

# INF 112 - Programação II

## TADs (Parte 2)

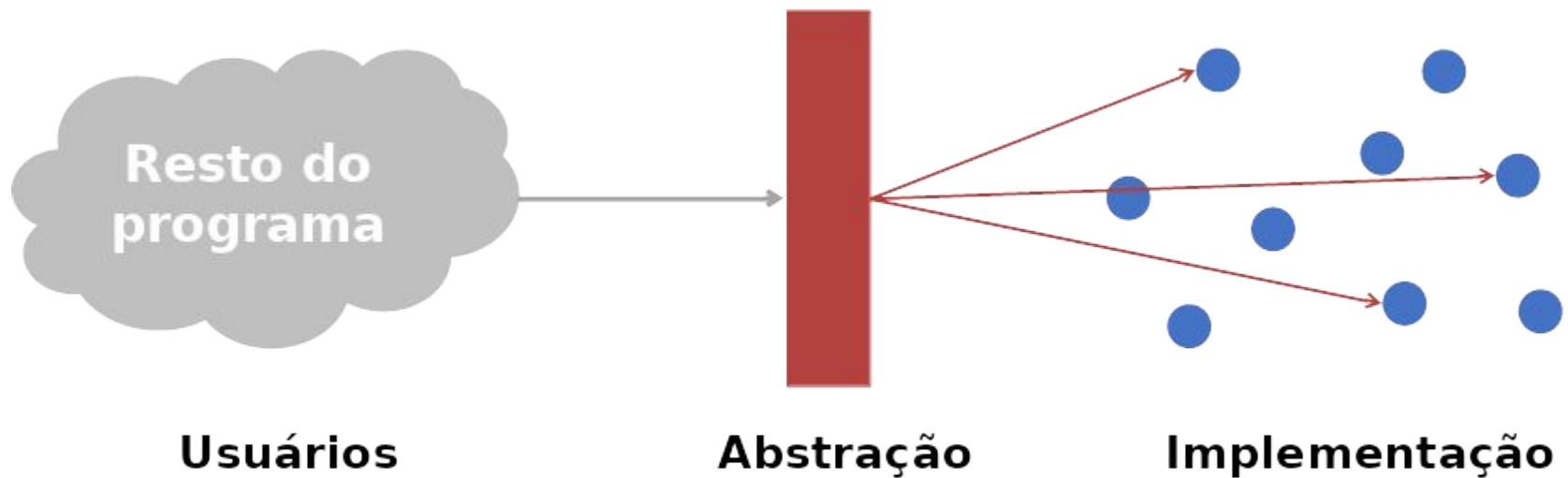
---

# Projeto Modular

---

# Lembrando do nosso objetivo

Com TADs queremos que o resto do programa seja cliente. Apenas use as operações do mesmo.



# Projeto Modular

## Propriedades

- Decomposição
- Composição
- Significado fechado
- Continuidade
- Proteção

# Projeto Modular

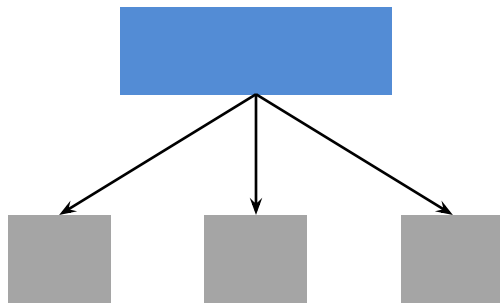
## Decomposição

- **Nível de Projeto**
  - Capaz de separar uma tarefa em subtarefas, que podem ser abordadas separadamente
- **Nível de Software**
  - Capaz de trabalhar em cada um dos módulos do software independente do outros módulos

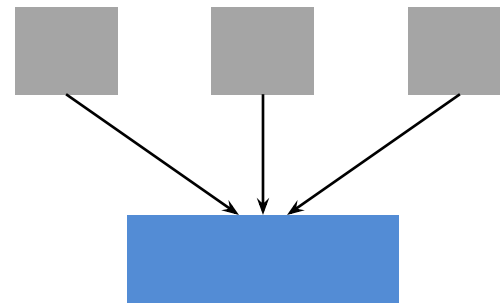
# Projeto Modular

## Composição

- Capacidade de conseguir combinar de forma livre diferentes elementos de software



Decomposição

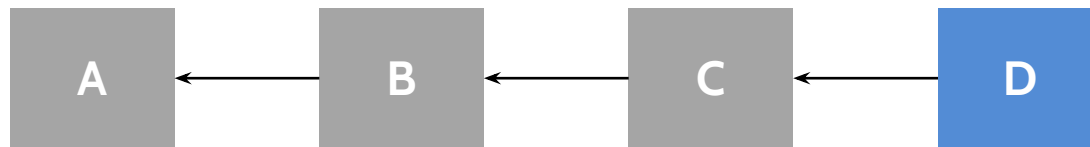


Composição

# Projeto Modular

## Significado fechado

- O programa deve ser compreendido por um leitor (usuário) que não possui acesso a outras (ou todas) partes do sistema



Problema: dependência sequencial.

# Projeto Modular

## Continuidade

- Alterações em parte da especificação demandam alterações em poucos módulos
- Bom exemplo
  - Utilização de constantes
- Mau exemplo
  - Dependência forte de um único módulo



# Projeto Modular

## Proteção

- Situações anormais em tempo de execução não são propagadas para outros módulos
  - Erros não detectados em outras partes
- Extensibilidade
- Validação dos dados nos módulos
  - Tipos, asserções, exceções

# Modularizando um TAD simples

---

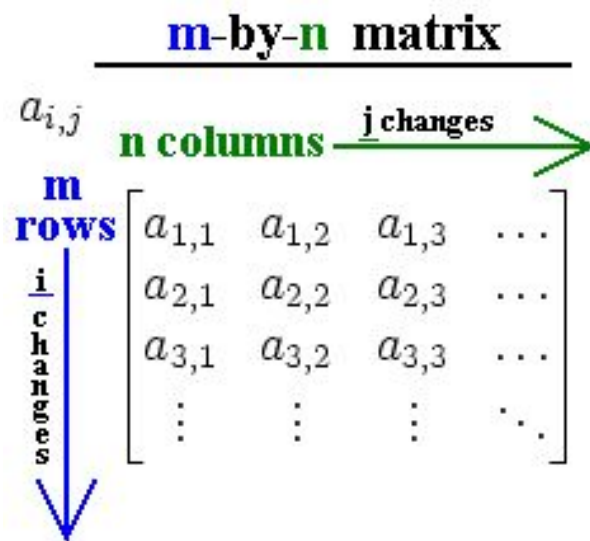
# Cabeçalhos

- Em C++, usamos arquivos de cabeçalhos
- Os mesmos descrevem os módulos

# Problema I

## Matriz

- Vamos criar um módulo matriz
- A mesma representa uma matriz que vai ser alocada dinamicamente



# Iniciando do .h

```
struct Matriz {  
    // Dados  
    int **_dados;  
    int _n_linhas;  
    int _n_colunas;  
  
    // Construtor  
    Matriz(int n_linhas, int n_colunas);  
    // Destrutor  
    ~Matriz();  
  
    // Métodos  
    void seta(int i, int j, int v); //  $M[i][j] = v$   
    int valor(int i, int j); // retorna valor  $i, j$   
    Matriz soma(Matriz &outra); // soma duas matrizes  
};
```

Note que não temos código

Já será explicado

# Código métodos seta e soma

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```

# Código métodos seta e soma

```
#include <iostream>
#include "matriz.h"
```

Note o include do módulo matriz

```
void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}
```

```
int Matriz::valor(int i, int j) {
    return _dados[i][j];
}
```

```
Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```

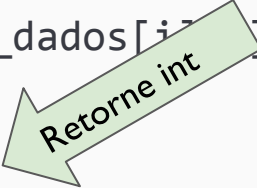
# Lendo 01: Retorne int

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```





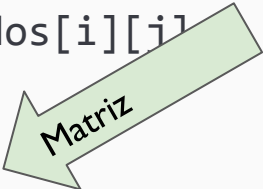
# Lendo 02: Na implementação do struct

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```



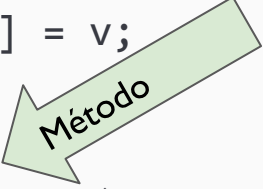
# Lendo 03: No método valor

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```



Método

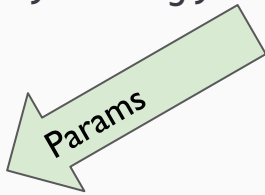
# Lendo 04: Que recebe i, j

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```



Params

# Lendo 04: Que recebe i, j

```
#include <iostream>
#include "matriz.h"

void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}

int Matriz::valor(int i, int j) {
    return _dados[i][j];
}

Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```

# Arquivo main

```
#include <iostream>
#include "matriz.h"

int main(void) {
    Matriz m1(2, 2);
    Matriz m2(2, 2);

    std::cout << m1.valor(0, 0) << std::endl;
    std::cout << m2.valor(0, 0) << std::endl;

    m1.seta(0, 0, 1);
    std::cout << m1.valor(0, 0) << std::endl;

    m2.seta(0, 0, 2);
    std::cout << m2.valor(0, 0) << std::endl;

    Matriz m3 = m1.soma(m2);
    std::cout << m3.valor(0, 0) << std::endl;
}
```

# Arquivo main

- Faz uso dos módulos
- Não se preocupa como a matriz é implementada, cliente do módulo

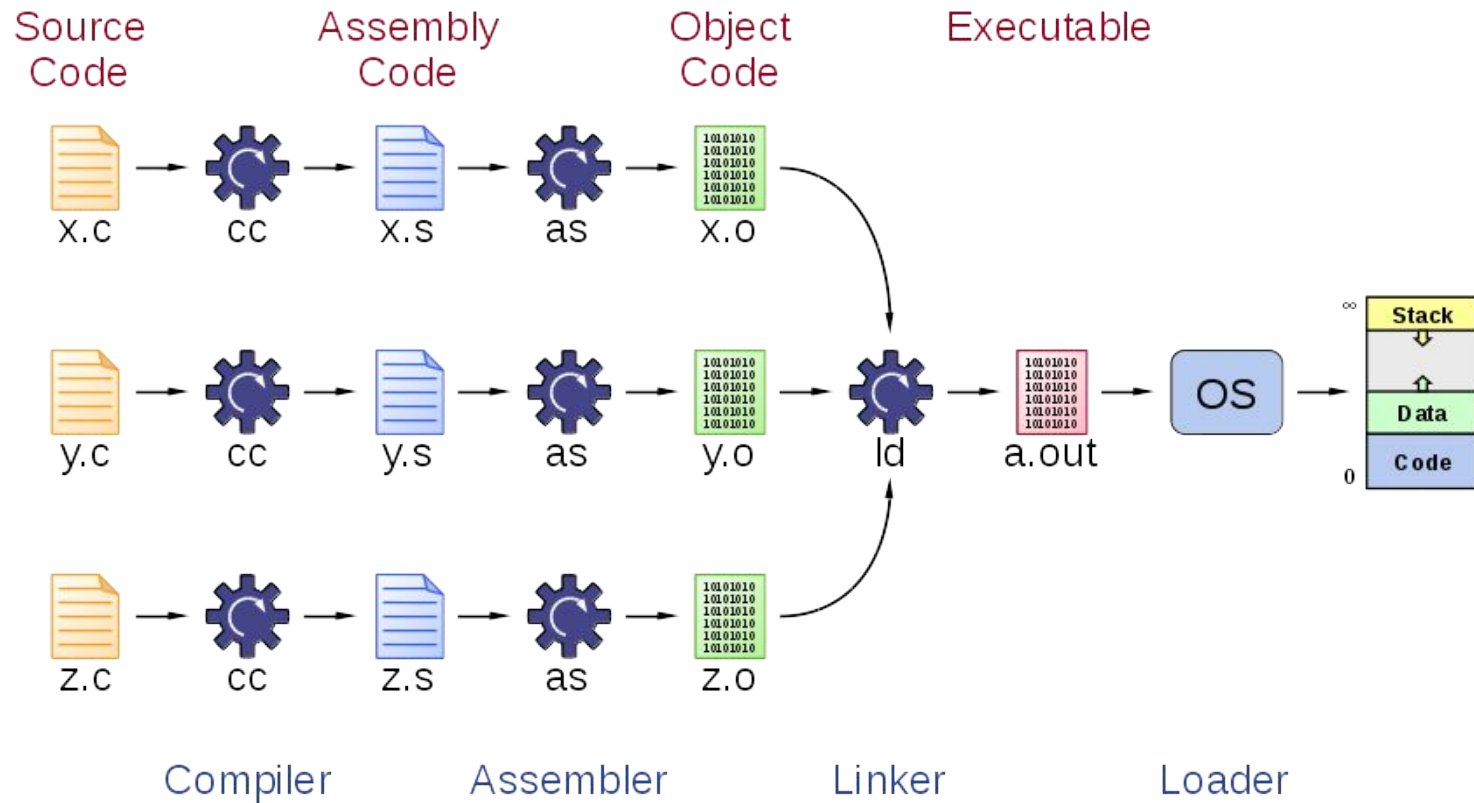
Note que não implementamos tudo aqui nos slides. Erro de compilação na prática. Estamos indo por partes!

# Compilando

```
$ g++ main.cpp matriz.cpp -o main
```

- Note que passamos dois arquivos
- O do main e o da matriz

# Compilação





# Construtores v. Destrutores

O nome diz tudo

- Procedimentos de inicialização
  - Usados apenas na criação de um novo objeto
- Procedimentos de destruição
  - Usados para liberar os recursos adquiridos na criação e utilizados por um certo objeto

# Construtores v. Destrutores

O nome diz tudo

- Destrutores tem um papel similar
- Liberar toda a memória que o objeto pode ter alocado
  - isto é, chamadas para **new**
- Também é útil para fechar recursos
  - Arquivos
  - Dentre outros

# Aquisição de Recurso é Inicialização

- Qual o motivo do destrutor?
- Uma boa prática é que todo objeto cuide da memória que o mesmo alocou
  - Em um sistema bem feito, apenas usamos os objetos

# Código Construtor e Destrutor

```
#include <iostream>
#include "matriz.h"

Matriz::Matriz(int n_linhas, int n_colunas) {
    std::cout << "Construindo uma matriz" << std::endl;
    _dados = new int*[n_linhas]();
    for (int i = 0; i < n_linhas; i++) {
        _dados[i] = new int[n_colunas];
    }
    _n_linhas = n_linhas;
    _n_colunas = n_colunas;
}

Matriz::~Matriz() {
    std::cout << "Destruindo uma matriz" << std::endl;
    for (int i = 0; i < _n_linhas; i++) {
        delete[] _dados[i];
    }
    delete[] _dados;
}
```

# Código

```
#include <iostream>
#include "matriz.h"

Matriz::Matriz(int n_linhas, int n_colunas) {
    std::cout << "Construindo uma matriz" << std::endl;
    _dados = new int*[n_linhas]();
    for (int i = 0; i < n_linhas; i++) {
        _dados[i] = new int[n_colunas];
    }
    _n_linhas = n_linhas;
    _n_colunas = n_colunas;
}


Matriz::~Matriz() {
    std::cout << "Destruindo uma matriz" << std::endl;
    for (int i = 0; i < _n_linhas; i++) {
        delete[] _dados[i];
    }
    delete[] _dados;
}
```

Tipo \*\*.  
Similar à malloc(n\*sizeof(int\*));

# Exemplificando Destrutores

```
#include "matriz.h"
```

```
int main(void) {
```



```
    Matriz *matriz = new Matriz(100, 100);  
    delete matriz;
```

```
    Matriz matriz2(100, 100);
```

```
    return 0;
```

```
}
```

```
$ ./main
```

# Exemplificando Destrutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;

    Matriz matriz2(100, 100);
    return 0;
}
```

```
$ ./main
Alocando matriz
```

# Exemplificando Destruutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;
    Matriz matriz2(100, 100);
    return 0;
}
```

```
$ ./main
Construindo matriz
Destruindo matrix
```



# Exemplificando Destruutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;

    Matriz matriz2(100, 100);
    return 0;
}
```



```
$ ./main
Construindo matriz
Destruindo matrix
Construindo matriz
```

# Exemplificando Destruutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;

    Matriz matriz2(100, 100);
    return 0;
}
```



```
$ ./main
Construindo matriz
Destruindo matrix
Construindo matriz
Destruindo matriz
```

# Destrutores

São chamados sempre que o objeto é desalocado

- Destrutores são chamados tanto para:
  - Objetos no heap
    - Depois de um delete
  - Objetos no stack
    - Depois que a função termina

# Destrutores

São chamados sempre que o objeto é desalocado

Lembrando que

- O computador cuida do stack
- Você cuida do heap
- Por isso fazemos o destrutor, a matriz é alocada dinamicamente no heap!

# Listas

---

# Listas Lineares

- Sequência de zero ou mais itens
  - $x_1, x_2, \dots, x_n$ , na qual  $x_i$
- Posições relativas
  - Assumindo  $n \geq 1$ ,  $x_1$  é o primeiro item da lista e  $x_n$  é o último item da lista.
  - $x_i$  precede  $x_{i+1}$  para  $i = 1, 2, \dots, n - 1$
  - $x_i$  sucede  $x_{i-1}$  para  $i = 2, 3, \dots, n$
  - $x_i$  é dito estar na  $i$ -ésima posição da lista

# Tipos Abstratos de Dados (TADs)

## Lista de números inteiros

- Considere uma uma lista de inteiros. Poderíamos definir TAD Lista, com as seguintes operações:
  - faça a lista vazia;
  - obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorna nulo;
  - insira um elemento na lista.

# Tipos Abstratos de Dados (TADs)

## Lista de números inteiros

- Quais outras operações podem ser definidas?
  - Retirar o i-ésimo item.
  - Localizar o i-ésimo item
  - Fazer uma cópia da lista linear.
  - Pesquisar a ocorrência de um item com um valor particular em algum componente.



# Solução Zero

```
#define TAMANHO 100
```

← Constante no .h. Em C++ também existe o const

```
struct ListaVetorInteiros {
```

```
    // Dados
```

```
    int *_elementos;
```

← Vetor de elementos que será alocado dinamicamente (heap)

```
    int _num_elementos_inseridos;
```

```
    // Construtor
```

```
    ListaVetorInteiros();
```

```
    // Destrutor
```

```
    ~ListaVetorInteiros();
```

```
    // Insere um inteiro na lista
```

```
    void inserir_elemento(int elemento);
```

```
    // Imprime a lista
```

```
    void imprimir();
```

```
};
```

# Como Implementar?

---

# Construtor (e início do .h)


```
#include <iostream>
```

← Em algum momento vamos imprimir a lista

```
#include "listavetor.h"
```

```
ListaVetorInteiros::ListaVetorInteiros() {  
    _elementos = new int[TAMANHO]();  
    _num_elementos_inseridos = 0;  
}
```

# Construtor (e início do .h)

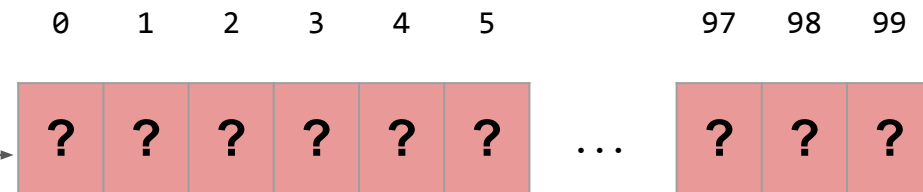
```
ListaVetorInteiros::ListaVetorInteiros() {  
 _elementos = new int[TAMANHO]();  
    _num_elementos_inseridos = 0;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos;  
}
```

# Construtor (e início do .h)

```
ListaVetorInteiros::ListaVetorInteiros() {  
    _elementos = new int[TAMANHO]();  
    _num_elementos_inseridos = 0;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos;  
}
```

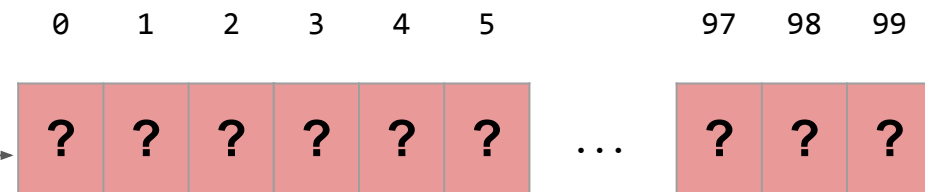


# Construtor (e início do .h)

```
ListaVetorInteiros::ListaVetorInteiros() {  
    _elementos = new int[TAMANHO]();  
    _num_elementos_inseridos = 0;  
}
```



```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
}
```



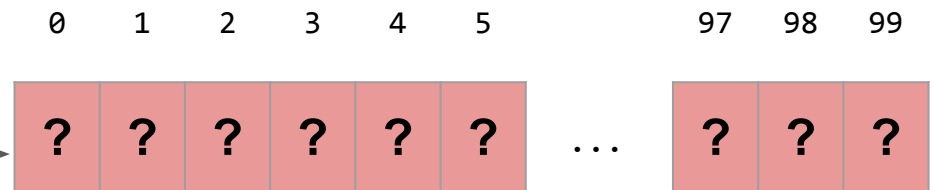
# Como adicionar elementos?

---

# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
}
```

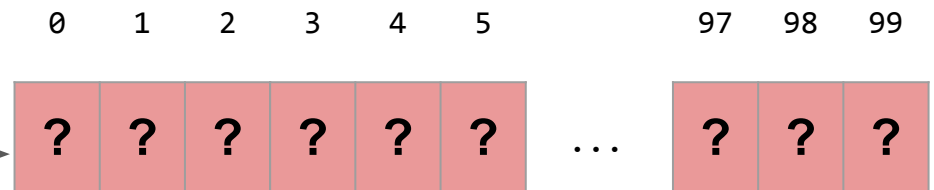




# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    → _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

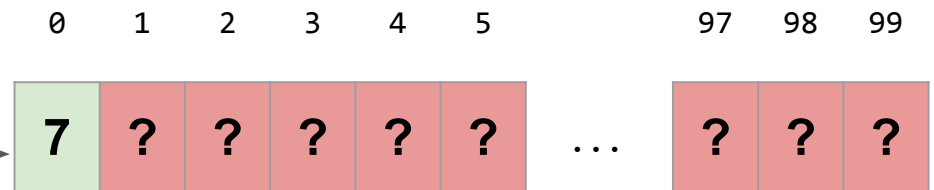
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

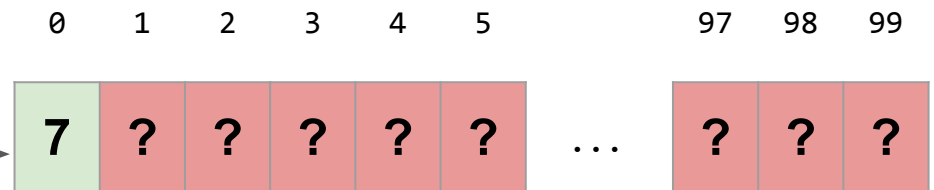
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

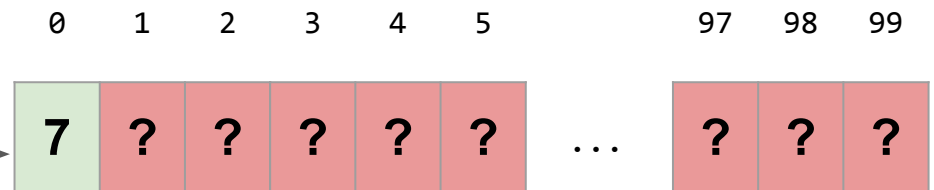
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 1;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

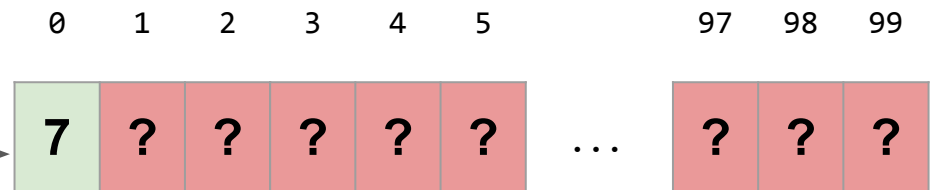
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 1;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    → _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

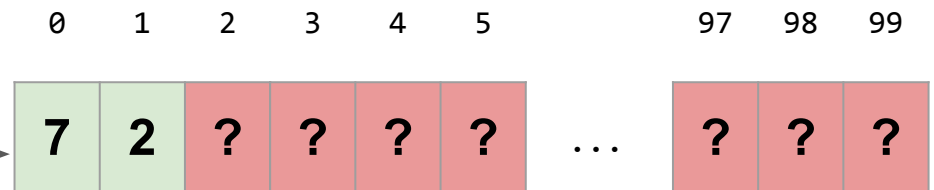
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 1;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

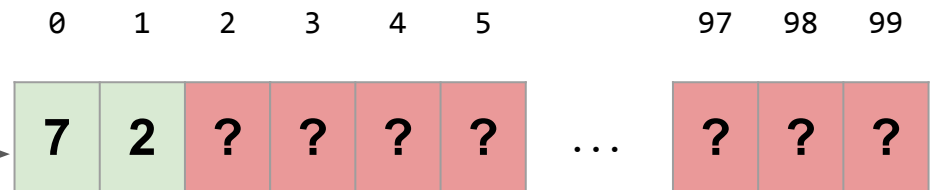
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 1;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

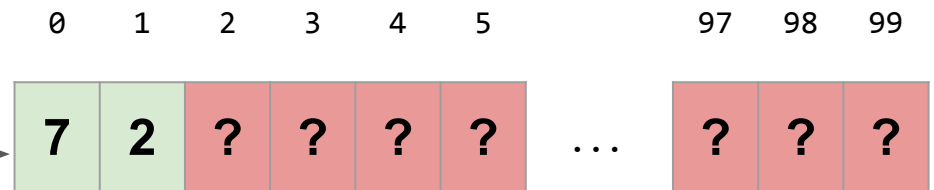
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 2;  
}
```



# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 2;  
}
```

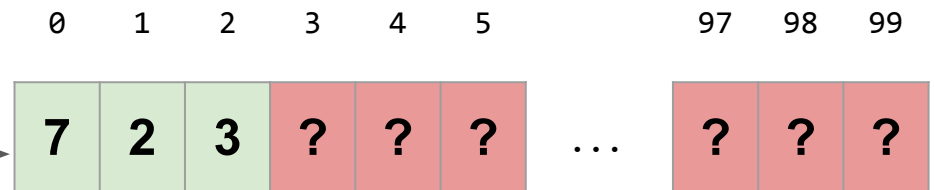




# Adicionando elementos?

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

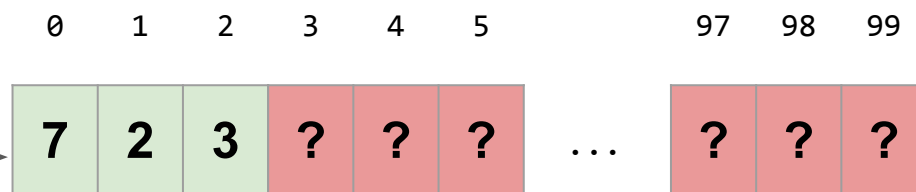
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 2;  
}
```



# E por aí vai...

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == TAMANHO) {  
        std::cerr << "Erro, lista cheia" << std::endl;  
        exit(1);  
    }  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
}
```



# Imprimir

- “Trivial”

```
void ListaVetorInteiros::imprimir() {  
    for (int i = 0; i < _num_elementos_inseridos; i++)  
        std::cout << _elementos[i] << " ";  
    std::cout << std::endl;  
}
```

# Destrutor

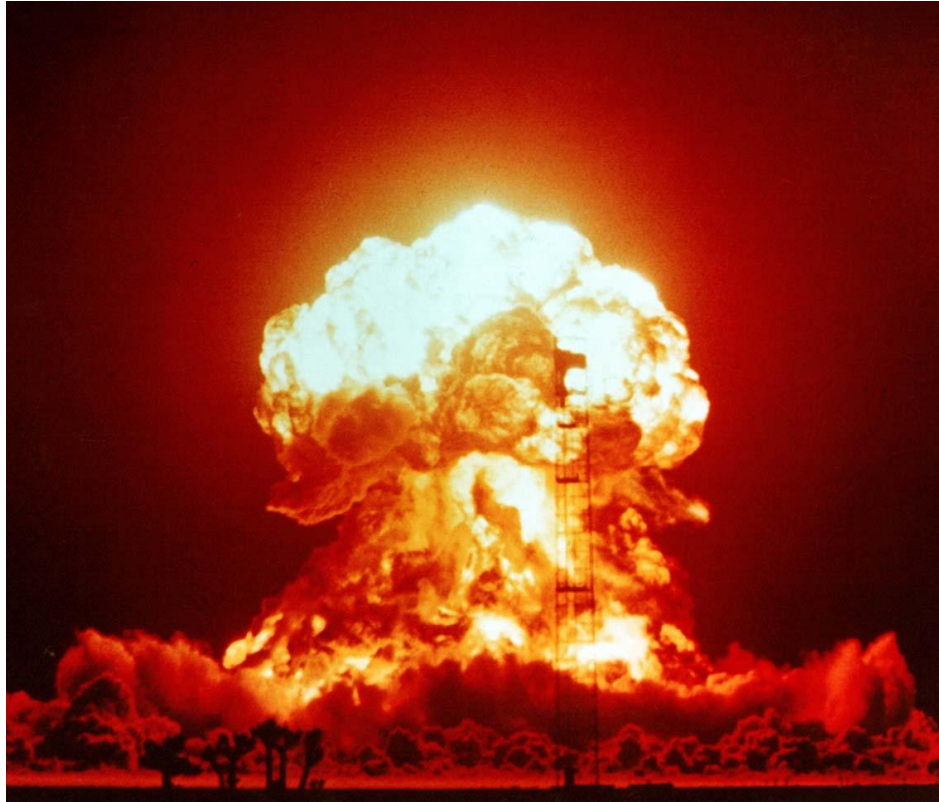
- Alocamos um vetor
  - **| new**
- Precisamos de **| delete**
- Lembrando, cada **new** → **| delete**

```
ListaVetorInteiros::~~ListaVetorInteiros() {  
    delete[] _elementos;  
}
```

**Mais de 100 elementos?**

---

# Mais de 100 elementos?



```
#define TAMANHO_INICIAL 100
```

Tamanho inicial que será aumentado

```
struct ListaVetorInteiros {
```

```
    // Dados
```

```
    int *_elementos;
```

```
    int _num_elementos_inseridos;
```

```
    int _capacidade;
```

Tamanho atual que também será aumentado

```
    // Construtor
```

```
    ListaVetorInteiros();
```

```
    // Destrutor
```

```
    ~ListaVetorInteiros();
```

```
    // Insere um inteiro na lista
```


```
    void inserir_elemento(int elemento);
```

```
    // Imprime a lista
```

```
    void imprimir();
```

```
};
```

# Construtor



```
ListaVetorInteiros::ListaVetorInteiros() {  
    _elementos = new int[TAMANHO_INICIAL]();  
    _num_elementos_inseridos = 0;  
    _capacidade = TAMANHO_INICIAL;  
}
```

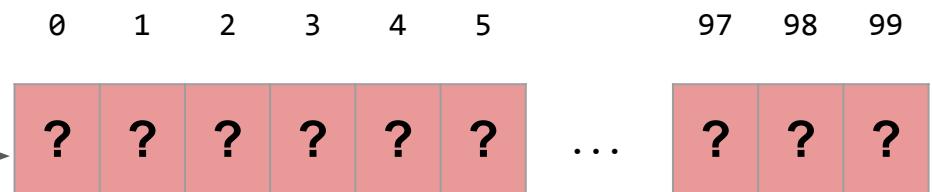


# Construtor

```
ListaVetorInteiros::ListaVetorInteiros() {  
    _elementos = new int[TAMANHO_INICIAL]();  
    _num_elementos_inseridos = 0;  
    _capacidade = TAMANHO_INICIAL;  
}
```



```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
    int _capacidade = 100;  
}
```



# Métodos que não mudam

- Imprime
- Destruitor

# Complicação

- Inserir elemento

# Ideia

- Inserir elemento
- Caso o vetor fique cheio
  - Duplicar o mesmo
  - Copiar tudo para o novo
  - Aumentar a capacidade
- Estamos implementando o **vector**
  - Nome do container na **STL**

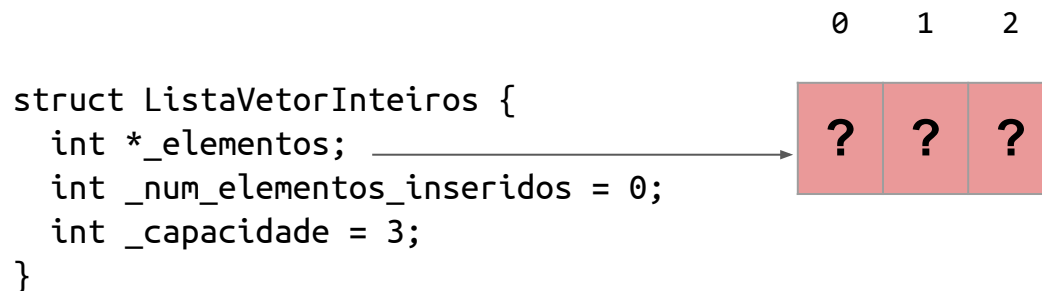
# Inserir elemento

```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    // . . .  
    _elementos[_num_elementos_inseridos] = elemento;  
    _num_elementos_inseridos++;  
}
```

# Passo a Passo

Alocamos tamanho inicial.

Vamos supor que seja igual a 3



# Passo a Passo

## Inserindo um elemento

```
struct ListaVetorInteiros {  
    int *_elementos;   
    int _num_elementos_inseridos = 1;  
    int _capacidade = 3;  
}
```

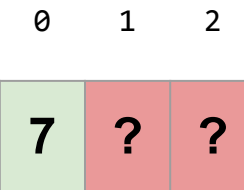
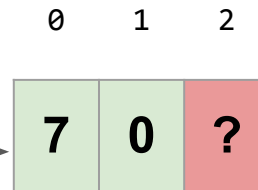


Diagram illustrating the insertion of an element into a vector. The vector has a capacity of 3 and currently contains 1 element (7) at index 0. The next two slots (indices 1 and 2) are empty (marked with ?).

# Passo a Passo

## Outro

```
struct ListaVetorInteiros {  
    int *_elementos; _____  
    int _num_elementos_inseridos = 2;  
    int _capacidade = 3;  
}
```

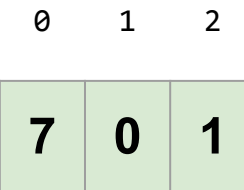




# Passo a Passo

+ |

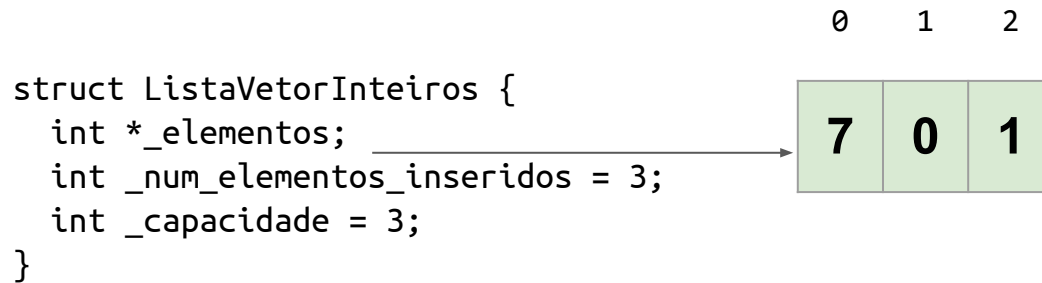
```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```



0	1	2
7	0	1

# Passo a Passo

+ outro!!

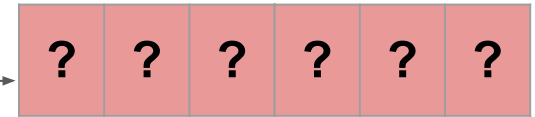


# Passo a Passo

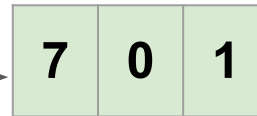
+ alocamos  
espaço

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 0;  
    int _capacidade = 3;  
}
```

int \*new\_data;



0    1    2



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```

# Passo a Passo

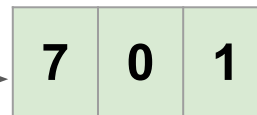
+ copiamos os dados

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```

int \*new\_data;



0    1    2



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```

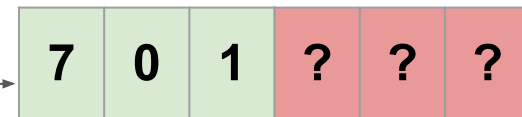
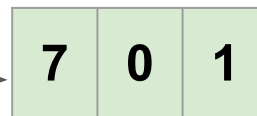
# Passo a Passo

+ copiamos os dados

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```

int \*new\_data;

0    1    2



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```

# Passo a Passo

+ apagamos os  
dados antigos

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```

int \*new\_data;



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```

# Passo a Passo

+ colocamos os  
novos no local

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```

# Passo a Passo

+ aumentamos a capacidade

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 6;  
}
```



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    if (_num_elementos_inseridos == _capacidade) {  
        int *new_data = new int[_capacidade * 2];  
  
        for (int i = 0; i < _num_elementos_inseridos; i++)  
            new_data[i] = _elementos[i];  
  
        delete[] _elementos;  
        _elementos = new_data;  
        _capacidade = _capacidade * 2;  
    }  
    // . . .  
}
```



# Passo a Passo

Agora estamos igual a antes

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 6;  
}
```

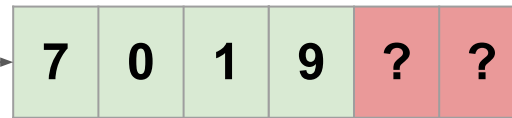


```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    // . . .  
    _elementos[_num_elementos_inserido] = elemento;  
    _num_elementos_inseridos++;  
}
```

# Passo a Passo

Agora estamos igual a antes

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 4;  
    int _capacidade = 6;  
}
```



```
void ListaVetorInteiros::inserir_elemento(int elemento) {  
    // . . .  
    _elementos[_num_elementos_inserido] = elemento;  
    _num_elementos_inseridos++;  
}
```

# E para remover o último elemento?

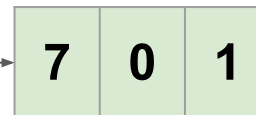
- Quero uma nova operação no meu TAD
  - Atualizar .h
  - Implementar no .cpp
- Remover o último elemento

# E para remover o último elemento?

- Quero uma nova operação no meu TAD
- Remover o último elemento

```
void ListaVetorInteiros::remover_ultimo() {  
    _num_elementos_inseridos--;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```

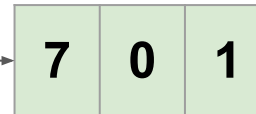


# E para remover o último elemento?

- Quero uma nova operação no meu TAD
- Remover o último elemento

```
void ListaVetorInteiros::remover_ultimo() {  
    → _num_elementos_inseridos--;  
}
```


```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```



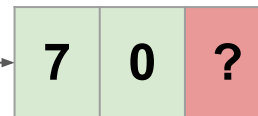
# E para remover o último elemento?

- Quero uma nova operação no meu TAD
- Remover o último elemento

```
void ListaVetorInteiros::remover_ultimo() {  
    _num_elementos_inseridos--;  
}
```



```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 3;  
    int _capacidade = 3;  
}
```




# Removendo o Primeiro Elemento

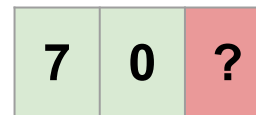
- Podemos copiar a ideia anterior
- Guardar um índice para o início
- Funciona bem?

# Removendo o Primeiro Elemento

- Podemos copiar a ideia anterior
- Guardar um índice para o início
- Chato não desperdiçar memória

```
void ListaVetorInteiros::remove_primeiro() {  
     _inicio++;  
    _num_elementos_inseridos--;  
}
```

```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 2;  
    int _capacidade = 3;  
    int _inicio = 0;  
}
```



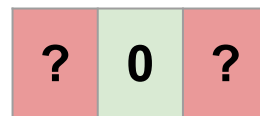


# Removendo o Primeiro Elemento

- Podemos copiar a ideia anterior
- Guardar um índice para o início
- Chato não desperdiçar memória

```
void ListaVetorInteiros::remove_primeiro() {  
    _inicio++;  
    _num_elementos_inseridos--;  
}
```


```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 2;  
    int _capacidade = 3;  
    int _inicio = 1;  
}
```



# Removendo o Primeiro Elemento

- Podemos copiar a ideia anterior
- Guardar um índice para o início
- Chato não desperdiçar memória

```
void ListaVetorInteiros::remover_primeiro() {  
    _inicio++;  
    _num_elementos_inseridos--;  
}
```



```
struct ListaVetorInteiros {  
    int *_elementos;  
    int _num_elementos_inseridos = 1;  
    int _capacidade = 3;  
    int _inicio = 1;  
}
```

