UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC004 - Algoritmos e Estruturas de Dados II

TADs específicos

Renato Martins

Email: renato.martins@dcc.ufmg.br

https://www.dcc.ufmg.br/~renato.martins/courses/DCC004

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Introdução

- Nenhum programa é escrito em uma linguagem de programação a partir do zero
- Geralmente
  - Linguagens vêm com bibliotecas
  - Impossível decorar todas
  - Usamos a documentação para entender
- Bibliotecas podem ser vistas como:
  - Conjunto de TADs e funções de uso geral

TADs que aprendemos

- Coleções/Containers
  - Listas, Árvores
- Números
  - Bignum
  - Complexo
- Geometria
  - Ponto

TADs do dia a dia

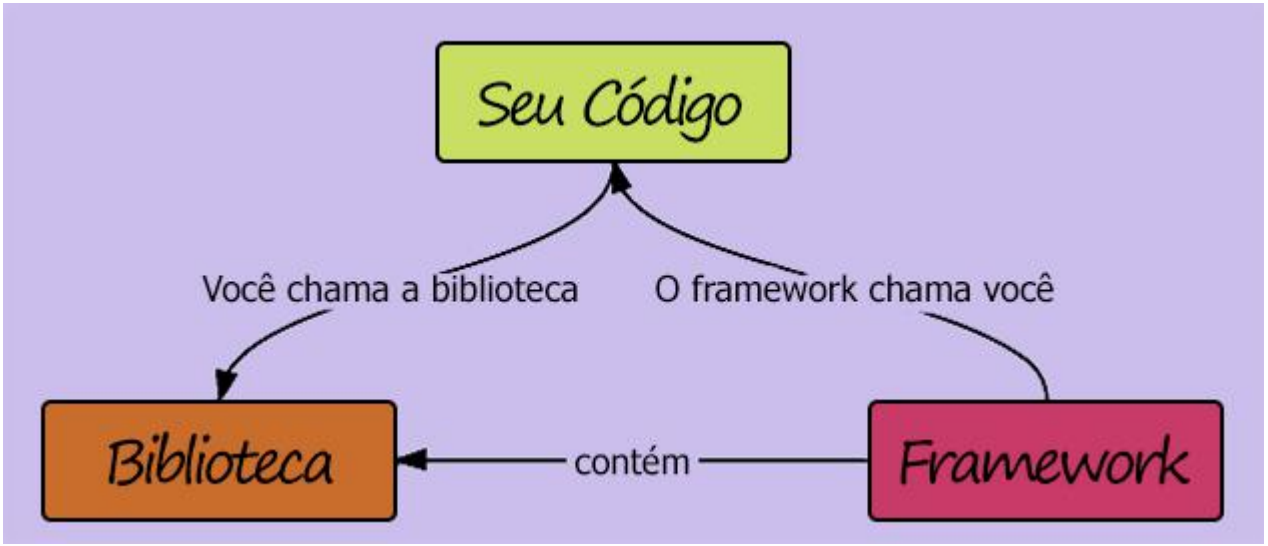
- Existem na biblioteca padrão de C++
- Para PDS2
  - Não precisamos ir muito além da padrão
- No pior dos casos, tente na Boost
  - https://www.boost.org/
  - Bignums e números complexos

Bibliotecas v. Frameworks

- Bibliotecas
  - Funcionalidades mais comuns
  - Containers, aritmética, matemática etc
- Frameworks
  - Servem para um propósito maior
  - Serviços web
  - Engines de jogos já prontas

Uma forma de ver

- A regra abaixo não vale sempre
- Mas se existir muitos callbacks
  - Chance de ser um framework



Componentes da biblioteca padrão

- Biblioteca padrão C++
- Strings
  - Suporte para expressões regulares
- Ponteiros inteligentes para gerenciamento de recurso (e.g. unique_ptr e shared_ptr)
- Um framework de containers (e.g. vector e map) e algoritmos (e.g. find() e sort())
  - Convencionalmente chamado STL (Standard Template Library)

Headers da Biblioteca Padrão

Qualquer funcionalidade da biblioteca padrão é fornecida através de um header padrão:

```
#include <string>
    #include <vector>
```

A biblioteca padrão é definida em um namespace chamado std. Para usar as funcionalidades, o prefixo std:: é usado:

```
std::string gato = "O gato miou.";
    std::vector<std::string> palavras = {"gato","pi"};
```

Headers da Biblioteca Padrão

Por simplicidade, podemos evitar o uso de std::

Geralmente não é uma boa prática carregar todos os nomes de um namespace no namespace global

```
#include <string>
using namespace std;
string s = "O gato miou.";
```

Explícito é melhor!

```
#include <string>
std::string s = "O gato miou.";
```

Strings em C++

Funções no nível da biblioteca

http://www.cplusplus.com/reference/string

fx Functions

Convert from strings

stoi C++11	Convert string to integer (function template)
stol C++11	Convert string to long int (function template)
stoul C++11	Convert string to unsigned integer (function template)
stoll C++11	Convert string to long long (function template)
stoull C++11	Convert string to unsigned long long (function template)
stof C++11	Convert string to float (function template)
stod C++11	Convert string to double (function template)
stold C++11	Convert string to long double (function template)

Convert to strings

to_string C++11	Convert numerical value to string (function)
to_wstring C++11	Convert numerical value to wide string (function)

Exemplo

to_string: Numérico → String

```
#include <iostream>
#include <string>

int main(void) {
  int valor = 0;
  std::cin >> valor;
  std::string valor_como_texto = "O valor foi: " + std::to_string(valor);
  std::cout << valor_como_texto;
}
```

Standard Template Library
Templates indicam um tipo genérico

- Programação Genérica
  - A mesma definição de função atua da mesma forma sobre objetos de diferentes tipos
- Polimorfismo universal – Paramétrico
  - Os tipos são passados como parâmetros
  - Código que pode ser reutilizado por classes em diferentes hierarquias tipo
- Templates (C++), Generics (Java)

Standard Template Library

Templates indicam um tipo genérico

```
#ifndef PDS2_LISTAGENERICA_H
#define PDS2_LISTAGENERICA_H

template <typename T>
struct node_t {
  T elemento;
  node_t *proximo;
};

template <typename T>
class ListaSimplesmenteEncadeada {
private:
  node_t<T> *_inicio;
  node_t<T> *_fim;
  int _num_elementos_inseridos;
public:
  ListaSimplesmenteEncadeada();
  ~ListaSimplesmenteEncadeada();
  void inserir_elemento(T elemento);
  void imprimir();
};
#endif
```

Standard Template Library

Templates indicam um tipo genérico

```
#ifndef PDS2_LISTAGENERICA_H
#define PDS2_LISTAGENERICA_H

template <typename T>
struct node_t {
  T elemento;
  node_t *proximo;
};

template <typename T>
class ListaSimplesmenteEncadeada {
private:
  node_t<T> *_inicio;
  node_t<T> *_fim;
  int _num_elementos_inseridos;
public:
  ListaSimplesmenteEncadeada();
  ~ListaSimplesmenteEncadeada();
  void inserir_elemento(T elemento);
  void imprimir();
};
#endif
```

Template para qualquer classe

Temos que definir para cada classe/struct

Aqui dizemos que node<T> usa T de ListaSimplesmente...

Standard Template Library
Templates indicam um tipo genérico

Fazendo uso

```
#include <string>

#include "listasimples.h"

int main(void) {
  ListaSimplesmenteEncadeada<int>  lista = ListaSimplesmenteEncadeada<int>();
  for (int i = 0; i < 1000; i++)
    lista.inserir_elemento(i);
  lista.imprimir();

  ListaSimplesmenteEncadeada<std::string>  lista2 = ListaSimplesmenteEncadeada<std::string>();
  lista2.inserir_elemento("flavio");
  lista2.inserir_elemento("douglas);
  lista2.imprimir();
  return 0;
}
```

Standard Template Library
Templates indicam um tipo genérico

- Exemplo de implementação no Github
  - Para quem quiser criar templates
- Não vamos nos preocupar tanto em como implementar, sim como fazer uso

Standard Template Library Containers

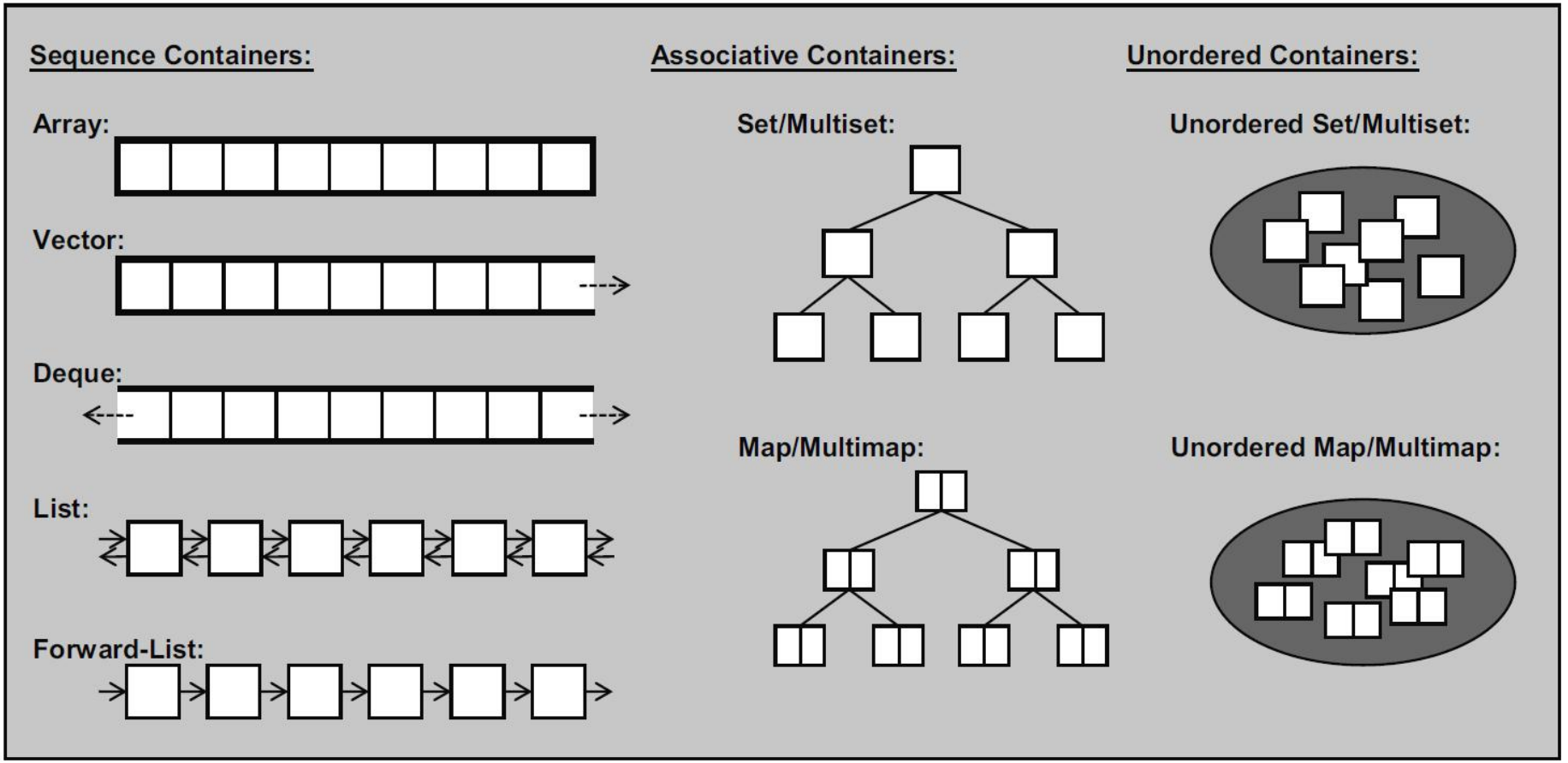
- Coleções de objetos
- Uso de containers apropriados para uma tarefa e suportá-los com operações fundamentais é crucial
- Containers usam templates por baixo
  - Assim fazemos uso de qualquer tipo
- Nem sempre o mesmo container é o melhor para diferentes problemas

Containers

Sequenciais	Associativos	Adaptadores
• Vector • Deque • List	• Set • Map • Multiset • Multimap	• Stack • Queue • Priority queue

Containers

Fazendo elo com os TADs



vector

- Um dos containers mais úteis é o vector
- Um vector é uma sequência de elementos
- Por baixo é uma lista com array

vector: uso

Vamos iniciar com uma classe Pessoa

Já implementada

```
#include <string>

class Pessoa {
private:
  const std::string _nome;
  int _idade;
public:
  // Construtor com lista de inicialização
  Pessoa(std::string nome, int idade):
            _nome(nome), _idade(idade) {}

  std::string get_nome() const {
            return this->_nome;
  }
  int get_idade() const {
            return this->_idade;
  }
};
```

No nosso caso _nome nunca muda, const

Primeira vez que vemos esse construtor.
Funciona igual ao anterior. Necessário const

Métodos const nunca mudam o objeto. Garantido.

vector: uso

Similar aos nossos TADs

push_back → Inserção; at → Acesso

```cpp
#include <iostream>
#include <string>
#include <vector>

#include "pessoa.h"

int main() {
  std::vector<Pessoa> pessoas;
  pessoas.push_back(Pessoa("Ana", 18));
  pessoas.push_back(Pessoa("Pedro", 19));

  // Primeira forma de acesso
  std::cout << pessoas[0].get_nome() << std::endl;
  std::cout << pessoas[1].get_nome() << std::endl;

  // Segunda forma, com at
  std::cout << pessoas.at(0).get_nome() << std::endl;
  std::cout << pessoas.at(1).get_nome() << std::endl;
  return 0;
}
```

vector de inteiros

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> v = {7, 5, 16, 8};
  v.push_back(25);
  v.push_back(13);

  for(int n : v) {
    std::cout << n << std::endl;
  }

  return 0;
}
```

Iterador for each. Percorre todos os elementos

for each: equivale ao laço abaixo

```cpp
for (int i = 0; i < v.size(); i++) int n = v[i];
```

vector

- Um vector pode ser copiado:

```
std::vector<Pessoa> lista2 = lista_tel;
```

- Atribuir um vector envolve copiar seus elementos. Após a inicialização de lista2, lista_tel e lista2 têm cópias separadas de cada elemento
- Tal inicialização pode ser cara
- Quando a cópia é indesejável, referências e ponteiros devem ser utilizados

Qual o problema das chamadas abaixo?

```
void ano_novo(std::vector<Pessoa> pessoas) {
  for (Pessoa pessoa : pessoas)
    pessoa.set_idade(pessoa.get_idade() + 1);
}
```

```
std::vector<std::string> pegar_nomes(std::vector<Pessoa> pessoas) {
  std::vector<std::string> nomes;
  for (Pessoa pessoa : pessoas)
    nomes.push_back(pessoa.get_nome());
  return nomes;
}
```

Qual o problema das chamadas abaixo?

```
void ano_novo(std::vector<Pessoa> pessoas) {
  for (Pessoa pessoa : pessoas)
    pessoa.set_idade(pessoa.get_idade() + 1);
}
```

```
std::vector<std::string> pegar_nomes(std::vector<Pessoa> pessoas) {
  std::vector<std::string> nomes;
  for (Pessoa pessoa : pessoas)
    nomes.push_back(pessoa.get_nome());
  return nomes;
}
```

(1) Passagem por cópia

(2) Mesmo se fosse por referência, for each faz cópia

Forma correta

```
// Sem const, vamos mudar a memória (cada pessoa aumenta de idade)
void ano_novo(std::vector<Pessoa> &pessoas) {
  for (int i = 0; i < pessoas.size(); i++)
    pessoas.at(i).set_idade(pessoas.at(i).get_idade() + 1);
}


// Com const, leitura apenas.
std::vector<std::string> pegar_nomes(std::vector<Pessoa> const &pessoas) {
  std::vector<std::string> nomes;
  for (Pessoa pessoa : pessoas)
    nomes.push_back(pessoa.get_nome());
  return nomes;
}
```

1. Note o uso de **const** quando apenas lemos
2. Note que não usamos for each quando alteramos os objetos. Laço normal.

Diferentes laços

Assumindo um vetor de inteiros, explique cada caso

Laço clássico

```
std::vector<int> dados = {0, 7, 8, 1, 3};
for (int i = 0; i < dados.size(); i++)
  std::cout << dados[i];
```

Laço compacto

```
for (int x : dados)
  std::cout << x;
```

Laço para a referência

```
for (int &x : dados)
  x *= 2;
```

list

- Lista duplamente encadeada
- Não temos mais acesso via índice. Motivo?
  - Iterador para acessar os elementos

```
#include <iostream>
#include <list>

int main() {
  std::list<int> l = {7, 5, 16, 8};

  // Add an integer to the front of the list
  l.push_front(25);
  // Add an integer to the back of the list
  l.push_back(13);

  for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
    std::cout << *it << std::endl;
  }
  return 0;
}
```

list

Sobre iteradores

- Funcionam de forma similar a ponteiros
- Lembre-se da aritmética de ponteiros

```cpp
#include <iostream>
#include <list>

int main() {
  std::list<int> l = {7, 5, 16, 8};

  // Add an integer to the front of the list
  l.push_front(25);
  // Add an integer to the back of the list
  l.push_back(13);

  for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
    std::cout << *it << std::endl;
  }
  return 0;
}
```

Iteradores

- Geralmente não acessamos elementos usando índices quando usamos uma lista encadeada
- Quando queremos identificar um elemento em uma list usamos um iterador
- Todo container da biblioteca padrão oferece as funções begin() e end(), que retorna um iterador pro primeiro e depois do último elemento

Iteradores

São basicamente ponteiros (pelo menos em C++)

l.begin() → ponteiro primeiro elemento

l.begin() + 1 → ponteiro para segundo

```cpp
#include <iostream>
#include <list>

int main() {
  std::list<int> l = {7, 5, 16, 8};

  // Add an integer to the front of the list
  l.push_front(25);
  // Add an integer to the back of the list
  l.push_back(13);

  for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
    std::cout << *it << std::endl;
  }
  return 0;
}
```

Iteradores

Usando um iterador:

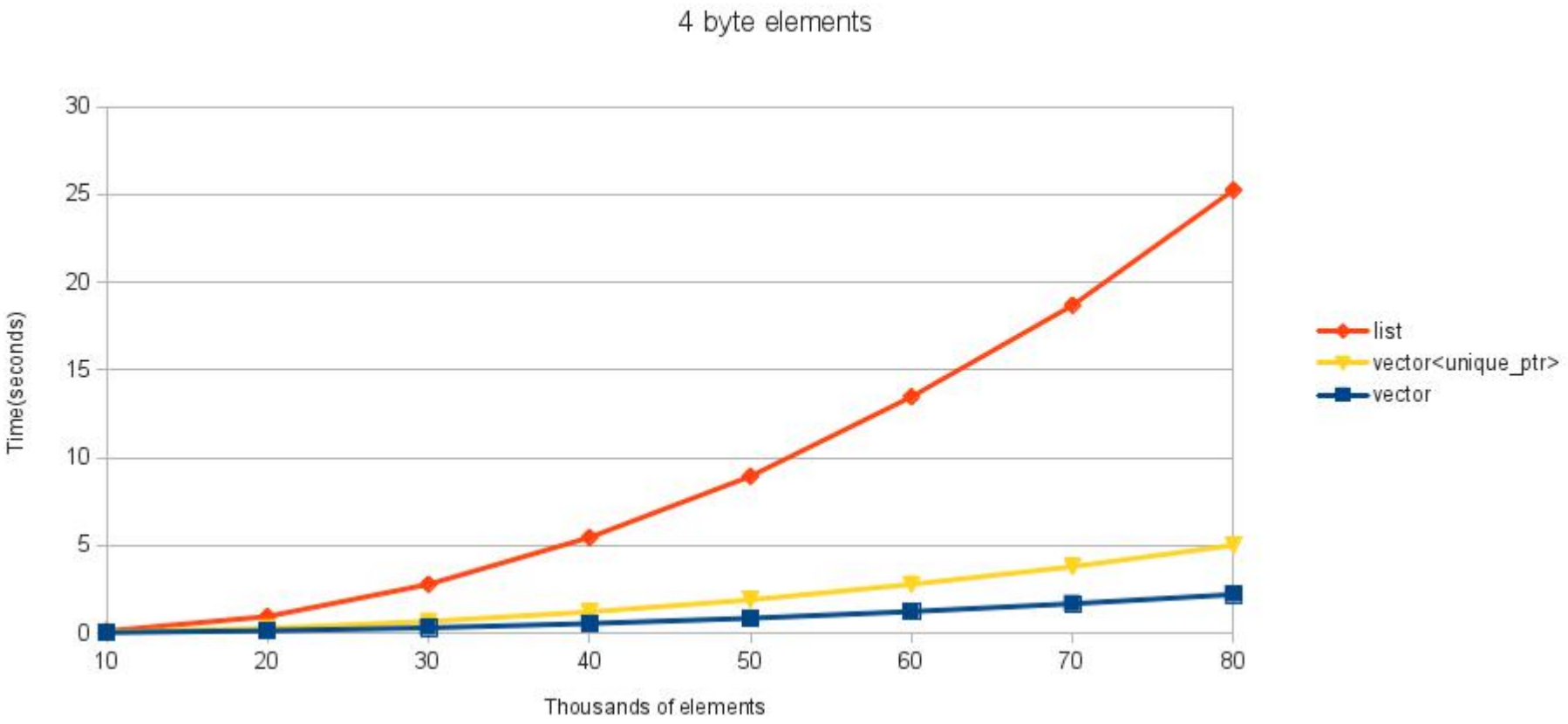
```
for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
  std::cout << *it << std::endl;
}
```

- Dado um iterador it, *it é o elemento que ele se refere, ++p avança p para o próximo elemento

list vs vector

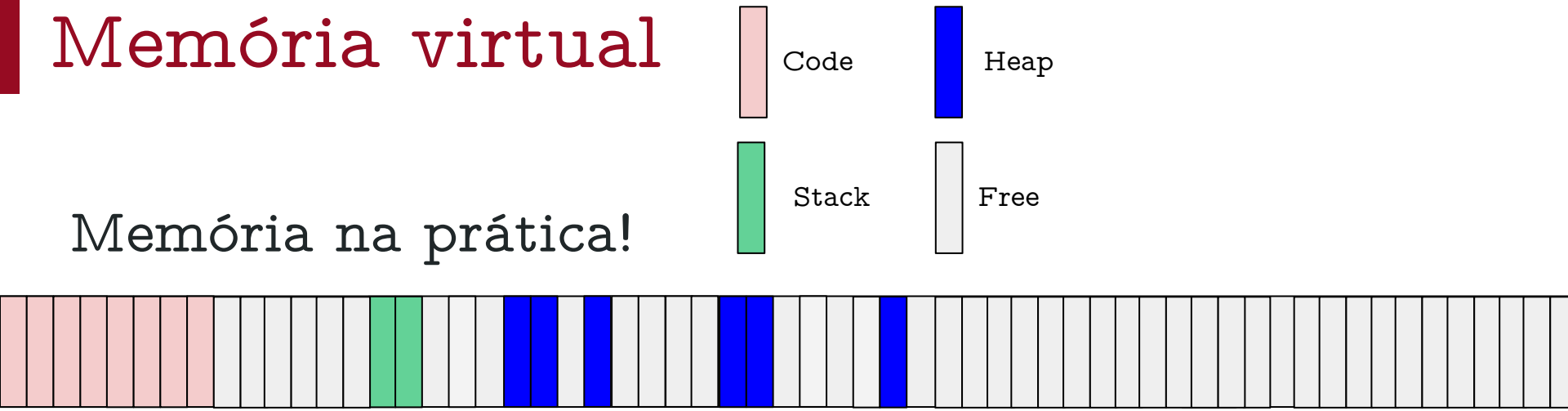
- Quando queremos uma sequência de elementos, podemos escolher entre vector e list
- A não ser que tenha um motivo, use vector, ele tem desempenho melhor para percorrer (e.g., find()), e para ordenar e pesquisar (e.g., sort())

Desempenho: elementos pequenos



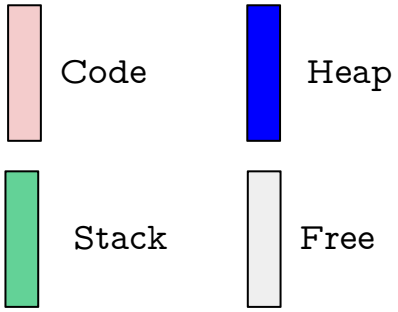
vector<unique_ptr> vetor de ponteiros

Memória virtual

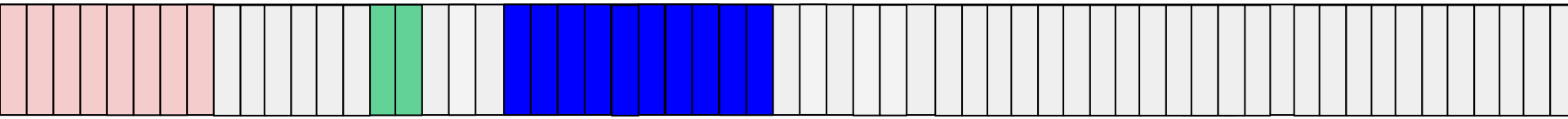


- Embora parece contígua para o programa
- A memória é na verdade toda fatiada
  - Tamanhos de 4KB geralmente
- O conceito de “endereços virtuais”
  - Faz com que tudo parece sequencial (contígua)

Memória virtual

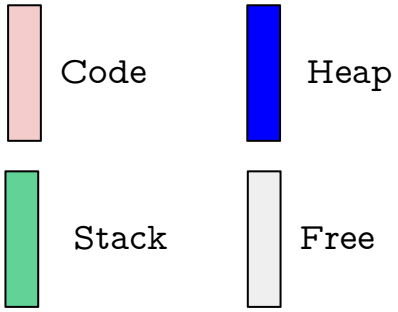


Memória na prática!

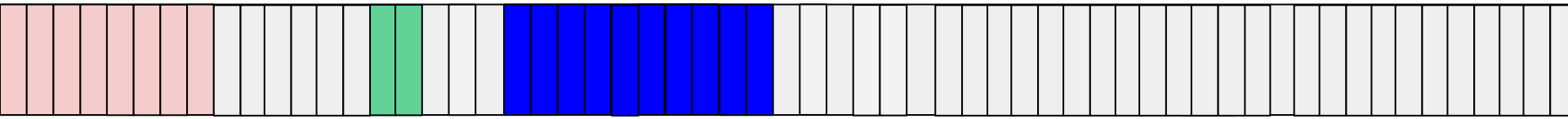


- Porém ao alocar um vector, é feito um melhor esforço para tudo ficar sequencial

Memória virtual

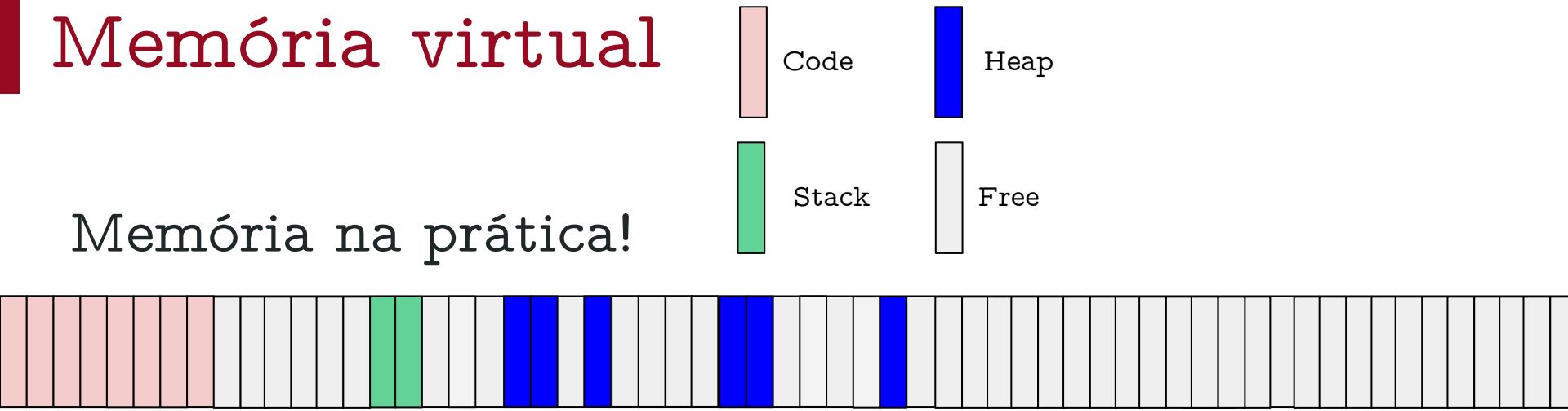


Memória na prática!



- Porém ao alocar um vector, é feito um melhor esforço para tudo ficar sequencial
- Sistema é mais rápido com memória alocada em sequência

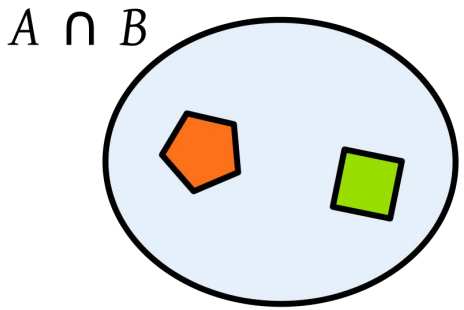
Memória virtual



- A lista fica assim (mais ou menos)
- Sem localidade
- Localidade:
  - Temporal
  - Espacial

set
Conjuntos matemáticos

- Elementos únicos
- Sem ordem
  - Embora podemos representar ordenado



set: uso

```cpp
#include <iostream>
#include <set>

int main() {
  std::set<int> s;
  for(int i = 1; i <= 10; i++) {
    s.insert(i);
  }

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }

  s.insert(7);

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }
  for(int i = 2; i <= 10; i += 2) {
    s.erase(i);
  }

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }
  return 0;
}
```

set: uso

```
#include <iostream>
#include <set>

int main() {
  std::set<int> s;
  for(int i = 1; i <= 10; i++) {
    s.insert(i);
  }

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }

  s.insert(7);

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }
  for(int i = 2; i <= 10; i += 2) {
    s.erase(i);
  }

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }
  return 0;
}
```

s = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

s = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

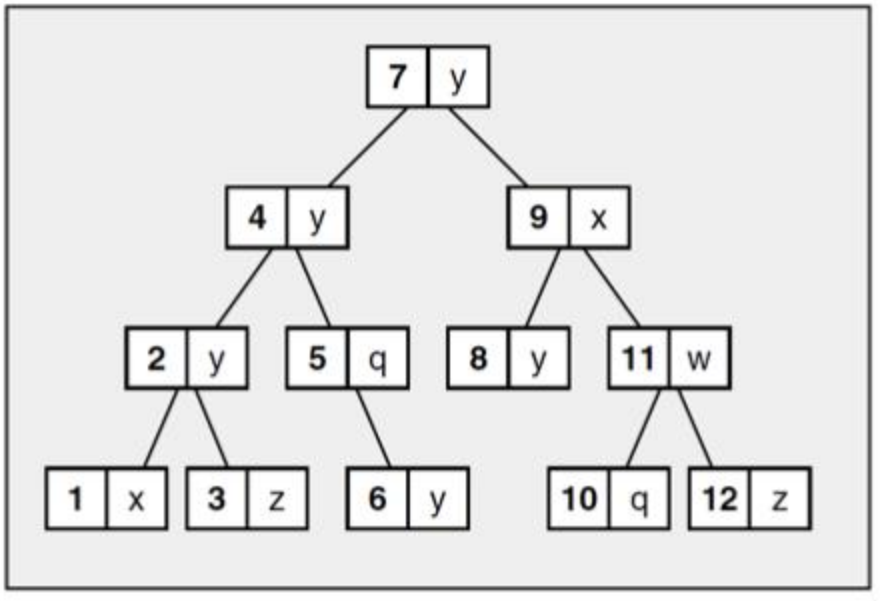
s = {1, 3, 5, 7, 9}

map

- Dicionário de chave → valor
- Uma árvore por baixo
  - Especial: sempre balanceada
  - Guarda pares de elementos

map
entendendo a memória por baixo

- Cada elemento é um nó
- Guarda uma chave (número neste caso)
- Valor (letra neste caso)



map
exemplo

```
#include <iostream>
#include <string>
#include <map>

int main() {
  std::map<int,std::string> m;
  m.insert(std::pair<int,std::string>(2017123456, "Joao"));

  m[2016123456] = "Maria";
  m[2018123456] = "Carlos";
  m[2015123456] = "Jose";
  m[2014123456] = "Joana";

  std::map<int,std::string>::iterator it;
  for (it = m.begin(); it != m.end(); it++) {
    std::cout << it->first << ": " << it->second << std::endl;
  }
  return 0;
}
```

Versões sem ordem unordered_map e unordered_set

- Por padrão, maps/sets são implementados como árvores binárias de busca
- Existem versões unordered_*
  - Mais eficazes na prática
  - Porém, não conseguimos ordenar as chaves
- Iterador em um map/set
  - Sempre em ordem

Criando mapas de tipos diferentes

Precisamos saber comparar. Usamos um struct

```
// O comparator sempre verificar se é <. Com < podemos criar >, == e != . Note que:
//          p1 < p2   <--> p2 > p1
//          p1 >= p2  <--> !(p1 < p2)
//          p1 == p2  <--> !(p1 < p2) && !(p2 < p1)
//          p1 != p2  <--> !(p1 == p2)
struct compara_pessoa_f {
  bool operator()(const Pessoa& p1, const Pessoa& p2) {
    return p1.get_idade() < p2.get_idade();
  }
};

int main() {
  std::set<Pessoa, compara_pessoa_f> pessoas;
  pessoas.insert(Pessoa("Ana", 18));
  pessoas.insert(Pessoa("Pedro", 19));
  pessoas.insert(Pessoa("Ana", 18));

  for (Pessoa p : pessoas)
    std::cout << p.get_nome() << std::endl;

  return 0;
}
```

Criando mapas de tipos diferentes

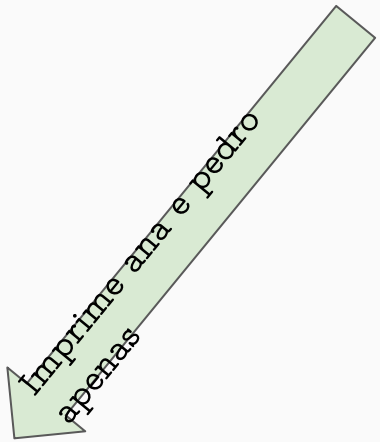
Precisamos saber comparar. Usamos um

```
// O comparator sempre verificar se é <. Com < podemos criar >, == e != . Note que:
//          p1 < p2   <--> p2 > p1
//          p1 >= p2  <--> !(p1 < p2)
//          p1 == p2  <--> !(p1 < p2) && !(p2 < p1)
//          p1 != p2  <--> !(p1 == p2)
struct compara_pessoa_f {
  bool operator()(const Pessoa& p1, const Pessoa& p2) {
    return p1.get_idade() < p2.get_idade();
  }
};

int main() {
  std::set<Pessoa, compara_pessoa_f> pessoas;
  pessoas.insert(Pessoa("Ana", 18));
  pessoas.insert(Pessoa("Pedro", 19));
  pessoas.insert(Pessoa("Ana", 18));

  for (Pessoa p : pessoas)
    std::cout << p.get_nome() << std::endl;

  return 0;
}
```



Outras bibliotecas de C++

<algorithm>	copy(), find(), sort()
<cmath>	sqtr(), pow()
<fstream>	fstream, ifstream, ofstream
<iostream>	istream,ostream, cin,cout
<memory>	unique_ptr, shared_ptr