Object oriented programming (OOP)

Really is nothing more than a data structure problem

To understand OOP at the assembly level, 2 things need to be first understood;

*A **functions** has an address and that address can be kept in a **pointer**. **Multiple functions** pointers can be kept into an **array of functions pointers**.*

Example

What are we looking at here?

What is the text that is getting printed on the command line?

If you understand this code, whenever you are tempted to say “OOP analysis is hard” please come back to this simple example and remind yourself that it’s all about arrays of function pointers.

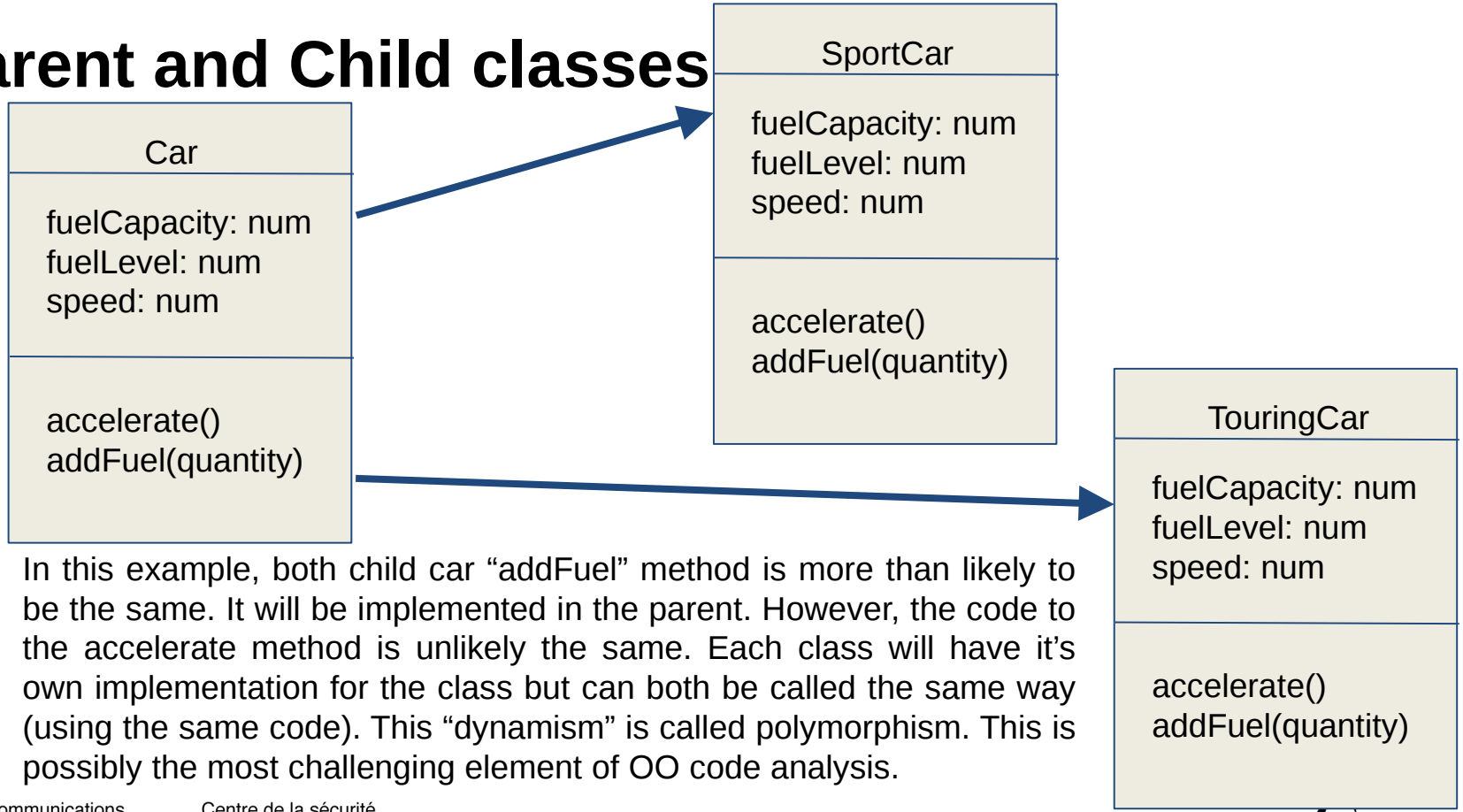
```
section .data
        greetings:
                dd english
                dd french
                dd spanish
        hello:
                db "Hello", 0x0a, 0
        bonjour:
                db "Bonjour", 0x0a, 0
        hola:
                db "Hola", 0x0a, 0
section .text
global main
extern printf

english:
        mov eax, hello
        push eax
        call printf
        pop eax
        ret
french:
        mov eax, bonjour
        push eax
        call printf
        pop eax
        ret
spanish:
        mov eax, hola
        push eax
        call printf
        pop eax
        ret
main:
        push ebp
        mov ebp, esp
        mov eax, [greetings + 4]
        call eax

        pop ebp
        ret
```

Object oriented programming (terminology)

- Class
  - The blueprint on how a part of the code will behave
- Object
  - Concretisation of a class
- Attribute
  - Classes are composed of attributes (a car has 4 wheels for example)
- Parent
  - A class that is reused in an inheritance relationship
- Child
  - A class defined as inheriting from a parent class
- Inheritance
  - Relationship between classes as Parent VS. Child

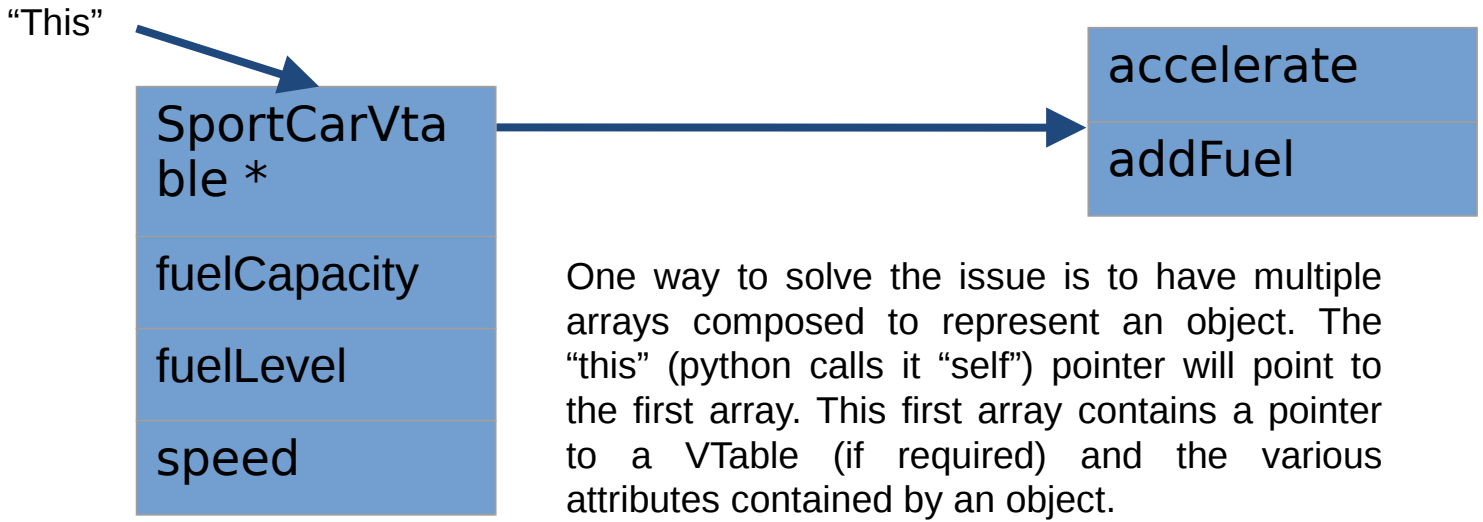
What makes OOP harder to analyse is (generally) related to inheritance and to the possibility that (at run time) functions (methods) could be dynamically called. Meaning that a method call could change at runtime.

Parent and Child classes



In this example, both child car “addFuel” method is more than likely to be the same. It will be implemented in the parent. However, the code to the accelerate method is unlikely the same. Each class will have it’s own implementation for the class but can both be called the same way (using the same code). This “dynamism” is called polymorphism. This is possibly the most challenging element of OO code analysis.

Object oriented assembly (organisation)



One way to solve the issue is to have multiple arrays composed to represent an object. The “this” (python calls it “self”) pointer will point to the first array. This first array contains a pointer to a VTable (if required) and the various attributes contained by an object.

Obviously this can be solved in multiple ways...

The VTable pointer points to an array of function pointers. At runtime, when calling a method on the object, the correct offset to the method is calculated and a call is made.

Sport car in action

```
1  section .data
2          VTableCar:
3                  dd 0
4                  dd C_addFuel
5          VTableSportCar:
6                  dd SC_accelerate
7                  dd C_addFuel
8
9          level:
10                 db "Current level is %d", 0x0a, 0
11
12 section .text
13 global main
14 extern printf, malloc, free
15
16 C_addFuel:      ;bool addFuel(this, quantity)
17         mov eax, [esp + 4] ; eax = this
18         mov ecx, [eax + 4] ; 0 is vtable, 4 is capacity
19         mov edx, [eax + 8] ; 8 is level
20         add edx, [esp + 8] ; accessing quantity
21         cmp edx, ecx
22         jle caf_levelIsGood
23         xor eax, eax
24         jmp caf_getOut
25     caf_levelIsGood:
26         mov [eax + 8], edx
27     caf_getOut:
28         ret
29
30 SC_accelerate:  ;accelerate(this)
31         mov eax, [esp + 4]
32         mov ecx, [eax + 0xc] ; 0xc is speed
33         ; you are right, this can lead to problem!
34         shl ecx, 2
35         ret
```

```
37 main:
38         push ebp
39         mov ebp, esp
40
41         sub esp, 0x04    ; making space for a local var
42         push 0x10        ; 0x10 is the size of attribute table
43                          ; including the vtable pointer
44                          ; for SportCar class
45         call malloc
46         add esp, 0x04    ; Stack cleaning
47         mov DWORD [eax], VTableSportCar
48         mov DWORD [eax + 4], 0x20       ;Capacity
49         mov DWORD [eax + 8], 0x10       ;Level
50         mov DWORD [eax + 0xC], 0x1      ;Speed
51
52         ; eax currently points to "this" for the object
53         push eax ; just keeping a safe copy of the pointer
54
55         push 0x5
56         push eax
57         mov eax, [eax]
58         call [eax + 0x04] ; Calling addFuel
59         add esp, 0x08     ; Stack cleaning
60
61         mov eax, [esp]  ; "This" should be here
62         mov ecx, [eax + 8]      ; Current level
63         push ecx
64         push level
65         call printf
66         add esp, 0x08
67
68         call free        ; No need to push since
69                          ; This is already top of stack
70         add esp, 0x08    ; Stack cleaning
71
72         pop ebp
73         ret
```

You can do everything with assembly! Isn’t that sweet =)

Let’s write some code!