OOP Fundamentals C++ Aspects

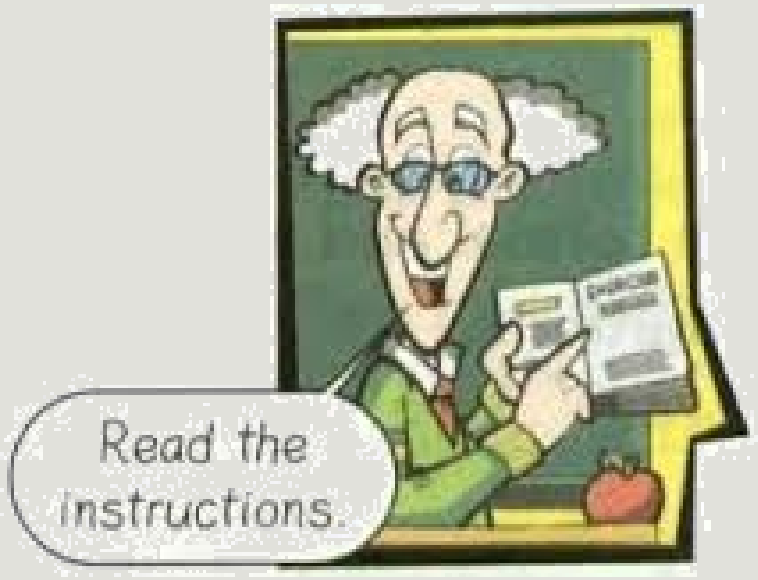
FUNDAMENTAL OOP ASPECTS. CLASSES AND OBJECTS

Programming paradigms

HOW DO YOU APPROACH A PROBLEM?

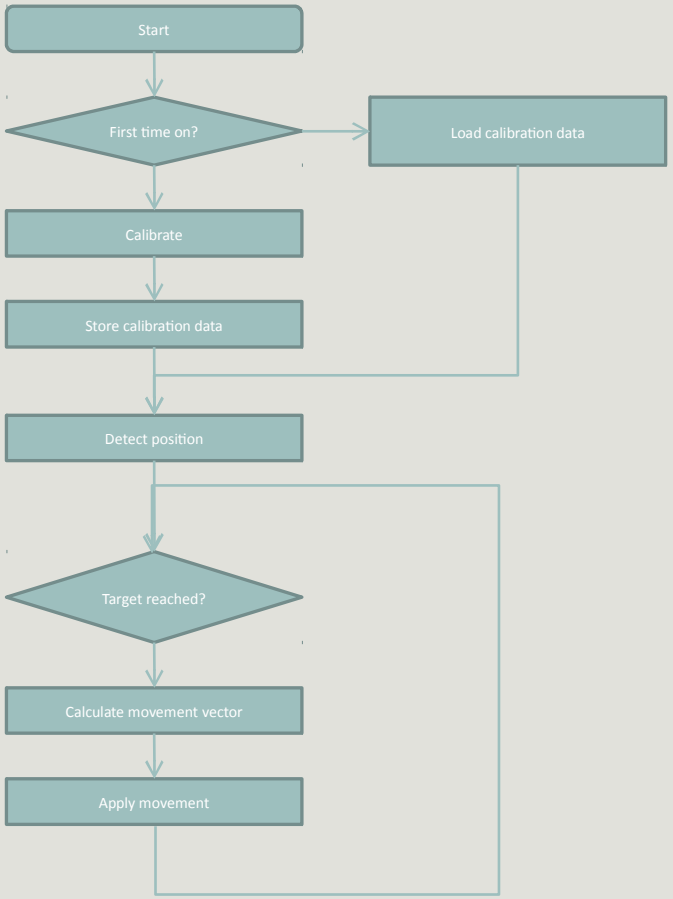
Programming paradigms
Imperative programming- Intro

- *"In computer science terminologies, **imperative programming** is a **programming** paradigm that describes computation in terms of statements that change a **program** state."*
- "Do this – do that" approach



Programming paradigms
Imperative programming – contd.

- A problem is approached using “step-by-step” modeling: break down the functionality into series of steps
- Best in products where sequence of steps is fixed or rarely changed
- Finer-level steps can be combined into larger blocks (***procedures***) to produce clear, concise and manageable sequence at each level



Programming paradigms
Imperative programming – Procedural

- Focus on ***procedures*** – a series of instructions that can be called from any point in the program
  - In *C/C++* - ***"functions"***
- ***Procedures*** help achieve some degree of ***modularity***
- Modularity allows for ***testability***, ***reusability***, ***maintainability***, …

Programming paradigms
Imperative programming – Pros & Cons

- Pros
  - Matches the underlying technology
  - Easy to create or comprehend, especially for small systems
  - Easy to derive directly from user requirements
  - (? pro/con) Supported by modern object-oriented and (some) functional languages
- Cons
  - No direct way to communicate the relation (and consistency) between data and procedures/functions
  - Basic reusability unit is ***function*** which is not very stable and is prone to frequent changes for both requirement and technical reasons
  - Does not reflect correctly the ***real world***, which is made of ***interacting entities***, not ***separate*** activities and data

Programming paradigms
Declarative programming

- *Unofficially: "Style of programming that is **not** imperative"*
- *"In computer science, declarative programming is a programming paradigm, a style of building the structure and elements of computer programs, that expresses the **logic** of a computation **without** describing its **control flow**." - Wikipedia*
- *"A program that describes **what** computation should be performed and not **how** to compute it"*

Programming paradigms
Declarative programming- Examples

- *SQL Example*

```
SELECT *
FROM Students
WHERE age > 35
ORDER BY name
```

- *Functional programming*

Programming paradigms
Object oriented programming- Intro

- Where does it stand? Is it *Imperative*, or *Declarative*?
- Based on the concept of ***objects*** which combine ***data*** and ***behavior***
- When solving a problem, the primary focus is on ***objects*** and their ***relations/interactions***
- Example: a ***Car*** *has an* ***Engine***, ***Chassis***, ***Wheels***, … Then ***Engine*** itself has ***Cylinders***, ***Sensors***, etc.

Programming paradigms
Object oriented programming – Pros & Cons

- Pros
  - Closer to the real world – **entities** that we want to model are often directly **object-oriented** ready
  - The link between data and processing code is built into the syntax – the system is ***aware*** about it
  - Reusability units are much more obvious (often a ***class*** is directly reusable) and, with proper design, much more stable than functions
- Cons
  - Requirements are not very object-oriented; from them, suitable ***objects*** / ***classes*** need to be extracted and defined by the software Designers / Architects
  - The underlying hardware, storage, communications, etc. operate as series of operations, i.e. closer to ***Imperative*** paradigm

Object Oriented Programming Fundamentals

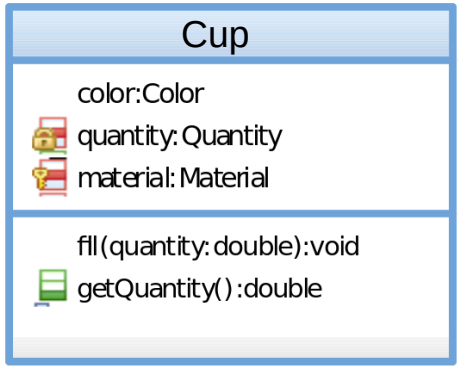
NO OBJECTIONS :)

Object Oriented Programming
Introduction

- Completely different way to approach a problem
- Start by describing ***objects***, their ***relations*** and ***interactions***
- Key question to start in OOP way: *"**What** are we talking about?" (instead of "what the program will **do**"!)*
- At the end, objects' behavior is still ***imperatively*** described (i.e. method implementations)
- In short, ***objects*** combine ***data*** and ***behavior***
- A developer can think in object-oriented manner and still use "procedural" language such as C
- However, in ***object-oriented*** languages there is suitable syntax to ***express*** the link between data and behavior of an ***object***
  - This helps ***find errors***, ease the process of ***development*** and developer ***testing***, ***redesign*** and/or expand the system more safely, etc.

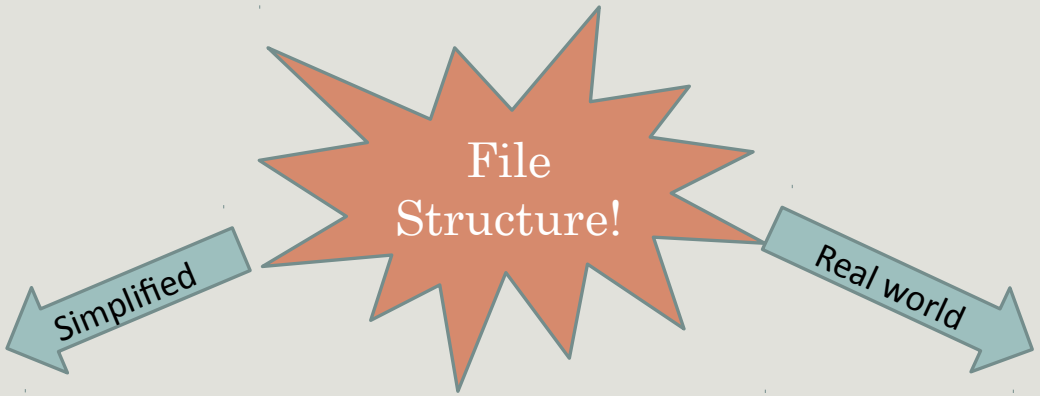
Object oriented programming
Objects & Classes – Access Modifiers

- ***Access modifiers*** (in a class) specify from where is a certain member (attribute or operation) accessible.
  - In example, for an ***attribute*** the term "access" can mean read its value, write its value, get address of, etc. For an ***operation*** "access" usually means to *execute the operaton (call* the method), but might also mean get its address.
- *Strongest* specifier is ***private***: only code belonging to the class itself can access a ***private*** member. *"Code"* here roughly means functions / methods
- *Weakest* specifier is ***public***: all functions and methods can access a ***public*** member
- There is a number of intermediate levels of access, in C++ only ***protected*** ()



Object Oriented Programming
Classes & Objects in C++

```
// main.cpp

class Cup {

public:

      void fill(double quantity) {

        this->quantity += quantity;

      }

      double getQuantity() {

            return quantity;

      }

private:

  double capacity;

  double quantity;

};
```




```
// cup.cpp
#include "Cup.hpp"

void Cup::fill(double quantity) {
      this->quantity += quantity;
}

double Cup::getQuantity() {
      return quantity;
}
```

```
// cup.hpp
#ifndef CUP_HPP_
#define CUP_HPP_

class Cup {
public:
      void fill(double quantity);
      double getQuantity();

private:
      double capacity;
      double quantity;
};

#endif /* CUP_HPP_ */
```

Object Oriented Programming Encapsulation

HIDING SECRETS

Object Oriented Programming
Encapsulation – Why & How ?

- Expose as little as possible to the outside world
  - ***Why****?*
- In a successful project, ***everything*** you expose (*attributes* and *operations*), ***will*** be used ***from outside***. And then the exposed attributes and operations become much harder to change (i.e. to improve, optimize, react to requirements change, fix bugs, etc.) because a lot of "foreign code" depends on them
- *How* to expose/hide? We already introduced the *access modifiers.* In C++
  - ***public*** – every piece of code can access members having this modifier
  - ***protected*** – only methods of the class itself and methods of its ***descendants*** can access members having this modifier
  - ***private*** – only methods of the class itself can access members having this modifier

Object Oriented Programming
Encapsulation – Interfaces. Best practices

- Several different, but logically close meanings of the term "Interface"
- ***Interface*** of a class – the ***exposed members*** of that class (operations & attributes)
  - Usually *"exposed" means **public**; but **protected** members are also a (special) interface for the descendants*
- ***Interface*** can also be a special construct in a given language, allowing to have just the "interface part" as a completely separate entity from implementation
  - In ***Java*** this is achieved using the ***interface*** keyword
  - In ***C++*** this is achieved by defining ***pure abstract classes*** (presented later in the course)
- *Synonym – **API** (Application Programming Interface)*
- What is better to expose (include in the ***interface***): ***attributes*** or ***operations***?
  - ***Why?***
  - The importance of having *control*
- Besides ***operations***, what else can be *safely* included in an ***interface***?
- ***Think carefully*** when designing interfaces!
  - *Think now, avoid problems later!*
- ***!!!** Correct usage of **OOP relies on interfaces** being **more stable** than **implementations !!!***
  - *So in order to use OOP **correctly**, your emphasis **must be** on designing the **interfaces** instead of implementation details*
  - *Make OOP your ally*