Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

May 5, 2025

Lecture 9

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

Today: Compilation of Object-oriented Languages

1. Object layout
2. Dynamic dispatching

Compiling Object Oriented Languages

Compiling Object Oriented 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).

Example

```
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; } }
```

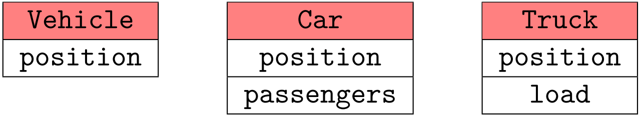
Representing Objects

An object is a 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).

Field Access

For each field, the compiler knows its position, that is the offset to add to the object pointer.

If for instance field `position` is at offset +16, then expression `e.position` is compiled to

```
...                      # compile e in %rcx
movl 16(%rcx), %rax      # field position at +16
```

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!

Terminology

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.

Method Call

The essence of OO languages lies in dynamic method call $e.m(e_1, \ldots, e_n)$ (aka dynamic dispatch / message passing).

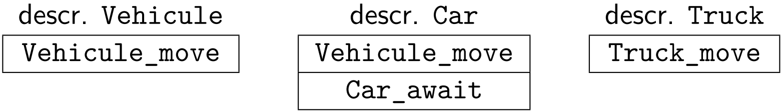
To do this, we build class descriptors containing addresses to method codes (aka dispatch table, vtable, etc.)

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

Vehicule_move

descr. Car

Vehicule_move

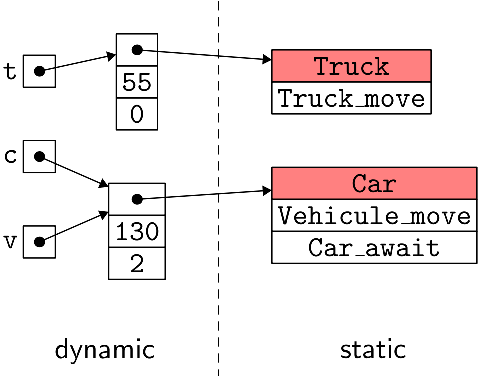
Car_await

descr. Truck

Truck_move

Example

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



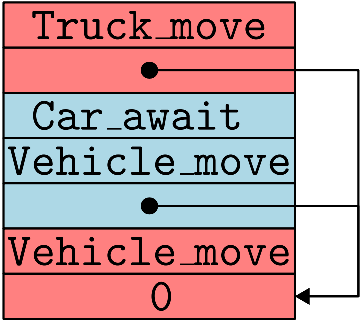
Super Class

In practice, the class descriptor for C also 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`).

Class Descriptors, Schematically



Truck_move

Car_await

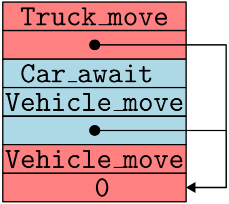
Vehicle_move

Vehicle_move

0

Class Descriptors, X86

```
        .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
```



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

```
static_start:
        .quad   10
```

Constructors

The constructor's code is a function that assumes

a) the object has been allocated and its address is in `%rdi`

b) the first field (class descriptor) is defined and

c) the arguments are in `%rsi`, `$a2`, `%rdx`, etc., and on the stack.

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

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

Methods

For the methods, we adopt the same convention: the object is in `%rdi` and the arguments of the method in `rsi`, `rdx`, etc., and stack if needed.

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

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

(similarly for the `move` method of `Truck`)

Method Call

To compile a call such as `e.move(10)`

1. we compile `e`; its value is a pointer to an object
2. this object contains a pointer to its class descriptor
3. inside, the code for method `move` is located at some offset known from the compiler (for instance +8)

```
...                   # compile e into %rdi
movq $10, %rsi        # parameter
movq (%rdi), %rcx     # get the class descriptor
call *8(%rcx)         # call method move
```

As for field access, the compiler has no need to know the actual class of the object (the dynamic type).

Be Careful

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).

A Few Words in C++

Example

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.

Example

```
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);
  }
};
```

Example (cont’d)

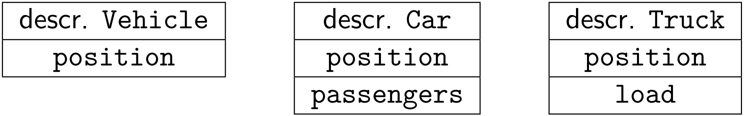
```
class Truck : public Vehicle {
public:
  int load;
  Truck() {}
  void move(int d) {
    if (d <= 55) position += d; else position += 55;
  }
};
```

Example (cont'd)

```
#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);
}
```

Representation

On this example, object representation is not different from JAVA's



descr. `Vehicle`

`position`

descr. `Car`

`position`

`passengers`

descr. `Truck`

`position`

`load`

Multiple Inheritance

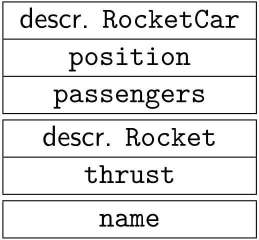
But in C++, we also multiple inheritance.

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.

Multiple Inheritance

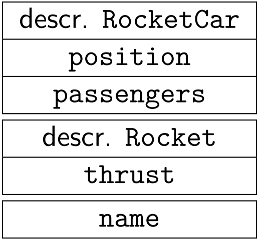
In particular, a cast such as

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

is compiled using pointer arithmetic

```
... rc + 16 ...
```

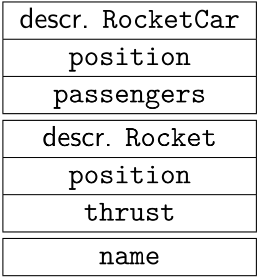
This is not a no-op anymore.



Multiple Inheritance

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`...

Multiple Inheritance

… 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
```

Multiple Inheritance

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; }
};
```

Virtual Inheritance

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.

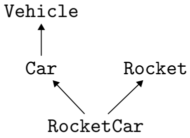
```
class Vehicle { ... };

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

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

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

There is no ambiguity anymore:

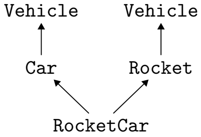
```
public:
  char *name;
  void move(int d) { position += 2*d; }
};
```

Three Class Diagrams

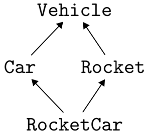
```
class Vehicle { ... };
class Car : Vehicle { ... };
class Rocket { ... };
class RocketCar : Car, Rocket { ... };
```



```
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)

If You Are Curious

g++'s command line option -fdump-lang-class outputs a text file containing objects and tables layout.

Java Interfaces

Though Java only features simple inheritance, interfaces make method call more complex, in a way analogous to multiple inheritance.

```java
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.

Multiple Dispatch

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).

Remark

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.