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

- **Função *friend* (amiga) de uma classe:**
  - Permite abrir o **acesso** aos membros de uma **classe** que se encontram ***private***;
  - A **função *friend*** não é membro da classe, mas é declarada dentro da mesma;
  - É uma **função definida** como **exterior** à classe, mas **tem acesso** a todos os membros da classe.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Função *friend* (amiga) de uma classe – (estrutura):**

```
class xpto {
        privat:
        …
        …
        public:
        …
        …
        friend tipo_dados fx_a (xpto x);
        …
        …
} ;
```

- **Função *friend* (amiga) de uma classe:**
  - A **classe "xpto",** contém uma declaração de uma **função amiga "fx_a",** declarada como ***friend***;
  - A função ***friend*** **"fx_a"** recebe um argumento do tipo da **classe "xpto x";**
  - A definição da função **"fx_a"** é feita fora da **classe**.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo:**

```
…
class venda {
private:
        float preco, valor ;
        int quantid;
public:
        venda (float p, int q){                          //construtor
                preco = p; quantid = q;
                }
                friend float valor_venda (venda v);//protótipo
} ;
…
```

```
…
float valor_venda (venda v)
{
                v.valor = v.preco * v.quantid;
                return v.valor;
};
main () {
venda v1 (1.5, 10);
cout << “Valor da venda: ” << valor_venda(v1);
system (“pause”);
}
```

```
Valor da venda: 15!
Prima qualquer tecla para continuar . . .
```

- **Função *friend* (amiga) de uma classe:**

  – O **construtor venda(),** permite passar os valores da inicialização da classe;

  – A função **friend valor_venda()** é colocada como **protótipo** na classe, e definida fora da mesma;

  – A definição da **função valor_venda()** retorna o valor da multiplicação entre a quantidade de produtos e o valor do mesmo.

- **Classe *friend* (amiga) de uma outra classe:**

  – Permite o acesso de **todos** os **objectos** de uma **classe** pela **classe *friend***, incluindo os **objectos *private***;

  – Todas as **funções – membro** da **classe *friend*** podem aceder a todos os objectos da **classe principal**.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Classe *friend* (amiga) de uma outra classe - (estrutura):**

```
class xpto {
        privat:
        …
        …
        public:
        …
        …
        friend class kkk;
        …
        …
} ;
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo:**

```
…
class produto {
        char * nome, float preco ;
public:
        produto (float p){preco = p;}      //construtor
        friend class venda;
} ;
class venda {
…
```

```
…
class venda {
                float valor; int quantid;
public:
                venda (int q) {quantid = q; }//construtor
                float valor_venda (produto p){
                                valor = p.preco * quantid;
                                return valor; }
} ;
main () {
…
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo:**

```
…
main () {
        produto p1 (1.5);
        venda v1 (10);
        cout << “Valor da venda: ” << v1.valor_venda(p1);
        system (“pause”);
}
…
```

```
Valor da venda: 15!
Prima qualquer tecla para continuar . . .
```

- **Classe *friend* (amiga) de uma outra classe:**

v1.valor_venda(p1)

– Na **classe venda**, existe uma **função valor_venda** que permite aceder a um **objecto** da **classe produto**;

– Assim, **valor_venda**, **acede ao preço** que foi introduzido na **classe produto.**

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Atribuição dinâmica de memória com classes e objectos:**

  – A **atribuição dinâmica de memória** é realizada da mesma forma que nas **estruturas de dados**;

  – É necessária **um ponteiro** para criar um **bloco de memória dinâmico**.

  – As **palavras – chave** da atribuição dinâmica de memória são ***new*** e ***delete***.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo de atribuição dinâmica de memória com classes:**

```
…
class aluno {
public:
        char nome[40] ;
        int numero ;
} a1;

main() {
        strcpy(a1.nome, “Ana Cruz”);
        a1.numero = 1;
…
```

```
…
        cout << a1.nome << ‘\t’ << a1.numero << ‘\n’;
        aluno *a;
        a = new aluno;
        strcpy(a->nome, “Rui Nunes”);
        a->numero = 2;
        cout << a->nome << ‘\t’ << a->numero << ‘\n’;
        delete a;
        system (“pause”);
}
```

```
Ana Cruz        1
Rui Nunes       2
Prima qualquer tecla para continuar . . .
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Atribuição dinâmica de memória com classes e objectos:**

`aluno *a;`

– **Criação** de um **ponteiro para a classe aluno**;

`a = new aluno;`

– **Aloca** na memória um **bloco** com os **dados** de um **novo objecto aluno**;

operador **(->)**

– **Referencia** os **campos do objecto** apontado, ex:

`a->numero; a->nome`

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo com um *array* de classes:**

```
…
class aluno {
      int numero; float nota;
public:
       void atribui (int num, float n) {numero = num; nota = n;}
       void escreve () {cout << "O aluno numero "<<numero <<"
tem a nota "<<nota<< "!\n";}
} ;
main () {
     aluno *a, *al ;
     int m = 3; a = new aluno[m];       //criacao de 3 alunos
…
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo com um *array* de classes:**

```
...
    a[0].atribui(1, 12.3);
    a[0].escreve() ;  //escreve João
    al = &a[1];
    al->atribui(2, 14.4);
    al->escreve();
    al++;
    al->atribui(3, 9.3);
    al->escreve();
    delete [ ]a;
...
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Atribuição dinâmica de memória com classes e objectos :**

```
a = new aluno[m]
```

– Cria um bloco de memória para um **array** de **"m" objectos** do **tipo aluno.**

```
a[0].atribui(1, 12.3);
```

– Usando **a[0]** para chamar a **função atribui(),** coloca nos campos respectivos os valores **1** como **numero** e **12.3** como **nota**.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Atribuição dinâmica de memória com classes e objectos :**

```
a[0].escreve();
```

– Usando **a[0]** para chamar a **função escreve(),** realiza a instrução mostrando o **numero** e a **nota** atribuida**.**

```
al = &a[1]
```

– O **ponteiro "al"** aponta para o segundo elemento do ***array*** **"a".**

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Atribuição dinâmica de memória com classes e objectos :**

```
al->atribui(2, 14.4);
```

– Através do **ponteiro "al"** é possível manipular os dados da classe alunos, mas com o **operador "->".**

```
delete [ ]a;
```

– Com a **instrução delete [ ]a**, apagamos o *array* a da memória, e qualquer tentativa de aceder aos dados daria erro.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Sobrecarga de operadores:**

  – O **polimorfismo** das classes abrange os **operadores aritméticos, de atribuição, relacionais, etc.** Por exemplo: = (igual); + (soma); <(menor); etc.

  – A este **polimorfismo** chamamos **sobrecarga de operadores;**

  – Além do significado normal do operador, é possível atribuir um significado especial através da **definição de uma função** com o determinado operador.

- **Sobrecarga de operadores – (estrutura):**

```
tipo_dados operator # (parâmetros)
{
<corpo de instruções>
}
```

– O **carácter #** representa um dos **operadores** que desejamos **sobrecarregar**.

– Estas **funções** são definidas como **função de operador** ou **função operadora**.

- **Exemplo :**

```
…
class aluno {
     int numero;
     float nota ;
public:
      aluno () {};                              //construtor
      aluno (int num, float n)                  //construtor
      {cout << "Novo aluno: \n" ; numero = num ; nota = n;}
      void atribuir (int num, float n) {numero = num; nota = n ;}
      int obternum() {return numero;} float obtern() {return nota ;}
} ;
…
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo :**

```
…
main () {
    aluno a1 (1, 10.0) ;        // Utiliza construtor
    cout << "O aluno "<<a1.obternum()<<" tem a nota "<< a1.obtern() << "!\n";
    aluno a2 (1, 20.0) ;        // Utiliza construtor
    cout << "O aluno "<<a2.obternum()<<" tem a nota "<< a2.obtern() << "!\n";
    a1.atribuir(1, 10.5) ;
    cout << "O aluno "<<a1.obternum()<<" tem a nota "<< a1.obtern() << "!\n";
    a2 = a1 ; cout <<"Copia automatica a2=a1;\n";
    cout << "O aluno "<<a2.obternum()<<" tem a nota "<< a2.obtern() << "!\n";
    //cout << "Soma: " << a1 + a2 ;  // Erro!
…
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

```
Novo aluno:
O aluno 1 tem a nota 10!
Novo aluno:
O aluno 1 tem a nota 20!
O aluno 1 tem a nota 10.5!
Copia automatica a2=a1;
O aluno 1 tem a nota 10.5!
Prima qualquer tecla para continuar . . .
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Sobrecarga de operadores (solução do operador “+”):**

```
…
class aluno {
…
        float operator + (aluno a){
        cout << “Funcao operadora + ” << “\n”;
        float resultado = nota;
        resultado = resultado + a.nota;
        return resultado; }
} ;
…
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

```
Novo aluno:
O aluno 1 tem a nota 10!
Novo aluno:
O aluno 1 tem a nota 20!
O aluno 1 tem a nota 10.5!
Copia automatica a2=a1;
O aluno 1 tem a nota 10.5!
Funcao operadora +
Soma: 21
Prima qualquer tecla para continuar . . .
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

```
Novo aluno:
O aluno 1 tem a nota 10!
Novo aluno:
O aluno 1 tem a nota 20!
O aluno 1 tem a nota 10.5!
Copia automatica a2=a1;
O aluno 1 tem a nota 10.5!
Funcao operadora +
Soma: 21
Prima qualquer tecla para continuar . . .
```

- **Relembrando**

  - O **encapsulamento** tem a ver com a forma como a definição de uma classe permite **proteger** o seu código e os seus dados.

  - Vantagens:
    - As classes e os objectos constituem unidades de código protegidas, flexíveis e reutilizáveis.

- **Relembrando**

  - O **polimorfismo** tem a ver com a possibilidade de criarmos diferentes versões para uma função operador.

  - Vantagens:
    - Aumenta a flexibilidade e a possibilidade de readaptação desses elementos de código a novos contextos.

- **Definição**
  - Designa-se por **derivação de classes** a criação de classes a partir de outras classes.
  - Em que a **classe-base**, é a classe que serve de base à criação de outra classe.
  - E **classe derivada**, é a classe que é criada a partir de outra classe.

- **Herança**
  - Uma **classe derivada herda**, em princípio, todos os membros (variáveis ou atributos e funções ou métodos) da sua classe base.
  - A este princípio geral, chama-se **herança**.

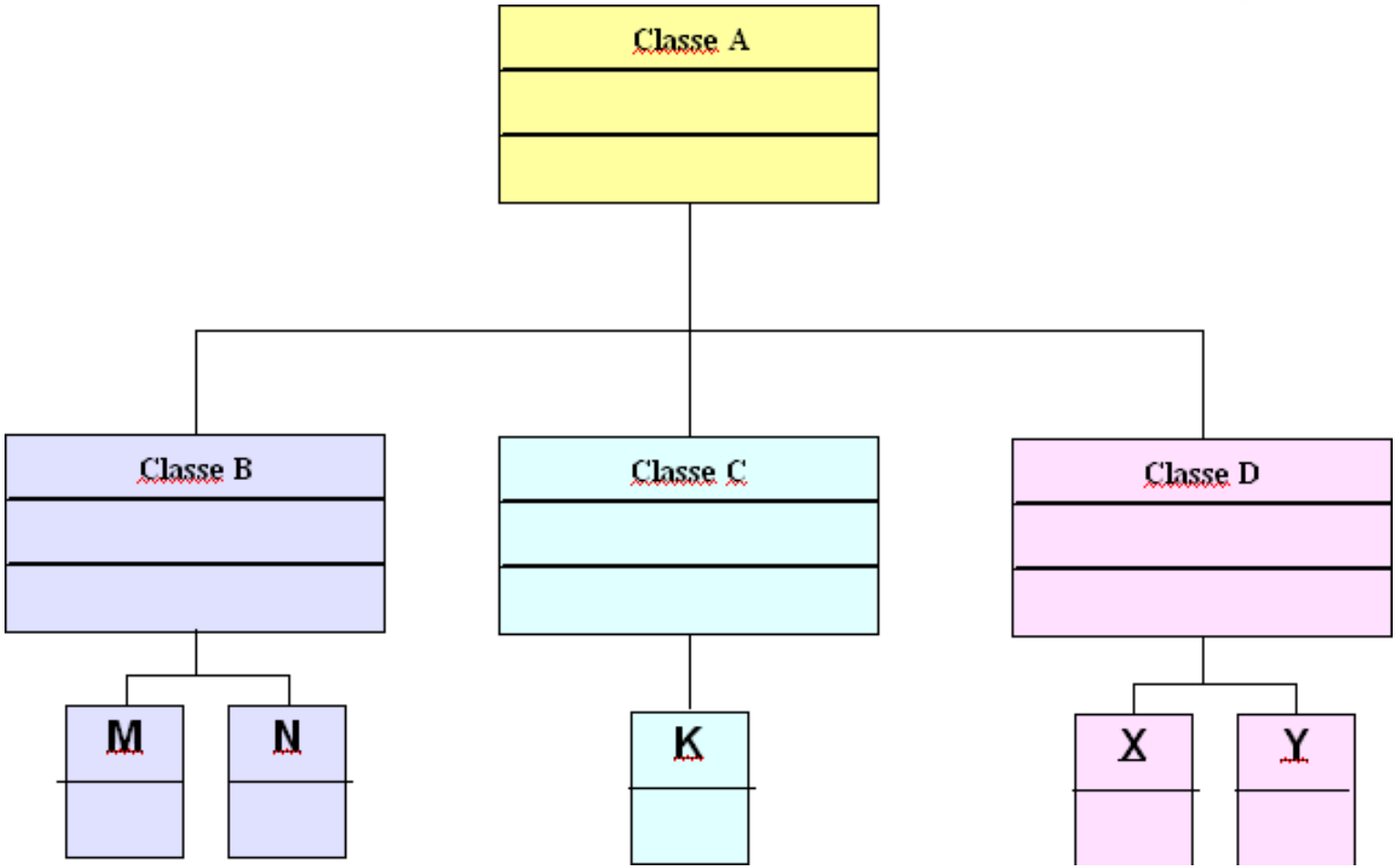
- **Hierarquia de classes**

  - Uma **classe derivada**, pode também servir de **classe-base** para a derivação de outras classes.

  - Podemos assim construir uma estrutura com vários níveis de hierarquia entre classes-base e derivadas.

  - A este princípio chama-se **hierarquia de classes**.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS



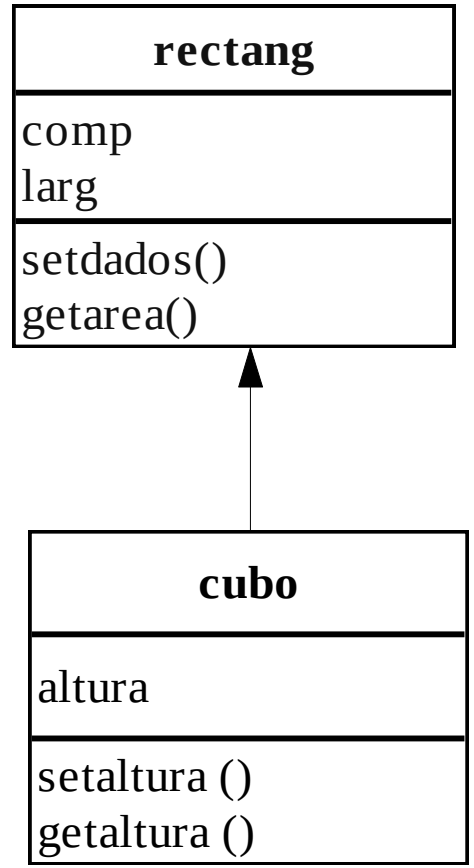
- **Ideia base**
  - **Derivação de classes com herança** permite criar novas classes a partir de outras já existentes, reutilizando-se unidades de código já criadas e readaptando-se a novas finalidades.
  - A **classe derivada**, também é constituída por elementos próprios, tanto a nível de atributos como de métodos.

- **Estrutura**
  - A **declaração de uma classe derivada** faz-se do seguinte modo:

```
class ClasseDerivada:  [especificador ] ClasseBase
{
        // declaração dos membros da classe derivada

};
```

- **Exercício**
  - Crie uma **classe-base rectang**, que representa um rectângulo, tendo dois membros de dados privados: **comp = 4** – comprimento; e **larg = 5** – largura. *Ver figura.*
  - Esta classe vai servir como classe-base para a **classe derivada cubo** que para além dos membros herdados, ainda tem um dado privado – **altura = 3**.
  - **Calcule o volume do cubo.**



UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

```
...
class rectang
{
      float comp, larg ;
      public:
      void setdados (float cp, float lg)
           {comp = cp ; larg = lg;}
      float getarea() {return comp*larg;}
};
class cubo : public rectang
{
      float altura;
      public:
      void setaltura (float a) {altura = a;}
      float getaltura () {return altura;}
} ;
```

```
main ()
{
    cubo t1;
    t1.setaltura (3);
    t1.setdados(4,5);
    cout << "Volume = " ;
    cout << t1.getaltura() * t1.getarea() ;
    cout << '\n';
    system ("Pause");
}
```

Output:
Volume = 60

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa**

  - A **classe rectang** tem duas funções-membro:
  - **setdados ()** – com dois parâmetros (**cp** e **lg**) para definirem (***set***) os valores do comprimento e da largura do rectângulo.

```
void setdados (float cp, float lg)
        {comp = cp ; larg = lg;}
```

  - **getarea ()** – que devolve um valor (**float**) correspondente à área.

```
float getarea () {return comp*larg;}
```

- **Análise do programa**

  - A **classe derivada cubo** herda todos os membros da classe-base rectang.
  - A **classe cubo** é composta por **três** campos de dados: **altura** (da própria classe); **comp** e **larg** (herdados da classe-base)

```
class cubo : public rectang
{
      float altura;
      public:
      void setaltura (float a) {altura = a;}
      float getaltura () {return altura;}
} ;
```

- **Análise do programa**

```
t1.setaltura (3);
```

  - Com esta instrução, o **objecto t1** de cubo chama a sua função setaltura () para fazer passar o valor 3 para o campo altura.

```
t1.setdados(4,5);
```

  - Neste caso, o **objecto t1** chama a função setdados () da classe-base para passar 4 e 5 respectivamente para o comprimento e largura.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa**

```
t1.getaltura() * t1.getarea() ;
```

  - Com esta expressão calcula-se o volume do cubo, multiplicando a altura pela área da base.
  - t1.getaltura (), o objecto t1 chama a função getaltura ().
  - Com t1.getarea () o objecto t1 chama a função getarea () da classe-base **rectang.**

- **Modalidade de acesso às classes:**
  - Quando é criada uma **classe derivada** é necessário **especificar o tipo de herança** que esta vai ter da sua **classe-base**.
  - Os **especificadores** podem ser:

public **private** **protected**

- **Modalidade de acesso às classes:**
  - Considerando os três tipos de membros de uma classe, a **forma geral** de defini-la pode ser assim:

```
class nome_da_classe
{
   [private: ]
         <lista de membros privados da classe>
   protected:
         <lista de membros protegidos>
   public:
         <lista de membros públicos>
}
   [lista de variáveis do tipo da classe];
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Especificadores**

public

- Indica que os membros da classe podem ser acedidos não apenas pelo interior da classe, mas também fora da classe, através de objectos dessa mesma classe.

private

- Indica que os membros da classe apenas podem ser acedidos no interior da sua classe; **fora dessa classe**, o acesso directo **é negado**, mesmo aos objectos da própria classe.

- **Especificadores**

protected

- Este especificador foi criado tendo em vista proporcionar **acesso a certos membros privados de uma classe-base** por parte das classes derivadas.
- Quando um membro é declarado **protected:**
  - É membro **public** para as classes derivadas
  - É membro **private** para as restantes classes.

- **Especificadores**

Diferenças

- Quanto à utilização das modalidades de acesso **public**, **protected** e **private** como especificadores de **herança** na criação de **classes derivadas** existem algumas **diferenças**:
  - Na forma como ficam os membros herdados da classe-base dentro das classes derivadas;
  - Como os objectos das classes derivadas têm acesso aos membros das classes-base.

- **Analisando diferentes casos**

  - **Caso 1:** Quando uma classe derivada **B** é criada com **public** a partir de uma classe-base **A**.

  - **Caso 2:** Quando uma classe derivada **C** é criada com **protected** a partir de uma classe-base **A**.

  - **Caso 3:** Quando uma classe derivada **D** é criada com **private** a partir de uma classe-base **A**.

- **Análise do caso 1**

class B : public classe A {}

- Os membros herdados da classe-base **A** mantêm, na classe derivada **B**, os membros qualificativos de **public**, **private** e **protected**.

- Os objectos da classe derivada **B** só têm acesso aos membros **public** da classe-base.

- **Análise do caso 2**

```
class C : protected classe A {}
```

- Os membros **public** e **protected** herdados da classe-base **A** passam todos a **protected** na classe derivada **C**.

- Os objectos da classe derivada **C** não têm acesso a nenhum membro da classe-base.

- **Análise do caso 3**

class D : private classe A {}

- Os membros **public** e **protected** herdados da classe-base **A** passam todos a **private** na classe derivada **D**.

- Os objectos da classe derivada **D** não têm acesso a nenhum membro da classe-base.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Exemplo**

```
...
class base {
    private:
    int priv ;
    protected:
    int prot ;
    public:
    int pub ;
    void setpriv (int i) {priv = i;}
    int getpriv () {return priv ;} };
class derivada1 : public base {
    public:
    void setprot (int i) {prot = i ;}
    int getprot () {return prot ;}  } ;
class derivada2 : protected base
{
    public:
    void setprot (int i) {prot = i ;}
    int getprot () {return prot ;}
    void setpub (int i) {pub = i ;}
    int getpub () {return pub ;} } ;
```

```
main ()
{
   cout << "classe derivada1 : public base \n";
   derivada1 d1 ;
   d1.setpriv(11); d1.setprot(12);
   d1.pub = 13 ;
   cout << d1.getpriv() << '\n';
   cout << d1.getprot() << '\n';
   cout << d1.pub << '\n';
   cout << "classe derivada2 : protected base \n";
   derivada2 d2 ;
   d2.setprot(22); d2.setpub (23);
   cout << d2.getprot () << '\n';
   cout << d2.getpub () << '\n';
}
```

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa**
  - Temos uma classe-base com três campos de dados: **private, protected** e **public**.
  - Possui ainda duas funções públicas para acesso ao dado privado.

```
class base {
      private:
      int priv ;
      protected:
      int prot ;
      public:
      int pub ;
      void setpriv (int i) {priv = i;}
      int getpriv () {return priv ;}
};
```

- **Análise do programa**

```
class derivada1 : public base {
      public:
      void setprot (int i) {prot = i ;}
      int getprot () {return prot ;}  }
```

- A partir da classe-base é criada uma classe derivada, **derivada1** com herança **public**.
- Um objecto d1, desta classe, tem acesso directo aos membros públicos da classe-base.

UNIDADE CURRICULAR : PROGRAMAÇÃO ORIENTADA A OBJECTOS

- **Análise do programa**

```
class derivada2 : protected base {
    public:
    void setprot (int i) {prot = i ;}
    int getprot () {return prot ;}
    void setpub (int i) {pub = i ;}
    int getpub () {return pub ;} } ;
```

  - Também é criada a partir da classe-base uma classe derivada, **derivada2** com herança **protected**.
  - Um objecto d2, desta classe, não tem acesso aos membros públicos da classe-base.

UNIVERSIDADE da MADEIRA

CTeSP CURSOS TÉCNICOS SUPERIORES PROFISSIONAIS