

Estruturas de Repetição

enquanto e repita enquanto

Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Gregghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva

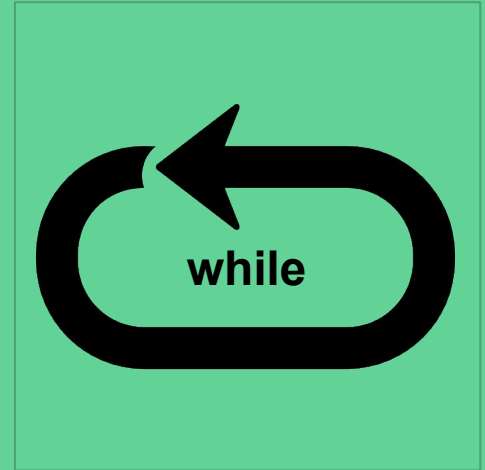


Roteiro

- Estrutura de repetição *enquanto*
- Estrutura de repetição *repita enquanto*
- Variáveis auxiliares em repetição
- Testando arquivos

Estrutura de Repetição

enquanto



Estrutura de Repetição

- Contextualização

- Considere o seguinte problema:

- Suponha que você precise implementar um programa para calcular a média aritmética de um conjunto de números positivos. Contudo, a quantidade de números será desconhecida. Sabe-se apenas que o usuário do programa irá digitar um número negativo para informar que não há mais números positivos disponíveis.

- A questão neste problema é: Como construir um programa que seja capaz de ler esta quantidade indefinida de números?

Estrutura de Repetição

- Definições

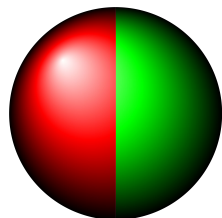
- Uma estrutura de repetição em uma linguagem de programação permite que um ou mais comandos sejam executados repetidas vezes até que: (1) uma determinada **condição de interrupção** seja satisfeita; ou (2) uma determinada **condição de permanência** não seja mais satisfeita.



Estrutura de Repetição

- A condição a ser verificada, seja ela de interrupção ou de permanência, pode ser avaliada no **início de cada ciclo de repetição** ou no **final dele**. Cada uma destas duas possibilidades nos fornecem um comando de programação diferente em C++.

Avaliação no início do ciclo



Avaliação no final do ciclo

ou

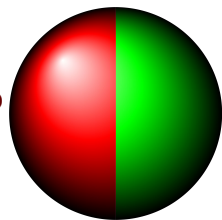
- Uma terceira possibilidade, que será melhor discutida no futuro, é executar um determinado conjunto de comandos por um número fixo de vezes.

Estrutura de Repetição

- Definições

- Quando a avaliação da condição é realizada no início de cada ciclo de repetição, temos em C++ o comando enquanto (**while**). Quando a avaliação é feita ao final de cada ciclo, temos o comando repita enquanto (**do..while**).

Avaliação no início do ciclo
Comando while



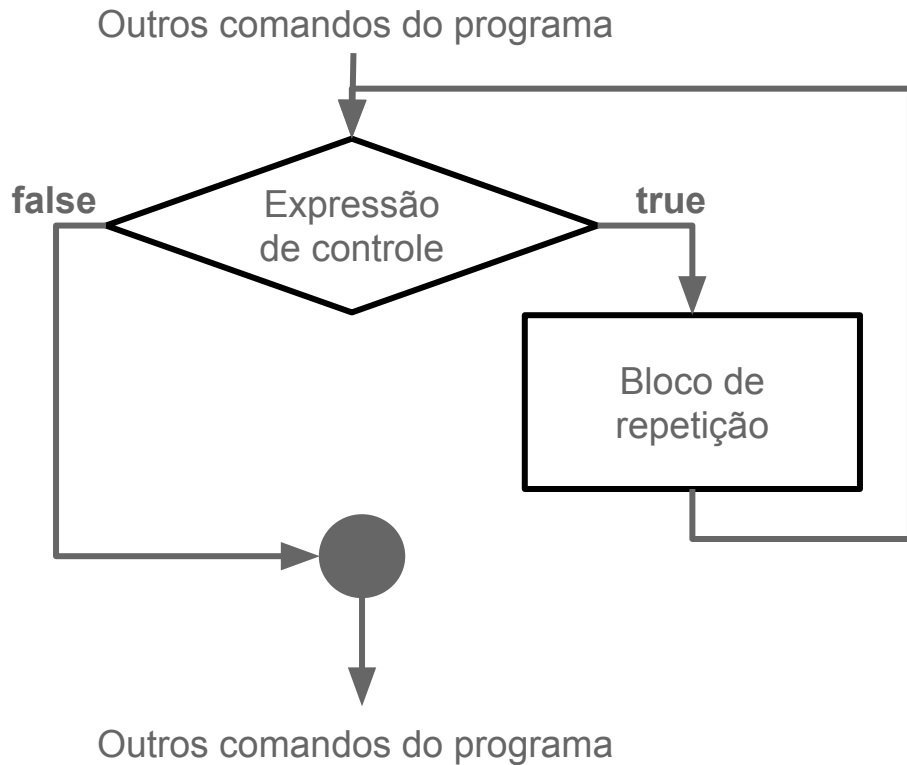
ou

Avaliação no final do ciclo
Comando do..while

Estrutura de Repetição *enquanto*

- A estrutura de repetição do tipo **enquanto** é caracterizada por dois componentes principais, a saber:
 - **Expressão de Controle:** expressão relacional ou lógica que deve ser avaliada a fim de se obter um valor booleano. Na prática, a expressão de controle caracteriza a condição de interrupção ou permanência do laço de repetição.
 - **Bloco de repetição:** conjunto de comandos que serão executados enquanto a expressão de controle assumir o valor verdadeiro (**true**).

Estrutura de Repetição *enquanto*




É importante observar que ao final da execução do **Bloco de repetição**, o fluxo de ações volta para a **Expressão de controle**, a fim de avaliá-la novamente e decidir se o **Bloco de repetição** será executado mais uma vez ou não.

Estrutura de Repetição *enquanto*

- Sintaxe em pseudocódigo:

```
Enquanto Expressão_controle Faça  
Início-Enquanto  
    Bloco_de_repetição  
Fim-Enquanto
```



Os comandos que forem descritos no **Bloco_de_repetição** serão executados enquanto a **Expressão_controle** continuar assumindo valores **true** ao ser avaliada. Caso o valor **false** seja obtido, todos os comandos do **Bloco_de_repetição** são completamente ignorados e é retomado o fluxo original de ações do programa.

Estrutura de Repetição *enquanto*

- **Sintaxe em C++:**

```
while (Expressão_controle) {  
    Bloco_de_repetição  
}
```

ou

```
while (Expressão_controle)  
{  
    Bloco_de_repetição  
}
```

Em C/C++, os símbolos de chaves, { e }, são utilizados para designar/delimitar um bloco de código.

Caso um bloco de código possua apenas um único comando, os símbolos de chaves podem ser omitidos se o programador assim o desejar.

Não deixar a **Expressão_controle** entre símbolos de parênteses, (e), gerará um erro de sintaxe.



Exemplo - Calcular MDC

Enunciado do problema:

Construir um programa em C++ que dado dois números inteiros positivos não nulos A e B, encontre o Máximo Divisor Comum (MDC) entre eles.

O MDC entre dois ou mais números naturais é o maior divisor possível de ambos os números.

Exemplo - Calcular MDC

Sugestão de algoritmo:

O MDC entre dois números naturais é o último resto não nulo do processo de divisões sucessivas de um dos valores (o dividendo) pelo outro (o divisor), substituindo-se a cada passo: i) o dividendo pelo divisor e ii) o divisor pelo resto obtido durante a divisão. A divisão é realizada enquanto o divisor é não-nulo.

Por exemplo, assumindo inicialmente A igual a 282 e B igual a 114:

(1º) A: 282, B: 114

(2º) Resto de 282 dividido por 114: 54

(3º) Substituindo o maior pelo menor e menor pelo resto: A: 114, B: 54

(4º) Resto de 114 dividido por 54: 6

(5º) Substituindo o maior pelo menor e o menor pelo resto: A: 54, B: 6

(6º) Resto de 54 dividido por 6: 0

Por exemplo, assumindo inicialmente A igual a 282 e B igual a 114:

(1º) A: 282, B: 114


(2º) Resto de 282 dividido por 114: 54

(3º) Substituindo o maior pelo menor e menor pelo resto: A: 114, B: 54

(4º) Resto de 114 dividido por 54: 6

(5º) Substituindo o maior pelo menor e o menor pelo resto: A: 54, B: 6

(6º) Resto de 54 dividido por 6: 0



Resto nulo: encerra a repetição.

Por exemplo, assumindo inicialmente A igual a 282 e B igual a 114

(1º) A: 282, B: 114

(2º) Resto de 282 dividido por 114: 54

(3º) Substituindo o maior pelo menor e menor pelo resto: A: 114, B: 54

(4º) Resto de 114 dividido por 54: 6

(5º) Substituindo o maior pelo menor e o menor pelo resto: A: 54, B: 6

(6º) Resto de 54 dividido por 6: 0

Último resto não nulo
da repetição.
Ou seja,
 $\text{MDC}(282,114) = 6$.

Resto nulo: encerra a
repetição.

Exemplo: Cálculo do MDC

programa MDC

a, b, r: **inteiros**

Início

leia(a,b)

enquanto ($b \neq 0$) **faça**

$r \leftarrow \text{resto}(a,b)$

$a \leftarrow b$

$b \leftarrow r$

fim-enquanto

// em a temos o último resto não nulo

escreva(a)

Fim



Tendo este pseudocódigo, é fácil convertê-lo para a linguagem C++.

Exemplo: Cálculo do MDC

```
#include <iostream>
using namespace std;

int main(){
    int dividendo, divisor, resto;
    cin >> dividendo >> divisor;

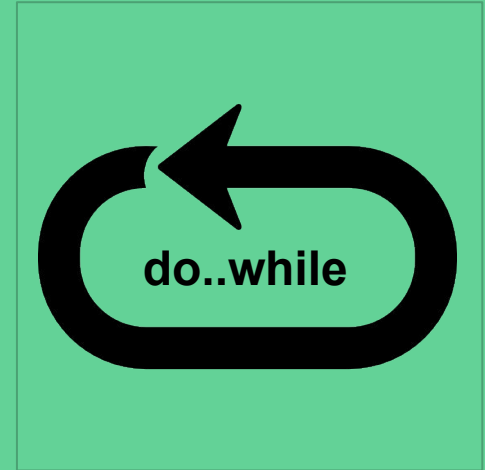
    while (divisor != 0) {
        resto = dividendo % divisor;
        dividendo = divisor;
        divisor = resto;
    }

    // no dividendo temos o último resto não nulo
    cout << dividendo << endl;
    return 0;
}
```



Estrutura de Repetição

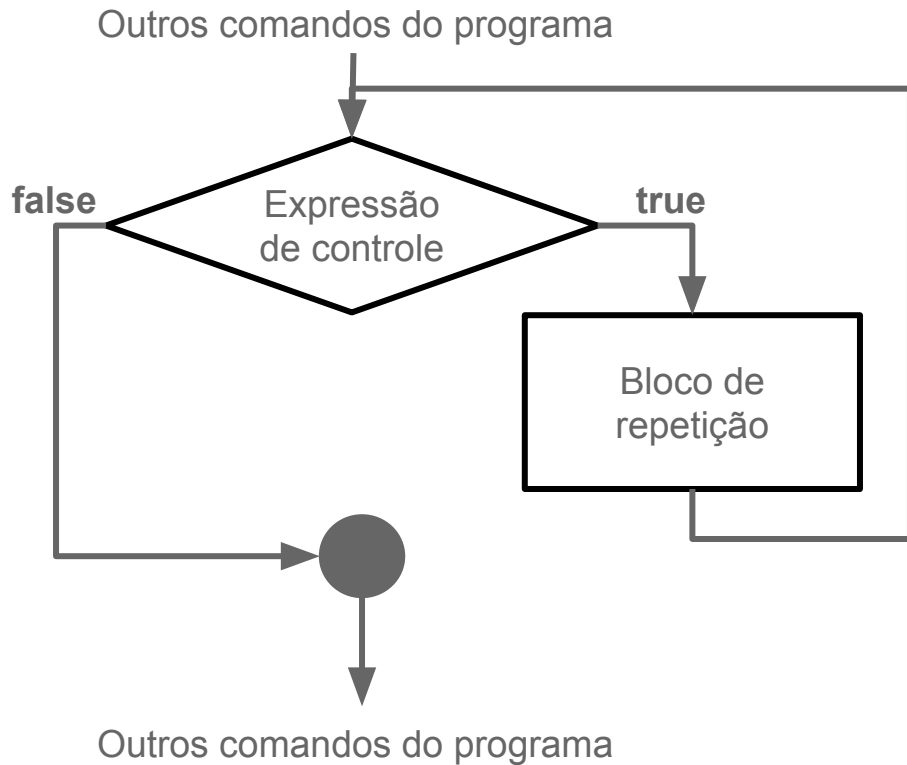
repita enquanto



Estrutura de Repetição *repita enquanto*

- A estrutura de repetição *repita enquanto*, também chamada *faça enquanto*, é caracterizada por dois componentes principais, a saber:
 - **Expressão de Controle:** expressão relacional ou lógica que deve ser avaliada a fim de se obter um valor booleano. Na prática, a expressão de controle caracteriza a condição de interrupção ou permanência do laço de repetição.
 - **Bloco de repetição:** conjunto de comandos que serão executados enquanto a expressão de controle mantiver o valor **true**.
- A principal diferença para a estrutura de repetição anterior (comando enquanto), é que na estrutura *repita enquanto*, a expressão de controle é avaliada após a execução do bloco de repetição, e não antes dele como no caso anterior. Assim a repetição acontece ao menos uma vez.

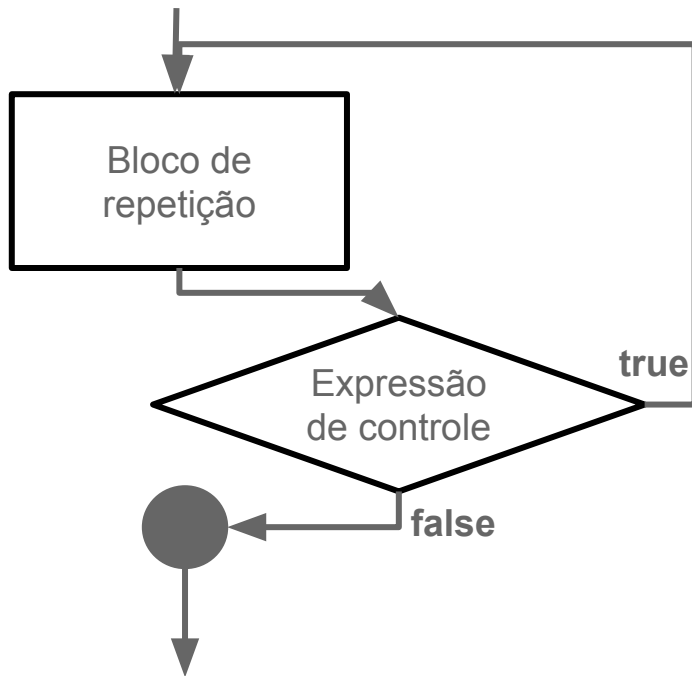
Estrutura de Repetição *enquanto*



É importante observar que ao final da execução do **Bloco de repetição**, o fluxo de ações volta para a **Expressão de controle**, a fim de avaliá-la novamente e decidir se o **Bloco de repetição** será executado mais uma vez ou não.

Estrutura de Repetição *repita enquanto*

Outros comandos do programa



Outros comandos do programa



É importante observar que, neste caso, a execução começa diretamente no **Bloco de repetição** e só então o fluxo de ações é direcionado para a **Expressão de controle**, a fim de avaliá-la e decidir se o **Bloco de repetição** será executado mais uma vez ou não.

Note que ao utilizar a estrutura do tipo **repita**, tem-se a garantia de que o **Bloco de repetição** será executado, no mínimo, uma vez.

Estrutura de Repetição *repita enquanto*

- Sintaxe em pseudocódigo:

Faça
Início-Faça
 Bloco_de_repetição
Enquanto Expressão_controle



Os comandos que forem descritos no **Bloco_de_repetição** serão executados enquanto a **Expressão_controle** continuar assumindo valores **true** ao ser avaliada. Caso o valor **false** seja obtido, todos os comandos do **Bloco_de_repetição** são completamente ignorados e é retomado o fluxo original de ações do programa.

Estrutura de Repetição *repita enquanto*

- **Sintaxe em C++:**

```
do  
{  
    Bloco_de_repetição  
} while (Expressão_controle);
```

ou

```
do {  
    Bloco_de_repetição  
} while (Expressão_controle);
```

Em C/C++, os símbolos de chaves, { e }, são utilizados para designar/delimitar um bloco de código.

Caso um bloco de código possua apenas um único comando, os símbolos de chaves podem ser omitidos se o programador assim o desejar.

Não deixar a **Expressão_controle** entre símbolos de parênteses, (e), gerará um erro de sintaxe. Assim como esquecer o símbolo de ponto-e-vírgula após a **Expressão_controle**.



Exemplo

- Problema: construir um programa em C++ que leia as notas dos candidatos que prestaram um concurso até que seja lido um valor de nota negativo. Calcule e informe no dispositivo de saída padrão a quantidade de candidatos que fizeram a prova e a maior de todas as notas.
- Assuma que sempre haverá no mínimo uma nota válida (um candidato) em qualquer concurso.

Exemplo: Maior nota e quantidade de candidatos

```
#include <iostream>
using namespace std;
int main() {
    float nota;
    float maior = -1;
    int quantidade = -1;
    do {
        cin >> nota;
        quantidade++;
        if (nota > maior)
            maior = nota;
    } while (nota >= 0);
    cout << "Quantidade de candidatos: " << quantidade << endl;
    cout << "Maior nota: " << maior << endl;
    return 0;
}
```

Exemplo: Maior nota e quantidade de candidatos

```
#include <iostream>
using namespace std;
int main() {
    float nota;
    float maior = -1;
    int quantidade = -1;
    do {
        cin >> nota;
        quantidade++;
        if (nota > maior)
            maior = nota;
    } while (nota >= 0);
    cout << "Quantidade de candidatos: " << quantidade << endl;
    cout << "Maior nota: " << maior << endl;
    return 0;
}
```

Observação 1

Note que o comando:
quantidade++;

Poderia ser escrito como:
quantidade = quantidade + 1;

Exemplo: Maior nota e quantidade de candidatos

```
#include <iostream>
using namespace std;
int main() {
    float nota;
    float maior = -1;
    int quantidade = -1;
    do {
        cin >> nota;
        quantidade++;
        if (nota > maior)
            maior = nota;
    } while (nota >= 0);
    cout << "Quantidade de candidatos: " << quantidade << endl;
    cout << "Maior nota: " << maior << endl;
    return 0;
}
```

Observação 2

Note na estrutura condicional deste exemplo, temos apenas um único comando dentro do seu Bloco Condicional. Sendo assim, o uso dos símbolos de chaves, { e }, são opcionais e, no exemplo, foram omitidos.

Exemplo: Maior nota e quantidade de candidatos

```
#include <iostream>
using namespace std;
int main() {
    float nota;
    float maior = -1;
    int quantidade = -1;
    do {
        cin >> nota;
        quantidade++;
        if (nota > maior)
            maior = nota;
    } while (nota >= 0);
    cout << "Quantidade de candidatos: " << quantidade << endl;
    cout << "Maior nota: " << maior << endl;
    return 0;
}
```

Observação 3

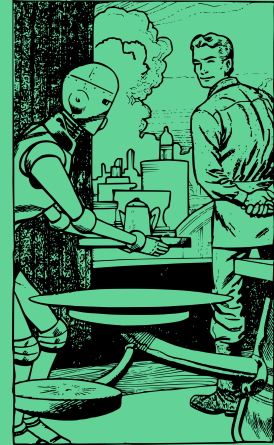
Por que as variáveis *maior* e *quantidade* foram inicializadas com o valor -1?

Considerando a solução em C++ dada, essas variáveis poderiam ser inicializadas com algum valor diferente?



Refleta e faça o testes
para descobrir você
mesmo!!!

Variáveis Auxiliares em repetições



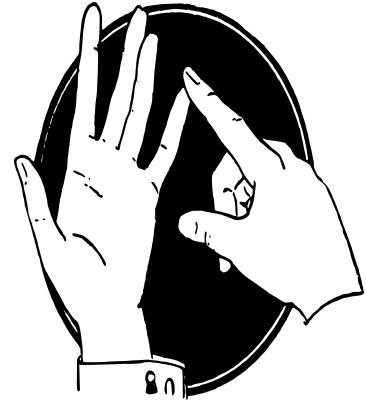
Variáveis auxiliares em repetições

Usualmente, ao manipular estruturas de repetição, três categorias de variáveis são muito comuns:

- **Contadores:** usadas para contar o número de repetições de um laço que satisfaz as necessidades de um dado problema qualquer.
- **Acumuladores:** armazenam valores acumulativos (ex: somatórios ou produtórios) nas diversas repetições realizadas.
- **Sentinelas:** variáveis utilizadas para controlar uma estrutura de repetição (seu valor lógico é verificado para continuar ou interromper a estrutura)

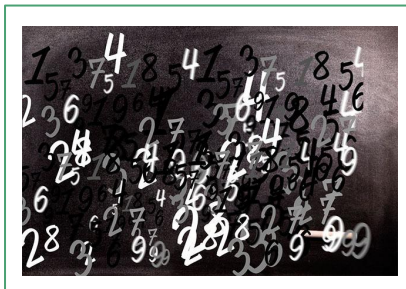
Contadores

- Variáveis contadoras são utilizadas para contabilizar o número de ocorrências de um determinado evento durante o ciclo de repetições de uma estrutura de repetição qualquer.
 - Quando contabilizamos em uma variável exatamente a quantidade de vezes que uma determinada estrutura de repetição é executada, chamamos essa variável de iteradora.
- Podem ser utilizadas de forma crescente ou decrescente.



Contadores - Exemplo

Problema: construir um programa em C++ que leia uma sequência de números inteiros positivos não nulos até que o valor zero seja fornecido. O programa construído deve calcular e exibir no dispositivo de saída padrão a quantidade de números pares não nulos existentes na sequência fornecida.



Exemplo: Quantidade de números pares

```
#include <iostream>
using namespace std;

int main()
{
    int num;
    int quantidade = 0;

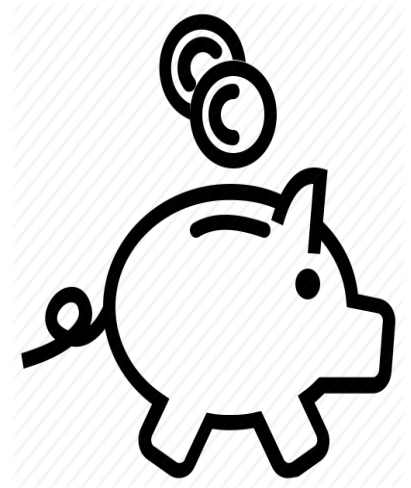
    do {
        cin >> num;

        if (((num % 2) == 0) and (num != 0))
            quantidade++;
    } while (num != 0);

    cout << "Quantidade de pares: " << quantidade << endl;
    return 0;
}
```

Acumuladores

- Variáveis acumuladoras são utilizadas para armazenar resultados de operações parciais, como somas ou produtos intermediários no cálculo de um somatório, produtório, fatorial ou de uma média, por exemplo.
- Variáveis acumuladoras armazenam resultados que são atualizados a cada passo (iteração) dentro de uma estrutura de repetição.



Acumuladores - Exemplo

Problema: construir um programa em C++ que leia um número inteiro positivo não nulo N , que indica a quantidade de elementos de uma sequência de números reais que será fornecida em seguida. Seu programa deverá calcular e exibir a média aritmética dos números reais da sequência fornecida.

Note que, neste exemplo, precisaremos de duas variáveis auxiliares: uma variável contadora (para contabilizar quantos números foram utilizados durante o processo de leitura) e uma variável acumuladora (para somar todos os números reais).

Exemplo: Média de uma sequência de números reais

```
#include <iostream>
using namespace std;

int main() {
    int num;
    int cont = 0;
    float valor;
    float soma = 0;
    cin >> num;
    while (cont < num) {
        cin >> valor;
        cont++;
        soma += valor;
    }

    cout << "Média: " << float(soma)/num << endl;
    return 0;
}
```

Exemplo: Média de uma sequência de números reais

```
#include <iostream>
using namespace std;
```

```
int main() {
    int num;
    int cont = 0;
    float valor;
    float soma = 0;
    cin >> num;
    while (cont < num) {
        cin >> valor;
        cont++;
        soma += valor;
    }

    cout << "Média: " << float(soma)/num << endl;
    return 0;
}
```

Observação 1

cont é uma variável contadora, responsável por indicar a quantidade de valores lidos durante a execução do programa. Como no momento da declaração da variável ainda não foi fornecido nenhum valor, a variável é inicializada como 0.

Exemplo: Média de uma sequência de números reais

```
#include <iostream>
using namespace std;
```

```
int main() {
    int num;
    int cont = 0;
    float valor;
    float soma = 0;
    cin >> num;
    while (cont < num) {
        cin >> valor;
        cont++;
        soma += valor;
    }
```

```
    cout << "Média: " << float(soma)/num << endl;
    return 0;
```

```
}
```

Observação 2

soma é uma variável acumuladora, responsável por somar todos os valores lidos durante a execução do programa. Como no momento da declaração da variável ainda não foi fornecido nenhum valor, a variável é inicializada como 0.

Exemplo: Média de uma sequência de números reais

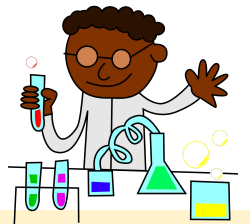
```
#include <iostream>
using namespace std;
```

```
int main() {
    int num;
    int cont = 0;
    float valor;
    float soma = 0;
    cin >> num;
    while (cont < num) {
        cin >> valor;
        cont++;
        soma += valor;
    }
```

```
    cout << "Média: " << float(soma)/num << endl;
    return 0;
}
```

Observação 3

O que aconteceria com este programa se as variáveis **cont** e **soma** não tivessem sido inicializadas com o valor 0?



Faça o teste e descubra
você mesmo!!!

Exemplo: Média de uma sequência de números reais

```
#include <iostream>
using namespace std;
```

```
int main() {
    int num;
    cin >> num;
    int cont = num;
    float valor;
    float soma = 0;
    while (cont > 0) {
        cin >> valor;
        cont--;
        soma += valor;
    }

    cout << "Média: " << float(soma)/num << endl;
    return 0;
}
```

Observação 4

Uma forma diferente de resolver o problema é utilizando um contador decrescente, que começa o número de valores a serem lidos. A repetição termina quando o valor torna-se nulo.

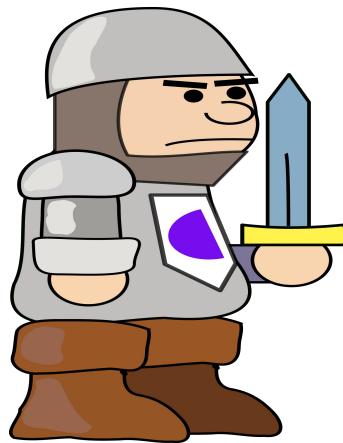


Compare! Tente você mesmo e verifique!

Sentinelas

Sentinelas são variáveis cujo valor lógico é utilizado para **controlar** uma estrutura de repetição.

Em geral são utilizadas para tornar código mais legível, ou mais eficiente, dependendo o caso, em situações em que muitas condições precisam ser avaliadas para continuar uma estrutura de repetição.



Sentinelas - Exemplo

Imagine que você precisa calcular uma soma de vários valores que serão digitados pelo usuário. Entretanto deve interromper o processo em uma das três situações a seguir:

1. A soma dos valores digitados é igual ou superior a 200.
2. O usuário já digitou 20 números.
3. Um número negativo foi digitado (nesse caso, ele não é somado).

Exemplo de uso de sentinelas

```
#include <iostream>
using namespace std;

int main() {
    int cont = 0, soma = 0, num;
    bool tudoOK = true;
    do {
        cin >> num;
        if (num >= 0) {
            cont++;
            soma+= num;
            if ((cont == 20) or (soma >= 200)) {
                tudoOK = false;
            }
        } else
            tudoOK = false;
    } while (tudoOK);
    cout << soma << endl;
    return 0;
}
```

Exemplo de uso de sentinelas

```
#include <iostream>
using namespace std;

int main() {
    int cont = 0, soma = 0, num;
    bool tudoOK = true;
    do {
        cin >> num;
        if (num >= 0) {
            cont++;
            soma += num;
            if ((cont == 20) or (soma >= 200)) {
                tudoOK = false;
            }
        } else
            tudoOK = false;
    } while (tudoOK);
    cout << soma << endl;
    return 0;
}
```

A variável **tudoOK** é usada como sentinela: a repetição só é realizada enquanto ela é verdadeira.

Também seria possível utilizar uma variável **deveTerminar**, por exemplo, em que a repetição ocorreria apenas enquanto ela se mantivesse falsa.

Exemplo de uso de sentinelas

```
#include <iostream>
using namespace std;

int main() {
    int cont = 0, soma = 0, num;
    bool tudoOK = true;
    do {
        cin >> num;
        if (num >= 0) {
            cont ++;
            soma += num;
            if ((cont == 20) or (soma >= 200)) {
                tudoOK = false;
            }
        } else
            tudoOK = false;
    } while (tudoOK);
    cout << soma << endl;
    return 0;
}
```

Por questões de espaço, declaramos várias variáveis em uma única linha, separando-as por vírgula.

Alguns desenvolvedores preferem que apenas uma única variável seja declarada em cada linha.

Uso de sentinelas para evitar uso de break

A maioria das linguagens de programação permite utilizar o `break` para interromper uma estrutura de repetição.

Apesar de ser um recurso poderoso, ele torna o código menos legível, uma vez que você não enxerga como a repetição termina realmente, sem olhar todo o código dela. Se for uma repetição longa, as chances de fazer confusão são grandes.

Cancelando Iterações (Verificar se é Primo)

```
#include <iostream>
using namespace std;

int main() {
    int numero;
    cin >> numero;

    bool primo = true;
    int divisor = 2;

    while ( divisor < numero ) {
        if (numero % divisor == 0) {
            primo = false;
            break;
        }
        divisor++;
    }

    return 0;
}
```

Código de difícil manutenção!
Não se sabe como a estrutura termina sem olhá-la inteira!
Evite uso de **break** fora de **switch**!

Cancelando Iterações (Verificar se é Primo)

```
#include <iostream>
using namespace std;

int main() {
    int numero;
    cin >> numero;

    bool primo = true;
    int divisor = 2;

    while (( divisor < numero ) and primo) {
        if (numero % divisor == 0) {
            primo = false;
        }
        divisor++;
    }

    return 0;
}
```

Código de mais fácil manutenção.

Todas as condições de saída da estrutura de repetição estão na condição.

Variável **primo** foi utilizada como sentinela para ajudar a interromper o laço.

Cancelando Iterações (Verificar se é Primo)

```
#include <iostream>
using namespace std;

int main() {
    int numero;
    cin >> numero;
```

Código de mais fácil manutenção.

Não há motivos razoáveis para uso do **break** em outras situações que não interromper um **switch**.

Mesmo que ele possa implicar algum ganho questionável de eficiência, sua manutenção é difícil e induz a erros e bugs diversos com o aumento do código.

As condições de saída da estrutura de repetição estão na condição.

A variável **primo** foi utilizada como flag para ajudar a interromper o laço.

```
return 0;
```

```
}
```

Testando arquivos



Relembrando arquivos com streams

Em C++, arquivos são entendidos como streams cujos dados estão guardados em um dispositivo de armazenamento secundário. Existem três tipos de streams para manipulação de arquivos:

- **ifstream** (para entrada/leitura de dados)
- **ofstream** (para saída/escrita de dados)
- **fstream** (para entrada e saída)

Para ter acesso, deve-se incluir a biblioteca `<fstream>`.

Manipulando arquivos com *streams*

Sobre a escrita (*ofstream*):

- Se o arquivo não existir, o mesmo será criado;
- Se o arquivo já existir, seu conteúdo será apagado.



Sobre a leitura (*ifstream*):

- Se o arquivo não existir, o mesmo não será criado.

A forma mais simples de utilizar arquivos é como um fluxo de texto:

- Escrevemos no arquivo de modo similar ao uso de **cout**;
- Lemos no arquivo de modo similar ao uso de **cin**



E se o arquivo de entrada não existir?? - i

Suponha que eu tenha uma variável **entrada** que deveria ser associada a um arquivo **dados.txt**. Mas por engano, foi declarada, por exemplo, como:

```
ifstream entrada("dados.txt");  
entrada >> minhaVariavel;
```

Caso seja feita a tentativa de leitura, como o arquivo associado não existe, a leitura não irá ocorrer adequadamente e **minhaVariavel** receberá lixo da memória. Além da leitura não ocorrer adequadamente, a variável **entrada** será marcada como inadequada para uso.

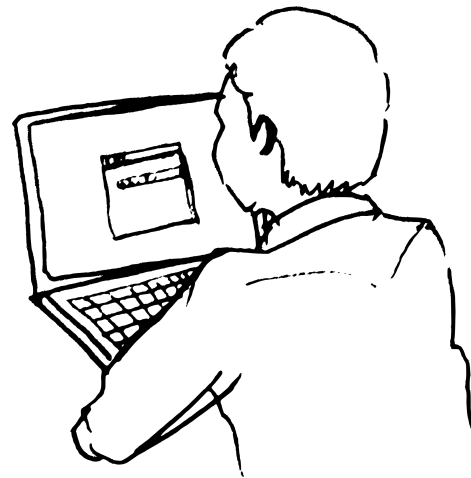
E se o arquivo de entrada não existir?? - ii

Nós podemos utilizar esse recurso de marcação da variável associada ao arquivo para verificar e garantir que o arquivo tenha sido aberto corretamente, utilizando por exemplo:

```
if (nomeVariavelArquivo) {  
    /* Bloco de instruções 1  
       Arquivo apto para uso */  
} else {  
    /* Bloco de instruções 2  
       Arquivo inapto para uso */  
}
```

Lendo do arquivo

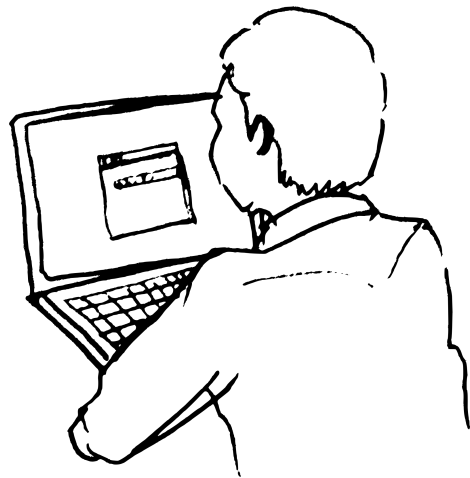
```
#include <fstream>
using namespace std;
int main(){
    ifstream arquivo("meus_dados.txt");
    string dados;
    if (arquivo) {
        while ( arquivo >> dados )
            cout << dados << endl;
        arquivo.close();
    }
    return 0;
}
```



Lendo do arquivo

```
#include <fstream>
using namespace std;
int main(){
    ifstream arquivo("meus_dados.txt");
    string dados;
    if (arquivo) {
        while ( arquivo >> dados )
            cout << dados << endl;
        arquivo.close();
    }
    return 0;
}
```

Esse while será repetido enquanto for possível ler um valor para a variável dados a partir do arquivo



Testando a leitura - i

O teste

```
while ( arquivo >> dados ) {  
    (...)
```



verifica se foi possível ler do arquivo, verificando, entre outros, se não chegou ao final do arquivo. Esse teste é mais efetivo e eficiente que:

```
while ( !arquivo.eof() ) {  
    arquivo >> dados;  
    (...)
```

Testando a leitura - ii

Da mesma forma que o operador de leitura, outros comandos de leitura, como `getline()` ou `get()`, podem ser usados como teste para indicar se a última leitura foi realizada ou não com sucesso. Assim, é possível ler dados de um arquivo até que eles acabem.

A variável que representa o arquivo também pode ser testada com finalidades similares.



Exemplo de uso de arquivos

Calcular a média de um conjunto de valores decimais disponibilizados em um arquivo texto chamado “dados.dat”.

Não se sabe, a priori, qual a quantidade de números no arquivo, deve-se efetuar a leitura até o último número.

```
#include <fstream>
#include <iostream>
using namespace std;
int main(){
    ifstream arq("dados.dat");
    float soma = 0.0;
    int num = 0;
    float valor;
    if (arq) {
        while ( arq >> valor ) {
            soma += valor;
            num++;
        }
        cout << soma/num << endl;
    } else
        cout << "arquivo não pode ser aberto!" << endl;
    return 0;
}
```



```
#include <fstream>
#include <iostream>
using namespace std;
int main(){
```

```
[joukim@harpia tmp]$ ./arqtexto
arquivo não pode ser aberto!
[joukim@harpia tmp]$
```

```
    while ( arq >> valor ) {
        soma += valor;
        num++;
    }
    cout << soma/num << endl;
} else
    cout << "arquivo não pode ser aberto!" << endl;
return 0;
}
```



```
#include <fstream>
#include <iostream>
```

```
[joukim@harpia tmp]$ cat dados.dat
2.35 5.87 4.16 1.21 3.333 9.11111 1 0 90
121 81 12 34 1.1 6.5 7.8
2.55
3.23454 9.1 3.4 12.12 13.13 14.14
[joukim@harpia tmp]$ ./arqtexto
19.0482
[joukim@harpia tmp]$
```

```
    } else
        cout << "arquivo não pode ser aberto!" << endl;
    return 0;
}
```



***Status* de um arquivo**

Além do controle de operações de Entrada e Saída (E/S), um arquivo também tem informações a respeito do seu *status*.

Quando uma operação de E/S falha, o arquivo fica marcado como impróprio para E/S e todas as operações de E/S seguintes falham.

Para continuar realizando operações de E/S em arquivo após um erro, será necessário resetar o status e o controle das posições, utilizando `arquivo.clear()`.

Sobre o Material



Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).