
```

ponteiros_alocacao(void* args) {



Com **malloc**, **calloc** e **free**, é possível alocar e liberar memória **dinamicamente**.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int *)malloc(sizeof(int)); // Aloca um inteiro

    if (p == NULL) {
        printf("Erro de alocação!\n");
        return 1;
    }

    *p = 42; // Atribui um valor à memória alocada
    printf("Valor: %d\n", *p);

    free(p); // Libera a memória alocada
    return 0;
}
```

- Sempre use **free()** para evitar vazamentos de memória.

```
1 erros_ponteiros(void* args) {
```

Ponteiro não inicializado

c

 Copy Edit

```
int *p;  
*p = 10; // ERRO! O ponteiro não foi inicializado
```

Acesso fora dos limites do array

c

 Copy Edit

```
int arr[5];  
int *p = arr + 10; // ERRO! Acesso inválido
```






Não liberar memória alocada

c

 Copy Edit

```
int *p = (int *)malloc(10 * sizeof(int));  
// Esquecer de chamar free(p) → VAZAMENTO DE MEMÓRIA!
```

```
14 }
```

```
1  resume_ponteiros(void* args) {
2
3      [  Ponteiros armazenam endereços de memória
4
5      [  Operador * desreferencia um ponteiro
6
7      [  Aritmetica de ponteiros permite percorrer arrays
8
9
10     [  Ponteiros para funções, permite modificar variaveis
11
12     [  Alocação dinâmica de memória, evita desperdícios, mas exige
13       o uso de free()
14 }
}
```


09 {

[Alocação Dinamica de Memoria]

{ 0 que é; }

{ malloc(); }

{ calloc(); }

{ realloc(); }

}

alocacao_dinamica_memoria(void* args)

{



Alocação dinâmica de memória permite que um programa reserve memória em tempo de execução, tornando-o mais flexível e eficiente no uso da RAM.



A memória pode ser alocada dinamicamente através da biblioteca `stdlib.h` com o uso das funções:

}

```
1 malloc(void* args) {
```



Aloca um bloco de memoria nao inicializada:

```
c Copy Edit

#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*) malloc(3 * sizeof(int)); // Aloca espaço para 3 inteiros

    if (p == NULL) {
        printf("Erro ao alocar memória.\n");
        return 1;
    }

    p[0] = 10; p[1] = 20; p[2] = 30;

    printf("Valores: %d, %d, %d\n", p[0], p[1], p[2]);

    free(p); // Libera a memória alocada
    return 0;
}
```

```
14 }
```

```
1 calloc(void* args) {
```



Aloca e inicializa com 0 um bloco de memória:

```
c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*) calloc(3, sizeof(int)); // Aloca e inicializa com zero

    if (p == NULL) {
        printf("Erro ao alocar memória.\n");
        return 1;
    }

    printf("Valores iniciais: %d, %d, %d\n", p[0], p[1], p[2]);

    free(p);
    return 0;
}
```

```
14 }
```

realloc(void* args) {



Redimensiona um bloco já alocado:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*) malloc(3 * sizeof(int)); // Aloca espaço para 3 inteiros

    if (p == NULL) {
        printf("Erro ao alocar memória.\n");
        return 1;
    }

    p[0] = 10; p[1] = 20; p[2] = 30;

    // Redimensiona para armazenar 5 inteiros
    p = (int*) realloc(p, 5 * sizeof(int));

    if (p == NULL) {
        printf("Erro ao realocar memória.\n");
        return 1;
    }

    p[3] = 40; p[4] = 50;

    printf("Valores: %d, %d, %d, %d, %d\n", p[0], p[1], p[2], p[3], p[4]);

    free(p); // Libera a memória alocada
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int*) malloc(5 * sizeof(int)); // Aloca espaço para 5 inteiros

    if (p == NULL) {
        printf("Erro ao alocar memória.\n");
        return 1;
    }

    p[0] = 10; p[1] = 20; p[2] = 30; p[3] = 40; p[4] = 50;

    // Reduz o tamanho para armazenar apenas 3 inteiros
    p = (int*) realloc(p, 3 * sizeof(int));

    if (p == NULL) {
        printf("Erro ao realocar memória.\n");
        return 1;
    }

    printf("Valores restantes: %d, %d, %d\n", p[0], p[1], p[2]);

    free(p); // Libera a memória alocada
    return 0;
}
```

10 {

[Structs]

{ Declaração; }

{ Acessando membros; }

{ typedef; }

{ ponteiros para struct; }

{ array de structs; }

} { structs e funções; }

```
1 o_que_e_struct(void* args) {
```



```
2     As structs permitem agrupar diferentes tipos de dados em uma  
3     única unidade, facilitando a organização de informações  
4     complexas. Elas são essenciais para representar entidades do  
5     mundo real.  
6  
7  
8  
9  
10  
11  
12  
13  
14 }
```

1 `declaracao_struct(void* args) {`

2 `c`

Copy

Edit

3 `#include <stdio.h>`

4 `// Definição da struct`

5 `struct Aluno {`

6 `char nome[50];`

7 `int idade;`

8 `float nota;`

9 `};`

10 `int main() {`

11 `struct Aluno aluno1 = {"Carlos", 20, 8.5}; // Inicialização direta`

12 `printf("Nome: %s\nIdade: %d\nNota: %.2f\n", aluno1.nome, aluno1.idade, aluno1.nota);`

13 `return 0;`

14 `}`

}


```
1 acessar_membros_struct(void* args) {
2
3
4
5
6
7
8
9
10
11
12
13
14 }
```

c

Copy

Edit

```
aluno1.idade = 21; // Modificando um membro
printf("Nova idade: %d\n", aluno1.idade);
```

1 **typedef(void* args) {**



2 O **typedef** pode ser usado para criar um alias, evitando a
3 necessidade de escrever **struct** repetidamente.

```
4
5
6 typedef struct {
7     char nome[50];
8     int idade;
9     float nota;
10 } Aluno;
11
12 int main() {
13     Aluno aluno1 = {"Ana", 22, 9.2};
14
15     printf("Nome: %s\nIdade: %d\nNota: %.2f\n", aluno1.nome, aluno1.idade, aluno1.nota)
16
17     return 0;
18 }
```

```
1 ponteiros_struct(void* args) {
```



Podemos usar ponteiros para acessar membros da **struct** usando o operador **→**.

```
2
3
4
5 #include <stdio.h>
6
7 typedef struct {
8     char nome[50];
9     int idade;
10 } Pessoa;
11
12 int main() {
13     Pessoa p = {"João", 30};
14     Pessoa *ptr = &p;
15
16     printf("Nome: %s\n", ptr->nome); // Equivalente a (*ptr).nome
17     printf("Idade: %d\n", ptr->idade);
18
19     return 0;
20 }
```

```
1 arrays_struct(void* args) {
```



Podemos armazenar múltiplas **structs** em um array.

```
2
3
4
5 #include <stdio.h>
6
7 typedef struct {
8     char nome[50];
9     int idade;
10 } Pessoa;
11
12 int main() {
13     Pessoa pessoas[2] = {{ "Maria", 25 }, { "Carlos", 28 }};
14
15     for (int i = 0; i < 2; i++) {
16         printf("Nome: %s, Idade: %d\n", pessoas[i].nome, pessoas[i].idade);
17     }
18
19     return 0;
20 }
```

```
14 }
```

```
1 struct_sfuncoes(void* args) {
```



2 Podemos passar **structs** para funções por **valor** ou

3 **referência**.

✓ Passagem por Valor

c

Copy

Edit

```
void exibirAluno(struct Aluno a) {  
    printf("Nome: %s, Nota: %.2f\n", a.nome, a.nota);  
}
```

✓ Passagem por Referência (Melhor para eficiência)

c

Copy

Edit

```
void modificarNota(struct Aluno *a) {  
    a->nota = 10.0; // Modifica diretamente a estrutura original  
}
```

```
14 }
```

```
1 11 {
2
3   [Arquivos]
4
5
6   { Introdução; }
7   { Operações; }
8
9   { Modos de Abertura; }
10  { Funções Importantes; }
11
12 }
13
14
```

```
1 introducao(void* args) {
```



Em C, a manipulação de arquivos permite armazenar e recuperar dados de maneira persistente. Isso é feito através da biblioteca **stdio.h** que fornece funções para **abrir**, **ler** e **fechar arquivos**

```
14 }
```

```
1 operacoes_arquivos(void* args) {
```



```
4 Abrir arquivo:
```

```
5 c
```

```
Copy
```

```
Edit
```

```
6  
7 FILE *arquivo;  
8 arquivo = fopen("dados.txt", "r");  
9 if (arquivo == NULL) {  
10     printf("Erro ao abrir o arquivo!\n");  
11     return 1;  
12 }  
13
```

```
14 }
```



```
1 modos_abertura(void* args) {
```



A função **fopen()** é usada para abrir um arquivo e recebe dois parâmetros: o nome do arquivo e o modo de abertura.

Modo	Descrição
"r"	Abre para leitura (o arquivo deve existir).
"w"	Abre para escrita (cria um novo arquivo ou sobrescreve um existente).
"a"	Abre para escrita no final do arquivo (mantém o conteúdo existente).
"r+"	Abre para leitura e escrita (o arquivo deve existir).
"w+"	Abre para leitura e escrita (cria um novo ou sobrescreve um existente).
"a+"	Abre para leitura e escrita (mantém o conteúdo e escreve no final).

```
14 }
```

```
operacoes_arquivos(void* args) {
```



Ler de um arquivo:

```
c
```

[Copy](#)[Edit](#)

```
FILE *arquivo = fopen("dados.txt", "r");
char linha[100];
while (fgets(linha, sizeof(linha), arquivo) != NULL) {
    printf("%s", linha);
}
fclose(arquivo);
```

```
}
```

```
1 operacoes_arquivos(void* args) {
```



```
2  
3     Escrever em um arquivo:
```

```
4  
5  
6  
7     FILE *arquivo = fopen("dados.txt", "w");  
8     fprintf(arquivo, "Olá, mundo!\n");  
9     fclose(arquivo);
```

```
10  
11  
12  
13  
14 }
```

```
1 funcoes_importantes(void* args) {
```

Função	Descrição
<code>fopen()</code>	Abre um arquivo.
<code>fclose()</code>	Fecha um arquivo.
<code>fprintf()</code>	Escreve em um arquivo (como <code>printf</code>).
<code>fscanf()</code>	Lê de um arquivo (como <code>scanf</code>).
<code>fgets()</code>	Lê uma linha de um arquivo.
<code>fputc()</code>	Escreve um caractere no arquivo.
<code>fgetc()</code>	Lê um caractere do arquivo.

```
14 }
```

header.h

content.h

1 exemplo_completo(void* args) {

2
3 #include <stdio.h>

4
5 int main() {

6 FILE *arquivo = fopen("exemplo.txt", "w");

7 if (arquivo == NULL) {

8 printf("Erro ao abrir o arquivo!\n");

9 return 1;

10 }

11 fprintf(arquivo, "Curso de C - Manipulação de Arquivos\n");

12 fclose(arquivo);

13 printf("Arquivo criado com sucesso!\n");

14 return 0;

}

1
2
3
4
5
6
7
8
9
10
11
12
13
14

```
return 0;
```