BCC 201 - Introdução à Programação I

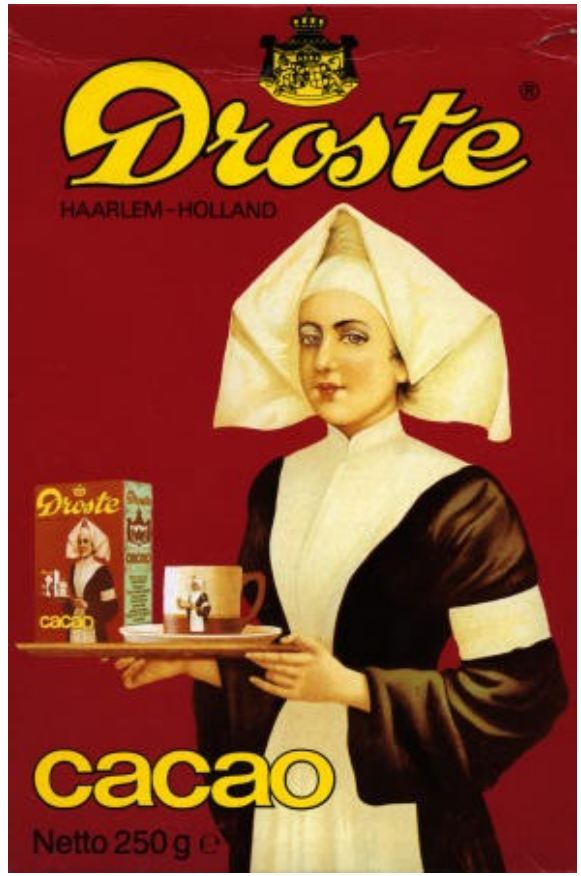
Recursividade

Guillermo Cámara-Chávez
UFOP

Introdução I

- Recursividade: é um método de programação no qual uma função pode chamar a si mesma
- Muitas estruturas têm natureza recursiva:
  - Estruturas encadeadas
  - Fatorial, serie Fibonacci
  - Uma pasta que contém outras pastas e arquivos

Introdução II



Uma forma visual de recursividade conhecida como efeito Droste

Introdução III

Introdução IV

- Recursão matemática
  - Como definir recursivamente a seguinte soma?

$$\sum_{k=m}^{n} k = m + (m+1) + \ldots + (n-1) + n$$

Introdução V

- Primeira definição recursiva

$$\sum_{k=m}^{n} k \begin{cases} m & \text{se } n = m \\ n + \sum_{k=m}^{n-1} k & \text{se } n > m \end{cases}$$

Introdução VI

- Recursão na computação

```
float soma(float m, float n);
int main(){
    float s;
    s = soma(5, 8);
    cout << s;
    return 0;
}
float soma(float m, float n)
{
    if (m == n)
        return m;
    else
        return (n + soma(m, n-1));
}
```

Na tela é mostrado:
26

Pilha de execução I

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
S = soma(5, 8);
```

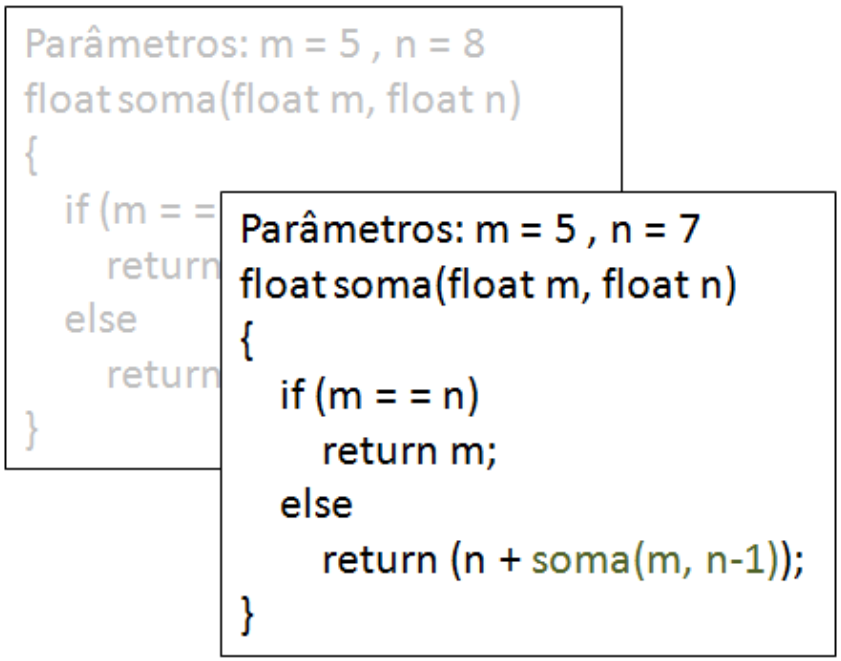
Pilha de execução II

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
Parâmetros: m = 5 , n = 8
float soma(float m, float n)
{
   if (m = = n)
      return m;
   else
      return (n + soma(m, n-1));
}
```

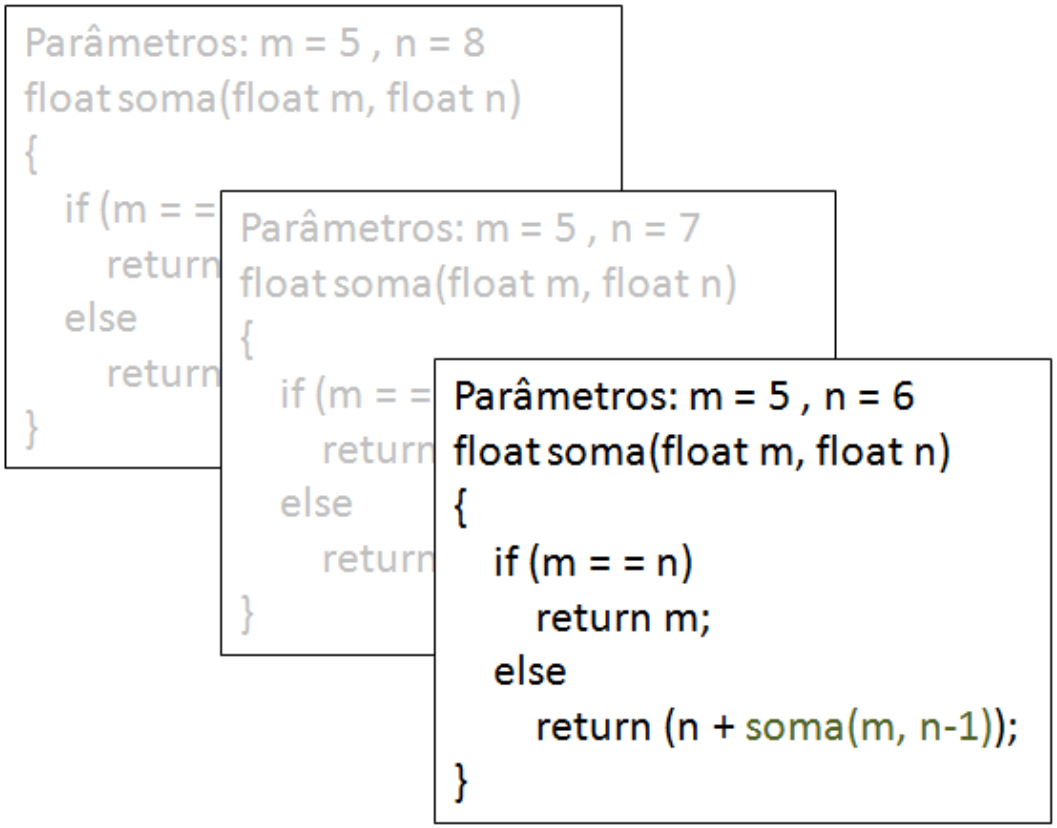
Pilha de execução III

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



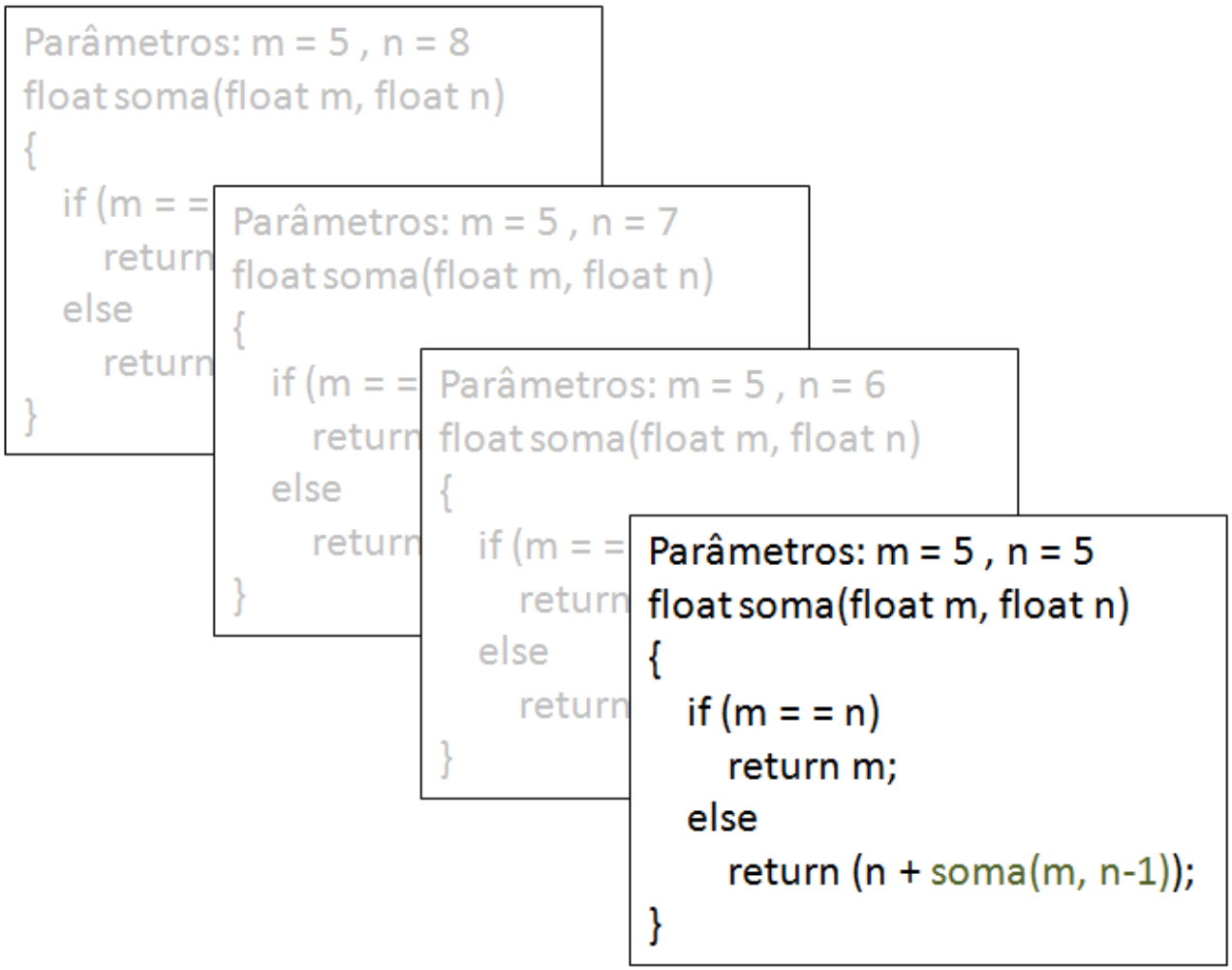
Pilha de execução IV

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



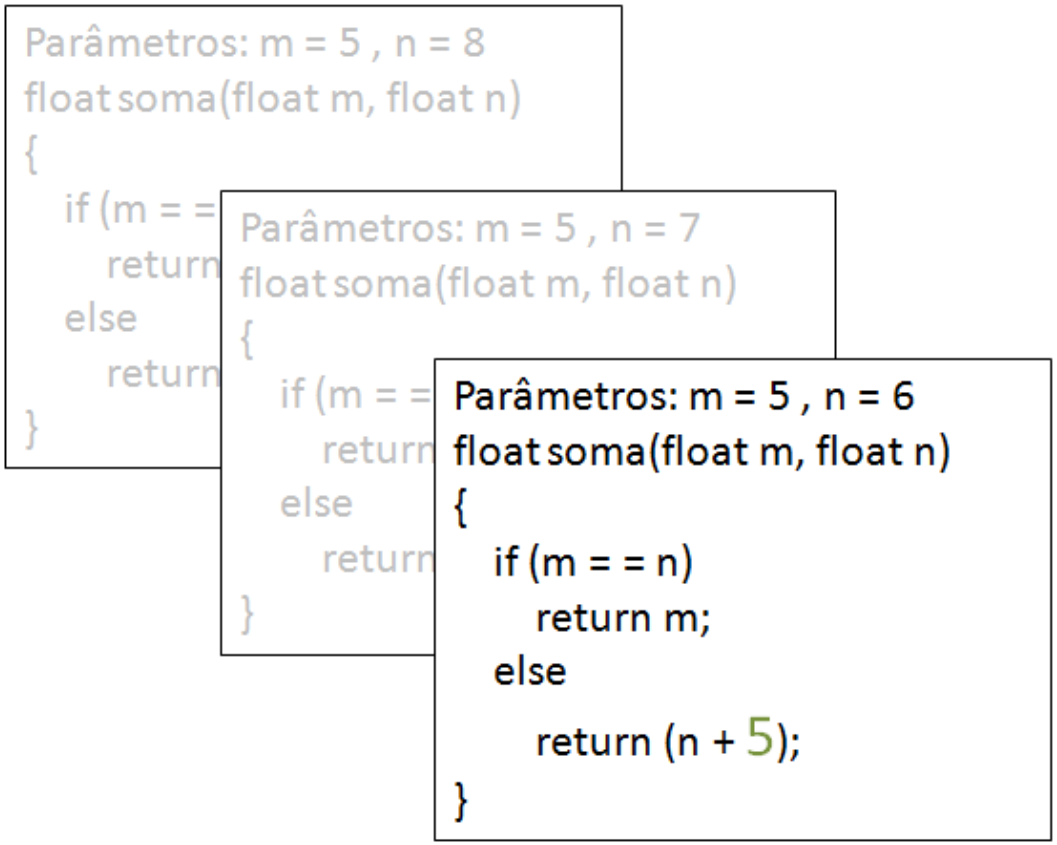
Pilha de execução V

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



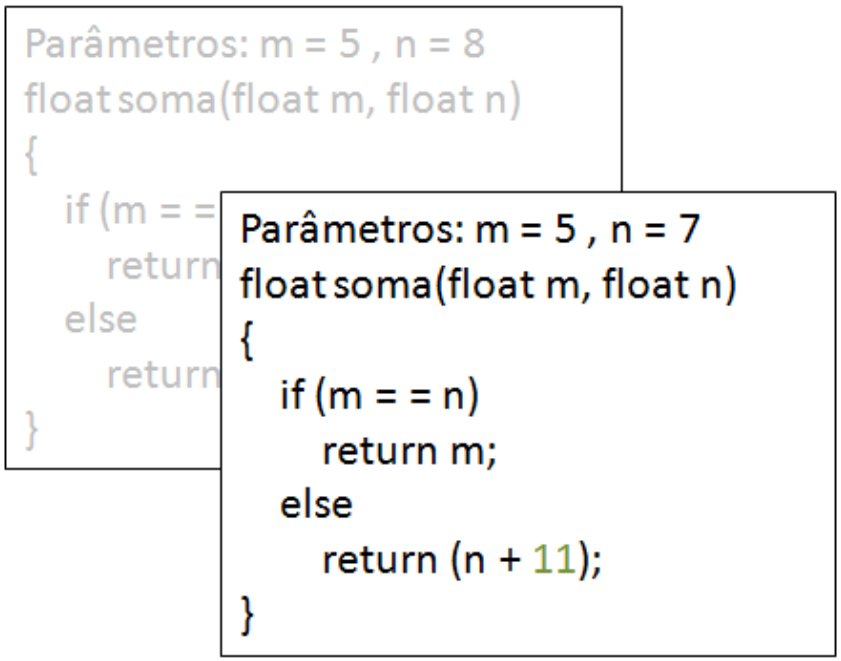
Pilha de execução VI

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



Pilha de execução VII

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



Pilha de execução VIII

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
Parâmetros: m = 5 , n = 8
float soma(float m, float n)
{
   if (m = = n)
      return m;
   else
      return (n +18);
}
```

Pilha de execução IX

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

S = 26

Estouro de pilha de execução I

- O que acontece se a função não tiver um caso base (ponto de parada)?
- O sistema de execução não consegue implementar infinitas chamadas

Fatorial I

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n * (n-1)! & \text{se } n > 0 \end{cases}$$

Fatorial II

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Potência I

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x * x^{n-1} & \text{se } n > 0 \end{cases}$$

Potência II

```
double potencia(double x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * potencia(x, n-1);
}
```

Exemplos I

O máximo divisor comum de dois números inteiros positivos pode ser calculado, utilizando o método de Euclides, cujo algoritmo é dado pela seguinte relação de recorrência:

$$mcd(m,n) = \begin{cases} m, & \text{se } n = 0 \\ mcd(n, m\%n), & \text{se } n \neq 0 \end{cases}$$

Exemplos II

```
int mdc(int m, int n)
{
    if (n == 0)
        return m;
    else
        return mdc(n, m % n);
}
```

Exemplos III

Escrever um programa que leia do teclado dois números inteiros positivos que correspondem ao numerador e denominador de uma fracção, permita reduzir a fracção e escreva no monitor a fracção reduzida assim como o seu quociente.

Exemplos IV

```
int mdc(int , int );
void reduzir(int , int , int&, int&);
int main()
{
    int num,den, n_num, n_den;
    cout << "Inserir fraccao: numerador, denominador \n";
    cin >> num >> den;
    reduzir(num, den, n_num, n_den);
    cout << num <<"/"<< den <<"=" << n_num <<"/"<< n_den;
    return 0;
}
```

Exemplos V

```
void reduzir(int num, int den, int& n_num, int& n_den)
{
    int fator = mdc (num, den);
    n_num = n_num / fator;
    n_den = n_den / fator;
}
int mdc(int m, int n)
{
    if (n == 0)
        return m;
    else
        return mdc(n, m  % n);
}
```

Exemplos VI

Encontrar a soma dos elementos de um vetor

Exemplos VII

```
int soma(int *vet, int n);
int main()
{
    int A[5] = {3, 5, 8, 23, 9};
    cout << soma(A, 5);
    return 0;
}
int soma(int *vet, int n)
{
    if (n == 1)
        return vet[0];
    else
        return vet[n-1] + soma(vet, n-1);
}
```

Standard Template Library (STL) I

- A biblioteca STL contem:
  - contenedor (*container*)
  - algoritmos (*algorithm*)
  - iteradores (*iterator*)
- Um contenedor é a forma como um conjunto de dados armazenados está organizado em memória
- Os algoritmos são procedimentos que são aplicados a contenedor para processar os dados
- Um iterador é um tipo de ponteiro que aponta elementos dentro de um contenedor

Contenedores I

- Um contenedor é uma forma de armazenar os dados
- O STL provê vários tipos de contenedores
  - <vector> : vetor unidimensional
  - <list>: lista duplamente encadeada
  - <deque>: fila de duas pontas
  - <queue>: fila
  - <stack>: pilha

Contenedores sequencias I

- Um contenedor sequencial armazena um conjunto de dados em sequência, `<vector>`, `<list>`, `<dequeue>` são contenedores sequências
- Em um vetor tradicional C++ o tamanho é fixo e não pode mudar durante a execução. Além disso, é tedioso inserir e apagar elementos. Vantagem: rápido acesso

Contenedores sequencias II

- Um `<vector>` é como um arranjo comum nos seguintes casos:
  - Um `<vector>` contem uma sequência de valores ou elementos
  - Um `<vector>` armazena seus dados em forma sequêncial
  - Pode ser usado o operador [] para acessar elementos individuais

Contenedores sequencias III

- Existem várias vantagens com respeito aos vetores comuns
  - Não é necessário declarar o número de elementos
  - Se é adicionado um novo elemento num `<vector>` que se encontra cheio, automáticamente incrementa de tamanho
  - Os `<vector>` podem retornar como resposta a quantidade de elementos que eles contêm

Declarando um *vector* I

- Deve ser incluída a biblioteca `#include<vector>`
- Logo deve ser declarado um objeto do tipo `<vector>`

```
vector<int> numeros;
```

- Declara `numeros` como um vetor de inteiros

Declarando um *vector* II

- Também pode se declarar um vetor com um tamanho inicial

```
vector<int> numeros(10);
```

Declaração	**Descrição**
`vector<float> V`	Declara V como um vetor de `float` vazio
`vector<int> V(15)`	Declara V como um vetor de `int` de 15 elementos
`vector<char> V(25,'A')`	Declara V como um vetor de 15 caracteres, todos inicializados com 'A'
`vector<double> V(B)`	Declara V como um vetor de `double`. Todos os valores de B, também `double` são copiados em V

Recuperando e salvando dados em um *vector* I

- Para acessar um valor em um posição existente dentro de vetor, pode ser usado o operador []

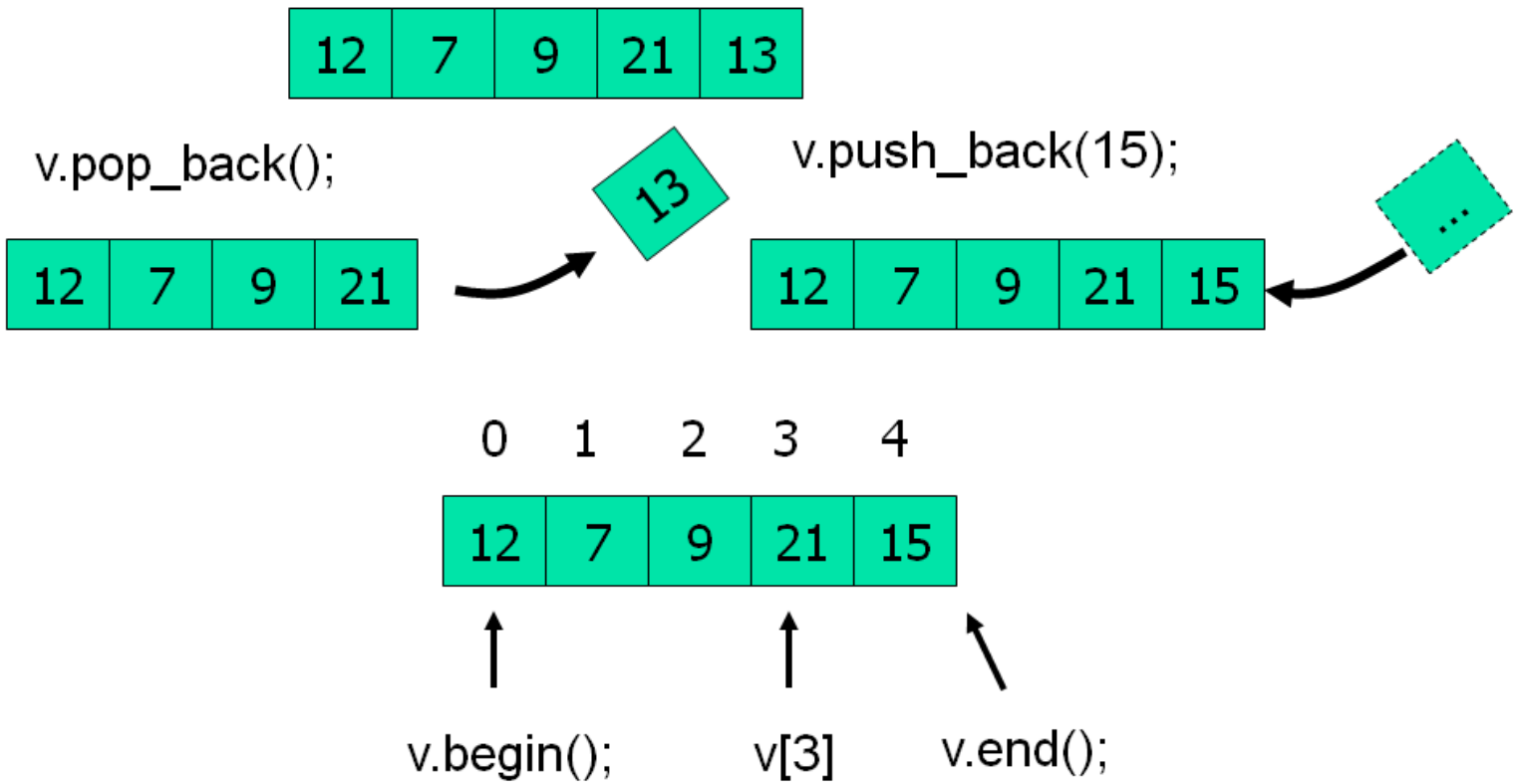
```
#include <vector>
int main(){
    vector<int> V(5);
    for (int i = 0; i < 5; i++)
    {
        cout << "Digite um valor";
        cin >> V[i];
    }
    for (int i = 0; i < 5; i++)
    {
        cout << V[i] << " ";
    }
    return 0;
}
```

Métodos (funções) implementadas na class *vector* I

- `size()`: retorna o número de elementos
- `resize()`: modifica o tamanho alocado
- `empty()`: confere se o vetor está vazio
- `begin()` e `end()`: iteram o vetor
- `at(i)`: acessa a posição i
- `front()` e `back()`: acessam o primeiro e o último elementos
- `erase()`: remove um elemento específico
- `push_back()`: adiciona o elemento T no fim
- `pop_back()`: remove o elemento do fim
- `insert(i,T);` adiciona um elemento T na posição i, empurrando os elementos
- `clear()`: limpa todo o vetor, deixando vazio

Usando a função *push_back* e *pop_back* I

```
int array[5] = {12, 7, 9, 21, 13};
vector<int> V(array, array+5);
```



Usando a função *push_back* e *pop_back* II

- O operador [] não pode ser usado para acessar um elemento que não existe
- Para inserir um valor em um <vector> vazio, ou se ele encontra-se cheio, usamos a função `push_back`

```
V.push_back(25);
```

Usando a função *push_back* e *pop_back* III

```
#include <vector>
int main(){
    vector<double> V;
    double tmp;
    for (int i = 0; i < 5; i++)
    {
        cout << "Digite um valor: ";
        cin >> tmp;
        V.push_back(tmp);
    }
    for (int i = 0; i < V.size(); i++)
    {
        cout << V[i]; // V.at(i)
    }
    return 0;
}
```

Usando a função *push_back* e *pop_back* IV

```
#include <vector>
#include <iostream>
int main(){
    // vetor padrao C
    int arr[] = { 12, 3, 17, 8 };
    // inicializa v com o vetor arr
    vector<int> v(arr, arr+4);
    // enquanto não esteja vazio
    while ( !v.empty())
    {
        // mostra o ultimo elemento
        cout << v.back() << " ";
        // apaga o ultimo elemento
        v.pop_back();
    }
    cout << endl
    return 0;
}
```

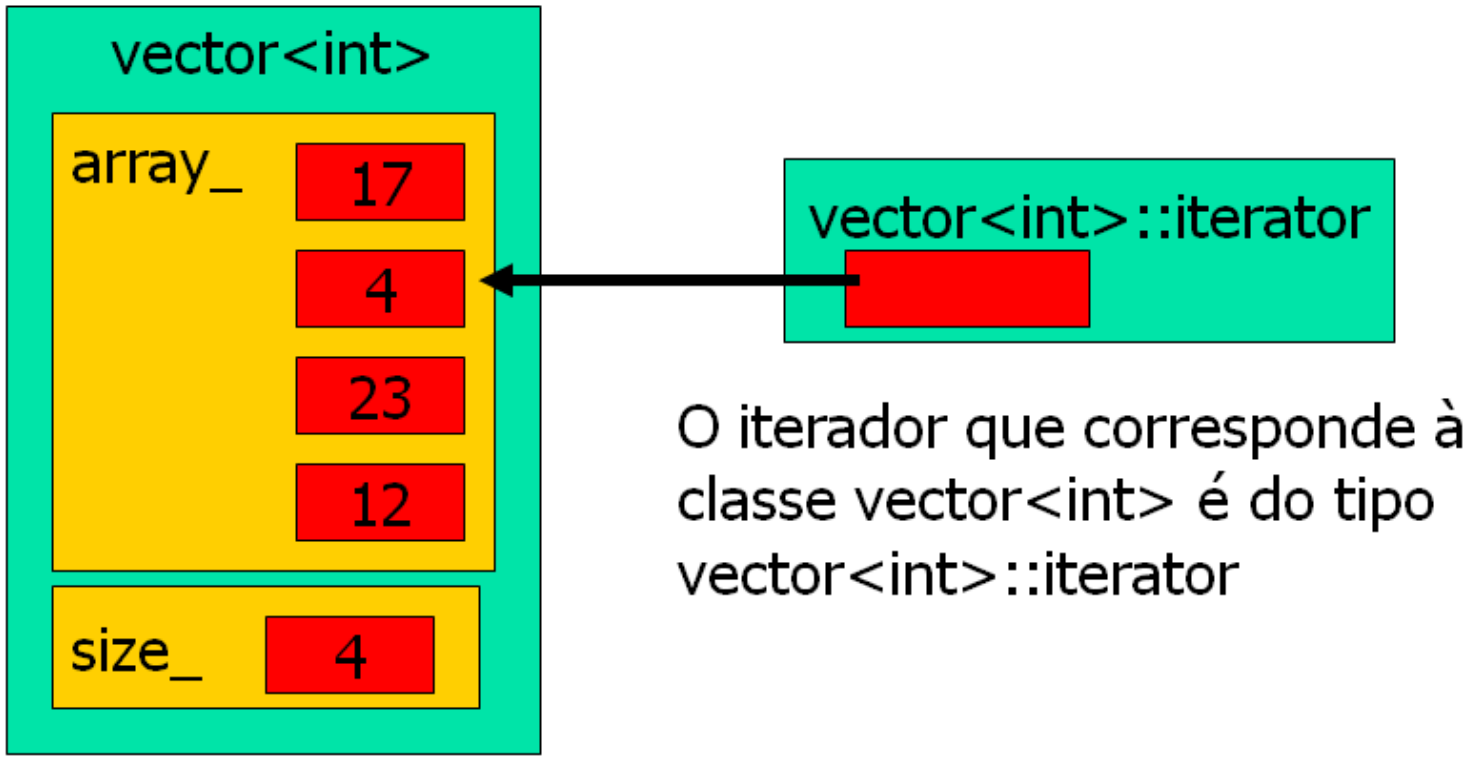
Usando a função *push_back* e *pop_back* V

```
template <typename T>
void Mostrar(vector<T>& v){
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}
int main(){
    int n[] = { 4, 6, 7, 9, 1 };
    vector<int> v(n,n+5);
    cout << "Tradicional \n"; Mostrar(v);
    cout << "\n Com o operador at \n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v.at(i) << " ";

    v.erase(v.begin() + 2); cout << "\n Removendo: \n" ;
    Mostrar(v);

    v.insert(v.begin() + 2, 200); cout << "\n  Inserindo \n"
    Mostrar(v); return 0;
}
```

Iteradores I



O iterador que corresponde à classe vector<int> é do tipo vector<int>::iterator

Iteradores II

- Permite acessar sequêncialmente elementos em um contenedor
- É uma generalização do ponteiro
- Deve permitir:
  - mover para o inicio
  - avançar para o próximo elemento
  - retornar o valor referenciado
  - verificar se está no final

Iteradores III

```
int main(){
    int n[] = { 4, 6, 7, 9, 1 };
    vector<int> v(n,n+5);

    cout << "Mostrando o vetor";
    for (vector<int>::iterator i = v.begin();
         i != v.end(); i++)
        cout << *i;

    return 0;
}
```

Instrução for com base em intervalo I

Para realizar operações com cada um dos elementos de um `<vector>`, a forma mais segura de realizar é através do uso de um comando introduzido no novo padrão: comando for com base em intervalo

```
for (declaracao : expressao)
    comando
```

onde `expressao` é um objeto que representa uma sequência e `declaracao` define uma variável que será usada para acessar a sequência

Instrução for com base em intervalo II

```
int main(){
    int n[] = { 4, 6, 7, 9, 1 };
    vector<int> v(n,n+5);

    cout << "Mostrando o vetor";
    for (auto i : v)
        cout << i;

    return 0;
}
```

FIM