

Vetores e repetições contadas

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



Roteiro

- Contextualização
- Tipos enumerados
- Arranjos / Vetores
- Repetições contadas
- Repetições *para cada*

Contextualização



Contextualização

- Comumente, os tipos fundamentais escalares (por exemplo: **char**, **int**, **bool**, **double**, etc.) não são suficientes para representar todos os tipos de informação.
 - Isso é particularmente verdadeiro quando se tem mais de uma informação relacionada ou quando se tem um volume grande de informação.
 - Exemplo: um programa que precisa representar todas as notas em avaliações dos alunos matriculados em uma disciplina.

Contextualização

- **Solução:** utilizar os tipos fundamentais escalares para se construir estruturas de dados mais complexas.



- Para compreender melhor esta situação, vamos analisar o seguinte problema:
 - Um professor quer implementar um aplicativo para calcular a média de notas da sua sala, com cinco alunos.

Contextualização

- Exemplo:

- O professor poderia representar a nota de cada aluno como:

```
float aluno1, aluno2, aluno3, aluno4, aluno5;
```

- E na sequência, ler os valores das variáveis e calcular a média:

```
cin >> aluno1 >> aluno2 >> aluno3 >> aluno4 >> aluno5;  
float media = (aluno1 + aluno2 + aluno3 + aluno4 + aluno5)/5;
```

Contextualização

- Exemplo:

- Em teoria, este tipo de solução funcionaria, porém ela traria uma série de desvantagens:
 - Torna difícil a manutenção do código.
 - Prejudica a legibilidade do programa.
 - Favorece a proliferação de erros.
- Tente imaginar o mesmo problema, porém em uma escala um pouco maior...



Contextualização

- Exemplo:
 - E se o professor possuir dez alunos? Ou se a turma possuir cem alunos?
 - A criação de uma variável distinta para cada aluno ainda é uma opção viável e adequada?
 - Em uma escala maior para este problema, é impraticável o uso de uma variável diferente para cada aluno, pois isto certamente levará a algum tipo de erro, principalmente de escrita dos nomes das variáveis ou de esquecimento de alguma delas.

Contextualização

- Como já comentado, a solução para esta situação, é utilizar os tipos fundamentais escalares para se construir estruturas de dados mais complexas, normalmente por meio da criação de tipos derivados de dados.
- Como exemplos de tipos derivados de dados pode-se citar:
 - Tipos enumerados.
 - Arranjos/vetores.
 - Matrizes.
 - Registros.

Tipos enumerados

123
456
789



Enum

- Um exemplo de tipo derivado são os tipos enumeráveis, que são úteis para simplificar alguns algoritmos.
- **Definição:** um enum (enumeração) é um tipo derivado de dados que é definido pelo usuário (programador) que consiste de constantes inteiras nomeadas.
- **Sintaxe em C++:**

```
enum identificador_enumeração {constantes_inteiras_nomeadas};
```

Enum

- Exemplo:

```
enum cor {vermelho, amarelo, verde, azul};  
cor minha_cor = vermelho;  
if (minha_cor == amarelo){  
    //...  
}
```

Em um tipo enum, a primeira constante recebe o valor 0 e as subsequentes recebem o valor da constante anterior mais 1. Ou seja, neste exemplo, vermelho é 0, amarelo é 1, verde é 2 e azul é 3.

Enum

- Exemplo:

```
enum cor {vermelho, amarelo, verde, azul};  
cor minha_cor = vermelho;  
if (minha_cor == amarelo){  
    //...  
}
```

Uma vez que o tipo **enum** para **cor** foi definido pelo programador, a partir de então, é possível declarar variáveis deste novo tipo. Sendo assim, neste exemplo, **cor** é o tipo da variável e **minha_cor** é o identificador (nome) da variável.

Em um tipo enum, a primeira constante recebe o valor 0 e as subsequentes recebem o valor da constante anterior mais 1. Ou seja, neste exemplo, vermelho é 0, amarelo é 1, verde é 2 e azul é 3.

Enum

- Em C++, também podemos alterar o número inteiro associado a cada constante. Por exemplo, fazendo:

```
enum cor {vermelho, amarelo, verde = 7, azul};  
cor minha_cor = vermelho;  
if (minha_cor == amarelo){  
    //...  
}
```

Neste exemplo, vermelho é 0, amarelo é 1, verde é 7 e azul é 8 (lembrando que constantes subsequentes recebem o valor da constante anterior mais 1).

Enum

- Em C++, também podemos alterar o número inteiro associado a cada constante. Por exemplo, fazendo:

```
enum cor {vermelho, amarelo, verde = 7, azul};  
cor minha_cor = vermelho;  
if (minha_cor == amarelo){  
    //...  
}
```

Como no enum trabalha-se o conceito de constantes inteiras nomeadas, as variáveis do tipo enum podem ser manipuladas como números inteiros. Sendo assim, poderíamos trabalhar da seguinte forma:

Neste exemplo, vermelho é 0, amarelo é 1, verde é 7 e azul é 8 (lembrando que constantes subsequentes recebem o valor da constante anterior mais 1).

```
if (minha_cor == 1){
```

Exemplo - dia de praticar algoritmos

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

enum diasUteis {segunda=2,terca,quarta,quinta,sexta};
int main() {
    int dia;
    cin >> dia;
    if (dia == segunda)
        cout << "Segunda é dia de praticar algoritmos" << endl;
    else if (dia == terca)
        cout << "Terça é dia de praticar algoritmos" << endl;
    else if (dia == quarta)
        cout << "Quarta é dia de praticar algoritmos" << endl;
    else if (dia == quinta)
        cout << "Quinta é dia de praticar algoritmos" << endl;
    else if (dia == sexta)
        cout << "Sexta é dia de praticar algoritmos" << endl;
    else
        cout << "Não é dia útil? Ótimo para praticar algoritmos!" << endl;
    return 0;
}
```


Arranjos/Vetores



Vetores

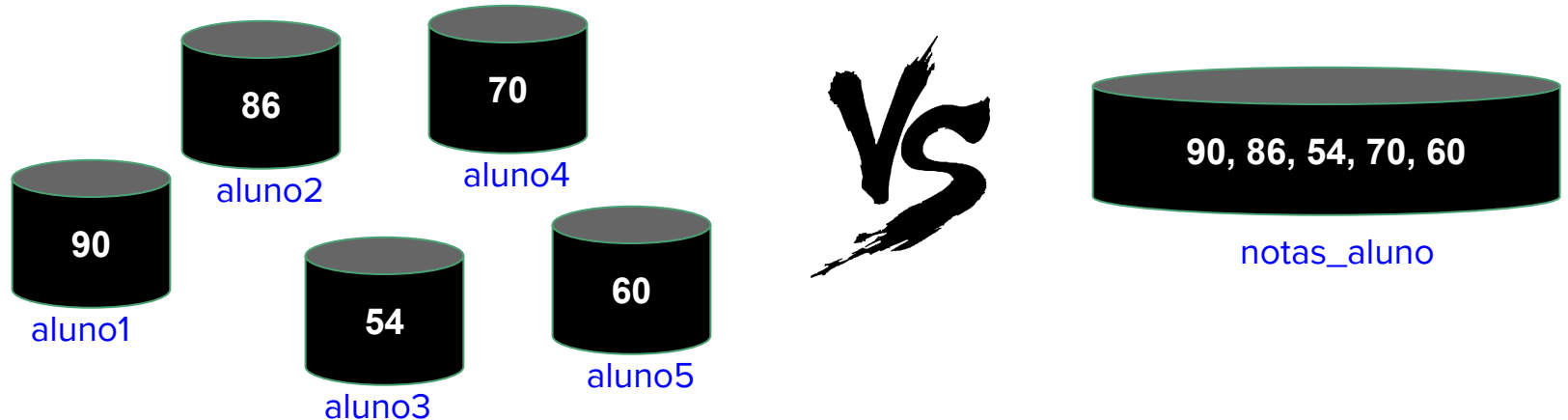
- Os tipos derivados são geralmente divididos em duas categorias: **variáveis homogêneas** e **variáveis heterogêneas**, de acordo com a forma que os dados são agrupados.
- Variáveis homogêneas permitem o agrupamento de um mesmo tipo de dado (informação de mesma natureza). Em geral, essas variáveis representam um mesmo tipo de conceito. Por exemplo, as notas de um mesmo aluno ou as notas de todos os alunos em uma mesma classe.
 - Podem ser do tipo lista, conjunto ou arranjos (vetores).

Vetores

- **Definição:** um arranjo, ou vetor, é um conjunto de locais consecutivos de memória para o armazenamento de elementos de um mesmo tipo.
 - Estes elementos podem ser selecionados de acordo com a sua posição relativa no arranjo (posição do elemento em relação aos demais).
 - Em outras palavras, é uma variável que possui várias posições de memória para armazenamento, também denominada variável composta homogênea.

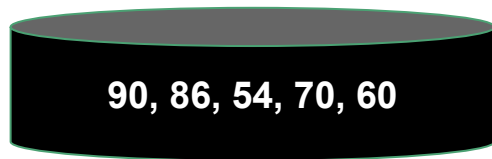
Vetores

- Por exemplo, ao invés de implementarmos cinco variáveis diferentes para armazenar as notas de cinco alunos, poderíamos implementar uma única variável e esta teria a capacidade de armazenar simultaneamente todos os valores necessários.



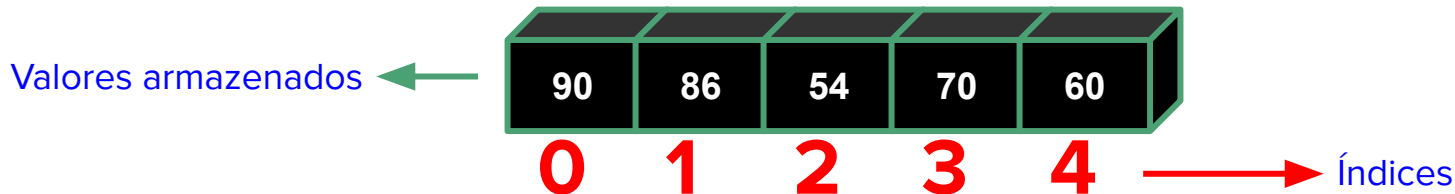
Vetores

- No exemplo:



notas_aluno

temos uma única variável do tipo vetor chamada **notas_alunos**. Esta variável ocupa **5** locais de memória para armazenamento de números inteiros. Cada um destes locais de memória pode ser identificado de acordo com a sua posição relativa (também conhecida em programação como o termo **índice**) em relação aos demais.



Vetores

- **Sintaxe em pseudocódigo (declaração de vetor):**

identificador_variável ← array **A..B** de tipo_dado

Em que **A** é um número inteiro que representa o índice da primeira posição do vetor e **B** representa o índice da última posição do vetor.
Exemplo:

V ← array 1..100 de inteiros

Algumas linguagens de programação iniciam o índice dos vetores em 0, como C/C++, Java, etc. Dessa forma, a primeira posição do vetor é **sempre** referenciada pelo índice 0. Em outras linguagens, como Pascal, a primeira posição é geralmente o valor 1.

Vetores

- **Sintaxe em C++ (declaração de vetor):**

```
tipo_dado identificador_variável[quantidade_elementos];
```

Em que **quantidade_elementos** é um número inteiro positivo não-nulo que especifica a quantidade total de posições disponíveis para armazenamento no vetor. Exemplos:

```
int A[5]; //5 posições de inteiros  
float B[7]; //7 posições de números reais  
char C[20]; //20 posições de caracteres
```



Não colocar a **quantidade_elementos** entre os símbolos de colchetes, [e], ocasionará um erro de sintaxe.

Vetores

- **Sintaxe em C++ (declaração e inicialização de vetor):**

```
tipo_dado identificador_variável[quantidade_elementos] = {lista_valores};
```

- Um vetor pode ser inicializado colocando seus elementos entre símbolos de chaves, { e }. Exemplo:

```
int notas_alunos[5] = {90, 86, 54, 70, 60};
```

Neste exemplo, o valor **90** é armazenado no índice 0, **86** no índice 1, **54** no 2, **70** no 3 e **60** no índice 4 do vetor **notas_alunos**.

Vetores

- **Sintaxe em C++ (declaração e inicialização de vetor):**

```
tipo_dado identificador_variável[quantidade_elementos] = {lista_valores};
```

- Um vetor pode ser inicializado colocando seus elementos entre símbolos de chaves, { e }. Exemplo:

```
int notas_alunos[5] = {60, 45, 52};
```

Neste exemplo, as três primeiras posições do vetor recebem os valores **60**, **45** e **52**, respectivamente. As demais posições recebem o valor **0**.

Acesso a posições

- Em C/C++, um vetor de tamanho **N** tem suas posições de índices numeradas de **0** a **N-1**. Para que possamos acessar individualmente um único elemento dentro do vetor, precisamos apenas indicar o nome da variável e a posição (índice) de interesse a ser manipulada.
- **Sintaxe em C++ (acesso):**

Identificador_variável[posição de interesse]

Acesso a posições

- Exemplos:

```
a[5] = 23; //atribui valor 23
           //à posição 5 do vetor a
cout << b[3]; // imprime posição 3
           // do vetor b
```

Um erro muito comum é esquecer que os índices do vetor começam em 0, e que para acessar o último elemento tem que se subtrair uma unidade da quantidade total de elementos.

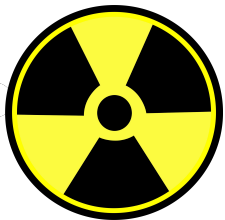
Acessar uma posição (índice) inválida de um vetor é um **erro** de programação **grave** que pode ocasionar a alteração da memória pertencente a uma outra variável que não seja o vetor de interesse.



Acesso indevido a posições

C e C++ **não impedem** o acesso fora da faixa de valores de índices estabelecida pelo vetor. Sendo assim, é possível acessar posições de memória antes e depois daquela reservada ao vetor, de forma incorreta.

Exemplo:



Este código compila sem erros, mas com resultados imprevisíveis.

```
#include <iostream>
using namespace std;
int main(){
    int vetor[5] = {1,2,3,4,5};
    //acesso indevido ao índice -1
    cout << vetor[-1] << endl;
    //acesso indevido ao índice 6
    cout << vetor[6] << endl;
    //acesso indevido ao índice 5
    cout << vetor[5] << endl;
    return 0;
}
```

Vetores e memória

Seja o vetor $v[10]$ de inteiros:

2	-3	101	104	0	1	2	3	-3	0
---	----	-----	-----	---	---	---	---	----	---

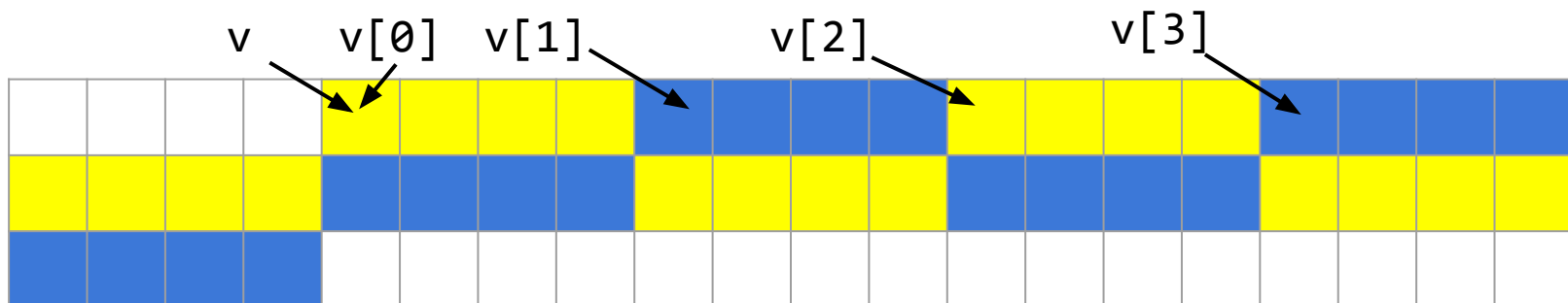
$v[0]$

Vetores e memória

2	-3	101	104	0	1	2	3	-3	0
---	----	-----	-----	---	---	---	---	----	---

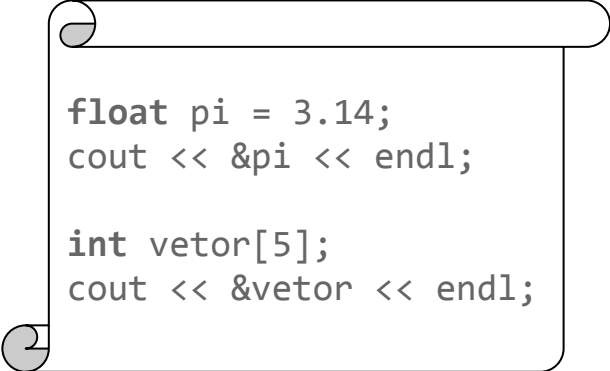
v[0]

Em memória (considerando 4 bytes por inteiro):



Vetores e memória

Em C/C++ é possível obter o endereço de memória de uma variável qualquer, independentemente de seu tipo, utilizando o operador **&**. Para isto, basta preceder o identificador da variável pelo símbolo **&**. Exemplos:



```
float pi = 3.14;  
cout << &pi << endl;  
  
int vetor[5];  
cout << &vetor << endl;
```

Vetores e memória

Em C/C++, o valor de um vetor é exatamente o seu endereço de memória (assim como o endereço do primeiro elemento armazenado no vetor).

```
#include <iostream>
using namespace std;
int main(){
    int vetor[5] = {1,2,3,4,5};
    cout << vetor << endl;
    cout << &vetor << endl;
    cout << &vetor[0] << endl;
    return 0;
}
```



Teste o programa deste exemplo, com e sem o operador **&** no terceiro **cout**. Compare o que será exibido em ambas as situações.

Vetores e memória

Em C/C++, o valor de um vetor é exatamente o seu endereço de memória (assim como o endereço do primeiro elemento armazenado no vetor).

```
#include <iostream>
using namespace std;
int main(){
    int vetor[5] = {1,2,3,4,5};
    cout << vetor << endl;
    cout << &vetor << endl;
    cout << &vetor[0] << endl;
    return 0;
}
```

```
[joukim@harpia tmp]$ ./vec
0x7ffd87dd81c0
0x7ffd87dd81c0
0x7ffd87dd81c0
[joukim@harpia tmp]$ ./vec
0x7ffe8cf649e0
0x7ffe8cf649e0
0x7ffe8cf649e0
```

Vetores e memória

Em C/C++, o valor de um vetor é exatamente o seu endereço de memória (assim como o endereço do primeiro elemento armazenado no vetor).

Como o valor de um vetor é um endereço de memória, ele é um ponteiro. Na verdade, um tipo especial de ponteiro (um ponteiro constante, que não muda seu endereço).

```
#include <iostream>
using namespace std;
int main(){
    int vetor[5] = {1,2,3,4,5};
    cout << vetor << endl;
    cout << &vetor << endl;
    cout << &vetor[0] << endl;
    return 0;
}
```

```
tmp]$ ./vec
```

```
[joukim@harpia tmp]$ ./vec
0x7ffe8cf649e0
0x7ffe8cf649e0
0x7ffe8cf649e0
```

Alocação de vetores em C/C++

Alguns compiladores exigem que o tamanho do vetor seja uma expressão constante.

Isto é, não é possível usar uma variável, ou uma fórmula envolvendo variáveis, para definir o tamanho do vetor. ***Este é o padrão especificado pela linguagem, inclusive.***

Nestes compiladores, **não** seria permitido algo do tipo:

```
int N = 100;  
int vetor[N];
```



Alocação de vetores em C/C++

Como o tamanho do vetor é utilizado em vários trechos de código, recomenda-se utilizar uma constante

```
#define TAMVETOR 100  
int vetor[TAMVETOR];
```



```
const int TAMVETOR = 100;  
int vetor[TAMVETOR];
```



A primeira opção é mais usada em códigos da linguagem C, enquanto a segunda é preferida em programas C++.

Alocação de vetores em C/C++

Versões mais recentes de vários compiladores aceitam declarações que permitam a atribuição e cálculo na determinação do tamanho do vetor.

Essa técnica, denominada VLA (*Variable Length Arrays*), possui limites de espaço, por usar um espaço de memória para variáveis temporárias e retorno de funções (chamado *stack*), ao invés do espaço de memória destinado a alocações dinâmicas de memória (chamado *heap*). Exemplo:

```
int N;  
cin >> N;  
int vetor[N];
```



Alocação de vetores em C/C++

Versões mais recentes de vários compiladores aceitam declarações de vetores com tamanho variável e cálculo na determinação do tamanho. Este método, apesar de ser mais fácil de usar, não é padrão e tem limitações de uso da memória.

Essa técnica (chamada *Variable Length Arrays*), possui limites de tamanho de memória para variáveis temporárias (chamado *stack*), ao invés do espaço de memória reservado para alocações dinâmicas.

Quando precisamos de vetores com tamanho variável, é melhor utilizar alocação dinâmica de memória. Por enquanto, iremos utilizar vetores com tamanhos pré-definidos.

```
int N;  
cin >> N;  
int vetor[N];
```



Repetições contadas



Repetições contadas

- As estruturas de repetição **while** e **do..while** são extremamente poderosas, uma vez que não precisam saber antecipadamente quantas vezes um determinado bloco de comandos será repetido.
- Em contrapartida, dependendo da situação, poderia ser pouco conveniente utilizar estas estruturas quando se sabe com exatidão quantas vezes um dado bloco será executado.
 - Por exemplo: escrever um programa que exiba a mensagem “Para aprender a programar, eu devo praticar.” cem vezes no dispositivo de saída padrão.

Repetições contadas

```
#include <iostream>
using namespace std;
int main(){
    int i = 0;
    string texto = "Para aprender a programar, eu devo praticar.";
    while (i < 100) {
        cout << texto << endl;
        i++;
    }
    return 0;
}
```

Este exemplo funciona perfeitamente bem. Contudo, em programação, há formas mais diretas de se construir esta repetição.

A maioria das linguagens de programação possui uma estrutura de repetição que permite executar um conjunto de comandos um número pré-estabelecido de vezes.

Esta estrutura é usualmente chamada de comando de repetição **para**.

Repetições contadas

```
#include <iostream>
using namespace std;
int main(){
    int i = 0;
    string texto = "Para aprender a programar, eu devo praticar.";
    while (i < 100) {
        cout << texto << endl;
        i++;
    }
    return 0;
}
```

Erro comum ao usar **while**
(ou **do..while**):
Esquecer de incrementar o
valor da variável contadora i.

Este exemplo funciona perfeitamente bem. Contudo, em programação, há formas mais diretas de se construir esta repetição.

A maioria das linguagens de programação possui uma estrutura de repetição que permite executar um conjunto de comandos um número pré-estabelecido de vezes.

Esta estrutura é usualmente chamada de comando de repetição **para**.

Repetições contadas

- Sintaxe em pseudocódigo:

Para **variável_controle** **De** **valor_inicial** **Até** **valor_final** **Faça**
 Bloco_de_repetição
Fim-Para



Em que:

- **variável_controle**: variável utilizada pela estrutura de repetição **para** para contar a quantidade de repetições executadas pelo laço.
- **valor_inicial**: valor que será inicialmente assumido pela **variável_controle** ao iniciar o laço de repetição.
- **valor_final**: último valor que a **variável_controle** irá assumir antes de encerrar a execução das repetições do laço.

Repetições contadas

- A inicialização da **variável_controle** é realizada implicitamente, com o **valor_inicial** informado na declaração da estrutura **para**.
- A condição para que a repetição ocorra é que o valor da **variável_controle** não tenha atingido o **valor_final** informado.
- Em termos práticos, a **variável_controle** é uma variável contadora.

Repetições contadas

- Ao final de cada iteração, o valor da **variável_controle** é incrementado em 1.
 - Para que a variável seja atualizada com valores diferentes de 1 é necessário informar explicitamente o valor a ser considerado no passo durante a declaração da estrutura **para**.
- Nesta estrutura não é necessário incrementar nem inicializar a **variável_controle**, isto é feito automaticamente na declaração.

Repetições contadas

- Sintaxe em C/C++:

```
for (inicialização; teste; atualização) {  
    Bloco_de_repetição  
}
```



Em que:

- **inicialização**: valor inicial da expressão ou variável de controle;
- **teste**: teste para verificar se repetição chegou ao fim, por meio do controle do valor da expressão ou variável de controle;
- **atualização**: atualização do valor da expressão ou variável de controle a cada iteração.

Repetições contadas

- Exemplo:

```
for (int i = 0; i < 100; i++) {  
    Bloco_de_repetição  
}
```



Em que:

- **inicialização**: inicia o valor de i em zero;
- **teste**: verifica se o valor de i é menor que 100;
- **atualização**: a cada passo, incrementa automaticamente o valor de i.

Repetições contadas

Todo comando **for** pode ser convertido para uma versão correspondente utilizando o comando **while**. Assim:

```
for (inicialização; teste; atualização) {  
    Bloco_de_repetição  
}
```

poderia ser convertido como:

```
inicialização;  
while (teste) {  
    Bloco_de_repetição  
    atualização;  
}
```


Repetições contadas



- Por causa desta característica (conversão de comandos **for** para **while**), formalmente C/C++ não possuem realmente laços controlados por contadores, mas apenas laços baseados em controladores lógicos. É possível, inclusive, utilizar o **for** do C/C++ em laços em que não há contadores.
- Um motivo para utilizar o **for** e não o comando **while** em repetições por contador é o fato de que o comando **for** permite um código mais simples, claro e com o uso de variáveis locais ao próprio comando.
- O fato de ser um laço controlado por lógica dá mais poder ao **for** do C/C++, permitindo com facilidade criar laços decrescentes, pulando de dois em dois, por exemplo, o que não é fácil de fazer com laços controlados por contadores de outras linguagens.

Repetições contadas - exemplo 1

Escrever um programa que exiba a mensagem “Para aprender a programar, eu preciso praticar.” cem vezes no dispositivo de saída padrão.

```
#include <iostream>
using namespace std;
int main(){
    string texto = “Para aprender a programar, eu devo praticar.”;
    for (int i = 0; i < 100; i++) {
        cout << texto << endl;
    }
    return 0;
}
```

Repetições contadas - exemplo 1

Escrever um programa que exiba a mensagem “Para aprender a programar, eu preciso praticar.” cem vezes no dispositivo de saída padrão.

```
#include <iostream>
using namespace std;
int main(){
    string texto = “Para aprender a programar, eu devo praticar.”;
    for (int i = 0; i < 100; i++) {
        cout << texto << endl;
    }
    return 0;
}
```

É mais difícil esquecer de incrementar a variável contador em um **for** que em um **while** (ou **do..while**).

Repetições contadas - exemplo 2

Vamos pensar em um problema simples: ler os dados de um vetor e calcular a média desses valores, sabendo que são números em ponto flutuante. Imprimir os elementos do vetor e a média encontrada. Sabe-se que no máximo o vetor conterà 20 elementos.

Repetições contadas - exemplo 2

```
#include <iostream>
using namespace std;
int main() {
    float vetor[20];
    int num, i;
    cin >> num;
    float soma = 0;
    for (i = 0; i < num; i++)
        cin >> vetor[i]; //leitura dos dados
    for (i = 0; i < num; i++)
        soma += vetor[i]; //somatório dos elementos
    for (i = 0; i < num; i++)
        cout << vetor[i] << " "; //impressão dos dados
    cout << endl;
    cout << "Média = " << soma/num << endl;
    return 0;
}
```

Repetições contadas - exemplo 2

```
#include <iostream>
using namespace std;
int main() {
    float vetor[20];
    int num, i;
    cin >> num;
    float soma = 0;
    for (i = 0; i < num; i++)
        cin >> vetor[i]; //leitura dos dados
    for (i = 0; i < num; i++)
        soma += vetor[i]; //somatório dos elementos
    for (i = 0; i < num; i++)
        cout << vetor[i] << " "; //impressão dos dados
    cout << endl;
    cout << "Média = " << soma/num << endl;
    return 0;
}
```

Note que apesar do vetor ter sido alocado com 20 posições, só será utilizada a quantidade de posições informada em num. Por isso o for não vai até 20 e sim até essa quantidade.

Repetições contadas - exemplo 2

```
#include <iostream>
using namespace std;
int main() {
    float vetor[20];
    int num;
    cin >> num;
    float soma = 0;
    for (i = 0; i < num; i++) {
        cin >> vetor[i]; //leitura dos dados
        soma += vetor[i]; //somatório dos elementos
    }
    for (i = 0; i < num; i++)
        cout << vetor[i] << " "; //impressão dos dados
    cout << endl;
    cout << "Média = " << soma/num << endl;
    return 0;
}
```

Otimização de código: calculando a soma à medida que vai lendo os dados.

Repetições contadas - exemplo 3

Escrever um programa que leia do dispositivo de entrada padrão 100 números inteiros que devem ser armazenados em um vetor.

Seu programa deve calcular e exibir a soma de todos os elementos pertencentes ao vetor que ocupam índices (posições) pares. Ou seja, deve-se calcular o somatório dos elementos que ocupam as posições 0, 2, 4, 6, 8, ..., e assim por diante, do vetor.

Repetições contadas - exemplo 3

```
#include <iostream>
using namespace std;
int main(){
    int const N = 100; //tamanho do vetor
    int numeros[N];
    int i, soma = 0;

    //leitura dos dados (todos os 100)
    for (i = 0; i < N; i++) cin >> numeros[i];

    //somatório dos elementos (posições pares)
    for (i = 0; i < N; i = i + 2) soma += numeros[i];

    cout << "Soma = " << soma << endl;
    return 0;
}
```

Repetições contadas - exemplo 3

```
#include <iostream>
using namespace std;
int main(){
    int const N = 100; //tamanho do vetor
    int numeros[N];
    int i, soma = 0;

    //leitura dos dados (todos os 100)
    for (i = 0; i < N; i++) cin >> numeros[i];

    //somatório dos elementos (posições pares)
    for (i = 0; i < N; i = i + 2) soma += numeros[i];

    cout << "Soma = " << soma << endl;
    return 0;
}
```

Bloco de comandos na mesma linha, por questões de espaço

$i = i + 2$
Poderia ser escrito como
 $i += 2$

Repetições para cada



Estrutura para cada

Mais recentemente linguagens de programação tem disponibilizado uma estrutura de repetição adicional, que permite percorrer todos os elementos em uma coleção, sem precisar de informações adicionais (como sentinelas ou contadores).

Também é conhecida como laço controlado por estrutura de dados (uma vez que permite percorrer todos os itens de uma dada estrutura).

Estrutura para cada

Para cada *elemento* em <coleção>

InícioRepita

Bloco_de_repetição

FimRepita

For (controlado por estrutura de dados) em C++

```
for (tipo elemento : coleção) {  
    sequência de comandos;  
}
```

```
for (tipo& elemento : coleção) {  
    sequência de comandos;  
}
```

For (controlado por estrutura de dados) em C++

```
for (tipo elemento : coleção) {  
    sequência de comandos;  
}
```

O '&' é necessário para alterar o elemento. Indica que será utilizada a própria variável e não uma cópia do valor.

```
for (tipo& elemento : coleção) {  
    sequência de comandos;  
}
```

Exemplo de uso do for (controlado por estrutura de dados)

```
string vetor[5] = {"joão",  
                  "maria", "pedro", "thiago",  
                  "carlos"};  
  
for (int i = 0; i < 5; i++) {  
    cout << vetor[i] ;  
}  
cout << endl;  
  
for (string t : vetor) {  
    t = "X" + t + "X";  
}
```

```
for (string& t : vetor) {  
    t = " " + t + " ";  
}  
for (string t : vetor) {  
    cout << t;  
}  
cout << endl;  
  
string teste = "Ola, mundo!";  
for (char& c: teste) {  
    cout << c << " ";  
}  
cout << endl;
```



Exemplo de uso do for (controlado por estrutura de dados)

```
string vetor[5] = {"joão",
```

```
"mar
```

```
"car
```

```
for (int i = 0; i < vetor.size(); i++) {  
    cout << vetor[i] << " ";  
}
```

```
cout << endl;
```

```
for (string t : vetor) {  
    t = "X" + t + "X";  
}
```

```
for (string& t : vetor) {  
    t = " " + t + " ";  
}
```

```
etor) {
```

```
[joukim@harpia tmp]$ ./foreach  
joão mariapedrothiagocarlos  
joão maria pedro thiago carlos  
O l a , m u n d o !  
[joukim@harpia tmp]$
```

```
string teste = "Ola, mundo!";  
for (char& c: teste) {  
    cout << c << " ";  
}  
cout << endl;
```

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/).