Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 9 - Compilação de linguagens orientadas a objectos

na aula de hoje, abordamos a compilação de **linguagens orientadas a objetos**

curto histórico :

- Simula I et Simula 67 (anos 60)
- Smalltalk (1980)
- C++ (1983)
- Java (1995)

utilizaremos Java para fins de ilustração

breve apresentação do conceito de objecto

o primeiro conceito objeto é o de **classe** ; a declaração de uma classe introduz um novo tipo

numa primeira aproximação - grosseira, uma classe pode ser vista como um registo

```
class Polar {
    double rho;
    double theta;
}
```

aqui `rho` e `theta` são dois **campos** da classe `Polar`, de tipo `double`

cria-se uma **instância** particular de uma classe, designada de **objeto**, com a construção `new` ; assim

```
Polar p = new Polar();
```

declara uma nova variável local `p`, de tipo `Polar`, cujo valor é uma instância da classe `Polar`

o objecto é alocado na *heap*; os seus campos recebem aqui os valores por omissão (aqui 0)

podemos aceder aos campos de `p`, e modificá-los via a notação usual

```
p.rho = 2;
p.theta = 3.14159265;
double x = p.rho * Math.cos(p.theta);
p.theta = p.theta / 2;
```

podemos introduzir um ou mais **constructores**, com o objectivo de inicializar os campos do objeto

```
class Polar {
  double rho, theta;
  Polar(double r, double t) {
    if (r < 0) throw new Error("Polar: negative length");
    rho = r;
    theta = t;
  }
}
```

o que então permite escrever

```
  Polar p = new Polar(2, 3.14159265);
```

imaginemos agora que queiramos manter um **invariante** seguinte sobre os objectos da classe `Polar`

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi$$

para isso, vamos declarar os campos `rho` e `theta` como **privados**, por forma a que estes deixam de ser visíveis no exterior da classe `Polar` onde estão declarados

```
class Polar {
    private double rho, theta;
    Polar(double r, double t) { /* garante o invariante */ }
}
```

dentro de uma outra classe :

```
  p.rho = 1;
```

```
 complex.java:19: rho has private access in Polar
```

o valor do campo `rho` pode no entanto ser providenciado pelo intermédio de um **método**, isto é, de uma função fornecida pela classe `Polar` e aplicável a todo o objeto desta classe

```
class Polar {
    private double rho, theta;
    ...
    double norm() { return rho; }
}
```

para um objecto `p` de tipo `Polar`, chamamos o método `norm` da seguinte forma

```
p.norm()
```

que podemos ver ingenuamente como a chamada `norm(p)` de uma função

```
  double norm(Polar x) { return x.rho; }
```

os objetos tomam assim um papel de **encapsulamento**

o equivalente em OCaml pode ser obtido a custa de tipos abstratos

é possível declarar um campo como **estático** e estes está agora ligado a classe, e não mais às instâncias desta classe ; dito de outra forma, este campo aparenta-se assim como uma variável global associada à classe.

```
class Polar {
  double rho, theta;
  static double two_pi = 6.283185307179586;
```

de igual forma um **método** pode ser **estático** e este aparenta-se então a uma função tradicional

```
  static double normalize(double x) {
    while (x < 0) x += two_pi;
    while (x >= two_pi) x -= two_pi;
    return x;
  }
}
```

o que não é estático é designado de **dinâmico**

o segundo conceito objeto é o da **herança** : uma classe `B` pode ser definida como herdando de uma classe `A`

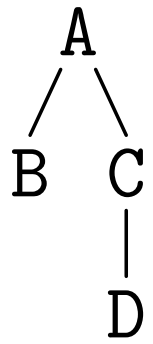
```
class B extends A { ... }
```

os objetos desta classe `B` herdam então de todos os campos e métodos da classe `A`, aos quais podemos juntar novos campos e novos métodos

a noção de herança acompanha-se de uma noção de **sub-tipagem** : todo o valor de tipo `B` pode ser visto como um valor de tipo `A` pode herdar de no máximo uma classe ; chamamos a este princípio **herança simples**, por oposição à herança múltipla (possível em linguagens como o C++)

a relação de herança forma então uma árvore

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```



consideremos uma classe `Graphical` para objetos gráficos (círculos, rectângulos, etc.)

```
class Graphical {
  int x, y;             /* centro */
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  void draw() { /* não faz nada */ }
}
```

para representar um rectângulo, herdamos da classe `Graphical`

```
class Rectangle extends Graphical {
```

herdamos assim dos campos `x`, `y`, `width` e `height` e dos métodos `move` e `draw`

podemos escrever um construtor que toma como argumento os dois lados do rectângulo

```
  Rectangle(int x1, int y1, int x2, int y2) {
    x = (x1 + x2) / 2;
    y = (y1 + y2) / 2;
    width = Math.abs(x1 - x2);
    height = Math.abs(y1 - y2);
  }
```

podemos utilizar directamente qualquer método herdado de `Graphical`

```
Rectangle p = new Rectangle(0, 0, 100, 50);
p.move(10, 5);
```

para o desenho, vamos optar por **redefinir** o método `draw` dentro da classe Rectangle (*overriding*)

```
class Rectangle extends Graphical {
  ...
  void draw() { /* desenha o rectângulo */ }
}
```

e o rectângulo será então efectivamente desenhado quando invocamos

```
p.draw();
```

procedemos de igual forma para os círculos ; aqui juntamos um campo `radius` para o raio, para poder conservar esta informação

```
class Circle extends Graphical {
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx;
    y = cy;
    radius = r;
    width = height = 2 * radius;
  }
  void draw() { /* desenho o círculo */ }
}
```

a construção `new C(...)` constrói um objeto de classe `C`, e a classe deste não pode ser mais alterada ; é designado como o **tipo dinâmico** do objecto

Em contraponto, o **tipo estático** de uma expressão, tal como este é calculado pelo compilador, pode ser diferente do tipo dinâmico, em virtude da relação de sub-tipagem introduzida pela herança.

exemplo

```
Graphical g = new Rectangle(0, 0, 100, 50);
g.draw(); // desenha o rectângulo
```

para o compilador, `g` tem por tipo `Graphical`, mas o rectângulo é convenientemente desenhado : é de facto o método `draw` da classe `Rectangle` que foi invocado

introduzimos finalmente um terceiro tipo de objectos gráficos, que são simplesmente a reunião de vários objetos gráficos

comecemos por introduzir listas encadeadas de objetos `Graphical`

```
class GList {
  Graphical g;
  GList next;
  GList(Graphical g, GList next) {
    this.g = g;
    this.next = next;
  }
}
```

(`this` designa o objeto do qual invocamos o método ; é utilizado para distinguir o parâmetro formal g do campo g de mesmo nome)

um grupo herda de `Graphical` e contém uma `GList`

```
class Group extends Graphical {
  GList group;

  Group() { group = null; }

  void add(Graphical g) {
    group = new GList(g, group);
    // + actualização de x, y, width, height
  }
```

resta redefinir os métodos `draw` e `move`

```
  void draw() {
    for (GList l = group; l != null; l = l.next)
      l.g.draw();
  }

  void move(int dx, int dy) {
    x += dx; y += dy;
    for (GList l = group; l != null; l = l.next)
      l.g.move(dx, dy);
  }
}
```

nota : é claro neste exemplo que o compilador não pode conhecer o tipo dinâmico de `l.g`

nota : não há grande sentido em criar instâncias da classe `Graphical` ; sendo assim, é o que designamos de **classe abstrata**

certos métodos, como `draw`, podem então só terem definição nas sub-classes

```
abstract class Graphical {
  int x, y;
  int width, height;

  void move(int dx, int dy) { x += dx; y += dy; }
  abstract void draw();
}
```

é então obrigatório definir `draw` em todas as sub-classes (não abstratas) de `Graphical`

em Java, vários métodos de uma mesma classe podem ter o nome desde que tenham argumentos em número diferentes e/ou de tipos diferentes ; é o que é designado por **sobrecarga** (*overloading*)

```
class Polar {
  ...
  void mult(Polar p) {
    rho *= p.rho; theta = normalize(theta + p.theta);
  }
  void mult(double f) {
    rho *= f;
  }
}
```

puis

```
  p.mult(p) ... p.mult(2.5) ...
```

a sobrecarga é resolvida durante a tipagem

é como se tivéssemos escrito

```
class Polar {
  ...
  void mult_Polar(Polar p) {
    rho *= p.rho; theta = normalize(theta + p.theta);
  }
  void mult_double(double f) {
    rho *= f;
  }
}
```

seguido por

```
  p.mult_Polar(p) ... p.mult_double(2.5) ...
```

podemos igualmente sobrecarregar os construtores

```
class Rectangle extends Graphical {
  Rectangle(int x1, int y1, int x2, int y2) {
    ...
  }
  Rectangle(int x1, int y1, int w) {
    this(x1, y1, x1 + w, y1 + w); /* constrói um quadrado */
  }
  ...
```

a noção de classe desempenha varias funções

- **encapsulamento**, a custa das regras de visibilidade
- organização do **espaço de nomes**, com base na possibilidade de utilizar o mesmo nome em classes diferentes ou para perfis diferentes.
- **factorização de código**, pela herança e a redefinição

são objectivos essenciais da **engenharia de software**, atingidos por meios diferentes em outras linguagens (exemplo : polimorfismo, ordem superior, ou ainda sistemas de módulos no caso de OCaml)

o que distinga

```
type graphical = Circle of ... | Rectangle of ...
```

de

```
class Graphical {...} class Circle extends Graphical {...}
```

em OCaml, o código de `move` está num só local e trata de todos os casos

```
  let move = function Circle _ -> ... | Rectangle _ -> ...
```

em Java, este está repartido no conjunto das classes

breve comparação functional / objeto

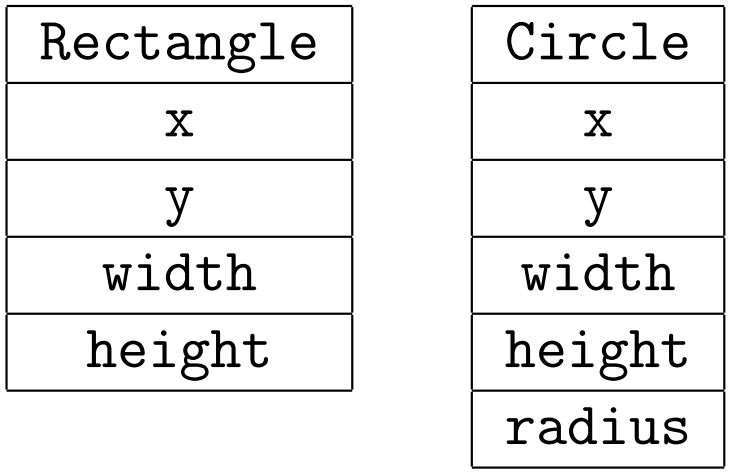
	extensão horizontal = junção de um caso	extensão vertical = junção de uma função
Java	**fácil** (um só ficheiro)	penoso (vários ficheiros)
OCaml	penoso (vários ficheiros)	**fácil** (um só ficheiro)

compilação das linguagens orientadas a objetos

um objeto é um bloco alocado na *heap*, contendo

- a sua classe
- o valor dos seus campos (como no caso de um registo)

```
class Graphical {
  int x, y, width, height; ... }
class Rectangle extends Graphical {
  ... }
class Circle extends Graphical {
  int radius; ... }

new Rectangle(0, 0, 100, 50)
new Circle(20, 20, 10)
```



o valor de um objecto é então um apontador para este bloco

nota-se que a herança simples permite arquivar o valor de um campo x num local constante dentro deste bloco ; os campos do próprio objecto são posicionados após os dos campos herdados

o cálculo dos valores direitos ou esquerdos de $e.x$ é assim facilitado

exemplo : compilamos `e.width` em

```
    # compila-se o valor de e dentro de %rcx, seguido de
    movq 24(%rcx), %rax
```

compilamos `e.width` = e' em

```
    # compilamos o valor de e dentro %rcx
    # compilamos o valor de e' dentro de %rax
    movq %rax, 24(%rcx)
```

em Java, o modo de passagem é **por valor** (mas o valor de um objecto não deixa de ser um apontador para a *heap*)

um método estático é compilado de forma tradicional

quer sejam para os constructores, os métodos estáticos ou dinâmicos, a sobrecarga é resolvida durante a compilação, e nomes distintos são dados aos diferentes constructores e métodos

```
class A {
  A() {...}
  A(int x) {...}

  void m() {...}
  void m(A a) {...}
  void m(A a, A b) {...}
```

```
class A {
  A() {...}
  A_int(int x) {...}

  void m() {...}
  void m_A(A a) {...}
  void m_A_A(A a, A b) {...}
```

a sobrecarga é no entanto mais problemática

```
class A {...}
class B extends A {
  void m(A a) {...}
  void m(B b) {...}
}
```

no código

```
{ ... B b = new B(); b.m(b); ... }
```

ambos os dois métodos podem potencialmente ser aplicados

é no entanto o método `m(B b)` que é invocada,
porque é **mais precisa** do ponto de visto do argumento

pode no entanto surgir ambiguidades

```
class A {...}
class B extends A {
  void m(A a, B b) {...}
  void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b, b); ... }
```

```
surcharge1.java:13: reference to m is ambiguous,
  both method m(A,B) in B and method m(B,A) in B match
```

a cada método definido na classe `C`

$$\tau\ \mathtt{m}(\tau_1\ x_1, ..., \tau_n\ x_n)$$

associamos o perfil $(\mathtt{C}, \tau_1, \ldots, \tau_n)$

ordenamos os perfis : $(\tau_0, \tau_1, \ldots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \ldots, \tau'_n)$ se e só se τ_i é um subtipo de τ'_i para todo o i

para uma chamada

$$e.m(e_1, \ldots, e_n)$$

onde e tem por tipo estático τ_0 e e_i o tipo estático τ_i, consideramos o conjunto dos elementos **minimais** no conjunto dos perfis compatíveis

- nenhum elemento $\Rightarrow$ nenhum método se aplica
- vário elementos $\Rightarrow$ ambiguidade
- um único elemento $\Rightarrow$ é o método por invocar

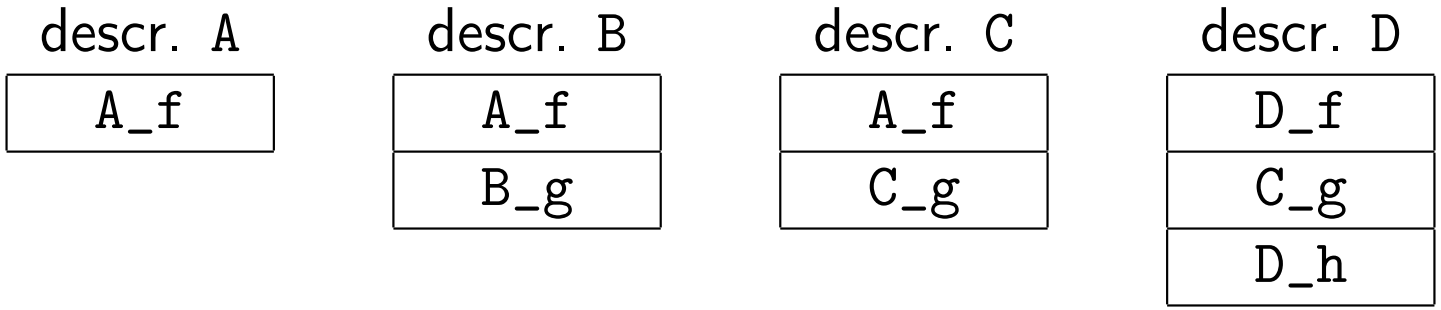
o elemento central da compilação das linguagens orientadas a objecto está na **chamada de um método dinâmico** $e.m(e_1, \ldots, e_n)$

por isso, constrói-se para cada classe um **descritor de classe** que contém os endereços dos códigos dos métodos dinâmicos desta classe

como para os campos, a herança simples permite arrumar o endereço do código de m num local constante no descriptor

os descritores de classe podem ser construídos no segmento de dados ; cada objecto contém no seu primeiro campo um apontador para o descriptor da sua classe

```
class A             { void f() {...} }
class B extends A { void g() {...} }
class C extends B { void g() {...} }
class D extends C { void f() {...}
                    void h() {...} }
```



em prática, o descriptor da classe C contém igualmente a informação da classe da qual C herda, designada de **super classe** de C

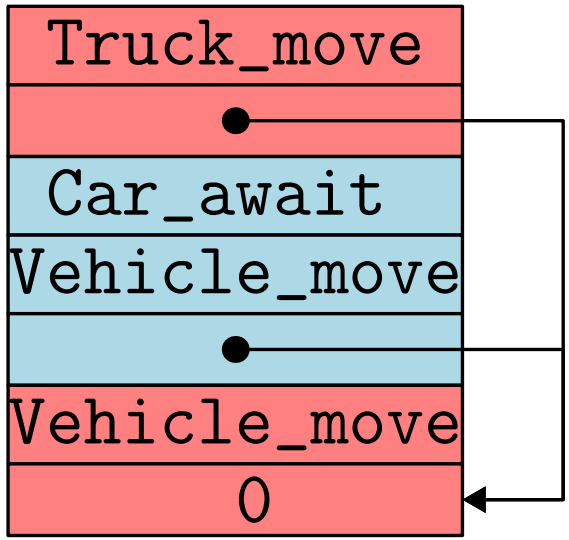
a super classe é representada por um apontador para o seu descriptor

podemos arruma-lo no primeiro campo do descriptor

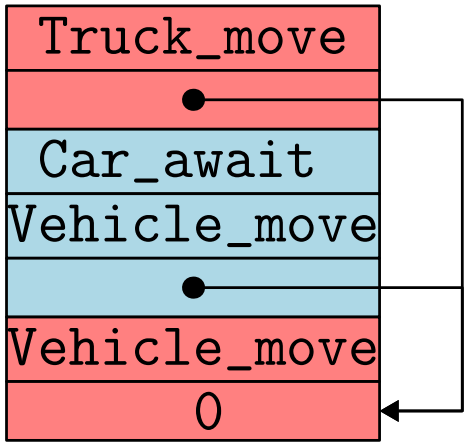
exemplo completo

```
class Vehicle {
  static int start = 10;
  int position;
  Vehicle() { position = start; }
  void move(int d) { position += d; } }

class Car extends Vehicle {
  int passengers;
  Car() { super(); }
  void await(Vehicle v) {
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10); } }
class Truck extends Vehicle {
  int load;
  Truck() { super(); }
  void move(int d) {
    if (d <= 55) position += d; else position += 55; } }
```

construímos os descritores seguintes no segmento de dados



```
        .data
descr_Vehicle:
        .quad 0
        .quad Vehicle_move
descr_Car:
        .quad descr_Vehicle
        .quad Vehicle_move
        .quad Car_await
descr_Truck:
        .quad descr_Vehicle
        .quad Truck_move
```



e o campo estático `Vehicle` está igualmente no segmento de dados

```
static_start:
        .quad 10
```

o código de um construtor é uma função que assume o objeto já alocado e o seu endereço em %rdi, o primeiro campo já preenchido (descritor de classe) e os argumentos do construtor no %rsi, %rdx, etc., e na pilha

```
class Vehicle {
  Vehicle() { position = start; }
}
```

```
new_Vehicle:
        movq    static_start, %rcx
        movq    %rcx, 8(%rdi)
        ret
```

para `Car`, o constructor limita-se a chamar o constructor da super classe, ou seja de `Vehicle`

```
class Car extends Vehicle {
  Car() { super(); }
}
```

reconhecemos aqui uma chamada terminal, o que dá simplesmente

```
new_Car:
        jmp new_Vehicle
```

de igual forma, para o constructor `Truck`

para os métodos, adotamos a mesma convenção : o objeto está em `%rdi` e os argumentos do método em `%rsi`, `%rdx`, etc., e na pilha, se necessário

```
class Vehicle {
  void move(int d) { position += d; }
}
```

```
Vehicle_move:
        addq    %rsi, 8(%rdi)
        ret
```

(de igual forma para o método `move` de `Truck`)

código com uma chamada dinâmica

```
class Car extends Vehicle {
  void await(Vehicle v) {
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10);
  }
}
```

```
Car_await:
    movq  8(%rdi), %rcx
    subq  8(%rsi), %rcx
    jle   L1
    movq  %rsi, %rdi
    movq  %rcx, %rsi
    movq  (%rdi), %rcx
    jmp   *8(%rcx)
L1: movq  $10, %rsi
    movq  (%rdi), %rcx
    jmp   *8(%rcx)
```

é um salto para um endereço calculado
(optimização com `jmp` no lugar de `call` porque a chamada é terminal)

```
class Main {
  public static void main(String arg[]) {
    Truck t = new Truck();
    Car c = new Car();
    c.passengers = 2;
    System.out.println(c.position); // 10
    c.move(60);
    System.out.println(c.position); // 70
    Vehicle v = c;
    v.move(70);
    System.out.println(c.position); // 140
    c.await(t);
    System.out.println(t.position); // 65
    System.out.println(c.position); // 140
  }
}
```

```
Truck t = new Truck();
```

começamos por alocar um bloco de 24 bytes na *heap*

```
movq  $24, %rdi
call  malloc
movq  %rax, %r12
```

arquivamos o descriptor de `Truck` no primeiro campo

```
movq  $descr_Truck, (%r12)
```

chamamos o código do constructor

```
movq  %r12, %rdi
call  new_Truck
```

(de igual forma para `c = new Car()`, arquivado em `%r13`)

a chamada

```
c.move(60);
```

está compilada em

```
movq    %r13, %rdi
movq    $60, %rsi
movq    (%rdi), %rcx
call    *8(%rcx)
```

a declaração de variáveis

```
    Vehicle v = c;
```

limita-se em criar uma aliás (um outro nome para o mesmo objecto)

se `v` está arquivado em `%r14`, o código produzido é simplesmente

```
        movq %r13, %r14
```

(é possível até dispensar esta instrução)

etc.

para mais eficiência, podemos procurar substituir uma chamada dinâmica (*i.e.* calculada durante a execução) por uma chamada estática (*i.e.* calculada durante a compilação)

para uma chamada $e.m(\ldots)$, e e de tipo estático C, é em notavelmente possível quando o método m não se encontra redefinida em nenhuma sub-classe C

uma outra possibilidade, mais complexa, consiste em propagar os tipos conhecidos durante a compilação (*type propagation*)

```
B b = new B();      // a classe de b é B
A a = b;            // a classe de a é B
a.m();              // sabemos exactamente
                    // de qual método se trata
```

como vimos, o tipo estático e o tipo dinâmico de uma expressão designando um objecto podem não coincidir (devido à sub-tipagem)

é às vezes necessário « forçar » o compilador, pretendendo que um objeto e pertence a uma certa classe C, ou mais precisamente, a uma das super-classes de C ; chamamos a isso **transtipagem** (*cast*)

a notação de Java é

$$(C)e$$

o tipo estático de uma tal expressão é C

consideremos a expressão

$$(C)e$$

sejam

- D o tipo estático da expressão e
- E o tipo dinâmico de (do objecto designado por) e

há três situações

- C é uma super classe de D : fala-se de **upcast** e o código produzido para $(C)e$ é o mesmo que para e (mas o *cast* tem influência sobre a tipagem visto o tipo de $(C)e$ ser C)
- C é uma sub-classe de D : fala-se de **downcast** e o código contém um **teste dinâmico** para verificar que E é realmente uma sub-classe de C
- C não é nem uma sub-classe nem uma super classe de D : o compilador recusa a expressão

exemplo (upcast)

```
class A {
  int x = 1;
}

class B extends A {
  int x = 2;
}
```

```
  B b = new B();
  System.out.println(b.x);       // 2
  System.out.println(((A)b).x); // 1
```

```
void m(Vehicle v, Vehicle w) {
  ((Car)v).await(w);
}
```

nada garante que o objecto passado para `m` seja bem um carro ; em particular este poderia nem sequer ter método `await` !

o teste dinâmico é por isso necessário

a excepção `ClassCastException` é levantada se o teste falha

para permitir uma programação um pouco mais defensiva. existe uma construção booleana

$$e \texttt{ instanceof } C$$

que determina se a classe de e é bem uma sub-classe de C

encontramos com alguma frequência o esquema

```
if (e instanceof C) {
  C c = (C)e;
  ...
}
```

neste caso, o compilador realiza classicamente uma optimização que consiste em não gerar um segundo teste para o *cast*

compilemos a construção

$$e \texttt{ instanceof } C$$

com a suposição de que o valor de e está em %rdi e o descritor de C em %rsi

```
instanceof:
        movq    (%rdi), %rdi
L:      cmpq    %rdi, %rsi      # mesmo descritor?
        je      Ltrue
        movq    (%rdi), %rdi    # passamos para a super-classe
        testq   %rdi, %rdi
        jz      Lfalse          # fim ?
        jmp     L
Ltrue:  ...
Lfalse: ...
```

o compilador pode optimizar as construções $(C)e$ e e `instanceof` C em certos casos

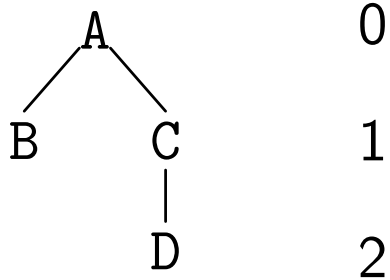
- se C é a única sub-classe de D então um único teste de igualdade no lugar de um ciclo
- se D é uma sub-classe de C então e `instanceof` C vale `true`

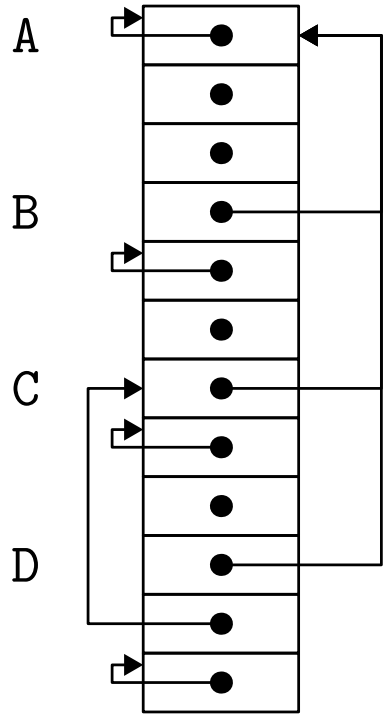
uma outra optimização é possível se o conjunto de classes é conhecido aquando da compilação ; seja n a profundidade máxima na hierarquia de classes

o descritor de uma classe C de profundidade k contém um vector de tamanho n onde as células $0..k$ contém apontadores para os descritores das super classes de C ; as outras células contém o apontador nulo

para testar se x é uma instância de D consideramos a profundidade j de D (conhecida estaticamente) e olha-se para a célula j do descritor de x para ver se há um apontador para ao descritor de D

```
class A {...}
class B extends A {...}
class C extends A {...}
class D extends C {...}
```





herança múltipla

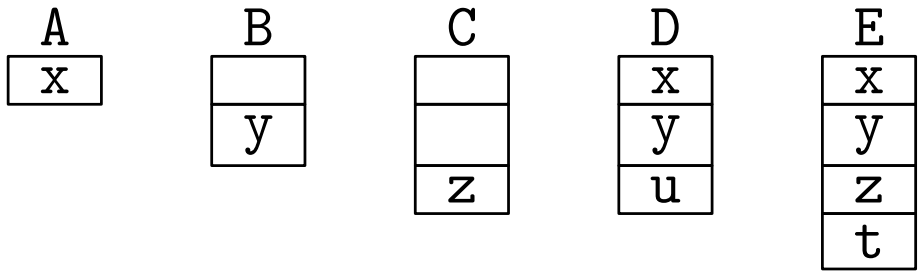
imaginemos que queiramos juntar **herança múltipla**

não podemos mais assentar no princípio segundo o qual

- a representação de um objecto de uma super-classe de C é um prefixo da representação de um objeto da classe C
- idem para os descritores de classe

```
class A { int x; ... }
class B { int y; ... }
class C { int z; ... }
class D extends A,B   { int u; ... }
class E extends A,B,C { int t; ... }
```

uma representação possível dos objectos é



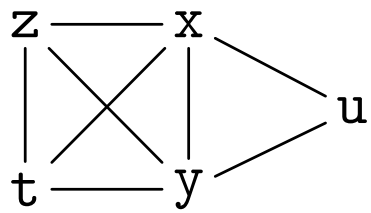
problemas :

- é preciso determinar estas localizações
- a representação não é compacta

imaginemos que se conheça o conjunto das classes aquando da compilação

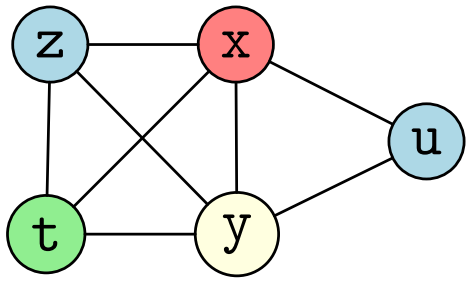
então podemos construir um **grafo de interferência**, no qual os campos f e g estão ligados desde que f e g sejam dois campos da mesma classe (directamente ou pelo intermédio da herança)

```
class A { int x; ... }
class B { int y; ... }
class C { int z; ... }
class D extends A,B   { int u; ... }
class E extends A,B,C { int t; ... }
```

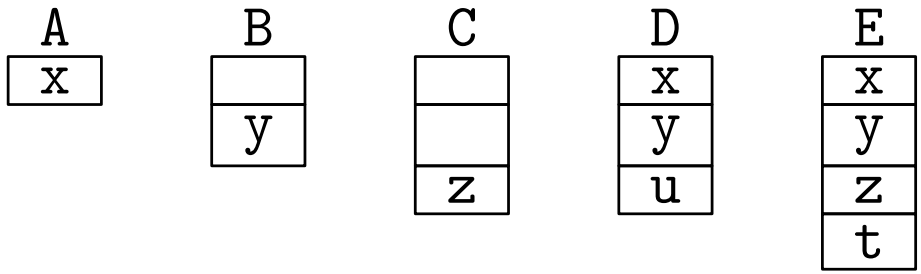


procuramos então atribuir um número mínimo de locais aos nodos deste grafo respeitando as interferências $\Rightarrow$ é um problema de **coloração de grafo**

aqui é possível de colorir este grafo com 4 cores



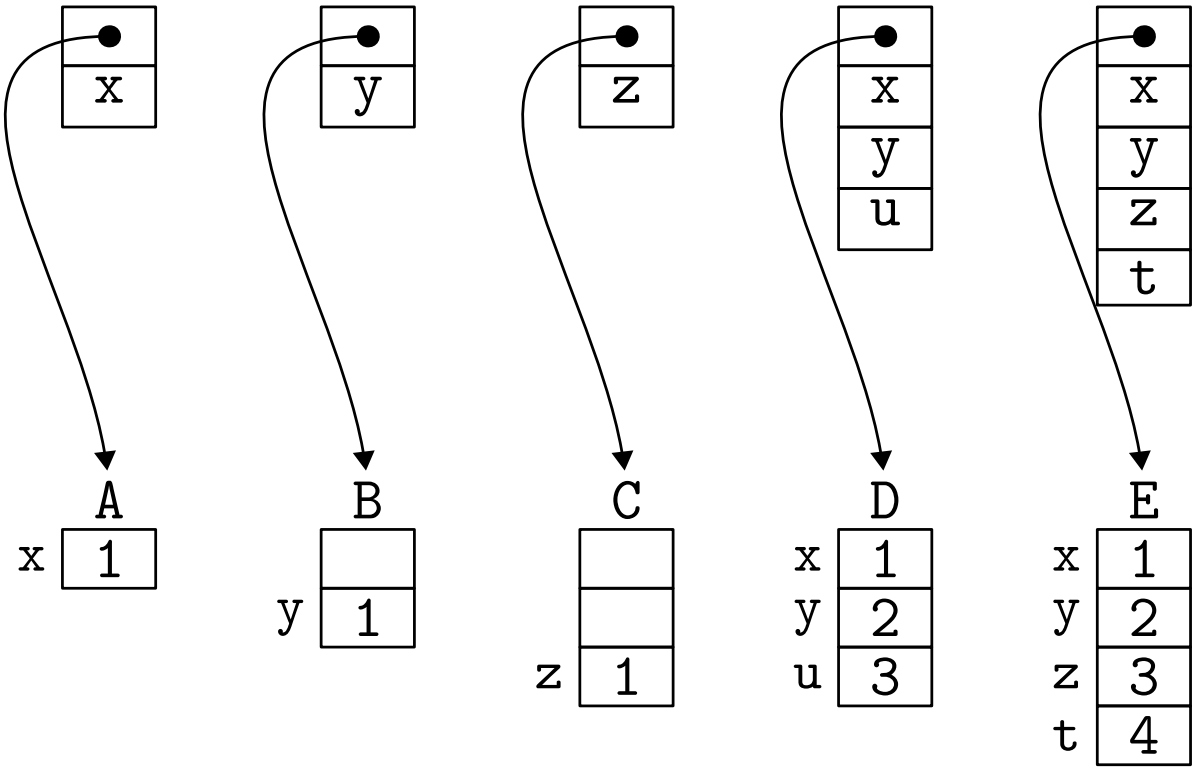
o que corresponde (entre outros) a esta escolha de representação



a mesma ideia pode ser aplicada à chamada de métodos

desejamos no entanto uma representação compacta dos objectos

para tal, juntamos uma indireção : são os descritores de classe que utilizam as localização disjuntas, para indicar as localizações reais que estão agora em locais contíguos



na prática, conhecer o conjunto das classes no momento da compilação não é realista, porque queremos

- compilação separada
- ou até, o carregamento dinâmico das classes

já não podemos calcular as localizações durante a compilação $\Rightarrow$ cada descritor de classe contém uma tabela (de dispersão) que dá o *offset* para cada campo / método

(cf Appel capítulo 14)

alguns detalhes sobre C++

retomemos o nosso exemplo dos veículos

```
class Vehicle {
  static const int start = 10;
public:
  int position;
  Vehicle() { position = start; }
  virtual void move(int d) { position += d; }
};
```

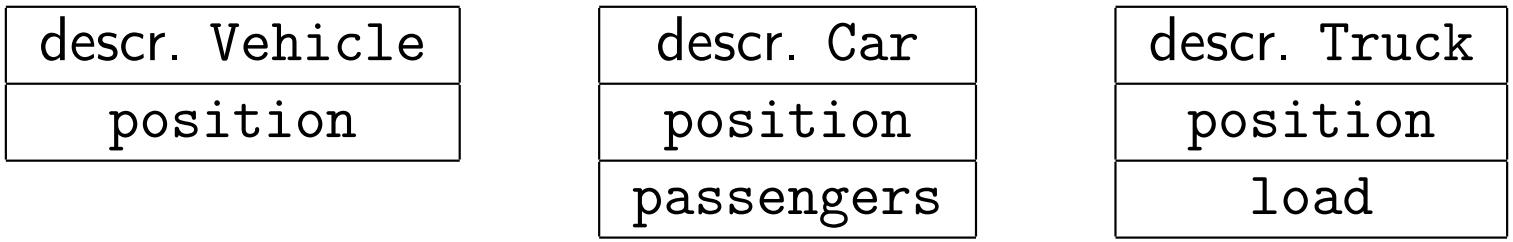
`virtual` significa que o método `move` poderá sofrer redefinição

```
class Car :  public Vehicle {
public:
  int passengers;
  Car() {}
  void await(Vehicle &v) { // passagem por referência
    if (v.position < position)
      v.move(position - v.position);
    else
      move(10);
  }
};
```

```
class Truck :  public Vehicle {
public:
  int load;
  Truck() {}
  void move(int d) {
    if (d <= 55) position += d; else position += 55;
  }
};
```

```
#include <iostream>
using namespace std;

int main() {
  Truck t;
  Car c;
  c.passengers = 2;
  cout « c.position « "\n"; // 10
  c.move(60);
  cout « c.position « "\n"; // 70
  Vehicle *v = &c; // alias
  v->move(70);
  cout « c.position « "\n"; // 140
  c.await(t);
  cout « t.position « "\n"; // 65
  cout « c.position « "\n"; // 140
}
```

neste exemplo, a representação de um objeto não é diferente da representação em Java

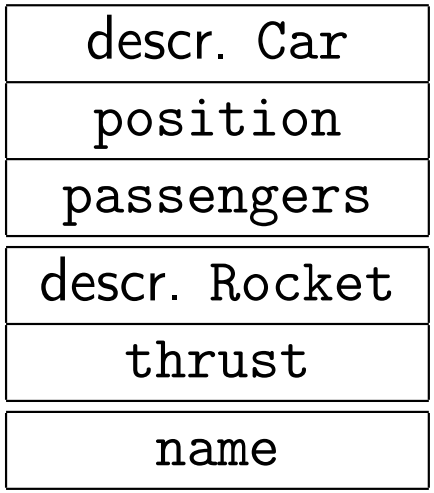


mas no C++, encontramos igualmente a possibilidade de **herança múltipla**

consequência: não podemos mais (sistematicamente) usar o princípio que diz que

- a representação de um objecto de uma super classe de C é um prefixo da representação de um objecto da classe C
- *idem* para os descriptores de classes

```
class Rocket {
public:
  float thrust;
  Rocket() { }
  virtual void display() {}
};

class RocketCar :  public Car, public Rocket
{
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```



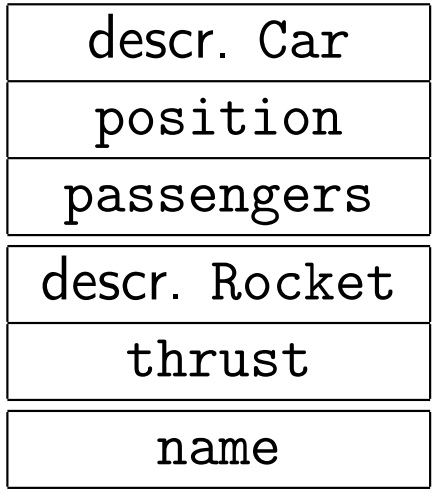
as representações de Car e Rocket são contíguas

em particular, um *cast* como

```
RocketCar rc;
Rocket r = (Rocket)rc;
```

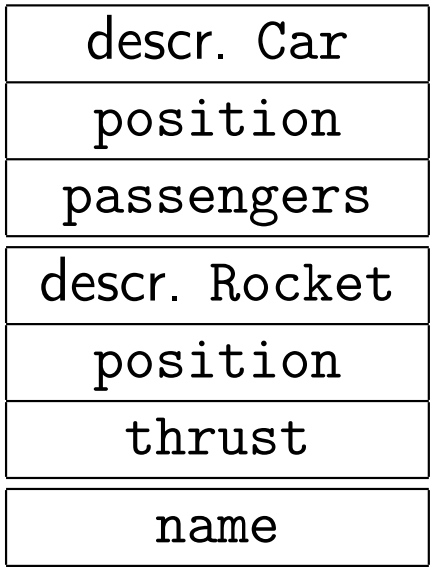
é traduzida numa aritmética de apontador

```
r <- rc + 12
```



imaginemos agora que `Rocket` herda igualmente de `Vehicle`

```
class Rocket :  public Vehicle {
public:
  float thrust;
  Rocket() { }
  virtual void display() {}
};

class RocketCar :  public Car, public Rocket
{
public:
  char *name;
  ...
};
```



temos **dois** campos `position`

há agora ambiguidade

```
class RocketCar :  public Car, public Rocket {
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```

```
vehicles.cc: In member function 'virtual void RocketCar::move(int)'
vehicles.cc:51:22: error: reference to 'position' is ambiguous
```

é necessário indicar de que campo `position` se trata

```
class RocketCar :  public Car, public Rocket {
public:
  char *name;
  void move(int d) { Rocket::position += 2*d; }
};
```

para só ter uma só instância de `Vehicle`, é preciso modificar a forma com que `Car` e `Rocket` herdam de `Vehicle` (herança virtual)

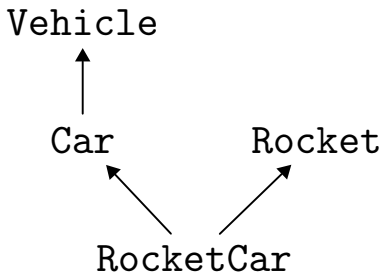
```
class Vehicle { ...  };

class Car :  public virtual Vehicle { ...  };

class Rocket :  public virtual Vehicle { ...  };

class RocketCar :  public Car, public Rocket {
```

já não há ambiguidade relativamente a `position` :

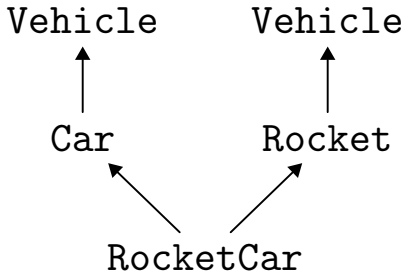
```
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```

três situações diferentes

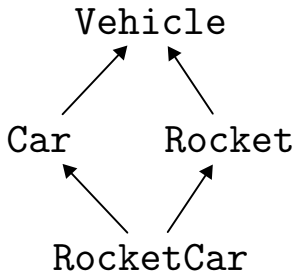
```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
```




```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket : Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```




```
class Vehicle { ... };
class Car : virtual Vehicle { ... };
class Rocket : virtual Vehicle { ... };
class RocketCar : Car, Rocket { ... };
```



(o diamante)

conclusão

- prática desta semana
  - produção de código
    (juntamos funções à mini linguagem **Arith**)
- próxima aula
  - alocação memória

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)