

Estruturas de Repetição

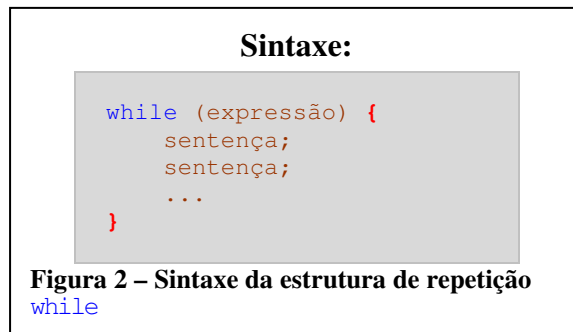
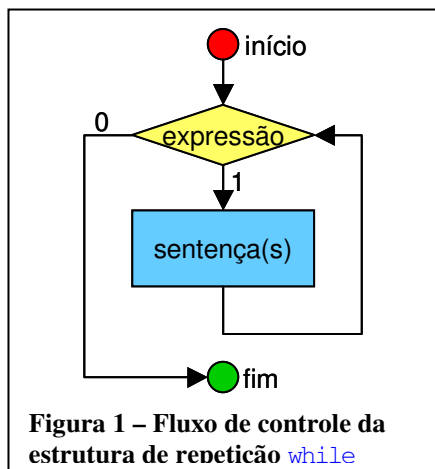
Introdução

No capítulo anterior verificamos que a execução seqüencial dos comandos da função `main` nos limita a uma programação de algoritmos muito simples. Passamos, então, a dedicar nossa atenção ao estudo de recursos de programação mais elaborados, tais como as estruturas condicionais `if...else` e `switch`, as quais permitem executar parte do código do programa somente sob determinadas condições.

Agora, estudaremos as estruturas de repetição, que permitem executar mais de uma vez um mesmo trecho de código. Trata-se de uma forma de executar blocos de comandos somente sob determinadas condições, mas com a opção de repetir o mesmo bloco quantas vezes for necessário. As estruturas de repetição são úteis, por exemplo, para repetir uma série de operações semelhantes que são executadas para todos os elementos de uma lista ou de uma tabela de dados, ou simplesmente para repetir um mesmo processamento até que uma certa condição seja satisfeita.

Estrutura de repetição `while`

O `while` é a estrutura de repetição mais simples. Ele repete a execução de um bloco de sentenças enquanto uma condição permanecer verdadeira. Na primeira vez que a condição se tornar falsa, o `while` não repetirá a execução do bloco, mas continuará a execução com a sentença ou comando que vem logo após o bloco do `while`, na seqüência do programa.



A Figura 1 ilustra a sintaxe da estrutura condicional `while` em C. O fluxo de execução desta estrutura está ilustrado na Figura 2. A *expressão* é uma condição que controla o `while`. Primeiro, o programa avalia a *expressão*. Ela utiliza os mesmos operadores de relacionais e lógicos estudados quando tratamos das estruturas condicionais. Caso o resultado da expressão seja não nulo (verdadeiro), então todo o *bloco de sentenças* será executado. Em seguida, o programa volta a avaliar a *expressão* e o processo se repete até que a expressão avalie como zero (falso). A *expressão* é sempre avaliada antes de decidir pela execução do *bloco de sentenças*.

Observação: A *expressão* deverá ser colocada, obrigatoriamente, entre parênteses.

A repetição do **while** é controlada por uma condição que verifica alguma característica do programa (por exemplo, valores de variáveis). Para o uso correto do **while**, o *bloco de sentenças* precisa modificar o estado do sistema de forma a afetar justamente as características testadas na *expressão*. Se isto não ocorrer, então o **while** executará eternamente.

Observação: O programa sempre executa o *bloco de sentenças* completo. Se neste tempo a condição se tornar falsa, o programa só verificará este fato quando avaliar novamente a expressão, preparando para uma nova repetição.

Exemplo

Para imprimir os números de 1 até 10:

```
int numero = 1;
while (numero <= 10) {
    printf("%d\n" , numero);
    numero = numero + 1;
}
```

Consulte: EstruturasRepeticao\while01\while01.vcproj

Declaramos uma variável **numero** que controlará o **while**. Ela armazena o valor a ser impresso. A expressão do **while** verifica se o número está dentro do limite desejado (menor ou igual a 10).

No início, o valor da variável **numero** é 1 e portanto satisfaz a expressão do **while**. O bloco de expressões é executado, imprimindo o valor de **numero** e aumentando seu valor em uma unidade. Note que isto afeta a condição que controla o bloco.

Na próxima repetição, a expressão será verdadeira para os valores de **numero** 2, 3, 4, ... 9 e 10. Quando **numero** armazenar o valor 11, a expressão que controla o **while** será falsa. Nesse ponto, o **while** termina, encerrando as repetições.

No final da execução, o valor da variável **numero** será 11, que foi justamente o valor que tornou a expressão falsa, impedindo uma nova execução do bloco **while**.

Estrutura de repetição **while** em uma linha

Sintaxe:

```
while (expressão)
    sentença;
```

Figura 3 – Sintaxe abreviada da estrutura condicional **while**

Quando o bloco da estrutura **while** contém apenas uma única sentença, pode-se omitir as chaves que delimitam o bloco, como na Figura 3. No entanto, essa forma não delimita claramente o código do **while** do restante do programa. Por ser mais confusa, evite a forma abreviada!

Exemplo

O próximo exemplo é um uso típico do **while** para realizar uma operação para um intervalo de números. Este programa imprime todos os divisores de um número inteiro positivo. Para o número *n* dado, o programa verifica se cada número de 1 até *n* é ou não um divisor de *n*.

Código Fonte

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

int main(int argc, char *argv[]) {
    int numero;
    int divisor;
    int resto;

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

    divisor = 1;
    while (divisor <= numero) {
        resto = numero % divisor;
        if (resto == 0) {
            printf("Divisor encontrado: %d \n", divisor);
        }
        divisor = divisor + 1;
    }

    return 0;
}
```

Consulte: EstruturasRepeticao\Divisores01\Divisores01.vcproj

Descrição passo a passo

```
int numero;
int divisor;
int resto;
```

Declara-se três variáveis inteiras. A variável `numero` armazenará o valor digitado pelo usuário, para o qual descobriremos os divisores. A variável `divisor` é um contador que conterá o próximo número que será testado como divisor. E `resto` é uma variável que armazenará temporariamente o resto da divisão de `numero` por `divisor`.

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

As duas linhas pedem ao usuário para digitar o número para o qual deseja descobrir os divisores.

```
divisor = 1;
while (divisor <= numero) {
    ...
    divisor = divisor + 1;
}
```

Este bloco corresponde à estrutura de repetição. Por motivos de simplicidade, o código executado dentro do bloco foi omitido com objetivo de entender sua lógica de funcionamento.

A repetição é controlada pelo valor da variável `divisor`. Isto quer dizer que o bloco precisa modificar o valor da variável `divisor` para, em algum momento, parar as repetições.

No início, antes de executar a repetição, a variável `divisor` é iniciada com um valor válido (1), que é o primeiro divisor possível. Em cada execução da repetição, a variável `divisor` é modificada, somando-se a ela uma unidade. Após certo número de repetições, valor da variável `divisor` ultrapassará o valor de `numero` e assim terminará a repetição.

```
resto = numero % divisor;
if (resto == 0) {
    printf("Divisor encontrado: %d \n", divisor);
}
```

O código executado dentro da repetição calcula o resto da divisão. Caso ele seja zero, significa que encontramos um divisor. Tal divisor é impresso.

Primeiro exemplo de execução:

```
Digite o numero: 100
Divisor encontrado: 1
Divisor encontrado: 2
Divisor encontrado: 4
Divisor encontrado: 5
Divisor encontrado: 10
Divisor encontrado: 20
Divisor encontrado: 25
Divisor encontrado: 50
Divisor encontrado: 100
```

Segundo exemplo de execução:

```
Digite o numero: 19
Divisor encontrado: 1
Divisor encontrado: 19
```

O que ocorre se o número informado for zero? E se for negativo?

Exemplo:

O próximo exemplo é um uso típico do `while` para realizar uma operação até que uma condição seja satisfeita. Esta condição não depende de uma variável contadora, tal como no exemplo anterior. Assim, não é possível prever facilmente o número de repetições. O programa calcula o máximo divisor comum (MDC) entre dois números positivos.

Código Fonte

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

int main(int argc, char *argv[]) {
    int numeroA;
    int numeroB;
    int resto;

    printf("Digite dois numeros (ordem crescente): ");
    scanf("%d %d", &numeroA, &numeroB);

    while (numeroA > 0) {
        resto = numeroB % numeroA;
```

```

        printf("numeroB = %d; numeroA = %d; ", numeroB, numeroA);
        printf("resto = %d\n", resto);
        numeroB = numeroA;
        numeroA = resto;
    }

    printf("MDC: %d", numeroB);

    return 0;
}

```

Consulte: EstruturasRepeticao\mdc01\mdc01.vcproj

Descrição passo a passo

O programa segue o mesmo algoritmo apresentado na introdução do curso, realizando divisões sucessivas até chegar no MDC.

```

int numeroA;
int numeroB;
int resto;

```

São declaradas três variáveis. As primeiras duas armazenam o valor digitado pelo usuário e, no decorrer do algoritmo, esses valores convergem para o MDC. A variável `resto` é apenas para armazenamento temporário dentro da repetição.

```

while (numeroA > 0) {
    resto = numeroB % numeroA;
    ...
    numeroB = numeroA;
    numeroA = resto;
}

```

A cada repetição, o algoritmo divide sucessivamente um valor pelo outro e guarda o resto. No final do bloco de repetição, a variável `numeroA` armazena o resto da divisão, e a variável `numeroB` armazena o último valor da variável `numeroA`.

A repetição é finalizada quando o resto for nulo, ou seja, quando a variável `numeroA` armazenar zero.

Exemplo de execução

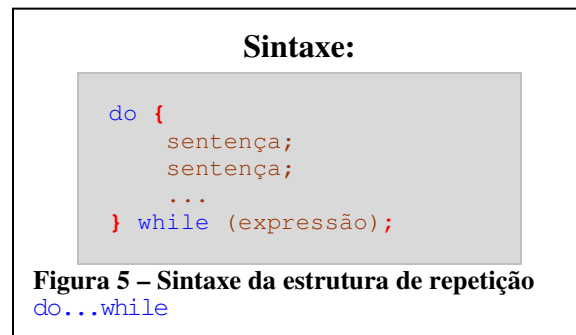
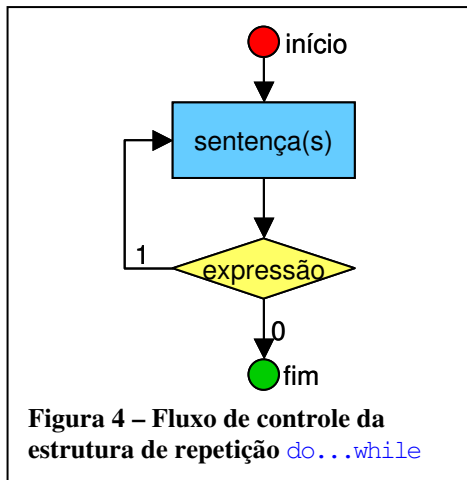
```

Digite dois numeros (ordem crescente): 30 36
numeroB = 36; numeroA = 30; resto = 6
numeroB = 30; numeroA = 6; resto = 0
MDC: 6

```

Estrutura de repetição `do...while`

Esta estrutura tem um comportamento muito semelhante ao `while`, com uma diferença crucial: a condição é verificada após executar o bloco.



A Figura 4 ilustra a sintaxe da estrutura condicional `do...while` em C. O fluxo de execução desta estrutura está ilustrado na Figura 5. A *expressão* é uma condição que controla o `do...while`.

Executa-se o *bloco de sentenças*, independentemente da condição. Somente então a *expressão* é avaliada. Caso ela seja não nula (verdadeira), então todo o *bloco de sentenças* será executado novamente. Este processo se repete até que a *expressão* avalie em zero (falso). A *expressão* é sempre avaliada depois da execução do *bloco de sentenças*.

Observação 1: A condição deverá ser colocada, obrigatoriamente, entre parênteses!

Observação 2: Não esquecer do ponto-e-vírgula após a expressão!

A diferença entre a estrutura `while` e `do...while` é sutil. Ela está no momento quando a condição de repetição é verificada: antes ou depois de executar o *bloco de sentenças*. A escolha dependerá do bom senso do programador, que optará pela estrutura que deixar o algoritmo mais simples e legível.

Exemplo

Para imprimir os números de 1 até 10:

```

int numero = 1;
do {
    printf("%d\n" , numero);
    numero = numero + 1;
} while (numero <= 10);
  
```

Consulte: EstruturasRepeticao\dowhile01\dowhile01.vcproj

Declaramos uma variável `numero` que controlará o `do...while`. Ela armazena o próximo valor a ser impresso. A expressão do `do...while` verifica se o número está dentro do limite desejado (menor ou igual a 10).

O bloco de sentenças é executado, imprimindo o valor de `numero` e aumentando seu valor em uma unidade. Após executar o bloco, a expressão verifica se a variável `numero` continua dentro do limite permitido. Caso afirmativo, o bloco é executado novamente. Assim, o bloco é executado quando `numero` armazenar 1, 2, 3, ..., 9 e 10.

No final da execução, o valor da variável `numero` será 11, que foi justamente o valor que tornou a expressão falsa, impedindo uma nova execução do bloco `do...while`.

Estrutura de repetição `do...while` em uma linha

Sintaxe:

```
do sentença while (expressão);
```

Figura 6 – Sintaxe abreviada da estrutura condicional `do...while`

Quando o bloco da estrutura `do...while` contém apenas uma única sentença, pode-se omitir as chaves que delimitam o bloco como na Figura 6. No entanto, essa forma não delimita claramente o código do restante do programa. Por ser mais confusa, evite a forma abreviada!

Exemplo

O programa MDC apresentado para a estrutura de repetição `while` pode ser re-escrito com um `do...while`. Este programa calcula o máximo divisor comum (MDC) entre dois números positivos.

Código Fonte

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

int main(int argc, char *argv[]) {
    int numeroA;
    int numeroB;
    int resto;

    printf("Digite dois numeros (ordem crescente): ");
    scanf("%d %d", &numeroA, &numeroB);

    do {
        resto = numeroB % numeroA;
        printf("numeroB = %d; numeroA = %d; ", numeroB, numeroA);
        printf("resto = %d\n", resto);
        numeroB = numeroA;
        numeroA = resto;
    } while (numeroA > 0);
    // ou while (resto > 0);

    printf("MDC: %d", numeroB);

    return 0;
}
```

Consulte: *EstruturasRepeticao\mdc02\mdc02.vcproj*

Descrição passo a passo

O programa segue a mesma lógica que o anterior. A descrição a seguir foca somente nas diferenças entre as estruturas de repetição.

```
do {
    resto = numeroB % numeroA;
    ...
    numeroB = numeroA;
    numeroA = resto;
} while (numeroA > 0);
```

Cada repetição divide um valor pelo outro e guarda o resto. Quando o resto for nulo, o valor do MDC foi encontrado e a repetição é terminada.

Exemplo de execução

```
Digite dois numeros (ordem crescente): 30 36
numeroB = 36; numeroA = 30; resto = 6
numeroB = 30; numeroA = 6; resto = 0
MDC: 6
```

Operadores de incremento

Talvez você tenha observado que, freqüentemente, as estruturas de repetição utilizam variáveis para controlar o número de repetições. No exemplo de imprimir números de 1 até 10, no final de cada iteração temos:

```
numero = numero + 1;
```

Em C, a sentença acima significa que a variável `numero` recebe um *novo valor* por causa do operador de atribuição. O novo valor é calculado da seguinte forma: somando-se 1 ao valor da variável no lado direito da atribuição. Como este tipo de atribuição é muito freqüente, a linguagem C oferece atalhos que podem ser práticos em estruturas de repetição:

Para:	Use o atalho:	Forma original:
Somar uma unidade ao valor da variável	<code>++numero;</code>	<code>numero = numero + 1;</code> <code>(retorne numero)</code>
Subtrair uma unidade do valor da variável	<code>--numero;</code>	<code>numero = numero - 1;</code> <code>(retorne numero)</code>

Observação:

1. É comum que programadores experientes utilizem estes operadores dentro de expressões complexas, até mesmo dentro das próprias condições que controlam a execução de um `while`. No momento, utilizaremos estes operadores apenas em expressões simples. É preferível criar um código simples e de fácil entendimento do que um código compacto.
2. O novo valor é atribuído à variável, cujo novo valor fica acrescido de uma unidade.
3. O novo valor é retornado como o valor da expressão.

Exemplo

Para imprimir os números de 1 até 10 com `while`:

```
int numero = 1;
while (numero <= 10) {
    printf("%d\n" , numero);
    ++numero;
}
```

Consulte: EstruturasRepeticao\while02\while02.vcproj

O mesmo exemplo com `do...while`

```
int numero = 1;
do {
    printf("%d\n" , numero);
```



```

    ++numero;
} while (numero <= 10);

```

Consulte: EstruturasRepeticao\dowhile02\dowhile02.vcproj

Existem comandos semelhantes aos anteriores, mas que retornam os valores que estavam armazenados nas variáveis *antes* de se realizar as operações de incremento ou de decremento.

Para:	Use o atalho:	Forma original:
Somar um ao valor da variável, retornando o valor original	<code>numero++;</code>	(retorne numero) <code>numero = numero + 1;</code>
Subtrair um do valor da variável, retornando o valor original	<code>numero--;</code>	(retorne numero) <code>numero = numero - 1;</code>

Exemplo

```

int numero = 1, val = 10;

val = numero--;
printf("val = %d, numero = %d\n" , val, numero);

```

Este trecho de código imprimirá:

```
val = 1, numero = 0
```

Observe que o novo valor da variável `val` é 1, ou seja, é o valor original de `numero`. Esse é o valor retornado pela expressão. Porém, o novo valor que é atribuído para `numero` é 0, ou seja, é o resultado da operação de decréscimo.

Operações aritméticas da forma

```
variavel = variavel op expressao
```

onde o `op` é um dos operadores aritméticos, também ocorrem com bastante frequência. Nesses caso, podemos escrever essas atribuições de forma simplificada como

```
variavel op= expressao
```

A tabela a seguir ilustra os casos possíveis.

Para:	Use o atalho:	Forma original:
Somar k unidades ao valor da variável	<code>numero += k;</code>	<code>numero = numero + k;</code>
Subtrair k unidades do valor da variável	<code>numero -= k;</code>	<code>numero = numero - k;</code>
Multiplicar o valor da variável por k	<code>numero *= k;</code>	<code>numero = numero * k;</code>
Dividir o valor da variável por k	<code>numero /= k;</code>	<code>numero = numero / k;</code>

Estrutura de repetição `for`

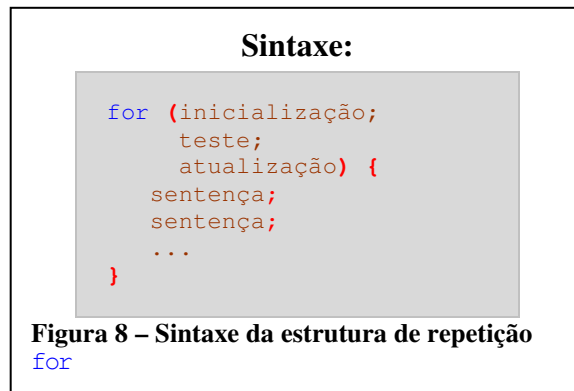
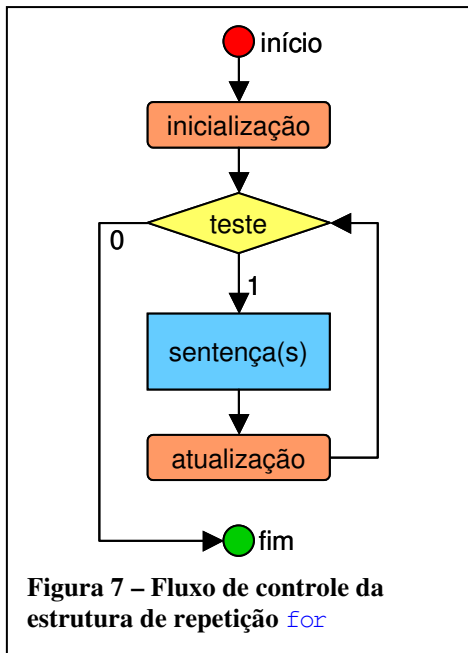
Na maioria dos casos, os algoritmos necessitam executar um bloco de sentenças por um número específico de vezes. Frequentemente, utiliza-se uma variável para controlar o número de repetições. Já aprendemos como fazer isso com `while` e `do...while` (vide exemplos para imprimir números de 1 até 10). Estas situações sempre apresentam uma **variável contadora** e as quatro etapas seguintes:

- **Inicialização:** Atribui um valor inicial à variável contadora.
- **Teste:** Verifica uma condição antes de decidir se executará o bloco de sentenças. Caso a condição seja falsa, interrompe. A condição verifica o valor da variável contadora.
- **Execução:** Executar o bloco de sentenças.
- **Atualização:** Atualiza a variável contadora para o próximo valor.

Vamos ilustrar com nosso primeiro exemplo: imprimir números de 1 até 10.

```
int numero = 1;           // Inicialização
while (numero <= 10) {    // Teste
    printf("%d\n", numero); // Execução
    ...
    numero = numero + 1;    // Atualização
}
```

O grande número de situações que requerem esta lógica justifica a próxima estrutura de repetição. Ela apresenta, de forma compacta, as etapas de *inicialização*, do *teste* e da *atualização*.



Um `for` sempre está acompanhado de uma variável contadora que armazena quantas vezes o bloco de sentenças do `for` deve ser executado. Na Figura 7 observamos que o programa faz a *inicialização*, que atribui o valor inicial da variável contadora. Em seguida avalia a *expressão*, que verifica se o valor da variável contadora está dentro do limite desejado. O *bloco de sentenças* é executado e, em seguida, é executada a *atualização*, que altera o valor da variável contadora. O processo se repete avaliando novamente a expressão. A sintaxe da estrutura `for` está na figura Figura 8.

Tipicamente, uma estrutura `for` ocorre como no modelo abaixo:

```

int contador;
for (contador = 1; contador <= 10; contador++) {
    ...
}
  
```

Exemplo

Para imprimir os números de 1 até 10:

```

int numero;
for (numero = 1; numero <= 10; numero++) {
    printf("%d ", numero);
}
  
```

Consulte: EstruturasRepeticao\for01\for01.vcproj

Declaramos uma variável `numero` que servirá como contador para o `for`. Ela armazenará a contagem de repetições. O `for` executa o bloco contendo o comando `printf` várias vezes, variando o valor da variável `numero` de 1 até 10. Observe a forma simplificada na escrita da expressão de atualização.

Resultado: 1 2 3 4 5 6 7 8 9 10

Para imprimir os números de 1 até 20, de 2 em 2:

```
int numero;  
for (numero = 1; numero <= 20; numero += 2) {  
    printf("%d ", numero);  
}
```

Consulte: EstruturasRepeticao\for02\for02.vcproj

A diferença está na atualização, que aumenta o valor de `numero` em duas unidades a cada repetição do bloco. Observe a forma simplificada na expressão de atualização.

Resultado: 1 3 5 7 9 11 13 15 17 19

Para imprimir os números de 10 até 1:

```
int numero;  
for (numero = 10; numero >= 1; numero--) {  
    printf("%d ", numero);  
}
```

Consulte: EstruturasRepeticao\for03\for03.vcproj

Ao invés de inicializar a variável `numero` com 1, ela é inicializada com 10. A cada repetição, a atualização deve reduzir seu valor em uma unidade, portanto escrevemos `numero--`. A condição deve verificar agora que o número é maior ou igual que 1.

Resultado: 10 9 8 7 6 5 4 3 2 1

Além de condensar uma lógica recorrente de programação em poucas linhas, o `for` possui outras duas vantagens importantes:

- O seu cabeçalho agrupa todas as instruções mais importantes que controlam a execução do `for`: a **inicialização**, o **teste** e a **atualização**. O programador é obrigado a declarar toda a lógica de execução em uma única linha e de uma só vez. Em uma estrutura `while`, um erro muito comum é o programador “esquecer” de inicializar ou atualizar a variável de controle.
- O cabeçalho separa claramente as instruções de controle de repetição das instruções de execução. No exemplo de imprimir números de 1 até 10, sem o uso do `for` a variável contadora é atualizada logo após a impressão. Em programas mais elaborados, as instruções de atualização tendem a ficarem escondidas ou diluídas dentro das demais instruções, tornando o programa obscuro e suscetível a erros de programação.

Toda a estrutura `for` é equivalente a um `while`. A escolha entre uma estrutura ou outra é uma questão de gosto e estilo de programação. Use o bom senso para realizar sua escolha.

Observação: Note a ordem correta das declarações no cabeçalho: *inicialização, teste e atualização*! Não esqueça do ponto-e-vírgula separando as declarações. Todo o cabeçalho deverá estar obrigatoriamente entre parênteses!

Estrutura de repetição `for` em uma linha

Sintaxe:

```
for (inicialização; teste; atualização)
    sentença;
```

Figura 9 – Sintaxe abreviada da estrutura condicional `while`

Quando o bloco da estrutura `for` contém apenas uma única sentença, pode-se omitir as chaves que delimitam o bloco, tal como na Figura 9. No entanto, essa forma não delimita claramente o código a ser executado repetidamente do restante do programa. Por ser mais confusa, evite a forma abreviada!

Exemplo

Este programa imprime todos os divisores de um número. Para um dado número n , o programa testa todos os números de 1 até n .

Código Fonte

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

int main(int argc, char *argv[]) {
    int numero;
    int divisor;
    int resto;

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

    for (divisor = 1; divisor <= numero; divisor++) {
        resto = numero % divisor;
        if (resto == 0) {
            printf("Divisor encontrado: %d \n", divisor);
        }
    }

    return 0;
}
```

Consulte: *EstruturasRepeticao\Divisores02\Divisores02.vcproj*

Descrição passo a passo

O programa utiliza a mesma lógica que o exemplo apresentado para a estrutura `while`. A discussão atém-se somente às diferenças na estrutura de repetição.

```
for (divisor = 1; divisor <= numero; divisor++) {
    resto = numero % divisor;
    if (resto == 0) {
        printf("Divisor encontrado: %d \n", divisor);
    }
}
```

A repetição é controlada pelo valor da variável `divisor`. Ela é inicializada em 1 e a repetição ocorre enquanto ela contiver valores menores ou iguais ao próprio

número. Dentro do bloco **for**, calcula-se o resto da divisão. Caso ele seja zero, significa que encontramos um divisor.

Primeiro exemplo de execução:

```
Digite o numero: 100
Divisor encontrado: 1
Divisor encontrado: 2
Divisor encontrado: 4
Divisor encontrado: 5
Divisor encontrado: 10
Divisor encontrado: 20
Divisor encontrado: 25
Divisor encontrado: 50
Divisor encontrado: 100
```

Segundo exemplo de execução:

```
Digite o numero: 19
Divisor encontrado: 1
Divisor encontrado: 19
```

Casos de Uso

Um programa pode ser escrito corretamente tanto com **while**, com **do...while** como com **for**. A escolha da estrutura cabe ao programador, que deve preferir aquela que deixa o código mais simples e fácil de entender.

```
while (expressão) { ... }
```

Objetivo: Executar o bloco apenas enquanto uma condição for verdadeira. Se a condição for falsa logo no início, o bloco não é executado nenhuma vez.

Sugerido quando:

- Não há necessidade de inicializar ou atualizar variáveis contadoras.
- As etapas de inicialização ou atualização requerem muitas instruções e não caberiam elegantemente em uma única linha do **for**.
- As informações necessárias para avaliar a condição não dependem de uma variável contadora ou são obtidas durante a execução do bloco.

Nestes três casos anteriores, prefira um **while** ao invés do **for**.

```
do { ... } while (expressão);
```

Objetivo: Executar o bloco pelo menos uma vez e repetir enquanto uma condição for verdadeira.

Sugerido quando:

- É necessário executar um bloco pelo menos uma vez para obter as informações necessárias para avaliar a condição.

É muito comum utilizar o **do...while** para leitura de dados. Um uso típico poderia ser repetir a leitura enquanto o dado não for válido.

```
for (inicialização; teste; reinicialização) { ... }
```

Objetivo: Executar o bloco um certo número de vezes, controlado por uma variável contadora.

Sugerido quando:

- O número de repetições é controlado por uma variável controladora.
- Há necessidade de inicialização e atualização, mas que sejam simples o suficiente para acomodar na linha do **for**. Para casos mais complexos, é melhor usar um **while**.
- A avaliação da condição não depende de dados obtidos na execução do bloco.

Considere o uso do **for** para separar claramente as instruções do controle de repetição (inicialização e atualização) das demais instruções do bloco.

Mas adiante, o **for** será uma ferramenta útil para percorrer elementos de vetores, listas e matrizes. A variável contadora do **for** será justamente o índice do vetor ou da matriz.

Exemplos Típicos

Para entender melhor quando utilizar cada uma das estruturas, analisaremos alguns exemplos que exigem execução repetida do mesmo bloco de código.

Caso 1, usando **for**: Ler uma quantidade fixa de valores

Primeiro, o usuário informa a quantidade de valores disponíveis e em seguida informa cada um dos valores. O programa calcula a média dos números lidos.

Neste caso, será conveniente utilizar um **for**, pois veremos que:

- Existe uma variável contadora que controla o número de repetições para a leitura dos valores.
- Há necessidade de inicializar e atualizar esta variável contadora. A inicialização e a atualização são instruções simples que são acomodadas facilmente em uma linha do **for**.
- É conveniente separar a inicialização e atualização do bloco que calcula a média.

Código fonte:

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

int main(int argc, char *argv[]) {
    int quantidade;
    int contador;
    double valor;
    double soma = 0;
    double media;
    // Solicita a quantidade de números que devem ser lidos
    printf("Quantidade de valores: ");
    scanf("%d", &quantidade);

    // Solicita cada um dos números e soma-o
    for (contador = 1; contador <= quantidade; contador++) {
        printf("Valor: ");
        scanf("%lf", &valor);
        soma += valor;
    }
    media = soma / quantidade;
    printf("Média: %f", media);
    return 0;
}
```

Descrição passo a passo:

```
int quantidade;
int contador;
double valor;
double soma = 0;
double media;
```

O programa começa declarando as variáveis. A variável `quantidade` armazenará o número de valores lidos para calcular a média. A variável `contador` controlará quantos valores já foram lidos. Já a variável `valor` armazenará o valor lido mais recentemente. E `soma` guardará a soma de todos os valores que são lidos. Finalmente, `media` armazenará o resultado calculado no final do programa.

```
printf("Quantidade de valores: ");
scanf("%d", &quantidade);
```

O programa solicita que o usuário digite o número de valores que devem ser lidos para calcular a média.

```
for (contador = 1; contador <= quantidade; contador++) {
    printf("Valor: ");
    scanf("%lf", &valor);
    soma += valor;
}
```

O `for` executa o bloco repetidamente, uma vez para cada valor de `contador` entre 1 e o número armazenado em `quantidade`. A cada repetição, solicita um número, o qual é somado ao valor anterior em `soma`, e que substitui o antigo valor nessa variável.

```
media = soma / quantidade;
printf("Média: %f", media);
```

No final, o programa calcula a média e apresenta o valor.

Exemplo de execução:

```
Quantidade de valores: 5
Valor: 3
Valor: 5
Valor: 4
Valor: 6
Valor: 5
Média: 4.600000
```

O que ocorre quando o usuário entra um valor zero ou negativo para `quantidade`? Modifique o programa para se precaver desses casos.

Caso 2, usando `while`: Ler uma quantidade fixa de valores

O mesmo programa pode ser escrito usando `while`, mas torna o código um pouco menos evidente.


```

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

int main(int argc, char *argv[]) {
    int quantidade;
    int contador;
    double valor;
    double soma = 0;
    double media;

    // Solicita a quantidade de números que devem ser lidos
    printf("Quantidade de valores: ");
    scanf("%d", &quantidade);

    // Solicita cada um dos números e soma-o
    contador = 1;
    while (contador <= quantidade) {
        printf("Valor: ");
        scanf("%lf", &valor);
        soma += valor;
        contador++;
    }

    // Calcula e mostra a média
    media = soma / quantidade;
    printf("Media: %f", media);
    return 0;
}

```

Consulte: EstruturasRepeticao\Caso2\Caso2.vcproj

Caso 3, usando **while**: Ler uma quantidade desconhecida de valores

Desejamos calcular a média de uma lista de números não negativos, de comprimento arbitrário. O usuário escreve a lista de números, indicando o fim da lista com um número negativo.

Neste caso, será conveniente utilizar um **while**, pois veremos que:

- Não existe uma variável que controla o número de repetições da leitura do valor.
- As informações necessárias para avaliar a condição são obtidas durante a execução do bloco.

Código fonte:

```

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

int main(int argc, char *argv[]) {
    int quantidade = 0;
    double valor;
    double soma = 0;
    double media;

    // Solicita cada um dos números e soma-o
    scanf("%lf", &valor);
    while (valor >= 0.0) {
        soma += valor;
        quantidade++;
        scanf("%lf", &valor);
    }
}

```

```

    }

    // Calcula e mostra a média
    media = soma / quantidade;
    printf("Media: %f", media);
    return 0;
}

```

Consulte: EstruturasRepeticao\Caso3\Caso3.vcproj

Descrição passo a passo:

```

int quantidade = 0;
double valor;
double soma = 0;
double media;

```

O programa declara variáveis de forma semelhante aos exemplos anteriores.

```

scanf("%lf", &valor);
while (valor >= 0.0) {
    soma += valor;
    quantidade++;
    scanf("%lf", &valor);
}

```

A diferença está no uso da estrutura **while** para determinar o momento para terminar a repetição da leitura.

O programa precisa ler o primeiro número antes de verificar a condição, o que justifica o **scanf** antes do **while**. Se ele for não negativo, então ele é somado no bloco do **while**. A variável **quantidade** é aumentada em uma unidade para saber quanto números foram somados até agora no cálculo da média. Por fim, é necessário ler o próximo número antes de avaliar novamente a condição do **while**. Por este motivo, a última linha do bloco contém um **scanf** para ler tal número.

```

media = soma / quantidade;
printf("Media: %f", media);

```

A média é calculada da mesma forma como nos dois exemplos anteriores. O que ocorre se a lista de números for vazia, isto é, o usuário entra logo com um primeiro valor que é negativo? Melhore o programa de forma que evite esse caso de contorno.

Exemplo de execução:

Escreva os valores, terminando com um número negativo.

```

3 5 4 6 5 -1
Media: 4.600000

```

Caso 4, usando **do...while**: Ler uma quantidade desconhecida de valores

O programa anterior pode ser reescrito usando a estrutura **do...while**.

Código Fonte:

```

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

int main(int argc, char *argv[]) {
    int quantidade = 0;

```

```

double valor;
double soma = 0;
double media;

// Solicita cada um dos números e soma-o
printf("Escreva os valores, terminando com um número negativo.\n");
do {
    scanf("%lf", &valor);
    if (valor >= 0.0) {
        soma += valor;
        quantidade++;
    }
} while (valor >= 0.0);

// Calcula e mostra a média
media = soma / quantidade;
printf("Media: %f", media);
return 0;
}

```

Consulte: EstruturasRepeticao\Caso4\Caso4.vcproj

Descrição passo a passo:

Este programa é semelhante ao caso anterior, variando apenas no uso da estrutura **do...while** ao invés do **while**.

O primeiro comando executado no bloco é o comando **scanf**, que lê o próximo valor. Se este valor for maior ou igual a zero, então ele é somando para o cálculo da média. Repare como agora resolvemos o problema usando o **scanf** apenas uma vez. Por outro lado, precisamos sempre testar o valor lido dentro do bloco do **scanf**. Poderíamos ter evitado o teste, sempre somando o valor lido dentro do bloco, mas desde que tenhamos o cuidado de, após terminada a execução do **do...while**, subtraímos o último valor somado, que terá sido somado indevidamente.

A condição de saída verifica se o último valor lido é maior ou igual a zero e repete o bloco caso afirmativo.

Comparação com o programa anterior

Para este programa, é difícil decidir qual estrutura de repetição é mais vantajosa: o **while** o **do...while**. Os dois programas são idênticos quanto ao resultado produzido.

O **do...while** é menos elegante pois é necessário verificar a condição duas vezes e por exigir um **if** dentro do **do...while**. O **while**, tem o inconveniente de exigir duas linhas com o comando **scanf**.

Caso 5, usando **do...while** : executar até que o usuário decida parar

Podemos mostrar um uso interessante do **do...while** com o seguinte programa. Ele é idêntico ao caso 1, mas possui a opção de ser executado várias vezes. Depois de calcular a média, o programa pergunta ao usuário se ele deseja repetir tudo de novo, para calcular uma nova média sobre números diferentes.

Código Fonte:

```

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

```

```

int main(int argc, char *argv[]) {
    int quantidade;
    int contador;
    double valor;
    double soma;
    double media;
    char repetir;

    do {
        // Solicita a quantidade de números que devem ser lidos
        printf("Quantidade de valores: ");
        scanf("%d", &quantidade);
        soma = 0; // inicializa soma com 0, inclusive nas demais iterações
        // Solicita cada um dos números e soma-o
        for (contador = 1; contador <= quantidade; contador++) {
            printf("Valor: ");
            scanf("%lf", &valor);
            soma += valor;
        }

        // Calcula e mostra a média
        media = soma / quantidade;
        printf("Media: %f\n\n", media);

        printf("Deseja executar o programa novamente? (s/n) ");
        scanf(" %c", &repetir);
    } while (repetir == 's');

    return 0;
}

```

Consulte: EstruturasRepeticao\Caso5\Caso5.vcproj

Descrição passo a passo:

Note que o bloco do **do...while** é idêntico ao programa do caso 1. No fim da execução do bloco, após mostrar a média, o programa solicita que o usuário digite uma das letras 's' ou 'n', para informar se ele quer executar novamente.

A condição do **do...while** é tal que, enquanto o usuário terminar o bloco com a letra 's', o programa é executado novamente.

Exemplo de execução:

```

Quantidade de valores: 5
Valor: 3
Valor: 4
Valor: 3
Valor: 6
Valor: 5
Media: 4.200000

Deseja executar o programa novamente? (s/n) s
Quantidade de valores: 3
Valor: 6
Valor: 8
Valor: 9
Media: 14.666667

Deseja executar o programa novamente? (s/n) n

```

Controle de Repetição

Forçar interrupção de repetição: `break`

O comando `break` é um modo conveniente de terminar imediatamente a execução de um bloco controlado por uma estrutura de repetição, sem necessidade de esperar a próxima avaliação da condição. O comando `break` é útil para interromper a execução de uma estrutura de repetição quando fica evidente que as demais repetições não produzirão novos resultados. Assim, o programa pode economizar algum tempo de execução. No próximo exemplo, que verifica se um número é primo, o comando `break` será usado para interromper as repetições assim que mais de dois divisores forem encontrados.

A Figura 10 ilustra o desvio no fluxo de execução causado pelo comando `break`. O `break` não espera o término da execução do restante do bloco. As linhas tracejadas mostram o

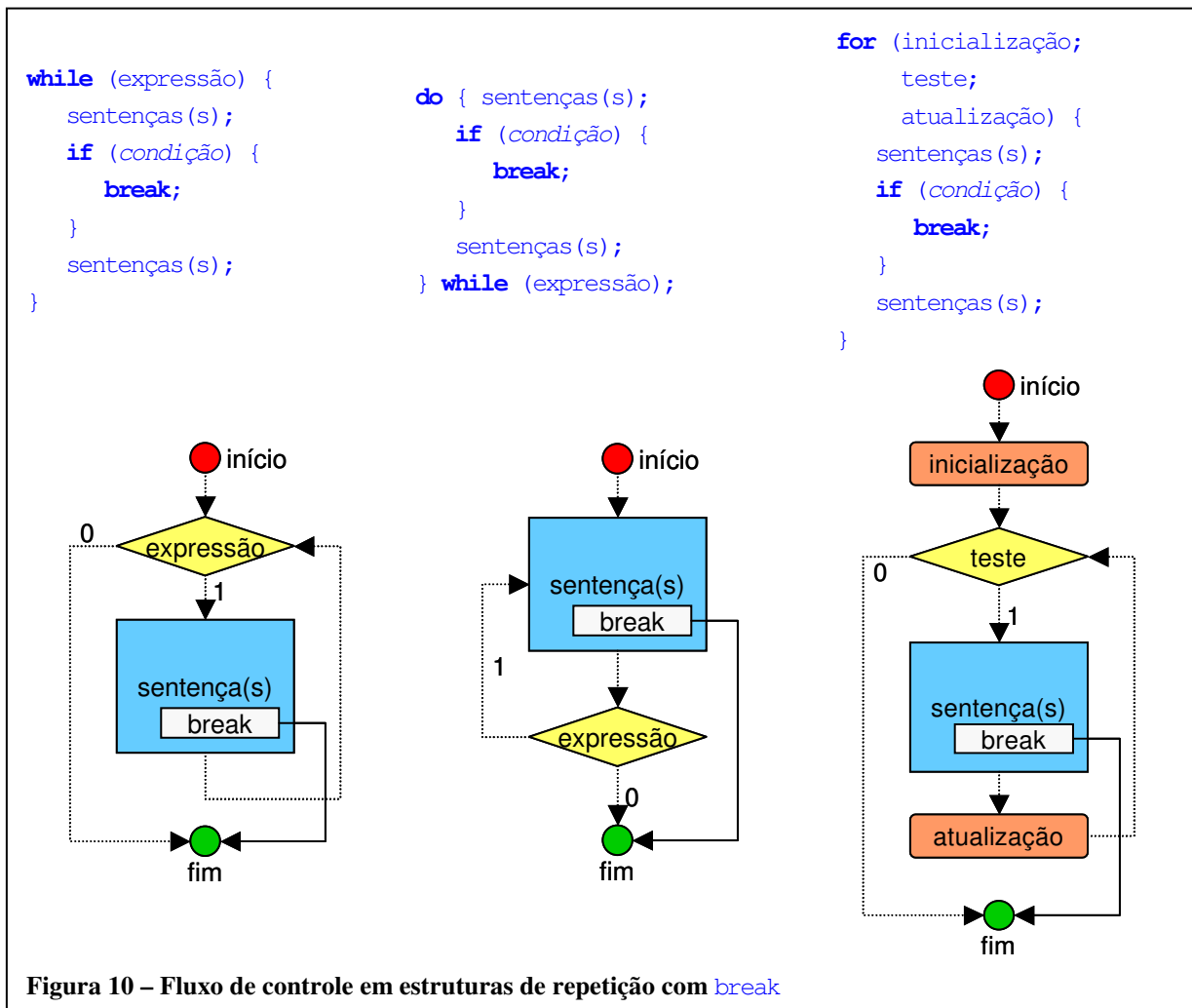


Figura 10 – Fluxo de controle em estruturas de repetição com `break`

fluxo convencional. A linha contínua representa o fluxo caso o `break` seja executado.

Observação: O uso do comando **break** costuma estar associado com uma estrutura condicional **if** para que a interrupção seja realizada somente sob determinadas condições excepcionais.

Exemplo

Um programa que verifica se um número é primo ou não. O programa utiliza o mesmo algoritmo utilizado para encontrar os divisores de um número.

Note que o número primo contém exatamente dois divisores (1 e o próprio número). Portanto, assim que o terceiro divisor for encontrado, ficará evidente que o número não é primo. Nesta situação, o programa utiliza o **break** para terminar a execução do **while**, evitando continuar as iterações seguintes, que não seriam úteis.

Código fonte

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

int main(int argc, char *argv[]) {
    int numero;
    int divisor;
    int resto;
    int numero_divisores;

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

    numero_divisores = 0;
    for (divisor = 1; divisor <= numero; divisor++) {
        resto = numero % divisor;
        if (resto == 0) {
            numero_divisores = numero_divisores + 1;
            if (numero_divisores >= 3) {
                break;
            }
        }
    }
    if (numero_divisores == 2) {
        printf("O numero %d eh primo!\n", numero);
    } else {
        printf("O numero %d NAO eh primo!\n", numero);
    }

    return 0;
}
```

Consulte: ControleExecucao\Divisores03\Divisores03.vcproj

Descrição passo a passo

```
int numero;
int divisor;
int resto;
int numero_divisores;
```

A variável **numero** armazena o valor digitado pelo usuário. A variável **divisor** é um contador para armazenar o próximo divisor a ser verificado. A variável **resto** é uma variável usada para armazenamento temporário dentro do bloco de repetição do **for**. E **numero_divisores** conta quantos divisores foram encontrados até o momento.

```

numero_divisores = 0;
for (divisor = 1; divisor <= numero; divisor++) {
    ...
}

```

A repetição é controlada pelo valor da variável `divisor`. Ela é inicializada em 1 e a repetição ocorre enquanto ela contiver valores menores ou iguais ao próprio número.

```

resto = numero % divisor;
if (resto == 0) {
    numero_divisores++;
    if (numero_divisores >= 3) {
        break;
    }
}

```

O código executado dentro da repetição calcula o resto da divisão. Sendo ele zero, significa que encontramos um divisor. Neste caso, o contador `numero_divisores` é atualizado. Verifica-se então se `numero_divisores` ultrapassou o número máximo de divisores para um número primo (2 divisores). Se verdadeiro, o `break` interrompe imediatamente a execução do `for`, independente de quantas repetições ainda faltem.

```

if (numero_divisores == 2) {
    printf("O numero %d eh primo!\n", numero);
} else {
    printf("O numero %d NAO eh primo!\n", numero);
}

```

No fim, se o número de divisores for 2, então temos um número primo. Em qualquer caso, uma mensagem é impressa informando este resultado. Note que o número 1 não é considerado primo. Nesse caso, o funcionamento do programa também é correto, pois apenas um divisor (ele mesmo) será encontrado.

Primeiro exemplo de execução

```

Digite o numero: 5
O numero 5 eh primo!

```

Primeiro exemplo de execução

```

Digite o numero: 10
O numero 10 NAO eh primo!

```

Reiniciar repetição: `continue`

O comando `continue` reinicia imediatamente a execução de um bloco de uma estrutura de repetição. O `continue` não espera o término da execução do restante do bloco. No caso do `while`, a execução retorna imediatamente para avaliar a expressão, antes de executar novamente o bloco, se for o caso. Se a expressão avaliar em falso, então o `while` é finalizado, caso contrário ele realiza uma nova iteração.

Para o **for**, o **continue** interrompe a execução normal do bloco, realiza imediatamente a atualização das variáveis de controle para, em seguida, realizar novamente o teste. Se o teste resultar em falso, então o **for** é finalizado, caso contrário ele realiza uma nova iteração.

No **do...while**, o **continue** simplesmente inicia uma nova iteração, no início do bloco.

O comando **continue** é útil para avançar para a próxima repetição quando fica evidente que a execução atual do bloco não se faz mais necessária. Veja o próximo exemplo para imprimir a tangente de ângulos de 0° a 180°. Quando a iteração chega no ângulo de 90°, é acionado o comando **continue** para avançar para o próximo ângulo, já que não existe tangente de 90°.

Observação: O uso do comando **continue** costuma estar associado com uma estrutura condicional **if** para que a repetição seja reiniciada somente sob determinadas condições.

A Figura 11 ilustra o desvio no fluxo de execução causado pelo comando **continue**. As linhas tracejadas mostram o fluxo convencional. A linha contínua representa o fluxo caso o **continue** seja executado.

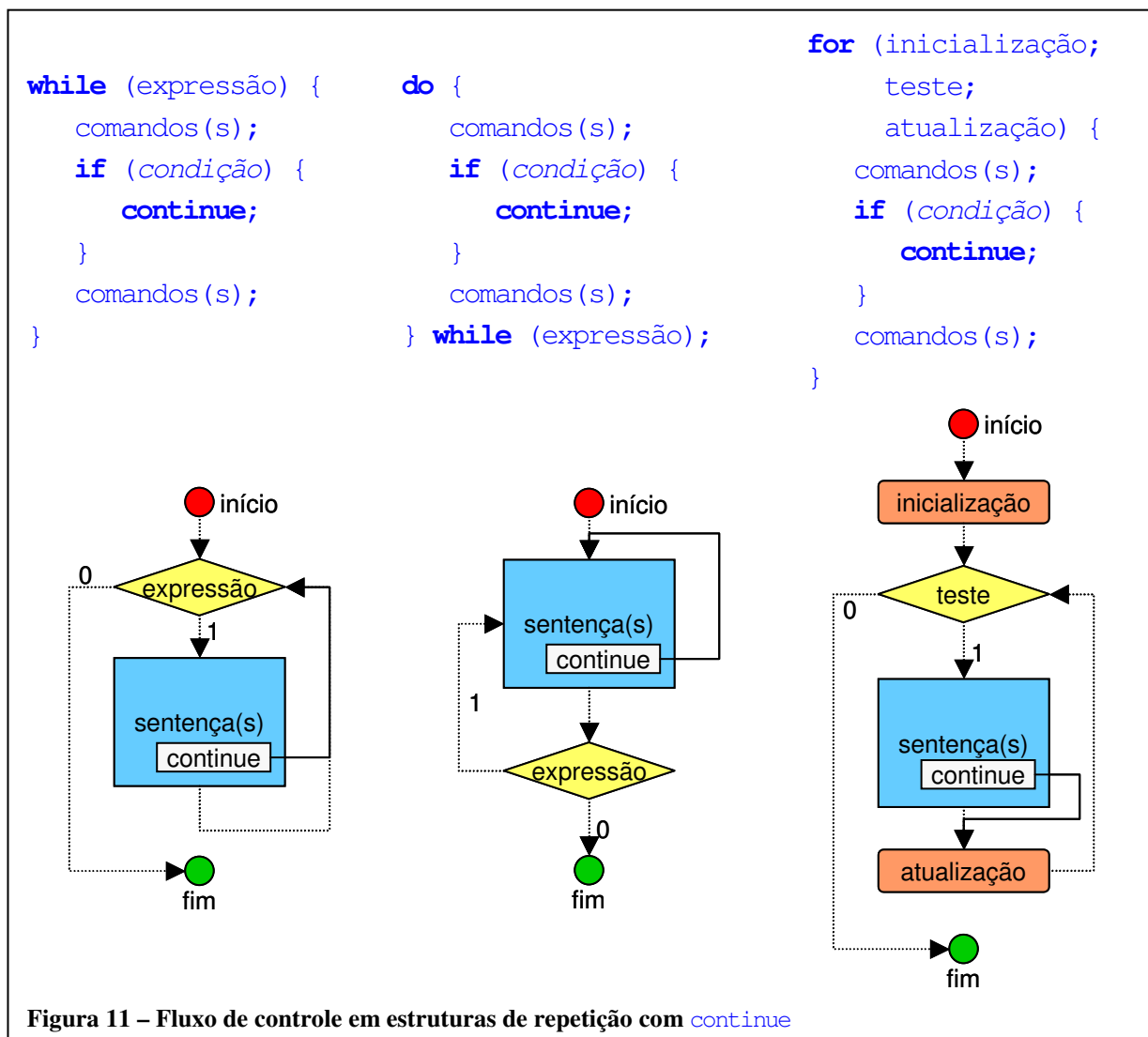


Figura 11 – Fluxo de controle em estruturas de repetição com **continue**

Exemplo:

Um programa que imprime uma tabela com a imagem da função tangente, em intervalos de 10 em 10 graus.

Código fonte

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

int main(int argc, char *argv[]) {
    double angulo;
    double pi = 3.14159265358979;

    for (angulo = 0; angulo <= 180; angulo += 10.0) {
        if (angulo == 90.0) {
            continue;
        }

        printf("tan(%f) = %f\n", angulo, tan(angulo/180*pi));
    }

    return 0;
}
```

Consulte: ControleExecucao\Tangete01\Tangente01.vcproj

Descrição passo a passo

```
double angulo;
double pi = 3.14159265358979;
```

A variável `angulo` percorre o intervalo de zero até 180 graus. Já `pi` é uma variável usada para realizar a conversão de graus para radianos.

```
for (angulo = 0; angulo <= 180; angulo += 10.0) {
    if (angulo == 90.0) {
        continue;
    }
    ...
}
```

O `for` executa o bloco para diferentes valores de ângulos, em passos de 10 em 10 graus. No entanto, ao chegar ao valor de 90.0, a função tangente não está definida! Por este motivo, este valor precisa ser ignorado. Para tal, utiliza-se o comando `continue` para reiniciar o `for` com o próximo valor (ou seja, 100 graus).

```
printf("tan(%f) = %f\n", angulo, tan(angulo/180*pi));
```

O `printf` não tem muito mistério. Ele imprime o ângulo e o valor da função tangente. No entanto, antes, é necessário realizar uma conversão para radianos, que é o formato esperado pela função `tan`.

Exemplo de execução

```
tan(0.000000) = 0.000000
tan(10.000000) = 0.176327
tan(20.000000) = 0.363970
tan(30.000000) = 0.577350
tan(40.000000) = 0.839100
tan(50.000000) = 1.191754
tan(60.000000) = 1.732051
```

```

tan(70.000000) = 2.747477
tan(80.000000) = 5.671282
tan(100.000000) = -5.671282
tan(110.000000) = -2.747477
tan(120.000000) = -1.732051
tan(130.000000) = -1.191754
tan(140.000000) = -0.839100
tan(150.000000) = -0.577350
tan(160.000000) = -0.363970
tan(170.000000) = -0.176327
tan(180.000000) = -0.000000

```

Fluxo de execução arbitrário: `goto`

O comando `goto` tem por finalidade desviar a execução do programa para qualquer outro ponto do programa, desconsiderando qualquer estrutura de repetição ou estrutura condicional. O `goto` pode desviar a execução para um ponto anterior na sequência (retrocesso), como também pode saltar para qualquer ponto situado mais para frente na sequência (avanço). Em qualquer um dos casos, é necessário declarar uma *marca* que será o destino do `goto`. A marca é um identificador seguido por dois-pontos (":"). O nome desse identificador segue as mesmas regras de escolha de nome que as mencionadas para as variáveis.

Sintaxe:

```

sentença(s);
...
marca1:
...
sentença(s);
...
goto marca1;
...
sentença(s);

```

Figura 12 – Sintaxe `goto`: retrocesso

Sintaxe:

```

sentença(s);
...
goto marca2;
...
sentença(s);
...
marca2:
...
sentença(s);

```

Figura 13 – Sintaxe `goto`: avanço

Exemplo:

Imprimir os números de 1 até 10.

Código fonte:

```

#include <stdio.h>
#include <stdlib.h>
/*
 * Imprime números de 1 até 10, usando apenas GOTO.
 */
int main() {
    int numero = 1;

    inicio_repeticao:
    if (numero > 10) { goto fim_repeticao; }
    printf("%d ", numero);
    numero++;
    goto inicio_repeticao;
    fim_repeticao:
    return 0;
}

```

Descrição passo a passo:

```
int numero = 1;
```

O programa começa declarando a variável `numero`, inicializada com o valor 1. Essa variável servirá de contador de repetições.

```
inicio_repeticao:
```

Em seguida, declara-se uma marca, que identifica o início do código que deverá ser repetido 10 vezes.

```
fim_repeticao:
```

No final do programa, declara-se outra marca, para definir o ponto para onde saltar após a execução das 10 iterações.

```
if (numero > 10) {
    goto fim_repeticao;
}
```

Assim que o contador de iterações ultrapassar o valor máximo permitido, o programa é instruído a saltar para o ponto `fim_repeticao`, localizado no final do código.

```
printf("%d\n" , numero);
numero++;
goto inicio_repeticao;
```

O programa imprime o número e atualiza a variável contadora. O comando `goto` força um desvio para o início, de modo a realizar novamente todas as operações para o próximo número na sequência.

Exemplo de execução:

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

Casos de uso do `goto`

Todas as estruturas de repetição podem ser escritas utilizando-se apenas marcas e `gotos`.

Mesmo assim, o emprego de `goto` não é recomendado por vários motivos, entre eles:

- Dificulta a visualização do destino de cada `goto`.
- Oculta a estrutura lógica e de execução do programa. Com o uso de `while` e `for`, é fácil identificar as condições de repetição, a inicialização e atualização de variáveis, como também o bloco que deve ser repetido. Com `gotos`, os blocos não estarão mais realçados por delimitações.
- Cada programador pode adotar sua própria lógica de fluxo de execução. Os programas tornar-se-ão incompreensíveis e muito difíceis de entender e de manter!
- Toda a lógica de `goto` pode ser re-escrita de forma muito mais elegante usando `while`, `do...while`, `for` e, se necessário, também `break` e `continue`.