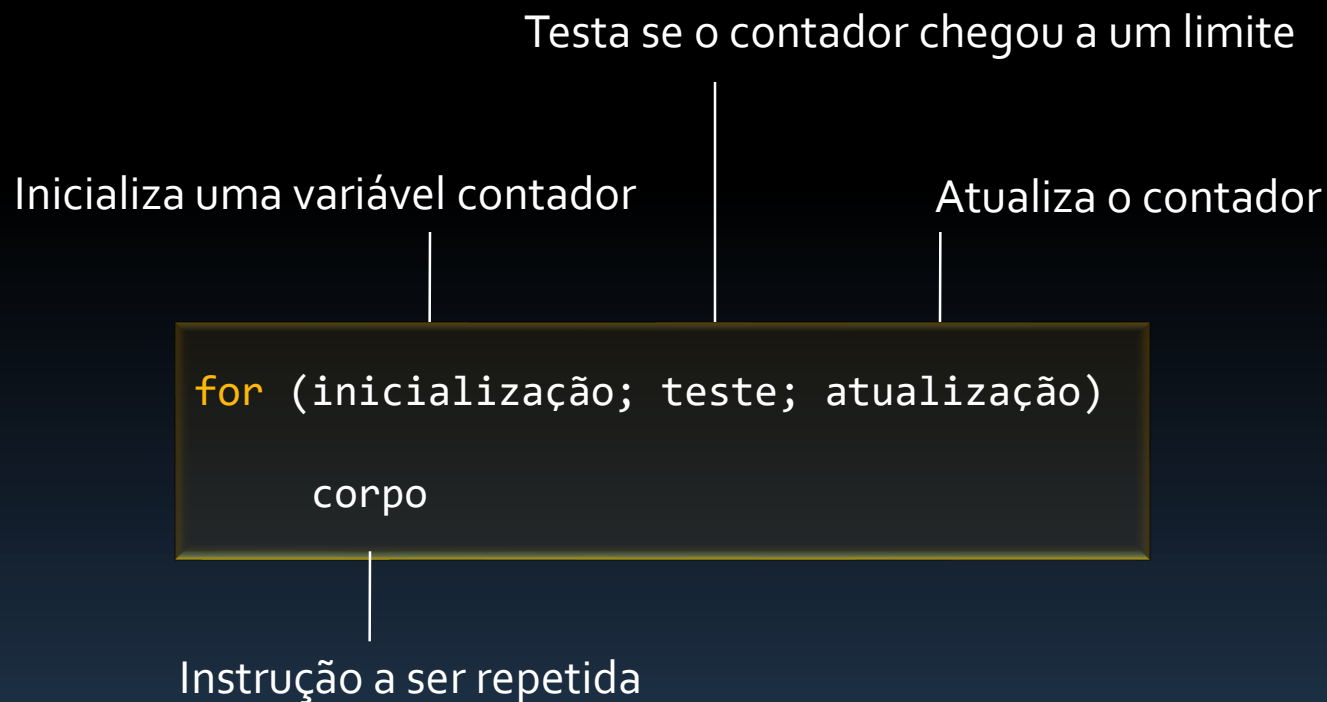


Programação de Computadores

# APLICAÇÕES DO LAÇO FOR

# Introdução

- As partes de um laço for



# Introdução

- O laço for permite realizar **tarefas repetitivas**

```
for (i = 0; i < 5; i++)  
    cout << "C++ sabe repetir\n";
```

- Ele é **frequentemente usado**:
  - Quando se conhece o número de repetições
  - Para processar elementos de um **vetor**

```
for (int i = 0; i < TamVet; i++)  
    cout << vet[i] << endl;
```

# Tamanho do Passo

- O **passo** é o valor adicionado (ou subtraído) da variável contador em cada repetição do laço for
- O **tamanho do passo** pode ser modificado através da expressão de atualização
  - Ao invés de usar **i++**, basta usar a expressão **i = i + k**, onde k é o tamanho do passo

```
for (int i=0; i<5; i=i+2)
    cout << "C++ sabe repetir.\n";
```

# Tamanho do Passo

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

int main()
{
    cout << "Entre com um valor inteiro: ";
    int passo;
    cin >> passo;

    cout << "Contando de " << passo << " em "
         << passo << ":\n";

    for (int i = 0; i < 100; i = i + passo)
        cout << i << endl;

    return 0;
}
```

# Tamanho do Passo

- A saída do programa:

Entre com um valor inteiro: 30

Contando de 30 em 30:

0

30

60

90

- A atualização pode ser uma expressão qualquer:

```
i = i*i + 10*i + 5;
```

# Atribuição Combinada

- C++ possui operadores que servem como **atalhos** para as principais **operações combinadas com atribuição**

## Aritméticas

```
a += 4;    // a = a + 4;  
b -= 3;    // b = b - 3;  
c *= 2;    // c = c * 2;  
d /= 5;    // d = d / 5;  
e %= 2;    // e = e % 2;
```

## Binárias

```
f >>= 1;   // f = f >> 1;  
g <<= 1;   // g = g << 1;  
h &= 2;    // h = h & 2;  
i |= 4;    // i = i | 4;  
j ^= 8;    // j = j ^ 8;
```

```
int * vet = new int[10];
```

```
for (int i=0; i<10; i+=2)  
    cin >> vet[i];
```

```
for (int i=1; i<10; i+=2)  
    vet[i] = 0;
```

```
delete [] vet;
```

# Laço for com Strings

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

int main()
{
    cout << "Entre com uma palavra: ";
    char palavra[20];
    cin >> palavra;

    // mostra as letras na ordem inversa
    for (int i = strlen(palavra)-1; i >= 0; i--)
        cout << palavra[i];
    cout << "\nTchau.\n";

    return 0;
}
```

0	a	0xCB20	palavra
1	n	0xCB21	
2	i	0xCB22	
3	m	0xCB23	
4	a	0xCB24	
5	l	0xCB25	
6	\0	0xCB26	
7		0xCB27	
8		0xCB28	
	...		
19			



# Laço for com Strings

- A saída do programa:

```
Entre com uma palavra: animal  
lamina  
Tchau.
```

- Este código é o ponto de partida para um programa que descobre se uma palavra é um palíndromo
  - Basta comparar a palavra original com sua versão invertida usando a função `strcmp()`\*

# Incremento e Decremento

- O operador ++ deu origem ao nome da linguagem
  - C++ significa "C incrementado"
- Os operadores de incremento e decremento podem ser usados fora de um laço for

```
int i;  
for (i = 5; i > 0; i--)  
    cout << palavra;  
i++;  
cout << i << endl;
```

# Incremento e Decremento

- Os operadores de incremento (++) e decremento (--) possuem **duas versões**:
  - Prefixo  
**Ex.:** --x, ++x
  - Sufixo  
**Ex.:** x++, x--
- As duas versões tem o mesmo efeito sobre o operando, mas elas **diferem no momento de atuação**

# Incremento e Decremento

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

int main()
{
    int a = 20;
    int b = 20;

    cout << "  a = " << a << ": b  = " << b << "\n";
    cout << "++a = " << ++a << ": b++ = " << b++ << "\n";
    cout << "  a = " << a << ": b  = " << b << "\n";

    return 0;
}
```

# Incremento e Decremento

- A saída do programa:

```
a = 20:  b  = 20
++a = 21: b++ = 20
a = 21:  b   = 21
```

- A notação `++x` significa:

- Primeiro incremente
- Depois use o novo valor

- A notação `x++` significa:

- Use o valor atual
- Depois incremente

# Incremento e Decremento

- Exemplos:

```
int x = 5;  
int y = ++x;    // incrementa x, depois atribui a y
```

```
int z = 5;  
int y = z++;    // atribui a y, depois incrementa z
```

- Não use o incremento/decremento para a mesma variável na mesma instrução

```
y = (4 + x++) + (6 + ++x);    // resultado imprevisível
```

# Blocos de Instruções

- A sintaxe do laço for pode parecer restritiva porque o corpo do laço deve ser uma única instrução

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

- Isso não é um problema porque C++ oferece a possibilidade de criar blocos de instruções usando um par de chaves {}
  - ▣ Qualquer instrução pode ser substituída por um bloco

# Blocos de Instruções

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

int main()
{
    cout << "Entre com cinco valores:\n";
    double numero;
    double soma = 0.0;
    for (int i = 1; i <= 5; ++i)
    {
        cout << "Valor " << i << ": ";
        cin >> numero;
        soma += numero;    // soma = soma + numero;
    }
    cout << "A soma é " << soma << "\n";
    cout << "A média é " << soma / 5 << "\n";
}
```



# Blocos de Instruções

- A saída do programa:

Entre com cinco valores:

Valor 1: 50

Valor 2: 46

Valor 3: 32

Valor 4: 90

Valor 5: 80

A soma é 298

A média é 59.6

- A indentação não define um bloco

```
for (int i = 1; i <= 5; ++i)
    cout << "Valor " << i << ": "; // laço acaba aqui
    cin >> numero;                  // após o laço
    soma += numero;
```

# Blocos de Instruções

- Um bloco define um **escopo local** para as **variáveis**

```
#include <iostream>
int main()
{
    int x = 20;

    {
        int y = 100;
        std::cout << x << std::endl; // ok
        std::cout << y << std::endl; // ok
    }

    std::cout << x << std::endl; // ok
    std::cout << y << std::endl; // erro, y não é conhecido
}
```

# Blocos de Instruções

- Ao usar **variáveis com o mesmo nome** dentro e fora de um bloco, a nova esconde a antiga

```
#include <iostream>
int main()
{
    int x = 20;                // x original

    {
        std::cout << x << std::endl; // usa x original
        int x = 100;              // novo x
        std::cout << x << std::endl; // usa novo x
    }

    std::cout << x << std::endl; // usa x original
}
```

# Operador Vírgula

- Um **bloco** permite usar várias instruções onde a sintaxe da linguagem C++ só aceita uma
- O **operador vírgula** faz algo semelhante:
  - Permite usar várias expressões onde a sintaxe só aceita uma

	Inicialização	Teste	Atualização
<code>for (int i=0, j=4; i &lt;= 4; i++, j--)</code>			
<code>cout &lt;&lt; i &lt;&lt; " != " &lt;&lt; j &lt;&lt; endl;</code>			

i	0	1	2	3	4	5
j	4	3	2	1	0	-1

# Operador Vírgula

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    cout << "Entre com uma palavra: ";
    char palavra[20];
    cin >> palavra;

    // inverte string, modificando a string original
    char temp;
    int i, j;
    for (j=0, i=strlen(palavra)-1; j<i; --i, ++j)
    {
        temp = palavra[i];
        palavra[i] = palavra[j];
        palavra[j] = temp;
    }
    cout << palavra << "\nPronto.\n";
}
```

j

0	1	2	3	4	5
r	e	v	i	v	\0
0	1	2	3	4	5

i

e temp

# Operador Vírgula

- A saída do programa:

Entre com uma palavra: **viver**  
**reviv**  
Pronto.

- O **valor de uma expressão** com o operador vírgula é sempre o valor mais à direita:

```
int gatos;
```

```
cout << ((gatos = 170),240,380); // gatos = 170, a expressão = 380
```

```
cout << (gatos = (170,240,380)); // gatos = 380, a expressão = 380
```

```
cout << (gatos = 170,240,380); // gatos = 170, a expressão = 380  
// a atribuição tem prioridade
```

# Laços Aninhados

- Os **laços** podem ser **aninhados**
  - Cria uma repetição dentro de outra

```
for (int i = 0; i < 2; i++)  
{
```

```
    for (int j = 0; j < 3; j++)  
    {  
        cout << "C++ sabe repetir\n";  
    }  
  
    cout << endl;
```

```
}
```

```
C++ sabe repetir  
C++ sabe repetir  
C++ sabe repetir
```

```
C++ sabe repetir  
C++ sabe repetir  
C++ sabe repetir
```

# Laços Aninhados

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

int main()
{
    cout << "Tabuada do Vezes\n\n";

    for (int i = 1; i <= 10; ++i)
    {
        for (int j = 1; j <= 10; ++j)
            cout << i << " x " << j << " = " << i * j << "\n";

        cout << endl;
    }

    return 0;
}
```



# Laços Aninhados

- A saída do programa:

Tabuada do Vezes

1 x 1 = 1	2 x 1 = 2	3 x 1 = 3	4 x 1 = 4	5 x 1 = 5	6 x 1 = 6	7 x 1 = 7	8 x 1 = 8	9 x 1 = 9	10 x 1 = 10
1 x 2 = 2	2 x 2 = 4	3 x 2 = 6	4 x 2 = 8	5 x 2 = 10	6 x 2 = 12	7 x 2 = 14	8 x 2 = 16	9 x 2 = 18	10 x 2 = 20
1 x 3 = 3	2 x 3 = 6	3 x 3 = 9	4 x 3 = 12	5 x 3 = 15	6 x 3 = 18	7 x 3 = 21	8 x 3 = 24	9 x 3 = 27	10 x 3 = 30
1 x 4 = 4	2 x 4 = 8	3 x 4 = 12	4 x 4 = 16	5 x 4 = 20	6 x 4 = 24	7 x 4 = 28	8 x 4 = 32	9 x 4 = 36	10 x 4 = 40
1 x 5 = 5	2 x 5 = 10	3 x 5 = 15	4 x 5 = 20	5 x 5 = 25	6 x 5 = 30	7 x 5 = 35	8 x 5 = 40	9 x 5 = 45	10 x 5 = 50
1 x 6 = 6	2 x 6 = 12	3 x 6 = 18	4 x 6 = 24	5 x 6 = 30	6 x 6 = 36	7 x 6 = 42	8 x 6 = 48	9 x 6 = 54	10 x 6 = 60
1 x 7 = 7	2 x 7 = 14	3 x 7 = 21	4 x 7 = 28	5 x 7 = 35	6 x 7 = 42	7 x 7 = 49	8 x 7 = 56	9 x 7 = 63	10 x 7 = 70
1 x 8 = 8	2 x 8 = 16	3 x 8 = 24	4 x 8 = 32	5 x 8 = 40	6 x 8 = 48	7 x 8 = 56	8 x 8 = 64	9 x 8 = 72	10 x 8 = 80
1 x 9 = 9	2 x 9 = 18	3 x 9 = 27	4 x 9 = 36	5 x 9 = 45	6 x 9 = 54	7 x 9 = 63	8 x 9 = 72	9 x 9 = 81	10 x 9 = 90
1 x 10 = 10	2 x 10 = 20	3 x 10 = 30	4 x 10 = 40	5 x 10 = 50	6 x 10 = 60	7 x 10 = 70	8 x 10 = 80	9 x 10 = 90	10 x 10 = 100

- O resultado não é exibido em **colunas**, mas poderia:

```
for (int i = 1; i <= 10; ++i)
{
    for (int j = 1; j <= 10; ++j)
        cout << j << " x " << i << " = " << i * j << "\t"; // trocando "\n" por "\t"
    cout << endl; // trocando i e j
}
```

# Laço for para Sequências

- O padrão C++11 introduziu uma forma de **laço for** que **simplifica a tarefa de percorrer todos os elementos de um vetor**

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

int main()
{
    int vet[5] = { 3, 5, 6, 7, 9 };

    for (int n : vet)
        cout << n << " ";
    cout << endl;
}
```

# Laço for para Sequências

- O padrão C++20 introduziu a possibilidade de realizar uma **inicialização** antes de percorrer os elementos
  - Requer um compilador compatível com C++20

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

int main()
{
    int vet[5] = { 3, 5, 6, 7, 9 };

    for (char ch = 'a'; int n : vet)
        cout << ch++ << ": " << n << "\n";
}
```

# Resumo

- O laço for é **comumente usado** em três cenários:

- Processar elementos individuais de um **vetor**

```
for (int i = 0; i < TamVet; i++)  
    cout << vet[i] << endl;
```

- Processar elementos de uma **string**

```
for (int i = 0; i < strlen(palavra); i++)  
    cout << palavra[i];
```

```
// cout << palavra;  
for (int i = 0; palavra[i]; i++)  
    cout << palavra[i];
```

# Resumo

- Acumular/Somar valores

```
int numero;  
int soma = 0;  
for (int i = 0; i < 5; i++)  
{  
    cin >> numero;  
    soma += numero;  
}
```

- De uma forma geral, ele é usado para executar **tarefas repetitivas**, principalmente quando se conhece o número de repetições