

Estruturas de Informação

Departamento de Engenharia Informática (DEI/ISEP)

Fátima Rodrigues

mfc@isep.ipp.pt

Aspetos Essenciais C++

Linguagem C

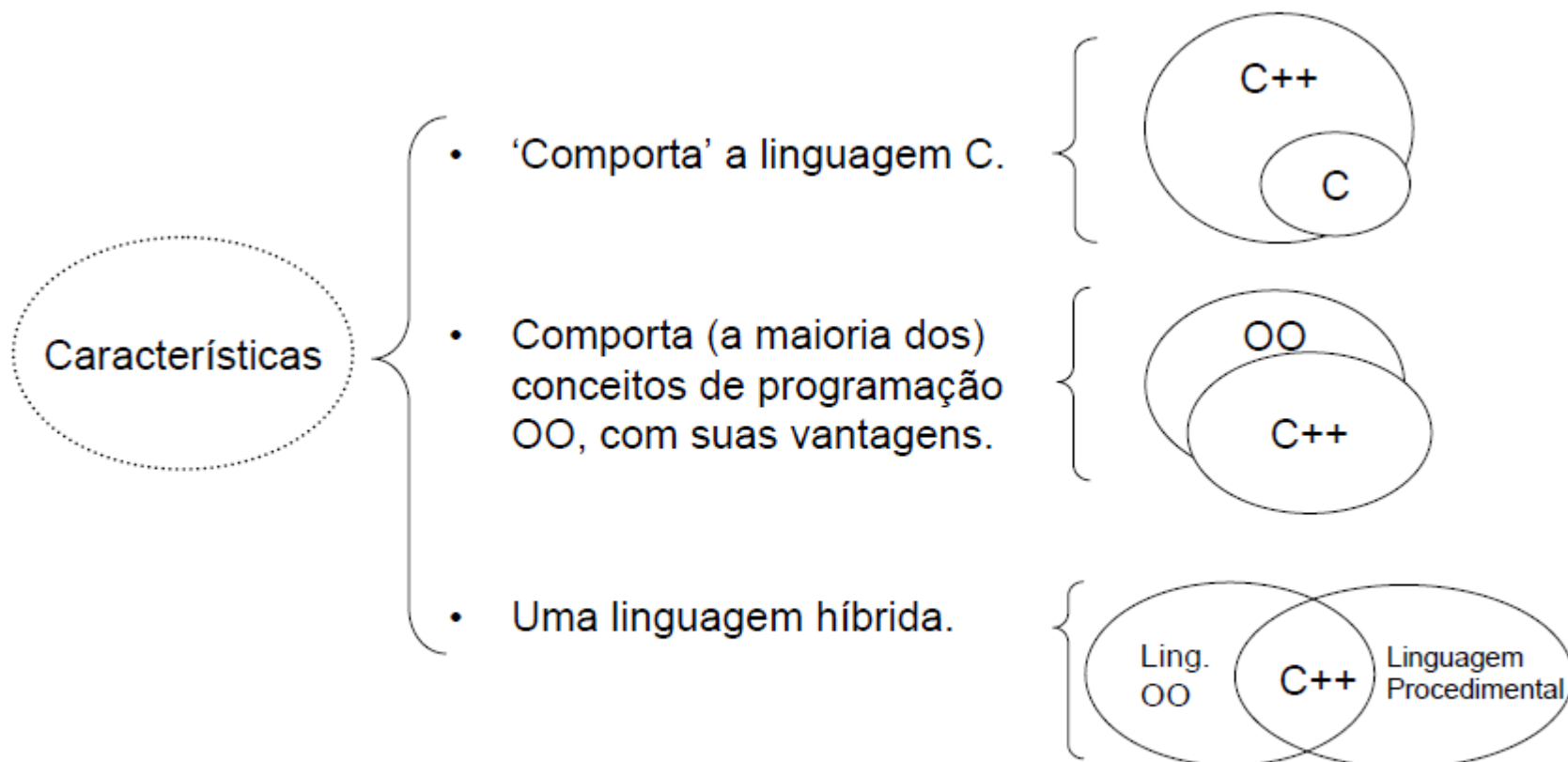
- Linguagem clássica, amplamente utilizada
- Sintaxe muito difundida, tem servido como inspiração tecnológica

Características

- Uma linguagem multinível:
 - *Permite compor programas com abordagens variando entre 'baixo e alto nível'*
- Organização:
 - *Funções e estruturas de dados*
 - Divisão de código fonte em diversos arquivos (.h e .c)
- Flexibilidade
 - *Apontadores: Permite a independência de memória pré-alocada*
- Paradigma imperativo-procedimental

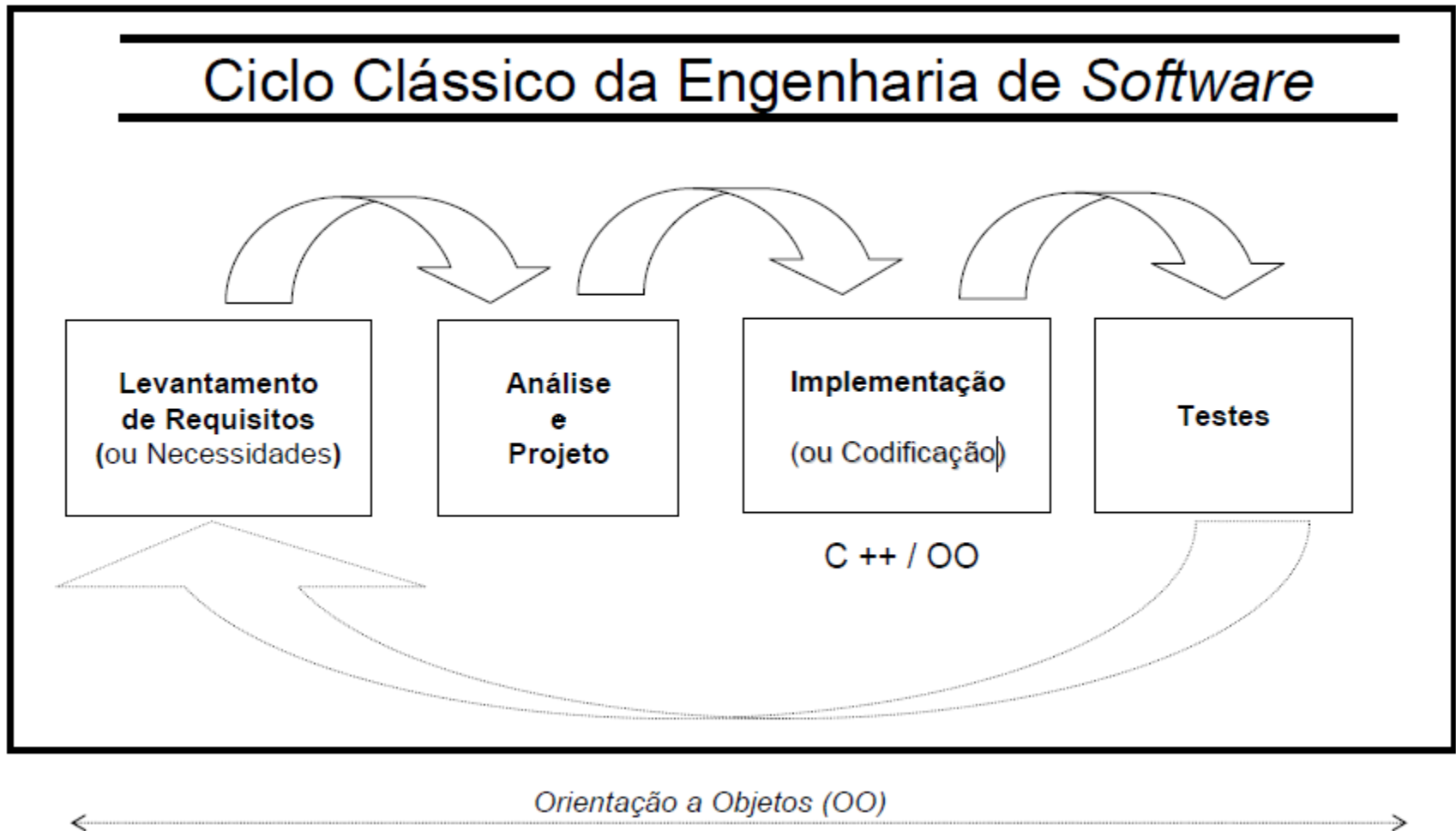
Linguagem C++

- É uma evolução da linguagem C...



C++ : Uma linguagem flexível

C++ vs. Engenharia de Software



C++ vs. Engenharia de Software

- Linguagem de Projeto Orientada a Objetos (OO):
 - UML (Unified Modeling Language)
 - Os conceitos da UML são, em geral, suportados pelo C++
- A maioria das ferramentas orientadas ao projeto e implementação - **Ferramentas C.A.S.E.** (Computer Aided Software Engineering) suportam UML, C++ e linguagens similares (Java e C#)
 - A maioria das ferramentas C.A.S.E suportam parte de geração de código (em C++) a partir de diagramas UML
 - Exemplos de Ferramentas C.A.S.E.: System Architect, Mega, Rational Rose e Star UML

Escrita de um Programa em C++

1. Desenho

- Definem-se as classes necessárias ao sistema a implementar
- Decidem-se as interações entre as classes
- Os dados que cada classe armazena
- As operações que cada classe disponibiliza

2. Codificação

- O código deve ser escrito de modo a ser fácil de ler e compreender
- Usar nomes significativos para as variáveis
- Usar constantes e enumerações em vez de valores incorporados
- Indentar blocos de instrução
- Usar comentários para explicar partes do código

3. Teste e Debugging

- Um plano de testes cuidadoso é uma parte essencial na escrita de um programa

Classes

Conceito de Classe

- O 1º objetivo do conceito de classe em C++ é colocar ao dispor do programador uma ferramenta para a criação de novos tipos de dados que podem ser usados de forma semelhante aos tipos de dados *built-in*
- Um tipo de dados é uma representação concreta de um conceito

Exemplo:

- o tipo **float** (*built-in*) do C++ com as suas operações $+$, $-$, $*$, \dots ,
- proporciona uma aproximação concreta ao conceito matemático de **número real**
- Os detalhes da representação interna de um float (1 byte para a expoente, 3 bytes para a mantissa, etc.) são escondidos \Rightarrow **encapsulamento**

Conceito de Classe

- Novos tipos de dados são projetados para representar conceitos da aplicação que não têm representação direta nos tipos *built-in*

Exemplo:

- Em muitas aplicações interessa poder definir um novo tipo **Data**
- Interessa poder esconder os detalhes da representação interna de uma data (três inteiros para o dia, mês e ano, ou um único inteiro com o número de dias decorridos desde uma data de referência)
⇒ **encapsulamento**
- Interessa poder usar os operadores -, ==, += , <<, >>, ... para subtrair, comparar, incrementar, escrever e ler datas
⇒ **sobrecarga (*overloading*) de operadores**

Conceito de Classe

- Uma classe (em *sentido lato*) é um tipo de dados definido pelo programador [Stroustrup, 1983]
 - o programador pode criar novos tipos de dados (classes em sentido lato) usando a palavra-chave **class**
 - tipos de dados definidos em bibliotecas *standard* são classes
- Para além de dados (*membros-dados*), uma classe pode também conter:
 - funções de manipulação desses dados (*membros-funções*),
 - restrições de acesso a dados e funções,
 - redefinições de quase todos os operadores de C++ para objectos da classe

Exemplo: Classe Ponto


```
class Ponto {  
    private:  
        double x, y ; // abcissa, ordenada  
  
    public:  
        Ponto() { // Construtor por defeito  
            x = 0 ;  
            y = 0 ; }  
  
        Ponto(double x, double y) { // Construtor com parâmetros  
            x = x ;  
            y = y ; }  
  
        Ponto(const Ponto& p) { // Construtor Cópia  
            x = p.X() ;  
            y = p.Y() ; }  
  
        ~Ponto() { }  
  
        double getX() const {  
            return x ; }  
  
        double getY() const {  
            return y ; }  
}
```

- Os membros privados só são visíveis pelos membros-função e amigos da classe
- Os membros públicos são visíveis por qualquer classe
- Permite esconder detalhes de implementação que não interessam aos clientes da classe !

Exemplo: Classe Ponto

```
class Ponto {  
    ...  
  
    public:  
        void SetX(double x) {  
            this->x = x; }  
  
        void SetY(double y) {  
            this->y = y; }  
  
        void leitura() {  
            cout << "\nAbcissa " ; cin >> x ;  
            cout << "\nOrdenada "; cin >> y ; }  
  
        void escreve() const {  
            cout << "(" << x << "," << y << ")" ; }  
};
```

Funções de entrada e saída de dados, incluídas na biblioteca <iostream>



Exemplo de um Programa C++

```
#include <iostream>           //para Bibliotecas C++ pré-definidas
using namespace std;

#include "Ponto.h"             //para fxs. definidos pelo utilizador

int main()
{
    Ponto p1;
    p1.escreve() ;
    Ponto p2(5,7) ;
    p2.escreve() ;
    Ponto p3(p2) ;
    p3.escreve() ;

    return 0 ;
}
```

Objectos e membros constantes (const)

- Aplicação do “princípio do privilégio mínimo” (Eng. de Software) aos objectos
- **Membro-função constante:**
 - declarado com sufixo **const** (a seguir ao fecho de parêntesis)
 - especifica que a função não modifica o objecto a que se refere a chamada
- **Objecto constante:**
 - declarado com prefixo **const**
 - especifica que o objecto não pode ser modificado
 - como não pode ser modificado, tem de ser inicializado
Exemplo: **const Data nascBeethoven (16, 12, 1770);**
 - não se pode chamar membro-função não constante sobre objecto constante
- **Membro-dado constante:**
 - declarado com prefixo **const**
 - especifica que não pode ser modificado (tem de ser inicializado)

Inicializadores de membros

Quando um membro-dado é constante, um **inicializador de membro** (também utilizável com dados não constantes) tem de ser fornecido a todos os construtores da classe para dar os valores iniciais do objecto

```
class Pessoa
{
    private:
        ...
        Data dtnasc;
        const long BI;                // membro de dado constante

    public:
        Pessoa();
        long getIdade() const;        // membro função constante
};

Pessoa::Pessoa(int i, long bi):BI(bi) ← inicializador de membro const
{
    ...
    idade = i ; }

long Pessoa::getIdade() const
{
    return idade;
}
```


Encapsulamento

Encapsulamento

Permite tratar um objecto como uma caixa preta que possui:

- estrutura de dados privada
- uma API constituída por métodos públicos
- um conjunto de métodos privados

- O Encapsulamento numa classe é feito pelo uso de **palavras reservadas** que estão associadas aos atributos e métodos da classe, designados por **modificadores de visibilidade**

- **Modificadores de visibilidade:**
 - **public** permite acesso a partir de qualquer classe
 - **private** permite acesso apenas na própria classe
 - **protected** permite acesso apenas na própria classe e nas subclasses (associado à **herança !**)

Modificadores de Visibilidade

	public	private
Atributos	Viola Encapsulamento	Reforça Encapsulamento
Métodos	Proporciona Serviços aos clientes	Suporta outros métodos na classe

Composição


Composição

- Mecanismo básico e simples de reutilização que consiste numa classe poder usar na sua definição classes já definidas, ou seja, as variáveis de instância definidas numa classe são associadas a classes já existentes
- A manipulação destas variáveis dentro da classe, torna-se simplificada, pois apenas se usam as mensagens que activam os métodos que são disponibilizados pelas classes já definidas
- A Composição é dita como uma relação **"has-a"**
- Membros-objecto são inicializados antes dos objectos de que fazem parte. Os argumentos para os construtores dos membros-objecto podem ser indicados de duas formas:

Exemplo: Composição

```
class Figura {  
    private:  
        Ponto centro ;  
        string cor ;  
        double raio ;  
  
    public:  
        Figura():centro()  
        { cor="" ; }  
  
        Figura(const Ponto& p, string c):centro(p)  
        { cor = c ; }  
  
        Figura(const Figura& f)  
        { cor = f.getCor() ;  
          centro = Ponto (f.getCentro()); }  
  
        ~Figura()  
        { }  
        ...  
};
```

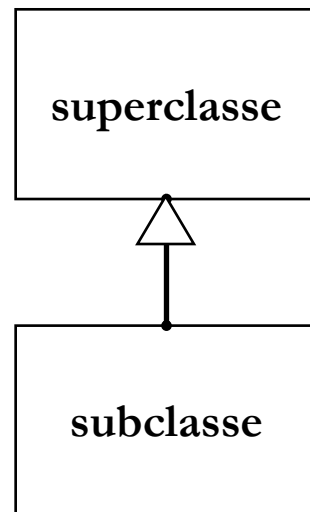
Inicialização de Membros-objeto



Herança

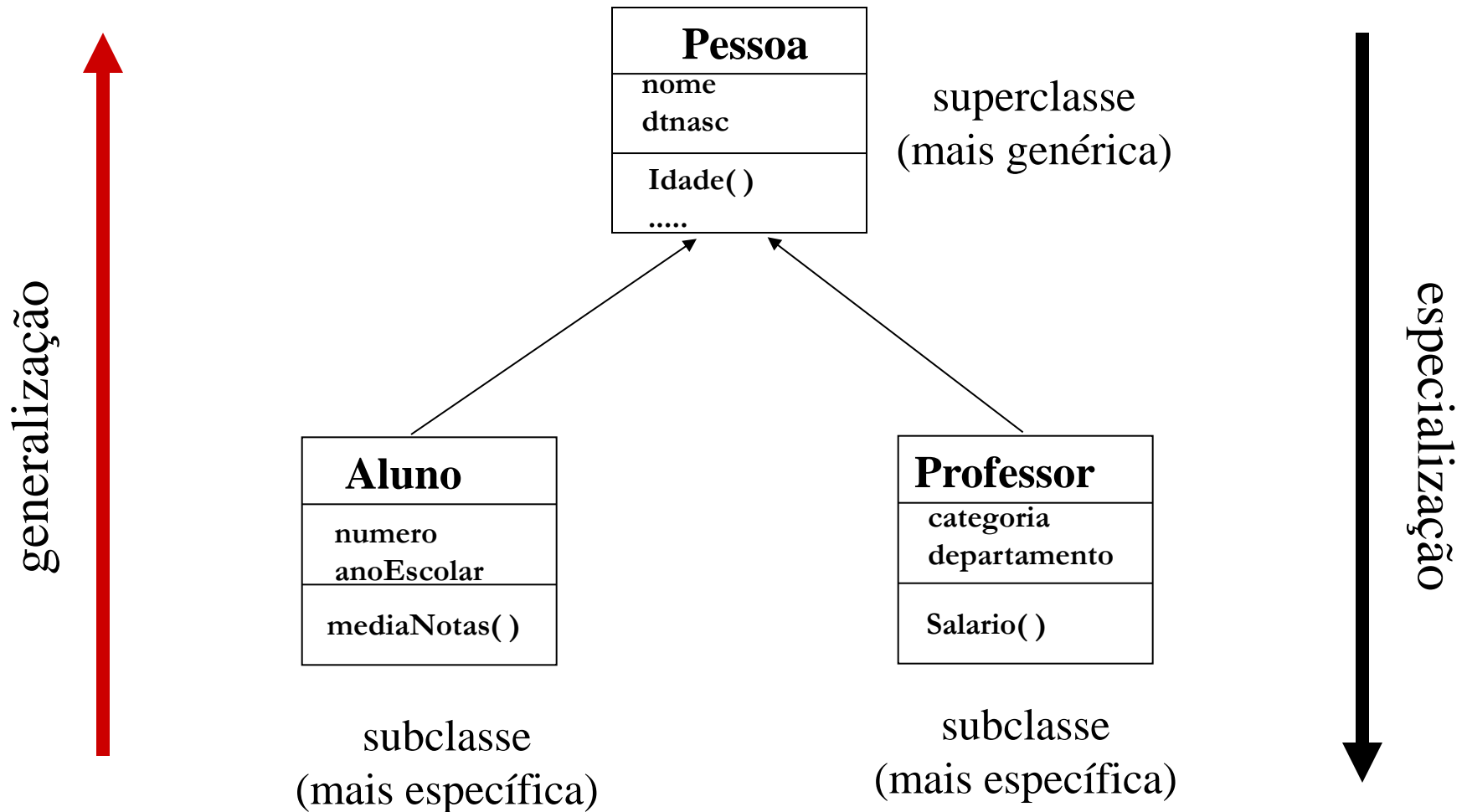
Herança

- Herança permite usar classes já definidas para derivar novas classes. A nova classe herda propriedades (dados e métodos) da classe base
- A **subclasse** constitui uma especialização da **superclasse**. A superclasse pode ser vista como **generalização** das subclasses
- A herança é dita como uma relação “**is-a**”
- **Classe mãe**: superclasse, classe base
- **Classe filha/filho**: subclasse, classe derivada

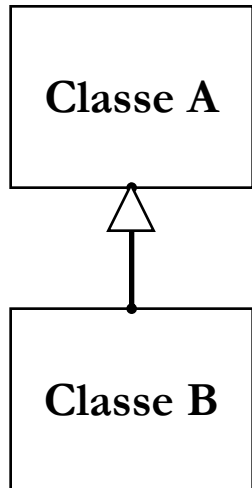


- Classe filha (mais específica) herda atributos e métodos da classe mãe (mais geral)
- Classe filha possui atributos e métodos próprios

Exemplo: Herança



Mecanismo de Herança



- Entre uma classe e a sua **superclasse**, é estabelecida uma relação de **especialização** que é automaticamente implementada através de um mecanismo de herança
- Este mecanismo automático de herança estabelece as seguintes propriedades entre uma **subclasse B** e a sua **superclasse A**:
 - **B** herda de **A** todas as variáveis e métodos de instância
 - **B** pode definir novas variáveis e novos métodos próprios
 - **B** pode redefinir variáveis e métodos **herdados**
 - **B** é uma nova classe, logo **A** (classe base) não é afectada por mudanças nas classes derivadas
 - **B** é compatível por tipo com a base, **mas o contrário é falso**

Formas de Herança

`class A: public B { ... }` `//herança pública`

- ♦ os membros públicos e protegidos da classe-base são herdados **com a mesma qualificação** pela classe derivada - os membros privados são herdados, mas ficam escondidos

`class D: protected B { ... }` `//herança protegida`

- ♦ os membros públicos e protegidos da classe base são herdados **como membros protegidos** pela classe derivada - a classe derivada pode usar os membros públicos (public) e protegidos (protected) da classe base como se fossem declarados na própria classe
- ♦ Restrição de acesso intermédia entre o acesso público e o privado

`class D: private B { ... }` `//herança privada`

- ♦ os membros públicos e protegidos de classe base são herdados como membros privados pela classe derivada

Exemplo Herança

```
class Pessoa {  
    private:  
        string nome ;  
        Data dtnasc ;  
        int alt ;  
  
    public:  
        Pessoa();  
        Pessoa(string n, const Data& d, int al);  
        Pessoa(const Pessoa& p);  
        virtual ~Pessoa();  
  
        virtual Pessoa* clone() const ;  
  
        string getNome() const;  
        int getAlt() const;  
        const Data& getDtnasc() const;  
  
        void setDtnasc(const Data& dt);  
        void setNome(string n);  
        void setAlt(int a);  
  
        int Idade() const ;  
        virtual void listar() const;  
  
};
```

Se uma classe define métodos virtuais, obrigatoriamente deve definir o **destrutor virtual**, mesmo que este seja vazio !

A declaração do destrutor virtual na classe base assegura que são invocados os destrutores das classes derivadas

Exemplo Herança

```
class Aluno: public Pessoa {  
    private:  
        int numero;  
        string curso;  
  
    public:  
        Aluno():Pessoa()  
        { }
```

```
        Aluno(const Pessoa& p, int num, string c):Pessoa(p) {  
            numero=num;  
            curso = c ; }
```

```
        Aluno(const Aluno &a):Pessoa(a) {  
            numero = a.numero ;  
            curso = a.curso ; }
```

```
        ~Aluno()  
        { }
```

```
        Pessoa* clone() const {  
            return new Aluno (*this) ; }
```

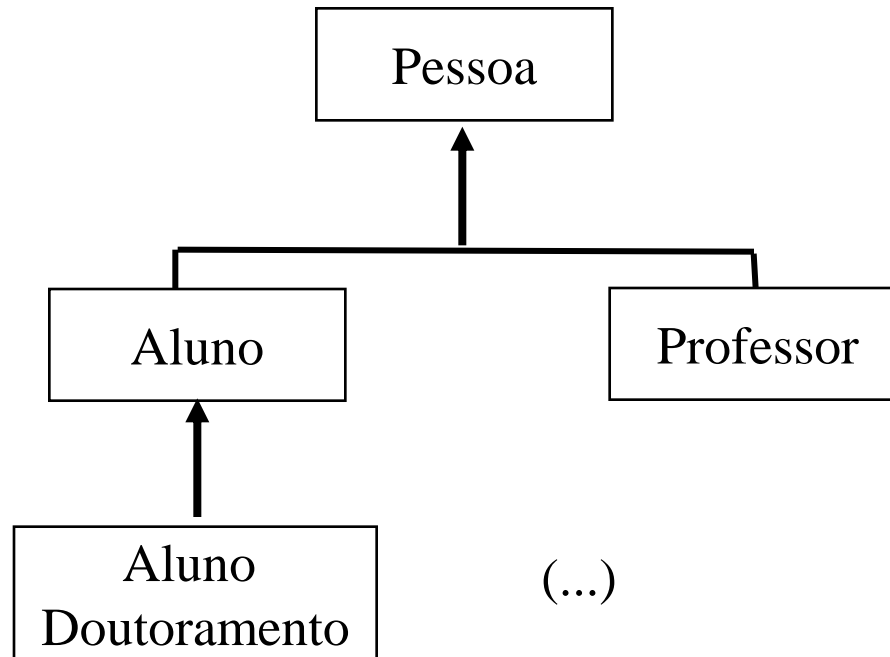
```
        ...  
        };
```

↑
← O construtor da classe derivada deve de invocar **previamente** o construtor da classe base

Classes Derivadas vs. Classe Base

- Os objectos da classe derivada incluem um objecto da classe base
- Existe **conversão implícita** de objectos (ou apontadores/referências) **da classe derivada para** objectos (ou apontadores/referências) **da classe base**, mas não existe conversão implícita no sentido inverso
- A conversão de um objecto da classe base para a classe derivada (quer implícita quer explícita) não tem razão de ser
- Na **derivação protegida e privada** não existe qualquer tipo de conversão implícita entre a classe derivada e a classe base
- Raramente se justifica o uso da derivação protegida ou privada, pelo que só consideraremos a **derivação pública**

Hierarquias de classes

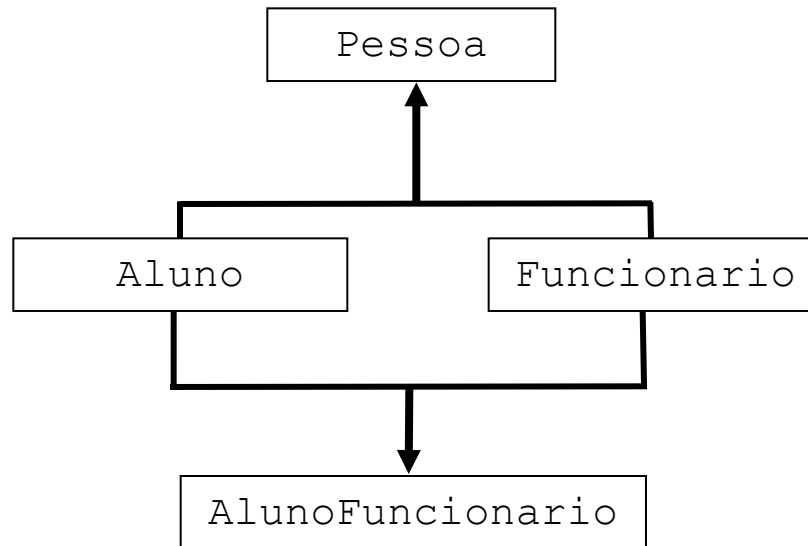


Em geral, pode-se ter uma hierarquia de classes relacionadas por herança

- em cada classe da hierarquia colocam-se as propriedades que são comuns a todas as suas subclasses
⇒ **evita-se redundância, promove-se reutilização de código !**

Herança Múltipla

- **Herança simples:** cada classe derivada tem apenas uma classe base (directa)
- **Herança múltipla:** uma classe derivada pode ter múltiplas classes base directas



Herança Múltipla

Se interessa ter uma única instância da classe base, partilhada pelos vários caminhos, é necessário usar **herança virtual**

- a classe base diz-se **virtual**
- o construtor da classe base é chamado uma única vez implícita ou explicitamente a partir do construtor do objecto completo (da classe "mais" derivada)
- funções virtuais da classe base não podem estar redefinidas ambigualmente

Exemplo:

```
class Pessoa { ... };  
class Aluno : public virtual Pessoa {...};  
class Funcionario : public virtual Pessoa {...};  
class AlunoFuncionario : public Aluno, public Funcionario{...};
```

Vantagens da Herança

- Modificação de uma classe (inserção de novos métodos e variáveis) sem mudanças na classe original
 - Reutilização do código
 - Alteração do comportamento de uma classe
- A partilha de recursos leva a melhores ferramentas e produtos mais lucrativos
 - não é necessário “reinventar a roda” a cada nova aplicação
- É possível modificar uma classe para criar uma nova classe com uma personalidade ligeiramente diferente
 - diversos objectos que executam ações diferentes, mesmo possuindo a mesma origem

Composição versus Herança

```
class Aluno : public Pessoa
{
    private:
        string curso;
        Data dtinscr-curs;
        ...
};
```

- Se conceptualmente há uma relação **IS-A** (um aluno é uma Pessoa) usar: **Herança**
- Se conceptualmente há uma relação **HAS-A** (um aluno possui uma data inscrição curso...) usar: **Composição**

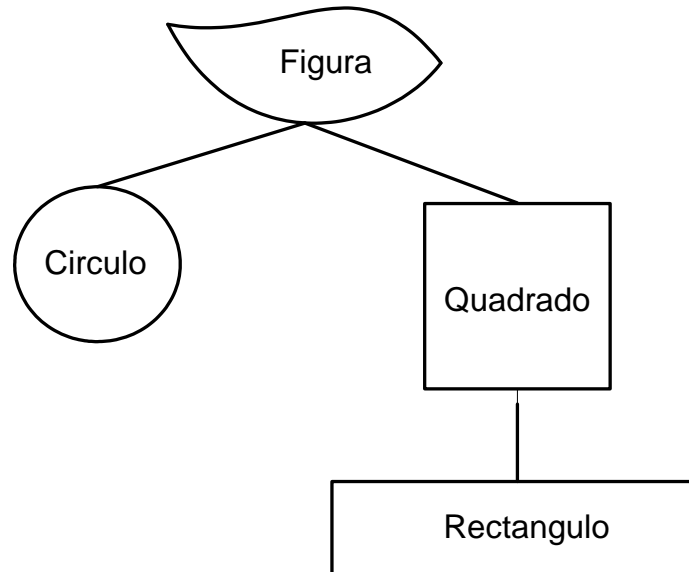
Classes Abstratas

Definição

- Um método compreende:
 - especificação (a sua assinatura)
 - implementação (o seu corpo)
- Há situações em que uma classe deve ter um método com determinada especificação mas nada se pode definir sobre o seu comportamento → **método abstrato**
- Classe que tenha pelo menos um método ***abstrato*** é uma **classe abstrata**
- Uma classe abstrata ***não pode criar instâncias***
- O mecanismo de herança mantém-se aplicável a classes abstratas
- A definição dos métodos abstratos fica a cargo das classes derivadas que herdam os métodos abstratos, usando o mecanismo de **redefinição de métodos**

Classe Figura...

- A classe **Figura** tem atributos que são comuns a todas as classes (ex. centro do objecto)
- Uma variável **Figura** pode referenciar qualquer objecto concreto derivado, tal como **Circulo** ou **Rectângulo**



```
Figura a, b;
a = new Circulo();    // correcto upcast
b = new Figura();     // errado
```

→ Um objecto de uma classe abstracta nunca pode ser criado

Classe Figura

```
class Figura                                // classe abstrata (com funções abstratas puras)
{
    protected:                             // só acessível a classes derivadas
        Ponto centro;

    public:
        Figura():centro() {}
        Figura(const Ponto& c):centro(c) {}
        virtual ~Figura() {}                // destrutor virtual

        virtual void desenhar();
        virtual double area() const =0 ;    // funções abstratas puras
        virtual double perimetro() const =0 ;

};
```

Herança com Classes Abstratas

O mecanismo de herança mantém-se aplicável a classes abstratas

Implicações:

- **qualquer subclasse de uma classe abstrata herda automaticamente todos os métodos da classe abstrata**, estejam implementados ou sejam abstratos
- qualquer subclasse de uma classe abstrata **terá de implementar todos os métodos abstratos herdados** da sua superclasse (sem exceção) para que possa ser uma classe de implementação, concreta, i.e., para que possa ter instâncias
- uma classe abstrata delega nas suas subclasses a responsabilidade pela implementação dos seus métodos abstratos, facilitando o aparecimento de diferentes implementações dos mesmos métodos nas suas subclasses

Classe 100% Abstrata

- **Aplicação:** Desenvolvimento de aplicações genéricas e extensíveis
- o mecanismo de herança garante que todas as suas subclasses vão herdar **o mesmo protocolo ou API**, podendo no entanto definir as suas extensões
- todas as subclasses da classe abstrata responderão ao mesmo protocolo, ou seja, *“falam a mesma linguagem”*
 - *normalização de vocabulário para as subclasses existentes e para as futuras subclasses*
- **classe 100% abstrata** pode ser vista como uma **assinatura** (especificação meramente sintática)

Polimorfismo

Definição

Polimorfismo, do grego: muitas formas

Uma acção diz-se **polimorfa** se for executada de diferentes formas dependendo do contexto em que for invocada

Polimorfismo é o princípio pelo qual **duas ou mais classes derivadas** de uma mesma superclasse podem invocar **métodos que têm a mesma identificação (assinatura)** mas **comportamentos distintos**, especializados para cada classe derivada, usando para tanto uma referência ou apontador a um objecto do tipo da superclasse

No polimorfismo, é necessário que os métodos tenham exactamente a mesma identificação, sendo utilizado o mecanismo de **redefinição de métodos**

A decisão sobre qual o método que deve ser seleccionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de **ligação dinâmica**

O mecanismo de **redefinição**, juntamente com o conceito de **ligação dinâmica**, são a chave do **Polimorfismo**

Redefinição versus Sobrecarga de Métodos

Um método é uma redefinição de um método herdado, quando está definido numa classe construída através de herança e possui o mesmo nome, valor de retorno e argumentos de um método herdado da superclasse

Temos **Redefinição** quando a assinatura do método é igual, ou seja, quando uma classe filha fornece apenas uma nova implementação para o método herdado, e não um novo método

Se a classe filha fornecer uma assinatura semelhante com a do método herdado (difere ou no número ou no tipo dos argumentos, ou então no tipo do valor de retorno) não temos redefinição de um método, mas sim **Sobrecarga**, pois criou-se um novo método

Ligação Estática / Ligação Dinâmica

- Na **Ligação Estática** (*static binding*) a decisão de que função invocar para resolver uma sobreposição é tomada em **tempo de compilação**
- Na **Ligação Dinâmica** (*dynamic binding* ou *run-time binding*) a decisão é tomada em **tempo de execução**
- C++ usa *static binding* por **default**, porque se considerava no passado que o *overhead* seria significativo
- Para forçar o *dynamic binding* os métodos devem ser especificados como **virtual**

Métodos virtuais e Polimorfismo

O contexto de uma acção diz-se **polimorfa** se pode ser determinado em tempo de compilação ou em tempo de execução

A linguagem C++ utiliza vários tipos de polimorfismo:

- overload de funções e de operadores
 - template de classes e funções
 - métodos definidos como virtuais numa arborescência de derivação
- } resolvidos em tempo de compilação ligação estática

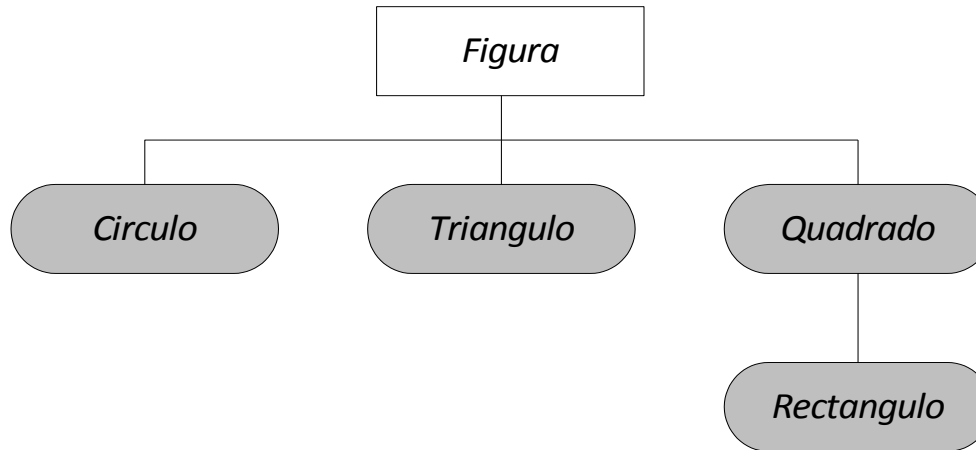
→ Ligação dinâmica

O polimorfismo resolvido em tempo de execução é suportado pelo mecanismo de **herança pública** em conjugação com **métodos virtuais** (com o mesmo nome, número e tipo de parâmetros)

Quando é invocado um método virtual através de um **apontador** ou uma **referência para a classe base**, a versão do método posto em execução é o da classe do objecto apontado ou referenciado e não o da classe correspondente ao tipo da declaração do apontador ou da referência

Exemplo: Polimorfismo

Seja a seguinte relação hierárquica entre classes:



Como poderemos manusear de forma transparente várias figuras geométricas?

Circulo,
Rectangulo,
Triangulo

- **Vector de apontadores para a classe base**

Classe Grafica

```
#include <iostream>
#include <typeinfo> // necessário para typeid
using namespace std ;
```

```
void main()
{
```

```
    Circulo c1(Ponto(10,75),"branco",3);
    Circulo c2(Ponto(5,88),"amarelo",5);
```

```
    Rectangulo* r = new Rectangulo(Ponto(1,1),"verde",5,7) ;
```

```
    Quadrado q(Ponto(1,1),"azul",4) ;
```

```
    Figura* figs [] = {&c1, &c2, r, &q} ;
```



vector de apontadores para a classe base, e não vector de objectos

```
    for (int i=0 ; i < 4; i++)
```

```
        cout << "Obj. " << typeid(*figs[i]).name() << ", Area: " << figs[i]->area()<<endl ;
```

```
}
```

```
Obj. class Circulo, Area: 28,27
Obj. class Circulo, Area: 78,53
Obj. class Rectangulo, Area: 35
Obj. class Quadrado, Area: 16
```


Identificação Dinâmica de Tipos

O método `area()`, definido com **o mesmo nome e argumentos** nas classes derivadas, **substitui (overrides)** o método `area()` da classe `Figura` (declarado com o prefixo **virtual**) para objectos da subclasse, do seguinte modo:

- Quando se chama `figs[i]→area()`, em que `figs[i]` é do tipo `Figura*`, o sistema chama a versão de `area()` correspondente ao tipo do objecto apontado (**determinado em tempo de execução**):

```
if (typeid(*figs[i]) == typeid(Circulo))           //é do tipo Circulo
    dynamic_cast<Circulo *>(figs[i]→Circulo::area());
else                                              // outros subtipos
    ... ..
```

Chama-se a isto **polimorfismo**: `area()` comporta-se de forma diferente consoante a (sub)classe do objecto a que é aplicada