1. Why Objects?

Max Cattafi
m.cattafi@imperial.ac.uk

EE2-12 – Software Engineering 2
Object Oriented Software Engineering

Imperial College
London

In this course

- More Software Engineering, i.e. strategies in the endless human struggle to tame (software) complexity.
- Building upon previous courses (Introduction to Computing, Algorithms and Data Structures. . .).
- Using Object Oriented Programming
- C++ as target programming language (introducing its OOP constructs).
- (A small subset of) UML as target modeling language.

In this lecture

- Introducing the idea of (software) objects.
- Basic OO syntax in C++ and UML.

Are you experienced?

Ever happened to you?

- The source code of a program you wrote one month (one week?) ago seems now totally obscure.
- It took *much* longer to debug a program than to write it.
- In the midst of writing some code you realized the need for a *small* change in requirements and:
  - It implied a broad restructuring of your code.
  - You decided to rather rewrite everything from scratch.
- Working on a program with someone took overall in proportion more time and effort than doing things by yourself.

The *Software Crisis*

The NATO ‘Software Engineering’ conference, (West) Germany 1968

- Why are software projects failing so often?
- Why is the price of software rising (while the price of hardware is falling)?
- Can we model *“software manufacture”* on the *“established branches of engineering”*?
  “The phrase ‘software engineering’ was deliberately chosen as being provocative.” [quotes from conference report]

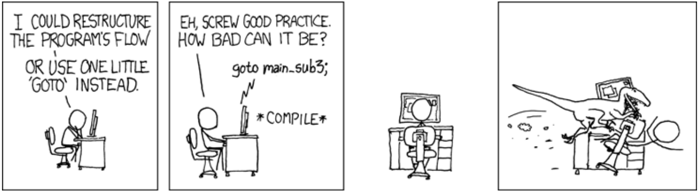
The code delusion

- *“Software is immaterial so it should be easier to produce.”*
- But what does it mean to *produce* (*‘manufacture’*) software?
- It’s more like *designing* a car, not *assembling* one!
- In addition the very kind of *required* vehicle (terrain, controls, performance. . .) changes everytime (and *all the time*)!

Why pamper life's complexity?

"The technique of mastering complexity is known since ancient times: 'Divide et impera' ('Divide and rule')."
– E.W. Dijkstra, *Programming Considered as a Human Activity*

- Structured programming:
  - Procedural decomposition in *functions*.
  - Keep control on what changes what:
    - Avoid global variables.
    - Single entry and exit points for code blocks (no *goto*).



A paradigm shift

- Computers understand mostly numbers, humans are usually concerned with concepts.
- Importance of ‘non-numerical’ computation, especially data processing in business applications.
- Programming as modeling concepts in numbers.
- But we want to operate with *conceptual* actions (*abstraction*).

The elevator object

An elevator has:

- A state represented by:
  - The position in the well.
  - The engine being off or on (and at a certain velocity).
  - The doors being closed or open (on in transition).
  - . . .
- We interact through well defined operations:
  - Calling it to the floor where we are (in some cases specifying if we want to ascend or descend).
  - Specifying the destination floor (when we are inside).
- The *state* is *encapsulated* (we are spared the actual functioning *complexity*).

Objects to objects

- Decomposition not just in terms of steps of a process but of *roles* and *responsibilities*.
- Program execution as process of entities interacting.
- Program entities apt to be represented in conceptual model (relations, hierarchies etc.) mirroring the real world.
  - Design.
- Emphasis on *what* each entity does (*interface*), *not how* (*implementation*).
- Once we (quickly) get for the first time how elevators are operated, we can use *any*.
  - Reuse, incrementality, maintainability.

The OOP four

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Déjà vu

- `string`
- `vector`

string *objects*

- Subsume C-style strings (arrays of `char`).

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string s1, s2;
    cout << "Insert the first string: " << endl;
    cin >> s1;
    cout << "Insert the second string: " << endl;
    cin >> s2;
    if( s1 == s2 ){
        cout << "The two strings are equal." << endl;
    }
    else{
        cout << "The two strings are different." << endl;
        if( s1[0] != s2[0] ){
            cout << "They do not even begin with the same letter!"
                 << endl;
        }
    }
    return 0;
}
```

string *objects*

```
#include <iostream>
#include <string>

using namespace std;

int main(){
    string tmp, fullname;
    cout << "What is your name?" << endl;
    cin >> tmp;
    fullname.append(tmp);
    cout << "What is your surname?" << endl;
    cin >> tmp;
    fullname.append(" ");
    fullname.append(tmp);
    cout << "Your full name is " << fullname << endl;
    return 0;
}
```

`std::string::append`

- `std::string::append` is a *public member function*.
- The current string content is extended by adding the argument at its end.
- We are not concerned with:
  - The internal representation of the string.
  - How its capacity changes (i.e. how memory is managed).
- Compare with C:
  - `strcat` and its buffer overflow vulnerabilities.
  - `strncat` and its additional argument for the buffer size.

Have a look inside?

```
    string tmp, fullname;
    cout << "fullname length: " << fullname.length() << endl;
    cout << "fullname capacity: " << fullname.capacity() << endl;
    cout << "What is your name?" << endl;
    cin >> tmp;
    fullname.append(tmp);
    cout << "fullname length: " << fullname.length() << endl;
    cout << "fullname capacity: " << fullname.capacity() << endl;
    cout << "What is your surname?" << endl;
    cin >> tmp;
    fullname.append(" ");
    cout << "fullname length: " << fullname.length() << endl;
    cout << "fullname capacity: " << fullname.capacity() << endl;
    fullname.append(tmp);
    cout << "fullname length: " << fullname.length() << endl;
    cout << "fullname capacity: " << fullname.capacity() << endl;
```

Double the stake

```
fullname length: 0
fullname capacity: 0
What is your name?
Max
fullname length: 3
fullname capacity: 3
What is your surname?
Cattafi
fullname length: 4
fullname capacity: 6
fullname length: 11
fullname capacity: 12
Your full name is Max Cattafi
```

- If the available memory is not enough, more is allocated.
- In this case: doubling the previously allocated one.
- No warranty of identical memory allocation pattern on other systems (implementation dependent).

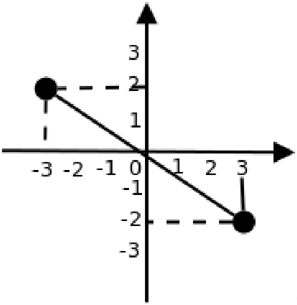
Aims

- Making proper use of objects provided by libraries.
- Defining our own.

$\rightarrow$ (Learning even better when and how to use what's already available.)

`struct Point` (with origin symmetry operations)

```cpp
// file point.hpp

struct Point{
    double x;
    double y;
};

Point origin_symmetric(Point p);
void change_origin_symmetric(Point& p);
```



(Passing by reference)

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

// in the comments comparison with C
void swap( int& a, int& b ){ // void swap(int *a, int *b){
    int c;
    c = a; // c = *a;
    a = b; // *a = *b;
    b = c; // *b = c;
}

int main(){
    int a, b;
    cout << "a: " << endl;
    cin >> a;
    cout << "b: " << endl;
    cin >> b;
    swap(a, b); // swap(&a, &b);
    cout << "a is now " << a << " while b is now " << b << endl;
    return 0;
}
```

Implementation

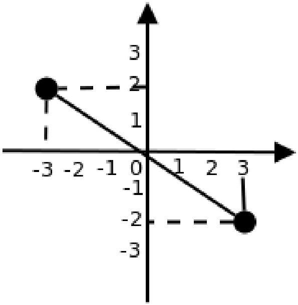
```cpp
// file point.cpp

#include "point.hpp"

Point origin_symmetric(Point p){
    Point tmp;
    tmp.x = - p.x;
    tmp.y = - p.y;
    return tmp;
}

void change_origin_symmetric(Point& p){
    p.x = - p.x;
    p.y = - p.y;
}
```

A test program

```
// file mainpoint.cpp
#include <iostream>
#include "point.hpp"

using namespace std;

int main(){
    Point p1;
    p1.x = 3;
    p1.y = -2;

    cout << "initial point: " << p1.x << " " << p1.y << endl;

    p1 = origin_symmetric(p1);
    cout << "after the first transformation: " << p1.x << " " << p1.y << endl;

    change_origin_symmetric(p1);
    cout << "after the second transformation: " << p1.x << " " << p1.y << endl;

    return 0;
}
```

Outcome

```
initial point: 3 −2
after the first transformation: −3 2
after the second transformation: 3 −2
```



`Point` objects

- The declaration `Point p1;` already means that `p1` is an object of type Point.
- However (keep in mind abstraction and encapsulation):
  - We can still change `p1.x` and `p1.y` to arbitrary values.
  - → What if only symmetry makes sense in our domain?
  - We can't write `cout « p1 « endl`.
  - We have to (remember to) initialize each field of the struct.
  - OOP constructs have a neater syntax:
    - "Message passing" style (e.g. `fullname.append(" ")` as "ask string object fullname to append a space at its end").
    - No & etc.
    - Putting closer together data and operations on data.
- (More on this involving also *inheritance*, *overriding*, *polymorphism* etc in future lectures.)

`struct Point` revised

```
// file point.hpp
struct Point{

    // functions (usually operating with or on the state) go here:
    public:
        Point origin_symmetric(Point p);
        void change_origin_symmetric(); // no parameter

    // variables representing the state go here:
    private:
        double x;
        double y;
}; // remember to end the struct with ;
```

- Bringing functions inside the struct (“member functions”, or also “methods”).

  (Struct fields: “member data” or also “attributes”.)
- Access keywords: `private` (“not accessible *from outside the class*), `public` (“accessible also from outside the class”).

Overloading (member) functions

```cpp
// file point.hpp
struct Point{

    public:
        Point origin_symmetric(Point p);
        void origin_symmetric();
        // same name, different parameters: overloading

    private:
        double x;
        double y;
};
```

- We can define functions (member or not) with the same name and different behaviour (implementation), as long as they have a different parameters list (“signature”).
- *Function overloading*.

Implementation

```
// file point.cpp
Point Point::origin_symmetric(Point p){
    Point tmp;
    tmp.x = - p.x;
    tmp.y = - p.y;
    return tmp;
}

void Point::origin_symmetric(){
    x = - x;
    y = - y;
}
```

class `Point`

```cpp
// file point.hpp
class Point{
    public:
        Point origin_symmetric(Point p);
        void origin_symmetric();
    private:
        double x;
        double y;
};
```

- In OOP: struct-like entities are usually termed *class*es.
- No difference with `struct` if visibility keywords are always used (otherwise things are `public` by default in a `struct` and `private` by default in a `class`).
- A *class* (like a struct) defines the structure to build *objects*.
- Objects are *instances* of the class.

Main?

```cpp
1  // file mainpoint.cpp
2  #include <iostream>
3  #include "point.hpp"
4  using namespace std;
5
6  int main(){
7      Point p1;
8      p1.x = 3;
9      p1.y = -2;
10     cout << "initial point: " << p1.x << " " << p1.y << endl;
11     p1 = p1.origin_symmetric(p1);
12     cout << "after the first transformation: " << p1.x << " " << p1.y << endl;
13     p1.origin_symmetric();
14     cout << "after the second transformation: " << p1.x << " " << p1.y << endl;
15     return 0;
16 }
```

BUT compiler errors: `point.hpp:8: error: 'double Point::x' is private mainpoint.cpp:8: error: within this context` and a few other analogous ones.

- `p1.x = 3`, `cout « p1.y` etc. are disabled by encapsulation (keyword `private`).

Constructors

```
// file point.hpp
class Point{
    public:
        Point();
        Point(double x_in, double y_in);
        // constructors (overloaded)
        Point origin_symmetric(Point p);
        void origin_symmetric();

    private:
        double x;
        double y;
};
```

Implementation

```cpp
// file point.cpp

...

Point::Point(){
    x = 0;
    y = 0;
}

Point::Point(double x_in, double y_in){
    x = x_in;
    y = y_in;
}
```

Constructors

Constructors are called on object instantiation.

- Special member functions: same name of the class, no return type.
- Initialize the object state together with the variable declaration (and memory allocation).
- *If and only if no constructor is defined*, a *default* one (with no arguments) is created by the compiler.

  (This is what happened every time we declared a variable of some `struct` type, e.g. `Point p1;` was calling the default constructor, created by the compiler, of `Point`.)
- If we define the constructor it's less likely that an object is initialized to a meaningless state.

Getters

```
struct Point{
    public:
        Point();
        Point(double x_in, double y_in);

        Point origin_symmetric(Point p);
        void origin_symmetric();

        double get_x{ return x; }
        double get_y{ return y; }
        // also in OOP function declaration and definition
        // can be simultaneous
        // but modularity is still preferable
    private:
        double x;
        double y;
};
```

Main

```
#include <iostream>
#include "point.hpp"

using namespace std;

int main(){
    Point p1(3, -2);

    cout << "initial point: " << p1.get_x() << " " << p1.get_y() << endl;

    p1 = p1.origin_symmetric(p1);
    cout << "after the first transformation: " << p1.get_x() << " " << p1.get_y() << endl;

    p1.origin_symmetric();
    cout << "after the second transformation: " << p1.get_x() << " " << p1.get_y() << endl;

    return 0;
}
```

Getters

- Getters enable a “read-only” access to member data.
- But also add a level of abstraction if the data representation is not (or not anymore) as one would expect (e.g. polar coordinates for points).
- Another alternative: defining a member function which prints the state (not as flexible...).
- A better alternative: defining our own meaning for the « operator (more on this in future lectures).

const references

- Consider `Point origin_symmetric(Point p);`.
- Passing by value means having to make a *copy*.
- For objects it can be inefficient (call to constructor, etc. we'll see more in detail).
- Passing by reference implies having to assume that the function can change the state of the argument we pass (still within the boundaries of access keywords).
- Passing by const reference:
  - No need to copy.
  - No need to worry about changes.

- In C++ `const` is for more than just defining constants.
- *Const correctness*.

Checking on `string::append`

For instance…

```
string::append ( const string& str );
```

- Using `const <type>&` parameters:
  - Passing by reference (of type <type>).
  - No changes are allowed on the variable referenced (*const*).
- `Point origin_symmetric(const Point& p);`

const member functions

```
Point Point::origin_symmetric(const Point& p){
    Point tmp;
    tmp.x = - p.x;
    tmp.y = - p.y;
    return tmp;
}
```

- No change on the current state (local `x` and `y` are not even read!).
- It's useful to mark this.
- (Notice that *from the same class* also `private` member data of *other objects* can be accessed.)

const member functions

```
// point.hpp
class Point{
    public:
    ...
    Point Point::origin_symmetric(const Point& p) const;
    ...
};

// point.cpp
...
Point Point::origin_symmetric(const Point& p) const{
    Point tmp;
    tmp.x = - p.x;
    tmp.y = - p.y;
    return tmp;
}
...
```

UML Point

```
class Point{
    public:
        Point();
        Point(double x_in, double y_in);

        Point origin_symmetric(const Point& p) const;
        void origin_symmetric();

        double get_x() const;
        double get_y() const;

    private:
        double x;
        double y;
};
```