PICC
Programação Orientada por Objectos em C++
Parte 1

Fernando Miguel Carvalho

Secção de Programação

ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Alocação de Memória

ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Alocação de memória

- Existem três tipos de alocação de memória:
  - automática
  - estática
  - Dinâmica
- A alocação **automática** está associada ao uso de variáveis locais.
- Uma variável local ocupa um espaço em memória o qual é reservado pelo sistema durante a execução do programa quando este encontrada a definição da variável
- A alocação **estática** difere da automática uma vez que é feita apenas uma vez e antes da variável ser utilizada

Alocação de memória... dinâmica

- permite ao programador o controlo total sobre o tempo de vida dos objectos
- Através dos operadores **new** e **delete**, o programador consegue criar e destruir (respectivamente) objectos de um determinado tipo
- Ao criar um novo objecto com o operador **new** este objecto não está sujeito às regras de *scope* das variáveis locais - O objecto criado com o operador **new** apenas pode ser destruído através do operador **delete**

Alocação de memória... dinâmica...

- O operador **new** "recebe como parâmetro" o tipo de dados do objecto que se quer criar e devolve um apontador para o novo objecto criado
- Ao operador **delete** é passado o apontador para um objecto criado com o operador **new** (o que é destruído é o apontado pelo apontador e não o próprio apontador!)

```
int *p = new int(20); // p aponta para um inteiro com o
                      //valor 20
*p = 10;          // o inteiro apontado por p passa
                  // a ter o valor 10
delete p;         // o espaço ocupado por *p é libertado
```

- Se o tipo do objecto apontado por p definir um destrutor então o **delete** chama esse destrutor.

```
T * p = new T();
...
delete p;
```

```
Se T definir um método T::~T()
então será chamado.
```

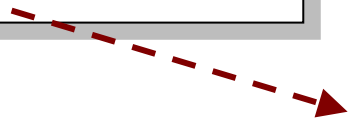
Alocação de memória... dinâmica...

- A utilização anterior do operador **new** cria apenas um objecto do tipo que foi especificado
- Se pretendermos criar um array de objectos podemos também recorrer ao operador **new**

```
int *p = new int[2];
...
cout << *p << *(p+1);
```

- No entanto, será que para libertar a memória alocada se procede da mesma forma que anteriormente? Ou seja:

```
...
delete p;
```



O resultado não é bem aquele que se espera...

Alocação de memória... dinâmica...

```
...
delete p;
```

- O código anterior liberta apenas o espaço ocupado por *p.
- O que se pretende é libertar todo o espaço ocupado pelo array, e não apenas o de uma posição.
- Para que o compilador conheça a nossa intenção, é necessário proceder da seguinte forma:

```
...
delete [] p;
```

Se existir, chama o destrutor de cada um dos elementos do array.

Referências

Referências

```
T v, x;
...
T & ref = v;
x = ref;
ref = x;
```

- A variável ref
  - é uma referência para o tipo T
  - refere v
- Usar (obter/alterar) ref é usar o valor referido.
- As referências têm que ser iniciadas da declaração.

- É um tipo composto tal como o ponteiro
  - T& é um tipo (referência para T)
- Do ponto de vista sintáctico
  - ref é um nome alternativo para v
- Quanto ao código gerado
  - ref guarda o endereço de v

Referências versus Ponteiros

```
void main()
{
  int i=10,j=20;
  int *pi = &i;
  int& ri=i;

  cout << "Com apontador:" << *pi << endl;
  cout << "Com referencia:" << ri << endl;
  ++pi;
  ++ri;
  cout << "Com apontador:" << *pi << endl;
  cout << "Com referencia:" << ri << endl;
  pi=&j;
  ri=j,++ri;
  cout << "Com apontador:" << *pi << endl;
  cout << "Com referencia:" << ri << endl;
}
```

```
Com apontador:10
Com referencia:10
```

```
Com apontador:-858993460
Com referencia:11
```

```
Com apontador:20
Com referencia:21
```

Referências versus Ponteiros

- O nome do ponteiro significa o endereço armazenado
  - Para obter o valor apontado é necessário o operador *.
- O nome da referência significa o valor referido.
- A referência é iniciada com o elemento a referir
  - Uma referência T & tem que ser iniciada com l-value do tipo T.
- A referência não pode ser alterada
  - Guarda sempre o mesmo endereço.

Parâmetros que são referências

- Vulgarmente designado por passagem de parâmetros por referência
  - Na chamada é indicada a expressão de iniciação da referência

```
void inc(int &v) { ++v; }
```

```
int x = 6;
...
inc(x);
```

- Função swap(): Referências versus ponteiros

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

```
int a = 10, b = 20;
```

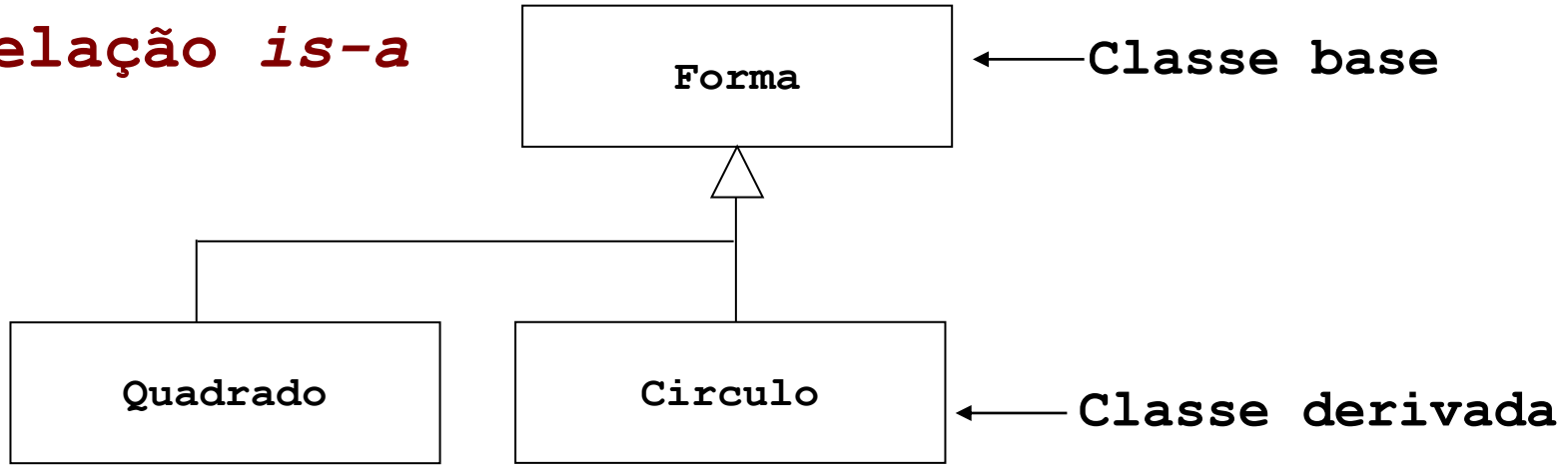
```
swap(a, b);
```

```
swap(&a, &b);
```

Herança

Herança

Relação *is-a*



- A herança é um mecanismo que permite a definir uma relação *is-a* entre classes, possibilitando que a classe derivada acrescente algo à classe base
- Por isso, a classe derivada é mais especifica e, normalmente, mais rica (em **métodos** e **campos**)

Herança II

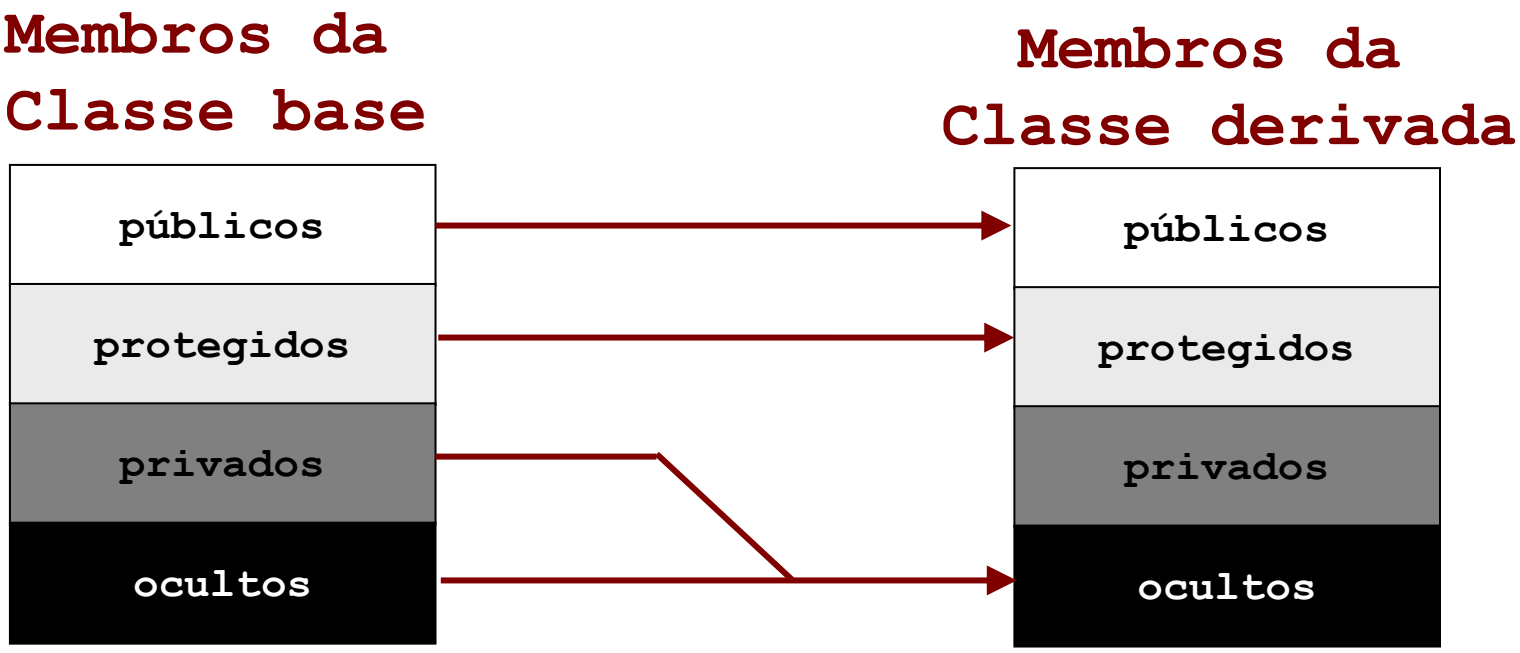
```
class Forma
{
        Ponto p;
        unsigned cor;
   public:
        Forma(Ponto pt,unsigned c){…}
};

class Quadrado: public Forma
{
        unsigned lado;
   public:
        Quadrado(unsigned l,Ponto p,unsigned cor):Forma(p,cor)
        {…}
};
class Circulo: public Forma
{
        double raio;
   public:
        Circulo(double r,Ponto p,unsigned cor):Forma(p,cor)
        {…}
};
```

Lista de derivação

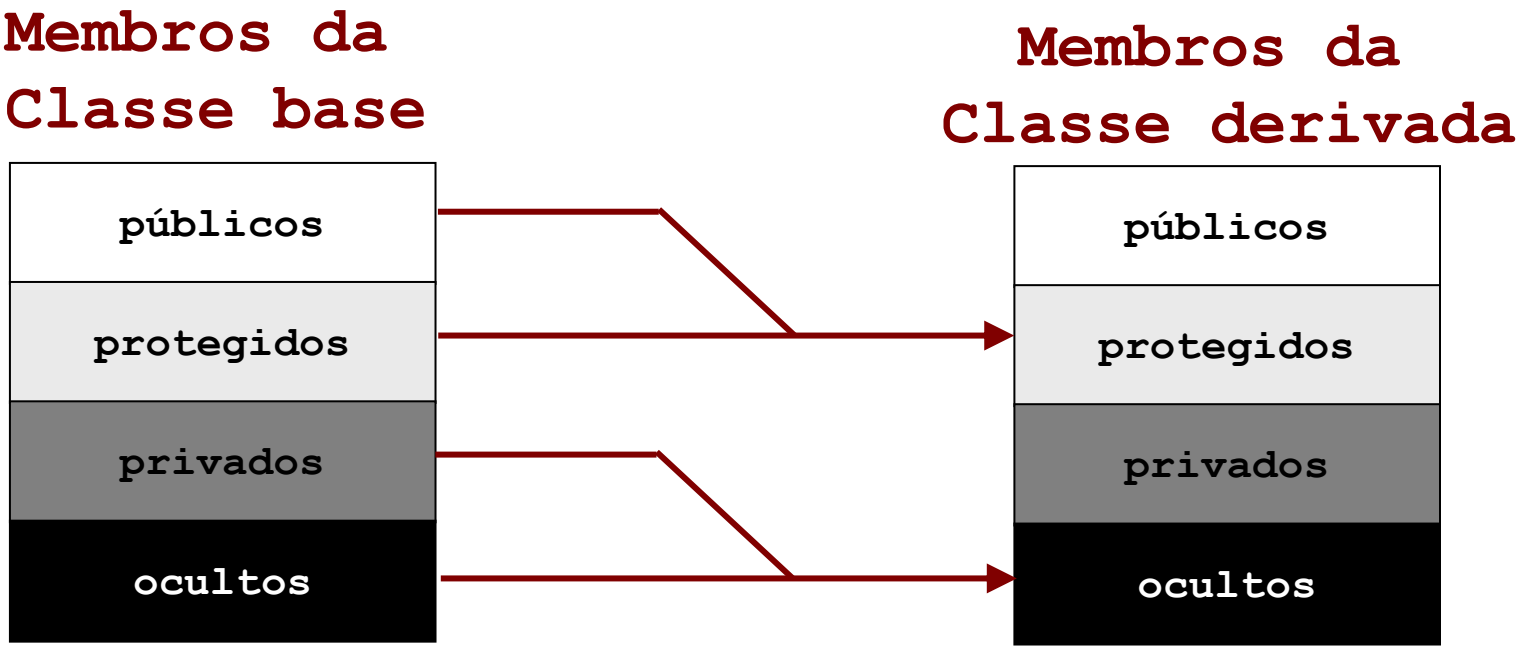
Tipos de herança

- Em c++ é possível indicar o tipo de derivação pretendida
  - Derivação pública `class B: public A`
  - Derivação protegida `class B: protected A`
  - Derivação privada `class B: private A`
- Em cada um destes tipos de derivação existe uma restrição sobre a visibilidade do que é herdado

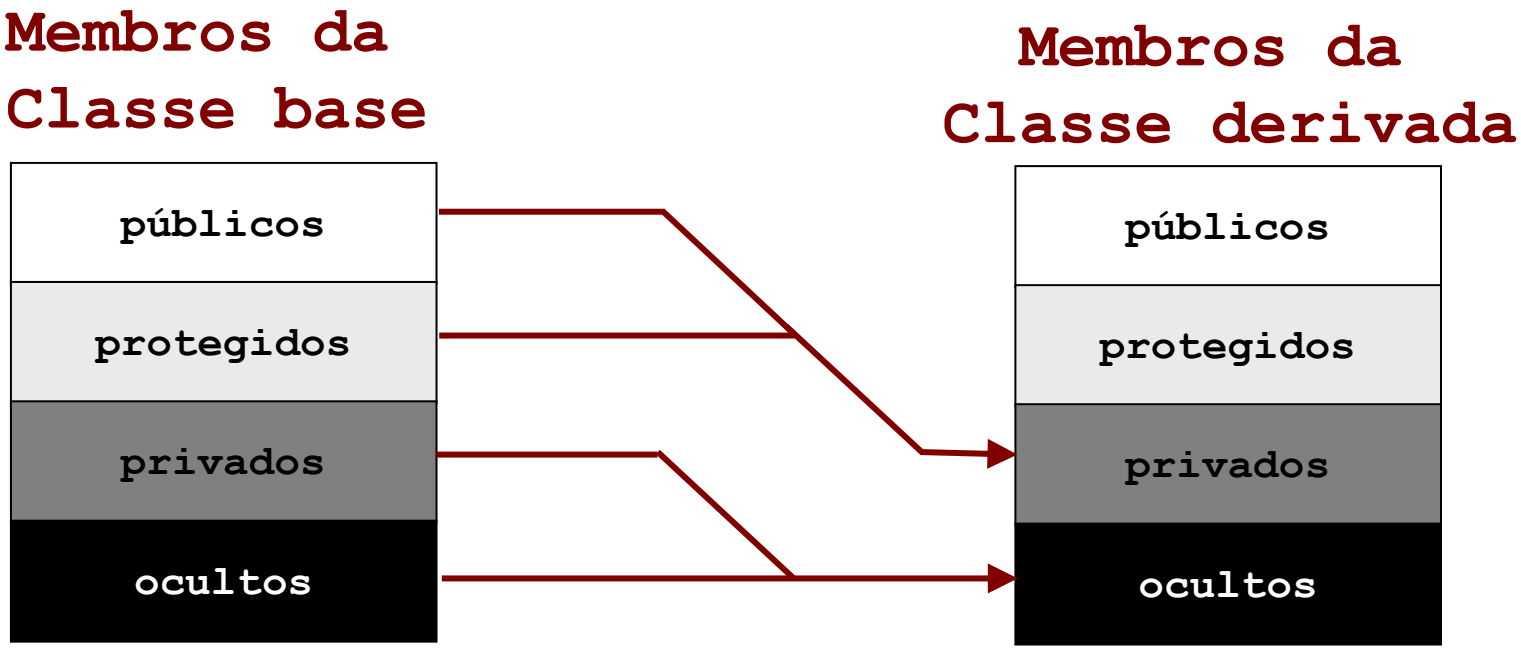
Derivação pública



Derivação protegida



Derivação privada



Construtores

- Quando uma instância da classe derivada é criada, é necessário evocar o construtor da classe base sempre que:
  - Ela não apresente um construtor sem parâmetros
  - Se quer especificar qual o construtor a evocar

```
Quadrado::Quadrado(unsigned l,Ponto p,unsigned cor){…}//erro
```

- A instância de Quadrado não explicita a evocação do construtor de Forma

Construtores II

- A ordem de evocação dos construtores é sempre feita de cima para baixo da árvore de derivação

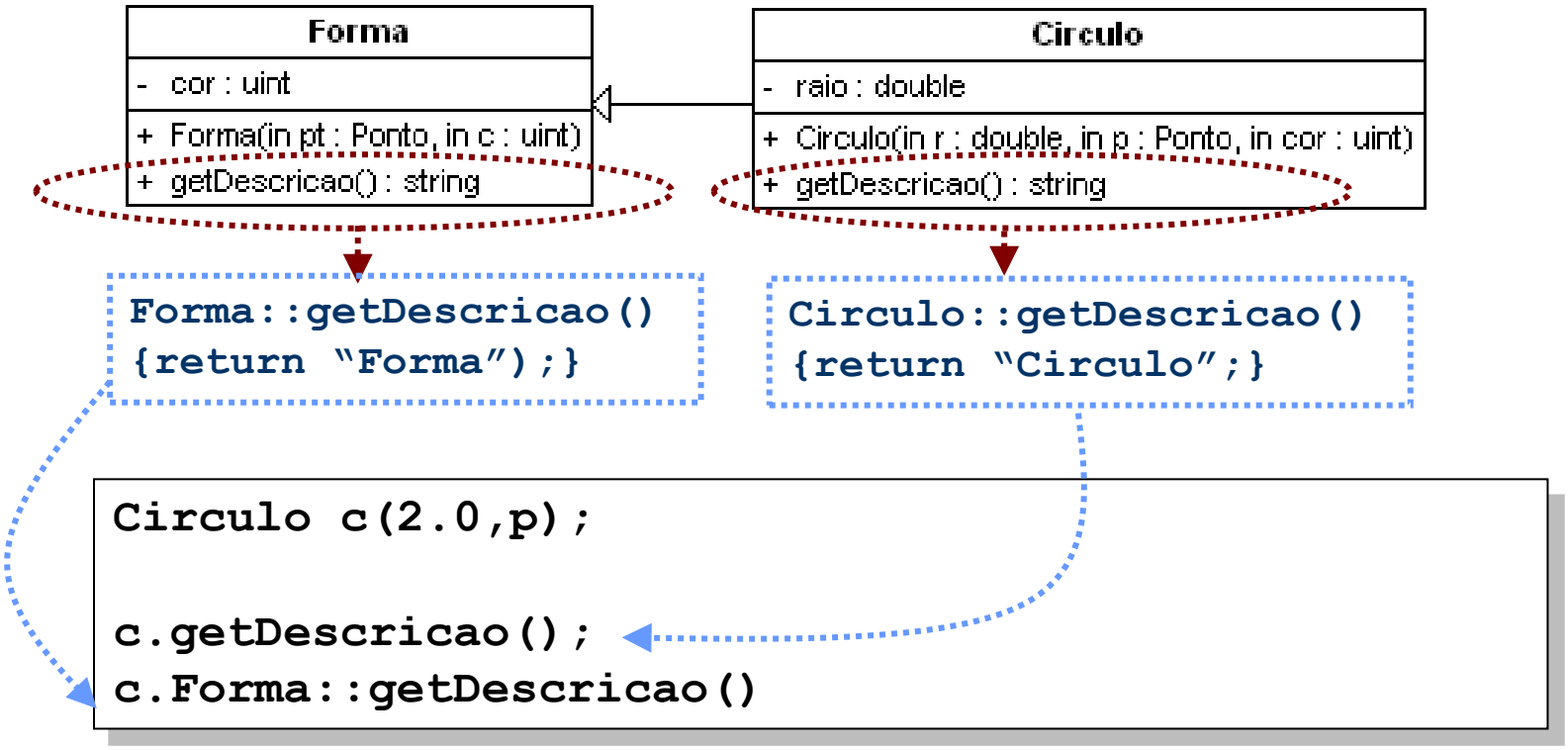
```
Forma::Forma(Ponto pt,unsigned c)
{
  p=pt;cor=c;
  cout<<"Criar Forma..."<<endl;
};

Quadrado::Quadrado(unsigned l,Ponto p,
                   unsigned cor):Forma(p,cor)
{
        lado=l;
        cout<<"Criar Quadrado..."<<endl;
};
```

```
Criar Forma...
Criar Quadrado...
```

Sobrecarga de métodos

- Vamos adicionar um método à classe *Forma*, para que seja possível obter uma descrição do objecto
- Como a descrição vai depender do tipo de objecto , faz sentido que as classes derivadas alterem a descrição.
- No entanto, como a relação modelada é uma *is-a*, as classes derivadas herdam de *Forma* o método *getDescricao*
- A declaração, nas classes derivadas, de um método com a mesma assinatura, faz com que o herdado da classe base necessite de ser explicitamente acedido com o operador de **scope ::**

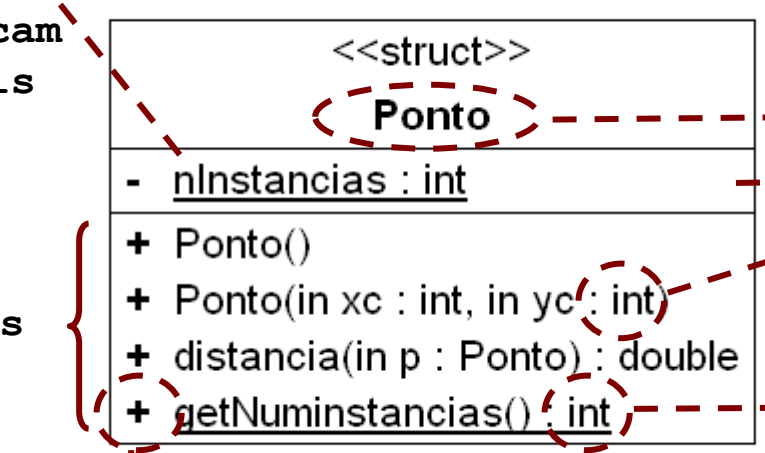
Sobrecarga de métodos II



- Existem dois métodos com a mesma assinatura, mas definidos em diferentes *scopes*

Simbologia UML

Quando sublinhados, os membros são estáticos.
A itálico indicam que são virtuais puros



Nome da classe

Campos

Tipo dos parâmetros

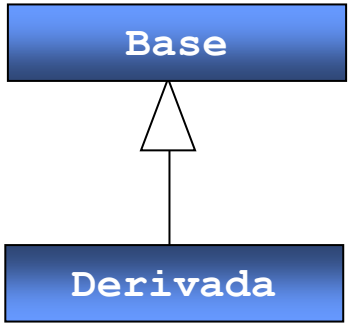
Métodos

Tipo de retorno

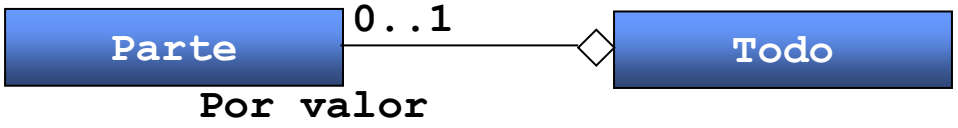
Visibilidade:
- Privada
Protegida
+ Pública

Simbologia UML II

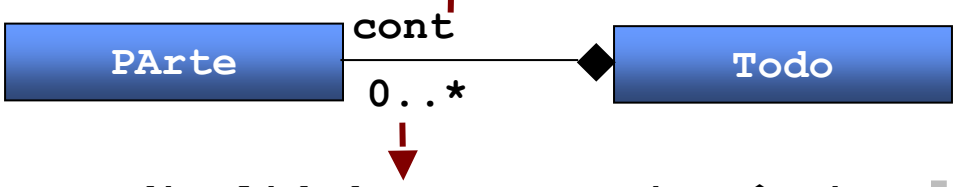
Herança



Agregação/Composição (◊/♦)



Nome do membro de *Todo* que implementa a associação



Dependência

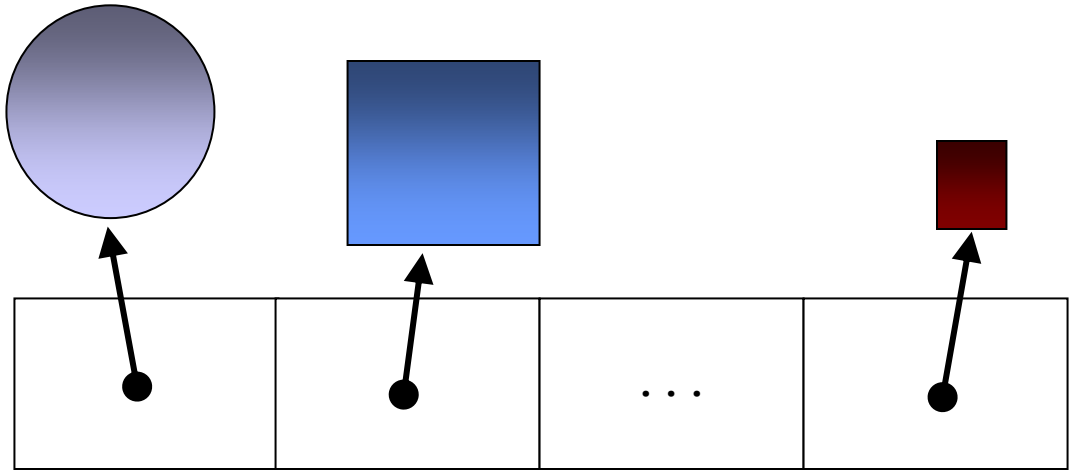
Associação

Cardinalidade. Quantas instâncias de *Parte* são contidas em *Todo*:

- 1 Exactamente uma
- 0..1 Zero ou uma
- 1..* Uma ou várias
- 0..* Zero ou várias

Polimorfismo

- Pretende-se ter um contentor de formas (independente do seu tipo real)



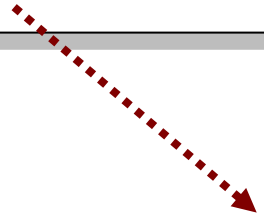
Forma* v[]

```
unsigned capacity = ...;
Forma* v[capacity];
Circulo c(10,Ponto(1,1),1);
Quadrado q(10,Ponto(1,1),1);
V[0] = &c;
V[1] = &q;
...
```

Polimorfismo

- Embora contendo apontadores para Forma, através da herança, um *Circulo* e um *Quadrado* são ambos do tipo *Forma*
- Assim, espera-se que o comportamento de cada um dos objectos armazenados seja coerente com o seu tipo
- No entanto…

```
...
cout<<v[0]->getDescricao();
```



Forma

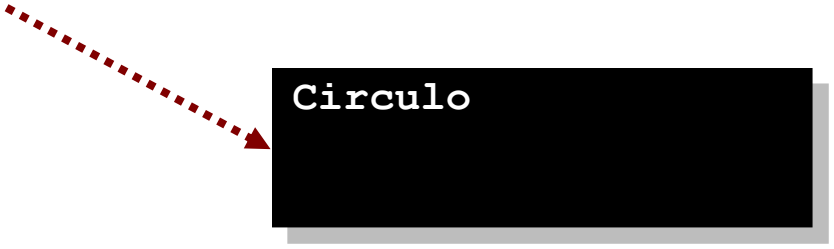
Polimorfismo II

- Para que o comportamento de alguns métodos sejam correspondente ao tipo real do objecto (e não ao tipo do apontador utilizado) é necessário definir na classe base que o comportamento será polimórfico, ou seja
  - virtual string getDescricao();
- O polimorfismo apenas é conseguido através de **apontadores** ou **referência** para objecto

Polimorfismo III: sobrecarregar ≠redifinir

- Sempre que numa classe derivada se define um método com uma assinatura igual a um método virtual da classe base, não se está a **sobrecarregar** o método mas sim a **redefini-lo**.
- Ou seja

```
cout<<v[0]->getDescricao();
```



Métodos virtuais puros ("abstractos")

- Pretende-se que todas as formas respondam ao método `area`
- No entanto, o cálculo da área depende do tipo de forma – terá de ser polimórfico
- Mas, dada a definição de Forma, qual o código que o método terá?

```
class Forma
{
        Ponto p;
        unsigned cor;
   public:
        Forma(Ponto pt,unsigned c){…}
        virtual double area(){????}
};
```

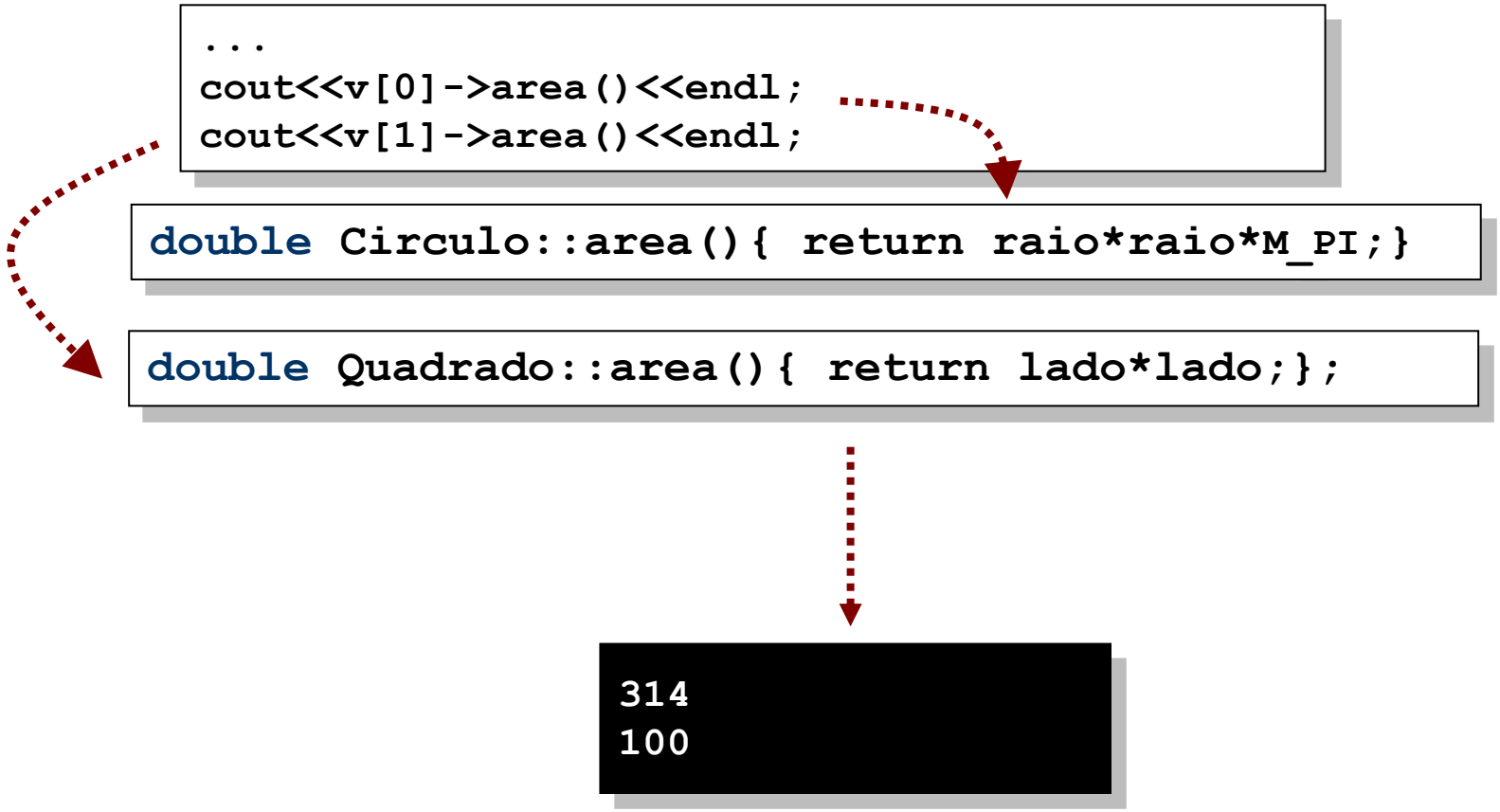
Métodos virtuais puros II

- Não faz sentido existir uma implementação do método na classe Forma, logo:

```
virtual double area()=0;
```

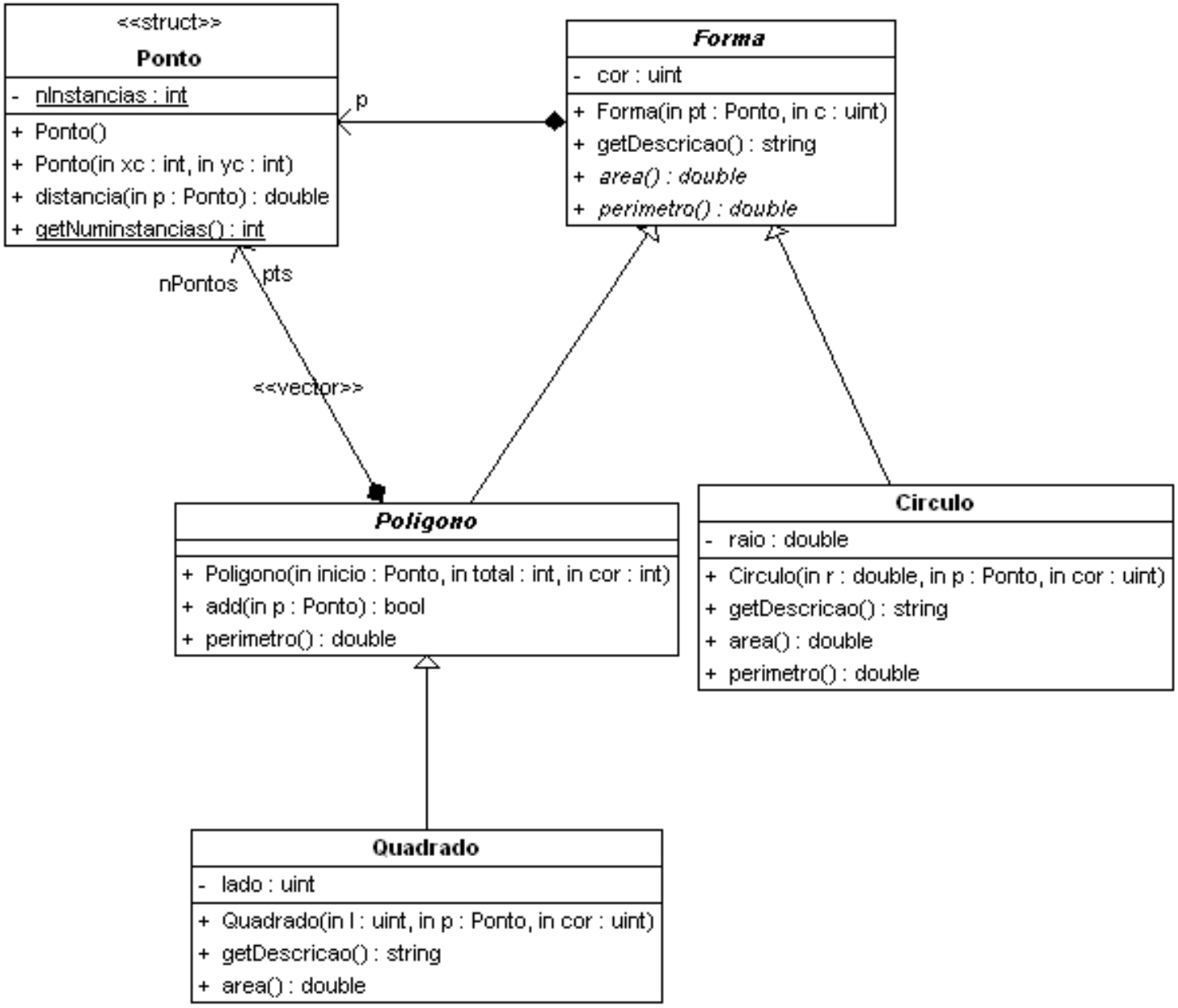
- O método será virtual puro ("abstracto"), fazendo com que a classe Forma passe a ser abstracta, i.e., não pode ter instâncias
- As classes derivadas que não sejam também abstractas têm de fornecer uma implementação para esse método

Métodos virtuais puros III



Polígono

- Pretende-se adicionar o conceito de polígono, do qual o Quadrado é um exemplo
- Um polígono é uma forma, mas é constituído por um conjunto de pontos
- A hierarquia de classes terá de ser alterada para acomodar esta nova classe


<<struct>>
Ponto
- nInstancias : int
+ Ponto()
+ Ponto(in xc : int, in yc : int)
+ distancia(in p : Ponto) : double
+ getNuminstancias() : int
p
Forma
- cor : uint
+ Forma(in pt : Ponto, in c : uint)
+ getDescricao() : string
+ area() : double
+ perimetro() : double
nPontos
pts
<<vector>>
Poligono
+ Poligono(in inicio : Ponto, in total : int, in cor : int)
+ add(in p : Ponto) : bool
+ perimetro() : double
Circulo
- raio : double
+ Circulo(in r : double, in p : Ponto, in cor : uint)
+ getDescricao() : string
+ area() : double
+ perimetro() : double
Quadrado
- lado : uint
+ Quadrado(in l : uint, in p : Ponto, in cor : uint)
+ getDescricao() : string
+ area() : double

Destrutor virtual

- Com esta hierarquia, como devemos proceder quando se pretende libertar o espaço ocupado por uma Forma?
- Que tipo de forma será?
- Terão todas implementações iguais para o destrutor?

Destrutor virtual (cont.)

- ... possivelmente não.
- De facto, levando à letra o ~, o destrutor é o contrário do construtor e deve proceder à limpeza e libertação dos recursos detidos pelo objecto
- Como uma classe base não sabe quais os requisitos para essa limpeza, não pode fornecer uma implementação correcta

Destrutor virtual (cont.)

- Como tal, neste caso, um destrutor deve ser virtual

```
class Forma
{
        Ponto p;
        unsigned cor;
   public:
        ...
        virtual ~ Forma(){}
};
```

- Assim, o polígono pode redefinir o destrutor e proceder à libertação dos recursos por ele alocados

Destrutor virtual (cont.)

- Ou seja

```
class Poligono: public Forma
{
  Ponto * pts;
...
   public:
        ...
        ~Poligono(){delete [] pts;};
};
```