

# Unidade II

## 3 OPERAÇÕES E CONTROLE DE FLUXO

Nesta unidade, iniciaremos nossa jornada com os conceitos fundamentais de controle de fluxo em linguagem C. O objetivo é desenvolver o raciocínio lógico e compreender como um programa pode tomar decisões com base em condições estabelecidas, tornando-se mais inteligente e dinâmico.

As estruturas condicionais são usadas para tomar decisões dentro de um programa, permitindo que certas instruções sejam executadas apenas se determinadas condições forem atendidas. Em C, as principais estruturas condicionais são:

- if
- if ... else
- if ... else if ... else

Essas estruturas são amplamente utilizadas em diversas linguagens de programação, como C, Python, Java, JavaScript e C#, demonstrando sua importância e universalidade. Vamos entender cada uma delas com explicações e exemplos práticos.

### 3.1 Estrutura condicional: if

A instrução if é muito utilizada para controle de fluxo em C. Ela permite que um programa tome decisões com base em condições específicas. Ou seja, o comando if é utilizado quando o programa precisa tomar decisões com base em uma condição. Ele avalia uma expressão lógica que deve resultar em verdadeiro (1) ou falso (0).

Se a condição for verdadeira, o bloco de instruções delimitado pelas chaves {} será executado.

Sintaxe:

```
if (condição) {  
    // Bloco de código executado se a condição for verdadeira  
}
```

- O fluxo começa na condição.
- Se a condição for verdadeira (V.), o programa segue para executar as instruções.

- Se a condição for falsa (F), o fluxo desvia para outro caminho.
- O fluxo continua após a execução das instruções.

A condição pode envolver operações de comparação (==, !=, >, < etc.) e operações lógicas (&&, ||, !).

Essa verificação permite que o programa siga caminhos diferentes de acordo com os dados de entrada ou o estado das variáveis.

O bloco de instruções é o conjunto de comandos executado apenas se a condição for atendida. Caso contrário, o programa ignora esse bloco e segue para a próxima instrução.

Exemplo: verificar idade.

```
int idade = 18;

if(idade >= 18){
    printf("Pode votar.");
}
```

Assim, a condição `idade >= 18` é avaliada. Como `idade` vale 18, o programa imprime "Pode votar".

Exemplo: verificar nota.

```
if (nota >= 60 && nota <= 100) {
    printf("Aprovado");
}
```

Operadores utilizados: `>=`, `<=`, `&&`, `if`

Exemplo: validações de entrada.

Para tal, são usados operadores que garantem que o dado inserido esteja no formato correto ou dentro de um intervalo válido.

```
if(idade < 0 || idade > 120) {
    printf("Idade inválida");
}
```

Podemos olhar essa estrutura condicional por meio de um fluxograma para entender melhor a operação:

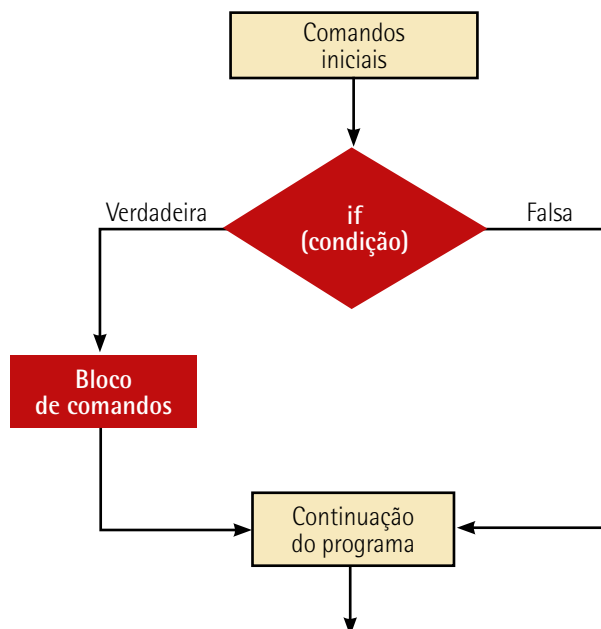


Figura 15 – Exemplo fluxograma estrutura if

A figura 15 representa um fluxograma da estrutura de decisão if na linguagem C. O fluxo de comandos tem os seguintes elementos:

- **Início:** o programa começa com a execução dos comandos iniciais. Em seguida, uma seta leva ao losango, que representa a análise da condição na estrutura de decisão if.
- **Condição if:** o losango representa uma estrutura condicional, na qual é avaliada uma expressão lógica como verdadeira ou falsa.
- **Saídas da condição:**
  - Se a condição for verdadeira (true), o fluxo segue para um bloco de comandos (um retângulo) dentro do if.
  - Se a condição for falsa (false), o fluxo desvia diretamente para a continuação do programa (seta inferior à direita).
- **Execução do bloco de comandos:** caso a condição seja verdadeira, o fluxo passa pelo bloco de comandos e, em seguida, retorna ao fluxo principal.
- **Continuação:** após a avaliação e possível execução do bloco de comandos, o fluxo continua para o restante do programa.

## 3.2 Estruturas condicionais: if – else

A estrutura if – else é uma extensão da estrutura condicional if e permite ao programa escolher entre dois caminhos de execução: um quando a condição for verdadeira e outro quando for falsa.

**Quadro 4 – Estrutura de decisão if if – else em C**

Estrutura	Funcionamento
if	Executa um bloco de código se a condição for verdadeira
else	Executa um bloco alternativo caso a condição do if seja falsa

O else é usado para definir um bloco de código executado caso a condição do if seja falsa (0).

Sintaxe:

```
if (condição) {  
    // Código executado se a condição for verdadeira  
}  
else {  
    // Código executado se a condição for falsa  
}
```

Exemplo: lógica de entrada validada com operadores lógicos.

```
int idade = 20;  
if (idade >= 18 && idade <= 60) {  
    printf("Idade válida para cadastro");  
}  
else {  
    printf("Idade fora do intervalo");  
}
```

Operadores utilizados: =,>,<,&&

Analisando a estrutura if – else por meio de um fluxograma, podemos perceber que, se a condição for verdadeira, um bloco de comandos será executado; caso contrário, outro bloco de comandos diferente será executado.

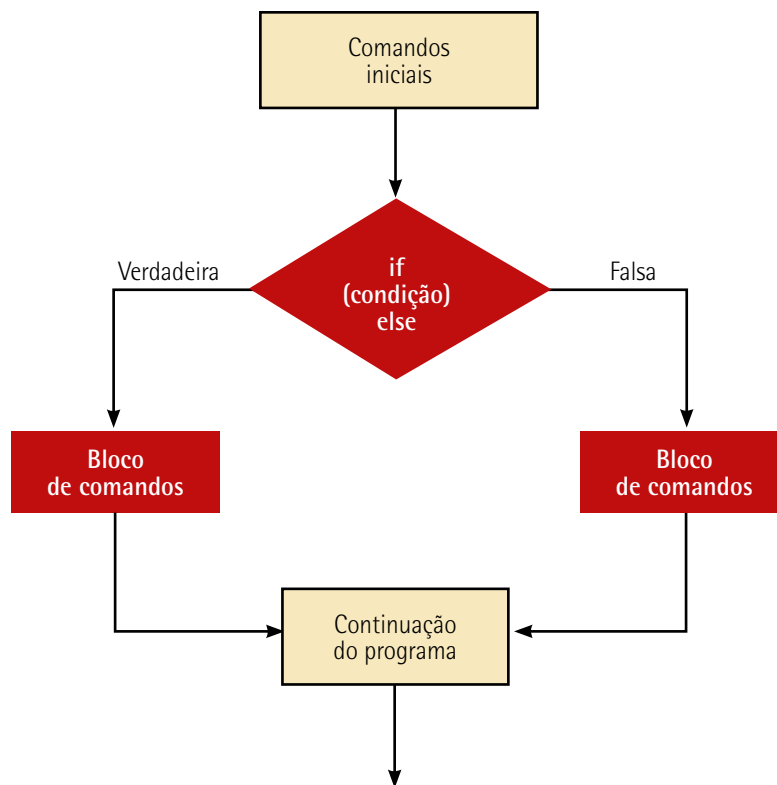


Figura 16 – Exemplo fluxograma estrutura if – else

A figura representa o fluxograma da estrutura de decisão `if – else` na linguagem C. Aqui temos a explicação do fluxograma:

- **Comandos iniciais:** indica o início do fluxo do programa.
- **Decisão if – else:** representado por um losango, quando uma condição é avaliada.
  - Se a condição for verdadeira, o fluxo segue para o bloco de comandos do lado esquerdo.
  - Se a condição for falsa, o fluxo segue para o bloco de comandos do lado direito.
- **Execução dos blocos de comandos:** dependendo do resultado da condição, um dos dois blocos é executado.
- **Continuação do programa:** após a execução do bloco correspondente, o fluxo converge para a continuação do programa.

O quadro 5 apresenta uma comparação entre if e if – else em C.

**Quadro 5**

Característica	if	if – else
Estrutura	Apenas verifica uma condição e executa um bloco de código se for verdadeira	Verifica uma condição e executa um bloco se for verdadeira; caso contrário, executa outro bloco
Bloco de código executado quando a condição é falsa	Nenhum (o programa segue para a próxima instrução)	Executa um bloco alternativo
Uso recomendado	Quando só há uma ação a ser executada caso a condição seja verdadeira	Quando há duas ações distintas: uma para verdadeiro e outra para falso
Exemplo	<pre>if (x &gt; 10){     printf("Maior que 10"); }</pre>	<pre>if (x &gt; 10){     printf("Maior que 10"); } else{     printf("Menor ou igual a 10"); }</pre>

## Exemplos gerais de uso da condição if

A seguir, há um exemplo de código em C com o uso da estrutura condicional if juntamente com os operadores lógicos e condicionais, logo, temos uma explicação do código:

```
#include <stdio.h>

int main() {
    int a = 5, b = 10, c = 5;

    if (a > 0 && b > a) {
        printf("Ambas as condições são verdadeiras.\n");
    }

    if (a == c || b < 0) {
        printf("Pelo menos uma das condições é verdadeira.\n");
    }

    if (!(a == b)) {
        printf("A condição 'a == b' é falsa, então '!(a == b)' é verdadeira.\n");
    }

    return 0;
}
```

Saída na tela:

```
Ambas as condições são verdadeiras.  
Pelo menos uma das condições é verdadeira.  
A condição 'a == b' é falsa, então '!(a == b)' é verdadeira.
```

Figura 17

Observe a seguir a explicação simplificada do código.

No primeiro if, verificamos se  $a > 0$  e  $b > a$ . Como ambas as condições são verdadeiras, a mensagem correspondente será exibida.

No segundo if, testamos se  $a == c$  ou  $b < 0$ . Como  $a == c$  é verdadeiro, o programa imprime que pelo menos uma das condições é verdadeira.

No terceiro if, usamos o operador ! (NOT) para inverter a verificação  $a == b$ . Como  $a$  e  $b$  são diferentes,  $a == b$  é falso, portanto, muda a condição do if para verdadeiro, então a última mensagem também será exibida.

O comando if – else é utilizado quando existem duas possibilidades de ação e você precisa especificar o que fazer em cada caso: tanto se a condição for verdadeira quanto se for falsa.

Use if – else quando:

- Você tiver uma condição que pode ser verdadeira ou falsa.
- Desejar executar uma ação diferente para cada um desses casos.



### Saiba mais

Entenda melhor o conteúdo abordado lendo a obra a seguir:

MIZRAHI, V. V. *Treinamento em linguagem C*. 2. ed. São Paulo: Pearson, 2008.

Observe outro exemplo: se um aluno precisa ser aprovado ou reprovado com base em uma nota. A seguir, acentua-se o uso da estrutura condicional if – else juntamente com os operadores lógicos e condicionais.

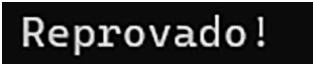
```
#include <stdio.h>

int main() {
    int nota = 50;

    if(nota >= 60){
        printf("Aprovado!\n");
    }
    else{
        printf("Reprovado!\n");
    }

    return 0;
}
```

Saída na tela:



Reprovado!

Figura 18

Observe a seguir a explicação simplificada do código.

- **Avaliação da condição `if(nota >= 60)`**
  - O programa verifica se a variável `nota` é maior ou igual a 60.
  - Se essa condição for verdadeira, o programa executa o comando dentro do bloco `{ }` do `if`, imprimindo "Aprovado!" e finalizando com quebra de linha `\n`.
- **Caso a condição seja falsa (`else`)**
  - Se a nota for menor que 60, o programa executa o bloco dentro do `else`, imprimindo "Reprovado!" e finalizando com quebra de linha `\n`.

Nesse exemplo, a nota do aluno é 50, portanto, será impresso "Reprovado!".

Esse tipo de estrutura é muito utilizado em sistemas de notas acadêmicas, validação de entrada de dados, tomada de decisões condicionais, entre outros.



Agora outro exemplo: programa simples de verificação de senha digitada pelo usuário:

```
#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;
}
```

Observe a seguir a explicação simplificada do código acentuado:

- A senha correta é armazenada em uma variável `senhaCorreta` do tipo `int`, com o valor 1234.
- O usuário digita a senha, que é armazenada na variável `senhaDigitada`.
- A comparação é feita diretamente com o comando:

```
if(senhaDigitada == senhaCorreta)
```

- Se forem iguais, exibe "Acesso permitido!".
- Se forem diferentes, exibe "Senha incorreta! Tente novamente.."

Esse método é útil para senhas numéricas, mas não funciona para senhas alfanuméricas, pois `scanf("%d",&senhaDigitada)`; aceita apenas números inteiros.

No exemplo a seguir, o comando verifica se um número é positivo.

```
#include <stdio.h>

int main() {
    int numero;

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

    if (numero >= 0) {
        printf("Número positivo.\n");
    }
    else {
        printf("Número negativo.\n");
    }

    return 0;
}
```

Explicação resumida: perceba que a decisão ocorre na linha 9. Se a condição for verdadeira, executará o comando na linha 10. Ou seja, se a variável `numero` for maior ou igual a zero, imprime "Número positivo."

Em caso falso, executa o comando da linha 13. Ou seja, caso contrário (`else`), imprime "Número negativo."



## Lembrete

As estruturas condicionais permitem que o programa tome decisões com base em condições. Use `if`, `if...else` e `if...else if...else` para controlar o que será executado. Elas são fundamentais para tornar seu código inteligente e adaptável às entradas do usuário.

Dica: sempre use chaves `{}` mesmo com apenas uma instrução dentro do `if` para evitar erros de lógica.

Por fim, a seguir temos um programa clássico em computação, que verifica se um determinado número é par ou ímpar.

```
1  #include <stdio.h>
2
3  int main() {
4      int numero;
5
6      printf("Digite um numero: ");
7      scanf("%d", &numero);
8
9      if(numero%2 == 0) {
10         printf("Número é par.\n");
11     }
12     else{
13         printf("Número é ímpar.\n");
14     }
15
16     return 0;
17 }
```

Os números no início de cada linha ajudam identificar as mudanças. Perceba que a decisão ocorre na condição `if(numero%2 == 0)` **linha 9**, ou seja, se a condição for verdadeira o número é par, caso contrário, sabemos que ele é ímpar. O operador `%` entre a variável `numero` e o número 2 retorna o resto da divisão inteira. Por exemplo: se o usuário digitar 5, o resto da divisão será 1. Portanto, a operação não será igual a zero, indo diretamente para o `else`, imprimindo "Número é ímpar.\n".

### 3.3 Estrutura condicional: `if`, `else if`, `else` (encadeadas)

Em C, podemos usar estruturas condicionais encadeadas para testar múltiplas condições sequenciais. Isso é feito com o uso de `if`, `else if` e `else`, permitindo que o programa tome diferentes decisões dependendo do valor de uma variável.

Como exemplo, temos um programa que permite imprimir uma mensagem caso o valor digitado pelo usuário seja maior que zero, igual a zero e se for menor que zero. O código a seguir mostra o programa que testa e imprime se um número é maior, menor ou igual a zero. Trata-se da estrutura condicional encadeada.

```
#include <stdio.h>

int main() {
    int numero;

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

    if(numero > 0){
        printf("Número positivo.\n");
    }
    else if(numero == 0){
        printf("Número igual a zero.\n");
    }
    else{
        printf("Número negativo.\n");
    }

    return 0;
}
```

O programa lê um número inteiro digitado pelo usuário e determina se ele é positivo, negativo ou zero, utilizando estruturas condicionais (if, else if, else).

- Lê um número inteiro usando scanf().
- Verifica a condição:
  - Se o número for maior que zero, imprime "Número positivo.."
  - Se for igual a zero, imprime "Número igual a zero.."
  - Se for menor que zero, imprime "Número negativo.."

O programa apresentado demonstra o uso correto das estruturas condicionais (if, else if, else) para classificar um número digitado pelo usuário como positivo, negativo ou zero.

Agora vamos comparar dois números. Leia dois números inteiros e determine se o primeiro é maior, menor ou igual ao segundo.

No código a seguir, temos o exemplo do programa que verifica se o primeiro é maior, menor ou igual ao segundo, conforme condições solicitadas.

```
#include <stdio.h>
int main() {
    int n1, n2;

    printf("Digite dois números: ");
    scanf("%d %d", &n1, &n2);

    if (n1 > n2)
        printf("%d é maior que %d\n", n1, n2);

    else if (n2 > n1)
        printf("%d é maior que %d\n", n2, n1);

    else
        printf("Os números são iguais\n");

    return 0;
}
```

Explicação passo a passo:

- Solicita ao usuário que digite dois números inteiros.
- A estrutura if – else compara os dois valores.
  - Se  $n1 > n2$ , exibe que  $n1$  é maior.
  - Se  $n2 > n1$ , exibe que  $n2$  é maior.
  - Caso nenhuma das condições seja verdadeira, executa o else, pois sabemos que os valores são iguais, e exibe mensagem de igualdade.



### Observação

No exemplo do código anterior, if é utilizado sem chaves {}. Isso é permitido na linguagem C quando há apenas uma instrução a ser executada dentro do bloco condicional. No entanto, essa prática pode ser perigosa e levar a erros lógicos, especialmente se outra linha de código for adicionada logo após o if, acreditando-se que também faz parte da condição. Por esse motivo, recomenda-se sempre usar chaves {}, mesmo quando o bloco condicional tiver apenas uma instrução.

## Exemplos de aplicação

### Exemplo 1. Classificação etária com if encadeado

Faça um programa que leia a idade de uma pessoa e informe se ela é criança (0-11), adolescente (12-17), adulta (18-59) ou idosa (60+).

### Resolução

```
#include <stdio.h>
int main() {
    int idade;

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

    if(idade < 12){
        printf("Criança\n");
    }
    else if(idade < 18){
        printf("Adolescente\n");
    }
    else if(idade < 60){
        printf("Adulto\n");
    }
    else{
        printf("Idoso\n");
    }

    return 0;
}
```

Explicação passo a passo:

- Solicitação da idade do usuário.
- Usa if – else if para testar a faixa etária:
  - Se for a variável idade menor que 12 ( $\text{idade} < 12$ ), imprime a palavra entre aspas "Criança".
  - Senão, se  $\text{idade} < 18$ , ou seja, neste caso, entre 12 e 17: "Adolescente".
  - Senão, se entre 18 e 59: "Adulto".
  - Último caso (else), 60 ou mais: "Idoso".

### Exemplo 2. Avaliação de nota com if encadeado

Faça um programa que leia uma nota e informe se o aluno está "aprovado" se (nota  $\geq 7$ ); senão, se nota entre (5 a 6.9), em "recuperação"; senão, "reprovado", caso de (nota  $< 5$ ), lembrando que este último caso não tem teste, ou seja, o else não tem caso de teste, os testes ocorrem apenas quando temos if.

### Resolução

```
#include <stdio.h>

int main() {
    float nota;

    printf("Digite a nota do aluno: ");
    scanf("%f", &nota);

    if(nota >= 7){
        printf("Aprovado\n");
    }
    else if(nota >= 5){
        printf("Recuperação\n");
    }
    else{
        printf("Reprovado\n");
    }
    return 0;
}
```

Explicação passo a passo:

- Recebe a nota como número decimal.
- Usa estrutura condicional para verificar o desempenho:
  - Se for a variável nota  $\geq 7$ , imprime a palavra entre aspas "Aprovado".
  - Senão, se nota  $\geq 5$ , imprime a palavra entre aspas "Recuperação".
  - Senão: ou seja, casos de nota  $< 5$ , imprime a palavra entre aspas "Reprovado".

### Conclusão

Aprendemos como utilizar estruturas de decisão e aplicações. Essa abordagem é essencial para tomadas de decisão em C, permitindo que o programa reaja dinamicamente a diferentes entradas do usuário.

As estruturas de decisão são fundamentais para controlar o fluxo de execução de um programa em C. Elas permitem que o código tome decisões com base em condições lógicas, garantindo um comportamento dinâmico e interativo.

Principais características:

- **if()**: executa um bloco de código se a condição for verdadeira.
- **else**: executa um bloco se nenhuma das condições anteriores for atendida.
- **if()else if()**: permite testar múltiplas condições sequenciais.

Essas estruturas tornam os programas mais flexíveis, eficientes e adaptáveis, sendo amplamente utilizadas em validações, cálculos condicionais e controle de fluxos complexos. Seu uso correto é essencial para desenvolver algoritmos lógicos e inteligentes.

## 4 ESTRUTURAS DE REPETIÇÃO (LAÇOS)

Estruturas de repetição, também conhecidas como laços de repetição ou loops, são essenciais em programação e você as utilizará com muita frequência ao iniciar seus estudos em qualquer linguagem. Imagine não precisar escrever instruções repetitivas manualmente, mas sim deixar o programa fazer esse trabalho automaticamente! As estruturas de repetição têm esse papel, tornando o código mais simples, legível, reutilizável e eficiente.

Em C, existem três estruturas principais de repetição:

- **while**
- **do – while**
- **for**

Essas estruturas também estão presentes em outras linguagens como Python, Java, JavaScript, o que reforça sua importância e aplicabilidade universal.

Durante o desenvolvimento de programas, é muito comum nos depararmos com situações em que uma ou mais instruções precisam ser executadas diversas vezes. Essa repetição é conhecida como estrutura de repetição ou laço de repetição (loop).

Um loop é um trecho de código que será repetido enquanto uma condição lógica for verdadeira. Assim que essa condição deixar de ser satisfeita, o programa interrompe a repetição e continua a execução normalmente.



Agora vamos estudar os tipos de repetição.

Existem basicamente dois tipos principais de repetição: controlada por contador (repetição determinada) e controlada por sentinela (ou repetição indeterminada). Vamos estudá-las a seguir:

### **Repetição controlada por contador**

Nesse modelo, o número de vezes que o laço será executado é conhecido previamente. Ou seja, o programador sabe exatamente quantas repetições são necessárias.

Para isso, usamos uma variável de controle, também chamada de contador, que normalmente começa com um valor inicial e é atualizada a cada ciclo do loop (geralmente incrementada de 1 em 1).

Como funciona?

- Define-se um valor inicial para o contador.
- Estabelece-se uma condição que define o fim da repetição.
- O contador é atualizado em cada iteração.
- Quando a condição é falsa, o laço é encerrado.

### **Repetição controlada por sentinela**

Esse tipo de laço é usado quando não sabemos quantas vezes o código deverá ser repetido. O processo continua enquanto os dados forem válidos e só termina quando um valor especial, chamado de sentinela, for informado.

O que é uma sentinela?

É um valor utilizado para marcar o fim da entrada de dados. Esse valor deve ser único ou distinto dos dados normais, para que o programa consiga identificar o momento certo de parar.

Neste livro-texto, você aprenderá o funcionamento de cada estrutura e, por meio de exemplos práticos, aplicará conceitos como soma de números, tabuada, validação de entradas e muito mais. Vamos explorar cada uma delas detalhadamente.

## **4.1 Laço de repetição: while**

Na linguagem C, o laço while é uma estrutura de repetição usada para executar um bloco de código enquanto uma condição for verdadeira. A condição é verificada antes de cada repetição, o que significa que, se ela for falsa desde o início, o bloco não será executado nenhuma vez.

Sintaxe:

```
// Inicialização da variável de controle
while (condição) {
    // Bloco de código a ser executado

    // Atualização da variável de controle
}
```

- Inicialização da variável de controle (antes do while).
- Bloco de código a ser executado enquanto a condição for verdadeira.
- Atualização da variável de controle (para evitar loop infinito).

A estrutura while é normalmente usada quando não sabemos exatamente quantas vezes um bloco de código será repetido, mas desejamos que a execução continue enquanto uma condição for verdadeira.

Exemplos:

- Solicitar ao usuário para digitar a senha correta ou esperar até que algo aconteça.
- Repita uma ação até que as condições mudem. Por exemplo, até que um sensor atinja uma determinada temperatura ou até que determinados dados sejam alterados.



## Observação

Se a condição for mantida para sempre, um **loop infinito** será criado. Para evitar tais problemas, precisamos sempre testar o código.

O código a seguir mostra a repetição controlada por contador (ou repetição determinada).

```
#include <stdio.h>

int main(){
    int i = 1; // inicialização

    while (i <= 5) { // condição
        printf("%d\n", i); //impressão na tela
        i++; // atualização
    }

    return 0;
}
```

Saída na tela:

```
1
2
3
4
5
```

Figura 19

Explicação passo a passo:

- Declaramos e inicializamos a variável do tipo inteiro `i` com o valor 1.
- Enquanto `i` for menor ou igual a 5, ele imprime o valor de `i` e (`\n` quebra linha ou pula linha).
- Depois, incrementa `i` em 1.
- O loop repete até que `i` seja 6, momento em que a condição `i <= 5` se torna falsa e o laço termina.



### Observação

Caso o programador cometesse um erro na condição ou modificasse o valor inicial de `i` de forma que a condição já fosse falsa na primeira verificação, o corpo do laço `while` não seria executado nenhuma vez. Ou seja, nenhuma mensagem seria impressa, pois o laço não seria sequer iniciado.

No código a seguir, o programa imprime na tela os números de 1 a 5 em uma única linha, separados por espaço.

```
#include <stdio.h>

int main() {
    int i = 1; // inicialização

    while (i <= 5) { // condição
        //impressão na tela sem quebra de linha
        printf("%d ", i);
        i++; // atualização
    }

    return 0;
}
```

Saída na tela:

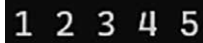


Figura 20

Explicação passo a passo:

- Declaramos e inicializamos a variável do tipo inteiro *i* com o valor 1.
- Enquanto *i* for menor ou igual a 5, ele imprime o valor de *i* na mesma linha. Perceba que, após o `%d`, agora temos um espaço, pois, sem o espaço, os valores impressos ficariam juntos.
- Depois, incrementa *i* em 1.
- O loop repete até que *i* seja 6, momento em que a condição `i <= 5` se torna falsa e o laço termina.



## Observação

Podemos manter a observação do exemplo anterior, para este exemplo ou qualquer outro usando laço `while` ou `for`.

O código a seguir faz uso de sentinela (repetição indefinida).

```
#include <stdio.h>

int main(){
    int nota;
    printf("Digite uma nota (-1 para encerrar): ");
    scanf("%d", &nota);

    while (nota != -1) {
        printf("Nota registrada: %d\n", nota);
        printf("Digite uma nota (-1 para encerrar): ");
        scanf("%d", &nota);
    }

    return 0;
}
```

Esse código apresenta um programa para ler notas até que o usuário digite -1, indicando que não há mais dados.

Explicação passo a passo:

```
int nota;
```

Aqui declaramos a variável `nota`, que será usada para armazenar o valor digitado pelo usuário. O `int` indica o tipo para números inteiros.

```
printf("Digite uma nota (-1 para encerrar): ");
```

Exibe-se uma mensagem no console pedindo que o usuário digite uma nota. Também é informado ao usuário que, ao digitar -1, o programa irá parar (esse é o valor sentinela).

```
scanf("%d", &nota);
```

Aguarda-se a entrada de um valor inteiro digitado pelo usuário. O valor digitado é armazenado na variável `nota`. Usamos `&nota` porque a função `scanf` precisa do endereço da variável para armazenar o valor lido.

```
while (nota != -1)
```

Essa é a estrutura de repetição do tipo `while`. A condição `nota != -1` significa: "enquanto a nota digitada for diferente de -1, continue repetindo o bloco".

Assim, o loop só termina quando o usuário digitar -1.

Bloco dentro do `while()` entre `{ ... }`:

```
printf("Nota registrada: %d\n", nota);
```

Exibe-se no console a nota que foi digitada, confirmando o registro da entrada.

```
printf("Digite uma nota (-1 para encerrar): ");  
scanf("%d", &nota);
```

Solicita-se uma nova entrada ao usuário e armazena-se esse novo valor na variável `nota`. A nova entrada será testada novamente na condição `while`.

O código a seguir mostra como ler caracteres até encontrar um ponto '.'.

Esse programa lê um caractere digitado pelo usuário por vez e imprime o caractere lido até que o usuário digite um ponto final (.). Quando o ponto é digitado, o laço termina e o programa finaliza.

```
#include <stdio.h>

int main() {
    char c;
    printf("Digite um caractere ou (ponto '.' para encerrar): ");
    scanf(" %c", &c);
    while (c != '.') {
        printf("Caractere lido: %c\n", c);
        printf("Digite um caractere ou (ponto '.' para encerrar): ");
        scanf(" %c", &c);
    }

    return 0;
}
```

Explicação passo a passo:

```
char c;
```

Declara-se a variável c do tipo caractere, que armazenará o valor digitado pelo usuário.

```
printf("Digite um caractere ou (ponto '.' para encerrar): ");
```

Mostra-se uma mensagem solicitando um caractere.

```
scanf(" %c", &c);
```

Lê-se um caractere e ele é armazenado em c.



## Observação

O espaço antes do %c, ou seja, antes da leitura de caractere, é importante: ele descarta espaços em branco anteriores, como \n ou espaço, que podem causar erros na leitura após o Enter.

```
while (c != '.')
```

Inicia-se um laço while, que será executado enquanto o caractere digitado for diferente de ponto (.).

```
printf("Caractere lido: %c\n", c);
```

Exibe-se o caractere lido na tela.

```
printf("Digite um caractere ou (ponto '.' para encerrar): ");  
scanf(" %c", &c);
```

O programa solicita um novo caractere e o lê novamente. Se for um ponto '.', o laço será encerrado.

O código a seguir traz um programa de um contador de 1 até 10 utilizando a estrutura de repetição while.

```
#include <stdio.h>  
  
int main() {  
  
    int i = 1;  
    while (i <= 10) {  
        printf("%d\n", i);  
        i++;  
    }  
  
    return 0;  
}
```

Observe a seguir a explicação simplificada:

- A variável i é declarada como do tipo inteiro (int) e inicializada com 1 antes do laço.
- A condição é verificada antes de cada repetição.
- Sendo verdadeiro o cálculo da condição, o programa faz a impressão do valor de i.
- Em seguida, o incremento de i acontece dentro do bloco do laço, e o valor de i passa a ser i + 1. Isso ocorre até i ter o valor 11, situação na qual o laço acaba, pois a condição passa ser falsa.
- Útil quando a repetição depende de uma condição que pode ser modificada dinamicamente.
- Serão mostrados na tela os valores de 1 até 10, um em cada linha.

O código a seguir apresenta um programa de um contador de 1 até n, sendo n um valor digitado pelo usuário, utilizando a estrutura de repetição while.

```
#include <stdio.h>

int main(){
    int i = 1, n;

    printf("Digite um valor para n: ");
    scanf("%d", &n);

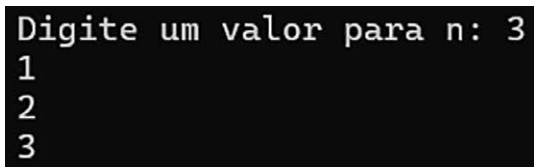
    while(i <= n){
        printf("%d\n", i);
        i++;
    }

    return 0;
}
```

Observe a seguir a explicação simplificada:

- A variável começa com 1.
- A condição  $i \leq n$  é verificada antes de cada repetição.
- Enquanto for verdadeira, imprime i e o incrementa.

Supondo que, ao executar esse programa, o usuário digite o valor 3, será impresso na tela:



```
Digite um valor para n: 3
1
2
3
```

Figura 21

O código a seguir demonstra o uso de estruturas de repetição para operações acumulativas simples (somador de 1 até 5) utilizando a estrutura de repetição while.



```
#include <stdio.h>

int main() {
    int i = 1;
    int soma = 0;

    while (i <= 5) {
        soma += i; //soma = soma + i;
        i++;
    }

    printf("Soma total: %d\n", soma);
    return 0;
}
```

Observe a seguir a explicação simplificada:

- A variável `i` é inicializada com 1.
- A variável `soma` é inicializada com 0. Isso é essencial, porque ela será responsável por acumular os valores a cada iteração. Caso não seja inicializada, poderá conter um valor indefinido da memória (lixo), o que geraria um resultado incorreto na soma final.
- Enquanto `i` for menor ou igual a 5, o valor é somado e armazenado na variável `soma`.
- `i` é incrementado a cada volta.
- No final imprime a mensagem "Soma total: %d\n" e o valor de `soma` no lugar do %d.

O código a seguir mostra impressão dos números de 10 até 1 (ordem decrescente). Ele realiza essa operação utilizando a estrutura de repetição `while`.

```
#include <stdio.h>

int main() {
    int i = 10;

    while (i >= 1) {
        printf("%d\n", i);
        i--;
    }

    return 0;
}
```

Observe a seguir a explicação simplificada:

- A variável `i` é inicializada com 10.
- Enquanto `i` for maior ou igual a 1, imprime o valor de `i` e decrementa.
- Quando `i` valer 0 o programa termina, pois a condição `while` será falsa.

Perceba que esse programa apresenta os valores na tela em ordem decrescente de 10 até 1.  
Saída na tela:



```
10
9
8
7
6
5
4
3
2
1
```

Figura 22

O programa a seguir mostra o exemplo de uma tabuada de um número `n` digitado pelo usuário.

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

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

    int n, i = 0;

    printf("Digite um número para ver a tabuada: ");
    scanf("%d", &n);

    while (i <= 10) {
        printf("%d x %d = %d\n", n, i, n * i);
        i++;
    }

    return 0;
}
```

A variável *i* inicia em 0. Enquanto *i* for menor ou igual a 10, o valor é calculado, exibe a multiplicação e incrementa *i*, repete o laço.

Supondo que, ao executar esse programa, o usuário digite o valor 3, será impresso na tela:

```
Digite um número para ver a tabuada: 3
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

Figura 23

### Exemplos de aplicação

A seguir, serão destacados exercícios com `while` na linguagem de programação C e, ao final, suas respectivas resoluções.

#### Exemplo 1. Bonificação de produtos de 1 até 20 (multiplicada por 10)

Um supermercado realizou uma promoção com 20 produtos cadastrados. Cada produto recebeu um valor de bonificação correspondente ao seu número de ordem multiplicado por 10. Por exemplo: produto 1 x 10 = R\$ 10.

Crie um programa em linguagem C que:

- Calcule o valor da bonificação para cada um dos 20 produtos, multiplicando o número do produto por 10.
- Exiba na tela uma linha para cada produto, informando o valor recebido e valor total pago em bonificações:
  - Exemplo para dois produtos:
    - "Produto 1 recebeu R\$ 10".
    - "Produto 2 recebeu R\$ 20".
    - "Valor total em bonificações R\$ 30".

## Resolução

O código a seguir apresenta a solução do programa bonificação de produtos de 1 até 20 (multiplicada por 10) com while.

```
#include <stdio.h>

int main() {
    int i = 1;
    int bonificacao;
    int total = 0;

    while (i <= 20) {
        bonificacao = i * 10;
        printf("Produto %d recebeu R$ %d\n", i, bonificacao);
        total = total + bonificacao; //ou total += bonificacao;
        i++;
    }

    printf("Valor total em bonificações R$ %d\n", total);

    return 0;
}
```

Declaração de variáveis:

- i é o contador que representa o número do produto e inicia com 1.
- bonificacao armazena o valor da bonificação de cada produto.
- total acumula o valor total pago em bonificações e inicia com 0.

Laço while(i <= 20): o laço será repetido 20 vezes. Dentro do laço:

- Para o cálculo da bonificação, em cada repetição, o valor da bonificação é calculado como número do produto x 10 (i \* 10).
- Exibição da bonificação individual.
- Acúmulo do total.
- O valor de i é incrementado em 1.

Exibição do total: após sair do laço, exibe a mensagem final.

### Exemplo 2. Somar notas de 20 alunos e calcular a média da turma

Uma professora precisa lançar as notas de 20 alunos e, ao final, saber a média geral da turma. Crie um programa em C que:

- Peça ao usuário que digite a nota de cada aluno.
- Some todas as notas.
- Calcule e exiba a média final da sala.

### Resolução

O código a seguir apresenta a solução do programa para somar as notas de 20 alunos e calcular a média da turma com while.

```
int main() {  
    int i = 1;  
    float nota, soma = 0, media;  
  
    while (i <= 20) {  
        printf("Digite a nota do aluno %d: ", i);  
        scanf("%f", &nota);  
  
        soma += nota; // soma = soma + nota  
        i++;  
    }  
  
    media = soma / 20;  
  
    printf("Média da turma: %.2f\n", media);  
  
    return 0;  
}
```

Observe a seguir a explicação simplificada:

- A variável *i* é o contador de alunos (inicia em 1).
- A variável *nota* armazena a nota digitada.
- A variável *soma* acumula todas as notas.
- A variável *media* armazenará o resultado final.

- **Laço while ( $i \leq 20$ ):** solicita a nota de cada aluno, de 1 até 20.
- **Acumulação da soma:** a cada iteração, a nota é somada à variável soma.
- **Cálculo da média:** após o laço, a média é calculada dividindo a soma por 20.
- **Saída formatada:** a média da turma é exibida com duas casas decimais (% 2f).

**Exemplo 3.** Imprimir os números inteiros ímpares de 1 até n

Um professor deseja gerar uma lista de números para um jogo de adivinhação com os alunos. O jogo deve conter apenas os números ímpares entre 1 e n, onde n é informado pelo próprio professor.

Crie um programa em C que solicite ao usuário um número n e imprima na tela todos os números ímpares de 1 até n, usando a estrutura de repetição while.

### Resolução

O código a seguir apresenta a resolução do problema para imprimir os números inteiros ímpares de 1 até n com while.

```
#include <stdio.h>

int main() {
    int n, i = 1;

    printf("Digite um número inteiro positivo: ");
    scanf("%d", &n);

    printf("Números ímpares de 1 até %d:\n", n);

    while(i <= n) {
        if(i % 2 != 0) {
            printf("%d\n", i);
        }
        i++;
    }

    return 0;
}
```

Observe a seguir a explicação simplificada:

- Primeiro, declaramos no programa duas variáveis:  $n$  (valor final) e  $i$  (contador), iniciando  $i$  com 1.
- Solicita que o usuário digite um número inteiro e armazena na variável  $n$ .
- Exibe uma mensagem indicando o início da lista de ímpares até  $n$ .
- Entra no laço `while`, que será executado enquanto  $(i \leq n)$ .
- Dentro do laço, verifica se  $i$  é ímpar com  $(i \% 2 \neq 0)$ . Se for ímpar, imprime o valor de  $i$ .
- Incrementa  $i$  em 1 a cada repetição.

### Exemplo 4. Imprimir os múltiplos de um número até 100

Um supermercado deseja identificar todos os produtos que receberão desconto múltiplo. Para isso, é definido um número  $n$  (por exemplo, 5), e todos os produtos cujo número de cadastro seja múltiplo de  $n$ , no intervalo de 2 até 100, receberão o desconto. Regras:

- 1) O número  $n$  deve ser digitado pelo usuário e estar entre 2 e 10, inclusive.
- 2) O programa deve identificar e exibir na tela todos os múltiplos de  $n$  entre 2 e 100.
- 3) Caso o número digitado esteja fora do intervalo, o programa deve repetir a solicitação até receber um número válido.

Escreva um programa em linguagem C que:

- Leia um número inteiro  $n$  do usuário, validando se ele está entre 2 e 10.
- Utilize a estrutura de repetição `while` para imprimir todos os múltiplos de  $n$  entre 2 e 100, inclusive.

### Resolução

O código a seguir apresenta a resolução do problema para imprimir os múltiplos de um número  $n$  até 100 com `while`.

```
#include <stdio.h>

int main() {
    int n, i = 2;

    // Solicita o número até que seja válido (entre 2 e 10)
    printf("Digite um número entre 2 e 10: ");
    scanf("%d", &n);

    while (n < 2 || n > 10) {
        printf("Valor inválido. Digite um número entre 2 e 10: ");
        scanf("%d", &n);
    }

    // Imprime os múltiplos de n entre 2 e 100
    printf("Múltiplos de %d entre 2 e 100:\n", n);
    while (i <= 100) {
        if (i % n == 0) {
            printf("%d\n", i);
        }
        i++;
    }

    return 0;
}
```

Observe a seguir a explicação simplificada:

- **Entrada com validação:** o programa solicita ao usuário que digite um número entre 2 e 10. Se o número estiver fora do intervalo, ele repete a solicitação até o valor ser válido.
- **Inicialização do contador:** o valor inicial de *i* é 2, pois o enunciado pede para verificar os múltiplos no intervalo de 2 até 100.
- **Laço while para imprimir os múltiplos:** enquanto *i* for menor ou igual a 100 (*i* <= 100), o programa verifica se *i* é divisível por *n*. Se *i* for divisível por *n*, imprime o valor. Ou seja, a condição (*i*%*n* == 0), se verdadeira, indica que *i* é múltiplo de *n*.
- **Incremento:** a cada passo, *i* é incrementado em 1.

**Exemplo 5.** Verificar se números são pares ou ímpares até digitar -1

Um sistema precisa analisar uma lista de números digitados pelo usuário e informar, para cada um deles, se é par ou ímpar. A entrada dos dados deve continuar até que o número -1 seja digitado, o que indica o fim da lista.



Regras:

- 1) O programa deve ler vários números inteiros digitados pelo usuário.
- 2) Para cada número, o programa deve informar se ele é par ou ímpar.
- 3) O programa deve encerrar quando o usuário digitar -1.

Escreva um programa em linguagem C que:

- Leia números inteiros digitados pelo usuário.
- Para cada número, informe se ele é par ou ímpar.
- O programa deve repetir esse processo até que o número -1 seja digitado.
- Quando isso ocorrer, o programa deve encerrar com uma mensagem de finalização.

### Resolução

Verificar se números são pares ou ímpares até digitar -1. O código a seguir apresenta um programa que verifica se números são pares ou ímpares até digitar -1, com while.

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

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

    int numero;

    printf("Digite números inteiros > 0.\n");
    printf("Para encerrar, digite -1.\n");

    // Inicializa com um valor diferente de -1
    numero = 0;

    while (numero != -1) {
        printf("\nDigite um número: ");
        scanf("%d", &numero);

        if (numero == -1) {
            printf("Programa encerrado.\n");
        } else if (numero % 2 == 0) {
            printf("%d é par.\n", numero);
        } else {
            printf("%d é ímpar.\n", numero);
        }
    }

    return 0;
}
```

Explicação do funcionamento: considerando que o programa será utilizado corretamente, ou seja, o usuário digitará apenas números inteiros maiores que 0 ou -1 para encerrar.

- **Declaração da variável**

- **numero**: inicializado com 0, depois armazenará o valor digitado pelo usuário.

- **Mensagem inicial**

- O programa explica ao usuário que deve digitar números inteiros  $> 0$  e que o valor -1 encerra a execução.

- **Laço while (numero! = -1)**

- Utilizado para manter a repetição até o usuário digitar -1.

- **Leitura do número**

- Em cada repetição, o programa solicita que o usuário digite um número inteiro.

- **Verificação da condição de parada**

- Se o número for -1, o programa exibe uma mensagem de encerramento e sai do laço, pois a condição será falsa.

- **Verificação de paridade**

- Se o número for divisível por 2 ( $\text{numero} \% 2 == 0$ ), ele é par e imprime que o número é par.

- Caso contrário, ele é ímpar e imprime que o número é ímpar.

- **Repetição**

- O processo continua até que -1 seja digitado.

Após sair do laço while, ou seja, quando o usuário digitar -1, o programa prossegue para o próximo comando, neste caso, `return 0`; Esse comando finaliza a execução do programa, indicando ao sistema operacional que ele foi concluído com sucesso.

---



### Observação

Esse programa pode ficar ainda mais apropriado e mais natural com o uso do laço `do – while`, especialmente porque a leitura do número precisa acontecer pelo menos uma vez antes da verificação da condição de parada. Ou seja, com `do – while`, a leitura do número ocorre pelo menos uma vez antes da verificação da condição de parada. Isso torna esse exemplo mais claro e adequado, já que sempre queremos que o usuário digite um número antes de avaliar se o programa deve continuar.

Agora vamos aprender como utilizar o laço `do – while` e ver uma versão desse programa com `do – while`.

### 4.2 Laço de repetição: `do – while`

O loop `do – while` é uma estrutura de repetição usada em C quando precisamos garantir que uma ação aconteça pelo menos uma vez, mesmo que a condição seja falsa logo no início.

Ele é ideal para situações nas quais a primeira execução deve ocorrer antes mesmo da verificação da condição, como em menus de sistemas, validações de entrada e solicitações interativas.

A sintaxe básica de um `do – while` em C é a seguinte:

```
do{  
    // código que será repetido  
} while (condição);
```

O que acontece aqui?

- O bloco de código é executado primeiro.
- Depois, a condição é verificada.
- Se a condição for verdadeira, o laço repete.
- Se a condição for falsa, o laço termina.

#### Exemplo prático no ambiente de trabalho

Imagine que você desenvolve um sistema no qual o usuário precisa acessar um menu até escolher a opção de sair. Mesmo que ele digite a opção de saída logo na primeira tentativa, o menu ainda precisa ser exibido pelo menos uma vez.

No primeiro exemplo, deve-se exibir menu até o usuário digitar '0' para sair.

```
#include <stdio.h>

int main() {

    int opcao;

    do{
        printf("MENU PRINCIPAL\n");
        printf("1 - Cadastrar\n");
        printf("2 - Consultar\n");
        printf("0 - Sair\n");
        printf("Escolha uma opção: ");
        scanf("%d", &opcao);
    } while (opcao != 0);

    return 0;
}
```

Explicando:

- O menu será exibido pelo menos uma vez, independentemente do valor inicial de opcao.
- O usuário digita a opção.
- O loop continua enquanto o valor digitado for diferente de 0.
- Ao digitar 0, o loop termina e o programa segue para o encerramento.

Observe como o loop do – while funciona passo a passo:

- Executa o bloco de código (exibe o menu).
- Lê a entrada do usuário.
- Verifica a condição (por exemplo, opcao != 0):
  - Se verdadeiro, repete.
  - Se falso, encerra o loop.

Use `do – while` quando você precisa executar o código pelo menos uma vez. O loop `do – while` é especialmente útil em situações nas quais precisamos executar uma ação ao menos uma vez, antes de verificar qualquer condição. Por exemplo, em situações como:

- Validação de dados de entrada.
- Repetição de menus interativos.
- Processamento de informações com verificação posterior.

Citaremos a seguir exemplos com loop `do – while` em C:

Um exemplo clássico é a validação de senha. Imagine que você precisa criar um sistema no qual o usuário deve digitar sua senha. O sistema deve permitir que ele digite a senha pelo menos uma vez e, se estiver incorreta, exibir uma nova solicitação. Nesse caso, o `do – while` é a escolha ideal.

```
#include <stdio.h>

int main() {

    char senha[20];

    do{
        printf("Digite sua senha: ");
        scanf("%s", senha);
    } while(strcmp(senha, "1234") != 0);

    printf("Acesso concedido!\n");

    return 0;
}
```

Observe a seguir a explicação simplificada do código:

- Pede ao usuário que digite uma senha.
- Compara com a senha correta ("1234").
- Enquanto a senha estiver incorreta, repete a solicitação.
- Assim que a senha estiver correta, exibe a mensagem de acesso liberado.

Por que não usar `while` aqui? Se você usasse um `while`, teria que pedir a senha antes do loop, o que tornaria o código menos intuitivo e mais propenso a erros.

O `do – while` permite lidar com esse tipo de situação de forma mais elegante, clara e segura.

Agora temos um menu de opções simples.

```
#include <stdio.h>

int main(){

    int opcao;

    do{
        printf("1 - Iniciar\n2 - Ajuda\n0 - Sair\nDigite sua opção: ");
        scanf("%d", &opcao);
    } while(opcao != 0);

    return 0;
}
```

Observe a seguir a explicação simplificada do código:

- Mostra um menu simples que se repete até que o usuário digite 0. O menu é exibido pelo menos uma vez.



## Lembrete

Use do – while quando:

- O bloco deve ser executado pelo menos uma vez.
- A verificação da condição deve ocorrer após a execução.
- É necessário interagir com o usuário antes de validar algo.

## Exemplos de aplicação

**Exemplo 1.** Imprimir os múltiplos de um número até menor que 1000, com do – while

Um supermercado deseja identificar todos os produtos que receberão desconto múltiplo. Para isso, é definido um número  $n$  (por exemplo, 10), e todos os produtos cujo número de cadastro seja múltiplo de  $n$ , no intervalo de 2 até 999, receberão o desconto. Regras:

- O número  $n$  deve ser digitado pelo usuário e, inclusive, estar entre 2 e 10.
- O programa deve identificar e exibir na tela todos os múltiplos de  $n$  entre 2 e 999.
- Caso o número digitado esteja fora do intervalo, o programa deve repetir a solicitação até receber um número válido.

- Escreva um programa em linguagem C que leia um número inteiro  $n$  do usuário, validando se ele está entre 2 e 10. Utilize a estrutura de repetição do – while para imprimir todos os múltiplos de  $n$  entre 2 e 999.

### Resolução

O código a seguir apresenta a resolução do problema para imprimir os múltiplos de um número  $n$  até menor que 1000.

```
#include <stdio.h>

int main() {
    int n, i;

    // Validação com do-while
    do{
        printf("Digite um número entre 2 e 10: ");
        scanf("%d", &n);

        if(n < 2 || n > 10) {
            printf("Valor inválido. ");
        }
    }while(n < 2 || n > 10);

    printf("Múltiplos de %d entre 2 e 999:\n", n);

    i = 2;
    do{
        if(i % n == 0) {
            printf("%d\n", i);
        }
        i++;
    }while(i < 1000);

    return 0;
}
```

- **Declaração de variáveis**

- $n$ : armazena o número digitado pelo usuário.
- $i$ : contador para percorrer os números de 2 até 999.

- **Validação da entrada com do – while**

- O bloco de entrada é executado pelo menos uma vez.
- O programa solicita que o usuário digite um número entre 2 e 10. Se o número estiver fora do intervalo, exibe a mensagem de erro e repete.

- **Laço do – while para exibir múltiplos**

- Inicializa i com 2.
- Enquanto ( $i < 1000$ ), o programa verifica se i é múltiplo de n.
  - Se a condição ( $i \% n == 0$ ) for verdadeira, imprime o valor de i, pois o valor de i será múltiplo de n.
  - A condição  $\text{while}(i < 1000)$ ; no final do bloco  $\text{do}\{\}$  será verdadeira até o contador i atingir 999. Quando for 1000, a condição será falsa e o laço termina.

**Exemplo 2.** Verificar se números são pares ou ímpares até digitar -1, com do – while

Um sistema precisa analisar uma lista de números digitados pelo usuário e informar, para cada um deles, se é par ou ímpar. A entrada dos dados deve continuar até que o número -1 seja digitado, o que indica o fim da lista. Regras:

- O programa deve ler vários números inteiros digitados pelo usuário.
- Para cada número, o programa deve informar se ele é par ou ímpar.
- O programa deve encerrar quando o usuário digitar -1.

Escreva um programa em linguagem C que: leia números inteiros digitados pelo usuário. Para cada número, informe se ele é par ou ímpar. O programa deve repetir esse processo até que o número -1 seja digitado. Quando isso ocorrer, o programa deve encerrar com uma mensagem de finalização.

## Resolução

Verificar se números são pares ou ímpares até digitar -1, com do – while .

Apresenta-se a seguir uma versão com uso do – while .

Comparação resumida entre as versões while e do – while: a principal diferença entre as duas versões ocorre no momento da verificação da condição.

- **Com while:** a condição é verificada antes de executar o bloco.
  - Se a condição for falsa desde o início, o bloco não será executado. Para evitar esse problema na versão com while, foi necessário fazer um laço infinito  $\text{while}(1)$  e uso do break para finalizar o laço.
- **Com do – while:** a condição é verificada depois de executar o bloco.



O código sempre é executado ao menos uma vez, mesmo que a condição já comece falsa. Não foi necessário o uso de break.

```
#include <stdio.h>
#include <locale.h> // Necessário para usar setlocale

int main() {
    setlocale(LC_ALL, "Portuguese"); // Ativa acentuação e formatação local

    int numero;

    printf("Digite números inteiros para saber se são pares ou ímpares.\n");
    printf("Para encerrar, digite -1.\n");

    do{
        printf("\nDigite um número: ");
        scanf("%d", &numero);

        if(numero == -1) {
            printf("Programa encerrado.\n");
        }
        else if(numero % 2 == 0) {
            printf("%d é par.\n", numero);
        }
        else{
            printf("%d é ímpar.\n", numero);
        }

    } while(numero != -1);

    return 0;
}
```

Explicação resumida:

- **Declaração da variável:** numero: armazena o valor digitado.
- **Laço do – while:** o corpo do laço será executado pelo menos uma vez.
- **Verificação de condição de parada:**
  - Se numero for diferente de -1, o programa verifica se é par ou ímpar usando o operador módulo (%).
  - if(numero % 2 == 0).
  - **Verdadeiro:** número par; caso contrário, número ímpar.
- Quando o usuário digitar -1, o laço termina.

## 4.3 Laço de repetição: for

O laço (loop) for é uma estrutura de repetição muito usada na programação, sobretudo quando sabemos exatamente quantas vezes queremos que uma ação aconteça.

Antes de mergulharmos nos detalhes, vamos entender o que é um laço (loop) for. Imagine que você precisa escrever a frase "Amo programar!" 100 vezes. Fazer isso manualmente seria trabalhoso, certo? É aí que os laços entram em cena! Eles nos permitem repetir um trecho de código várias vezes sem precisar escrevê-lo repetidamente.

Além disso, podemos pensar em uma situação comum no trabalho: você precisa gerar 50 crachás com o nome e o número de identificação dos funcionários. Em vez de escrever um por um, você pode usar um for para repetir automaticamente essa tarefa no seu sistema.

Sintaxe do laço for: vamos analisar a estrutura básica de um laço for em C:

```
for(inicialização; condição; atualização){  
    // bloco de código a ser repetido  
}
```

Pode parecer complicado à primeira vista, então vamos explicar cada parte:

- **Inicialização:** define uma variável de controle (geralmente chamada de contador) e executa essa etapa apenas uma vez, no início do loop.
- **Condição:** é verificada antes de cada repetição (ou iteração). Se for verdadeira, o bloco de código dentro do loop é executado. Se for falsa, o loop é encerrado.
- **Atualização:** modifica o valor do contador ao final de cada iteração. É normalmente usada para avançar para o próximo passo.

### Fluxo de controle de um loop for

Observe a seguir o passo a passo para entender como o for funciona na prática:

- A inicialização é executada uma única vez, no início do loop.
- A condição é verificada:
  - Se for verdadeira, o bloco de código é executado.
  - Se for falsa, o loop termina.

- Após a execução do bloco de código, a atualização é realizada.
- O ciclo volta à etapa 2 (verificação da condição), repetindo o processo até que a condição se torne falsa.

Citemos um exemplo: enviar e-mails automáticos para 100 clientes.

Imagine que você quer enviar o mesmo e-mail para 100 clientes cadastrados. Você pode usar um `for` para automatizar isso, conforme podemos ver no código a seguir:

```
#include <stdio.h>

int main() {
    int i; //declaração da variável de controle

    for(i = 1; i <= 100; i++){
        printf("Enviando e-mail para o cliente %d\n", i);
    }

    return 0;
}
```

Assim, será impressa 100 vezes a mensagem "Enviando e-mail para o cliente %d\n", uma em cada linha. Analisando sua estrutura, temos:

- **Início:** `int i = 1;` começamos com o cliente número 1.
- **Condição:** `(i <= 100);` enquanto `i` for menor ou igual a 100, o loop continua.
- **Atualização:** `i + +;` a cada volta, aumentamos o número em 1.

Revendo o passo a passo para o exemplo:

- O valor inicial da variável (`i = 1`) é definido.
- A condição (`i <= 100`) é verificada:
  - Se for verdadeira, o código dentro do bloco é executado.
  - Se for falsa, o loop termina.
- Após executar o bloco, o valor de `i` é atualizado (no caso, incrementado em 1).
- O processo se repete até a condição se tornar falsa, neste caso, quando `i = 101`.



## Observação

Lembre-se de declarar a variável de controle antes de utilizá-la no loop for. Isso evita erros ao usar compiladores antigos.

Em compiladores mais modernos da linguagem C, é possível declarar a variável diretamente dentro do próprio for, como em:

```
for(int i = 1; i <= 100; i++){  
    printf("Enviando e-mail para o cliente %d\n", i);  
}
```

A razão pela qual compiladores antigos de C não aceitam a declaração de variáveis dentro do cabeçalho do for está ligada à evolução do padrão da linguagem C.

Antes do C99 (C90 e anteriores), nos padrões mais antigos da linguagem C (como o ANSI C89/C90), todas as variáveis precisavam ser declaradas no início de um bloco de código. Isso significa que você não podia declarar uma variável "no meio do caminho", como dentro do cabeçalho de um for.

A partir do C99, a linguagem passou a permitir a declaração de variáveis em qualquer parte do bloco, inclusive dentro do for. Isso trouxe mais clareza e modularidade, possibilitando, por exemplo, que variáveis usadas apenas dentro de um loop fiquem restritas ao seu escopo.

Por que isso importa?

- **Compatibilidade:** muitos compiladores antigos ou configurados para seguir o padrão C90 ainda exigem que variáveis sejam declaradas no início.
- **Boa prática:** mesmo com compiladores modernos, declarar fora do for pode ser útil para manter compatibilidade com versões mais antigas ou com padrões mais restritos (como em sistemas embarcados).

Para evitar esse problema e garantir que seu código funcione em qualquer ambiente, **neste livro-texto adotaremos a prática de declarar todas as variáveis no início do programa**, antes do uso do for e de outros comandos e funções da linguagem C.

Agora citemos um exemplo para gerar relatórios de 12 meses:

```
#include <stdio.h>

int main() {
    int mes; //declaração da variável de controle

    for(mes = 1; mes <= 12; mes++) {
        printf("Gerando relatório do mês %d\n", mes);
    }

    return 0;
}
```

Assim, será impressa 12 vezes a mensagem do código para representar os 12 meses do ano, uma em cada linha.

Explicação passo a passo:

```
int mes;
```

Aqui estamos declarando a variável do tipo inteiro. Essa variável será usada como contador no loop for.

```
for(mes = 1; mes <= 12; mes++)
```

Esse é o loop for, dividido em três partes:

- **Inicialização:** mes = 1; o loop começa com o mês 1, pois é o valor que foi atribuído inicialmente à variável mes.
- **Condição:** mes <= 12; o loop continua enquanto mes for menor ou igual a 12.
- **Atualização:** mes + +; a cada repetição, o valor de mes é incrementado em 1.

Ou seja, o código será executado 12 vezes, uma para cada mês do ano.

```
printf("Gerando relatório do mês %d\n", mes);
```

Essa linha exibe a mensagem "Gerando relatório do mês %d\n", onde %d é substituído pelo número do mês atual e finalizado por uma quebra de linha indicado pelo símbolo de escape \n.

A função printf está usando o formato %d para exibir o valor da variável mes.

Mais um exemplo: listar produtos em estoque numerados de 1 a 100.

```
#include <stdio.h>

int main(){
    int i;

    for(i = 1; i <= 100; i++){
        printf("Produto número %d\n", i);
    }

    return 0;
}
```

Será impressa no código a mensagem "Produto número %d\n" 100 vezes, uma em cada linha. O entendimento do código segue o que já foi explicado anteriormente.

O for é útil sempre que você precisar fazer algo várias vezes de forma organizada, economizando tempo e evitando erros repetitivos.

Serão acentuados a seguir exercícios com for na linguagem de programação C.

Os exercícios deste tópico foram propositalmente repetidos com as estruturas de repetição while e for, permitindo ao estudante comparar o uso de cada uma em situações equivalentes. Essa abordagem reforça a lógica de programação.

Ao resolver o mesmo exercício com while anteriormente e agora com for, espera-se que o aluno perceba as diferenças, por exemplo:

- Que o for é mais direto em casos com número fixo de repetições.
- Que o while é mais indicado quando não se sabe previamente o número de vezes que o laço vai se repetir.

Essa experiência ajuda a escolher melhor qual estrutura usar em situações reais.

## Exemplos de aplicação

**Exemplo 1.** Bonificação de produtos de 1 até 10 (multiplicada por 10) com for

Um supermercado realizou uma promoção com 10 produtos cadastrados. Cada produto recebeu um valor de bonificação correspondente ao seu número de ordem multiplicado por 10. Por exemplo: produto 1 x 10 = R\$ 10.

Crie um programa em linguagem C que:

- Calcule o valor da bonificação para cada um dos 10 produtos, multiplicando o número do produto por 10.
- Exiba na tela uma linha para cada produto, informando o valor recebido e o valor total pago em bonificações:
  - Exemplo para dois produtos:
    - "Produto 1 recebeu R\$ 10".
    - "Produto 2 recebeu R\$ 20".
    - "Valor total em bonificações R\$ 30".

### Resolução

A código a seguir apresenta a solução do programa bonificação de produtos de 1 até 10 (multiplicada por 10) com for.

```
#include <stdio.h>

int main() {
    int i, bonificacao, total = 0;

    for (i = 1; i <= 10; i++) {
        bonificacao = i * 10;
        printf("Produto %d recebeu R$ %d\n", i, bonificacao);
        total += bonificacao;
    }

    printf("Valor total em bonificações R$ %d\n", total);
    return 0;
}
```

- **Declaração de variáveis:**
  - i é o contador que representa o número do produto.
  - bonificacao armazena o valor da bonificação de cada produto.
  - total acumula o valor total pago em bonificações, inicia com 0.

- **Laço for**( $i = 1; i \leq 10; i++$ ): o laço será repetido 10 vezes.
- **Cálculo da bonificação**: em cada repetição,  $\text{bonificacao} = i * 10$ .
- Exibição da bonificação individual.
- **Acúmulo do total**:  $\text{total} += \text{bonificacao}$ .
- **Exibição do total**: após o laço, imprime o valor total acumulado.

**Exemplo 2.** Somar notas de 10 alunos e calcular a média da turma com for

Uma professora precisa lançar as notas de 10 alunos e, ao final, precisará saber a média geral da turma.

Crie um programa em C que:

- Peça ao usuário que digite a nota de cada aluno.
- Some todas as notas.
- Calcule e exiba a média final da sala.

### Resolução

O código a seguir apresenta a solução do programa para somar as notas de 10 alunos e calcular a média da turma com for.

```
#include <stdio.h>

int main() {
    int i;
    float nota, soma = 0, media;

    for (i = 1; i <= 10; i++) {
        printf("Digite a nota do aluno %d: ", i);
        scanf("%f", &nota);
        soma += nota; // soma = soma + nota
    }

    media = soma / 10;
    printf("Média da turma: %.2f\n", media);

    return 0;
}
```



- **Declaração de variáveis:**
  - A variável `i` é o contador de alunos (inicia em 1).
  - A variável `nota` armazena a nota digitada.
  - A variável `soma` acumula todas as notas.
  - A variável `media` armazenará o resultado final.
- **Laço `for(i = 1; i <= 10; i++)`:** solicita a nota dos 10 alunos.
- **Acumulação da soma:** a cada iteração, a nota é somada à variável `soma`.
- **Cálculo da média:** após o laço, a média é calculada.
- **Saída formatada:** a média da turma é exibida com duas casas decimais (`%0.2f`).

**Exemplo 3.** Imprimir os números inteiros ímpares de 1 até `n` com `for`

Um professor deseja gerar uma lista de números para um jogo de adivinhação com os alunos. O jogo deve conter apenas os números ímpares entre 1 e `n`, onde `n` é informado pelo próprio professor.

Crie um programa em C que solicite ao usuário um número `n` e imprima na tela todos os números ímpares de 1 até `n`.

### Resolução

O código a seguir apresenta a resolução do problema para imprimir os números inteiros ímpares de 1 até `n` com `for`.

```
#include <stdio.h>

int main() {
    int n, i;

    printf("Digite um número inteiro positivo: ");
    scanf("%d", &n);

    printf("Números ímpares de 1 até %d:\n", n);

    for(i = 1; i <= n; i++) {
        if(i % 2 != 0) {
            printf("%d\n", i);
        }
    }

    return 0;
}
```

- **Declaração de variáveis:**
  - **n**: valor digitado pelo usuário.
  - **i**: contador que percorre de 1 até **n**.
- **Entrada de dados:** solicita que o usuário digite um número e armazena na variável **n**.
- **Laço for(**i = 1**; **i <= n**; **i++**):**
  - Inicializa **i** com 1; verifica se **i** é menor ou igual a **n**; incrementa **i** em 1 a cada repetição. Percorre os números de 1 até **n**.
- Dentro do laço, verifica se **i** é ímpar com `if(i % 2 != 0)`.
  - Se a condição for verdadeira, ou seja, se o valor de **i** for ímpar, imprime o valor de **i**.

**Exemplo 4.** Imprimir os múltiplos de um número até 100 com for

Um supermercado deseja identificar todos os produtos que receberão desconto múltiplo. Para isso, é definido um número **n** (por exemplo, 5), e todos os produtos cujo número de cadastro seja múltiplo de **n**, no intervalo de 2 até 100, receberão o desconto. Regras:

- O número **n** deve ser digitado pelo usuário e estar entre 2 e 10, inclusive.
- O programa deve identificar e exibir na tela todos os múltiplos de **n** entre 2 e 100.
- Caso o número digitado esteja fora do intervalo, o programa deve repetir a solicitação até receber um número válido.

Escreva um programa em linguagem C que:

- Leia um número inteiro **n** do usuário, validando se ele está entre 2 e 10.
- Utilize a estrutura de repetição for para imprimir todos os múltiplos de **n** entre 2 e 100, inclusive.

### Resolução

O código a seguir apresenta a resolução do problema para imprimir os múltiplos de um número **n** até 100 com validação de valor de entrada do usuário, utilizando estrutura de repetição for.

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

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

    int n = 0, i;

    //Primeiro laço: garante que n seja maior ou igual a 2
    for (; n < 2; ) {
        printf("Digite um número maior ou igual a 2: ");
        scanf("%d", &n);
    }

    //Segundo laço: garante que n seja menor ou igual a 10
    for (; n > 10; ) {
        printf("Digite um número menor ou igual a 10: ");
        scanf("%d", &n);
    }

    //Exibe os múltiplos de n entre 2 e 100
    printf("\nMúltiplos de %d entre 2 e 100:\n", n);
    for (i = 2; i <= 100; i++) {
        if (i % n == 0) {
            printf("%d\n", i);
        }
    }

    return 0;
}
```

- **Declaração das variáveis:**

- **n:** guarda o número digitado pelo usuário (valor entre 2 e 10).
- **i:** contador usado para percorrer os números de 2 até 100.

- **Validação da entrada (dois laços for):**

- O primeiro for garante que o usuário digite um número maior ou igual a 2.
  - Enquanto  $n < 2$ , o programa pede outro número.
- O segundo for garante que o número seja menor ou igual a 10.
  - Enquanto  $n > 10$ , o programa pede novamente outro número.

- Com esses dois laços separados, o número final estará sempre no intervalo correto (2 a 10).
- A condição  $i \% n == 0$  verifica se  $i$  é múltiplo de  $n$ . Se verdadeira, imprime o valor de  $i$ .



### Lembrete

O programa garante que o usuário digite um número entre 2 e 10. Em seguida, mostra todos os múltiplos desse número entre 2 e 100. Usa somente laços for com lógica simples e clara.

Nesse exemplo, a validação da entrada foi feita com dois laços for separados, cada um cuidando de uma parte da verificação: primeiro, se o número é maior ou igual a 2; depois, se é menor ou igual a 10.

Essa abordagem evita o uso de operadores lógicos complexos ( $\&\&$ ) e do comando break, tornando o código mais claro e simples.

Embora o while seja tradicionalmente utilizado para validações, o uso do for dessa forma também é válido e cumpre bem seu papel didático, especialmente quando se busca simplificar a lógica para facilitar o aprendizado.

### Exemplo 5. Verificar se números são pares ou ímpares até digitar -1 com for

Um sistema precisa analisar uma lista de números digitados pelo usuário e informar, para cada um deles, se é par ou ímpar. A entrada dos dados deve continuar até que o número -1 seja digitado, o que indica o fim da lista.

Regras:

- O programa deve ler vários números inteiros digitados pelo usuário.
- Para cada número, o programa deve informar se ele é par ou ímpar.
- O programa deve encerrar quando o usuário digitar -1.

Escreva um programa em linguagem C que:

- Leia números inteiros digitados pelo usuário.
- Para cada número, informe se ele é par ou ímpar.
- O programa deve repetir esse processo até que o número -1 seja digitado. Quando isso ocorrer, o programa deve encerrar com uma mensagem de finalização.

### Resolução

O código a seguir apresenta um programa que verifica se números são pares ou ímpares até digitar -1 com for.

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

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

    int numero;

    printf("Digite números inteiros > 0 .\n");
    printf("Para encerrar, digite -1.\n");

    //Atribui um valor diferente de -1 para iniciar o laço
    for(numero = 0; numero != -1; ) {
        printf("\nDigite um número: ");
        scanf("%d", &numero);

        if(numero != -1) {
            if(numero % 2 == 0) {
                printf("%d é par.\n", numero);
            } else {
                printf("%d é ímpar.\n", numero);
            }
        } else {
            printf("Programa encerrado.\n");
        }
    }

    return 0;
}
```

Explicação do funcionamento:

- O laço for usa `numero != -1` como condição de permanência.
- A entrada de dados (`scanf`) acontece dentro do laço, e o valor digitado já afeta a condição.
- O teste `if(numero != -1)` garante que não será exibido "par/ímpar" após digitar -1, apenas a mensagem de encerramento.
- **Verificação de paridade:** `if(numero % 2 == 0)` é par; senão, é ímpar.
- **Continuação:** o laço se repete até o usuário digitar -1.

## 4.4 Laços de repetição aninhados: for, while e do – while

**Laços aninhados** (encadeados), ou seja, um laço dentro de outro, são úteis em sistemas que geram relatórios, cálculos repetitivos ou comparações cruzadas. Além disso, os laços aninhados são essenciais para trabalhar com matrizes.

Um exemplo clássico de laço encadeado é a geração de tabuadas de 1 até 10, todas de uma vez. Esse tipo de código pode ser feito com (for, while ou do – while).

Observe o primeiro exemplo: Imprimir tabuadas de 1 a 10 usando for com laço encadeado com while.

```
#include <stdio.h>

int main() {
    int i = 1;
    while (i <= 10) {
        printf("Tabuada do %d\n", i);
        int j = 0;
        while (j <= 10) {
            printf("%d x %d = %d\n", i, j, i * j);
            j++;
        }
        printf("\n");
        i++;
    }
    return 0;
}
```

Observe a seguir a explicação simplificada:

- As variáveis i e j controlam os dois níveis do laço.
- Enquanto i <= 10, o programa imprime a tabuada do número i.
- Em cada iteração do interno (j), imprime i x j.
- Ao final de cada bloco interno, j volta a 0 e i é incrementado.

Agora vamos imprimir tabuadas de 1 a 10 usando for com laço encadeado com for.

```
#include <stdio.h>

int main() {
    int i, j;

    for(i = 1; i <= 10; i++) {
        printf("Tabuada do %d\n", i);

        for(j = 0; j <= 10; j++) {
            printf("%d x %d = %d\n", i, j, i * j);
        }

        printf("\n");
    }

    return 0;
}
```

- **Declaração de variáveis:**

- As variáveis *i* e *j* controlam os dois níveis dos laços for.
- *i*: controla o número da tabuada (de 1 a 10).
- *j*: controla os multiplicadores (de 0 a 10).
- O laço externo `for(i = 1; i <= 10; i++)` percorre os números de 1 a 10 e define qual tabuada será exibida.
- Para cada valor de *i*, o laço interno `for(j = 0; j <= 10; j++)` imprime os produtos de *i* x *j*.
  - Cada linha exibe a multiplicação no formato: *i* x *j* = resultado.
- Ao final de cada tabuada, é impresso um espaço em branco (`\n`) para separar visualmente as tabuadas.

No próximo exemplo, temos um triângulo de asteriscos com *n* linhas. Será impresso um triângulo de asteriscos (\*) em C, e o usuário define o número de linhas com laço while.

Entrada esperada: se o usuário digitar  $n = 5$ , a saída será:

```
#include <stdio.h>

int main() {
    int n, i = 1;

    printf("Digite a quantidade de linhas: ");
    scanf("%d", &n);

    while(i <= n) {
        int j = 1;
        while(j <= i) {
            printf("* ");
            j++;
        }
        printf("\n");
        i++;
    }

    return 0;
}
```

Entrada esperada: se o usuário digitar  $n = 5$ , a saída será:

```
*
**
***
****
*****
```

Explicação passo a passo:

```
int n, i = 1;
```

- **Declara as variáveis:**  $n$  para armazenar o número de linhas digitado pelo usuário e a variável  $i$ , iniciada em 1, que será o contador de linhas.

```
printf("Digite o número de linhas: ");
scanf("%d", &n);
```



- Solicita e lê o valor da variável `n`.

```
while(i <= n) {  
    int j = 1;
```

- O laço externo `while(i <= n)` controla cada linha.
- Para cada linha, inicia a variável `j` com 1, e `j` será o contador de colunas, ou seja, o número de asteriscos (\*).

```
while(j <= i) {  
    printf("* ");  
    j++;  
}
```

- O laço interno imprime \* e um espaço até que `j > i`.
- Ou seja, na linha 1 imprime um asterisco, na linha 2 imprime dois asteriscos, e assim por diante.

```
printf("\n");  
i++;  
}
```

Após imprimir todos os asteriscos de uma linha, imprime uma quebra de linha (`\n`) e passa para a próxima linha (`i++`) incrementando o valor de `i`.



### Lembrete

O laço externo (`i`) determina quantas linhas serão impressas.

O laço interno (`j`) imprime os asteriscos por linha.

Cada linha tem a quantidade de asteriscos igual ao número da linha.

Saída na tela: exemplo para `n = 3`.

```
Digite a quantidade de linhas: 3  
*  
* *  
* * *
```

Figura 24

Agora temos um triângulo de asteriscos com n linhas. Será impresso um triângulo de asteriscos (\*) em C, e o usuário define o número de linhas com laço for.

```
#include <stdio.h>

int main() {
    int n, i, j;

    printf("Digite a quantidade de linhas: ");
    scanf("%d", &n);

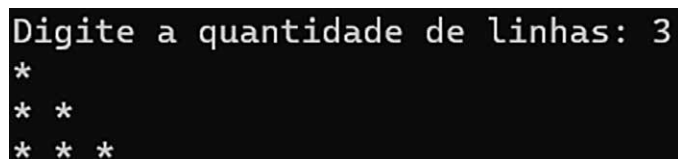
    for(i = 1; i <= n; i++) {
        for(j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }

    return 0;
}
```

Explicação passo a passo:

- **Declaração das variáveis:** n, i e j, do tipo inteiro. O usuário digita n.
- **Laço externo for(i = 1; i <= n; i ++):**
  - Controla a quantidade de linhas que serão impressas 1 até n.
- **Laço interno for(j = 1; j <= i; j ++):**
  - Em cada linha, imprime a quantidade de \* e um espaço correspondente ao número da linha atual.
- **Quebra de linha:** executa printf("\n"); para ir para a próxima linha.

Saída na tela: se o usuário digitar n = 3, a saída será:



```
Digite a quantidade de linhas: 3
*
* *
* * *
```

Figura 25

No exemplo a seguir, temos um triângulo de números em sequência de 1 até 3. Será impresso um triângulo de números em sequência em C.

```
#include <stdio.h>

int main() {
    int i, j;

    // Laço externo controla as linhas
    for(i = 1; i <= 3; i++) {
        // Laço interno imprime o valor de j de 1 até i
        for(j = 1; j <= i; j++) {
            printf("%d ", j);
        }
        printf("\n");
    }

    return 0;
}
```

O código anterior apresenta um programa em C que utiliza dois laços de repetição for aninhados (um dentro do outro). O objetivo desse programa é imprimir uma sequência de números inteiros em formato crescente e organizados em linhas. Esse padrão é conhecido como formação de pirâmide crescente de números, no qual a quantidade de números por linha aumenta conforme o valor de i.

Explicação passo a passo:

- **Declaração das variáveis:** i e j, do tipo inteiro.
- O laço externo for(i = 1; i <= 3; i++) controla o número de linhas que serão impressas (no caso, 3 linhas).
- O laço interno for(j = 1; j <= i; j++) é responsável por imprimir os números de 1 até o valor de i em cada linha.
  - Na 1ª repetição (i = 1), imprime: 1.
  - Na 2ª repetição (i = 2), imprime: 1 2.
  - Na 3ª repetição (i = 3), imprime: 1 2 3.
- printf("\n") para ir para a próxima linha após cada linha impressa.

Observe a seguir um laço for aninhado: tabela de multiplicação (3x3) em C.

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

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

    int i, j;
    printf("\tTabela de Multiplicação:\n\n");

    for(i = 1; i <= 3; i++){
        for(j = 1; j <= 3; j++){
            printf("%d x %d = %d\t", i, j, i * j);
        }
        printf("\n\n");
    }

    printf("\n");

    return 0;
}
```

O código anterior apresenta um programa em C que utiliza dois laços de repetição for aninhados (um dentro do outro). O objetivo desse programa é imprimir uma sequência de multiplicação organizados em linhas e colunas.

Explicação:

- **Declaração das variáveis:** i e j, do tipo inteiro.
- O laço externo for(i = 1; i <= 3; i++) itera de 1 até 3 (inclusive), representando as linhas da tabela de multiplicação.
- O laço interno for(j = 1; j <= 3; j++) também itera de 1 até 3, representando as colunas.
- Dentro do laço interno, printf("%d x %d = %d\t", i, j, i \* j); imprime o resultado da multiplicação para cada combinação de i e j, seguido de um caractere de tabulação (\t) para formatar a saída.
- Após a conclusão do laço interno para uma determinada linha, insere uma nova linha (\n), printf("\n\n") e um separador.

Saída na tela:

Tabela de Multiplicação:		
1 x 1 = 1	1 x 2 = 2	1 x 3 = 3
2 x 1 = 2	2 x 2 = 4	2 x 3 = 6
3 x 1 = 3	3 x 2 = 6	3 x 3 = 9

Figura 26

No exemplo a seguir, temos um laço while aninhado: simulação de relógio digital (horas e minutos) em C.

O código a seguir apresenta um programa em C que utiliza dois laços de repetição while aninhados (um dentro do outro). O objetivo desse programa é imprimir uma simulação de horas e os minutos.

Explicação:

- **Declaração das variáveis:** a variável hora é inicializada com 0. O laço externo while(hora < 24) continua enquanto a variável hora for menor que 24.
- Dentro do laço externo, a variável minuto é inicializada com 0. O laço interno while(minuto < 60) continua enquanto a variável minuto for menor que 60.
- `printf("%02d:%02d\n", hora, minuto);` imprime a hora e o minuto formatados com dois dígitos (usando %02d para adicionar um zero à esquerda se o número for menor que 10), seguido de uma nova linha.
- Após cada iteração do laço interno, `minuto++`; incrementa o minuto.
- Após a conclusão do laço interno (quando os minutos chegam a 60) `hora++`; incrementa a hora.

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

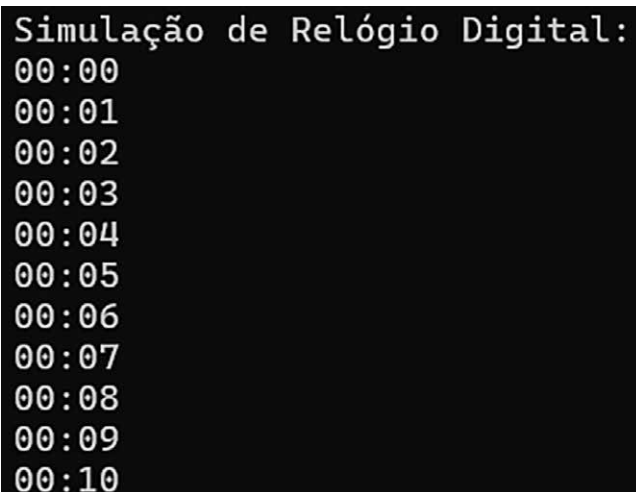
int main() {
    setlocale(LC_ALL, "Portuguese");

    printf("Simulação de Relógio Digital:\n");

    int hora = 0;
    while(hora < 24){
        int minuto = 0;
        while(minuto < 60){
            printf("%02d:%02d\n", hora, minuto);
            minuto++;
        }
        hora++;
    }
    printf("\n");

    return 0;
}
```

Saída na tela: nesse exemplo mostramos apenas um recorte da saída, pois ela irá mostrar uma lista de minuto a minuto, desde 00:00 até 23:59 um em cada linha.



```
Simulação de Relógio Digital:
00:00
00:01
00:02
00:03
00:04
00:05
00:06
00:07
00:08
00:09
00:10
```

Figura 27

Agora temos um laço do - while aninhado: validação de coordenadas em um grid 2x2 em C.

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

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

    printf("Validação de Coordenadas:\n");

    do{
        printf("Digite a linha (0 ou 1): ");
        scanf("%d", &linha);
        do{
            printf("Digite a coluna (0 ou 1): ");
            scanf("%d", &coluna);
        }while(coluna < 0 || coluna > 1);
    }while(linha < 0 || linha > 1);

    printf("Coordenadas válidas: (%d, %d)\n", linha, coluna);

    return 0;
}
```

O código anterior apresenta um programa em C que utiliza dois laços de repetição do – while aninhados (um dentro do outro). O objetivo desse programa é imprimir se dois valores digitados pelo usuário formam uma coordenada válida, para este exemplo, valores válidos 0 ou 1.

Explicação:

- O laço externo do{...}while(linha < 0 || linha > 1); garante que o bloco de código dentro seja executado pelo menos uma vez. Ele continua repetindo enquanto a linha inserida pelo usuário for menor que 0 ou maior que 1 (ou seja, inválida).
- Dentro do laço externo, do{...} while(coluna < 0 || coluna > 1); faz o mesmo para a coluna, garantindo que o usuário insira 0 ou 1.
- scanf("%d",&linha); e scanf("%d",&coluna); solicitam e leem a entrada do usuário para a linha e coluna.
- Somente quando o usuário insere valores válidos para linha (0 ou 1) e coluna (0 ou 1) ambos os laços do-while terminam, e a mensagem "Coordenadas válidas: (%d, %d)\n" é impressa.

Saída na tela: nessa simulação, o usuário digitou 1 para linha e 0 para coluna.

```
Validação de Coordenadas:  
Digite a linha (0 ou 1): 1  
Digite a coluna (0 ou 1): 0  
Coordenadas válidas: (1, 0)
```

Figura 28

## 4.5 Comparação entre laços de repetição: for, while e do – while

Já aprendemos a utilizar as três estruturas fundamentais de repetição em C: for, while e do – while. Cada uma delas tem características próprias, sendo mais indicada para determinadas situações de programação.

A seguir, apresentamos uma comparação detalhada, com foco na estrutura, no comportamento e nos casos de uso mais comuns, para que o programador possa escolher a forma mais adequada de laço conforme a necessidade do algoritmo.

**Quadro 6 – Características das estruturas de repetição (laços) em C**

	for	while	do – while
Uso mais comum	Quando se sabe quantas vezes repetir	Quando não se sabe quantas vezes	Quando se precisa repetir pelo menos uma vez
Estrutura	Condensada (início, condição, incremento)	Condição no início	Condição no final
Execução mínima	Pode não executar nenhuma vez	Pode não executar nenhuma vez	Sempre executa ao menos uma vez
Sintaxe	<pre>for(início; cond; inc){ ...comandos }</pre>	<pre>while(cond){ ...comandos }</pre>	<pre>do{ ...comandos }while (cond);</pre>
Leitura e clareza	Alta, ideal para laços contadores	Boa para laços com condição externa	Útil para menus, confirmações, validações
Exemplo típico de uso	Imprimir números de 1 a 10	Ler valores até digitar -1	Menu de opções com execução inicial garantida
Estilo	Mais compacto e direto	Mais flexível	Garantia de pelo menos uma execução



### Observação

Na sintaxe dos laços, usamos as siglas cond (condicao) condição e inc (incremento) para representar os elementos padrão do cabeçalho do laço.



Agora vamos analisar o quadro 6.

O laço `for` é mais utilizado quando sabemos de antemão quantas vezes o bloco de código precisa ser executado. Ele é ideal para laços contadores, quando existe um valor inicial, um valor final e um incremento claro. Por reunir inicialização, condição e incremento no cabeçalho, proporciona um código mais compacto e legível.

O laço `while` é mais indicado quando a repetição depende de uma condição dinâmica, que pode ou não ser verdadeira logo na primeira verificação. Ele é muito utilizado em validações de entrada, leituras condicionais, laços indefinidos ou quando o controle do fluxo depende de uma lógica externa.

O `do – while` garante que o bloco de código seja executado ao menos uma vez, antes de verificar a condição. É útil em situações como menus interativos, confirmações, validações com retorno obrigatório ou ações que devem ser apresentadas ao menos uma vez ao usuário.

Nos exemplos anteriores, o uso de `while` ou `for` não apresenta diferenças de desempenho, nem impacto significativo na quantidade de linhas ou na complexidade do código. No entanto, há diferenças de estilo e de clareza.

Ao usar `while`, o controle do laço (inicialização, condição e incremento) é realizado em partes separadas, o que pode deixar o código mais flexível, porém menos direto em estruturas contadoras.

Com `for`, o código, como no caso da tabuada ou da soma de valores de 1 a 100, torna-se mais conciso e organizado, pois essas situações representam repetições com número fixo de iterações, nas quais o `for` se encaixa perfeitamente.

Já o `do – while` se destaca em situações nas quais a primeira execução é obrigatória, só depois será verificada a necessidade de repetir o bloco de instruções.

### 4.6 Comando `break`

O `break` é um comando usado para interromper imediatamente a execução de estruturas de repetição (`for`, `while`, `do – while`) ou de seleção (`switch – case`). Também podemos utilizar o `break` dentro de `if`, caso o `if` esteja dentro de um (`for`, `while`, `do – while`) ou `switch`.

Dentro de estruturas de repetição (laços: `while`, `for`, `do – while`), ele é utilizado para sair antes da condição terminar. Ou seja, ele faz o programa sair do laço imediatamente, indo para a instrução que vem logo após o fim do laço.

Dentro da estrutura `switch`, a instrução `break` desempenha um papel fundamental: terminar a execução do bloco de código do `switch` assim que uma correspondência (`case`) for encontrada e o código correspondente for executado.

Quando um `break` é encontrado dentro de um bloco `case` (em um `switch`), o fluxo de controle do programa salta para a primeira instrução após o fechamento do bloco `switch`.

### Observação

O comando `break` é usado para encerrar imediatamente a execução de um laço de repetição (`for`, `while` ou `do – while`) ou de uma estrutura `switch`. Ao ser executado, ele faz com que o programa saia do bloco atual e continue sua execução a partir da próxima instrução após esse bloco.

No comando `switch`, o uso do `break` é vital para evitar que o programa execute, de forma contínua, os blocos de código de outros casos. Se o `break` for omitido, os comandos dos próximos `case` serão executados mesmo que suas condições não correspondam – um comportamento conhecido como “queda” (`fallthrough`). Embora essa característica possa ser útil em casos específicos, normalmente deve ser evitada para prevenir erros lógicos.

Em estruturas de repetição (`for`, `while` ou `do – while`), o `break` permite interromper o ciclo antecipadamente, ou seja, antes que a condição do laço se torne falsa. Isso é útil, por exemplo, ao encontrar um valor que satisfaça uma condição específica.

Vale destacar que o `break` afeta apenas o bloco de controle mais próximo onde está inserido. Em casos de comandos aninhados (como laços dentro de laços), ele encerrará apenas o nível mais interno.

Além disso, embora o `break` possa ser usado dentro de uma estrutura `if`, ele não pertence ao `if` em si. O `if` apenas determina se o `break` será executado; o impacto do comando continuará sendo sobre o laço ou `switch` ao qual ele está associado.

Exemplo: uso do comando `break` com laço `for` para imprimir uma sequência de números até determinada condição.

O código a seguir apresenta um programa em C que demonstra o uso do comando `break` dentro de um laço `for` e condicionado por uma estrutura `if`.

Explicação do programa:

- O programa inicializa uma variável `i` e utiliza um laço `for` para percorrer os valores de 0 a 9.
- A cada iteração, ele verifica se o valor de `i` é igual a 5.
- Quando essa condição for verdadeira, o programa exibe a mensagem “Encerrando o laço `for`”.

- Em seguida, executa o comando `break`, que interrompe imediatamente o laço `for`, mesmo que a condição `i < 10` ainda seja verdadeira.

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

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

    int i;

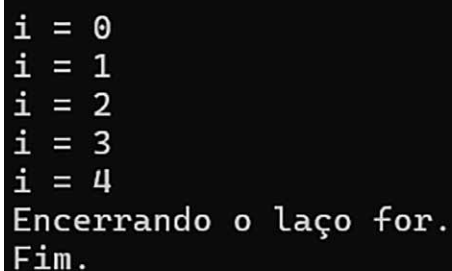
    for(i = 0; i < 10; i++){
        if(i == 5){
            printf("Encerrando o laço for.\n");
            break; // Sai do laço
        }
        printf("i = %d\n", i);
    }

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

Explicação resumida:

- A condição `if(i == 5)` controla quando o `break` será chamado.
- Quem sofre o efeito do `break` é o `for`, que será encerrado quando o valor de `i` for igual a 5.
- O programa não imprime `i = 5`, pois o `break` ocorre antes do comando `printf("i = %d\n", i);`, que imprime o valor de `i` na tela.

Saída na tela:



```
i = 0
i = 1
i = 2
i = 3
i = 4
Encerrando o laço for.
Fim.
```

Figura 29

## 4.7 Comando continue

É usado para pular a iteração atual e volta ao início do laço. Ou seja, o continue é um comando em C que interrompe apenas a iteração atual do laço (for, while ou do-while) e pula diretamente para a próxima repetição, sem executar o restante do código dentro do laço.

Simplificando: o continue não sai do laço como o break faz. Ele pula o que vem depois dentro do laço e volta para o início da próxima repetição.

Exemplificando: analogia com um jogo.

- Imagine que você está em um jogo passando por várias fases (1, 2, 3, 4, ..., n), sendo n a última fase do jogo.
- Se uma fase está bloqueada, você a pula usando o comando (continue), mas não sai do jogo (como faria com o comando break).

Vejamos um exemplo: pulando números pares em uma sequência de números de 1 até 10, com uso do comando continue e laço for.

O código a seguir apresenta um programa em C que demonstra o uso do comando continue dentro de um laço for e condicionado por uma estrutura if. O programa:

- Deve percorrer os números de 1 a 10.
- Pular os números pares usando continue.
- Imprimir apenas os ímpares e pular uma linha após cada impressão.

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

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

    int i;

    for(i = 1; i <= 10; i++) {
        if(i % 2 == 0) {
            continue; //pula os números pares
        }
        printf("%d\n", i);
    }

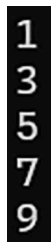
    return 0;
}
```

Explicação:

- `for(i = 1; i <= 10; i++)`: laço que repete de 1 até 10 (inclusive).
- A condição `if(i % 2 == 0)` controla quando o `continue` será chamado. Neste caso, se `i` é par, a condição é verdadeira e executa o `continue`.
  - O operador `%` retorna o resto da divisão. Quando o resultado é zero, o valor é par, como `2 % 2 == 0 → par`; `3 % 2 == 1 → ímpar`.
- `continue` faz o programa pular o restante do bloco do laço e ir direto para a próxima repetição do `for`.
- `printf("%d\n", i);`
  - Só será executado se `i` não for par.
  - Imprime o valor de `i` seguido de uma quebra de linha.

Resumindo: o `continue` faz o programa ignorar o `printf()` quando `i` é par.

Saída na tela:



1  
3  
5  
7  
9

Figura 30

### Exemplo de aplicação

Faça uma reflexão sobre o programa anterior e responda:

Qual será o valor final de `i` e por que não imprime o valor?

O valor final de `i` será 11. Isso ocorre porque o laço `for` foi definido para repetir enquanto `i <= 10`. Na última repetição, `i` vale 10. Depois disso, o `i++` é executado, incrementando `i` para 11.

Como 11 não satisfaz mais a condição do laço, ele é encerrado. Ou seja, quando `i` chega a 11, o laço já terminou, então o código que imprime o número não é mais executado. Mesmo sendo ímpar, o valor 11 nunca entra no corpo do laço.

## 4.8 Switch – case

A estrutura switch é uma forma de seleção múltipla utilizada na programação para permitir que o programa execute diferentes blocos de código, de acordo com o valor de uma variável. Essa estrutura é comumente conhecida como switch – case.

Conforme descrito por diversos autores, o switch – case é composto por uma sequência de rótulos, chamados case, além de um default (opcional), que representa o bloco a ser executado caso nenhum dos case seja correspondente. Em resumo, a estrutura oferece vários caminhos possíveis baseados em um único valor.

Na prática, o switch pode ser adotado como alternativa ao uso de múltiplas instruções if – else if. A instrução switch avalia o valor de uma expressão (geralmente uma variável) e o compara com os valores definidos em cada case. Quando ocorre uma correspondência, o bloco de código relacionado é executado.

```
switch(expressao) {  
    case constante 1:  
        //Bloco executado se expressao igual constante1  
        break; //interrompe imediatamente a execucao  
    case constante 2:  
        //Bloco executado se expressao igual constante2  
        break;  
    case constante n:  
        //Bloco executado se expressao igual constanteN  
        break;  
    default:  
        //Bloco executado se não entrar em nenhum case  
}
```

- **A estrutura básica switch(expressao):** avalia a expressão (geralmente uma variável do tipo int ou char), lembrando que as variáveis e palavras-chaves utilizadas em C não podem conter acentuação, por isso expressao dentro do switch está sem acento.
- **case constante:** valor que será comparado com a (expressao).
- **break;:** impede que as outras opções sejam executadas (evita o comportamento de queda (fallthrough)).
- **default:** opcional, executado se nenhum case dos anteriores for verdadeiro. Ou seja, se nenhum case for executado.

As vantagens do switch – case são:

- Mais legível que múltiplos if – else para comparações com valores fixos.

- Melhor desempenho em alguns compiladores.
- Ideal para menus, controle por códigos numéricos ou caracteres fixos.

Entre as diversas aplicações, podemos citar as mais comuns:

- Sistemas de menu de opções.
- Cálculos de preço baseado em código de produto.
- Classificações de letras ou números.

Deve-se utilizar switch quando:

- Você tem uma única variável para comparar com múltiplos valores exatos.
- As comparações são com valores constantes inteiros ou caracteres (int, char).
- Para deixar o código mais organizado e fácil de ler.

**Não** use switch – case quando:

- Você precisa fazer comparações mais complexas, como intervalos entre valores ( $x > 5 \ \&\& \ x < 10$ ), condições múltiplas ou verificar mais de uma variável ao mesmo tempo.
- Vai comparar valores não constantes ou do tipo float, double, strings ou booleanos, pois o switch não aceita esses tipos.

### Quadro 7 – Comparação do uso de switch – case ou if – else em C

Situação	Use switch – case	Use if – else
Comparar um valor exato (int/char)	Sim	Sim
Comparar intervalos de valores	Não	Sim
Verificar condições compostas	Não	Sim
Comparar tipos como float ou string	Não	Sim
Organizar muitas opções simples	Sim	Sim (menos legível)



## Lembrete

**switch:** estrutura de decisão com múltiplas alternativas.

**case:** define os valores testados dentro do switch.

**break:** encerra a execução do case atual ou de um laço.

**continue:** interrompe a iteração atual e pula para a próxima repetição.

Vejamos um exemplo: programa que cria um menu interativo de jogos utilizando a estrutura switch – case. O programa deve ler a escolha do jogador e tomar uma decisão.

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

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

    int opcao;

    printf("Ação:\n1. Atacar\n2. Defender\n3. Desistir\n");
    scanf("%d", &opcao);

    switch (opcao) {
        case 1:
            printf("Você atacou o inimigo!\n");
            break;
        case 2:
            printf("Você defendeu o ataque!\n");
            break;
        case 3:
            printf("Você desistiu da batalha.\n");
            break;
        default:
            printf("Opção inválida.\n");
    }

    return 0;
}
```



### Explicação resumida:

- **int opcao;**: declara a variável opcao do tipo inteiro. Ela será usada para armazenar na memória do computador a escolha do usuário.
- Inicia um comando switch, que serve para tomar decisões baseadas no valor da variável opcao.
- O programa exibe um menu com três opções para o jogador escolher. Após a escolha do usuário, o programa exibe uma mensagem.

No exemplo a seguir, temos uma calculadora com switch – case: programa que funciona como uma simples calculadora, permitindo ao usuário realizar uma das quatro operações básicas entre dois números.

### Explicação:

- **Variáveis:**

- int opcao;

- opcao: armazena a escolha do usuário.

- float num1, num2, resultado;

- num1, num2: os dois números inseridos pelo usuário.

- resultado: armazena o valor calculado.

- **scanf("%f %f",&num1,&num2);**

- Lê dois números reais que serão usados nas operações.

- Exibe um menu com quatro opções para o jogador.

- Inicia um comando switch, que serve para tomar decisões baseadas no valor da variável opcao.

- **switch(opcao)**: avalia o valor de opcao para decidir qual operação executar.

- Se opcao não for 1, 2, 3 ou 4, entrada inválida.

- Se opcao entre for 1, 2 ou 3, imprime o resultado.

- Se opcao for 4, valida se num2 é diferente de zero antes de dividir.

– Se diferente de zero, imprime o resultado; caso contrário, imprime mensagem de erro.

Resumo do programa:

- O programa exibe um menu com quatro opções de operação.
- Lê a opção escolhida pelo usuário.
- Lê dois números reais (float).
- Executa a operação correspondente (soma, subtração, multiplicação ou divisão).
- Exibe o resultado com duas casas decimais.
- Se a opção for inválida ou houver tentativa de divisão por zero, mostra uma mensagem de erro.

Saída na tela: exemplo para caso o usuário escolha operação 3 e valores 3 e 5.

```
-----CALCULADORA-----  
1 - Soma  
2 - Subtração  
3 - Multiplicação  
4 - Divisão  
Escolha uma operação: 3  
  
Digite dois números: 3 5  
  
Multiplicação: 15,00
```

Figura 31

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

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

    int opcao;
    float num1, num2, resultado;

    printf("-----CALCULADORA-----\n");
    printf("1 - Soma\n");
    printf("2 - Subtração\n");
    printf("3 - Multiplicação\n");
    printf("4 - Divisão\n");
    printf("Escolha uma operação: ");

    scanf("%d", &opcao);

    printf("\nDigite dois números: ");
    scanf("%f %f", &num1, &num2);

    switch(opcao){
        case 1:
            resultado = num1 + num2;
            printf("\nSoma: %.2f\n", resultado);
            break;
        case 2:
            resultado = num1 - num2;
            printf("\nSubtração: %.2f\n", resultado);
            break;
        case 3:
            resultado = num1 * num2;
            printf("\nMultiplicação: %.2f\n", resultado);
            break;
        case 4:
            if(num2 != 0){
                resultado = num1 / num2;
                printf("\nDivisão: %.2f\n", resultado);
            }
            else{
                printf("\nErro!\nNão pode dividir por zero!\n");
            }
            break;
        default:
            printf("\nOpção inválida!\n");
    }

    return 0;
}
```



### Saiba mais

Amplie seus conhecimentos consultando a obra a seguir:

SOUZA, M. A. F. *et al.* *Algoritmos e lógica de programação*: um texto introdutório para a engenharia. São Paulo: Cengage Learning, 2019.



### Resumo

Aprendemos dois conceitos muito importantes na programação em C: o controle de fluxo e as estruturas de repetição. Esses dois temas ajudam o programa a tomar decisões e a repetir ações automaticamente, tornando-o mais inteligente e funcional.

Vimos como o programa pode tomar decisões com base em condições. Para isso, usamos as estruturas condicionais:

- **if**: executa um bloco de código se a condição for verdadeira.
- **if...else**: escolhe entre dois caminhos, dependendo se a condição é verdadeira ou falsa.
- **if...else if...else**: permite testar várias condições diferentes.

Essas estruturas funcionam junto com operadores relacionais (`==`, `!=`, `>`, `<`) e lógicos (`&&`, `||`, `!`) para comparar valores e montar condições. Com elas, conseguimos criar programas que verificam se uma pessoa pode votar, se uma nota é suficiente para aprovação, se um número é par ou ímpar, entre outros exemplos do dia a dia.

Em seguida, aprendemos a repetir blocos de código automaticamente usando os laços de repetição. As principais estruturas são:

- **while**: repete enquanto a condição for verdadeira.
- **do...while**: executa ao menos uma vez e depois continua se a condição for verdadeira.
- **for**: ideal para quando já sabemos quantas vezes queremos repetir algo.

Esses laços são muito úteis para tarefas repetitivas, como imprimir números em sequência, ler várias entradas do usuário ou realizar cálculos acumulativos. Também aprendemos os comandos `break` (para interromper o laço) e `continue` (para pular para a próxima repetição).

Com o controle de decisões e repetições, agora conseguimos criar programas mais completos e dinâmicos, capazes de reagir a diferentes situações e realizar tarefas repetidas.



## Exercícios

**Questão 1.** Avalie as afirmativas a seguir sobre os laços na linguagem C.

I – O laço "for" é idealmente utilizado em situações nas quais o número de iterações é desconhecido. Ele não pode ser utilizado em outras situações.

II – Adotamos o comando "continue" para interromper a execução de uma iteração de um laço, voltando para a checagem da sua condição e, posteriormente, passando para a próxima iteração ou saindo do laço, de acordo com a lógica do programa.

III – Quando utilizamos apenas o comando "while" para fazer um laço, sabemos que o bloco interno ao comando "while" será executado ao menos uma vez, independentemente da sua condição associada. Para checar a condição sempre e impedir esse comportamento, devemos utilizar o comando "do" seguido do comando "while".

IV – O comando "break" termina abruptamente a execução do programa, abortando toda a sua execução.

É correto apenas o que se afirma em:

A) I.

B) II.

C) III.

D) II e IV.

E) I e III.

Resposta correta: alternativa B.

### Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: a afirmativa deveria referir-se ao laço "while", e não ao laço "for".

### II – Afirmativa correta.

Justificativa: a afirmativa mostra como o comando "continue", ainda que similar ao comando "break", trabalha de forma um pouco diferente: ele não interrompe totalmente a execução do laço (muito menos do programa), mas apenas avança para a checagem de condições e para a próxima iteração.

### III – Afirmativa incorreta.

Justificativa: a afirmativa apresenta a descrição invertida: ela refere-se ao par de comandos "do" e "while", e não somente ao comando "while".

### IV – Afirmativa incorreta.

Justificativa: o comando "break" não termina completamente a execução do programa: ele termina apenas a execução do laço no qual está inserido. Se existirem outros comandos após o laço, o programa poderá continuar a sua execução.

**Questão 2.** Considere o programa a seguir, escrito na linguagem C.

```
#include <stdio.h>
int main()
{
    int var1 = 0;
    int var2 = 11;
    var1 = 3;
    while(var2>0)
    {
        var2 = var2 - var1;
    }
    printf("Resultado: %d e %d \n", var1, var2);
    return 0;
}
```

Qual deve ser a saída desse programa após a sua execução?

A) Resultado: 11 e 3.

B) Resultado: 3 e 10.

C) Resultado: 3 e 11.

D) Resultado: 3 e -1.

E) Resultado: -1 e 3.

Resposta correta: alternativa D.

## Análise da questão

Para resolver essa questão, podemos fazer um teste de mesa, observando o valor das variáveis "var1" e "var2" no final do laço "while" (ou seja, dentro do laço, após a linha "var2 = var2 - var1;" e imediatamente antes da verificação da condição  $\text{var2} > 0$ ). Lembre-se de que, a princípio, poderíamos avaliar as condições das variáveis "var1" e "var2" a qualquer momento ou até mesmo durante todo o programa. No caso, estamos focando no intervalo relevante, uma vez que, ao final desse intervalo, os valores de "var1" e "var2" serão mostrados na tela pela função "printf". Adicionalmente, vamos incluir os valores dessas variáveis antes da entrada no laço "while", uma vez que essa é a condição inicial do problema.

Veja a tabela a seguir.

**Tabela 7**

Iteração	Conteúdo de var1	Conteúdo de var2
1 (antes do laço "while" )	3	11
2	3	8
3	3	5
4	3	2
5	3	-1

Observe que, ao final da quinta iteração, a variável "var1" tem o valor 3 e a variável "var2" tem o valor -1. Note também que o valor da variável "var1" praticamente não muda ao longo da execução do programa, nem muda durante a execução do laço "while". Isso ocorre porque, após a sua declaração, o programa atribui o valor 3 a essa variável, e não ocorre mais nenhuma alteração do seu valor. O conteúdo da variável "var1" é utilizado (lido) em outros pontos do programa, mas não é alterado.

A variável "var2" é alterada pelo laço "while", de acordo com a linha "var2 = var2 - var1;". Isso significa que, a cada execução do laço, o programa subtrai três do conteúdo de "var2" e atualiza o valor dessa variável com o resultado do cálculo. Ao final de cada iteração, a condição " $\text{var2} > 0$ " é checada, até que, finalmente, na quinta iteração, "var2" assume o valor -1, que é menor do que zero, o que torna a condição do laço falsa e interrompe a execução do laço.

Dessa forma, quando o programa chega à linha do "printf", as variáveis "var1" e "var2" têm os valores 3 e -1, respectivamente.

De acordo com a formatação indicada pelo comando "printf" e com os argumentos utilizados, a saída do programa deve ser: Resultado: 3 e -1.