École Polytechnique

INF564 – Compilation

Jean-Christophe Filliâtre

stratégies d'évaluation et passage des paramètres
compilation des langages objets

stratégie d'évaluation et passage des paramètres

dans la **déclaration** d'une fonction

```
function f(x1, ..., xn) =
  ...
```

les variables `x1,...,xn` sont appelées **paramètres formels** de `f`

et dans l'**appel** de cette fonction

```
f(e1, ..., en)
```

les expressions `e1,...,en` sont appelées **paramètres effectifs** de `f` dans un langage comprenant des modifications en place, une affectation

```
e1 := e2
```

modifie un emplacement mémoire désigné par l'expression `e1`

l'expression `e1` est limitée à certaines constructions,
car des affectations comme

```
42 := 17
true := false
```

n'ont en général pas de sens

on parle de **valeur gauche** pour désigner les expressions légales à gauche d'une affectation la stratégie d'évaluation d'un langage spécifie l'ordre dans lequel les calculs sont effectués

on peut la définir à l'aide d'une sémantique formelle (cf cours 2)

le compilateur se doit de respecter la stratégie d'évaluation en particulier, la stratégie d'évaluation **peut** spécifier

- à quel moment les paramètres effectifs d'un appel sont évalués
- l'ordre d'évaluation des opérandes et des paramètres effectifs

certains aspects de l'évaluation peuvent cependant rester **non spécifiés**

cela laisse alors de la latitude au compilateur, notamment pour effectuer des optimisations (par exemple en ordonnant les calculs comme il le souhaite) on distingue notamment

- l'**évaluation stricte** : les opérandes / paramètres effectifs sont évalués avant l'opération / l'appel

  exemples : C, C++, Java, OCaml, Python

- l'**évaluation paresseuse** : les opérandes / paramètres effectifs ne sont évalués que si nécessaire

  exemples : Haskell, Clojure
  mais aussi les connectives logiques `&&` et `||` de la plupart des langages

un langage impératif adopte systématiquement une évaluation stricte, pour garantir une séquentialité des effets de bord qui coïncide avec le texte source

par exemple, le code Java

```
int r = 0;
int id(int x) { r += x; return x; }
int f(int x, int y) { return r; }

{ System.out.println(f(id(40), id(2))); }
```

affiche 42 car les deux arguments de `f` ont été évalués une exception est faite pour les connectives logiques && et || de la plupart des langages, ce qui est bien pratique

```
void insertionSort(int[] a) {
  for (int i = 1; i < a.length; i++) {
    int v = a[i], j = i;
    for (; 0 < j && v < a[j-1]; j--)
      a[j] = a[j-1];
    a[j] = v;
  }
}
```

la non-terminaison est également un effet

ainsi, le code Java

```
int loop() { while (true); return 0; }
int f(int x, int y) { return x+1; }

{ System.out.println(f(41, loop())); }
```

ne termine pas, bien que l'argument `y` n'est pas utilisé

programmation purement applicative

un langage purement applicatif (= sans traits impératifs) peut en revanche adopter la stratégie d'évaluation de son choix, car une expression aura toujours la même valeur (on parle de **transparence référentielle**)

en particulier, il peut faire le choix d'une évaluation paresseuse le programme Haskell

```
loop () = loop ()
f x y = x
main = putChar (f 'a' (loop ()))
```

termine (après avoir affiché a)

la sémantique précise également le **mode de passage** des paramètres lors d'un appel

on distingue notamment

- l'**appel par valeur** (*call by value*)
- l'**appel par référence** (*call by reference*)
- l'**appel par nom** (*call by name*)
- l'**appel par nécessité** (*call by need*)

(on parle aussi parfois de **passage** par valeur, par référence, etc.)

de **nouvelles** variables représentant les paramètres formels reçoivent les **valeurs** des paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 41
```

les paramètres formels désignent les **mêmes valeurs gauches** que les paramètres effectifs

```
function f(x) =
  x := x + 1

main() =
  int v := 41
  f(v)
  print(v)    // affiche 42
```

les paramètres effectifs sont **substitués** aux paramètres formels, textuellement, et donc évalués seulement si nécessaire

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué deux fois
  // 2+2 est évalué deux fois
  // 1/0 n'est jamais évalué
```

les paramètres effectifs ne sont évalués que si nécessaire,
mais **au plus une fois**

```
function f(x, y, z) =
  return x*x + y*y

main() =
  print(f(1+2, 2+2, 1/0)) // affiche 25
  // 1+2 est évalué une fois
  // 2+2 est évalué une fois
  // 1/0 n'est jamais évalué
```

quelques mots sur le langage Java

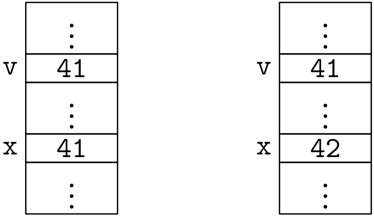
Java est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation est spécifié gauche-droite

une valeur est

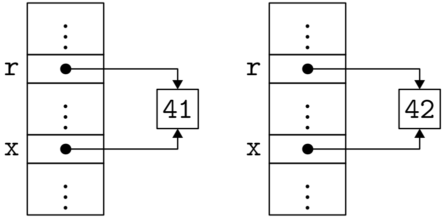
- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un objet alloué sur le tas

```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```



un objet est alloué sur le tas

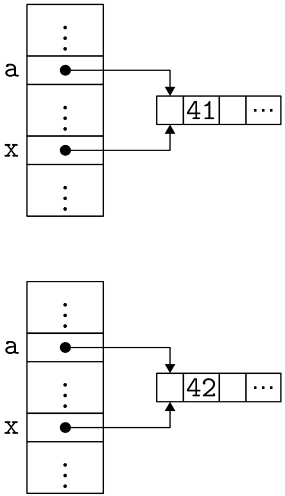
```
class C { int f; }

void incr(C x) {
  x.f += 1;
}
void main () {
  C r = new C();
  r.f = 41;
  incr(r);
  // r.f vaut maintenant 42
}
```



c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers un objet

un tableau est un objet comme un autre

```
void incr(int[] x) {
  x[1] += 1;
}
void main () {
  int[] a = new int[17];
  a[1] = 41;
  incr(a);
  // a[1] vaut maintenant 42
}
```



on peut **simuler l'appel par nom** en Java, en remplaçant les arguments par des fonctions ; ainsi, la fonction

```
int f(int x, int y) {
  if (x == 0) return 42; else return y + y;
}
```

peut être réécrite en

```
int f(Supplier<Integer> x, Supplier<Integer> y) {
  if (x.get() == 0)
    return 42;
  else
    return y.get() + y.get();
}
```

et appelée comme ceci

```
int v = f(() -> 0, () -> { throw new Error(); });
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en Java

```
class Lazy<T> implements Supplier<T> {
  private T cache = null;
  private Supplier<T> f;

  Lazy(Supplier<T> f) { this.f = f; }

  public T get() {
    if (this.cache == null) {
      this.cache = this.f.get();
      this.f = null; // permet au GC de récupérer f
    }
    return this.cache;
  }
}
```

(c'est de la mémoïsation)

et on l'utilise ainsi

```
int w = f(new Lazy<Integer>(() -> 1),
          new Lazy<Integer>(() -> { ...gros calcul... }));
```

quelques mots sur le langage OCaml

OCaml est muni d'une stratégie d'évaluation stricte, avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

une valeur est

- soit d'un type primitif (booléen, caractère, entier machine, etc.)
- soit un pointeur vers un bloc mémoire (tableau, enregistrement, constructeur non constant, etc.) alloué sur le tas en général

valeurs gauches

les valeurs gauches sont les éléments de tableaux

```
a.(2) <- true
```

et les champs mutables d'enregistrements

```
x.age <- 42
```

rappel : une référence est un enregistrement

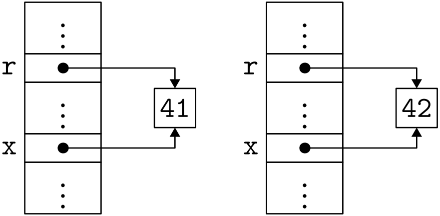
```
type 'a ref = { mutable contents: 'a }
```

et les opérations `:=` et `!` sont définies par

```
let (!)  r   = r.contents
let (:=) r v = r.contents <- v
```

une référence est allouée sur le tas

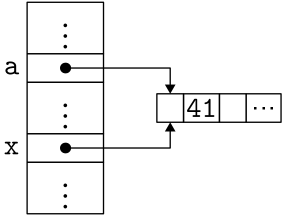
```
let incr x =
  x := !x + 1

let main () =
  let r = ref 41 in
  incr r
  (* !r vaut maintenant 42 *)
```

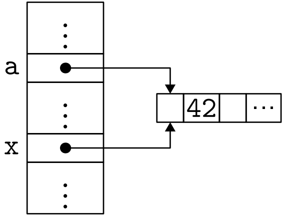


c'est toujours un passage **par valeur**,
d'une valeur qui est un pointeur (implicite) vers une valeur mutable

un tableau est également alloué sur le tas

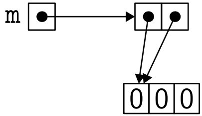
```
let incr x =
  x.(1) <- x.(1) + 1

let main () =
  let a = Array.make 17 0 in
  a.(1) <- 41;
  incr a
  (* a.(1) vaut maintenant 42 *)
```





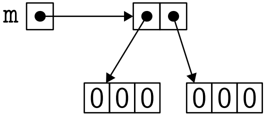
pour construire une matrice, on n'écrit pas

```
let m = Array.make 2 (Array.make 3 0)
```



mais

```
let m = Array.make_matrix 2 3 0
```

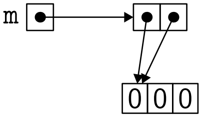


les modèles d'exécution de Java et d'OCaml sont **très semblables**, même si leurs langages de surface sont très différents

note : Python a le même modèle d'exécution, *i.e.*, passage par valeur uniquement, de valeurs toujours atomiques qui sont le plus souvent des pointeurs vers le tas

en Python, on ne construit pas non plus une matrice en faisant

```
m = [[0] * 3] * 2
```



on peut **simuler l'appel par nom** en OCaml, en remplaçant les arguments par des fonctions

ainsi, la fonction

```
let f x y =
  if x = 0 then 42 else y + y
```

peut être réécrite en

```
let f x y =
  if x () = 0 then 42 else y () + y ()
```

et appelée comme ceci

```
let v = f (fun () -> 0) (fun () -> failwith "oups")
```

plus subtilement, on peut aussi **simuler l'appel par nécessité** en OCaml

on commence par introduire un type pour représenter les calculs paresseux

```
type 'a value = Value  of 'a
              | Frozen of (unit -> 'a)

type 'a by_need = 'a value ref
```

et une fonction qui évalue un tel calcul si ce n'est pas déjà fait

```
let force l = match !l with
  | Value  v -> v
  | Frozen f -> let v = f () in l := Value v; v
```

(c'est de la mémoïsation)

on définit alors la fonction `f` comme ceci

```
let f x y =
  if force x = 0 then 42 else force y + force y
```

et on l'utilise ainsi

```
let v = f (ref (Frozen (fun () -> 1)))
          (ref (Frozen (fun () -> ...gros calcul...)))
```

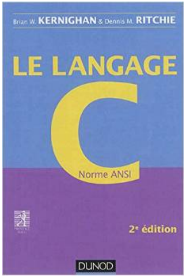
note : la construction `lazy` d'OCaml fait quelque chose de semblable (un peu plus subtilement)

quelques mots sur le langage C

le langage C est un langage impératif relativement bas niveau, notamment parce que la notion de pointeur, et d'arithmétique de pointeur, y est explicite

on peut le considérer inversement comme un assembleur de haut niveau

un ouvrage toujours d'actualité :
Le langage C
de Brian Kernighan et Dennis Ritchie



le langage C est muni d'une stratégie d'évaluation stricte,
avec appel **par valeur**

l'ordre d'évaluation n'est pas spécifié

- on trouve des types de base tels que `char`, `int`, `float`, etc. (mais pas de booléens)

- un type τ`*` des pointeurs vers des valeurs de type τ

  si *p* est un pointeur de type τ`*`, alors `*`*p* désigne la valeur pointée par *p*, de type τ

  si *e* est une valeur gauche de type τ, alors `&`*e* est un pointeur sur l'emplacement mémoire correspondant, de type τ`*`

- des enregistrements, appelés *structures*, tels que

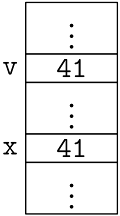
```
    struct L { int head; struct L *next; };
```

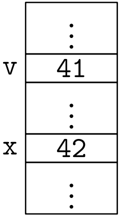
  si e a le type `struct L`, on note `e.head` l'accès au champ en C, une valeur gauche est de la forme

- `x`, une variable
- `*e`, le déréférencement d'un pointeur
- `e.x`, l'accès à un champ de structure,
  si e est elle-même une valeur gauche

- `t[e]`, qui n'est autre que `*(t+e)`
- `e->x`, qui n'est autre que `(*e).x`

```
void f(int x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut toujours 41
}
```



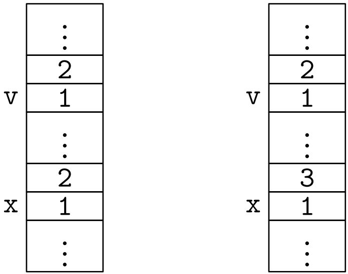


l'appel par valeur implique que les **structures sont copiées** lorsqu'elles sont passées en paramètres ou renvoyées

les structures sont également copiées lors des affectations de structures, *i.e.* des affectations de la forme `x = y`, où `x` et `y` ont le type `struct S`

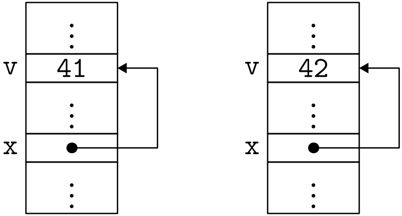
structures

```
struct S { int a; int b; };

void f(struct S x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut toujours 2
}
```



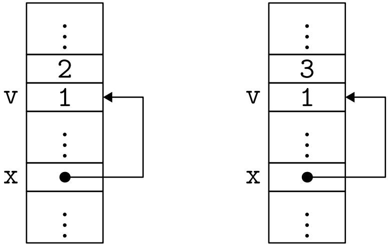
on peut **simuler** un passage par référence en passant un pointeur explicite

```
void incr(int *x) {
  *x = *x + 1;
}

int main() {
  int v = 41;
  incr(&v);
  // v vaut maintenant 42
}
```



mais ce n'est qu'un passage de pointeur **par valeur** pour éviter le coût des copies, on passe des pointeurs sur les structures le plus souvent

```
struct S { int a; int b; };

void f(struct S *x) {
  x->b = x->b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(&v);
  // v.b vaut maintenant 3
}
```



la manipulation explicite de pointeurs peut être dangereuse

```
int* p() {
  int x;
  ...
  return &x;
}
```

cette fonction renvoie un pointeur qui correspond à un emplacement sur la pile qui vient justement de disparaître (à savoir le tableau d'activation de `p`), et qui sera très probablement réutilisé rapidement par un autre tableau d'activation

on parle de référence fantôme (*dangling reference*) la notation `t[i]` n'est que du sucre syntaxique pour `*(t+i)` où

- `t` désigne un pointeur sur le début d'une zone contenant 10 entiers
- `+` désigne une opération d'*arithmétique de pointeur* (qui consiste à ajouter à `t` la quantité `4i` pour un tableau d'entiers 32 bits)

le premier élément du tableau est donc `t[0]` c'est-à-dire `*t` un tableau peut être alloué sur la pile, comme ceci

```
int t[10];
```

et il sera désalloué à la sortie de la fonction

ou sur le tas, comme ceci

```
int *t = (int*)malloc(10 * sizeof(int));
```

et il faudra le désallouer avec `free`

petite différence

on ne peut affecter des tableaux, seulement des pointeurs

ainsi, on ne peut pas écrire

```
void p() {
  int t[3];
  int u[3];
  u = t;    // <- erreur
}
```

	t[2]
	t[1]
t →	t[0]
	u[2]
	u[1]
u →	u[0]

car `t` et `u` sont des tableaux (alloués sur la pile) et l'affectation de tableaux n'est pas autorisée quand on passe un tableau en paramètre, on ne fait que passer le pointeur (par valeur, toujours)

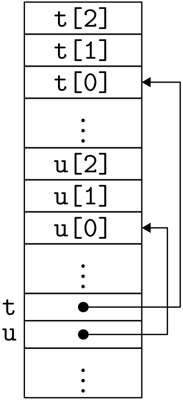
on peut donc écrire

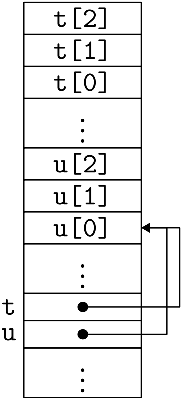
```
void q(int t[3], int u[3]) {
  u = t;
}
```

car c'est exactement la même chose que

```
void q(int *t, int *u) {
  u = t;
}
```

et l'affectation de pointeurs est autorisée





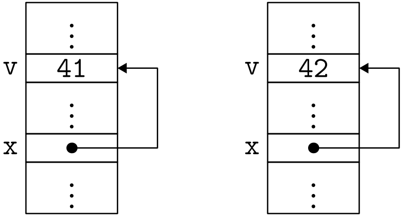
quelques mots sur le langage C++

en C++, on trouve (entre autres) les types et constructions du C, avec une stratégie d'évaluation stricte

le mode de passage est **par valeur** par défaut

mais on trouve aussi un passage **par référence**
indiqué par le symbole & au niveau de l'argument formel

```
void f(int &x) {
  x = x + 1;
}

int main() {
  int v = 41;
  f(v);
  // v vaut maintenant 42
}
```

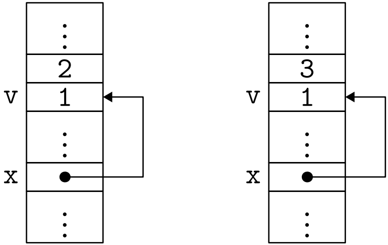


en particulier, c'est le compilateur qui

- a pris l'adresse de `v` au moment de l'appel
- a déréférencé l'adresse `x` dans la fonction `f`

on peut passer une structure par référence

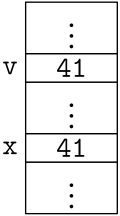
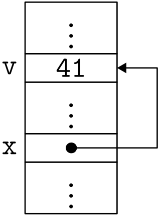
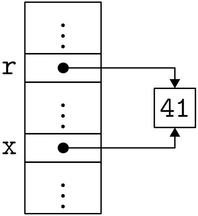
```
struct S { int a; int b; };

void f(struct S &x) {
  x.b = x.b + 1;
}

int main() {
  struct S v = { 1, 2 };
  f(v);
  // v.b vaut maintenant 3
}
```



on peut passer un pointeur par référence

par exemple pour ajouter un élément dans un arbre

```
struct Node { int elt; Node *left, *right; };

void add(Node* &t, int x) {
  if      (t == NULL ) t = create(NULL, x, NULL);
  else if (x < t->elt) add(t->left,  x);
  else if (x > t->elt) add(t->right, x);
}
```

			
C	entier par valeur	pointeur par valeur	pointeur par valeur
OCaml	entier par valeur	—	pointeur par valeur (`ref`, tableau, etc.)
Java	entier par valeur	—	pointeur par valeur (objet)
C++	entier par valeur	pointeur par valeur entier par référence	pointeur par valeur ou par référence

compilation des langages objets

expliquons

- comment un objet est représenté
- comment est réalisé l'appel d'une méthode

en prenant le cas de Java (pour l'instant) un objet est un bloc alloué sur le tas, contenant

- sa classe
- les valeurs de ses champs

la valeur d'un objet est le pointeur vers le bloc

l'héritage **simple** permet de stocker la valeur d'un champ x à un emplacement constant dans le bloc : les champs propres viennent après les champs hérités

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

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

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

chaque objet est un bloc sur le tas, de la forme

`Vehicle`
`position`

`Car`
`position`
`passengers`

`Truck`
`position`
`load`

noter l'absence du champ `start`, qui est statique donc alloué ailleurs (par exemple dans le segment de données) pour chaque champ, le compilateur connaît la position où ce champ est rangé, c'est-à-dire le décalage à ajouter au pointeur sur l'objet

si par exemple le champ `position` est rangé à la position +16 alors l'expression `e.position` est compilée comme

```
  ...                    # on compile e dans %rcx
  movl 16(%rcx), %eax    # champ position
```

ceci est correct, alors que le compilateur ne connaît que le type statique de `e`, qui peut être différent du type dynamique (la classe de l'objet)

il pourrait même s'agir d'une sous-classe de `Vehicule` non encore définie !

toute la subtilité de la compilation des langages à objets est dans l'**appel d'une méthode dynamique** $e.m(e_1, \dots, e_n)$

pour cela, on construit pour chaque classe un **descripteur de classe** qui contient les adresses des codes de méthodes dynamiques de cette classe

comme pour les champs, l'héritage simple permet de ranger l'adresse du code de la méthode m à un emplacement constant dans le descripteur

les descripteurs de classes peuvent être alloués dans le segment de données ; chaque objet contient dans son premier champ un pointeur vers le descripteur de sa classe

```
class Vehicule                   { void move(int d) {...} }
class Car    extends Vehicule { void await(Vehicule v) {...}}
class Truck extends Vehicule { void move(int d) {...} }
```

descr. Vehicule	descr. Car	descr. Truck
Vehicule_move	Vehicule_move	Truck_move
	Car_await	

pour compiler un appel comme `e.move(10)`

1. on compile `e` ; sa valeur est un pointeur vers un objet
2. cet objet contient un pointeur vers le descripteur de sa classe
3. le code la méthode `move` est situé à un emplacement connu (par exemple +8) dans ce descripteur

```
...                    # compiler e dans %rdi
movq $10, %rsi         # argument
movq (%rdi), %rcx      # descripteur de class
call *8(%rcx)          # méthode move
```

comme pour l'accès au champ, à aucun moment on n'a eu besoin de connaître la classe effective de l'objet (son type dynamique) si on écrit

```
Truck v = new Truck();
((Vehicule)v).move();
```

c'est toujours la méthode `move` de `Truck` qui est appelée
car l'appel de méthode reste compilé de la même façon

le transtypage n'a ici qu'une influence au moment du typage
(existence de la méthode + résolution de la surcharge)

en pratique, le descripteur de la classe C contient également l'indication de la classe dont C hérite, appelée **super classe** de C

la super classe est représentée par un pointeur vers son descripteur (qu'on peut ranger dans le premier champ du descripteur, par exemple)

cela permet entre autres de compiler le test dynamique derrière un *downcast* ou un `instanceof`

quelques mots sur C++

on reprend l'exemple des véhicules

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

`virtual` signifie que la méthode `move` pourra être redéfinie

```
class Car : public Vehicle {
public:
  int passengers;
  Car() {}
  void await(Vehicle &v) { // passage par référence
    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; // objets alloués ici sur la pile
  Car c;
  c.passengers = 2;
  cout << c.position << endl; // 10
  c.move(60);
  cout << c.position << endl; // 70
  Vehicle *v = &c; // alias
  v->move(70);
  cout << c.position << endl; // 140
  c.await(t);
  cout << t.position << endl; // 65
  cout << c.position << endl; // 140
}
```

sur cet exemple, la représentation d'un objet n'est pas différente de Java

descr. `Vehicle`
`position`

descr. `Car`
`position`
`passengers`

descr. `Truck`
`position`
`load`

mais en C++, on trouve aussi de l'**héritage multiple**

conséquence : on ne peut plus (toujours) utiliser le principe selon lequel

- la représentation d'un objet d'une super classe de C est un préfixe de la représentation d'un objet de la classe C
- de même pour les descripteurs 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; }
};
```

descr. RocketCar
position
passengers
descr. Rocket
thrust
name

les représentations de Car et Rocket sont juxtaposées en particulier, un transtypage comme

```
RocketCar rc;
... (Rocket)rc ...
```

est traduit par une arithmétique de pointeur

```
... rc + 12 ...
```

```
descr. RocketCar
position
passengers
descr. Rocket
thrust
name
```

supposons maintenant que `Rocket` hérite également de `Vehicle`

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

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

descr. RocketCar
position
passengers
descr. Rocket
position
thrust
name

on a maintenant **deux** champs `position` et donc une ambiguïté potentielle

```
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
```

il faut préciser de quel champ `position` il s'agit

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

pour n'avoir qu'une seule instance de `Vehicle`, il faut modifier la façon dont `Car` et `Rocket` héritent de `Vehicle` (héritage virtuel)

```
class Vehicle { ... };

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

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

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

il n'y a plus d'ambiguïté quant à `position` :

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

trois situations différentes

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

$$\begin{tikzcd}[column sep=small]
\texttt{Vehicle} & & \\
\texttt{Car} \arrow[u] & & \texttt{Rocket} \\
& \texttt{RocketCar} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

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

$$\begin{tikzcd}[column sep=small]
\texttt{Vehicle} & & \texttt{Vehicle} \\
\texttt{Car} \arrow[u] & & \texttt{Rocket} \arrow[u] \\
& \texttt{RocketCar} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

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

$$\begin{tikzcd}[column sep=small]
& \texttt{Vehicle} & \\
\texttt{Car} \arrow[ur] & & \texttt{Rocket} \arrow[ul] \\
& \texttt{RocketCar} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

(le diamant)

- TD 5
  - typage mini-C (suite)

- prochain cours
  - production de code efficace (1/4)