

# Programmation orientée objet en C++ :

## - Abstraction et encapsulation -

Groupe étudiant : CIR2

Nils Beaussé

E-Mail : [nils.beausse@isen-ouest.yncrea.fr](mailto:nils.beausse@isen-ouest.yncrea.fr)

Numéro de bureau : A2-78

N° de téléphone : 0230130573



# Introduction à l'abstraction et l'encapsulation

# Types de programmation

Programmation procédurale/impérative (1957 - ...)



...



Programmation orientée objet -POO- (1960 - ...)



...



# Types de programmation

Programmation procédurale/impérative (1957 - ...)



...



Programmation orientée objet -POO- (1960 - ...)



...



Qu'apporte la POO ?

# Types de programmation

Programmation procédurale/impérative (1957 - ...)



...



- **Abstraction**
- **Encapsulation**
- Héritage
- Polymorphisme

Programmation orientée objet -POO- (1960 - ...)



...

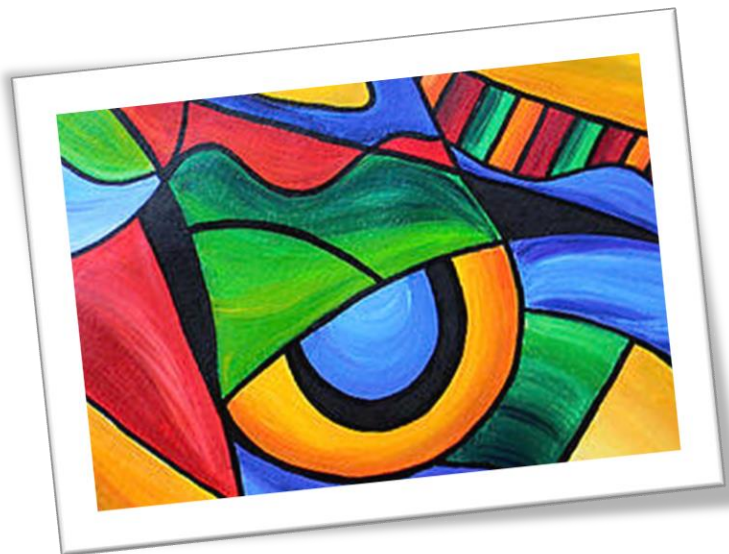


## Qu'apporte la POO ?

# L'abstraction



# L'abstraction



## 1. Introduction

# Abstraction : Intro



Chaque mot échangé dans une conversation est une abstraction, une image de la réalité !



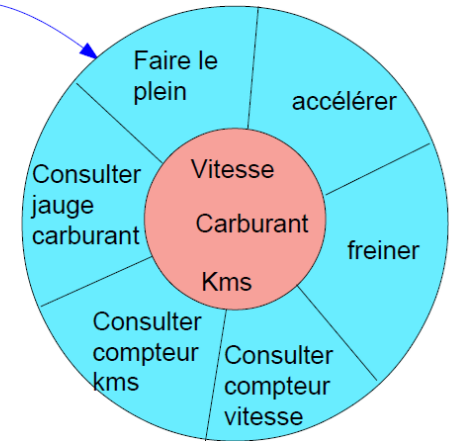
# Abstraction : Intro

- L'objet est une représentation simplifiée (une **abstraction**) d'une entité du monde réel
- L'**abstraction** permet de :
  - manipuler les données à un plus haut niveau
  - conserver que les caractéristiques jugées pertinentes

Entité réelle



Abstraction Objet

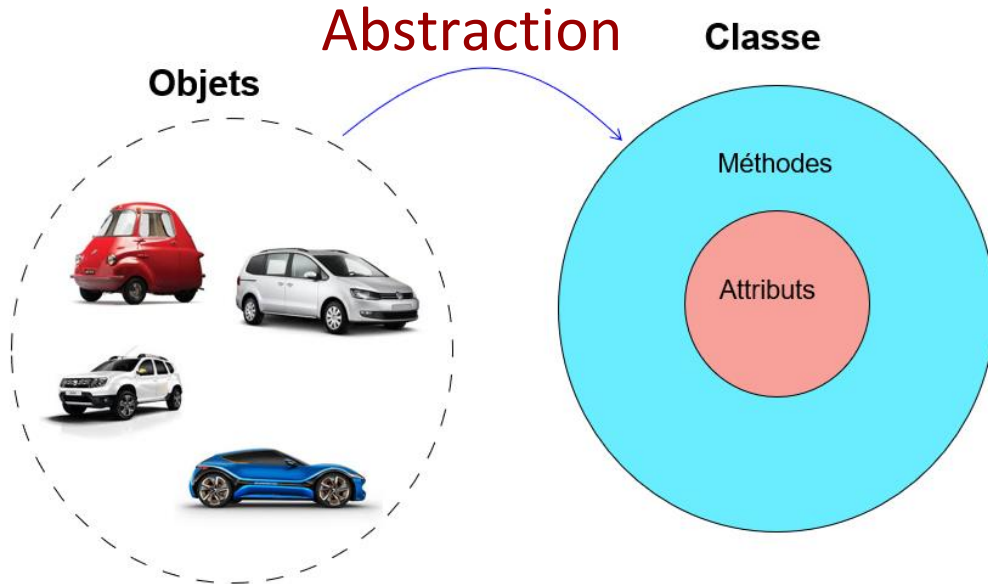


État (attributs)



Comportement (méthodes)

# Abstraction : Intro

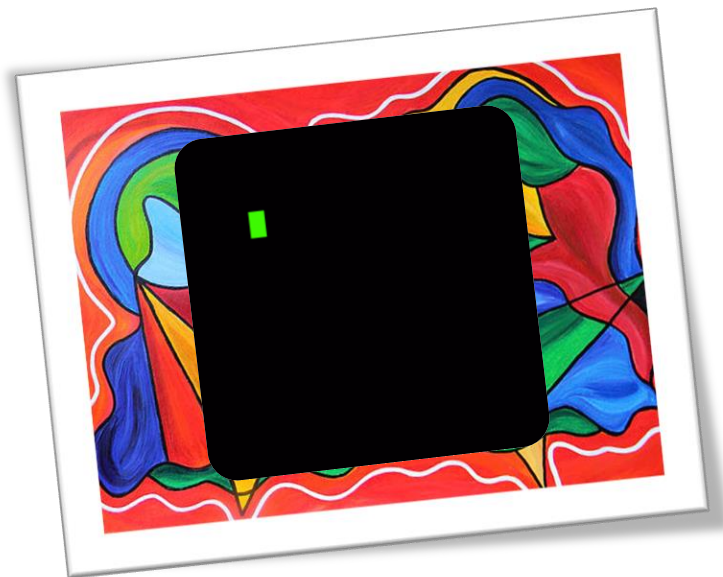


- Permet de manipuler par un nom, ici «voiture» un objet complexe.
- Modélisation d'un ensemble d'objets de même nature
- Factorisation des propriétés communes à ces objets

# L'abstraction



# L'abstraction



## 2. En programmation C

# Abstraction : en C

- Calcul de la note d'un semestre (pour une matière donnée) à partir des notes de devoir et d'examen :

- Solution 1

```
#include <iostream>
double noteS3(double devoir, double exam);
int main(){
    double devoir=15;
    double exam=14;
    std::cout << "la note de C++ en S3 est : "
        << noteS3(devoir,exam);
    return 0;
}
double noteS3(double devoir, double exam){
    return 0.5*devoir+0.5*exam;
}
```

Adresse	Contenu
@2_n	devoir
...	
@2_1	
@1_n	exam
...	
@1_1	

# Abstraction : en C

- Calcul de la note d'un semestre (pour une matière donnée) à partir des notes de devoir et d'examen :

- Solution 2**

```
#include <iostream>
double noteS3(double notes[]);
int main(){
    double notes[2];
    notes[0]=15;
    notes[1]=14;
    std::cout << "la note de C++ en S3 est : "
               << noteS3(notes);
    return 0;
}
double noteS3(double notes[]){
    return 0.5*notes[0]+0.5*notes[1];
}
```

Adresse	Contenu
@2_n	notes[1]
...	
@2_1	
@1_n	notes[0]
...	
@1_1	

# Abstraction : en C

- Solution 3 : utilisation des structures de données

```
#include <iostream>
typedef struct{
    double devoir;
    double exam;
} Notes;

double noteS3(Notes notes);
int main(){
    Notes notes;
    notes.devoir = 15;
    notes.exam = 14;
    std::cout << "la note de C++ en S3 est : "
                << noteS3(notes);
    return 0;
}
double noteS3(Notes notes){
    return 0.5*notes.devoir+0.5*notes.exam;
}
```

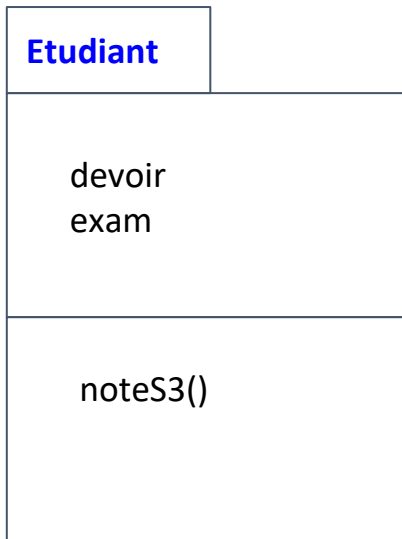
Adresse	Contenu
@2_n	notes.devoir
...	
@2_1	
@1_n	notes.exam
...	
@1_1	

# POO : Abstraction

- Plus d'abstraction pourrait être pratique ! Par exemple ici ne manipuler qu'un « étudiant » dont on aurait les notes ET la fonction de calcul de note.

```
#include <iostream>
double noteS3(double devoir, double exam);
int main(){
    double devoir=15;
    double exam=14;
    std::cout << "la note de C++ en S3 est : "
        << noteS3(devoir,exam);
    return 0;
}
double noteS3(double devoir, double exam){
    return 0.5*devoir+0.5*exam;
}
```

Diagram illustrating the abstraction of the `noteS3` function into an **Etudiant** object. A blue arrow points from the `noteS3` function call in the code to the **Etudiant** object, which contains the `devoir` and `exam` attributes and the `noteS3()` method.



Adresse	Contenu
@1_m	Etudiant e1;
...	
...	
@1_1	



# L'Encapsulation

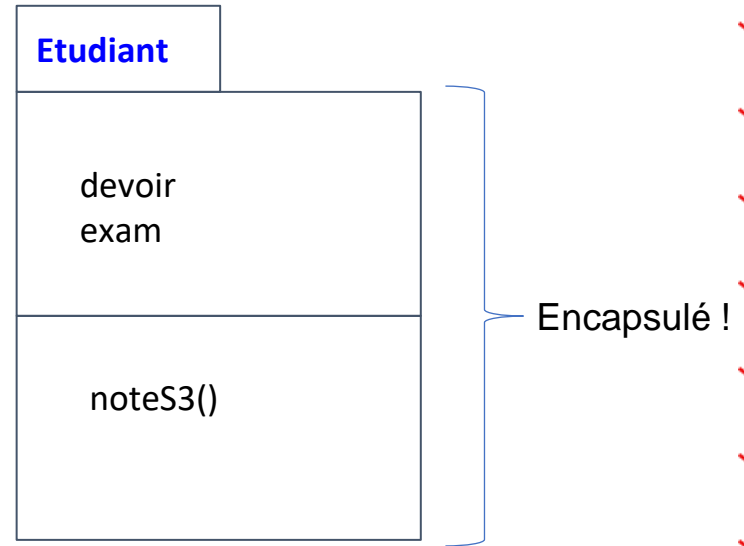


## 1. Introduction

# L'Encapsulation

**Encapsulation** = regroupement des données ou/et des traitements dans un même objet


**L'encapsulation** peut être vue comme une conséquence de l'abstraction (on encapsule la réalité, des variables ici, dans un concept abstrait, le nom du groupe ici (etudiant))



# L'Encapsulation en C

```
#include <iostream>
typedef struct{
    double devoir;
    double exam;
} Notes;

double noteS3(Notes notes);
int main(){
    Notes notes;
    notes.devoir = 15;
    notes.exam = 14;
    std::cout << "la note de C++ en S3 est : "
                << noteS3(notes);
    return 0;
}
double noteS3(Notes notes){
    return 0.5*notes.devoir+0.5*notes.exam;
}
```



La structure encapsule  
deux variables : la note des  
devoir et la note des exams.

# L'Encapsulation en C

```
#include <iostream>
typedef struct{
    double devoir;
    double exam;
} Notes;

double noteS3(Notes notes);
int main(){
    Notes notes;
    notes.devoir = 15;
    notes.exam = 14;
    std::cout << "la note de C++ en S3 est : "
               << noteS3(notes);
    return 0;
}
double noteS3(Notes notes){
    return 0.5*notes.devoir+0.5*notes.exam;
}
```

La structure encapsule deux variables : la note des devoir et la note des exams.

Par défaut en C standard la fonction qui permet de manipuler la structure n'est pas dans la structure.

# L'Encapsulation en C

- Pourtant il peut être pratique d'encapsuler la fonction avec les variables.
- Par exemple :

```
#include <iostream>
typedef struct{
    int humidite;
    int taille;
} Plante;

double arroser_plante(Plante ma_plante){
    ma_plante.humidite += 10;
}
```

Ici la fonction « arroser\_plante » ne sera appliquer qu'à des plantes. Ce serait quand même plus intuitif si la fonction était liée à la structure !

# L'Encapsulation en C

- C'est possible de le faire en C, mais c'est moche :
- Par exemple :

```
#include <iostream>
double arroser_plante(Plante ma_plante);

typedef struct{
    int humidite;
    int taille;
    (double*)ma_function_arroser_plantes;
} Plante;

double arroser_plante(Plante ma_plante){
    ma_plante.humidite += 10;
}
```



Ça marche si on initialise  
ma\_function\_arroser\_plantes =  
&arroser\_plante dans le main.  
Mais c'est moche et pas pratique, et  
la fonction arroser\_plante reste  
appelable « normalement », ce qui  
peut donner des erreurs d'inattention  
et de la confusion.

# Pourquoi abstraire/encapsuler ?

1. Une meilleure
  - a. visibilité
  - b. cohérence
  - c. modularité

## Programmation procédurale

```
double devoir1=15; etd1
double exam1=16;
```

```
double devoir2=10; etd2
double exam2=9;
```

```
double note1=noteS3(devoir1,exam1);
double note2=noteS3(devoir2,exam2);
```



## Programmation orientée objet

```
Etudiant etd1(15,16);
Etudiant etd2(10,9);
double note1=etd1.noteS3();
double note2=etd2.noteS3();
```

# L'encapsulation



## 2. Le concept de masquage

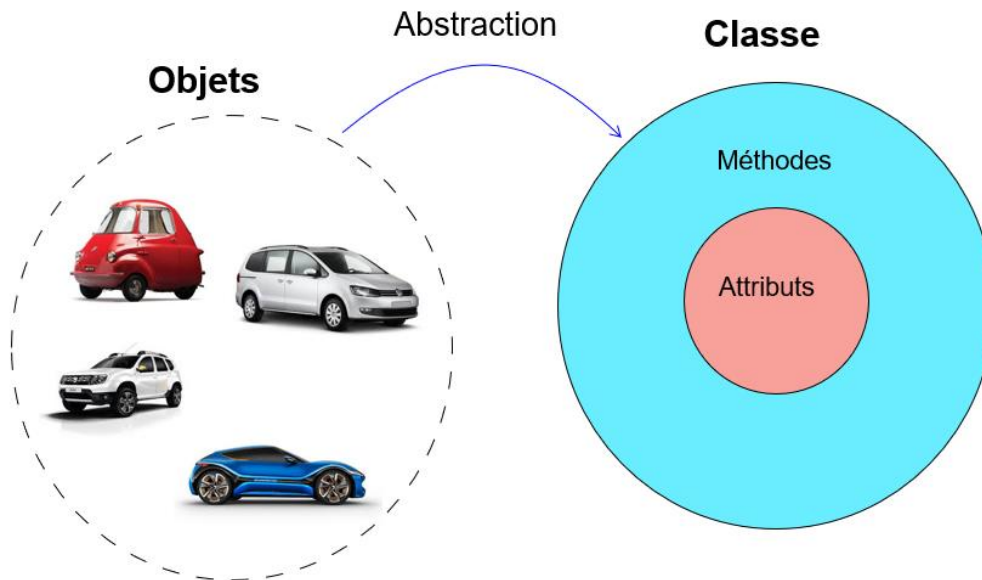


# L'encapsulation

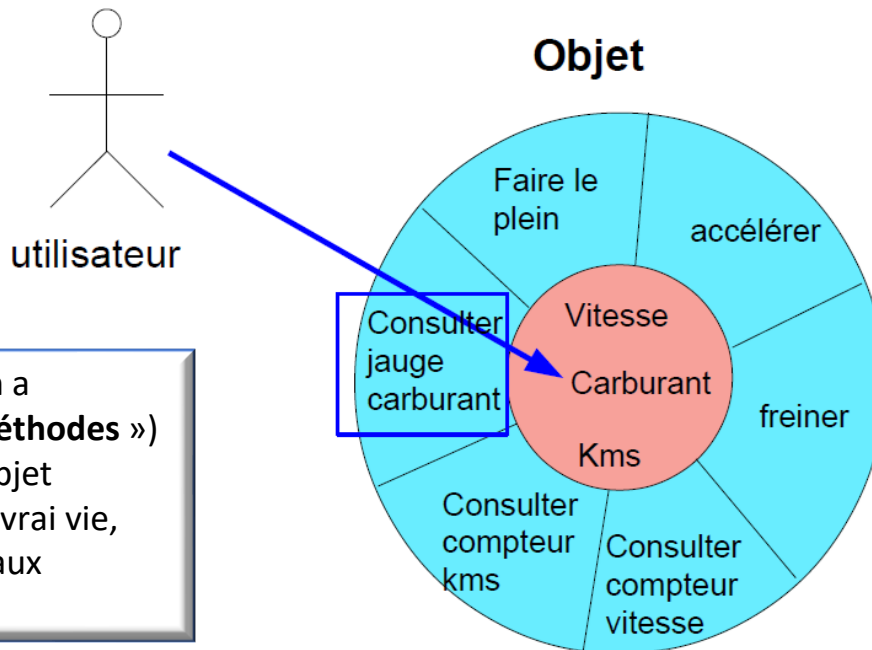


## 2. Le concept de masquage

# Le masquage

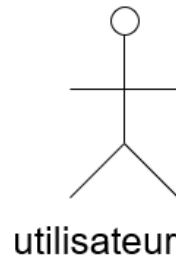


# Le masquage



On a abstrait un objet et on a encapsulé des actions (« **méthodes** ») et des paramètres liés à l'objet (« **attributs** ») mais dans la vrai vie, peut-on vraiment accéder aux attributs directement ?

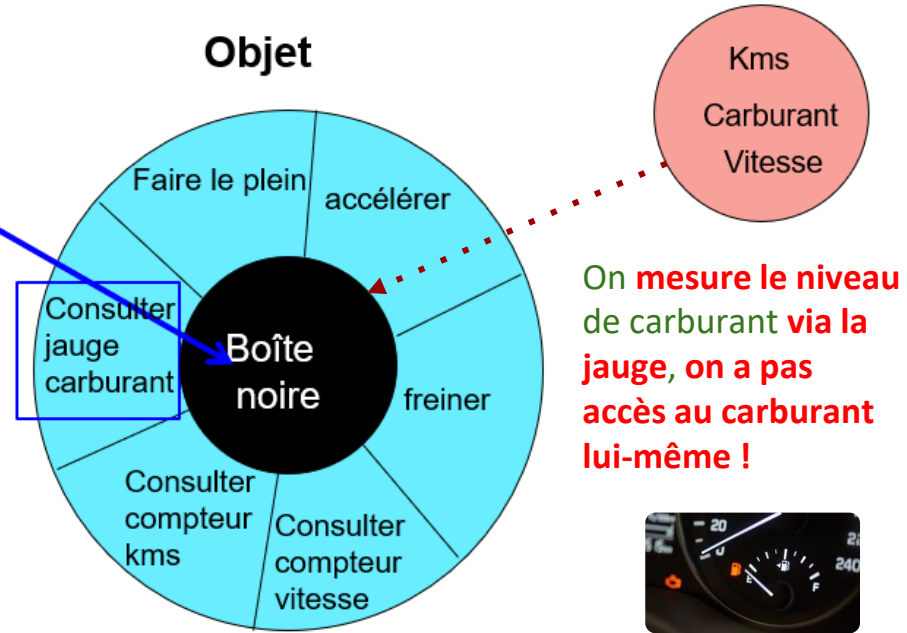
# POO : Encapsulation



## Masquage :

Dans cet exemple :

- L'état interne de l'objet (**les attributs**) est en fait caché (**boîte noire**)
- L'objet est manipulé par le biais des éléments encapsulés à l'extérieur du cercle et qui permettent de manipuler les attributs (« consulter la jauge », on parle de « **méthodes** »)



# Peut-on masquer en C ?

```
#include <iostream>
typedef struct{
    int humidite;
    int taille;
} Plante;

int lecture_humidite(Plante ma_plante){
    return ma_plante.humidite;
}
```

# Peut-on masquer en C ?

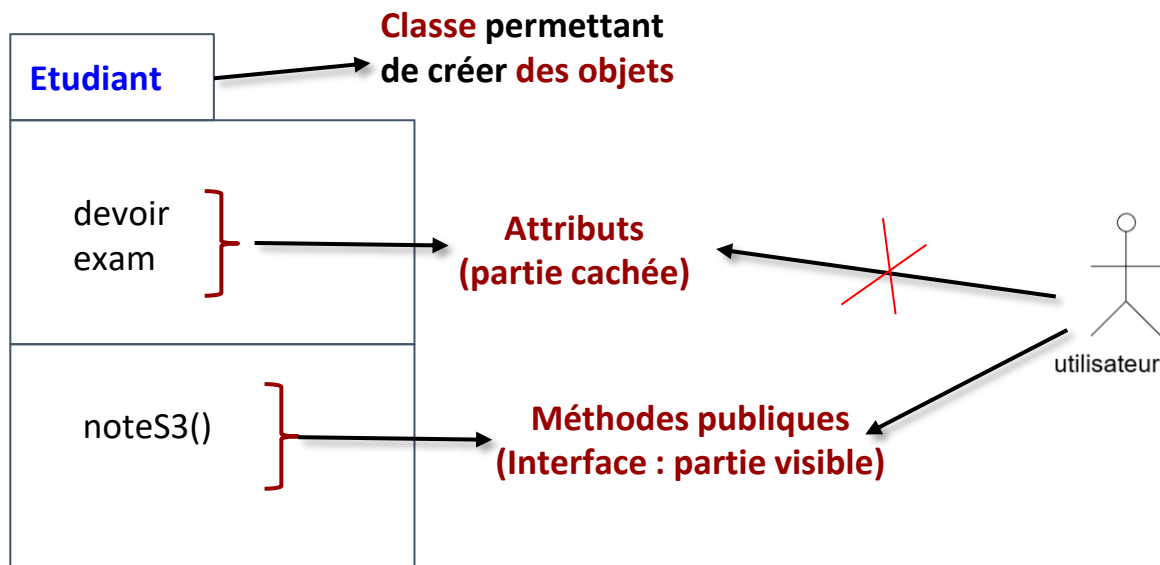
```
#include <iostream>
typedef struct{
    int humidite;
    int taille;
} Plante;

int lecture_humidite(Plante ma_plante){
    return ma_plante.humidite;
}
```

**Non !** Car il n'y a aucun moyen de rendre « *humidite* » et « *taille* » inaccessible depuis le programme principal tout en les laissant accessible à la fonction « *lecture\_humidite* »

# À quoi ça sert le masquage ?

Ça fournit un cadre rigoureux à l'utilisation des objets : les attributs sont cachés → pas de risque de modification de la structure de la classe depuis l'extérieur **n'importe comment**.



# À quoi ça sert le masquage ?

```
#include <iostream>
typedef struct{
    double devoir;
    double exam;
} Notes;

double noteS3(Notes notes);
int main(){
    Notes notes;
    notes.devoir = -5;
    notes.exam = 14;
    std::cout << "la note du devoir est" << devoir << endl;
    return 0;
}
```

Oups, j'ai commis une erreur !



# À quoi ça sert le masquage ?

```
#include <iostream>
typedef struct{
    double devoir;
    double exam;
} Notes;

void input_note(Notes notes, double la_note)
{
    if (la_note < 0 || la_note > 20)
        std::cout << "ERREUR !!!" << endl;
    else
        notes.devoir = notes;
}

int main(){
    Notes notes;
    notes.devoir = input_note(notes, -5);
    notes.exam = 14;
    std::cout << "la note du devoir est" << notes.devoir <<
endl;
    return 0;
}
```

L'erreur est détectée !

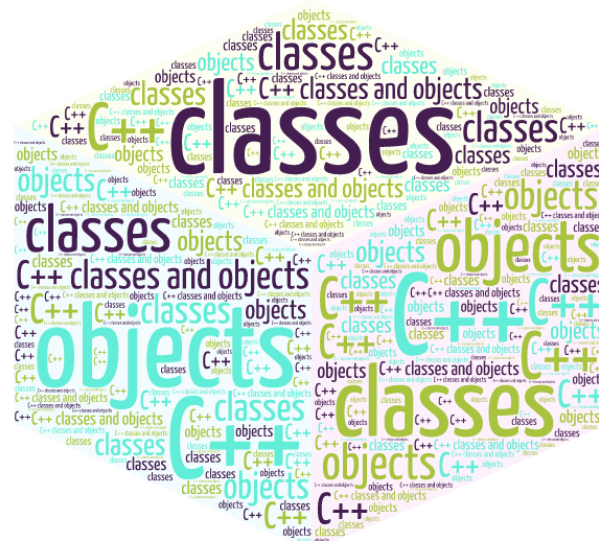
Le masquage permet de s'assurer que dès qu'on touche à une donnée il y ait des sécurités !

# Du C au C++

- L'**encapsulation** et le **masquage** sont très pratique pour des raisons de **sécurité** et pour construire des concepts plus simples à manipuler, des concepts **abstraits** **qui allègent le code**.
- Le C n'est pas très adapté pour tout ceci, les structures ne permettent pas de tout faire.
- Le C++ étends les structures du C en introduisant de nombreuses choses en plus !

# L'abstraction et l'encapsulation en C++

# Les classes



## 1. Introduction : Les classes, extensions des structures

# Qu'est ce qu'une classe en C++ ?

- Une classe en C++ fonctionne au départ comme une structure :

```
#include <iostream>
typedef struct notes{
    double devoir;
    double exam;
} Notes;

int main(){
    Notes notes;
    notes.exam = 14;

    std::cout << "la note de l'exam est" <<
notes.exam << endl;

    return 0;
}
```



```
#include <iostream>
class Notes{
    double devoir;
    double exam;
};

int main(){
    Notes notes;
    notes.exam = 14;

    std::cout << "la note de l'exam est" <<
notes.exam << endl;

    return 0;
}
```

Plus besoin de « typedef » ! C'est plus simple avec les classes !

# Qu'est ce qu'une classe en C++ ?

Là où c'était compliqué en C, une classe peut tout à fait contenir une fonction en C++



On peut alors appeler la fonction (la méthode) comme on appelle les variables d'une structures (les attributs) **avec un simple « . »**



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

class Notes{
    double devoir;
    double exam;
    double notes3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;
    notes.devoir = 4;

    cout << "la note finale est" << notes.notes3() << endl;

    return 0;
}
```

# Les valeurs par défauts

On peut tout à fait définir des arguments par défauts de façon très simple



Par exemple ici on ne définit pas « **devoir** » mais le calcul dans **noteS3()** prendra la valeur 2 comme valeur par défaut pour devoir.



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

class Notes{
    double devoir = 2;
    double exam = 4;
    double noteS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;

    cout << "la note finale est" << notes.noteS3() << endl;

    return 0;
}
```

# Petit résumé :

- **Déclaration d'une classe :**

- Syntaxe :
  - `class Nom_classe { ... };`
- Exemple :
  - `class Etudiant { ... };`

- **Instanciation d'une classe = création des objets :**

- **Allocation statique :**

- Syntaxe :
 

```
Nom_classe nom_instance;
```
- Exemple :
 

```
Etudiant etd1;
```

- **Allocation dynamique :**

- Syntaxe :
 

```
Nom_classe* nom_instance=new Nom_classe();
```
- Exemple :
 

```
Etudiant* etd1=new Etudiant();
```



# Petit résumé :

- **Déclaration d'une classe :**

- **Syntaxe :**

- `class Nom_classe { ... };`

- **Exemple :**

- `class Etudiant { ... };`

- **Instanciation d'une classe = création des objets :**

- **Allocation statique :**

- **Syntaxe :**

- `Nom_classe nom_instance;`

- **Exemple :**

- `Etudiant etd1;`

- **Allocation dynamique :**

- **Syntaxe :**

- `Nom_classe* nom_instance=new Nom_classe();`

- **Exemple :**

- `Etudiant* etd1=new Etudiant();`

Avec les classes **on utilise plutôt la façon de faire du C++** (le « **new** », plutôt que la façon de faire du C (le « **malloc** »). **Malloc** n'est pas interdit mais **il y a une bonne raison à cela qu'on expliquera ensuite !**



# Petit résumé :

- **Déclaration d'une classe :**

- **Syntaxe :**

- `class Nom_classe { ... };`

- **Exemple :**

- `class Etudiant { ... };`

Ne pas oublier  
d'appeler delete  
après utilisation !

Avec les classes **on utilise plutôt la façon de faire du C++** (le « **new** », plutôt que la façon de faire du C (le « **malloc** »). **Malloc** n'est pas interdit mais **il y a une bonne raison à cela qu'on expliquera ensuite !**

- **Instanciation d'une classe = création des objets :**

- **Allocation statique :**

- **Syntaxe :**

- `Nom_classe nom_instance;`

- **Exemple :**

- `Etudiant etd1;`

- **Allocation dynamique :**

- **Syntaxe :**

- `Nom_classe* nom_instance=new Nom_classe();`

- **Exemple :**

- `Etudiant* etd1=new Etudiant();`

# Petit résumé :

- **Déclaration des attributs :**

- Syntaxe :
  - `type nom_attribut;`
- Exemple :
  - `double devoir1;`

- **Accès aux attributs :**

- **Allocation statique :**

- Syntaxe :  
`nom_instance.nom_attribut;`
- Exemple :  
`etd1.devoir1;`

- **Si on a affaire à des pointeurs :**

- Syntaxe :  
`nom_instance->nom_attribut;`
- Exemple :  
`etd1->devoir1;`

## • Déclaration des attributs :

- Syntaxe :
  - `type nom_attribut;`
- Exemple :
  - `double devoir1;`

## • Accès aux attributs :

### • Allocation statique :

- Syntaxe :
  - `nom_instance.nom_attribut;`
- Exemple :
  - `etd1.devoir1;`

### Rappel :

Si « type » est le type d'une variable alors « type\* » est un pointeur vers une variable de ce type.

```
type A; // je déclare A
type* B; // je déclare un pointeur vide qui ne pointe sur rien
B = &A; // je le fait pointer sur A
*B = 5 // permet de travailler sur la valeur pointée (ici A).
```

Si A est une structure, on accède aux membres de A en faisant :  
*A.truc* ou *(\*B).truc* ou *B->truc* .

### • Si on a affaire à des pointeurs :

- Syntaxe :
  - `nom_instance->nom_attribut;`
- Exemple :
  - `etd1->devoir1;`



# Petit résumé :

- **Déclaration d'une méthode :**

- **Syntaxe :**

- `type_retour nom_methode(type_param1 nom_param1, ...)`  
`{`  
`// corps de la méthode`  
`}`

- **Exemple :**

- `double noteS3(double devoir, double exam)`  
`{`  
`return 0.5*devoir+0.5*exam;`  
`}`

- **Accès à une méthode :**

- **Classique :**

- **Syntaxe :**

- `nom_instance.nom_method(val_arg1, ...);`

- **Exemple :**

- `etd1.noteS3(15,14);`

- **Si on a affaire à des pointeurs :**

- **Syntaxe :**

- `nom_instance->nom_method(val_arg1, ...);`

- **Exemple :**

- `etd1->noteS3(15,14);`

# Portée des attributs et méthodes

- **Notion de portée de classe** : Les attributs d'une classe sont accessibles dans toutes les méthodes de la classe !!!

- Syntaxe :

```
type nom_attribut;
type_retour nom_methode(){
    // Utilisation des attributs
}
```

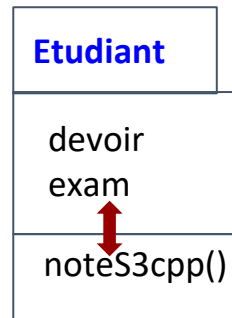
- Exemple :

```
double devoir1;
double exam;
double noteS3(){
    return 0.5*devoir1+0.5*exam; }
```

# Portée des attributs et méthodes

- Les méthodes sont :
  - des fonctions propres à la classe
  - qui ont accès aux attributs de classe
  - avec ou sans arguments

```
double devoir1;
double exam;
double noteS3(){
    return 0.5*devoir+0.5*exam;
}
```



# Déclaration des méthodes à l'extérieur de la classe

- Objectif :
  - une meilleur visibilité du code
  - modularité

- Syntaxe :

```
type_retour Nomclasse::nom_methode(type_param1 nom_param1,...)
{
    // corps de la méthode
}
```

- Exemple :

```
Class Etudiant
{
    double devoir1;
    double exam;
    double noteS3(); //prototype
};
double Etudiant::noteS3()
{
    return 0.5*devoir+0.5*exam;
}
```



# Déclaration des méthodes à l'extérieur de la classe

- Objectif :
  - une meilleur visibilité du code
  - modularité

- Syntaxe :

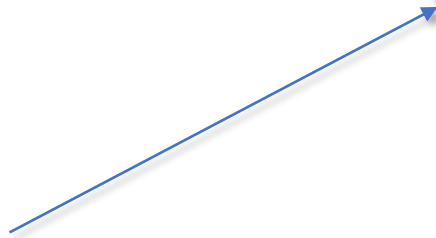
```
type_retour Nomclasse::nom_methode(type_param1 nom_param1,...)
{
    // corps de la méthode
}
```

- Exemple :

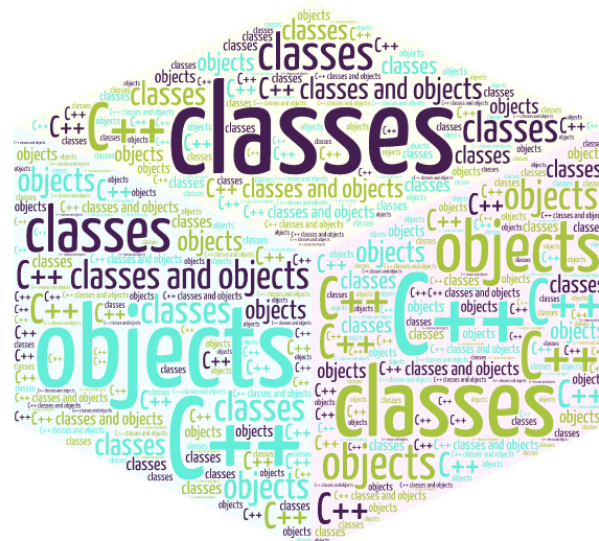
```
Class Etudiant
{
    double devoir1;
    double exam;
    double noteS3(); //prototype
};
double Etudiant::noteS3()
{
    return 0.5*devoir+0.5*exam;
}
```

Pratique si on veut mettre la classe dans un « header » (.h en C .hpp en C++) et la déclaration de la méthode dans un .cpp.

**Permet de partager les classes entres plusieurs fichiers !**



# Les classes



[illegible]

## A series of ten parallel red diagonal lines slanting downwards from left to right, located at the bottom of the page.

# Les classes et le masquage

**Reprenons  
un exemple  
précédent :**

# Les classes et le masquage

Reprenons  
un exemple  
précédent :

```
//...
class Etudiant{
    double devoir; } attributs
    double exam;
    double noteS3(){return 0.5*devoir+0.5*exam;} → méthode
};
int main(){
    Etudiant etd1;
    etd1.devoir=15;
    etd1.examen=16;
    std::cout << "la note de C++ en S3 de l'Etudiant etd1 est : "
    << etd1.noteS3();

    Etudiant etd2;
    etd2.devoir=14;
    etd2.examen=13;
    std::cout << "la note de C++ en S3 de l'Etudiant etd2 est : "
    << etd2.noteS3();
}
```

définition de la classe

# Les classes et le masquage

Reprenons  
un exemple  
précédent :

```
//...
class Etudiant{
    double devoir: 1 attributs
    dou
    dou
};
int m
E
e
e
st
E
e
    etd2.exam=13;
    std::cout << "la note de C++ en S3 de l'Etudiant etd2 est : "
        << etd2.noteS3();
}
```

définition  
lasse

Ce code ne fonctionne pas !!!!

# Les classes et le masquage

Reprenons  
un exemple  
précédent :

```
//...
class Etudiant{
    double devoir:1 attributs
    dou
    dou
};
int n
    B
    e
    e
    st
    B
    e
    e
    etd2.exam=13;
    std::cout << "la note de C++ en S3 de l'Etudiant etd2 est : "
        << etd2.noteS3();
}
```

définition  
classe

**Ce code ne fonctionne pas !!!!**

Erreur de compilation :

- 1-Etudiant::devoir is private within this context
- 2-Etudiant::exam is private within this context
- 3-Etudiant::noteS3() is private within this context

# Les classes et le masquage

Reprenons  
un exemple  
précédent :

```
//...
class Etudiant{
    double devoir:1 attributs
    dou
    dou
};
int n
    B
    e
    e
    st
    B
    e
    e
    etd2.exam=13;
    std::cout << "la note de C++ en S3 de l'Etudiant etd2 est : "
        << etd2.noteS3();
}
```

définition  
classe

**Ce code ne fonctionne pas !!!!**

Erreur de compilation :

- 1-Etudiant::devoir is private within this context
- 2-Etudiant::exam is private within this context
- 3-Etudiant::noteS3() is private within this context

Pourquoi ?



# Les classes et le masquage

Reprenons  
un exemple  
précédent :

```
//...
class Etudiant{
    double devoir:1 attributs
    dou
    dou
    dou
};
int n
    B
    e
    e
    st
    B
    e
    e
    etd2.exam=13;
    std::cout << "la note de C++ en S3 de l'Etudiant etd2 est : "
        << etd2.noteS3();
}
```

définition  
classe

**Ce code ne fonctionne pas !!!!**

Erreur de compilation :

- 1-Etudiant::devoir is private within this context
- 2-Etudiant::exam is private within this context
- 3-Etudiant::noteS3() is private within this context

Pourquoi ?



Parce que  
depuis le  
début je vous  
ai menti !



# Les classes et le masquage

- **Souvenons nous** : le masquage c'est le concept de rendre inaccessible depuis l'extérieur certains éléments tout en les laissant accessible à d'autres pour respecter le concept de **l'encapsulation**.
  - **Par exemple** : on aimerait rendre inaccessible les variables d'une structure tout en les laissant accessible aux fonctions membre de cette structure.

# Les classes et le masquage

- **Souvenons nous** : le masquage c'est le concept de rendre inaccessible depuis l'extérieur certains éléments tout en les laissant accessible à d'autres pour respecter le concept de **l'encapsulation**.
  - **Par exemple** : on aimerait rendre inaccessible les variables d'une structure tout en les laissant accessible aux fonctions membre de cette structure.
- En **C** c'est **impossible**, en **C++** c'est le **comportement par défaut** des classes!

# Les classes et le masquage

- On introduit deux mot clé en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

# Les classes et le masquage

- On introduit deux mot clé en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

```
class Notes{
    double devoir;
    double exam;
    double noteS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.examen = 14;
    notes.devoir = 4;
    noteS3();
    return 0;
}
```

# Les classes et le masquage

- On introduit deux mots clés en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

```
class Notes{
    double devoir;
    double exam;
    double noteS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;
    notes.devoir = 4;
    noteS3();
    return 0;
}
```

Par défaut en C++ les **attributs**  
(les variables) sont **privés** !

# Les classes et le masquage

- On introduit deux mots clés en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

```
class Notes{
    double devoir;
    double exam;
    double notesS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;
    notes.devoir = 4;
    notesS3();
    return 0;
}
```

Par défaut en C++ les **attributs**  
(les variables) sont **privés** !

Ceci déclenche **une erreur à la compilation** !

# Les classes et le masquage

- On introduit deux mots clés en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

```
class Notes{
    double devoir;
    double exam;
    double noteS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;
    notes.devoir = 4;
    noteS3();
    return 0;
}
```

Par défaut en C++ les **attributs**  
(les variables) sont **privés** !

Par défaut en C++ les **méthodes**  
(les fonctions) sont **publiques** !

Ceci déclenche **une erreur à la compilation** !



# Les classes et le masquage

- On introduit deux mots clés en C++ pour en parler, on dit qu'un élément est privé (**private**) quand il n'est pas accessible depuis l'extérieur d'une classe, et qu'il est public (**public**) quand il est accessible.

```
class Notes{
    double devoir;
    double exam;
    double notesS3(){return 0.5*devoir+0.5*exam;}
};

int main(){
    Notes notes;
    notes.exam = 14;
    notes.devoir = 4;
    notesS3();
    return 0;
}
```

Par défaut en C++ les **attributs**  
(les variables) sont **privés** !

Par défaut en C++ les **méthodes**  
(les fonctions) sont **publiques** !

Ceci déclenche **une erreur à la compilation** !

Ceci est **OK** !

# Les classes et le masquage

- Il est possible de changer le comportement par défaut grâce à deux mots clés, les mots clés « **private** » et « **public** » :

- **Privée :** ← accessible depuis les méthodes de la classe uniquement = inaccessible depuis l'extérieur de la classe

- **Publique :** ← accessible depuis l'intérieur et l'extérieur de la classe

```
class Etudiant
{
    private:
        double devoir;
        double exam;
    public:
        double notes3(){return 0.5*devoir+0.5*exam;}
};
```

# Les classes et le masquage

- Il est possible de changer le comportement par défaut grâce à deux mots clés, les mots clés « **private** » et « **public** » :

**- Privée :** ←  
accessible depuis les méthodes de la classe uniquement = inaccessible depuis l'extérieur de la classe

**- Publique :** ←  
accessible depuis l'intérieur et l'extérieur de la classe

```
class Etudiant
{
    private:
        double devoir;
        double exam;
    public:
        double notes3(){return 0.5*devoir+0.5*exam;}
        double note_finale;
};
```

Rien ne m'interdit de mettre une variable en public explicitement !

# Les classes et le masquage

- Il est possible de changer le comportement par défaut grâce à deux mots clés, les mots clés « **private** » et « **public** » :

**- Privée :**  
accessible depuis les méthodes de la classe uniquement = inaccessible depuis l'extérieur de la classe

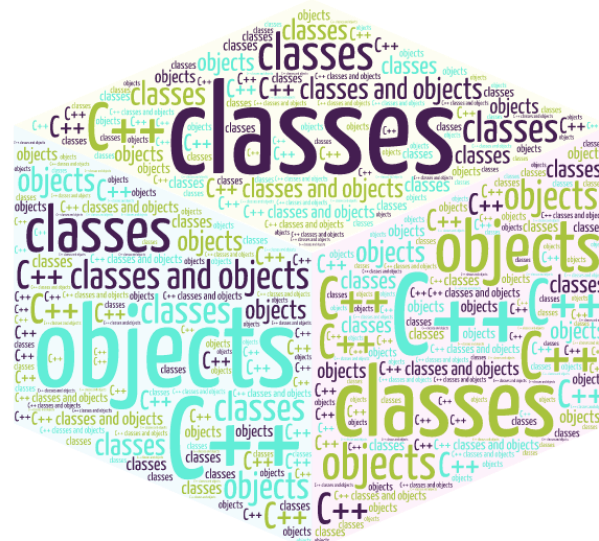
**- Publique :**  
accessible depuis l'intérieur et l'extérieur de la classe

```
class Etudiant
{
private:
    double devoir;
    double exam;
    double calcul_cache();
public:
    double notes3(){return 0.5*devoir+0.5*exam;}
    double note_finale;
};
```

Rien ne m'interdit de mettre une méthode en privé explicitement !

Rien ne m'interdit de mettre une variable en public explicitement !

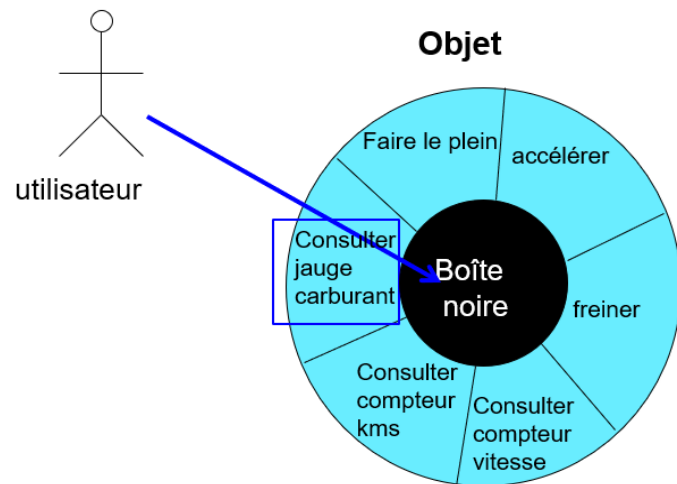
# Les classes



## 3. L'encapsulation appliqué aux classes grâce au masquage. Accesseur et mutateur !

# POO : Encapsulation (visibilités des membres)

- Règle :
  - les informations qu'il n'est pas nécessaire de connaître à l'extérieur d'un objet doivent être privées en utilisant le mot clé private
  - dans la plupart des cas :
    - Déclarer comme privé :
      - tous les attributs
      - la plupart des méthodes
    - Déclarer comme publique :
      - quelques méthodes bien choisies (interface)



# POO : Encapsulation (visibilités des membres)

- **Problématique** : Si les attributs sont tous **privés**, comment accéder aux attributs depuis **l'extérieur** de la classe ?

-> Par exemple : Comment manipuler les notes de devoir et d'exam d'un Etudiant ?

```
Etudiant etd1;  
etd1.devoir=15;  
etd1.exam=16;
```



- Solution : Utilisation des accesseurs et manipulateurs

# Accesseurs et Manipulateurs

- **Accesseurs (getters)**

- Consultation (prédicat) → on ne modifie pas l'objet
- retour de la valeur d'une variable d'instance précise

```
double getDevoir() {return devoir;}
double getExam() {return exam;}
```

ou

```
double getdevoir() const {return devoir;}
double getexam() const {return exam;}
```



# Accesseurs et Manipulateurs

- **Manipulateurs = mutateurs (setters)**
  - Modification (action) de l'objet

```
void setDevoir(double d) {devoir = d;}  
void setExam(double e) {exam = e;}
```

# Accesseurs et Manipulateurs

- **Manipulateurs = mutateurs (setters)**
  - Modification (action) de l'objet

Mais pourquoi ne pas définir les attributs publics dès le début ?

```
void setDevoir(double d) {devoir = d;}  
void setExam(double e) {exam = e;}
```

# Accesseurs et Manipulateurs

- **Manipulateurs = mutateurs (setters)**
  - Modification (action) de l'objet

Mais pourquoi ne pas définir les attributs publics dès le début ?

```
void setDevoir(double d) {devoir = d;}
void setExam(double e) {exam = e;}
```

Supposant qu'un enseignant a entré une valeur négative d'une note.

→ Il est nécessaire de faire le test sur les notes avant de calculer la note finale

```
Etudiant etd1;
etd1.devoir = -3;
etd1.exam = -9;
```

```
class Etudiant
public:
    double devoir;
    double exam;
    double noteS3(){return 0.5*devoir+0.5*exam;}
};
```

# Accesseurs et Manipulateurs

- **Manipulateurs = mutateurs (setters)**

→ Il est nécessaire de faire le test sur les notes avant de calculer la note finale

```
Etudiant etd1;
etd1.setDevoir(-3);
etd1.setExam(-9);
```

```
class Etudiant
private:
    double devoir;
    double exam;
public:
    double noteS3(){return 0.5*devoir+0.5*exam;}
    void setDevoir(double d) {
        if(d<0){std::cout<<"erreur";}
        else{devoir = d;}
    }
    void setExam(double e) {
        if(e<0){std::cout<<"erreur";}
        else{exam = e;}
    }
};
```

# Accesseurs et Manipulateurs

- Manipulateurs = mutateurs (setters)  
Problème de masquage en POO

```
void setDevoir(double devoir) {
    if(devoir<0){std::cout<<"erreur"}
    else{devoir = devoir;}
}
```

Un attribut cache le paramètre de la fonction

Erreur de compilation !!!

# Accesseurs et Manipulateurs

- Manipulateurs = mutateurs (setters)  
Problème de masquage en POO

```
void setDevoir(double devoir) {
    if(devoir<0){std::cout<<"erreur"}
    else{devoir = devoir;}
}
```

Un attribut cache le paramètre de la fonction

Erreur de compilation !!!

Solution 1 :

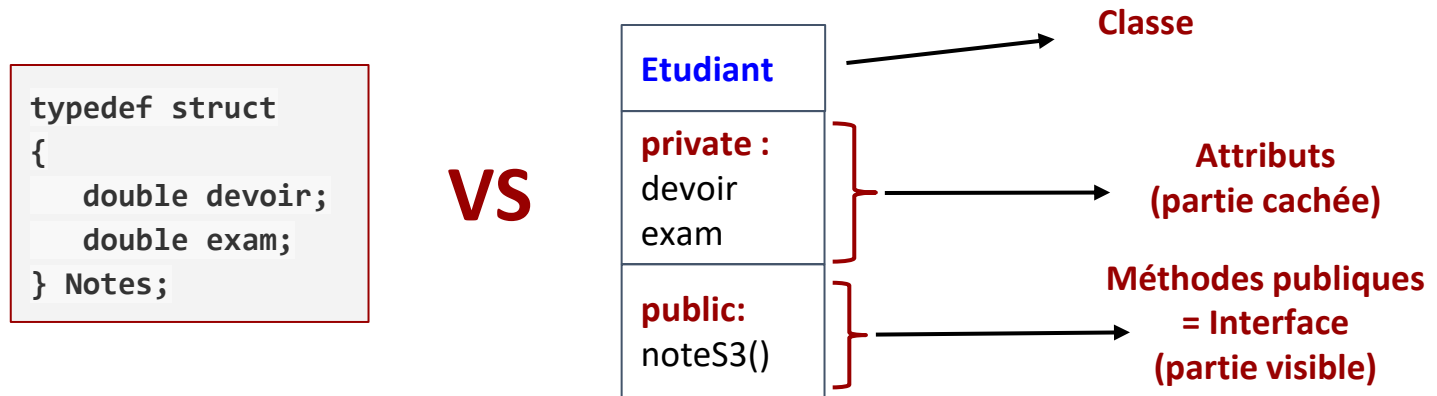
```
void setDevoir(double d) {
    if(devoir<0){std::cout<<"erreur"}
    else{devoir = d;}
}
```

Solution 2 : utilisation de this = un pointeur sur l'instance courante

```
void setDevoir(double devoir) {
    if(devoir<0){std::cout<<"erreur"}
    else{this->devoir = devoir;}
}
```

# Classes VS struct

- Les classes sont un prolongement naturel des structures
- Une classe est une struct
  - qui contient des fonctions → méthodes
  - certains champs peuvent être cachés : **private :**
  - certains champs peuvent être public → l'interface : **public:**



# Où définir les classes ?

## Etudiant.h ou Etudiant.hpp

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

## Etudiant.cpp

```
#include "Etudiant.h"

double Etudiant::getDevoir() {
    return devoir;
}

double Etudiant::getExam() {
    return exam;
}

void Etudiant::setDevoir(double d) {
    if(d<0){std::cout<<"erreur"}
    else{devoir = d;}
}

void Etudiant::setExam(double e) {
    if(e<0){std::cout<<"erreur"}
    else{exam = e;}
}

double Etudiant::notes3(){return 0.5*devoir+0.5*exam;}
```



# Où définir les classes ?

## Etudiant.h ou Etudiant.hpp

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double noteS3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

## main.cpp

```
#include <iostream>
#include "Etudiant.h"

int main() {
    Etudiant etd1;
    etd1.setDevoir(15);
    etd1.setExam(13);
    std::cout << etd1.noteS3();
    return 0;
}
```

Utilisateur externe de  
la classe

## Etudiant.cpp

```
#include "Etudiant.h"

double Etudiant::getDevoir() {
    return devoir;
}

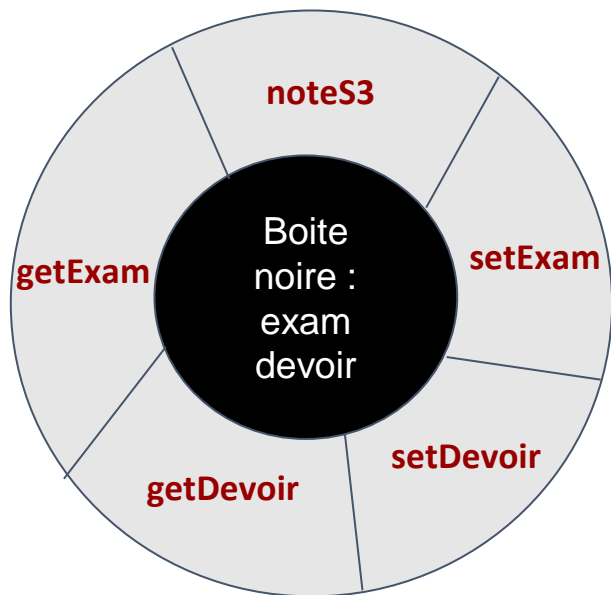
double Etudiant::getExam() {
    return exam;
}

void Etudiant::setDevoir(double d) {
    if(d<0){std::cout<<"erreur"}
    else{devoir = d;}
}

void Etudiant::setExam(double e) {
    if(e<0){std::cout<<"erreur"}
    else{exam = e;}
}

double Etudiant::noteS3(){return 0.5*devoir+0.5*exam;}
```

# Abstraction et encapsulation



L'application de l'encapsulation au concept

**Etudiant** permet d'avoir une classe avec :

- Un modèle interne caché (attributs)
  - Des méthodes à accès en lecture et en écriture aux variables :
    - exam
    - Devoir
- Une méthode à accès publique

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.devoir=15;
e1.exam=14;
std::cout<<e1.devoir<<std::endl;
```

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.devoir=15;
e1.exam=14;
std::cout<<e1.devoir<<std::endl;
```

Sortie du compilateur g++

Etudiant::devoir is private

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.setDevoir(15);
std::cout<<e1.getDevoir()<<std::endl;
```

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.setDevoir(15);
std::cout<<e1.getDevoir()<<std::endl;
```

Sortie du compilateur g++

15

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
std::cout<<e1.getDevoir()<<std::endl;
```

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
std::cout<<e1.getDevoir()<<std::endl;
```

Sortie du compilateur g++

6.93895e-310

C'est une valeur générée  
aléatoirement

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir;
    double exam;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```



# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
std::cout<<e1.getDevoir()<<std::endl;
```

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir=15;
    double exam=9;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
std::cout<<e1.getDevoir()<<std::endl;
```

Sortie du compilateur g++

15

```
#ifndef _Etudiant_H
#define _Etudiant_H

class Etudiant{
private:
    double devoir=15;
    double exam=9;
public:
    double notes3();
    //manipulateurs
    void setDevoir(double d);
    void setExam(double e);
    //accesseurs
    double getDevoir();
    double getExam();
};

#endif
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.setDevoir(-9);
std::cout<<e1.getDevoir()<<std::endl;
```

```
double Etudiant::setDevoir(double d) {
    if(d<0){std::cout<<"erreur\n"}
    else{devoir = d;}
}
```

# Accesseurs et Manipulateurs

Que produit le code suivant ?

```
Etudiant e1;
e1.setDevoir(-9);
std::cout<<e1.getDevoir()<<std::endl;
```

```
double Etudiant::setDevoir(double d) {
    if(d<0){std::cout<<"erreur\n"}
    else{devoir = d;}
}
```

Sortie du compilateur g++

```
erreur
6.94083e-310
```

Testons tout ceci dans un TP !