Abstracção e generalização

Overload de operadores

Polimorfismo estático (ou paramétrico)

Funções puramente virtuais

Classes abstractas

Polimorfismo dinâmico (ou de subtipo)

Interfaces

Overload de Operadores

- Capacidade de utilizar a maior parte dos operadores da linguagem sobre tipos do programador (não nativos)
  - Alguns operadores mais conhecidos
    `=, ==, ++, +=, <, ...`
  - Alguns operadores menos conhecidos, mas muito utilizados
    `[], ->, new, new [], ...`

- É muito utilizado com *templates*, mas não só...

  - Gestão de memória
  - Legibilidade do código

- Os **operadores *overloaded* são funções com nomes especiais** e parâmetros com regras bem definidas
  - Por exemplo `tipo_retorno` **`operator+`**`(parametros)`

- Como o significado é muito específico
  - Deve manter-se a funcionalidade reduzida ao espectável
    - ou então, exagerar/abusar da técnica para confundir

Manter a coerência da linguagem

Overload de Operadores

- Há restrições
  - Não podem ser criados novos operadores
  - Mantêm-se as regras de precedência, agrupamento e número de operandos
  - Nem todos os operadores existentes podem ser *overloaded*
    scope resolution (`::`), member access (`.`), member access through pointer to member (`.*`), and ternary conditional (`?:`)
  - Operadores lógicos `&&` e `||` perdem a avaliação "curto-circuito"
    Na expressão `a && b && c`, mesmo que a seja falso b e c são avaliados

- Operadores *overloaded* podem ser
  - Funções-membro
    - Têm prefixo "`T::`" e podem ser `const`
  - Funções globais
    - Têm como primeiro argumento adicional "`T&`", que pode ser `const`
    - Ou são `friend` de `T`, ou necessitam de métodos get/set

Definição de Operadores

- Atribuição (e.g. +=)
  T& T::operator +=(const T2& rhs);

- Incremento/decremento (e.g., ++)
  (prefixo) T& T::operator++();
  (pósfixo) T T::operator++(int);

- Aritméticos (e.g., +)
  (unários) T T::operator~() const;
  (binários) T T::operator+(const T2 &rhs) const;

- Lógicos
  - (unário) bool T::operator!() const;
  - (binários) bool T::operator&&(const T2 &rhs) const;

- Comparação
  bool T::operator==(const T2 &rhs) const;

http://en.cppreference.com/w/cpp/language/operators

Parâmetro *dummy*, só para distinguir

Formas de referência para operadores semelhantes

Overload de Operadores

- Exemplo

```
class Complex
{
    double real;
    double imag;
public:
    Complex(double re, double im)
            : real(re), imag(im)
            {};
    Complex operator+(const Complex& other) {
        double re = real + other.real;
        double im = imag + other.imag;
        return Complex(re, im);
    }
};
```

```
int main()
{
    Complex a(1.2, 1.3);
    Complex b(2.1, 3);
    Complex c = a + b;
    a = b = c;
}
```

Templates

- Exemplos de problema
  - Função para trocar valores de duas variáveis
    - Uma função para int, outra para float, …

```
void Swap (int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```
void Swap (float& a, float& b) {
    float temp = a;
    a = b;
    b = temp;
}
```

  - Classe que implementa uma fila
    - Uma classe para int, outra para strings, …

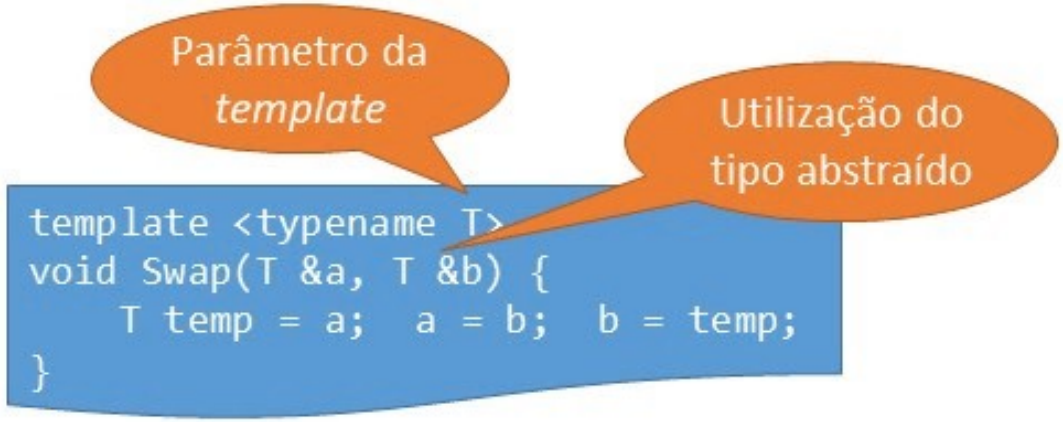
- *Templates* C++
  - Mecanismo que permite abstrair tipos de dados (int, string …), utilizando-os como parâmetros
  - Especificam o comportamento sem definir o tipo de dados que são tratados
  - Existem dois tipos: *templates* de funções e *templates* de classes

Templates de funções

- Modelo de função que opera com tipo(s) de dado(s) genérico(s)
- Sintaxe

```
template <typename T> tipo_r nome_funcao(T par1, ...)
```

- Exemplo
  - *Template* para trocar dois números
    (a função **swap** já existe na STL)



  - Utilização igual às outras funções

Templates de funções

- Modo de funcionamento
  - Durante a compilação, o compilador procura primeiro por uma função e depois por uma *template* que corresponda à invocação
    - Se os argumentos (tipos) da *template* não são fornecidos/explícitos
    - Tenta deduzi-lo(s) automaticamente a partir do(s) tipo(s) da(s) variáveis/objectos utilizados
  - Se não encontrar, dá erro
    - Nome e tipo(s) do(s) argumento(s) não são concordantes
  - **Gera** o código de uma função com o respectivo tipo (substituindo as ocorrências de T) – **instanciação da template**
  - O código gerado é compilado e pode conter erros
    - O exemplo funciona com todos os tipos que possam ser passados por referência e suportem a atribuição ('=')

```
int a = 1; float b = 2.2;
Swap(a, b);
```

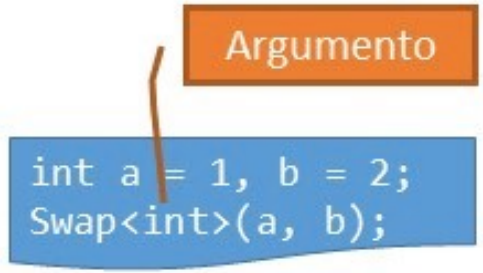
Serão todos?

- Características
  - É um **mecanismo estático**, e portanto não tem impacto no tempo de execução
  - São criadas múltiplas funções a partir do modelo para diferentes tipos (ou formatos/formas)

Polimorfismo estático ou paramétrico

Templates de funções

- Dedução do(s) argumento(s) da *template*
  - Pode ser evitada tornando-o(s) explícito(s)



  - Nem sempre é possível
    - Exemplo

```
int a, b;
long c;
c = Mult(a, b); // erro
```

```
template <typename T1, typename T2>
T2 Mult(T1 a, T1 b) {
    return static_cast<T2>(a * b);
}
```

    - O compilador não consegue deduzir o tipo de retorno
    - Outros casos incluem quando a instanciação não envolve o(s) parâmetro(s) da *template*

  - Resolve-se colocando em primeiro lugar o tipo que não é dedutível e explicitando-o

```
long c;
c = Mult<long>(a, b);
```

```
template <typename T1, typename T2>
T1 Mult(T2 a, T2 b) {
    return (T1) (a * b);
}
```

Templates de classes

- Moldes de classes que operam com tipo(s) de dado(s) genérico(s)
- Sintaxe

```
template <typename T> class nome_classe {...};
```

- Exemplo
  - *Template* que implementa uma stack
    (a *template* **stack** já existe na STL)
  - Utilização

```
Stack<float> fs(3);
float f = 1.1 ;
while (fs.push(f)) f += 1.1;
while (fs.pop(f)) cout << f << endl;
```

```
Stack<string> ss(3);
string s("first");
while (ss.push(s)) s += "a";
while (ss.pop(f)) cout << s << endl;
```

Instanciação explicita

Parâmetro da *template*

```
template <typename T>
class Stack {
size_t size;
    size_t top;
    T* array;
public:
    Stack(size_t = 10);
    ~Stack();
    bool push(const T&);
    bool pop(T&);
    bool isEmpty() const;
    bool isFull() const;
};
```

identifica o tipo abstraído

Necessária, compilador não consegue deduzir

Templates de classes

- Sintaxe das funções membros definidas externamente à classe
  - Algumas funções

```
template <typename T>
Stack<T>::Stack(size_t sz)
{
    size = sz;
    top = -1 ;
    array = new T[size];
}

template <typename T>
Stack<T>::~Stack() {
    delete[] array;
}
```

```
template <typename T>
bool Stack<T>::pop(T& popValue) {
    if (!isEmpty()) {
        popValue = array[top--] ;
        return true;  // success
    }
    return false;  // no success
}

template <typename T>
bool Stack<T>::isEmpty() const {
    return top == -1 ;
}
```

Identificação de função membro

Utilização do parâmetro

Identificação de FM

- Operações realizadas sobre o tipo T
  - Definição de array
  - Acesso a posição de array

Funciona com todos os tipos dos quais se possa criar um array

Templates de classes

- Exemplo completo

```
template <typename T>
class Stack {
    size size;
    size top;
    T* array;

public:
    Stack(size sz = 10) {
        size = sz;
        top = -1 ;
        array = new T[size];
    }
    ~Stack() {
         delete[] array;
    }
    bool isEmpty() const {
        return top == -1 ;
    }
    bool push(const T&) {
        if (!isFull()) {
            array[++top] = item;
            return true;  // success
        }
        return false;  // no success
    }
    bool pop(T& popValue) {
        if (!isEmpty()) {
            popValue = array[top--];
            return true;  // success
        }
        return false;  // no success
    }
    bool isFull() const {
        return top == size - 1;
    }
};
```

Polimorfismo estático ou paramétrico

- As *templates* podem assumir várias formas
  - De acordo com o tipo do(s) seu(s) parâmetro(s)

```
template <typename T>
class Stack {
private:
    int size;
    int top;
    T* array;
public:
    Stack(int = 10);
```

```
template <typename T>
void Swap(T &a, T &b) {
    T temp = a;  a = b;  b = temp;
}
```

Templates

- Processo de desenvolvimento aconselhado
  - Desenvolver a classe/função para um (ou mais) tipo(s) de dados
  - Percorrer a classe e substituir as ocorrências do(s) tipo(s) alvo pelo(s) tipo(s) genérico(s) T(1, T2, …)
    - E aplicar a sintaxe específica das *templates*

Especialização de *templates*

- Consiste em fornecer código especifico para um determinado argumento/tipo de uma *template*
- Quando o código gerado
  - Origina erros (não compila) ou tem que ser diferente
  - Precisa de ser optimizado
- Exemplo
  - Stack para inteiros (desnecessário)
  - Quando o compilador encontra o código abaixo, utiliza a especialização fornecida ao lado e não gera código

```
Stack<int> si;
```

```
template <>
class Stack<int> {
    // ...
    int* array;
    // ...
    bool push(const int&);
    bool pop(int&);
    // ...
```

- No caso de *templates* com mais que um parâmetro a especialização pode ser parcial

```
template<typename T1, typename T2>
class X { ... };
```

```
template<class T1>
class X<T1, int>
{ ... };
```

Especialização para parâmetro T2 = int

Templates e compilação separada

- Considere o seguinte código

A.h
```
template <class T>
class A {
private:
    T t;
public:
    A();
};
```

A.cpp
```
#include "A.h"

template <class T>
A<T>::A()
{
}
```

Main.cpp
```
#include "A.h"

int main ()
{
    A<int> a;
    return 0;
}
```

  - Cada ficheiro .cpp é compilado individualmente
  - Quando o compilador compila a *class template* não sabe os tipos com os quais ela vai ser instanciada (noutros módulos)
    - não gera o respectivo código
    - linker dá erro dizendo que não encontra os métodos correspondentes à instanciação feita

- Soluções
  - Explicitar as instanciações feitas no respectivo .cpp
    - Denomina-se de *Template Instantiation*
  - Incluir o .cpp da template class **no final** do respectivo .h
    - Utilização equivalente à proporcionada pela biblioteca padrão

A.cpp
```
// ...
template class A<int>;
```

A.h
```
// ...
#include "A.cpp"
```

Mais *templates*

- *Templates* de funções como membros de *Templates* de classes
  - Independentes, com o(s) seu(s) próprio(s) parâmetros

```
template<typename T1> class A {
    // ...
    template<typename T2> T2 f(T2 par);
};
```

  - Quando se define/implementa a função, tem que se identificar as duas *templates*

```
template<typename T1>   // class templ.
template<typename T2>   // funct templ.
A<T1>::f(T2 par) {
    // ...
```

Programação genérica

- As *templates*
  - Não são um mecanismo de programação orientado-a-objectos
  - São o principal mecanismo de um outro paradigma da programação

Polimorfismo dinâmico

- Conceito
  - Aplica-se a classes
  - Polimorfismo significa a capacidade de assumir múltiplas formas
  - Dinâmico significa que acontece durante a execução do programa
    - assenta em mecanismos dinâmicos

Uma **classe** é **polimórfica**
quando tem pelo menos **uma função virtual**

subclasses implementam essa função de diferentes formas

Polimorfismo dinâmico

- Problema exemplo
  - Desenho que representa as suas diferentes figuras sem ter que lidar com as especificidades de cada uma

- Desenvolvimento
  - Cada classe trata da sua própria representação


```
class Circle {
  Point center;
  float radius;
public:
  void draw() {
    cout << "circle";
  }
  // ...
```

```
class Square {
  Point center;
  float side;
public:
  void draw() {
    cout << "square";
  }
  // ...
```

Polimorfismo dinâmico

- Desenvolvimento do exemplo
  - Base define como se desenha qualquer figura numa função virtual

```
class Shape {
public:
  virtual void draw() {
    // ??...
  }
  // ...
```

Mais tarde...

  - Utilização

```
Circle c(...);
Square s(...);
Shape *p;
p = &c;
P->draw();
P = &s;
P->draw();
```

Desenha círculo

Desenha quadrado

Shape é polimórfica

Polimorfismo dinâmico

- Desenvolvimento do exemplo
  - A classe Drawing
    - Contém as figuras
    - Tem uma função para representar o desenho

```
class Drawing {
    list<Shape*> shapes;
    // ...
public:
    void refresh();
    // ...
```

- Desenhar as figuras na "tela"

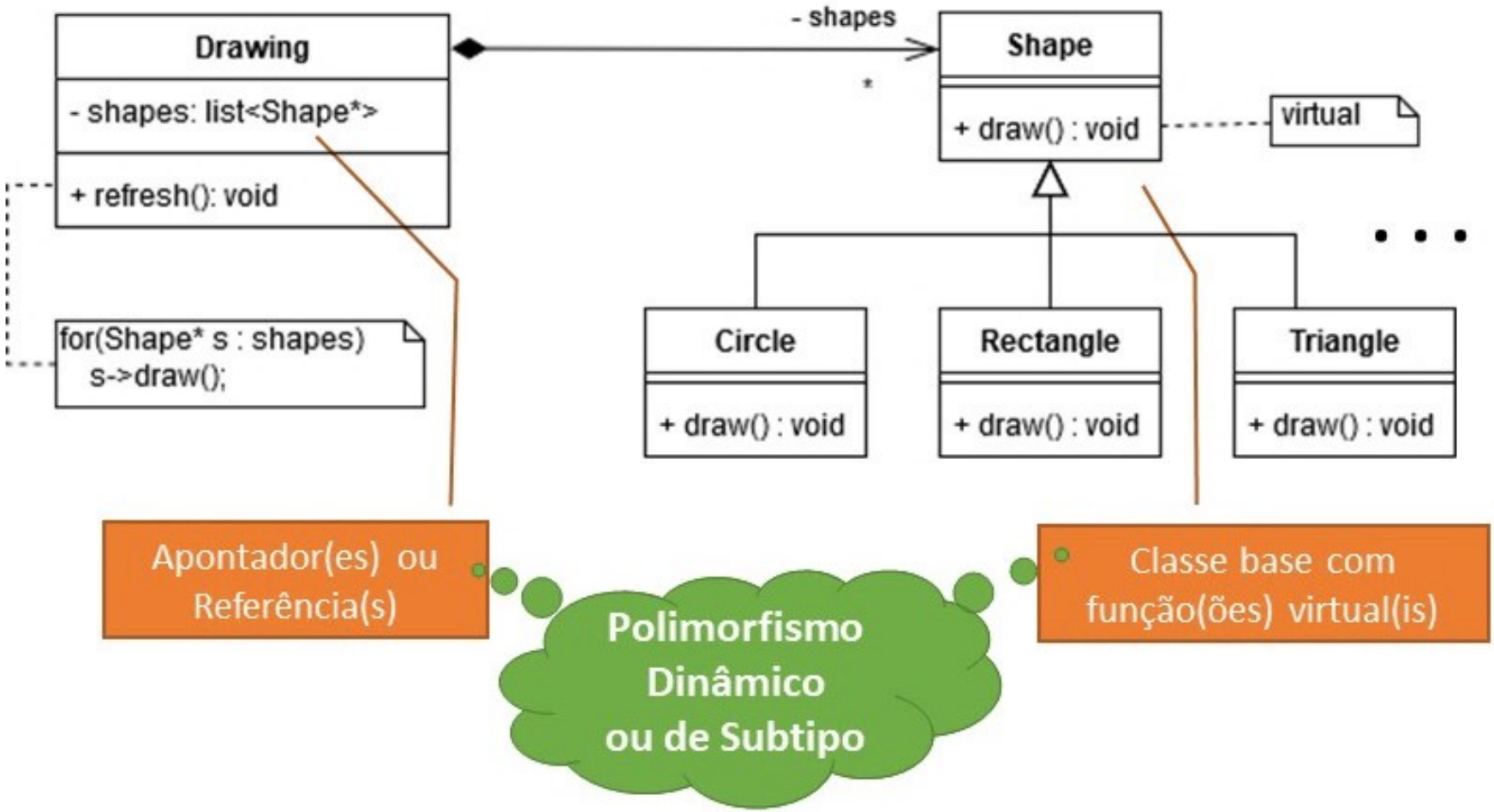
```
void Drawing::refresh() {
    for(Shape* s : shapes) {
        s->draw();
    }
}
```

É executada a função da classe específica de cada objecto

***Binding* dinâmico**

Polimorfismo dinâmico

- Exemplo em UML



Polimorfismo dinâmico

- Análise do exemplo
  - Drawing
    - Trata da mesma forma todos os objectos das diversas subclasses de Shape
    - Depende apenas de Shape
      (é independente das subclasses)

```
class Drawing {
    list<Shape*> shapes;
public:
    void refresh() {
        for(Shape* s : shapes)
            s->draw();
    }
    // ...
```

Loose coupling

  - Shape
    - draw() é escolhida em tempo de execução de acordo com o tipo efectivo de cada objecto
    - Define o que há de comum às diferentes Shapes
      (ignora os detalhes)

```
class Shape {
public:
    virtual void draw() {
        // ??...
    }
```

Abstracção

Polimorfismo dinâmico ou de subtipo

- Receita
  - Ingredientes
    A. Classe base genérica com funções virtuais (polimórfica)
    B. Classes derivadas específicas que sobrepõem os métodos virtuais
    C. Apontadores/Referências do tipo A
  - Implementação
    - Lidar com quaisquer objectos dos tipos B utilizando os apontadores/referências C para
      - Receber objectos existentes em argumentos de funções
      - Realizar todas as operações sobre os objectos

Abstracção

- Relembrar Shape

```
class Shape {
public:
    virtual void draw() {
     // ??...
    }
```

- Como implementar a função `draw()`?
  - Deixar vazia não faz sentido porque permite às subclasses reutilizarem/invocarem ou herdarem sem sobrepor
  - Produzir um erro a sinalizar que as subclasses a devem sobrepor, é melhor ser feito em tempo de compilação (pelo compilador)
  - Uma Shape não tem representação definida!

Pensando ao nível da programação

Situação é geral...

Pensando ao nível conceptual

`draw()` não deve ser implementada em Shape!

Abstracção

- Faz sentido adicionar um objecto Shape ao desenho?

```
Drawing d;
d.add(new Circle(...));
d.add(new Square(...));
d.add(new Shape(...));
d.refresh();
```

???

  - O que é uma shape?
    - Representa toda e qualquer figura
    - É um conceito!

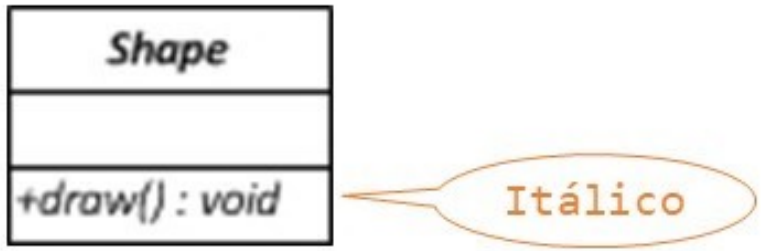
Função virtual pura

- Definição
  - Ao nível de programação
    - Função virtual igualada a zero

O compilador não procura pela implementação

```
class Shape {
public:
    virtual void draw() = 0;
    // ...
```

  - Ao nível de desenho/conceptual
    - Não tem implementação



- Consequências
  - Só "existirá" nas (instâncias das) subclasses que a definam/implementem

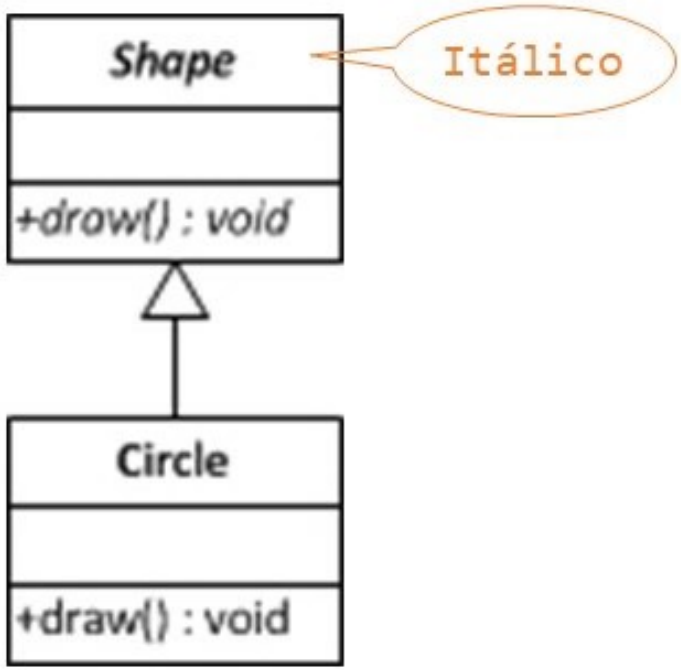
```
drawing.push_back(new Shape);
```

Erro!

A classe é **abstracta**!

Classe abstracta

- Definição
  - Ao nível de programação
    - Tem pelo menos uma função puramente virtual (sua ou herdada)
    - Tem pelo menos uma função implementada
  - Ao nível de desenho/conceptual
    - Não pode ser instanciada, pois é incompleta
    - Traduz um conceito abstracto a ser concretizado por subclasses



A definição de objectos é feita por fases, através de herança, nas subclasses

Abstracção

- Relembrar Shape

```
class Shape {
    // ...
public:
    virtual void draw() = 0;
    // ...
};
```

- O que mais deve conter Shape?
  - Ponto centro?
    - Algumas figuras têm, outras não
    - As que têm vértices podem utilizar/ter um centro ou ter os vértices
  - Apenas o que é comum a todas as subclasses
    - Área, perímetro, …

É um detalhe (de implementação)

A continuar…

Abstracção

- Completar Shape
  - Adicionar funções-membro para
    - Calcular área
    - Calcular Perímetro
    - Mover
    - ...

Também não é possivel implementá-las

```
class Shape {
public:
    virtual void draw() = 0;
    virtual float area() = 0;
    virtual float perimeter() = 0;
    virtual void move(float dx, float dy) = 0;
};
```

Shape é, mais precisamente, uma **interface**

Interface

- Definição
  - É uma classe (ou uma construção semelhante) em que todas as funções são virtuais puras

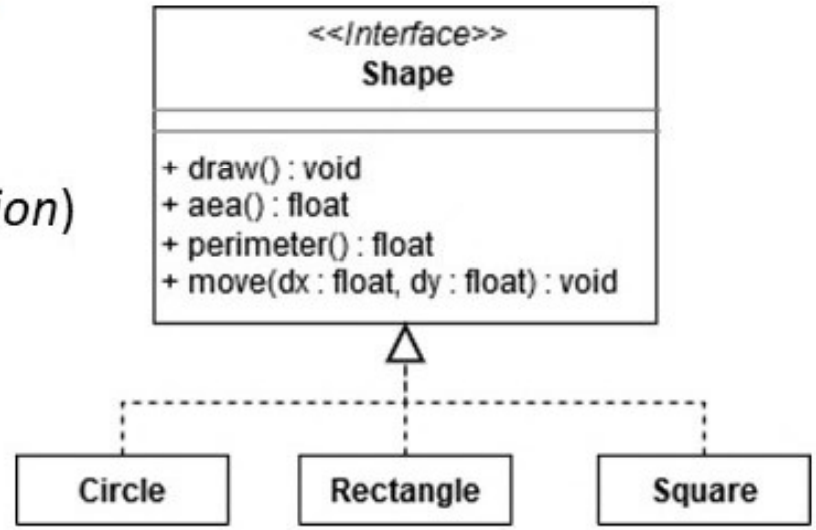
Em C++ é uma classe com certas características

Noutras linguagens (e.g., Java, C#) é uma construção de 1ª categoria

  - Conceptualmente a relação entre uma classe e uma interface chama-se de implementação (e não herança)

Em C++ é herança pública

- Representação UML
  - Estereotipo «interface»
  - Relação de implementação (*Realization*) e não herança (*Generalization*)

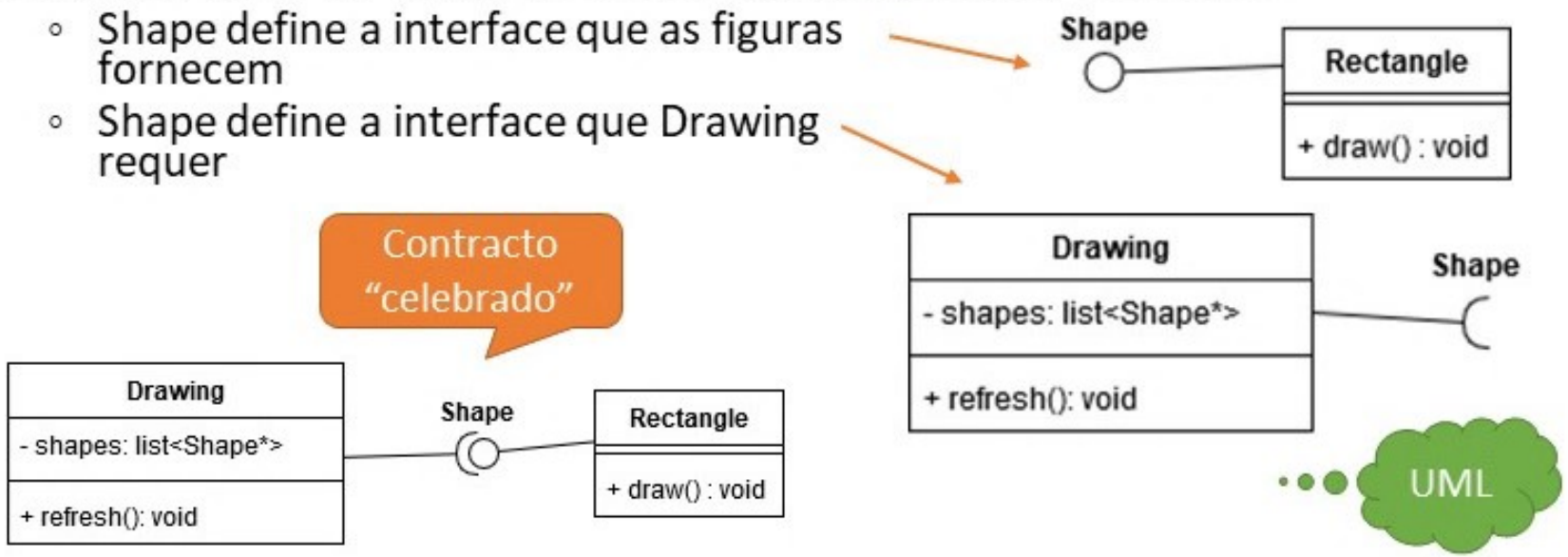


Interfaces

- Quando uma classe implementa uma interface
  - Não reutiliza/herda código, apenas definições de funções
  - (Re)utiliza-se uma ideia, **desenho (puro)**

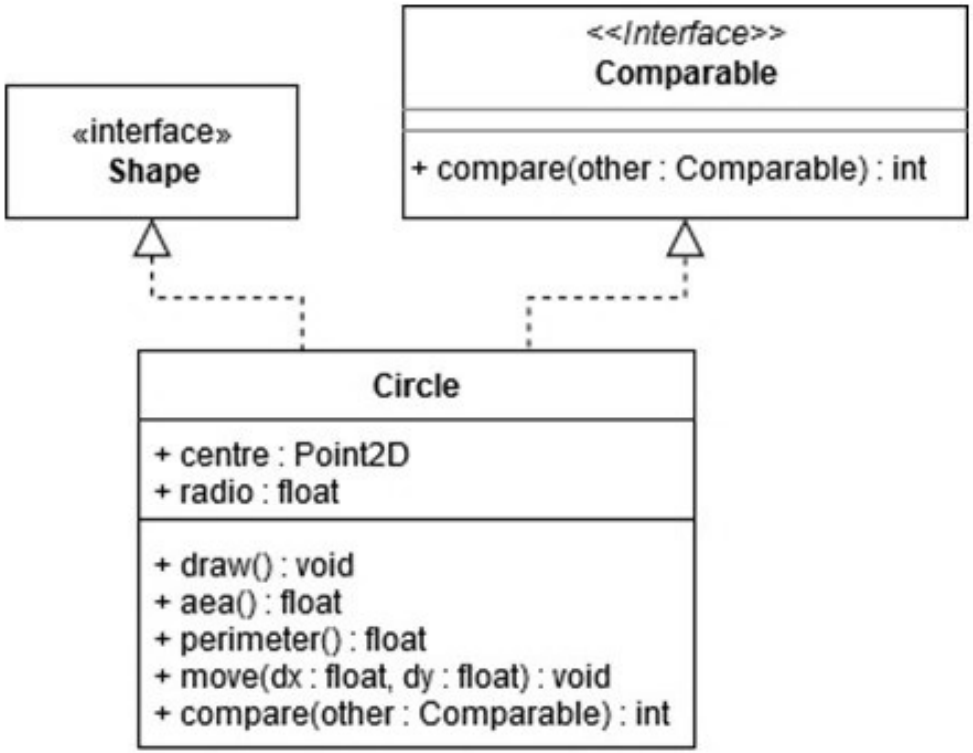
Mais importante

- Isolam as dependências
  - Drawing depende apenas de Shape e ignora as suas subclasses
  - Estão na base do conceito de **programação por contrato**
    - Shape define a interface que as figuras fornecem
    - Shape define a interface que Drawing requer



Interfaces

- Perspectiva orientada-a-objectos
  - Uma classe pode implementar várias interfaces




```
class Circle :
    public Shape,
    public Comparable {
public:
    Point2D centre;
```