



**UNIP**  
**UNIVERSIDADE PAULISTA**

# Roteiros

**Programação Estruturada em C**



**Instituto de Ciências  
Exatas e Tecnologia**

**Disciplina:** Programação Estruturada em C  
**Título da Aula:** Programação Estruturada em C

**AULA 01**

## 1. Objetivos da Aula

Compreender a estrutura básica de um programa em linguagem C, reconhecendo seus principais componentes.

Apresentar os tipos de dados primitivos, identificadores e variáveis, bem como seu uso correto conforme as boas práticas da linguagem.

Demonstrar o uso dos comandos básicos de entrada (scanf) e saída (printf), por meio de exemplos simples e funcionais.

Estimular a construção de pequenos programas em C com organização e clareza, utilizando comentários explicativos.

Orientar a elaboração de um relatório conciso contendo a fundamentação teórica, códigos desenvolvidos e reflexões sobre a aprendizagem do conteúdo.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 1 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

### 3. Estrutura da Aula

#### 3.1. Abertura (10 minutos)

- Apresentação dos objetivos da disciplina e da linguagem C como linguagem estruturada e fundamental.
- Explicação breve do que é um algoritmo e como ele é transformado em código executável.

#### 3.2. Revisão Conceitual (10 minutos)

- Apresentação dos principais conceitos do capítulo 1 do livro-texto, com explicações e demonstrações no Code::Blocks:
- Introdução à lógica básica de programação com foco em variáveis, entrada e saída de dados e estrutura básica de um programa em C.

#### 3.3. Demonstração Prática (60 minutos)

##### Exemplo 1: Programa Hello World!

```
#include <stdio.h>

int main() {
    printf("Olá, mundo!\n");
    return 0;
}
```

Figura 1 - A

Explicação: estrutura mínima de um programa em C, função **main()**, diretiva **#include** e uso de **printf()**.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    printf("Olá, mundo!\n");

    return 0;
}
```

Figura 1 - B

Explicação: estrutura mínima de um programa em C, função **main()**, diretiva **#include** e uso de **printf()** e uso do cabeçalho **locale.h**.

### Exemplo 2: Uso de variáveis.

```
#include <stdio.h>

int main() {
    float valor1, valor2, valor3, total;

    valor1 = 10.5;
    valor2 = 20.5;
    valor3 = 5.75;

    total = valor1 + valor2 + valor3;
    printf("Total: %.2f", total);

    return 0;
}
```

Explicação: programa simples em linguagem C que realiza a soma de três valores do tipo **float** e exibe o resultado formatado com duas casas decimais.

### Exemplo 3: Leitura de dados com scanf().

```
#include <stdio.h>

int main() {
    int idade;

    printf("Digite sua idade: ");
    scanf("%d", &idade);

    printf("Você tem %d anos.\n", idade);
    return 0;
}
```

Explicação: uso de **scanf()** com operador **&**, tipo de variável **int**, e formatação com **%d**.

### Exemplo 4: Uso de variáveis e leitura de dados com scanf().

```
#include <stdio.h>

int main() {
    int idade;
    float altura;
    char inicial;

    printf("Digite sua idade: ");
    scanf("%d", &idade); // Lê um número inteiro

    printf("Digite sua altura: ");
    scanf("%f", &altura); // Lê um número de ponto flutuante

    printf("Digite a inicial do seu nome: ");
    scanf(" %c", &inicial); // Lê um caractere (observe o espaço antes do %c)

    printf("\nIdade: %d, Altura: %.2f, Inicial: %c\n", idade, altura, inicial);
    return 0;
}
```

Explicação: programa em linguagem C que faz a leitura de entrada de dados do usuário utilizando a função **scanf()**, permitindo que o usuário forneça três tipos diferentes de dados. A saída no console exibe as informações fornecidas pelo usuário de forma organizada.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

- O aluno deverá digitar o programa abaixo seguindo as observações e se possível fazer as ampliações sugeridas:

- Objetivo do programa: um mini formulário com saudação personalizada
- O programa permite:

1. Pedir o nome do usuário.

2. Pedir sua idade.

3. Exibir uma mensagem formatada como: "Olá, *Aluno*! Você tem *X* anos.", onde *Aluno* e *X* devem ser substituídos, respectivamente, pelo nome e pela idade digitados pelo aluno.

Observações:

- Usar boas práticas de indentação.
- Comentar o código explicando o que faz cada bloco.
- Cuidar da ortografia das mensagens.

Exemplo mínimo esperado:

```
#include <stdio.h>

int main() {
    char nome[50];
    int idade;

    printf("Digite seu nome: ");
    scanf("%s", nome);

    printf("Digite sua idade: ");
    scanf("%d", &idade);

    printf("Olá, %s! Você tem %d anos.\n", nome, idade);

    return 0;
}
```

Ampliações sugeridas:

- Leitura do nome completo com `scanf()` formatado ou `fgets()`.
- Adicionar e configurar o arquivo de cabeçalho `locale.h` para permitir a exibição correta de acentuação em português.
- Adicionar outros campos ao cadastro, como curso, matrícula, telefone ou e-mail, para expandir o formulário do aluno.

## 5. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecer eventuais problemas encontrados na implementação.
- Reforço dos conceitos de variáveis, entrada e saída. Consolidar o uso de variáveis e estruturas básicas. Introduzir novos comandos de forma gradual.
- Estimular a criatividade e personalização dos programas.
- **Instruções sobre o relatório 1:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.



**Instituto de Ciências  
Exatas e Tecnologia**

**Disciplina:** Programação Estruturada em C

**Título da Aula:** Operadores Básicos em C

**AULA 02**

## 1. Objetivos da Aula

Compreender e aplicar os operadores básicos da linguagem C de: atribuição, aritméticos, relacionais, lógicos, incremento e decremento, no desenvolvimento de expressões e instruções fundamentais, fortalecendo o raciocínio lógico e a construção de programas simples e funcionais.

Estimular a construção de pequenos programas em C com uso dos operadores básicos em C.

Orientar a elaboração de um relatório conciso contendo a fundamentação teórica, códigos desenvolvidos e reflexões sobre a aprendizagem do conteúdo.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 1 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.



### 3. Estrutura da Aula

#### 3.1.Abertura (10 minutos)

- Apresentação dos objetivos da aula, com foco no uso dos operadores em linguagem C.
- Breve contextualização da importância dos operadores na construção de expressões, cálculos, comparações e controle de fluxo.
- Conexão com situações práticas: como o computador entende e executa operações lógicas e matemáticas.

#### 3.2.Revisão Conceitual (10 minutos)

- Explicação teórica dos principais tipos de operadores em C:
  - Aritméticos (+, -, \*, /, % (operador de resto da divisão inteira))
  - Atribuição (=, +=, -=, etc.)
  - Relacionais (==, !=, <, >, <=, >=)
  - Lógicos (&& (E), || (OU), ! (NÃO))
  - Incremento e Decremento (++ , --)
- Exemplificação de expressões usando cada tipo de operador.
- Discussão sobre a ordem de precedência e associatividade dos operadores.
- Demonstrações simples no Code::Blocks com trechos de código curtos que destacam o uso prático dos operadores em expressões e decisões.

### 3.3.Demonstração Prática (60 minutos)

#### Exemplo 1: Operadores Aritméticos

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int a = 10, b = 3;

    printf("Soma: %d\n", a + b);
    printf("Subtração: %d\n", a - b);
    printf("Multiplicação: %d\n", a * b);
    printf("Divisão: %d\n", a / b);
    printf("Resto da divisão: %d\n", a % b);

    return 0;
}
```

Figura 1

Explicação: esse programa demonstra os operadores aritméticos básicos (+, -, \*, /, %) aplicados a dois inteiros.

#### Exemplo 2: Operadores de Atribuição.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int x = 5;

    x += 3; //ou x = x + 3
    printf("Resultado de x += 3: %d\n", x);

    x *= 2; //ou x = x * 2
    printf("Resultado de x *= 2: %d\n", x);

    return 0;
}
```

Explicação: Neste programa vemos operadores de atribuição combinados com operações (+, \*=). Eles facilitam a escrita de expressões matemáticas comuns.

### Exemplo 3: Operadores Relacionais.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int idade = 18;

    printf("Idade igual a 18? %d\n", idade == 18);
    printf("Idade maior que 17? %d\n", idade > 17);
    printf("Idade diferente de 20? %d\n", idade != 20);

    return 0;
}
```

Explicação: Os operadores relacionais (==, !=, >) comparam valores e retornam 1 (verdadeiro) ou 0 (falso). Muito usados em estruturas condicionais.

### Exemplo 4: Operadores Lógicos.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int a = 5, b = 10;

    // Avalia expressões lógicas e imprime o resultado
    printf("a > 0 E b > 0: %d\n", (a > 0) && (b > 0));
    printf("a > 0 OU b < 0: %d\n", (a > 0) || (b < 0));
    printf("NÃO (a > b): %d\n", !(a > b));

    return 0;
}
```

Explicação: Esse programa avalia três expressões lógicas:

- **&&** (E lógico): só retorna 1 se ambas as condições forem verdadeiras.
- **||** (OU lógico): retorna 1 se pelo menos uma for verdadeira.
- **!** (NÃO lógico): inverte o valor lógico da condição.

A saída será:

- $a > 0 \text{ E } b > 0$ : 1
- $a > 0 \text{ OU } b < 0$ : 1
- **NÃO** ( $a > b$ ): 1

Observação: Ideal para mostrar que operadores lógicos retornam 0 ou 1 e podem ser impressos diretamente como resultado de expressões. O `//` indica linha de comentário.

### Exemplo 5: Operadores de Incremento e Decremento

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int i = 5;
    printf("Valor de i: %d\n", i);
    printf("Pós-incremento: %d\n", i++);
    printf("Valor após o incremento: %d\n", i);
    printf("Pré-decremento: %d\n", --i);

    return 0;
}
```

Explicação: Mostra a diferença entre pós-incremento (`i++`, primeiro usa depois soma) e pré-decremento (`--i`, primeiro subtrai depois usa). Essencial em laços e contadores.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

- O aluno deverá digitar o programa abaixo seguindo as observações e, se possível, implementar alguma das ampliações sugeridas, seguindo os passos abaixo:
- Objetivo do Programa: Solicitar nome, idade e duas notas do aluno, calcular a média e exibir expressões lógicas avaliadas diretamente no console.

O programa faz as seguintes operações:

1. Ler o nome, idade, e duas notas do usuário.
2. Calcular a média das notas.
3. Exibir os seguintes resultados lógicos:
  - Se a média é maior ou igual a 7.
  - Se a idade é menor que 18.

Observações:

- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

Exemplo mínimo esperado:

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    char nome[50];
    int idade;
    float n1, n2, media;

    printf("Digite seu nome: ");
    scanf(" %[^\\n]", nome);

    printf("Digite sua idade: ");
    scanf("%d", &idade);

    printf("Digite duas notas: ");
    scanf("%f %f", &n1, &n2);

    media = (n1 + n2) / 2;

    printf("\\nOlá, %s!\\n", nome);
    printf("Média: %.2f\\n", media);
    printf("Média >= 7? %d\\n", media >= 7);
    printf("Idade < 18? %d\\n", idade < 18);

    return 0;
}
```

Ampliação sugerida:

- Comentar o código explicando cada linha principal.
- Exibir diretamente o resultado de expressões como: idade >= 18, media >= 7, (nota1 == 10 || nota2 == 10).
- Adicionar o uso de operadores aritméticos em novos cálculos (ex: soma das notas ou média com pesos).
- Utilizar operadores de atribuição composta (+=, -=, \*=) para acumular valores ou transformar variáveis.

## 5. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecer eventuais problemas encontrados na implementação dos códigos.
- Reforço dos conceitos estudados sobre o uso de operadores básicos em C.
- Estimular a criatividade e personalização dos programas.
- **Instruções sobre o relatório 2:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.



**Instituto de Ciências  
Exatas e Tecnologia**

**Disciplina:** Programação Estruturada em C  
**Título da Aula:** Operações e Controle de Fluxo

**AULA 03**

## 1. Objetivos da Aula

Compreender e aplicar as estruturas condicionais da linguagem C (if, if...else, if...else if...else), desenvolvendo o raciocínio lógico e a tomada de decisão.

Orientar a elaboração de um relatório conciso contendo a fundamentação teórica, códigos desenvolvidos e reflexões sobre a aprendizagem do conteúdo.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 2 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

## 3. Estrutura da Aula

### 3.1. Abertura (10 minutos)

- Apresentação dos objetivos da aula e da importância do controle de fluxo na programação.
- Contextualização prática: "Como os programas tomam decisões?".
- Breve revisão de operadores relacionais e lógicos que serão usados nas condições. Conexão com situações práticas.



### 3.2.Revisão Conceitual (10 minutos)

Explicação teórica das estruturas condicionais em C:

- if
  - if...else
  - if...else if...else
- 
- Demonstração da sintaxe correta e dos erros comuns.
  - Discussão sobre blocos de código, indentação e boas práticas.
  - Demonstrações simples no Code::Blocks de exemplos de códigos com tomada de decisões.

### 3.3.Demonstração Prática (60 minutos)

#### Exemplo 1: Uso simples do if

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int idade;
    printf("Digite sua idade: ");
    scanf("%d", &idade);

    if(idade >= 18){
        printf("Você é maior de idade.\n");
    }

    return 0;
}
```

Figura 1

Explicação: Neste programa o bloco if é executado apenas se a condição for verdadeira.

## Exemplo 2: Uso do if-else.

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

int main(){
    setlocale(LC_ALL, "Portuguese");

    int numero;
    printf("Digite um número: ");
    scanf("%d", &numero);

    if(numero % 2 == 0){
        printf("O número é par.\n");
    }
    else{
        printf("O número é ímpar.\n");
    }

    return 0;
}
```

Figura 2

Explicação: O programa mostra o uso do else para tratar o caso contrário da condição.

### Exemplo 3: Uso do if-else.

```
#include <stdio.h>

int main() {
    int senhaDigitada;
    int senhaCorreta = 1234; // Senha predefinida

    // Entrada do usuário
    printf("Digite a senha: ");
    scanf("%d", &senhaDigitada);

    // Comparação da senha digitada com a senha correta
    if(senhaDigitada == senhaCorreta) {
        printf("Acesso permitido!\n");
    }
    else{
        printf("Senha incorreta! Tente novamente.\n");
    }

    return 0;
}
```

Figura 3

Explicação: Programa de verificação de senha digitada pelo usuário.

Passos do programa:

1. A senha correta é armazenada na variável senhaCorreta.
2. O usuário digita a senha.
3. Se for correta, imprime "Acesso permitido!".
4. Caso contrário, imprime "Senha incorreta! Tente novamente."

#### Exemplo 4: Uso do if-else aninhado.

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

int main(){
    setlocale(LC_ALL, "Portuguese");

    int nota;
    printf("Digite a nota (0 a 10): ");
    scanf("%d", &nota);

    if(nota >= 7){
        printf("Aprovado.\n");
    }
    else if(nota >= 5){
        printf("Em recuperação.\n");
    }
    else {
        printf("Reprovado.\n");
    }

    return 0;
}
```

Explicação: Esse programa avalia múltiplas possibilidades em sequência

Bloco condicional:

- if(nota >= 7)
  - Se a nota for maior ou igual a 7, o programa imprime: "Aprovado."
- else if(nota >= 5)
  - Se a nota não for maior ou igual a 7, mas for maior ou igual a 5, o programa imprime: "Em recuperação."
- else
  - Se a nota for menor que 5, o programa imprime: "Reprovado."

Observação: Importância da ordem, as condições são avaliadas de cima para baixo. Assim que uma for verdadeira, as demais são ignoradas.

#### Exemplos de entrada e saída:

Entrada	Saída
8	Aprovado.
6	Em recuperação.
3	Reprovado.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

- O aluno deverá criar um programa que leia nome, idade e três notas, calcule a média e exiba uma mensagem personalizada de acordo com o desempenho do aluno, utilizando estruturas condicionais.
- Objetivo do Programa: Coletar nome, idade e três notas do usuário, calcular a média aritmética e exibir uma mensagem personalizada com base na média e idade informadas.

O programa deve fazer as seguintes operações:



1. Ler o nome do aluno (usar `fgets()` ou `scanf()` com `%[^\n]`).
2. Ler sua idade.
3. Ler três notas (ponto flutuante).
4. Calcular a média aritmética usando operadores aritméticos.
5. Exibir mensagens personalizadas com base nos resultados:
  - Se  $media \geq 9$  exibir a mensagem "Aprovado com excelente desempenho!"
  - Senão se  $media \geq 7$  exibir a mensagem "Aprovado, mas ainda pode melhorar."
  - Senão exibir a mensagem "Reprovado. Continue se esforçando!"

Observações:

- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

## 5. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecer eventuais problemas encontrados na implementação dos códigos.
- Reforçar os conceitos estudados.
- Estimular a criatividade e personalização dos programas.
- **Instruções sobre o relatório 3:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.

  <p><b>Instituto de Ciências Exatas e Tecnologia</b></p>	<p><b>Disciplina:</b> Programação Estruturada em C</p> <p><b>Título da Aula:</b> Estruturas de Repetição (Laços)</p>	<p><b>AULA 04</b></p>
---	--	-----------------------

## 1. Objetivos da Aula

Compreender o funcionamento das estruturas de repetição while, do-while e for na linguagem C.

Identificar as diferenças entre laços de repetição determinados (com contador) e indeterminados (com sentinela).

Aplicar estruturas de repetição para resolver problemas práticos e automatizar tarefas repetitivas.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 2 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

## 3. Estrutura da Aula

### 3.1. Abertura (10 minutos)

- Apresentação dos objetivos da aula e da importância dos laços de repetição na programação.
- Discussão breve: "Por que repetir tarefas automaticamente no código?"
- Conexão com situações reais (ex.: contagem de alunos, cálculo de parcelas, repetição de menu).

### 3.2.Revisão Conceitual (10 minutos)

- Explicação teórica dos três laços de repetição em C:
- while (teste no início).
- do...while (teste no final - executa pelo menos uma vez).
- for (estrutura com inicialização, condição e incremento).
- Conceitos de:
  - Repetição determinada (com contador).
  - Repetição indeterminada (com sentinela ou condição externa).
- Boas práticas: controle de variável, saída do laço, uso de break e continue.

### 3.3.Demonstração Prática (60 minutos)

**Exemplo 1: while (repetição controlada por contador (ou repetição determinada)).**

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int i = 0;

    while(i <= 5){
        printf("%d\n", i);
        i++;
    }

    return 0;
}
```

Figura 1

Explicação: Neste programa enquanto i for menor ou igual a 5, ele imprime o valor de i e utiliza \n para quebrar a linha. Depois incrementa i em 1.

O loop repete até que i seja 6, momento em que a condição  $i \leq 5$  se torna falsa e o laço termina.



## Exemplo 2: while (repetição indeterminada com sentinela)

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int valor = 0;

    while(valor != -1){
        printf("Digite um número (-1 para sair): ");
        scanf("%d", &valor);
    }

    printf("Programa encerrado.\n");
    return 0;
}
```

Figura 2

Explicação: Neste programa o laço continua enquanto o valor digitado for diferente de -1. Isso é uma repetição indeterminada.

## Exemplo 3: do...while (garante execução ao menos uma vez).

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    char opcao;

    do {
        printf("Deseja continuar? (s/n): ");
        scanf(" %c", &opcao);
    } while(opcao == 's' || opcao == 'S');

    printf("Fim do programa.\n");
    return 0;
}
```

Figura 3

Explicação: O bloco será executado ao menos uma vez, mesmo que a condição seja falsa desde o início. Ou seja, mesmo que o usuário digite 'n' de início.

#### Exemplo 4: for (repetição determinada com contador).

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int i;
    for(i = 1; i <= 5; i++){
        printf("Valor: %d\n", i);
    }

    return 0;
}
```

Figura 4

Explicação: Neste programa o for executa exatamente 5 vezes. Ideal para contagens ou repetições previsíveis.

#### Exemplo 5: Laço for Aninhado: Tabuada Completa

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int i, j;

    for(i = 1; i <= 10; i++){ // Número base da tabuada
        printf("\nTabuada do %d:\n", i);

        for(j = 1; j <= 10; j++){ // Multiplicador de 1 a 10
            printf("%d x %d = %d\n", i, j, i * j);
        }
    }

    return 0;
}
```

Figura 5

Explicação: Neste programa temos um laço for aninhado em linguagem C: para imprimir a tabuada completa de 1 a 10.

- O laço externo (i) percorre os números de 1 a 10 — ou seja, o número da tabuada.
- O laço interno (j) gera os multiplicadores de 1 a 10 para cada i.
- A saída mostra a tabuada completa, organizada por blocos.

#### **4. Atividade proposta 1 (Atividade Prática 20 minutos)**

O aluno deverá criar um programa em C que receba vários números do usuário e calcule a média. O programa deve permitir entrada contínua até o usuário digitar 0, e então calcular a média dos valores positivos.

Requisitos do programa:

- Utilizar o laço while para ler valores (parar quando o número for 0).
- Somar apenas valores positivos.
- Contar quantos valores foram somados.
- Exibir a média.

Observações:

- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

#### **5. Atividade proposta 2 (Atividade Prática 20 minutos)**

O aluno deverá criar um programa em C utilizando laços for aninhados para imprimir um triângulo de asteriscos com base no número de linhas informado pelo usuário.

Requisitos do programa:

- Solicitar ao usuário um número inteiro positivo representando o número de linhas do triângulo.
- Utilizar dois laços for aninhados para imprimir o padrão.
- A saída deve ter o seguinte formato (exemplo com 5 linhas).

```
Digite o número de linhas do triângulo: 5
*
**
***
****
*****
```



Figura 6

Observações:

- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

## 6. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecer eventuais problemas encontrados na implementação dos códigos.
- Reforçar os conceitos estudados.
- Estimular a criatividade e personalização dos programas.
- Revisão comparativa entre while, do...while e for.
- Sugestão de desafio extra: implementar um menu com repetição utilizando do...while.
- **Instruções sobre o relatório 4:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.

  <b>Instituto de Ciências Exatas e Tecnologia</b>	<b>Disciplina:</b> Programação Estruturada em C <b>Título da Aula:</b> Vetores (Arrays Unidimensionais)	<b>AULA 05</b>
---	--	----------------

## 1. Objetivos da Aula

Compreender o conceito de vetores (arrays unidimensionais) em linguagem C.

Declarar, preencher e acessar elementos de um vetor por meio de seus índices.

Utilizar vetores para armazenar coleções de dados do mesmo tipo.

Aplicar operações básicas com vetores, como leitura, escrita, soma, média e busca de valores.

Desenvolver a lógica para percorrer vetores utilizando estruturas de repetição.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 3 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

## 3. Estrutura da Aula

### 3.1. Abertura (10 minutos)

- Apresentação dos objetivos da aula e da importância dos vetores na programação.
- Analogia introdutória: caixas ou armários numerados que armazenam valores acessados por índices.
- Explicação do uso de vetores em situações reais: listas de notas, pontuação em jogos, preços, inventários, etc.

### 3.2.Revisão Conceitual (10 minutos)

- Definição de vetor (array unidimensional): Estrutura de dados que armazena elementos do mesmo tipo.
- Característica principal: Acesso aos elementos por índices (de 0 até  $n-1$ ).
- Explicação da declaração, inicialização e acesso a vetores em C.
- Destaque: Em C, o índice sempre começa em 0.

Conceito básico: Em C, vetores (ou arrays unidimensionais) são estruturas que armazenam vários valores do mesmo tipo em posições sequenciais (ou consecutivas) da memória. Cada elemento do vetor é acessado por meio de um índice numérico, que em C sempre começa em 0 e vai até tamanho - 1. Assim, em um vetor de tamanho  $n$ , o primeiro elemento está na posição 0 e o último na posição  $n-1$ .

Exemplo: Imagine um conjunto de caixas numeradas (índice) de 0 até  $n-1$ , sendo  $n=6$  a quantidade total de caixas. Cada caixa guarda um número (valor).

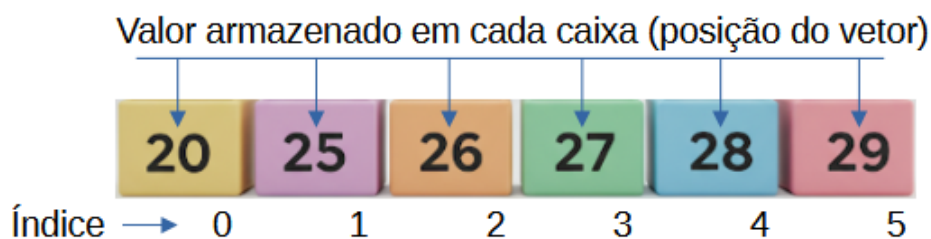


Figura 1

Em C o índice do vetor sempre é iniciado com 0 e vai até o tamanho do vetor menos 1.

Em C um vetor deve ter: `tipo nome_do_vetor[tamanho];`

Exemplo: sintaxe de um vetor na linguagem C: **`int notas[5];`**, onde `int` é o tipo (inteiro), `notas` é o nome do vetor e 5 entre colchetes é o tamanho do vetor, ou seja, quantas posições o vetor possui.

### 3.3.Demonstração Prática (60 minutos)

Podemos declarar o vetor:

```
int notas[5];
```

Podemos acessar suas posições da seguinte forma (nome e posição):

```
notas[0], notas[1], notas[2], notas[3], notas[4]
```

O índice sempre começa em 0, portanto, em um vetor de tamanho 5, o último elemento estará na posição 4 (ou seja, tamanho - 1).

Podemos inicializar as posições do vetor na declaração do vetor ou depois da declaração.

#### Exemplo 1: Preenchendo o vetor (inicializando na declaração):

```
int idades[3] = {10, 20, 30};
```

, esse vetor foi declarado do tipo inteiro (*int*), com o nome de *idades* e inicializado com os valores 10, 20 e 30.

Se você fornecer menos valores do que o tamanho, os restantes serão zerados automaticamente.

#### Exemplo 2: Preenchendo o vetor (inicializando após a declaração):

```
int notas[2];  
notas[0] = 10; // primeiro elemento  
notas[1] = 30; // último elemento
```

Cuidado com os limites do vetor: Em C não há verificação automática de limites, ou seja, não há proteção contra acesso fora dos limites do vetor.

### Exemplo 3: Vetor com tamanho definido pelo usuário:

Podemos pedir ao usuário para digitar o tamanho do vetor e depois de ter esse valor declaramos o vetor e passamos a variável que possui o tamanho ao vetor.

```
int tam;

printf("Digite o tamanho do vetor: ");
scanf("%d", &tam); //tam armazena o tamanho do vetor

int vetor[tam];
```

Caso o usuário digitasse 3, seria armazenado este valor na variável *tam*, consequentemente o vetor ao ser criado teria o tamanho 3.

O preenchimento do vetor é feito, normalmente, utilizando um laço for, que permite percorrer todas as posições de forma controlada.

### Exemplo 4: com tamanho pré-definido e uso de laço for:

```
int i, vetor[10];

for(i = 0; i < 10; i++){
    printf("Digite o %dº valor: ", i+1);
    scanf("%d", &vetor[i]);
}
```

Esse programa lê 10 números inteiros digitados pelo usuário e armazena cada um deles em um vetor chamado *vetor*.

Explicação:

- Laço `for(i = 0; i < 10; i++){...}`. Se repete 10 vezes, com *i* variando de 0 até 9.
- A cada repetição, uma mensagem é exibida ao usuário e um novo valor é lido e armazenado no vetor.
- `printf("Digite o %dº valor: ", i+1);`: Exibe a mensagem no formato "Digite o 1º valor: ", "Digite o 2º valor: ", ..., até o 10º valor.
  - Utilizamos *i+1* na exibição da mensagem para tornar a numeração mais intuitiva para o usuário (começando do 1), embora o índice real do vetor inicie em 0.



- `scanf("%d",&vetor[i]);`: Lê o número digitado pelo usuário e armazena na posição *i* do vetor.

Resumo: O programa vai pedir 10 valores ao usuário, um por um, e armazená-los sequencialmente no vetor nas posições `vetor[0]` até `vetor[9]`.

### Impressão do vetor:

**Exemplo 5: impressão sem uso de laço de repetição:** Esse método é viável apenas para vetores com poucos elementos, pois se torna inviável em vetores grandes.

```
printf("%d ", vetor[0]);  
printf("%d ", vetor[1]);  
...  
printf("%d ", vetor[n-1]);
```

O uso de laço *for* é o método mais comum para percorrer o vetor, permitindo acessar e imprimir cada item com precisão, independentemente do tamanho do vetor.

**Exemplo 6: Impressão com laço de repetição *for*:** as variáveis *i*, *tamanho* e o *vetor*, devem ser declaradas antes do *for*. Conforme abaixo.

```
int i, tamanho = 5;  
int vetor[tamanho];  
  
for(i = 0; i < tamanho; i++){  
    printf("%d ", vetor[i]);  
}
```

Exemplo 3: Impressão em ordem inversa com laço de repetição *for*:

```
int i, tamanho = 5;  
int vetor[tamanho];  
  
for(i = tamanho; i >= 0; i--){  
    printf("%d ", vetor[i]);  
}
```

Também é possível permitir que o usuário digite o tamanho do vetor, conforme já visto anteriormente.

## Exemplos completos com vetores:

### Exemplo 7 – Declaração e preenchimento manual de um vetor:

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int notas[5];

    notas[0] = 10;
    notas[1] = 8;
    notas[2] = 9;
    notas[3] = 7;
    notas[4] = 6;

    printf("Nota da terceira posição: %d\n", notas[2]);

    return 0;
}
```

Figura 2

Explicação: Cria um vetor de 5 posições e preenche manualmente. Acessa o valor da terceira posição (índice 2).

**Exemplo 8:** Programa que preenche manualmente um vetor e imprime seus valores em ordem crescente com um com laço *for*.

```
#include <stdio.h>

int main(){
    int i, tamanho = 3;
    int vetor[tamanho];

    vetor[0] = 1;
    vetor[1] = 2;
    vetor[2] = 3;
    //ou sem a variável tamanho
    //int vetor[5]={1, 2, 3};

    for(i = 0; i < tamanho; i++){
        printf("%d ", vetor[i]);
    }

    return 0;
}
```

Figura 3

Explicação: Neste programa declaramos um vetor com tamanho 3, preenchemos suas posições manualmente e, imprimimos esses valores usando um laço *for*. Saída na tela: 1 2 3.

### Exemplo 9: Operações: soma e média dos elementos.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int i, valores[5], soma = 0;
    float media;

    for(i = 0; i < 5; i++){
        printf("Digite o valor %d: ", i + 1);
        scanf("%d", &valores[i]);
        soma += valores[i];
    }

    media = soma / 5.0;

    printf("\nSoma: %d\n", soma);
    printf("Média: %.2f\n", media);

    return 0;
}
```

Figura 4

Explicação: Este programa lê 5 valores, acumula a soma dos valores do vetor em uma variável chamada *soma* e calcula e armazena a média em uma variável chamada *media*, no final imprime a soma e a média.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

O aluno deverá criar um programa em C que: Declare um vetor de 10 números inteiros. Solicite ao usuário que digite os 10 valores que serão armazenados no vetor.

Requisitos do programa:

Calcule e exiba:

- A soma dos valores pares armazenados no vetor.
- A média de todos os elementos do vetor.
- Utilize um laço for para percorrer o vetor.
- Para identificar os valores pares, utilize o operador %, que retorna o resto da divisão inteira.
- A média deve ser exibida com duas casas decimais, se possível.
- Comente o código explicando os principais blocos.
- Utilize mensagens claras e bem escritas para interagir com o usuário.

Observações:

- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

## 5. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecer eventuais problemas encontrados na implementação dos códigos.
- Reforçar os conceitos estudados.
- Estimular a criatividade e aprimoramentos: Funcionalidades extras, simulações e capturas de tela (prints).
- **Instruções sobre o relatório 5:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.



**Instituto de Ciências  
Exatas e Tecnologia**

**Disciplina:** Programação Estruturada em C  
**Título da Aula:** Matrizes (Arrays Bidimensionais)

**AULA 06**

## 1. Objetivos da Aula

Compreender o conceito e a estrutura de uma matriz (array bidimensional) em linguagem C.

Declarar e acessar elementos de uma matriz por meio de dois índices: linha e coluna.

Utilizar laços for aninhados para percorrer e manipular os dados armazenados em uma matriz.

Aplicar operações comuns com matrizes: leitura, exibição, preenchimento e soma de elementos.

Desenvolver a lógica para percorrer matrizes utilizando estruturas de repetição aplicada a estruturas tabulares (tabelas, mapas, tabuleiros, notas).

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 3 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

### 3. Estrutura da Aula

#### 3.1. Abertura (10 minutos)

- Apresentação do conceito de matriz: uma estrutura que armazena valores em duas dimensões (linhas e colunas).
- Explicação da analogia da matriz como um tabuleiro ou tabela de valores.
- Aplicações práticas: mapas, grades, tabelas de notas, inventários, sistemas de jogo etc.

#### 3.2. Revisão Conceitual (10 minutos)

- Definição de matriz (array bidimensional): estrutura de dados que armazena elementos do mesmo tipo organizados em duas dimensões – linhas e colunas.
- Acesso aos elementos: realizado por dois índices – o primeiro representa a linha e o segundo a coluna (`matriz[i][j]`).
- Declaração em C: `tipo nome[linhas][colunas];`
  - Exemplo: `int notas[3][4];` declara uma matriz com 3 linhas e 4 colunas.
- Preenchimento e leitura: normalmente feito com dois laços `for` aninhados, onde o primeiro percorre as linhas e o segundo percorre as colunas.
- Destaque: em C, os índices começam em 0, ou seja, a primeira linha é 0 e a última é `linhas - 1`; o mesmo vale para as colunas.

Conceito básico: Em linguagem de programação C, matrizes são vetores com duas dimensões: linhas e colunas. Elas são usadas para representar tabelas, grades, mapas, ou qualquer estrutura que envolva dois eixos (linha, coluna) como: (mapas e tabuleiros).

### 3.3.Demonstração Prática (60 minutos)

Matriz é um tabuleiro com linhas e colunas. Ideal para mapas, fases, batalhas.

Sintaxe da declaração de matriz:

*tipo nomeMatriz[tamanho1Linha][tamanho2Coluna];*

*int matriz1[4][3];*

*double matriz2[4][3];*

índice	Colunas		
	0	1	2
0			
1			
2			
3			

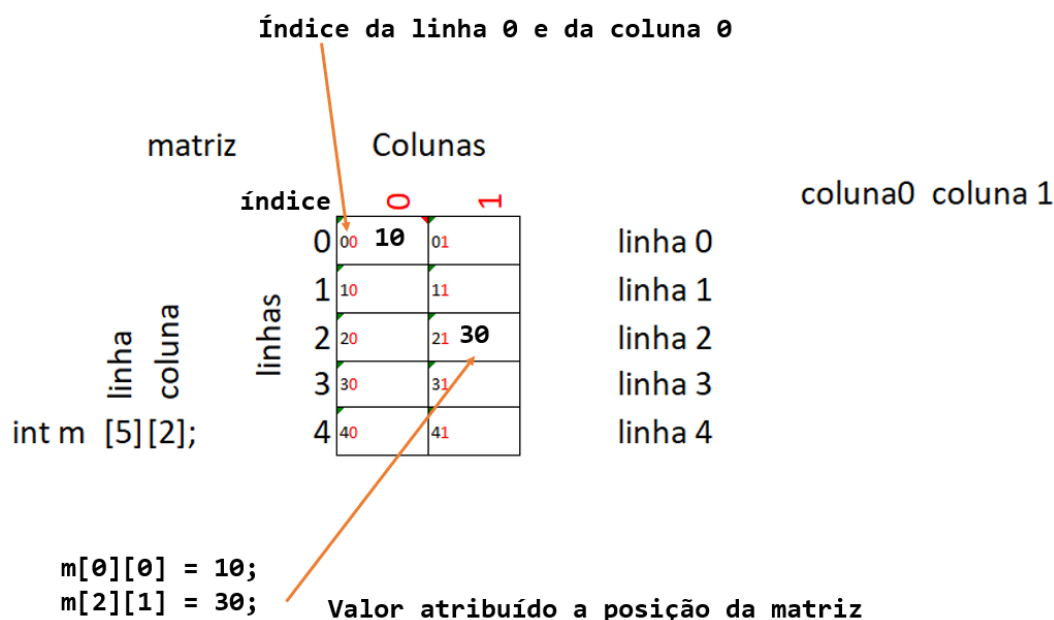
Exemplo da estrutura da matriz: O exemplo mostra uma representação visual de uma matriz bidimensional em C. Em uma matriz do tipo `matriz[lin][col]`, o primeiro índice representa a linha e o segundo índice representa a coluna.

	matriz[2][2]	
	coluna0	coluna1
linha0	matriz[0][0]	matriz[0][1]
linha1	matriz[1][0]	matriz[1][1]

	matriz[2][4]			
	coluna0	coluna1	coluna2	coluna3
linha0	matriz[0][0]	matriz[0][1]	matriz[0][2]	matriz[0][3]
linha1	matriz[1][0]	matriz[1][1]	matriz[1][2]	matriz[1][3]



Primeiro acessamos a linha, depois a coluna.



Matriz é um vetor de vetores: Internamente.

### Exemplo 1 – Impressão de uma matriz com valores pré-definidos:

```
#include <stdio.h>

int main() {
    int m[3][4] = {{3,4,5,5},{1,3,6,5},{8,1,2,5}};

    int i, j;
    for(i=0; i<3; i++){
        for(j=0; j<4; j++){
            //imprimindo a matriz
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Figura 1

Explicação: Programa para imprimir todos os elementos de uma matriz com valores pré-definidos na tela, formatando em forma de tabela (linhas e colunas).

- Cria uma matriz com 3 linhas e 4 com valores pré-definidos:

`int m[3][4] = {{3,4,5,5},{1,3,6,5},{8,1,2,5}};` Ilustração da matriz.

	Coluna 0	Coluna 1	Coluna2	Coluna 3
Linha 0	3	4	5	5
Linha 1	1	3	6	5
Linha 2	8	1	2	5

- O laço externo (*i*) percorre as linhas da matriz.
- O laço interno (*j*) percorre as colunas de cada linha.
- `printf("%d ",m[i][j]);` imprime cada valor da linha.
- `printf("\n");` separa visualmente as linhas da matriz.

Saída na tela:

3	4	5	5
1	3	6	5
8	1	2	5

**Exemplo 2:** Preenchimento da matriz com valores baseados na soma de índices.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int lin = 5, col = 2;
    int matriz[5][2];
    int i, j;

    for(i = 0; i < lin; i++){
        for(j = 0; j < col; j++){
            matriz[i][j] = i + j;
        }
    }

    printf("Matriz preenchida com i + j:\n");

    for(i = 0; i < lin; i++){
        for(j = 0; j < col; j++){
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Figura 3

Explicação: Cada posição da matriz recebe o valor da soma dos índices  $i + j$ .

Saída na tela:

```
Matriz preenchida com i + j:
0 1
1 2
2 3
3 4
4 5
```

### Exemplo 3: Soma de todos os elementos de uma matriz.

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

int main(){
    setlocale(LC_ALL, "Portuguese");

    int m[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int i, j, soma = 0;

    for(i = 0; i < 3; i++){
        for(j = 0; j < 3; j++){
            soma += m[i][j];
        }
    }

    printf("Soma dos elementos da matriz: %d\n", soma);

    return 0;
}
```

Figura 4

Explicação: Este programa soma todos os valores da matriz 3x3.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

Desenvolver um programa em linguagem C que trabalhe com matriz bidimensional, utilizando os conceitos de declaração, preenchimento, laços aninhados e operações básicas sobre seus elementos.

Requisitos do programa: O aluno deverá implementar um programa que:

- Declare uma matriz de inteiros com tamanho 3x3.
- Solicite ao usuário que insira os 9 valores da matriz (preenchimento linha a linha).

Após o preenchimento:

- Exiba a matriz em formato de tabela, com as linhas separadas por quebras de linha.
- Calcule e exiba a soma total de todos os elementos da matriz.
- Calcule e exiba a soma apenas dos valores pares da matriz.

Observações:



- Para identificar valores pares, use o operador de módulo:  $\text{valor} \% 2 == 0$ .
- Usar boas práticas de indentação.
- Cuidar da ortografia das mensagens.

### **Desafio de Ampliação (opcional):**

- Calcular a média dos valores da matriz.
- Identificar o maior e o menor valor armazenado.
- Solicitar dinamicamente o tamanho da matriz, permitindo que o usuário defina o número de linhas e colunas.

## **5. Encerramento e Orientações Finais (20 minutos)**

- Tempo para dúvidas: Esclarecimento de dúvidas sobre a estrutura e manipulação de matrizes.
- Comparação entre vetores e matrizes.
- Destaque para a utilidade dos laços aninhados.
- **Instruções sobre o relatório 6:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.

  <b>Instituto de Ciências Exatas e Tecnologia</b>	<b>Disciplina:</b> Programação Estruturada em C <b>Título da Aula:</b> Funções em C	<b>AULA 07</b>
---	--	----------------

## 1. Objetivos da Aula

Compreender o conceito de funções em linguagem C como forma de modularizar e reutilizar código.

Aprender a declarar, definir e chamar funções.

Aplicar a passagem de parâmetros (por valor).

Entender o escopo de variáveis.

Estimular o uso de funções para melhorar a organização, manutenção e clareza dos programas.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou a um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 4 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

## 3. Estrutura da Aula

### 3.1. Abertura (10 minutos)

- Apresentação do conceito de modularidade na programação.
- Explicação rápida do que são funções e por que são úteis: reutilização, organização e separação de responsabilidades.

### 3.2. Revisão Conceitual (10 minutos)

- Definição de função: bloco de código nomeado que pode ser reutilizado.
- Diferença entre função com retorno e função void.
- Escopo de variáveis: global vs. local.
- Declaração (protótipo), definição e chamada de funções.
- Passagem de parâmetros por valor (padrão em C).


### 3.3. Demonstração Prática (60 minutos)

Sintaxe básica:

```
tipo_retorno nome_funcao(lista de parâmetros) {  
    ...  
}
```

Exemplo:

```
int calcula_quadrado(int num) {  
    return num * num;  
}
```



Usado para retornar  
o valor da função.

Em C, apenas um valor  
pode ser retornado.

Componentes:

- Tipo de retorno: define o tipo de dado que a função devolve (*int*, *float*, *void* etc.).
- Nome da função: identificador usado para chamá-la.
- Parâmetros: são os dados recebidos (opcional).
- Corpo da função: instruções que serão executadas.
- *return*: interrompe a função e retorna o resultado (exceto em funções do tipo *void*, pois este tipo não possui retorno).

As funções são utilizadas para dividir um programa em módulos menores e reutilizáveis, facilitando a organização e manutenção do código. Ao definir uma função, as variáveis criadas dentro dela são chamadas de variáveis locais, ou seja, só podem ser usadas dentro daquele bloco de função.

Uso comum de funções:

- Usadas para modularizar o código.
- Uma função executa uma operação e retorna um valor.

Procedimento:

- Um procedimento é uma função que não retorna um valor.
- Usamos *void* para indicar isso.

```
void mostra_quadrado(int num) {  
    printf("%d\n", num * num);  
}
```



Função *main*: A função *main* em C deve ser declarada como *int main()* pois esse é o padrão recomendado pela linguagem C, conforme o padrão ISO C (normas ANSI/ISO).

```
int main() {  
    ...  
    return 0;  
}
```

- *main* é uma função também! Chamada automaticamente quando o programa é iniciado;
- A função *main* retorna um inteiro:
  - Retorna 0 quando o programa termina como esperado;
  - Retorna outro valor em caso de erro!
  - Por convenção, a função *main()* retorna um valor inteiro ao sistema operacional ao final da execução.

Exemplo 1 – Programa que calcula o quadrado de um número usando função antes da função *main()*.

```
#include<stdio.h>  
  
int calcula_quadrado(int num) {  
    return num * num;  
}  
  
int main() {  
    int n1;  
    scanf("%d", &n1);  
  
    int quad = calcula_quadrado(n1);  
  
    printf("%d\n", quad);  
  
    return 0;  
}
```




Figura 1

Explicação: Este programa lê um número inteiro digitado pelo usuário, calcula o seu quadrado por meio de uma função chamada `calcula_quadrado()` e depois imprime o resultado na tela.

- `int quad = calcula_quadrado(n1);` chamada da função deste exemplo. Aqui, o número digitado pelo usuário é passado como argumento para a função `calcula_quadrado`.
- O valor retornado pela função é armazenado na variável `quad`.

**Exemplo 2: Programa que calcula o quadrado de um número dentro de uma função e retorna o resultado.**

**Protótipo ou assinatura da função**

```
#include<stdio.h>

int calcula_quadrado(int num);

int main() {
    int n1;
    scanf("%d", &n1);
    int quad = calcula_quadrado(n1);
    printf("%d\n", quad);
    return 0;
}

int calcula_quadrado(int num) {
    return num * num;
}
```




Figura 2

Explicação:

- A declaração `int calcula_quadrado(int num);` diz ao compilador que existe uma função que retorna `int` e recebe um `int` como parâmetro.
- `calcula_quadrado(n1);` é chamada da função dentro do `main()`.
- A definição completa da função aparece depois do `main()` porque já foi previamente declarada.

**Exemplo 3:** Exemplo para ilustrar a passagem por referência para função (usando ponteiros), o conceito de ponteiro será discutido no próximo capítulo:

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

void modifReferencia(int *x) {
    printf("Dentro da função, antes da modificação: *ptr = %d\n", *x);
    *x = 100;
    printf("Dentro da função, depois da modificação: *ptr = %d\n", *x);
}

int main(){
    setlocale(LC_ALL, "Portuguese");

    int valor = 10;

    printf("Antes da chamada da função: valor = %d\n", valor);

    //Passa o endereço de memória de valor
    modifReferencia(&valor);

    printf("Depois da chamada da função: valor = %d\n", valor);

    return 0;
}
```

Figura 3

Explicação: Neste exemplo, ao passar o endereço de memória de *valor* para a função *modifReferencia*, a função pôde alterar o valor original da variável *valor* na função *main* através do ponteiro *\*x*.

Saída na tela:

```
Antes da chamada da função: valor = 10
Dentro da função, antes da modificação: *ptr = 10
Dentro da função, depois da modificação: *ptr = 100
Depois da chamada da função: valor = 100
```

Resumo:

- Passagem por referência: a função recebe o endereço da variável. Alterações afetam o original.

**Exemplo 4:** Este programa ilustra o uso de variáveis com escopos diferentes em funções. Uso de variáveis locais e globais.

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

```
double resultado; ←
```

```
void calcula_quadrado(double num) {  
    resultado = num * num;  
}
```

```
double calcula_soma(double n1, double n2) {  
    double r; ←  
    r = n1 + n2;  
    return r;  
}
```

```
int main() {  
    int a = 2, b = 3; ←  
    resultado = calcula_soma(a, b);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
    calcula_quadrado(resultado);  
    printf("%.2lf\n", resultado);  
  
    return 0;  
}
```

Variável global

A variável **resultado** pode ser acessada a partir de qualquer função!

Variável local da função **calcula\_soma**

Variáveis locais da função **main**

Variáveis locais são acessíveis apenas da função onde foram declaradas.

Figura 4

Explicação:

Em vermelho, temos uma variável global *resultado*, que pode ser acessada por qualquer parte do programa. Já em verde, estão destacadas as variáveis *a* e *b*, que são locais da função *main()*, ou seja, só podem ser utilizadas dentro dela. Outra de cor verde a variável *r*, usada dentro da função *calcula\_soma()*, também é local e visível apenas nesse escopo.

Além disso, os parâmetros das funções também são considerados variáveis locais. Por exemplo, o parâmetro *num* da função *calcula\_quadrado()* e os parâmetros *n1* e *n2* da função *calcula\_soma()* são acessíveis apenas dentro de suas respectivas funções.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

O aluno deverá digitar manualmente o programa apresentado a seguir, compreender seu funcionamento, comentar cada bloco de código com explicações claras e, se possível, realizar melhorias com base nas ampliações sugeridas.

Requisitos do programa: O aluno deverá:

- Digitar o programa fornecido exatamente como apresentado.
- Comentar o código explicando o que faz cada parte (declarações, entrada de dados, chamada de função, etc.).
- Aplicar as melhorias propostas (opcional, mas recomendável).

**Atividade proposta 1:** Programa para contar caracteres de um nome: Este programa conta e exibe quantos caracteres possui um nome digitado pelo usuário, utilizando uma função que percorre a *string* manualmente (sem usar *strlen*).

```

#include <stdio.h>
#include <locale.h>

int contar_caracteres(char nome[]);

int main() {
    setlocale(LC_ALL, "Portuguese");

    char nome[50];
    printf("Digite seu nome: ");
    scanf(" %[^\\n]", nome); // lê nome com espaços

    int total = contar_caracteres(nome);
    printf("O nome possui %d caracteres.\\n", total);

    return 0;
}

int contar_caracteres(char nome[]) {
    int i = 0;
    //conta junto os espaços entre nome e sobrenome
    while(nome[i] != '\\0'){
        i++;
    }
    return i;
}

```

Figura 5

### Desafio de Ampliação (opcional):

- Usar fgetc() no lugar do scanf() para maior segurança na leitura do nome.
- Adicionar o campo idade e exibir a mensagem com nome e idade.
- Inserir outros campos como curso, matrícula ou e-mail.
- Criar uma função separada para exibir a mensagem final.
- Utilizar locale.h para exibir acentuação corretamente.

## 5. Encerramento e Orientações Finais (20 minutos)

- Tempo para dúvidas: Esclarecimento de dúvidas sobre funções.
- **Instruções sobre o relatório 7:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue.



Instituto de Ciências  
Exatas e Tecnologia

**Disciplina:** Programação Estruturada em C

**Título da Aula:** Ponteiros e Manipulação  
de Arquivos

**AULA 08**

## 1. Objetivos da Aula

Compreender o conceito de ponteiros em linguagem C e seu papel na manipulação de memória.

Aprender a declarar, inicializar e utilizar ponteiros.

Entender a relação entre ponteiros e arrays.

Compreender como funciona a passagem por referência utilizando ponteiros.

Introduzir a manipulação de arquivos: abertura, leitura, escrita e fechamento.

Aplicar operações básicas com arquivos: criação, escrita e leitura.

## 2. Recursos Necessários

- Computadores ou dispositivos com acesso à internet (laboratório de informática).
- Acesso ao compilador Code::Blocks (instalado localmente) ou um compilador online, como [OnlineGDB](#) ou [OneCompiler](#).
- Material de apoio (Capítulo 4 do livro-texto).
- Editor de texto (opcional), para organização e formatação do relatório final.

## 3. Estrutura da Aula

### 3.1. Abertura (10 minutos)

- A Introdução aos conceitos de ponteiros e arquivos em C.
- Discussão sobre a importância de manipular endereços de memória e arquivos para persistência de dados em C.



### 3.2. Revisão Conceitual (10 minutos)

- Ponteiros: variáveis que armazenam endereços de memória.
- Operador \* (desreferenciação) e & (endereço).
- Diferença entre ponteiros e variáveis normais.
- Arquivos: entrada e saída de dados em arquivos .txt.
- Funções: fopen(), fprintf(), fscanf(), fclose() e modos de abertura ("w", "r", "a", etc.).

### 3.3. Demonstração Prática (60 minutos)

Ponteiro é uma variável que armazena um endereço de memória de outra variável. Ou seja, um ponteiro guarda o endereço de memória de uma variável, e por meio dele é possível acessar ou modificar diretamente o valor armazenado nesse endereço. Serve para modificar valores diretamente. Para declarar um ponteiro, utiliza-se um asterisco (\*) antes do nome da variável, indicando que ela armazenará um endereço de memória:

```
int *p;
```

Neste exemplo, *p* é um ponteiro que pode armazenar o endereço de uma variável do tipo *int*.

Operadores com ponteiros:

- & (e comercial): retorna o endereço de uma variável.
- \* (Asterisco): usado para acessar o valor que está sendo apontado.

Exemplos de uso de ponteiros em C:

- Para manipular valores de variáveis diretamente na memória.
- Para passar variáveis por referência em funções (permitindo alterações reais).
- Leitura e escrita em arquivos usando ponteiros.
- Para trabalhar com *arrays* e *strings*.
- Para alocação dinâmica de memória.

### Exemplo 1: Exemplo simples com uso básico de ponteiros em C.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int x = 10;

    int *p;

    p = &x; // p recebe o endereço de x

    printf("Valor de x = %d\n", x);
    printf("Endereço de x = %p\n", &x);
    printf("Endereço armazenado em p = %p\n", p);
    printf("Valor apontado por p = %d\n", *p);

    return 0;
}
```

Figura 1

#### Explicação

- $x$  é uma variável comum.
- $p$  é um ponteiro que recebe o endereço de  $x$  com  $p = \&x$ ;
- $*p$  acessa o valor de  $x$ .
- $\&x$  mostra o endereço de memória da variável  $x$ .

Saída na tela: os valores do endereço de  $x$  mostrado na tela podem variar em cada execução do programa, porque o endereço na memória é determinado pelo sistema operacional no momento da execução do programa.

```
Valor de x = 10
Endereço de x = 0061FF18
Endereço armazenado em p = 0061FF18
Valor apontado por p = 10
```

Exemplo 2: Acesso direto a variáveis com ponteiros: Este programa mostra como acessar e alterar o valor de uma variável por meio de um ponteiro.

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    int x = 10;
    int *p = &x;

    printf("Valor de x: %d\n", x);
    printf("Endereço de x: %p\n", &x);
    printf("Valor apontado por p: %d\n", *p);

    *p = 200;
    printf("Novo valor de x: %d\n", x);

    return 0;
}
```

Figura 2

Explicação:

- $p$  aponta para o endereço de  $x$ .
- Ao fazer  $*p = 20$ , alteramos o valor de  $x$  diretamente.
- Mostramos o endereço de  $x$  e como o ponteiro acessa o mesmo valor.

Saída na tela:

```
Valor de x: 10
Endereço de x: 0061FF18
Valor apontado por p: 10
Novo valor de x: 200
```

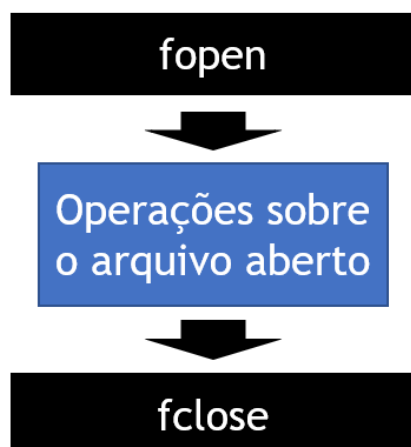
## Manipulação de Arquivos:

Arquivos em C são usados para armazenar dados de forma permanente, ou seja, mesmo após o encerramento do programa, as informações permanecem salvas no disco.

O uso de arquivos permite salvar dados entre execuções do programa, possibilitando diversas aplicações práticas, como:

- Armazenamento de dados para uso posterior.
- Gravação de pontuações, progresso de jogos ou status de atividades.
- Geração de relatórios, logs ou estruturas simples de banco de dados.
- Leitura de dados de entrada sem depender do teclado.
- Processamento de arquivos de texto, como .txt, .csv e arquivos binários.

Para usar arquivos, precisamos primeiro abrir (*fopen*) e depois fechar os arquivos (*fclose*).



- Usamos *fopen*, que recebe o modo de abertura do arquivo.
- Usamos *fclose* para fechar o arquivo.

## Principais funções de arquivos (*stdio.h*)

Função	Descrição
<code>fopen()</code>	Abre (ou cria) um arquivo
<code>fclose()</code>	Fecha o arquivo
<code>fprintf()</code>	Escreve dados formatados no arquivo
<code>fscanf()</code>	Lê dados formatados do arquivo
<code>fgetc()</code>	Lê um caractere do arquivo
<code>fputc()</code>	Escreve um caractere no arquivo
<code>fgets()</code>	Lê uma string do arquivo
<code>fputs()</code>	Escreve uma string no arquivo
<code>feof()</code>	Verifica se chegou ao final do arquivo (EOF)

Boas práticas com arquivos:

- Sempre verifique se `fopen()` retornou `NULL`.
- Sempre feche o arquivo com `fclose()`.
- Evite escrever fora do modo adequado (por exemplo, escrever em um arquivo aberto apenas para leitura).
- Use nomes claros para os arquivos e caminhos válidos no sistema operacional.

### Exemplo 3: Escrita em Arquivo.

```
#include <stdio.h>

int main() {
    FILE *arquivo = fopen("dados.txt", "w");
    if(arquivo != NULL) {
        fprintf(arquivo, "Olá, mundo!\n");
        fclose(arquivo);
        printf("Arquivo escrito com sucesso.\n");
    }
    else {
        printf("Erro ao abrir o arquivo.\n");
    }

    return 0;
}
```

Figura 3

Explicação: Cria ou sobrescreve um arquivo e grava uma mensagem de texto.

### Exemplo 4: Programa básico de leitura em arquivo:

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

int main() {
    setlocale(LC_ALL, "Portuguese");

    FILE *arquivo = fopen("saida.txt", "r");
    char linha[100];

    if(arquivo != NULL) {
        while(fgets(linha, 100, arquivo) != NULL) {
            printf("%s", linha);
        }
        fclose(arquivo);
    } else {
        printf("Erro ao abrir o arquivo para leitura.\n");
    }

    return 0;
}
```

Figura 4

Explicação:

- `FILE *arquivo = fopen("saida.txt","r");` abre o arquivo `saida.txt` no modo de leitura ("`r`"). `arquivo` é um ponteiro do tipo `FILE`.
- `char linha[100];` cria um vetor de `char` com 100 posições para armazenar cada linha lida do arquivo.
- ```
while (fgets(linha, 100, arquivo) != NULL) {  
    printf("%s", linha);  
}
```
- `while` lê linha por linha do arquivo com `fgets()`:
- A variável `linha` é onde será armazenado o texto. 100 é o tamanho máximo da `linha` (evita estouro).
- `arquivo` é o ponteiro para o arquivo aberto.
- A cada linha lida, imprime no console com `printf("%s",linha);`.
- `fclose(arquivo);` Fecha o arquivo após a leitura.

#### 4. Atividade proposta 1 (Atividade Prática 40 minutos)

O aluno deverá criar um programa em C que utilize ponteiros e arquivos para realizar o seguinte:

- Solicitar ao usuário seu nome e idade.
- Utilizar ponteiros para armazenar e modificar os dados.
- Gravar as informações em um arquivo chamado `cadastro.txt`.
- Ler o conteúdo do arquivo e exibir na tela.

Requisitos do programa: O aluno deverá:

- Declarar ponteiros para as variáveis nome e idade e usá-los para modificar diretamente os valores digitados pelo usuário.
- Utilizar `fprintf()` para gravação e `fscanf()` ou `fgets()` para leitura.
- Comentar o código explicando os principais blocos.
- Aplicar boas práticas de indentação e ortografia.

### **Desafio de Ampliação (opcional):**

- U adicionar novos campos no cadastro, validar entradas, separar funções para modularizar.

### **5. Encerramento e Orientações Finais (20 minutos)**

- Tempo para dúvidas: Revisão dos conceitos de ponteiros e arquivos.
- **Instruções sobre o relatório 8:** Orientar como deve ser o formato e o conteúdo do relatório que será entregue e prazo de entrega.