UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC004 - Algoritmos e Estruturas de Dados II

Programação Orientada a Objetos (Encapsulamento)

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

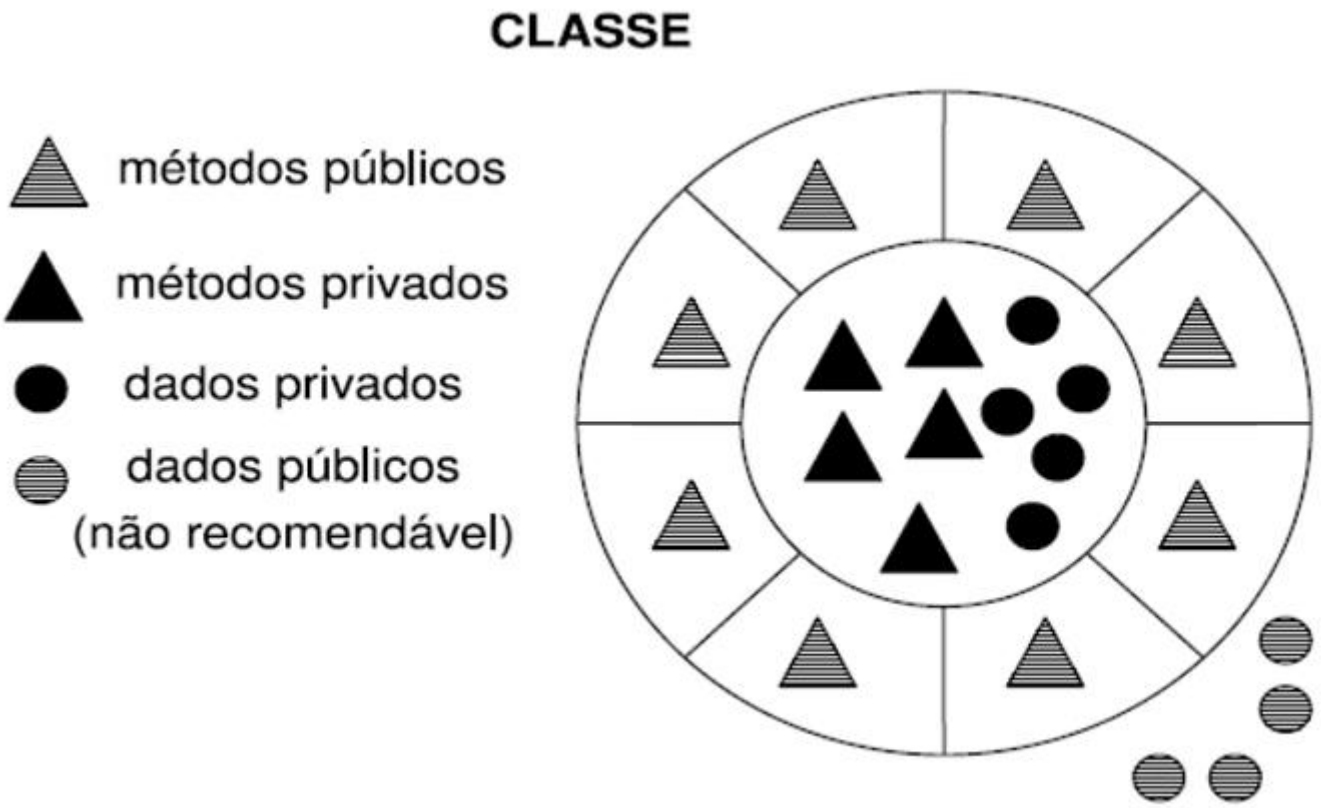
Introdução

- Abstração
  - Simplificação de um problema difícil
  - É o ato de representar as características essenciais sem incluir os detalhes por trás
- Ocultação de dados
  - Informações desnecessárias devem ser escondidas do mundo externo (usuários)

Introdução

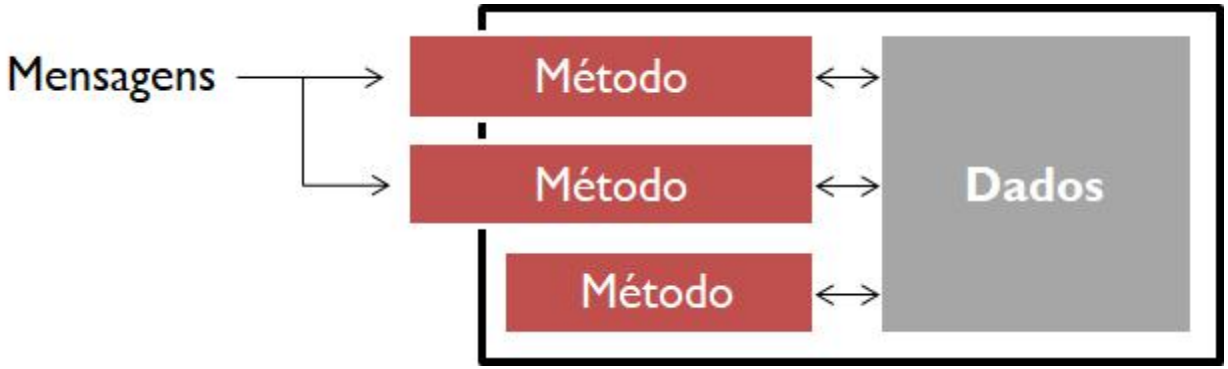
- Encapsulamento
  - Mecanismo que coloca juntos os dados e suas funções associadas, mantendo-os controlados em relação ao seu nível de acesso
- Proporciona abstração
  - Separa a visão externa da visão interna
  - Protege a integridade dos dados do Objeto

Introdução


CLASSE
métodos públicos
métodos privados
dados privados
dados públicos
(não recomendável)

Introdução

- Informações encapsuladas em uma Classe
  - Estado (dados)
  - Comportamento (métodos)



Encapsulamento
Benefícios

- Desenvolvimento
  - Melhor compreensão de cada classe
  - Facilita a realização de testes
- Manutenção/Evolução
  - Impacto reduzido ao modificar uma classe
  - Interface deve ser o mais constante possível

Encapsulamento

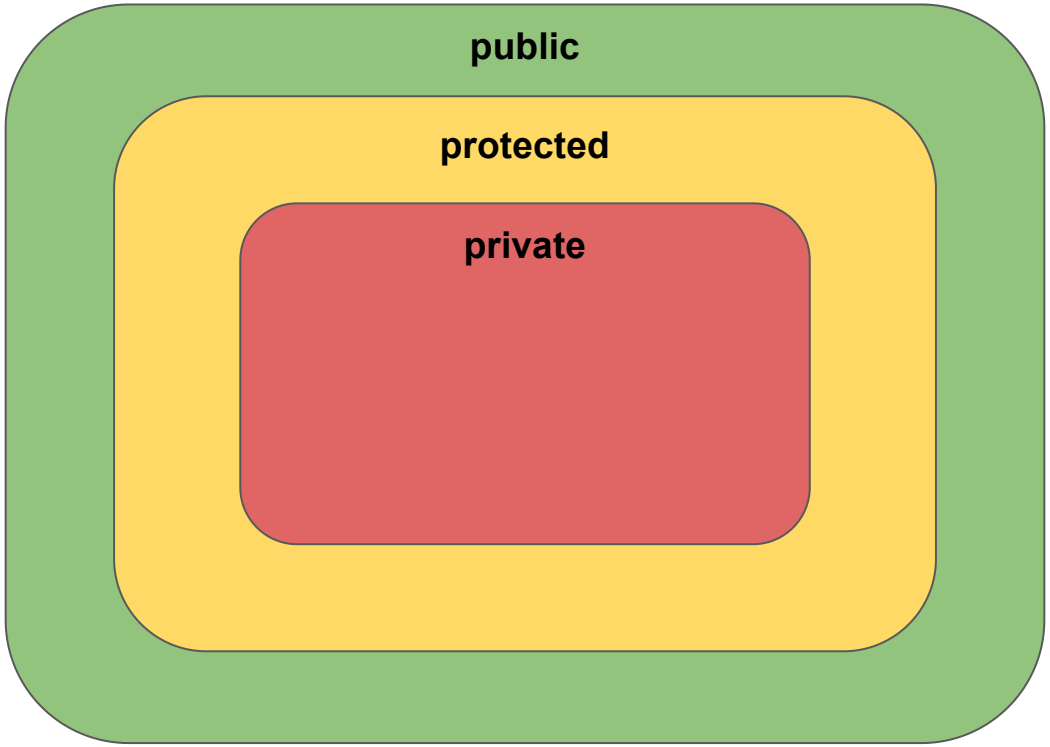
- Encapsulamento ocorre nas classes
- O comportamento e a interface de uma classe são definidos pelos seus membros
  - Atributos
  - Métodos
- Fazem uso dos modificadores de acesso

Encapsulamento
C++

- Modificadores de acesso
  - Public
  - Protected
  - Private
- Membros declarados após o modificador

https://en.cppreference.com/w/cpp/language/access

Encapsulamento
Modificadores de acesso



Encapsulamento
Modificadores de acesso - Public

- Permite que o membro público possa ser acessado de qualquer outra parte do código
- Mais liberal dos modificadores
  - Fazem parte (definem) o contrato da classe
  - Deve ser usado com responsabilidade
    - Por quê?

Encapsulamento
Modificadores de acesso - Public

```
class Ponto {

   public:
      int x;
      int y;

      Ponto(int x, int y) {
         this->x = x;
         this->y = y;
      }

};
```

Encapsulamento
Modificadores de acesso - Protected

- Permite que o membro possa ser acessado apenas por outras classes que:
  - Fazem parte da hierarquia (derivadas)
  - Classes “amigas”
    - Algo bem específico em C++

Encapsulamento
Modificadores de acesso - Protected

```
class Base {
   protected:
      int i = 0;
};

class Derived : Base {
   public:
      int f()
      {
         i++;
         return i;
      }
};
```

```
int main()
{
    Base b;
    //cout << b.i << endl;     //Erro

    Derived d;
    cout << d.f() << endl;

    return 0;
}
```

Encapsulamento
Modificadores de acesso - Private

- Permite que o membro privado possa ser acessado por métodos da mesma classe
- O mais restritivo dos modificadores
  - Deve ser empregado sempre que possível
  - Utilizar métodos auxiliares de acesso
- Quando não há declaração explícita
  - Nível padrão (implícito)

Encapsulamento
Modificadores de acesso - Private

```
class Ponto {

   private:
      int x;
      int y;

   public:
      Ponto(int x, int y) {
         this->x = x;
         this->y = y;
      }

};
```

Encapsulamento
Modificadores de acesso - Private
(Exemplo 1)

```
class Base {
   private:
      int i = 0;
};

class Derived : Base {
   public:
      int f()
      {
         i++;
         return i;
      }
};
```

```
int main()
{
    Base b;
    cout << b.i << endl;

    Derived d;
    cout << d.f() << endl;

    return 0;
}
```

Encapsulamento
Modificadores de acesso - Private
(Exemplo 2)

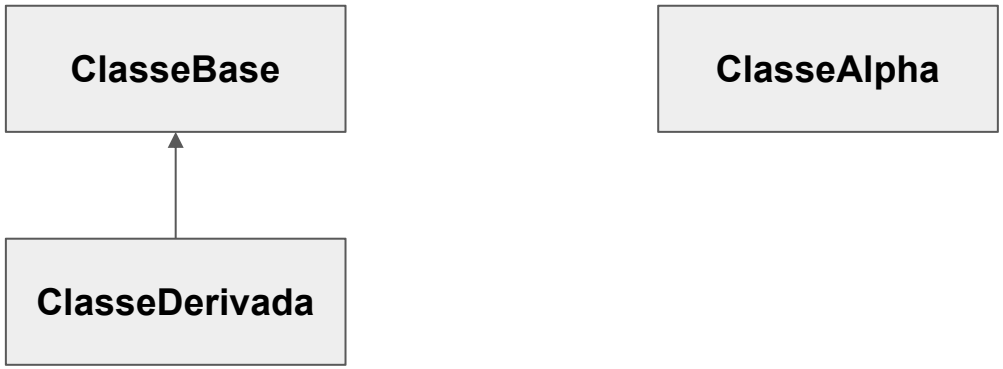
```
class Ponto {

   private:
      int x;
      int y;

      Ponto(int x, int y) {
         this->x = x;
         this->y = y;
      }

};
```

Encapsulamento
Modificadores de acesso - Private
(Exemplo 3)

```
class Ponto {

   private:
      class EstruturaPonto {
         public:
            double x;
            double y;
      };

      EstruturaPonto p;

   public:
      Ponto(int x, int y) {
         p.x = x;
         p.y = y;
      }
};
```

Encapsulamento
Modificadores de acesso

Considerando a visibilidade dos membros da **ClasseBase** de acordo com o modificador de acesso



	ClasseBase	ClasseDerivada	ClasseAlpha
Public	■	■	■
Protected	■	■	■
Private	■	■	■

Encapsulamento
Modificadores de acesso

	Classe	Subclasse	Mundo
Public			
Protected			
Private			

Encapsulamento
Acessando e modificando atributos

- Evitar a manipulação direta de atributos
  - Acesso deve ser o mais restrito possível
  - De preferência todos devem ser private
- Sempre utilizar métodos auxiliares
  - Melhor controle das alterações
  - Acesso centralizado

Encapsulamento
Getters e Setters

- Convenção de nomenclatura dos métodos
- Get
  - Os métodos que permitem apenas o acesso de consulta (obter) devem possuir o prefixo get
- Set
  - Os métodos que permitem a alteração (definir) devem possuir o prefixo set

Encapsulamento
Getters e Setters

```
class Ponto {
   private:
      double x;
      double y;

   public:
      Ponto(double x, double y) {
         setX(x);
         setY(y);
      }

      void setX(double x) { this->x = x; }
      void setY(double y) { this->y = y; }

      double getX() { return this->x; }
      double getY() { return this->y; }
};
```

Encapsulamento
Getters e Setters

- Todos os atributos devem possuir get e set
- Nomenclatura alternativa
  - Atributos booleanos devem utilizar o prefixo “is” ao invés do prefixo get
- Melhora a legibilidade e entendimento

Encapsulamento
Getters e Setters

```
class Cliente {
   private:
      string nome;
      bool ativo;

   public:
      Cliente(string nome, bool ativo) {
         setNome(nome);
         setAtivo(ativo);
      }

      void setNome(string nome) { this->nome = nome; }
      void setAtivo(bool ativo) { this->ativo = ativo; }

      string getNome() { return this->nome; }
      bool isAtivo() { return this->ativo; }
};
```

Exercício

- Modelar uma conta bancária
  - Quais atributos devem existir?
  - Quais métodos devem existir?

Exercício

```
class Conta {

   public:
      int agencia;
      int numero;
      double saldo;

      Conta(int agencia, int numero) : agencia(agencia), numero(numero){
      }

      // Métodos auxiliares
};
```

Exercício

```
class Conta {

   private:
      int agencia;
      int numero;
      double saldo = 0;

   public:
      Conta(int agencia, int numero) {...}

      void setAgencia(int ag) { this->agencia = ag; }
      void setNumero(int num) { this->numero = num; }
      void setSaldo(double saldo) { this->saldo = saldo; }

      int getAgencia() { return this->agencia; }
      int getNumero() { return this->numero; }
      double getSaldo() { return this->saldo; }
};
```

Exercício

```
class Conta {

   private:
      int agencia;
      int numero;
      double saldo = 0;

   public:

      {...}

      void depositar(double valor) {
         this->saldo += valor;
      }

      void sacar(double valor) {
         this->saldo -= valor;
      }
};
```

Exercício

```
class Conta {

   {...}

   public:

      {...}

      void depositar(double valor) {
         this->saldo += valor;
         this->saldo -= 0.25;
      }

      void sacar(double valor) {
         this->saldo -= valor;
         this->saldo -= 0.25;
      }
};
```

Exercício

```
class Conta {

   {...}

   public:

      {...}

      void depositar(double valor) {
         this->saldo += valor;
         descontarTarifa();
      }

      void sacar(double valor) {
         this->saldo -= valor;
         descontarTarifa();
      }

      void descontarTarifa() {
         this->saldo -= 0.25;
      }
};
```

Exercício

```
class Conta {

   private:
      void descontarTarifa() {
         this->saldo -= 0.25;
      }

   public:

      {...}

      void depositar(double valor) {
         this->saldo += valor;
         this->descontarTarifa();
      }

      void sacar(double valor) {
         this->saldo -= valor;
         this->descontarTarifa();
      }
};
```

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

DCC004 - Algoritmos e Estruturas de Dados II

Modularização

Renato Martins

Email: renato.martins@dcc.ufmg.br

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

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Introdução

- Um programa em C++ consiste de várias partes, tais como funções, tipos definidos pelo usuário, hierarquia de classes, e templates
- A parte mais importante é distinguir entre interface e sua implementação
- C++ representa interfaces por declarações
- Uma declaração especifica tudo que é necessário para usar uma função ou tipo

Introdução

- O ponto chave é que a definição das funções ficam em outro lugar

```
// a função sqrt(raiz quadrada) recebe um double e retorna um double
double srqt(double);

class Vector {
public:
  Vector(int s);
  double& operator[](int i);
  int size();
private:
  double *elem;
  int sz;
}
```

Introdução

- A definição de sqrt() será assim:

```
double sqrt(double d) {
    // algoritmo encontrado em livros de matemática
}
```

- Para Vector, precisamos definir as três funções membros:

```
Vector::Vector(int s) :elem{new double[s]} { // inicializa membros
}
double& Vector::operator[](int i) { // operador p/ acessar com indice
  return elem[i];
}
int Vector::size() {
  return sz;
}
```

Compilação

- C++ suporta a compilação separada onde o usuário só enxerga declarações de tipos e funções usadas
  - A definição desses tipos e funções estão em arquivos separados
- Organiza o programa em fragmentos independentes de códigos
  - Minimiza tempo de compilação
  - Separa logicamente as partes do programa

Compilação

Geralmente colocamos as declarações da interface em um arquivo com um nome que indica seu uso:

```
// Vector.h
class Vector {
public:
  Vector(int s);
  double& operator[](int i);
  int size();
private:
  double *elem;
  int sz;
}
```

Compilação

- O usuário inclui o arquivo, chamado de header, para a acessar a interface:

```
// user.cpp
#include "Vector.h" // usa a interface
#include <cmath> // biblioteca padrao de matematica incluindo sqrt()
using namespace std;
double soma_raiz_quadrada(Vector& v){
  double soma = 0;
  for (int i=0; i != v.size(), ++i) {
      soma += sqrt(v[i]); // soma as raizes quadradas
  }
  return soma;
}
```

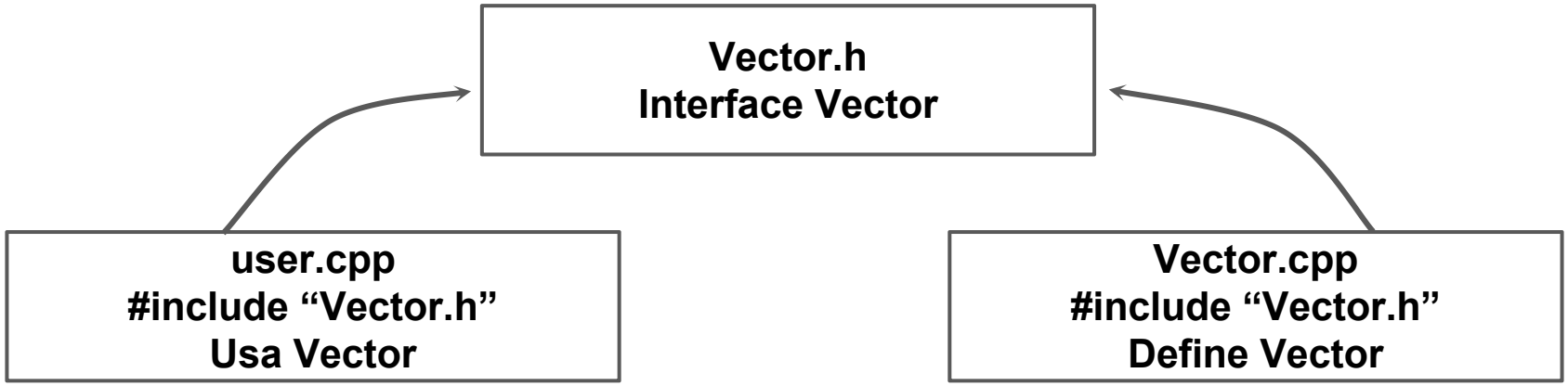
Compilação

- O arquivo .cpp fornecendo a implementação de Vector também inclui o .h

```
// Vector.cpp
#include "Vector.h"

Vector::Vector(int s) :elem{new double[s]} { // inicializa membros
}
double& Vector::operator[](int i) { // operador p/ acessar com indice
  return elem[i];
}
int Vector::size() {
  return sz;
}
```

Compilação

- O código em user.cpp e Vector.cpp compartilham a interface em Vector.h, embora os dois arquivos sejam independentes e possam ser compilados separadamente



Namespaces

- Além de funções, classes, e enumeradores, C++ oferece namespaces como mecanismo de dizer que declarações estão juntas e seus nomes não devem entrar em conflito com outros nomes

```
namespace meu_codigo {
  class complex{ // numero complexo
              //...
  };
  complex sqrt(complex);
  int main();
}
```

```
int meu_codigo::main(){
  complex z{1,2}; // {real, imaginaria}
  auto z2 = sqrt(z); //
  cout << "{" << z2.real() << "," << z2.imag() << "}\n";
}

int main() {
  return meu_codigo::main();
}
```

Namespaces

- Ao colocar o código em um namespace, garantimos que os nomes não entram em conflito com nomes da biblioteca padrão

- A forma mais simples de acessar um nome em outro namespace é qualificá-lo com o nome do namespace (e.g. std::cout e meu_codigo::main)

Namespaces

- Para ter acesso a membros da biblioteca padrão usamos uma diretiva using:
- Uma diretiva faz os nomes do namespace acessíveis como se fossem locais ao escopo em que a diretiva foi usada, assim após a diretiva podemos escrever cout ao invés de std::cout
- Namespaces são usados principalmente para organizar componentes de grandes programas

Tratamento de Erros

- Ao invés de construir programas a partir de tipos da linguagem (e.g., int, char) e controle (e.g., if, while), construímos tipos mais apropriados para nossas aplicações (e.g., string, map) e algoritmos (e.g., sort(), find())

- Tais construções simplificam nossa programação e diminuem a chance de erros (e.g., você não vai aplicar um caminhamento em árvore em uma caixa de diálogo)

Tratamento de Erros

- Um efeito dessa modularização e abstração (em particular, o uso de bibliotecas) é que onde um erro de execução é detectado é separado de onde ele é tratado

- Uma vez que programas cresce, é importante estratégias para lidar com erros desde o início

Exceções

- Considere o exemplo de Vector novamente
- O deve ser se tentarmos acessar um elemento fora do intervalo para o vector?
  - O programador de Vector não sabe o que o usuário gostaria de fazer nesse caso, tipicamente, ele sequer sabe que tipo de programa irá usar a classe
  - O usuário de Vector não pode detectar o erro sempre, se ele pudesse, o erro não aconteceria em primeiro lugar

Exceções

- A solução é o programador de Vector detectar a tentativa de acesso fora do intervalo e então informar o usuário
- O usuário toma a ação apropriada
- Ao detectar o acesso fora de intervalo uma exceção pode ser lançada

Exceções

```
double& Vector::operator[](int i) {
  if (i < 0 || i >=size()){
      throw out_of_range{"Vector::operator[]"};
  }
  return elem[i];
}
```

- O throw transfere o controle para função que chamou Vector::operator[]()

```
void usa_vector(Vector& v){
     try { // as excecoes sao tratadas pelo codigo abaixo
      v[v.size()] = 7; // tenta acessar alem do fim de v
      }
      catch (out_of_range) { // erro out_of_range
       // trata o erro de intervalo
      }
    }
```

Exceções

- O tipo out_of_range é definido na biblioteca padrão (em <stdexcept>) e é de fato usado por algumas bibliotecas padrão para acessar funções
- O uso de mecanismos de tratamento de exceções pode tornar o tratamento de erros, mais simples, mais sistemático, e mais legível, entretanto, não exagere em blocos try

Exceções

- Frequentemente, uma função não consegue terminar sua tarefa após uma exceção ser lançada. Então, “tratar” uma exceção é simplesmente fazer um trabalho mínimo de limpeza e relançá-la novamente