Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

Lecture 9 - object-oriented languages

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

today we focus on the compilation of object-oriented languages

- object layout
- dynamic dispatch

lab session: self-study / focus on projects

compiling OO languages

let us explain

- how an object is represented
- how a method call is implemented

let us use Java as an example (for the moment)

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

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

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

an object is an heap-allocated block, containing

- its class (and a few other items of information)
- the values of its fields

the value of an object is a pointer to the block

key idea: simple inheritance allows us to store the value of some field x at some fixed position in the block: own fields are placed after inherited fields

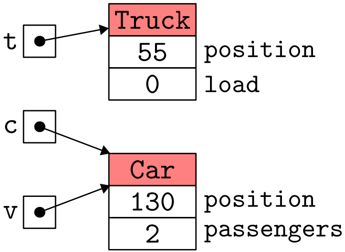
`Vehicle`
`position`

`Car`
`position`
`passengers`

`Truck`
`position`
`load`

note the absence of field `start`, which is static and thus allocated elsewhere (for instance in the data segment)

```
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
```



for each field, the compiler knows its position, which is the offset to add to the object pointer

if for instance field `position` is at offset +4, then expression `e.position` is compiled to

```
                 # we compile the value of e in $t0 and then
lw  $t1, 4($t0) # we store the content of the offset in $t1
```

this is sound, even if the compiler **only knows the static type** of `e`, which may differ from the dynamic type (the class of the object)

it could even be a sub-class of `Vehicule` that is not yet defined!

overriding is the ability to redefine a method in a subclass
(so that objects in that subclass behave differently)

example: in class `Truck`

```
class Truck extends Vehicle {
  ...
  void move(int d) { ... }
}
```

the method `move`, inherited from class `Vehicle`, is **overridden** the essence of OO languages lies in **dynamic method call** $e.m(e_1, \dots, e_n)$ (aka dynamic dispatch / message passing)

to do this, we build **class descriptors** containing addresses to method codes (aka **dispatch table**, **vtable**, etc.)

as for class fields, simple inheritance allows us to store the address of (the code of) method m at a fixed offset in this descriptor

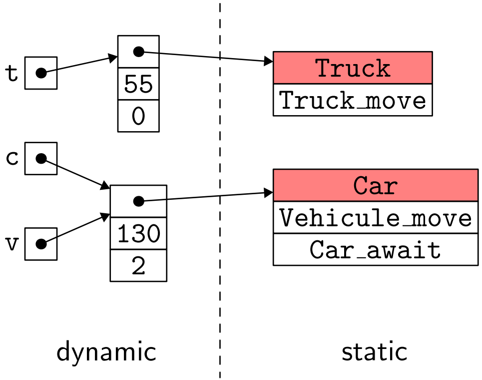
class descriptors can be allocated in the data segment; each object points to its class descriptor

example

```
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	

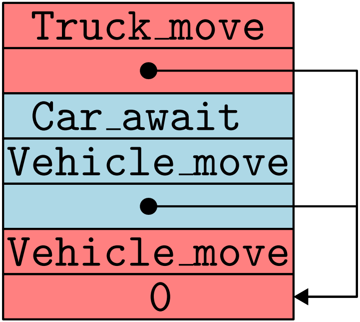
```
Truck t = new Truck();
Car c = new Car();
c.passengers = 2;
c.move(60);
Vehicle v = c;
v.move(70);
c.await(t);
```



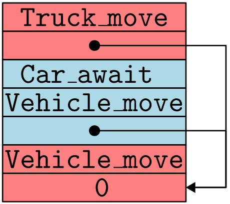
in practice, the class descriptor for C also contains points to the class that C inherits from, called the **super class** of C

this can be a pointer to the descriptor of the super class
(for instance stored in the very first slot of the descriptor)

this allows subtyping tests at runtime (*downcast* or `instanceof`)


Truck_move
Car_await
Vehicle_move
Vehicle_move
0

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



and the static field of `Vehicle` is put in the data segment as well

```
static_start:
        .word   10
```

the constructor's code is a function that assumes that *a)* the object has been allocated and its address is in `$a0` *b)* the first field (class descriptor) is defined and *c)* the arguments are in `$a1`, `$a2`, `$a3` and on the stack

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

```
new_Vehicle:
        lw      $t0, static_start
        sw      $t0, 4($a0)
        jr      $ra
```

for `Car`, the constructor just calls the one of its super class, that is `Vehicle` in our example

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

and since `super();` is a tail call to the vehicle constructor, the compilation consists of a mere jump:

```
new_Car:
        j       new_Vehicle
```

(similarly for the constructor of `Truck`) for the methods, we adopt the same convention: the object is in `$a0` and the arguments of the method in `$a1`, `$a2`, `$a3` (and stack if needed)

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

```
Vehicle_move:
        lw      $t0, 4($a0)
        add     $t0, $t0, $a1
        sw      $t0, 4($a0)
        jr      $ra
```

(similarly for the `move` method of `Truck`) code with a dynamic call

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

```
Car_await:
        lw      $t0, 4($a1)
        lw      $t1, 4($a0)
        bge     $t0, $t1, L1
        move    $a0, $a1
        sub     $a1, $t1, $t0
        lw      $t0, 0($a0)
        lw      $t0, 4($t0)
        jr      $t0
L1:
        li      $a1, 10
        lw      $t0, 0($a0)
        lw      $t0, 4($t0)
        jr      $t0
```

note a `jr` jump (instead of `jalr`) since `move(10)` is a tail call if we write

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

this is the method `move` from class `Truck` that is called
since the call is always compiled the same way

the cast only has an influence on the static type
(existence of the method + overloading resolution; see lecture 6)

```
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();
```

we start by allocating a bloc of 12 bytes on the heap

```
li       $a0, 12
li       $v0, 9  #call sbrk
syscall
```

we store the class descriptor of `Truck` in the first field

```
la       $t0, descr_Truck
sw       $t0, 0($v0)
```

we decide to put `t` in `$s1`

```
move     $s1, $v0
```

we invoke the code of the constructor

```
move     $a0, $v0
jal      new_Truck
```

(similarly for `c = new Car()`, stored in `$s2`) the call

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

is compiled into

```
move    $a0, $s2
li      $a1, 60
lw      $t0, 0($s2)
lw      $t0, 4($t0)
jalr    $t0
```

and finally we make use of `jalr` !

the variable declaration

```
    Vehicle v = c;
```

is doing nothing but creating an alias

if `v` is stored in `$s3`, the generated code is simply

```
        move    $s3, $s2
```

etc.

a few words on C++

let us reuse the vehicles example

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

`virtual` means that method `move` can be overridden

```
class Car : public Vehicle {
public:
  int passengers;
  Car() {}
  void await(Vehicle &v) { // call by reference
    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; // objects are stack-allocated
  Car c;
  c.passengers = 2;
  c.move(60);
  Vehicle *v = &c; // alias
  v->move(70);
  c.await(t);
}
```

on this example, object representation is not different from Java's

descr. `Vehicle`
`position`

descr. `Car`
`position`
`passengers`

descr. `Truck`
`position`
`load`

but in C++, we also **multiple inheritance**

consequence: we cannot use anymore the principle that

- the object layout for the super class is a prefix of the object layout of the subclass
- the descriptor for the super class is a prefix of the descriptor for the subclass

multiple inheritance

```
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

representations of Car and Rocket are appended in particular, a cast such as

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

is compiled using pointer arithmetic

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

this is not a no-op anymore

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

let us now assume that `Rocket` also inherits from `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

we now have **two** fields `position`

and thus a possible ambiguity

```
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
```

we have to say which one we refer to

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

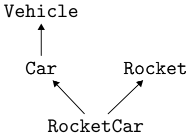
to have a single instance of `Vehicle` inside `RocketCar`, we need to modify the way `Car` and `Rocket` inherit from `Vehicle`; this is **virtual inheritance**

```
class Vehicle { ... };

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

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

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

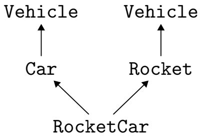
there is no ambiguity anymore:

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

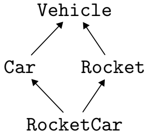
```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
```




```
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 { ... };
```



(the diamond)

`g++`'s command line option `-fdump-lang-class` outputs a text file containing objects and tables layout though Java only features simple inheritance, interfaces make method call more complex, in a way analogous to multiple inheritance

```
interface I {
  void m();
}

class C {
  void foo(I x) { x.m(); }
}
```

when compiling `x.m()`, we have no idea what the class of object `x` will be instead of dispatching according to the type of the object, we can use the types of **all** the actual parameters; this is called **multiple dispatch**

an example: Julia, a mathematically-oriented language

```
function +(x::Int64  , y::Int64  ) ... end
function +(x::Float64, y::Float64) ... end
function +(x::Date   , y::Time   ) ... end
```

another example: CLOS (Common Lisp Object System)

pattern matching, as we find in OCaml for instance, e.g.,

```
let rec eval = function
| Const n -> ...
| Call ("print", [e]) -> ...
| Call (f, el) -> ...
```

is a form of dynamic dispatch: the branch is selected according to some runtime information