UNIVERSIDADE da MADEIRA

CTeSP CURSOS TÉCNICOS SUPERIORES PROFISSIONAIS

TECNOLOGIAS DE PROGRAMAÇÃO DE SISTEMAS DE INFORMAÇÃO

Programação orientada a objectos

PROGRAMAÇÃO ORIENTADA A OBJECTOS| Prof. Doutora Frederica Gonçalves

Cofinanciado por:

MADEIRA 14-20

REGIÃO AUTÓNOMA DA MADEIRA

PORTUGAL 2020

UNIÃO EUROPEIA

Fundo Social Europeu

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Herança Múltipla:**

  - Uma classe pode ser derivada a partir de duas ou mais classes-base. A este processo chama-se **herança múltipla.**

```
class b1
{ … } ;
class b2
{ … } ;
class d: [public] b1, [public] b2
{ … } ;
```

- **Herança Múltipla:**

  - A **classe derivada herda** todos os membros das classes-base mencionadas no seu cabeçalho e tem **acesso directo** a todos eles, **excepto** em relação aos membros privados.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
…
class produto {
protected: float preco ;
public:
void setpreco (float p) {preco = p;}
float getpreco () {return preco ;}
};

class cliente {
protected: char *nome ;
public:
void setnome (char *n) {nome = n;}
char *getnome() {return nome ;}
};
class venda : public cliente, public produto {
public:
int quantidade;
float valorvenda()
{return quantidade * getpreco (); }
};
```

```
main ()
{
venda v1 ;
v1.setnome("Sousa e Silva");
v1.setpreco(1.5); v1.quantidade = 20 ;
cout << v1.getnome() << '\t';
cout << v1.valorvenda() << '\n';
}
```

Output:
Sousa e Silva 30

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Derivar classes em cadeia e hierarquia de classes**
  - O conceito e a técnica de **derivação de classes com herança** estão concebidos de modo a podermos criar **hierarquias de classes**, em que umas são derivadas de outras em vários níveis.
  - A **derivação de classes em cadeia**, ou seja, umas a partir de outras, sucessivamente, não levanta qualquer problema.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Conceito (*continuação*)**

  - Consideremos uma classe-base A e uma classe derivada B.
  - Por sua vez, a classe B pode servir de base a uma classe derivada C.

```
class A
{ … } ;
class B :[public] A
{ … } ;
class C: [public] B
{ … } ;
```

  - Ao fazermos com que a classe **C** seja derivada e herde da classe **B**, a classe **C** herda todos os membros da classe **B** e da classe **A**.

- CONSTRUTORES DE CLASSES DERIVADAS E LISTAS DE ARGUMENTOS

  - Quando **é criado um objecto** de uma **classe derivada**, em **primeiro** lugar é chamado o construtor da classe-base e só depois o construtor da classe derivada.

  - Quando **um objecto** de uma classe derivada é **destruído**, em **primeiro lugar** é chamado o destrutor da classe derivada e só depois o destrutor da classe-base.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
...
class base {
float comp, larg ;
public:
base(float c, float l) {
  cout << "Construtor cl. base.\n" ;
  comp = c ; larg = l;  }
~base() {cout << "Destrutor cl. base.\n" ;}
float getarea() {return comp*larg;}
};
class cubo : public base {
float altura;
public:
cubo (float a, float c, float l):base (c, l)
{cout << "Construtor cl. derivada.\n" ;
  altura = a;}
~cubo() {cout << "Destrutor cl. derivada.\n" ;}
float volume ()
  {return getarea() * altura ;}
} ;
```

```
main ()
{
   cubo t1 (3,4,5);
   cout << "Volume do cubo = "
   cout << t1.volume() << '\n';
}
```

Output:
Construtor c1. base.
Construtor c1. derivada.
Volume do cubo = 60

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
…
class base {
float comp, larg ;
public:
base(float c, float l) {
  cout << "Construtor cl. base.\n" ;
  comp = c ; larg = l;  }
~base() {cout << "Destrutor cl. base.\n" ;}
float getarea() {return comp*larg;}
};
class cubo : public base {
float altura;
public:
cubo (float a, float c, float l):base (c, l)
{cout << "Construtor cl. derivada.\n" ;
  altura = a;}
~cubo() {cout << "Destrutor cl. derivada.\n" ;}
float volume ()
  {return getarea() * altura ;}
} ;
```

```
void funcaocubo();// protótipo

main () {
    funcaocubo();
    system("pause");
}

void funcaocubo(){
    tanque t1 (3,4,5);
    cout << "Volume do cubo = " <<
t1.volume() << '\n';
    }
```

Output:
Construtor c1. base.
Construtor c1. derivada.
Volume do cubo = 60
Destrutor c1. derivada.
Destrutor c1. base.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa**
  - O **construtor** da nossa classe-base é o seguinte:

```
base(float c, float l)
{
  cout << "Construtor cl. base.\n" ;
  comp = c ; larg = l;
}
```

  - Imprime uma mensagem que indica o momento da sua execução.
  - Utiliza dois parâmetros **c** (comprimento) e **l** (largura da base do rectângulo.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa (*continuação*)**

```
cubo (float a, float c, float l):base (c, l)
{
cout << "Construtor cl. derivada.\n" ;
altura = a;
}
```

  - A classe derivada cubo tem três parâmetros, **a** (altura); **c** (comprimento) e **l** (largura).
  - Só usa a instrução **altura=a**; e os parâmetros c e l, fornecem os valores respectivamente (sendo estes membros da classe-base).

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa (*continuação*)**

```
cubo (float a, float c, float l):base (c, l)
{
cout << "Construtor cl. derivada.\n" ;
altura = a;
}
```

  - A parte do cabeçalho **:base (c,l)** é que faz com que os dois parâmetros sejam passados para a classe-base.

  - Isto designa uma **lista de argumentos**.

- **Análise do programa (*continuação*)**

  - Uma lista de argumentos em geral é da seguinte forma:

    construtor (parâmetros) : classe_base (arg.s)

  - Depois dos parâmetros do construtor, escrevem-se **dois pontos (:),** seguidos do **nome da classe-base** e da **lista de argumentos (arg.s)** a passar para o construtor dessa classe.

• REDEFINIR FUNÇÕES DE CLASSE-BASE EM CLASSES DERIVADAS

- Como vimos anteriormente, as **classes derivadas** herdam todos os membros das classes-base.
- No entanto, uma classe derivada pode **redefinir** ou **criar uma função** com o mesmo nome de uma outra existente na classe-base.
- Para além de podermos **reescrever a função-membro** dentro da classe derivada, o mesmo **também** pode ser feito **fora da classe derivada**, tendo para isso que utilizar o operador de resolução de **escopo (::).**

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
...
class base {
public:
void fx()
{cout << "F. fx() na cl. base.\n";}
};

class derivada : public base {
public:
void fx()
{cout << "F. fx() na cl. derivada.\n";}
};

main () {
base b ;
b.fx();     // chama fx() de b da cl. base
derivada d ;
d.fx();    // chama fx() de d da cl. derivada
}
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
...
class pessoa {
 char *nome ; int idade ;
 public:
 void setnome (char *n) {nome = n;}
 void setidade (int i) {idade = i;}
 void mostradados() {
   cout << "Nome: " << nome << '\t';
   cout << "Idade: " << idade << '\n'; }
} ;
class aluno : public pessoa {
 char *curso; int nota;
 public:
 void setcurso (char *c) {curso = c;}
 void setnota (int n) {nota = n;}
 void mostradados();      // prototipo
};
```

```
void aluno::mostradados()
{
  pessoa::mostradados();
  cout << "Curso: " << curso << '\t';
  cout << "Nota: " << nota << '\n';
}

main ()
{
aluno a1;
a1.setnome("Ricardo Silva");
a1.setidade(17);
a1.setcurso("Programador");
a1.setnota (15);
a1.mostradados() ;
}
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

• FUNÇÕES VIRTUAIS E SUA REDIFINIÇÃO EM CLASSES DERIVADAS

- Uma **função virtual** é uma função-membro de uma classe-base que é declarada com a palavra-chave virtual e que pode (ou deve) ser redefinida nas classes derivadas dessa classe-base.
- As **funções virtuais** constituem uma importante forma de **polimorfismo** em C++.
- Permitem que as classes derivadas readaptem aos seus contextos uma função membro da classe-base.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
…
class base {
public:
virtual void fx()
{cout << "F. virtual na cl. base.\n";}
};
class deriv1 : public base {
public:
void fx()
{cout << "F. virtual na cl. deriv1.\n";}
};

class deriv2 : public base {
// fx() nao redefinida nesta classe
};
```

```
main ()
{
base b, *p ; // declara b1 e p para base
p = &b ;     // coloca p a apontar para b1
p->fx();     // chama fx() de b1 através de p
deriv1 d1 ;  // declara d1 como obj. de deriv1
p = &d1 ;    // coloca p a apontar para d1
p->fx();     // chama fx() de d1 através de p
deriv2 d2 ;  // declara d2 como obj. de deriv2
p = &d2 ;    // coloca p a apontar para d2
p->fx();     // chama fx() de d2 através de p
}
```

- FUNÇÕES VIRTUAIS PURAS E CLASSES ABSTRACTAS

  - Uma **função virtual pura** é uma função que é declarada virtual numa classe-base e não recebe aí nenhuma definição, obrigando a que seja definida nas classes derivadas.
  - A sua forma geral é a seguinte:

```
virtual tipo_de_dados f_nome ([parâmetros]) =0;
```

- FUNÇÕES VIRTUAIS PURAS E CLASSES ABSTRACTAS

  - Uma **classe abstracta** é uma classe que contém pelo menos uma função virtual pura.
  - Implica que uma classe abstracta não possa ser usada para criar objectos a partir dela, uma vez que existe, pelo menos, um membro não definido.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
...
class pessoa {
 char *nome ;
 public:
 void setnome(char *n) {nome = n;}
 char *getnome() {return nome ;}
 virtual void mostradados() = 0 ;
};
class aluno : public pessoa {
 int nota;
 public:
 void setnota(int n) {nota = n;}
 void mostradados() {
   cout << "Nome: " << getnome() << '\t';
   cout << "Nota: " << nota << '\n';
 }
};
```

```
class empregado : public pessoa {
 float ordenado;
 public:
 void setordenado (float ord)
 {
ordenado = ord;
}
 void mostradados()
{
   cout << "Nome: " << getnome() << '\t';
   cout << "Ordenado: " << ordenado << '\n';
}
};
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo (*continuação*)**

```
main ()
{
aluno a1;
a1.setnome("Ricardo Silva");
a1.setnota (15);
a1.mostradados();       //da cl. aluno
empregado e1;
e1.setnome("Avelino Lopes");
e1.setordenado (1000);
e1.mostradados();       //da cl. empregado

pessoa *p ;
p = &a1;
p->mostradados();    //da cl. aluno
p = &e1;
p->mostradados();    //da cl. empregado
}
```

- HERANÇA E POLIMORFISMO CONJUGADOS

  - Podemos utilizar **funções virtuais** (**polimorfismo**) conjugadamente com **classes derivadas** (**herança**) e também ponteiros e acesso dinâmico à memória.

  - As funções virtuais permitem concretizar uma forma de **polimorfismo estático** (resolvidos em tempo de compilação do código).

  - Exemplo destas, é a **sobrecarga de funções**. Permitem que um programa declare várias funções com o mesmo nome diferenciando-se entre si os tipos de dados e parâmetros.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- HERANÇA E POLIMORFISMO CONJUGADOS

  - O polimorfismo que se obtém a partir das funções virtuais, podemos chamar de **polimorfismo dinâmico**.

  - Exemplo deste, é a **herança.** No qual ponteiros de uma classe base podem referenciar objectos de classes derivadas, o que permite que uma chamada à função virtual seja resolvida em tempo de execução do código.

UNIVERSIDADE da MADEIRA

CTeSP CURSOS TÉCNICOS SUPERIORES PROFISSIONAIS