EE2-12 Software Engineering 2: Object-Oriented Programming

Week 1 - Classes and Objects I: Introduction

Sahbi Ben Ismail

Imperial College London, Department of Electrical and Electronic Engineering

Imperial College London

Autumn Term 2018-19

Module Syllabus

- Week 1 **Classes and Objects I: Introduction**
- Week 2 Classes and Objects II: Constructors, and Operator Overloading
- Week 3 Objects and Dynamic Memory
- Week 4 Classes Relationships I: Association, Aggregation, and Composition
- Week 5 Classes Relationships II: Generalisation/Inheritance
- Week 6 Polymorphism and Virtual Functions
- Week 7 Generic Programming: Templates, and the Standard Template Library (STL)
- Week 8 Exceptions Handling
- Week 9 C++ to Java
- Week 10 Revision

Week 1 Intended Learning Outcomes (ILOs)

By the end of this week you should be better able to:

Lecture

1. Understand the difference between Structures and Classes in C++
2. Understand the difference between Classes and Objects
3. Apply UML notations and encapsulation rules to model a basic class

Lab

1. Write a class declaration and definition in C++, and test it by instantiating objects
2. Build a basic C++ software architecture using a Makefile

Outline

1. Introduction: Programming paradigms
2. Structures
3. Classes
4. UML notations

Outline

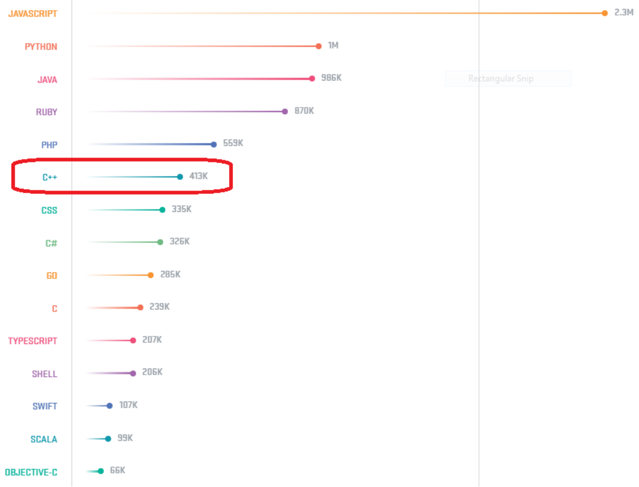
1. Introduction: Programming paradigms
2. Structures
3. Classes
4. UML notations

Why C++ in EIE1 and EIE2?

- Popularity
  - **TIOBE**: Programming Community Index https://www.tiobe.com/tiobe-index/
  - **GitHub**: GitHub Octoverse 2017,Highlights from the last twelve months https://octoverse.github.com/
  - **Job** research websites (Indeed.com, totaljobs.com, etc...)
- **Department** suitability: EEE applications
- **Employability**: Coding interviews (alongside with EE1-8), Job market (including the City)
- Relatively easy **transition** to other OO languages (Java, C#, etc..)

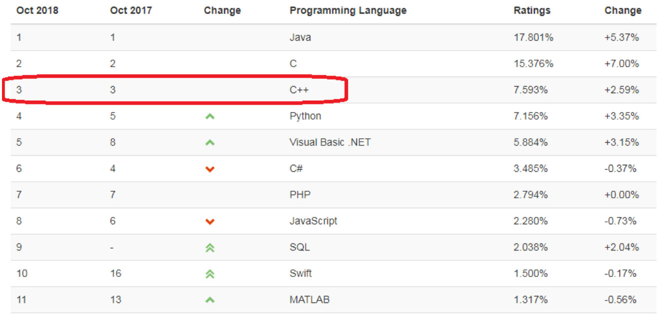
The fifteen most popular languages on GitHub

by opened pull request



TIOBE Programming Community Index

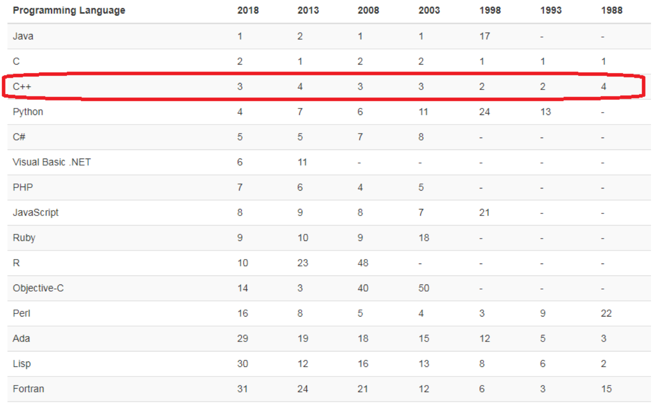
TIOBE Index for October 2018: #3



Oct 2018	Oct 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.801%	+5.37%
2	2		C	15.376%	+7.00%
3	3		C++	7.593%	+2.59%
4	5	▲	Python	7.156%	+3.35%
5	8	▲	Visual Basic .NET	5.884%	+3.15%
6	4	▼	C#	3.485%	-0.37%
7	7		PHP	2.794%	+0.00%
8	6	▼	JavaScript	2.280%	-0.73%
9	-	⏫	SQL	2.038%	+2.04%
10	16	⏫	Swift	1.500%	-0.17%
11	13	▲	MATLAB	1.317%	-0.56%

TIOBE Programming Community Index

Very Long Term History: Top 3 or 4



Programming Language	2018	2013	2008	2003	1998	1993	1988
Java	1	2	1	1	17	-	-
C	2	1	2	2	1	1	1
C++	3	4	3	3	2	2	4
Python	4	7	6	11	24	13	-
C#	5	5	7	8	-	-	-
Visual Basic .NET	6	11	-	-	-	-	-
PHP	7	6	4	5	-	-	-
JavaScript	8	9	8	7	21	-	-
Ruby	9	10	9	18	-	-	-
R	10	23	48	-	-	-	-
Objective-C	14	3	40	50	-	-	-
Perl	16	8	5	4	3	9	22
Ada	29	19	18	15	12	5	3
Lisp	30	12	16	13	8	6	2
Fortran	31	24	21	12	6	3	15

Programming languages - classifications

Different points of view

- distance to hardware/human: **assembly** ▷ **low-level** ▷ **high-level**
- problem related: Matlab (matrices, Linear Algebra) vs Fortran
- transformation to executable binaries: **compilation** (vs **semi-compilation**) vs **interpretation**
- **portability**
- **programming paradigm**: way of thinking, solving problems

Programming paradigms classification

Imperative programming

The programmer says explicitly the order in which the instructions will be executed

- **procedural programming**
- **object programming**
- parallel programming

Declarative programming

The order of execution of the instructions is not defined by the user, but by the interpreter

- logical programming (**Prolog**: "*Say what you want, not how you want it done*")
- **functional programming**
- programming by constraints

Programming in the EIE curriculum: paradigm shift

EIE1: Procedural programming (EE1-07, EE1-08)

variables, flow control, functions, procedures C++ (non oo)

EIE2: Object-Oriented programming (EE2-12)

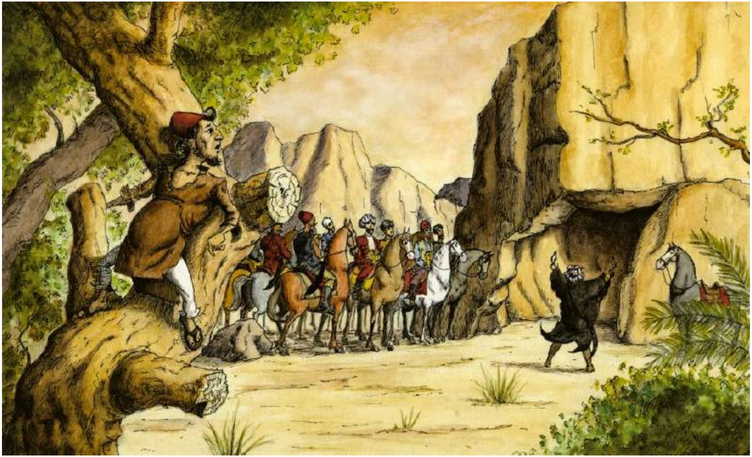
classes, objects ("*all is object*") C++ (oo)

EIE3: Functional programming (EE3-22)

programming without variables ("*all is function*") F#

Metaphor: Ali Baba & the Forty Thieves I

Problem: How to open Sesame, the cave door?



Metaphor: Ali Baba & the Forty Thieves II

Solution 1: Procedural programming style

- Sesame is a **data**, manipulated by the **user** Ali Baba.
- Ali Baba has both
  - the intention of opening the door: **WHAT** to do?
  - and the expertise (know-how) of opening the door: **HOW** to do it? (open() procedure, I grab the handle of the door; I turn this handle; and I push the door.)

1. data sesame; //door sesame;
2. open(sesame);

Metaphor: Ali Baba & the Forty Thieves III

Solution 2: Object-Oriented programming style

- Sesame is an **object**, to which we delegated the expertise
- Ali Baba has the intention of opening the door,
- But now it is the door which has the know-how!

1. data sesame; //door sesame;
2. sesame.open();

- "**Open Sesame**!", or "Sesame, open (yourself)!” (French: *Sésame, ouvre-toi*)

Another problem: $a.x^2 + b.x + c = 0$ equation

potential solutions $x_{12} = (-b + -sqrt(b^2 - 4.a.c))/(2.a)$

Procedural programming

solve_2nd_degree_equation(a, b, c);

Object programming

- equation is an object
- equation.solve();

C++ Module target programming language

"*C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.*" [Wikipedia]

- 1979 "**C with classes**" (classes, a notion coming from Simula (1962))
- 1982 "C with classes" becomes "**C++**"
- multi-paradigm, not only OO, an improved C enabling OO and generic programming

Outline

1. Introduction: Programming paradigms
2. Structures
3. Classes
4. UML notations

Structures vs Arrays

- Aggregate data types ("grouping")
    - **Array**: collection of values of same type
    - **Structure**: collection of values of different types
- Treated as a single item
- Major difference: must first "define" struct
    - Prior to declaring any variables

Structure Types

- Define struct globally (typically)
- No memory is allocated
  - Just a "placeholder" for what our struct will "look like"
- Definition:

```
struct date {
        int day;
        int month;
        int year;
};
```

```
struct bdnode {
        std::string val;
        bdnode* left;
        bdnode* right;
};
```

Declare Structure Variable

- With structure type defined, now declare variables of this new type:
  date d;
  - Just like declaring simple types
  - Variable **d** now of type **date**
  - It contains "member values"
    - Each of the struct "parts"

Accessing Structure Members

- Dot Operator to access members
  - d.day;
  - d.month;
  - d.year;
- Called "**member variables**"
  - The "parts" of the structure variable
  - Different structs can have same name member variables
    - No conflicts

Structure Pitfall

Semicolon after structure definition

- ; MUST exist:

```
struct weather_data {
       double temperature;
       double windVelocity;
}; //REQUIRED semicolon!
```

- Required since you "can" declare structure variables in this location

Warm up Exercise: point structure

1) Data only (variables)

- Write a structure point modelling a 2D cartesian point (x, y)
- Test it in a main function

2) Behaviour (functions)

- Add a function to display the state of the point
- Test it in the main function

Warm up Exercise: point structure - Answer

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

struct point_struct {
    double x;
    double y;

    void display() {
        cout << "(" << x << ",␣" << y << ")" << endl;
    }
};

int main() {
    point_struct p;
    p.x = 2.0;
    p.y = 3.0; //can also initialise at declaration p = {2.0, 3.0}

    p.display();

    p.x = 1.0;
    p.display();

    return 0;
}

/* Output:
    (2, 3)
    (1, 3)
```

Outline

1. Introduction: Programming paradigms
2. Structures
3. Classes
4. UML notations

Classes

- Similar to structures:
  - member data (variables)
  - member functions
- In C++, variables of class type are objects
- header file (.hpp): only member function's prototype
- Function's implementation is elsewhere (.cpp)

Exercise 2: point class (without main) - Answer I

Exercise 2: point class (without main) - Answer I

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

class point {
        double x;
        double y;

        void display(); //only the member function proto
};

//member function definition
void point::display() {
        cout << "(" << x << ",␣" << y << ")" << endl;
}
```

Class Member Access

- Members accessed same as structures:
    - today.month
    - today.day
- And to access member function:
    - today.output(); //Invokes member function

Class Member Functions

- Must define or "implement" class member functions
- Like other function definitions
  - Can be after main() definition
  - Must specify class:
  - void point::display() {. . . }
  - **::** is **scope resolution operator**
  - Instructs compiler "what class" member is from
  - Item before :: called **type qualifier**

Exercise 3: point class (with main) - Answer

Exercise 3: point class (with main) - Answer I

[with compilation error (private member variables)]

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

class point {
        double x;
        double y;

        void display(); //only the member function proto
};

//member function definition
void point::display() {
        cout << "(" << x << ",␣" << y << ")" << endl;
}
```

Exercise 3: point class (with main) - Answer II

```
int main() {
        point p;
        p.x = 2.0; //not allowed (private member variabl
        p.y = 3.0; //not allowed (private member variabl

        p.display(); //allowed (public member function)

        p.x = 1.0;  //not allowed
        p.display();

        return 0;
}
```

Encapsulation

- Any data type includes
  - Data (range of data)
  - Operations (that can be performed on data)
- Encapsulation means "bringing together as one"
- Binding data (member variables) & operations on the data (member functions) together
- But but keep "details" hidden

Public and Private Members

- Given previous example
- Declare object:
  point p;
- Only **public** members are accessible
  - p.x = 2.0; //not allowed (private member variable)
  - cout << p.x; //not allowed (private member variable)
  - p.display(); //allowed (public member function)

Public and Private Style

- Can mix & match public & private
- More typically place public first
    - Allows easy viewing of portions that can be USED by programmers using the class
    - Private data is "hidden", so irrelevant to users
- Outside of class definition, cannot change (or even access) private data

Accessor and Mutator Functions

- Object needs to "do something" with its data
- Accessor member functions
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data
- Mutator member functions
  - Allow object to change data
  - Manipulated based on application

Exercise 3: point class (with main) - Answer I

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

class point {
        public:
                double get_x();
                double get_y();

                void set_x(double x_in);
                void set_y(double y_in);

                void display();

        private:
                double x;
```

Exercise 3: point class (with main) - Answer II

```
                double y;
};

//member function definition
void point::display() {
        cout << "(" << x << ",␣" << y << ")" << endl;
}

double point::get_x() {
        return x;
}

double point::get_y() {
        return y;
}
```

Exercise 3: point class (with main) - Answer III

```
void point::set_x(double x_in) {
        x = x_in;
}

void point::set_y(double y_in) {
        y = y_in;
}

int main() {
        point p;
        p.set_x(2.0);
        p.set_y(3.0);

        p.display();
```

Exercise 3: point class (with main) - Answer IV

```
        p.set_x(1.0);
        p.display();

        return 0;
}

/*  Output:
        (2, 3)
        (1, 3)
*/
```

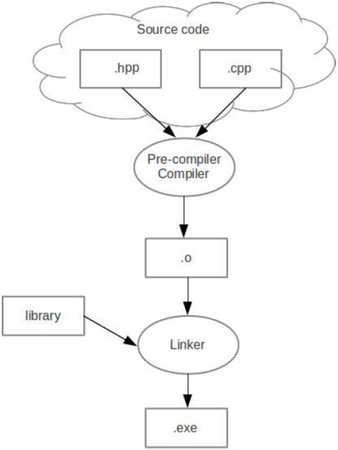
Separate Interface and Implementation

- User of class does not need to see details of how class is implemented
  - Principle of OOP ▷ **encapsulation**
- User only needs "rules"
  - Called "interface" for the class
  - In C++ public member functions and associated comments
- Implementation of class hidden
  - Member function definitions elsewhere
  - User need not see them

Structures vs Classes

- Structures
  - Typically all members public
  - No member functions, (C-like) [convention, even if allowed by the language]
- Classes
  - Typically all data members private
  - Interface member functions public

Compilation



Outline

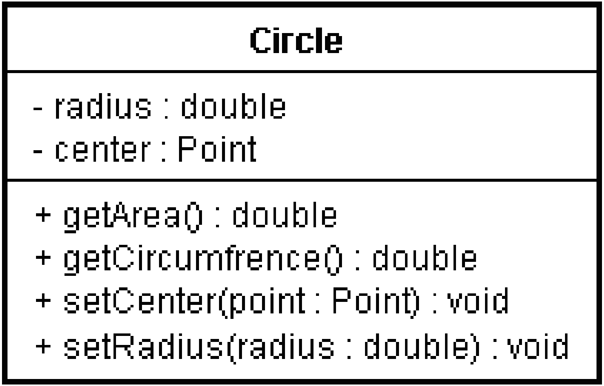
1. Introduction: Programming paradigms
2. Structures
3. Classes
4. UML notations

UML

- Unified Modelling Language
- set of graphical notations to describe a “system”, from different points of view
  - **functional**: use case diagram
  - **structural**: class diagram, component diagram, deployment diagram
  - **dynamic**: state diagram, activity diagram, sequence diagram, collaboration diagram
- [not a programming language!]
- simply drawing diagrams, with unified semantics

UML class diagram

Class name + Attributes + Operations



Circle

- radius : double
- center : Point

+ getArea() : double
+ getCircumfrence() : double
+ setCenter(point : Point) : void
+ setRadius(radius : double) : void

Vocabulary

	data	**operations**
C++	member variables	member functions
UML	attributes	operations
Java	attributes	methods

Exercise 3: class point UML diagram

Summary

- **Structure** is a collection of different types
- **Class** used to combine data and functions into single unit ▷ **object**
- **Member variables** and **member functions**
  - Can be **public**: accessed outside class
  - Can be **private**: accessed only in a member function's definition
- C++ class definition: should separate two key parts
  - **Interface**: what user needs
  - **Implementation**: details of how class works
- OOP ~= "*writing classes*"

What to do next?

Next Lab

- more functions and tests on class point
- Makefile

Next Lecture

Week 2 - Classes and Objects II: Constructors, and Operator Overloading