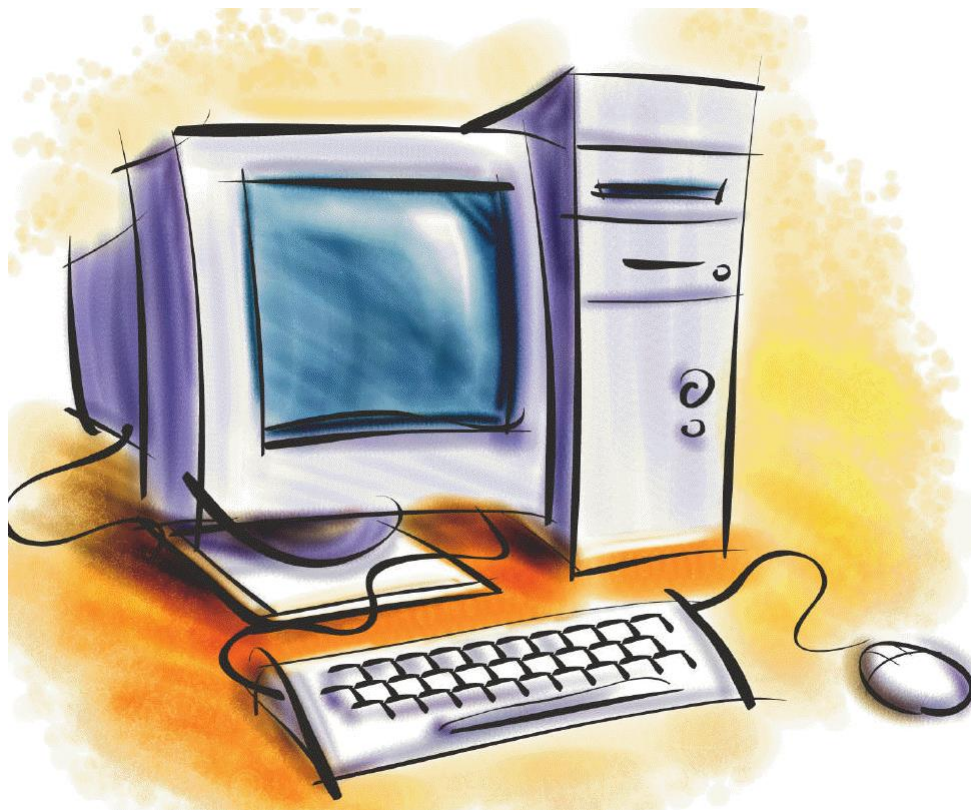


ALGORITMOS E ESTRUTURAS DE DADOS I



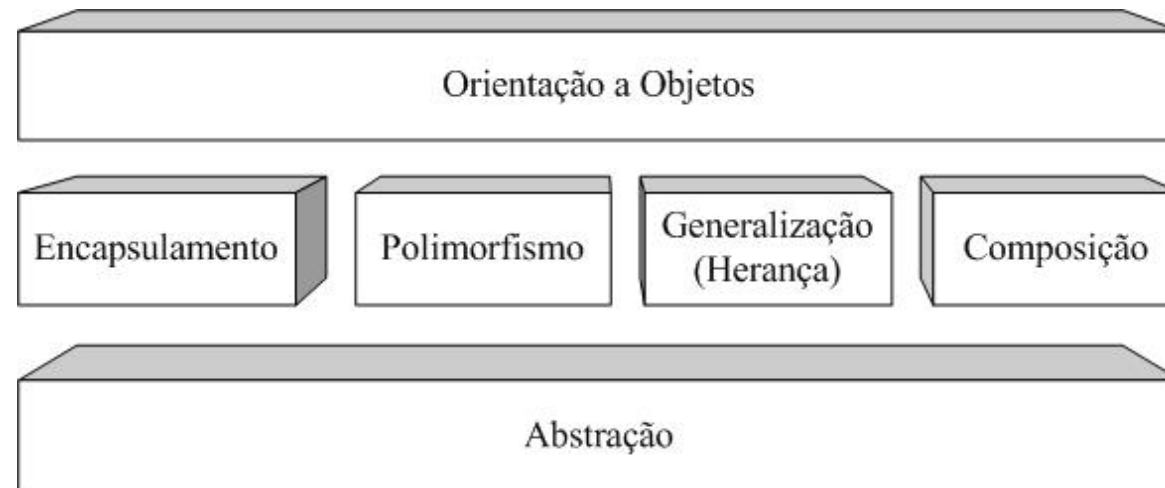
UNIDADE 7

INTRODUÇÃO À ORIENTAÇÃO A OBJETOS

PROF. NAÍSSÉS ZÓIA LIMA

Princípios da Orientação a Objetos

- A programação orientada a objetos (POO) baseia-se em **três princípios básicos**:
 - Encapsulamento
 - Composição / Herança
 - Polimorfismo



Encapsulamento

- Um dos pontos essenciais de POO é o de esconder as estrutura de dados dentro de certas entidades (objetos),
 - aos quais são associados métodos (funções) que manipulam essas estruturas de dados.
- Na orientação a objetos, esconder os detalhes de implementação de uma classe é um conceito conhecido como **encapsulamento**.
- Como os detalhes de implementação da classe estão escondidos, todo o acesso deve ser feito através de seus métodos públicos. Não permitimos aos outros saber **COMO** a classe faz o trabalho dela, mostrando apenas **O QUÊ** ela faz.

Encapsulamento

- Trabalhamos com **struct** até o momento. Structs definem apenas atributos, mas não métodos.
- Para definir métodos, usamos funções externas à struct que recebem uma variável do tipo da struct.
- Em C++, introduziu-se a funcionalidade de classe e todo o suporte a orientação à objetos.
- Com classes, podemos definir atributos bem como métodos, ou seja, objetos possuem características (atributos) e comportamentos (métodos).
- Objetos são variáveis cujos tipos são classes. Ao criarmos uma variável do tipo de uma classe, estamos instanciando um objeto.

Encapsulamento

- Qualificadores de Acesso:

- + **public** : um método definido como public pode ser acessado por qualquer classe de qualquer projeto

- **private** : é mais restritivo, somente a classe onde ele foi definido é que pode acessá-lo, nenhuma outra tem permissão, nem mesmo classes que herdam da classe onde o método foi definido

- # **protected** : somente as classes que herdam da classe que contem o método protegido tem permissão para acessá-lo e as classes que estão no mesmo pacote.

Hello World C++

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Ola mundo!" << endl;
    return 0;
}
```

Classes

- Com classes, podemos definir atributos bem como métodos, ou seja, objetos possuem características (atributos) e comportamentos (métodos).
- Objetos são variáveis cujos tipos são classes. Ao criarmos uma variável do tipo de uma classe, estamos instanciando um objeto.
- Para criar um objeto, usamos uma função específica para criação chamada de **construtor**.
- Todo objeto possui um ponteiro para si mesmo denominado **this**.
- A classe pode ter métodos com mesmo nome, porém com lista de parâmetros distintos: **sobrecarga** de métodos.

Classes

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

```
class Quadrado {
private:
    int lado;

public:
    Quadrado(int l): lado(l) {}

    int getLado() {
        return lado;
    }
}
```

```
int perimetro() {
    return 4*lado;
}

int area() {
    return lado * lado;
}

void print() {
    cout << "[Quadrado, lado=" << this->lado << "]" <<
endl;
}
};
```


Classes

```
int main()
{
    //Constroi quadrado de lado 10
    Quadrado q(10);

    //Imprime o lado do quadrado
    cout << "lado do quadrado = " << q.getLado()
    << endl;

    //Imprime o perimetro do quadrado
    cout << "perimetro do quadrado = " <<
    q.perimetro() << endl;

    //Imprime o area do quadrado
```

```
    cout << "area do quadrado = " << q.area() <<
    endl;

    //Imprime o quadrado
    q.print();

    return 0;
}
```

```
lado do quadrado = 10
perimetro do quadrado = 40
area do quadrado = 100
[Quadrado, lado=10]
```

```
Process returned 0 (0x0)    execution time : 0.056 s
Press any key to continue.
```

Classes

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

```
class Quadrado {
private:
    int lado;

public:
    Quadrado(): lado(0) {}
    Quadrado(int l): lado(l) {}
```

```
    void setLado(int lado) {
        this->lado = lado;
    }
```

```
    int getLado() {
```

```
        return lado;
    }
```

```
    int perimetro() {
        return 4*lado;
    }
```

```
    int area() {
        return lado * lado;
    }
```

```
    void print() {
        cout << "[Quadrado, lado=" << this->lado << "]" <<
endl;
    }
};
```

Classes

```
int main()
{
    //Constroi quadrado
    Quadrado q;

    //Configura o lado do quadrado como 20
    q.setLado(20);

    //Imprime o lado do quadrado
    cout << "lado do quadrado = " << q.getLado() << endl;

    //Imprime o perimetro do quadrado
    cout << "perimetro do quadrado = " << q.perimetro()
    << endl;
```

```
//Imprime o area do quadrado
cout << "area do quadrado = " << q.area() << endl;

//Imprime o quadrado
q.print();

return 0;
```

```
lado do quadrado = 20
perimetro do quadrado = 80
area do quadrado = 400
[Quadrado, lado=20]
```

```
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

Construtores e Destrutores

- A criação de objetos é feita por meio de **construtores**. São funções com o mesmo nome da classe e sem retorno, que inicializam o objeto e seus atributos.
- Construtor default:
 - é um construtor que pode ser chamado sem argumentos;
 - se uma classe não tem nenhum construtor, é criado automaticamente pelo compilador um construtor default;
 - porém, a criação de qualquer construtor pelo programador, faz com que este construtor default não seja mais criado automaticamente!

Construtores e Destrutores

- Destrutor é função complementar às funções construtoras de uma classe. Sempre que o escopo de um objeto encerra-se, esta função é chamada.
- Cada classe pode ter somente um destrutor que jamais recebe parâmetros. O destrutor também não tem nenhum tipo de retorno.

```
class Y {  
    public:  
        ~Y();  
};
```

Alocação Dinâmica

- Em C++, alocação dinâmica de memória é feita usando o operador `new` e `new[]`.

```
pointer = new type
```

```
pointer = new type [number_of_elements]
```

```
int * pInt = new int; //1 inteiro
```

```
int * pInts = new int[5]; //5 inteiros
```

Alocação Dinâmica

- Em C++, desalocação de memória dinâmica é feita usando o operador `delete` e `delete[]`.

```
delete pointer;  
delete[] pointer;
```

```
int * pInt = new int; //1 inteiro  
int * pInts = new int[5]; //5 inteiros
```

```
delete pInt;  
delete[] pInts;
```

Alocação Dinâmica

```
int main()
{
    //Constroi quadrado de lado 5
    Quadrado *q = new Quadrado(5);

    //Imprime o lado do quadrado
    cout << "lado do quadrado = " << q->getLado() <<
endl;

    //Imprime o perimetro do quadrado
    cout << "perimetro do quadrado = " << q-
>perimetro() << endl;

    //Imprime o area do quadrado
    cout << "area do quadrado = " << q->area() << endl;
```

```
//Imprime o quadrado
q->print();
```

```
//Exclui o quadrado
delete q;
```

```
lado do quadrado = 5
perimetro do quadrado = 20
area do quadrado = 25
[Quadrado, lado=5]
```

```
Process returned 0 (0x0)    execution time : 0.012 s
Press any key to continue.
```


Alocação Dinâmica e Destrutor

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

```
class Quadrado {
```

```
private:
```

```
    int *lado;
```

```
public:
```

```
    Quadrado(int l) {
        lado = new int(l);
    }
```

```
    ~Quadrado() {
        delete lado;
    }
```

```
    void setLado(int lado) {
        *(this->lado) = lado;
```

```
    int getLado() {
        return *lado;
    }
```

```
    int perimetro() {
        return 4*(*lado);
    }
```

```
    int area() {
        return (*lado) * (*lado);
    }
```

```
    void print() {
        cout << "[Quadrado, lado=" << *(this->lado) << "]"
        << endl;
    }
```

Alocação Dinâmica e Destrutor

```
int main()
{
    //Constroi quadrado de lado 100
    Quadrado q(100);

    //Imprime o lado do quadrado
    cout << "lado do quadrado = " << q.getLado() <<
endl;

    //Imprime o perimetro do quadrado
    cout << "perimetro do quadrado = " << q.perimetro()
<< endl;

    //Imprime o area do quadrado
    cout << "area do quadrado = " << q.area() << endl;

    //Imprime o quadrado
    q.print();

    return 0;
}
```

```
lado do quadrado = 100
perimetro do quadrado = 400
area do quadrado = 10000
[Quadrado, lado=100]
```

```
Process returned 0 (0x0)   execution time : 0.013 s
Press any key to continue.
```

Membros Estáticos - static

- Os atributos definidos na classe são independentes por objeto, isto é, cada objeto possui um valor próprio e independente para seus atributos (locais de memória separados). São chamadas variáveis de instância.
- É possível criar variáveis que são compartilhadas por todos os objetos da classe, isto é, todos os objetos farão referência a essa variável que terá o mesmo valor (mesmo local de memória). São chamadas variáveis estáticas ou variáveis de classe. Devem ser inicializados fora da classe (são inicializadas uma única vez).
- Para criar variáveis estáticas, usamos a palavra **static** na sua definição.
- Métodos de instância: acessam atributos estáticos e não estáticos.
- Métodos estáticos: acessam apenas atributos estáticos. Podem ser chamados diretamente da classe ao invés de um objeto. Dentro de um método estático, não existe o ponteiro **this**.

Membros Estáticos - static

```
class Car
{
    private:
        static int total;
    public:
        Car();
        ~Car();
        static int getTotal();
};
```

```
int Car::total = 0;
```

```
Car::Car()
{
    total++;
}
```

```
Car::~~Car()
{
    total--;
}
```

```
int Car::getTotal()
{
    return total;
}
```

Membros Estáticos - static

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

int main()
{
    Car *ptr1, *ptr2, *ptr3;
    ptr1 = new Car();
    ptr2 = new Car();
    ptr3 = new Car();
    cout<<"Numero de objetos instanciados: "<<
Car::getTotal() <<endl;
    delete ptr3;
    cout<<"Numero de objetos instanciados: "<<
ptr1->getTotal() <<endl;
    delete ptr1;
```

```
    cout<<"Numero de objetos instanciados: "<<
Car::getTotal() <<endl;
    delete ptr2;
    cout<<"Numero de objetos instanciados: "<<
Car::getTotal() <<endl;
    return 0;
}
```

```
Numero de objetos instanciados: 3
Numero de objetos instanciados: 2
Numero de objetos instanciados: 1
Numero de objetos instanciados: 0
```

```
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

Membros Estáticos - static

```
#ifndef CAR_H
#define CAR_H

class Car {
    private:
        static int total;
        char modelo[50];

    public:
        Car(const char*);
        ~Car();
        static int getTotal();
        const char* getModelo();
};

#endif // CAR_H
```

Membros Estáticos - static

```
#include "Car.h"
#include <string.h>

int Car::total = 0;

Car::Car(const char* modelo)
{
    strcpy(this->modelo, modelo);
    total++;
}

Car::~~Car()
{
    total--;
}

int Car::getTotal()
{
    return total;
}

const char* Car::getModelo()
{
    return modelo;
}
```

Membros Estáticos - static

```
#include <iostream>
#include "Car.h"
using namespace std;

int main()
{
    //Criando o primeiro carro
    char modelo1[50];
    cout << "Digite o modelo do carro 1: ";
    cin >> modelo1;
    Car carro1(modelo1);
    cout << "Modelo do carro 1: " << carro1.getModelo()
    << endl;
    cout << "Numero de objetos instanciados: " <<
    Car::getTotal() << endl;

    //Criando o primeiro carro
    char modelo2[50];
    cout << "Digite o modelo do carro 2: ";
    cin >> modelo2;
    Car *ptrCarro2 = new Car(modelo2);
    cout << "Modelo do carro 2: " <<
    ptrCarro2->getModelo() << endl;
    cout << "Numero de objetos instanciados: " <<
    Car::getTotal() << endl;

    delete ptrCarro2;

    cout << "Numero de objetos instanciados: " <<
    Car::getTotal() << endl;
    return 0;
}
```


Membros Estáticos - static

```
Digite o modelo do carro 1: audi
Modelo do carro 1: audi
Numero de objetos instanciados: 1
Digite o modelo do carro 2: bmw
Modelo do carro 2: bmw
Numero de objetos instanciados: 2
Numero de objetos instanciados: 1

Process returned 0 (0x0)    execution time : 6.676 s
Press any key to continue.
```

Variáveis Estáticas – static – em Funções

- Podemos definir variáveis estáticas dentro de funções.
- São inicializadas uma única vez e mantêm seu valor em chamadas futuras à função.
- Úteis em situações que devemos manter um estado em chamadas sucessivas à função.
- Exemplo: strtok

Variáveis Estáticas – static – em Funções

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

```
void printCalls() {
    static int count = 0;
    count++;
    cout << "Calls: " << count << endl;
}
```

```
int main()
{
    for(int i = 0; i < 5; i++) {
        printCalls();
    }
    return 0;
}
```

```
Calls: 1
Calls: 2
Calls: 3
Calls: 4
Calls: 5
```

```
Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.
```

Strings

- C++ define um tipo próprio para trabalhar com strings, chamado **string**
- Possuem métodos próprios de manipulação e facilita trabalhar com cadeia de caracteres, em evolução ao C

Strings

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    //Criando string
    string s1 = "ola mundo";
    cout << "string 1: " << s1 << endl;
    cout << "tamanho da string 1: " << s1.size() << endl;

    //Lendo string
    string s2;
    cout << "Digite um texto: ";
    getline(cin, s2);
    cout << "string 2: " << s2 << endl;

    cout << "tamanho da string 2: " << s2.length() <<
    endl;

    //Concatenando strings
    string s3 = s1 + " " + s2;
    cout << "string resultante: " << s3 << endl;
    cout << "tamanho da string resultante: " << s3.size()
    << endl;
    return 0;
}
```

```
string 1: ola mundo
tamanho da string 1: 9
Digite um texto: do aeds
string 2: do aeds
tamanho da string 2: 7
string resultante: ola mundo do aeds
tamanho da string resultante: 17

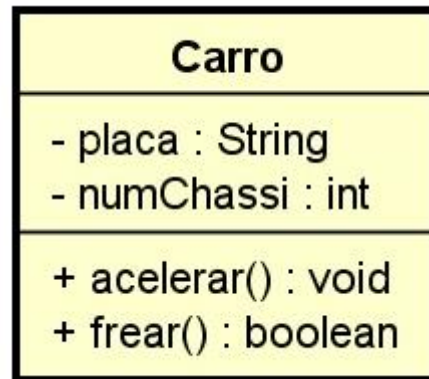
Process returned 0 (0x0)   execution time : 3.858 s
Press any key to continue.
```

Diagrama de Classes

- Diagrama de classes é uma representação da estrutura e relações das classes que compõem um sistema de software orientado a objetos.
- Apresenta uma visão estática de como as classes estão organizadas.
- É um dos diagramas da UML (Unified Modeling Language).
- UML - Linguagem de Modelagem Unificada) é uma linguagem-padrão de notação para a elaboração da estrutura de projetos de software.

Diagrama de Classes

- Classe Carro



- Atributos:

- placa: do tipo string, privado
- numChassi: do tipo int, privado

- Métodos:

- acelerar: público, não recebe parâmetros, sem retorno (void)
- frear: público, não recebe parâmetros, retorna um valor lógico (boolean)

Diagrama de Classes

- Exercício: Para a classe Carro vista, acrescente um atributo para a velocidade. Desenhe o diagrama de classe correspondente.
- Implemente a classe em C++, de forma que:
 - Um objeto carro é criado sempre parado
 - Ao acelerar, a sua velocidade aumenta 10 km/h
 - Ao frear, a sua velocidade reduz 10 km/h
- Crie um programa para testar seu carro. No programa, o carro deve acelerar até atingir uma velocidade de 110 km/h. Após, o carro deve reduzir sua velocidade para 60 km/h.

Diagrama de Classes

```
#include <iostream>
#include <string>
using namespace std;
class Carro {
private:
    string placa;
    int numChassi;
    int velocidade;
public:
    Carro(string placa) {
        this->placa = placa;
        velocidade = 0;
    }
    int getVelocidade() {
        return velocidade;
    }
};
```

```
void acelerar() {
    velocidade += 10;
}
int frear() {
    if(velocidade > 0)
        velocidade -= 10;
    return velocidade == 0;
}
void imprimir() {
    cout << "Carro placa " << placa << " velocidade "
        << velocidade << " Km/h" << endl;
}
};
```

Diagrama de Classes

```
int main()
{
    Carro carro("ABC1234");
    carro.imprimir();
    cout << "Acelerando..." << endl;
    for(int i = 0; i < 11; i++)
        carro.acelerar();
    carro.imprimir();
    cout << "Freando..." << endl;
    for(int i = 0; i < 5; i++)
        carro.frear();
    carro.imprimir();
    return 0;
}
```

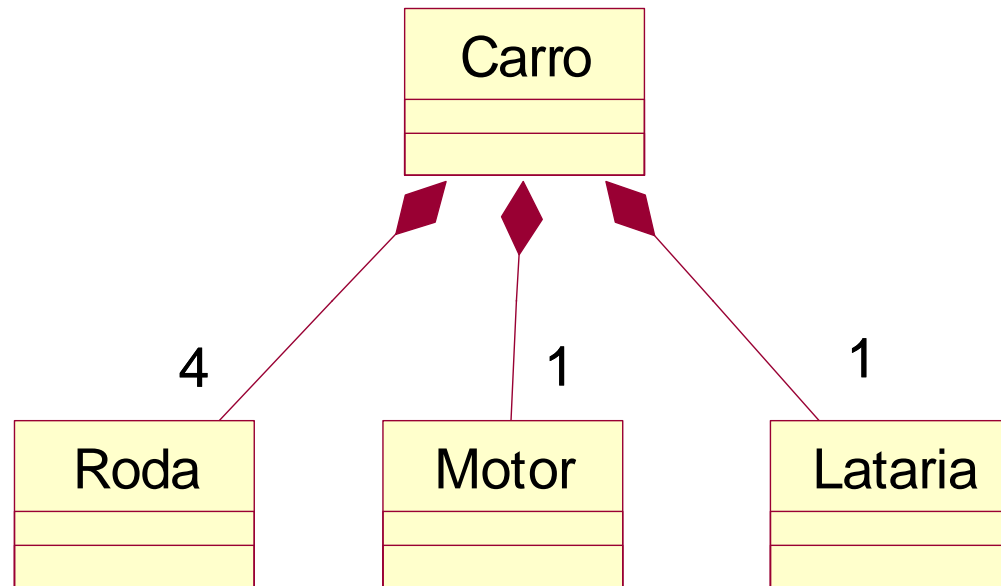
```
Carro placa ABC1234 velocidade 0 Km/h
Acelerando...
Carro placa ABC1234 velocidade 110 Km/h
Freando...
Carro placa ABC1234 velocidade 60 Km/h

Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

Composição

- Na composição utilizam-se objetos das classes existentes dentro da nova classe: a nova classe é composta de objetos de classes existentes.
- Trata-se de uma forma de reutilização do paradigma OO.
- É um tipo de relacionamento entre classes.
- Como se identifica a composição?
 - Identifica-se a possibilidade de composição através dos seguintes verbos típicos: conter, possuir. Ex: Um carro contém 4 rodas, 1 motor, 1 lataria, ...
- A composição também é chamada de relacionamento do tipo parte-todo.
- O lado todo do relacionamento possui as partes.
- Trata-se de um relacionamento forte, pois no caso do objeto todo deixar de existir, as partes também deixam.

Composição



Herança

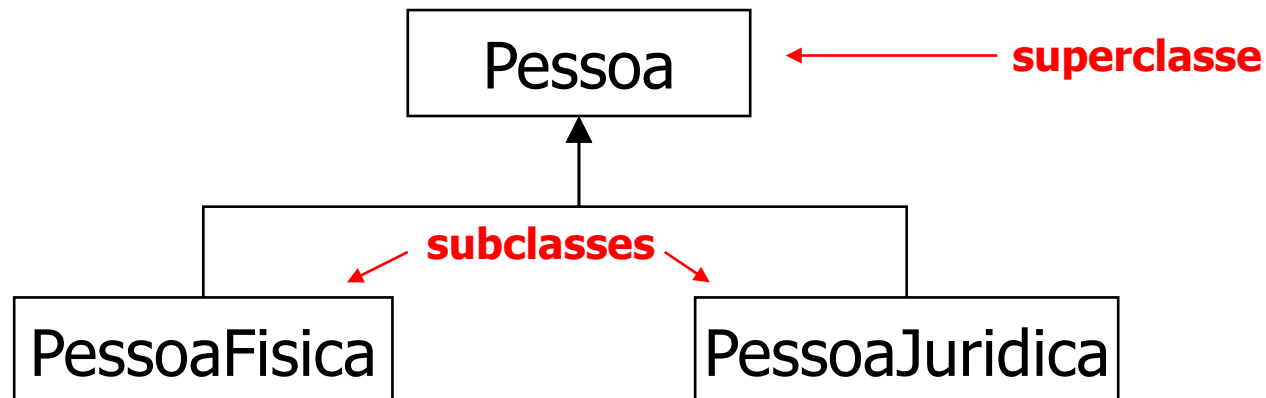
- Muitas vezes, classes diferentes têm características comuns
- Por exemplo:
 - Imagine uma classe **Pessoa** com as seguintes características:
 - nome, endereço, telefone
 - Se imaginarmos uma pessoa física, esta possui as seguintes características:
 - nome, endereço, telefone, CPF, RG
 - Se imaginarmos uma pessoa jurídica, esta possui as seguintes características:
 - nome, endereço, telefone, CNPJ, IE
- Então, ao invés de criarmos uma nova classe para pessoa física e uma nova para pessoa jurídica, com todas essas características, podemos usar as características da classe Pessoa já existente.

Herança

- Trata-se de uma forma de reutilização do paradigma OO.
- É um tipo de relacionamento entre classes.
- Herança é um dos conceitos fundamentais de POO
- É um recurso que permite que novas classes sejam definidas a partir de uma classe já definida
- Na prática, significa a possibilidade de construir classes especializadas que herdam as características de classes mais generalistas, podendo adicionar novas características
- Permite o reuso do comportamento de uma classe na definição de outra

Herança

- **Superclasses** (ou ascendente): são as ascendentes de um classe
 - Também chamada de classe pai (mãe) ou classe base
- **Subclasses** (ou descendente): são as descendentes de um classe
 - Também chamada de classe filho (filha) ou classe derivada



PessoaFisica é **um tipo** de Pessoa
PessoaJuridica é **um tipo** de Pessoa

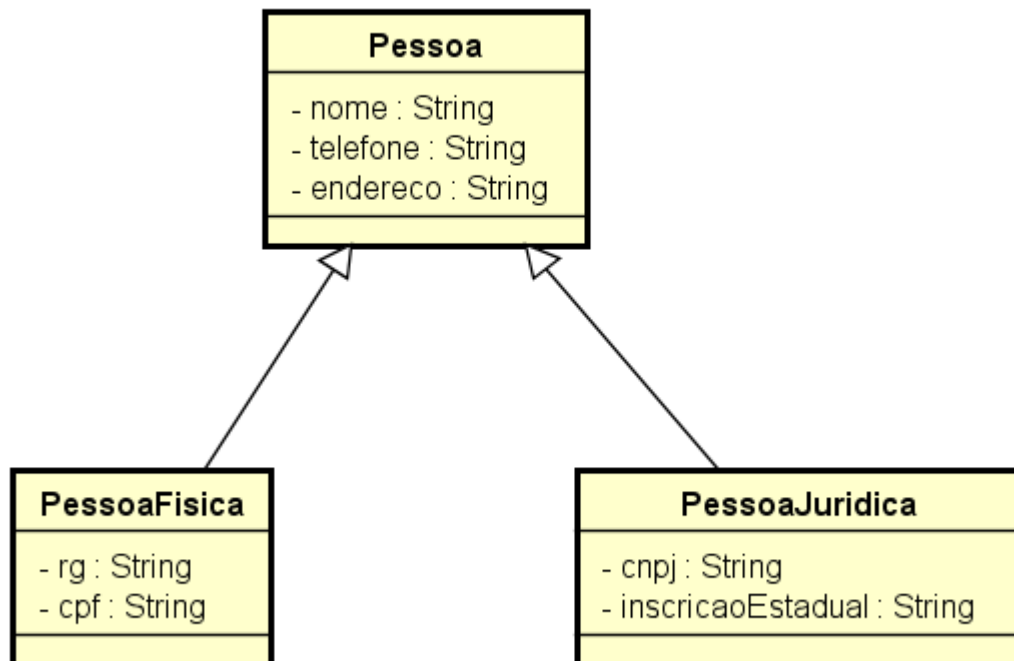
Herança

- Em C++, definimos uma classe derivada de uma classe base da seguinte forma:

```
class Subclasse : public Superclasse {  
    //definições da subclasse  
};
```


Herança

- Implemente as classes em C++ conforme diagrama de classes a seguir. Por simplificação, na classe base considere apenas o atributo nome e, nas classes derivadas, apenas cpf e cnpj.



Atenção:

- Se um atributo na classe Base é privado (private), ele não é acessível nas subclasses.

- Se um atributo na classe Base é protegido (protected), ele é acessível nas subclasses. Vamos implementar, neste exemplo, os atributos em Pessoa como protected!

Herança

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

```
class Pessoa {
protected:
    string nome;


public:
    Pessoa(string nome){
        this->nome = nome;
    }
};
```

```
class PessoaFisica : public Pessoa {
    string cpf;

public:
    PessoaFisica(string nome, string cpf): Pessoa(nome) {
        this->cpf = cpf;
    }

    void imprimir() {
        cout << "[PessoaFisica, nome=" << nome << ",
            cpf=" << cpf << "]" << endl;
    }
};
```

Chamando o construtor da superclasse



Herança

```
class PessoaJuridica : public Pessoa {
    string cnpj;

public:
    PessoaJuridica(string nome, string cnpj): Pessoa(nome) {
        this->cnpj = cnpj;
    }

    void imprimir() {
        cout << "[PessoaJuridica, nome=" << nome << ",
                cnpj=" << cnpj << "]" << endl;
    }
};
```

```
int main()
{
    PessoaFisica pessoaFisica("Joao da Silva",
                               "123456789-00");
    pessoaFisica.imprimir();
    PessoaJuridica pessoaJuridica("Empresa do Joao SA",
                                    "12.299.535/0001-94");
    pessoaJuridica.imprimir();
    return 0;
}
```

```
[PessoaFisica, nome=Joao da Silva, cpf=123456789-00]
[PessoaJuridica, nome=Empresa do Joao SA, cnpj=12.299.535/0001-94]

Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.
```

Polimorfismo

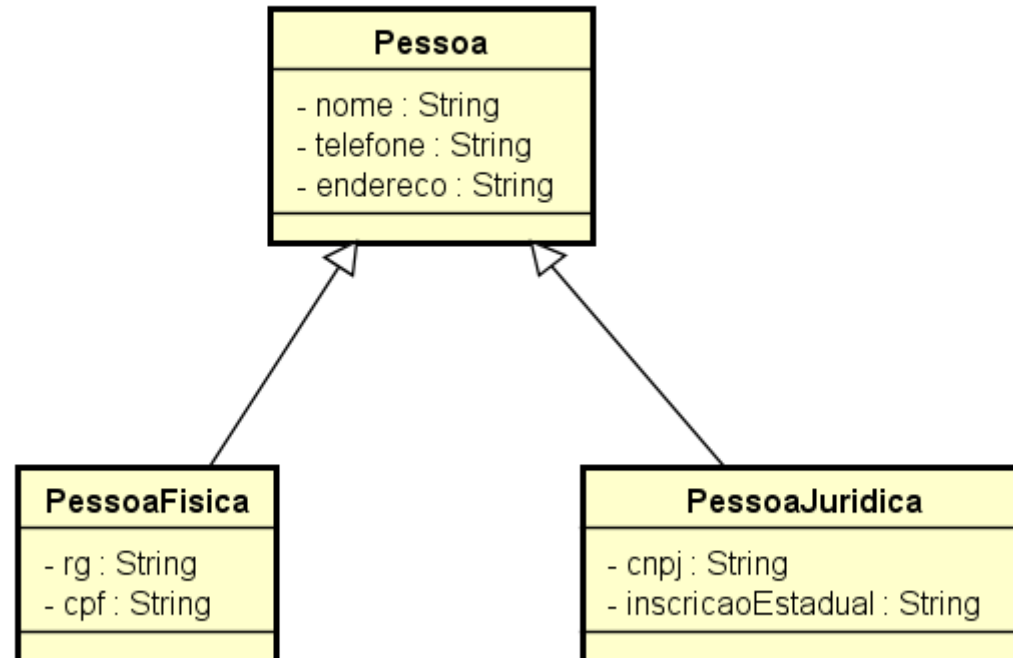
- Do grego poli + morphos = múltiplas formas
- Em termos de programação, significa que **um único nome** de classe ou de **método** pode ser usado para **representar comportamentos diferentes**.
- **A decisão** sobre qual comportamento utilizar é **tomada em tempo de execução**.

Polimorfismo

- Clientes das classes podem ser implementados genericamente para chamar uma operação de um objeto sem saber o tipo do objeto.
- Se são criados novos objetos que suportam uma mesma operação, o cliente não precisa ser modificado para suportar o novo objeto.
- O polimorfismo torna a programação orientada por objetos eficaz, permitindo a escrita de código genérico, fácil de manter e de estender.

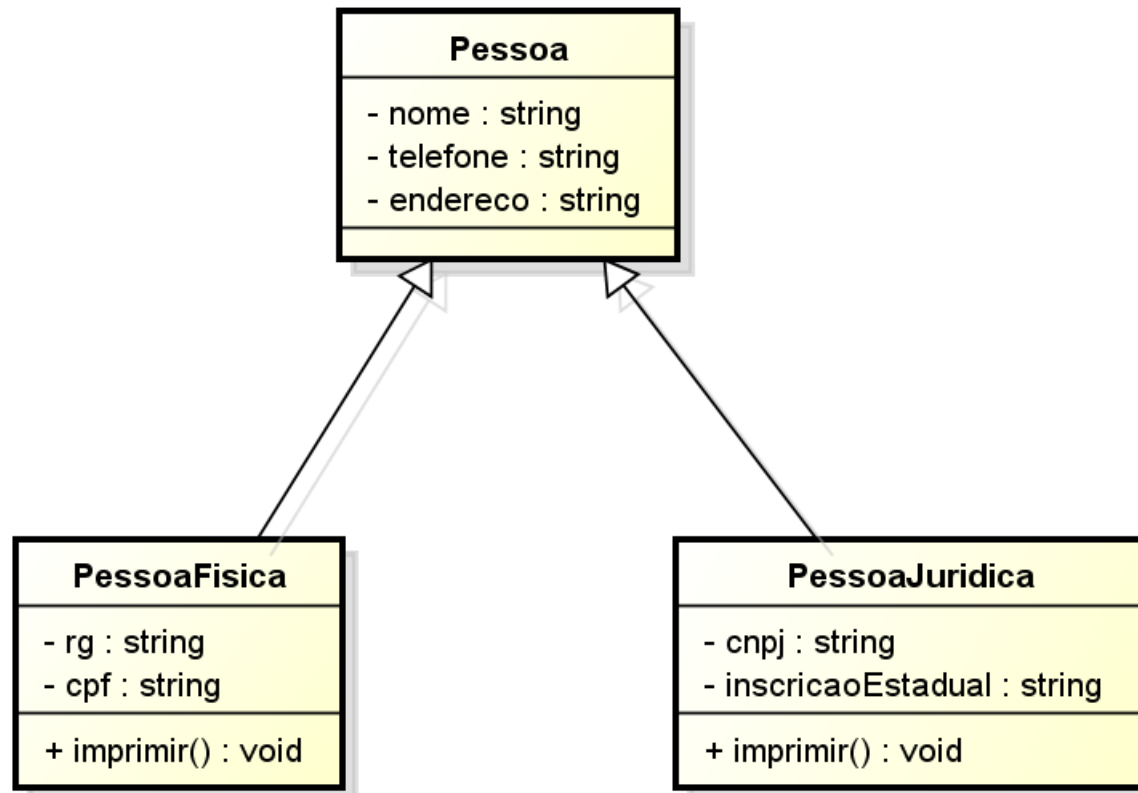
Polimorfismo

- Suponha o mesmo exemplo anterior da hierarquia de Pessoa.



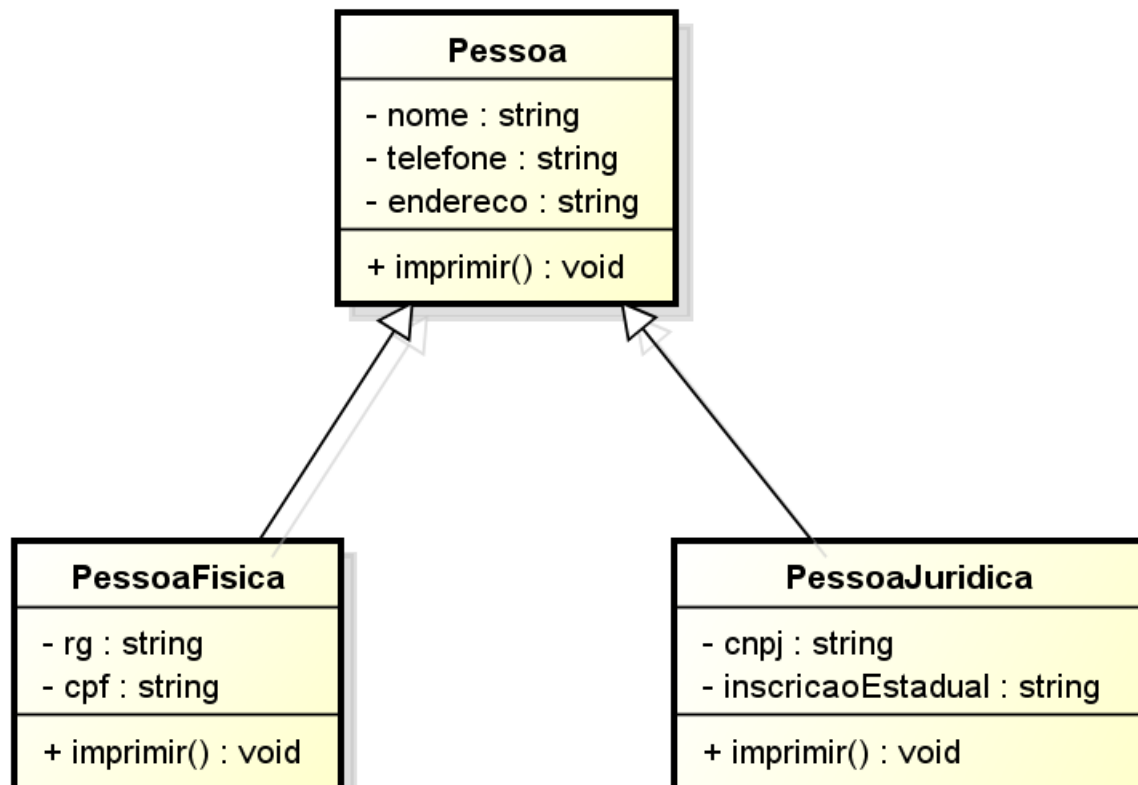
Polimorfismo

- Vamos supor que queremos disponibilizar um método para imprimir uma PessoaFisica ou uma PessoaJuridica. Portanto:



Polimorfismo

- Podemos fazer melhor! Vamos disponibilizar também o método imprimir na superclasse Pessoa! Com isso, podemos imprimir uma pessoa sem precisar saber a sua natureza (física ou jurídica).



Isso é Polimorfismo!

- A pessoa ora irá se comportar como uma PessoaFisica, ora como uma PessoaJuridica.

- O polimorfismo ocorre em tempo de execução ao chamar o método imprimir sobre um ponteiro para Pessoa.

- O método imprimir() de Pessoa está sendo sobrescrito nas subclasses.

Polimorfismo

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

class Pessoa {
protected:
    string nome;
public:
    Pessoa(string nome){
        this->nome = nome;
    }
    virtual ~Pessoa() {}
    virtual void imprimir(){
        cout << "[Pessoa, nome=" << nome << "]" << endl;
    }
};
```

```
class PessoaFisica : public Pessoa {
    string cpf;

public:
    PessoaFisica(string nome, string cpf): Pessoa(nome) {
        this->cpf = cpf;
    }

    void imprimir() {
        cout << "[PessoaFisica, nome=" << nome << ", cpf=" <<
            cpf << "]" << endl;
    }
};
```

Polimorfismo

```
class PessoaJuridica : public Pessoa {
    string cnpj;

public:
    PessoaJuridica(string nome, string cnpj): Pessoa(nome) {
        this->cnpj = cnpj;
    }

    void imprimir() {
        cout << "[PessoaJuridica, nome=" << nome << ", cnpj="
            << cnpj << "]" << endl;
    }
};
```

```
int main()
{
    PessoaFisica pessoaFisica("Joao da Silva", "123456789-00");
    PessoaJuridica pessoaJuridica("Empresa do Joao SA",
                                    "12.299.535/0001-94");
    Pessoa* pessoa;
    pessoa = &pessoaFisica;
    pessoa->imprimir();
    pessoa = &pessoaJuridica;
    pessoa->imprimir();
    return 0;
}
```

```
[PessoaFisica, nome=Joao da Silva, cpf=123456789-00]
[PessoaJuridica, nome=Empresa do Joao SA, cnpj=12.299.535/0001-94]

Process returned 0 (0x0)   execution time : 0.014 s
Press any key to continue.
```

Polimorfismo

```
void imprimirPessoa(Pessoa* pessoa) {  
    pessoa->imprimir();  
}
```

```
int main()  
{  
    PessoaFisica pessoaFisica("Joao da Silva", "123456789-00");  
    PessoaJuridica pessoaJuridica("Empresa do Joao SA",  
        "12.299.535/0001-94");  
    imprimirPessoa(&pessoaFisica);  
    imprimirPessoa(&pessoaJuridica);  
    return 0;  
}
```

```
[PessoaFisica, nome=Joao da Silva, cpf=123456789-00]  
[PessoaJuridica, nome=Empresa do Joao SA, cnpj=12.299.535/0001-94]  
  
Process returned 0 (0x0)    execution time : 0.015 s  
Press any key to continue.
```

Tratamento de Exceção

- O que é uma Exceção?
 - Uma exceção é um evento indesejado ou inesperado que ocorre durante a execução do programa, rompendo o fluxo de execução normal das instruções do programa.
- Por que tratar Exceções?
 - Se as exceções não forem tratados, a aplicação é interrompida e encerrada. Espera-se isso de uma aplicação?
 - Seja qual for a causa da falha, espera-se que a aplicação seja capaz de detectar que houve uma situação anormal de funcionamento e gerenciar essa anormalidade de maneira adequada.

Tratamento de Exceção

- Usamos tratamento de exceção com as cláusulas
 - throw: lança uma exceção.
 - try-catch: executa código que pode lançar exceção (try) e, caso seja lançada uma exceção, captura e trata (catch).

```
try {  
    // protected code  
} catch( ExceptionName e1 ) {  
    // catch block  
} catch( ExceptionName e2 ) {  
    // catch block  
} catch( ExceptionName eN ) {  
    // catch block  
}
```

Tratamento de Exceção

```
#include <iostream>
#include <exception>

using namespace std;

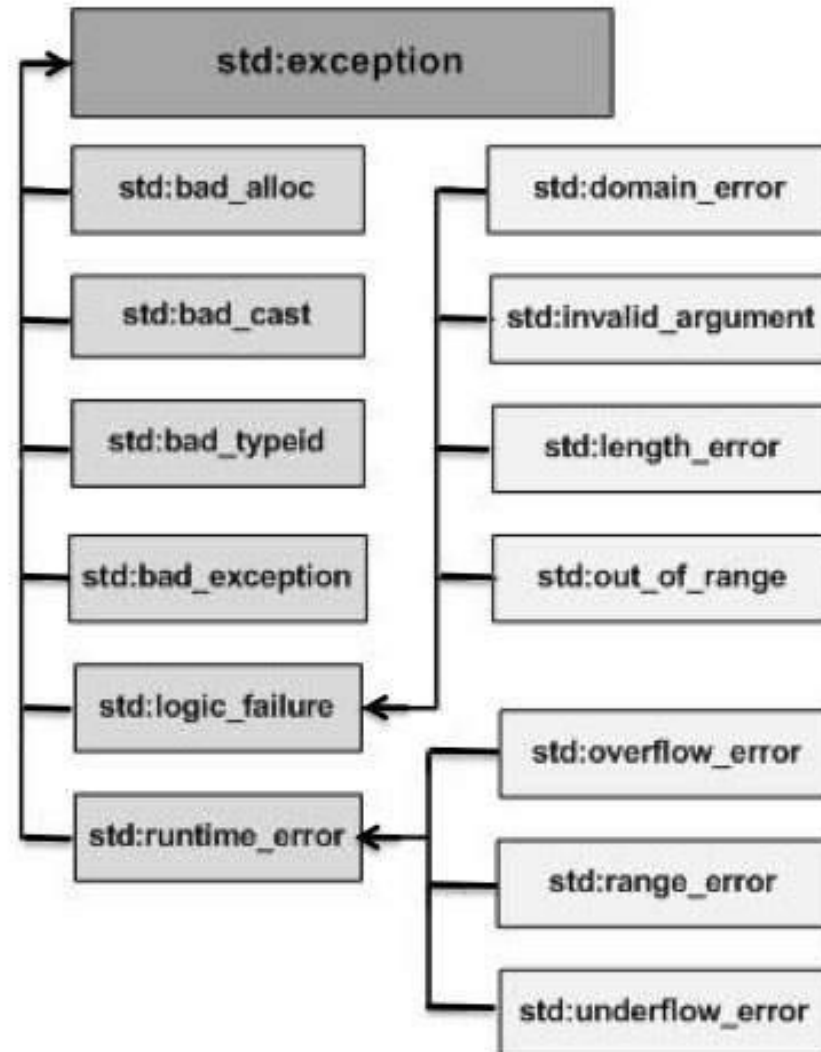
int main() {

    try {
        int x, y, z;
        cout << "Digite dois numeros inteiros: ";
        cin >> x >> y;
        if(y == 0) {
            throw exception();
        } else {
            z = x / y;
        }

        cout << "x/y = " << z << endl;
    }
    catch (exception& e) {
        cout << "excecao ocorreu: divisao por zero" << endl;
    }

    return 0;
}
```

Tratamento de Exceção



Tratamento de Exceção

```
#include <iostream>
#include <exception>
using std::cout;
using std::endl;
using std::bad_alloc;

int main()
{
    double *ptr;
    try
    {
        while (1)
        {
            cout << "Tentando Alocar...\n";
            ptr = new double[500000];
        }
    }
    catch (bad_alloc E)
    {
        cout << "Sem Memoria...\n" <<
        endl;
    }
    for(int i=0; i< 100;i++)
        cout << "Fim..." << endl;

    return 0;
}
```


Tratamento de Exceção

```
#include <iostream>    // std::cerr
#include <stdexcept>    // std::out_of_range
#include <vector>       // std::vector

int main (void) {
    std::vector<int> myvector(10);
    try {
        myvector.at(20)=100;    // vector::at throws an out-of-range
    }
    catch (const std::out_of_range& oor) {
        std::cerr << "Out of Range error: " << oor.what() << '\n';
    }
    return 0;
}
```

Tratamento de Exceção

Sr.No	Exception & Description	Sr.No	Exception & Description
1	<code>std::exception</code> An exception and parent class of all the standard C++ exceptions.	8	<code>std::invalid_argument</code> This is thrown due to invalid arguments.
2	<code>std::bad_alloc</code> This can be thrown by <code>new</code> .	9	<code>std::length_error</code> This is thrown when a too big <code>std::string</code> is created.
3	<code>std::bad_cast</code> This can be thrown by <code>dynamic_cast</code> .	10	<code>std::out_of_range</code> This can be thrown by the 'at' method, for example a <code>std::vector</code> and <code>std::bitset<>::operator[]()</code> .
4	<code>std::bad_exception</code> This is useful device to handle unexpected exceptions in a C++ program.	11	<code>std::runtime_error</code> An exception that theoretically cannot be detected by reading the code.
5	<code>std::bad_typeid</code> This can be thrown by <code>typeid</code> .	12	<code>std::overflow_error</code> This is thrown if a mathematical overflow occurs.
6	<code>std::logic_error</code> An exception that theoretically can be detected by reading the code.	13	<code>std::range_error</code> This is occurred when you try to store a value which is out of range.
7	<code>std::domain_error</code> This is an exception thrown when a mathematically invalid domain is used.	14	<code>std::underflow_error</code> This is thrown if a mathematical underflow occurs.