POO & C++

Programmation Orientée Objet
&
Langage C++

UVSQ université PARIS-SACLAY
ISTY
Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

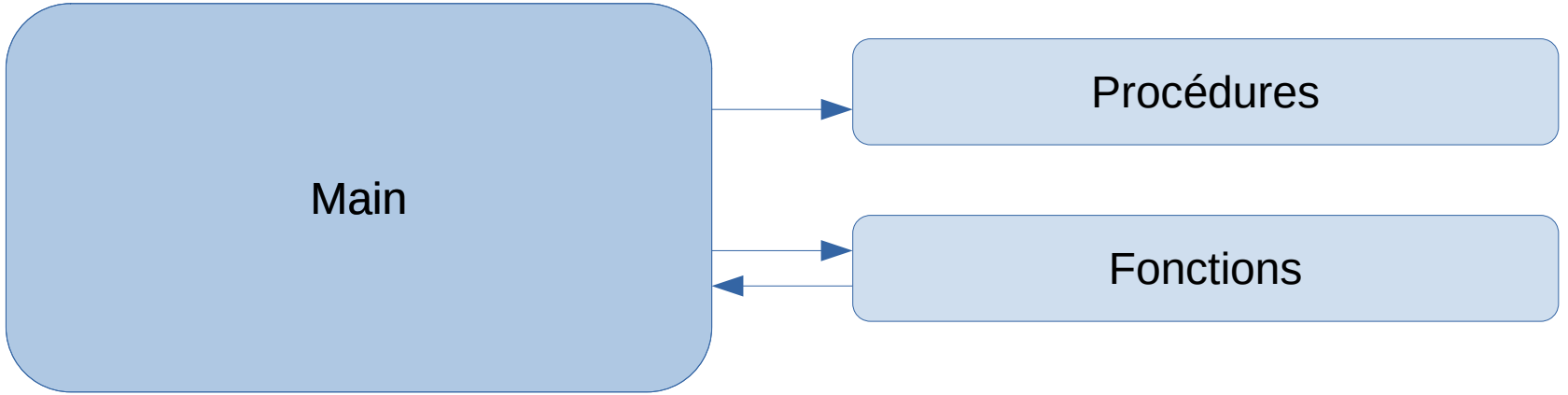
Olivier Snoeck

olivier@snoeck.fr

Les programmations

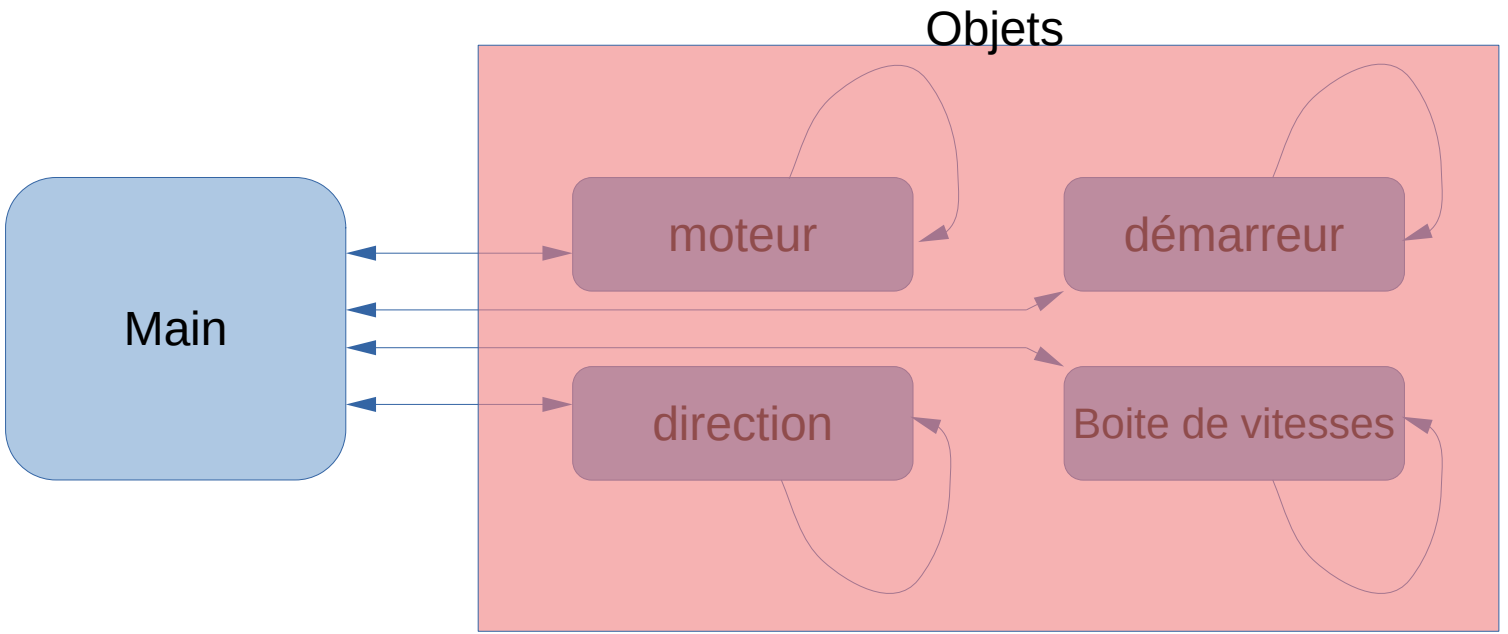
- La programmation procédurale
- La programmation orientée objet

La programmation procédurale est un paradigme de programmation qui gère des actions, de la logique, des événements.



Les programmations

- La programmation procédurale
- La programmation orientée objet

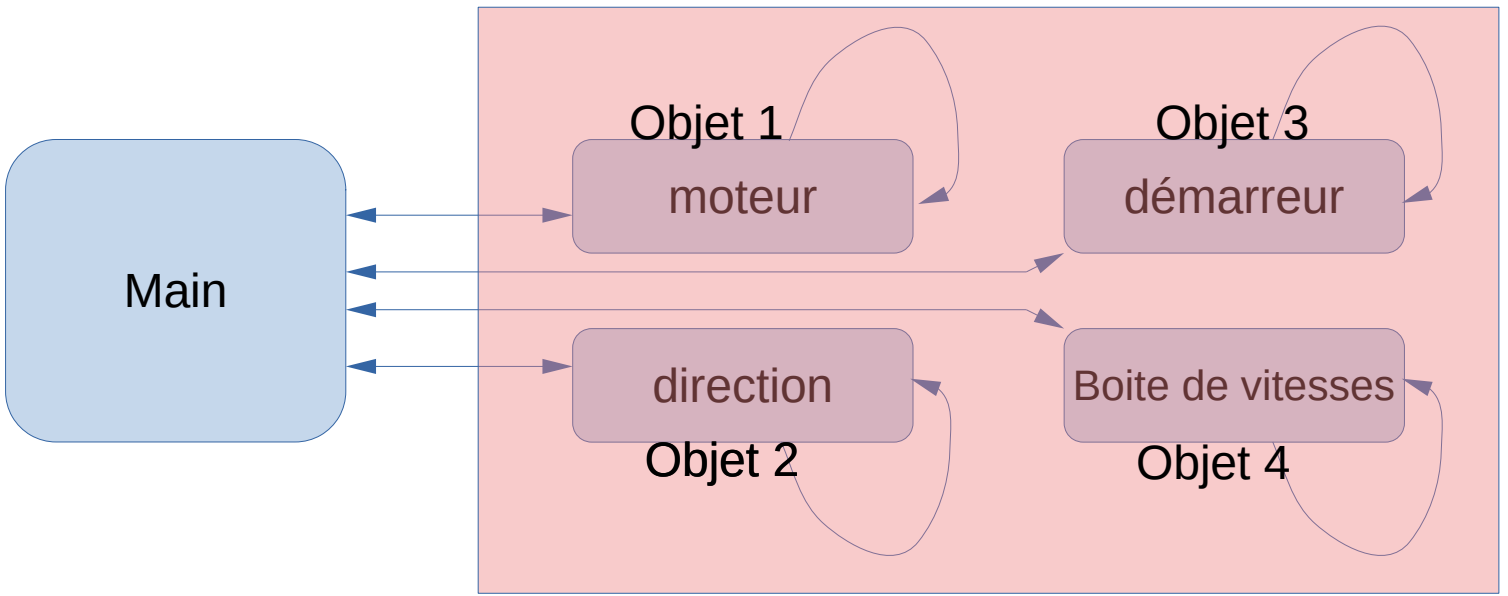


Les programmations

- La programmation procédurale

- La programmation orientée objet

La POO est un paradigme de programmation qui gère des briques logicielles



POO

La POO permet de :

- Développer un projet au sein d’une équipe,
- Pouvoir utiliser des outils collaboratifs,
- Faire évoluer le projet informatique,
- Sécuriser le projet informatique,
- Clarifier le rôle de chaque objet,
- Gérer des objets communs à plusieurs projets informatique,
- ….

C++

Le C++ moderne est en pleine expansion. On le retrouve sur :

- les objets connectés (IoT)
- Les smartphones;
- Les postes de travail;
- Les serveurs;
- Le cloud et l’intelligence artificielle (IA).

C++ est le langage le plus puissant et le plus rapide qui existe. Il tire parti au maximum de l’architecture matérielle et des processeurs avec une empreinte mémoire limitée.

POO

Les possibilités de Programmation Orientée Objet de C++ reposent sur le concept de classe.

Une classe est la généralisation de la notion de type défini par l'utilisateur, dans lequel se trouvent associées à la fois des données (on parle de « membres donnée ») et des fonctions (on parle de « fonctions membre » ou de méthodes).

En P.O.O. pure, les données sont « encapsulées », ce qui signifie que leur accès ne peut se faire que par le biais des méthodes.
C++ vous autorise à n'encapsuler qu'une partie seulement des données d'une classe.

POO

Les classes

POO : Les classes

Un des aspects majeurs de la programmation par objets est la création de types abstraits, faciles à utiliser et faciles à modifier.
Pour améliorer la sécurité, on préférera séparer en 2 parties :

- La partie visible par l'utilisateur de l'objet
- La partie invisible par l'utilisateur de l'objet.

POO : Les classes

Un des aspects majeurs de la programmation par objets est la création de types abstraits, faciles à utiliser et faciles à modifier.
Pour améliorer la sécurité, on préférera séparer en 2 parties :

- La partie visible par l'utilisateur
- La partie invisible par l'utilisateur

Une classe est un type abstrait défini par l'utilisateur, qui contient des données et qui effectue des actions
Il y aura donc plusieurs parties :

- Les variables ou les données, appelées attributs,
- Les fonctions, appelées méthodes.

POO : Les classes

Une classe est un type abstrait défini par l'utilisateur, qui contient des données et qui effectue des actions
Il y aura donc plusieurs parties :

- Les variables ou les données, appelées attributs,
- Les fonctions, appelées méthodes.

Une classe est un type abstrait défini par l'utilisateur, qui contient des données et qui effectue des actions
Il y aura donc plusieurs parties :

- Les variables ou les données, appelées attributs,
- Les fonctions, appelées méthodes.

POO : Les classes

Exemple 1 :

```
class heure
{
//Déclaration des attributs
public:
int nb_heures;
int nb_minutes;
int nb_secondes;
//Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
public:
void affiche() const; // la méthode ne permet que d'afficher donc, const.
void augmente(); // la méthode permet de modifier
void raz()
{
    nb_heures = nb_minutes = nb_secondes = 0;
}
}; // attention, le ; est obligatoire en fin de classe
```

POO : Les classes

Exemple 1 :

```
class heure
{
//Déclaration des attrributs
public:
int nb_heures;
int nb_minutes;
int nb_secondes;
//Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
public:
void affiche() const; // la méthode ne permet que d’afficher donc, const.
void augmente(); // la méthode permet de modifier
void raz()
{
    nb_heures = nb_minutes = nb_secondes = 0;
}
}; // attention, le ; est obligatoire en fin de classe
```

```
void heure::affiche() const
{
std::cout << nb_heures << " : " << nb_minutes << " : " << nb_secondes << std::endl;
}
```

POO : Les classes

Exemple 1 :

```
class heure
{
//Déclaration des attrributs
public:
int nb_heures;
int nb_minutes;
int nb_secondes;
//Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
public:
void affiche() const; // la méthode ne permet que d'afficher donc, const.
void augmente(); // la méthode permet de modifier
void raz()
{
    nb_heures = nb_minutes = nb_secondes = 0;
}
}; // attention, le ; est obligatoire en fin de classe
```

```
void heure::augmente()
{
    nb_secondes++;
    if (nb_secondes == 60)
    {
        nb_secondes = 0;
        ++nb_minutes;
        if (nb_minutes == 60)
        {
            nb_minutes = 0;
            ++nb_heures;
        }
    }
}
```

POO : Les classes

```
int main()
{
    heure letemps;
//instanciation de la classe heure: L'instanciation est l'action d'instancier,
de créer un objet à partir d'un modèle.
    // l'objet letemps est donc créé!!

    return 0;
}
```

POO : Les classes

```
int main()
{
    heure letemps;
//instanciation de la classe heure: L'instanciation est l'action d'instancier,
de créer un objet à partir d'un modèle.
    // l'objet letemps est donc créé!!
    letemps.raz();

    return 0;
}
```

POO : Les classes

```
int main()                                                                    0 : 0 : 0
{
    heure letemps;
//instanciation de la classe heure: L'instanciation est l'action d'instancier,
de créer un objet à partir d'un modèle.
    // l'objet letemps est donc créé!!
    letemps.raz();
    letemps.affiche();

    return 0;
}
```

POO : Les classes

```
int main()
{
    heure letemps;
//instanciation de la classe heure: L'instanciation est l'action d'instancier, de créer un objet à partir d'un modèle.
    // l'objet letemps est donc créé!!
    letemps.raz();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();

    return 0;
}
```

```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
0 : 0 : 3
```

POO : Les classes

```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
0 : 0 : 3
21 : 59 : 59
22 : 0 : 0
```

```
int main()
{
    heure letemps;
//instanciation de la classe heure: L'instanciation est l'action d'instancier,
de créer un objet à partir d'un modèle.
    // l'objet letemps est donc créé!!
    letemps.raz();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
                                        letemps.nb_heures=21;
                                        letemps.nb_minutes=59;
                                        letemps.nb_secondes=59;
                                        letemps.affiche();
                                        letemps.augmente();
                                        letemps.affiche();

                                        return 0;
                                    }
```

POO : Les classes

Nous avons vu

- La définition d'une classe : class heure { … } ;
- L'instanciation d'une classe en 1 objet : heure letemps;
- La déclaration des attributs (accessibles en mode public)
- La déclaration et la programmation des méthodes (accessibles en mode public)

Pour des méthodes complexes, membre d'une classe, on les programme en dehors de la classe

```
void heure::affiche() const
{}


void heure::augmente()
{}
```

POO : Les classes

Nous avons vu

- La définition d'une classe : class heure { … } ;
- L'instanciation d'une classe en 1 objet : heure letemps;
- La déclaration des attributs (accessibles en mode public)
- La déclaration et la programmation des méthodes (accessibles en mode public)

??? Et la sécurité ???

→ Si je ne veux pas que l'utilisateur de la classe heure soit capable de modifier les heures,…

```
letemps.nb_heures=21;
letemps.nb_minutes=59;
letemps.nb_secondes=59;
```

→ comment faire ???

POO : Les classes

Nous avons vu

- La définition d'une classe : class heure { … } ;
- L'instanciation d'une classe en 1 objet : heure letemps;
- La déclaration des attributs (accessibles en mode public)
- La déclaration et la programmation des méthodes (accessibles en mode public)

??? Et la sécurité ???

→ Si je ne veux pas que l'utilisateur de la classe heure soit capable de modifier les heures,…

```
letemps.nb_heures=21;
letemps.nb_minutes=59;
letemps.nb_secondes=59;
```

→ comment faire ???

L'encapsulation : public / private !!

TD

Programme 1 :
Écrire un programme utilisant une classe rectangle qui offre les fonctions suivantes :
- Calcul du périmètre (méthode)
- Calcul de la surface (méthode)
- Affichage (méthode) du périmètre et de la surface

Programme 2 :
Écrire un programme utilisant une classe cercle qui offre les fonctions suivantes :
- Calcul du périmètre (méthode)
- Calcul de la surface (méthode)
- Affichage (méthode) du périmètre et de la surface

POO

L’encapsulation

POO : L’encapsulation

```
1  #include <iostream>
2
3  class heure
4  {
5  //Déclaration des attrributs
6  private:
7  int nb_heures;
8  int nb_minutes;
9  int nb_secondes;
10 //Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
11 public:
12 void affiche() const; // la méthode ne permet que d’afficher donc, const.
13 void augmente(); // la méthode permet de modifier
14 void raz()
15 {⟺}
18 }; // attention, le ; est obligatoire en fin de classe
19
20 //écriture de la fonction affiche()
21 void heure::affiche() const
22 {⟺}
25 //écriture de la fonctione augmente()
26 void heure::augmente()
27 {⟺}
```

POO : L’encapsulation

```
#include <iostream>

class heure
{
//Déclaration des
private:
int nb_heures;
int nb_minutes;
int nb_secondes;
//Déclaration de
public:
void affiche() co
void augmente();
void raz()
{}
}; // attention,

//écriture de la
void heure::affic
{}
//écriture de la
void heure::augme
{}
```

```
g++ /tmp/2Pu1zVcrVT.cpp
/tmp/2Pu1zVcrVT.cpp: In function 'int main()':
/tmp/2Pu1zVcrVT.cpp:54:13: error: 'int heure::nb_heures' is private within this context
   54 |     letemps.nb_heures=21;
      |             ^~~~~~~~
/tmp/2Pu1zVcrVT.cpp:7:5: note: declared private here
    7 | int nb_heures;
      |     ^~~~~~~~
/tmp/2Pu1zVcrVT.cpp:55:13: error: 'int heure::nb_minutes' is private within this context
   55 |     letemps.nb_minutes=59;
      |             ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:8:5: note: declared private here
    8 | int nb_minutes;
      |     ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:56:13: error: 'int heure::nb_secondes' is private within this context
   56 |     letemps.nb_secondes=59;
      |             ^~~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:9:5: note: declared private here
    9 | int nb_secondes;
      |     ^~~~~~~~~~~
```

POO : L’encapsulation

```
1  #include <iostream>
2
3  class heure
4  {
5  //Déclaration des
6  private:
7  int nb_heures;
8  int nb_minutes;
9  int nb_secondes;
10 //Déclaration de
11 public:
12 void affiche() co
13 void augmente();
14 void raz()
15 {}
18 }; // attention,
19
20 //écriture de la
21 void heure::affic
22 {}
25 //écriture de la
26 void heure::augme
27 {}
```

```
g++ /tmp/2Pu1zVcrVT.cpp
/tmp/2Pu1zVcrVT.cpp: In function 'int main()':
/tmp/2Pu1zVcrVT.cpp:54:13: error: 'int heure::nb_heures' is private within this context
   54 |     letemps.nb_heures=21;
      |             ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:7:5: note: declared private here
    7 | int nb_heures;
      |     ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:55:13: error: 'int heure::nb_minutes' is private within this context
   55 |     letemps.nb_minutes=59;
      |             ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:8:5: note: declared private here
    8 | int nb_minutes;
      |     ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:56:13: error: 'int heure::nb_secondes' is private within this context
   56 |     letemps.nb_secondes=59;
      |             ^~~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:9:5: note: declared private here
    9 | int nb_secondes;
      |     ^~~~~~~~~~~
```

On a des attributs privés

POO : L'encapsulation

```
#include <iostream>

class heure
{
//Déclaration des
private:
int nb_heures;
int nb_minutes;
int nb_secondes;
//Déclaration de
public:
void affiche() co
void augmente();
void raz()
{}
; // attention,
//écriture de la
void heure::affic
{}
//écriture de la
void heure::augme
{}
```

```
g++ /tmp/2Pu1zVcrVT.cpp
/tmp/2Pu1zVcrVT.cpp: In function 'int main()':
/tmp/2Pu1zVcrVT.cpp:54:13: error: 'int heure::nb_heures' is private within this context
   54 |     letemps.nb_heures=21;
      |             ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:7:5: note: declared private here
    7 | int nb_heures;
      |     ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:55:13: error: 'int heure::nb_minutes' is private within this context
   55 |     letemps.nb_minutes=59;
      |             ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:8:5: note: declared private here
    8 | int nb_minutes;
      |     ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:56:13: error: 'int heure::nb_secondes' is private within this context
   56 |     letemps.nb_secondes=59;
      |             ^~~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:9:5: note: declared private here
    9 | int nb_secondes;
      |     ^~~~~~~~~~~
```

On a des attributs privés

Mais alors !
Comment les initialiser ???

POO : L'encapsulation

```
g++ /tmp/2Pu1zVcrVT.cpp
/tmp/2Pu1zVcrVT.cpp: In function 'int main()':
/tmp/2Pu1zVcrVT.cpp:54:13: error: 'int heure::nb_heures' is private within this context
   54 |     letemps.nb_heures=21;
      |             ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:7:5: note: declared private here
    7 | int nb_heures;
      |     ^~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:55:13: error: 'int heure::nb_minutes' is private within this context
   55 |     letemps.nb_minutes=59;
      |             ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:8:5: note: declared private here
    8 | int nb_minutes;
      |     ^~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:56:13: error: 'int heure::nb_secondes' is private within this context
   56 |     letemps.nb_secondes=59;
      |             ^~~~~~~~~~~
/tmp/2Pu1zVcrVT.cpp:9:5: note: declared private here
    9 | int nb_secondes;
      |     ^~~~~~~~~~~
```

On a des attributs privés

Mais alors !
Comment les initialiser ???

Les constructeurs !!!

POO

Les constructeurs
Les destructeurs

POO : Les constructeurs

Le constructeur n’est pas obligatoire

Le constructeur est dans la définition de la classe ou en dehors de la classe (selon sa taille)

Le constructeur a le même nom que la classe

Le constructeur peut prendre des paramètres

POO : Les constructeurs

```
1  #include <iostream>
2
3  class heure
4  {
5  //Déclaration des attributs
6  private:
7  int nb_heures;
8  int nb_minutes;
9  int nb_secondes;
10
11 public:
12     heure(int h, int m, int s):nb_heures(h), nb_minutes(m),nb_secondes(s){}
13     heure():nb_heures(0), nb_minutes(0),nb_secondes(0){}
14
15 //Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
16 public:
17 void affiche() const; // la méthode ne permet que d'afficher donc, const.
18 void augmente(); // la méthode permet de modifier
19 void raz()
20 {⟺}
23 }; // attention, le ; est obligatoire en fin de classe
24
25 //écriture de la fonction affiche()
26 void heure::affiche() const
27 {⟺}
30 //écriture de la fonctione augmente()
31 void heure::augmente()
32 {⟺}
```

Constructeur
(si il y a une initialisation)

Constructeur par défaut
(si aucune initialisation)

POO : Les constr

```cpp
#include <iostream>

class heure
{
//Déclaration des attributs
private:
int nb_heures;
int nb_minutes;
int nb_secondes;

public:
    heure(int h, int m, int s):nb_heures(h), nb_minutes(m),nb_secondes(s){}
    heure():nb_heures(0), nb_minutes(0),nb_secondes(0){}

//Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car t
public:
void affiche() const; // la méthode ne permet que d'afficher donc, const.
void augmente(); // la méthode permet de modifier
void raz()
{}
}; // attention, le ; est obligatoire en fin de classe

//écriture de la fonction affiche()
void heure::affiche() const
{}
//écriture de la fonctione augmente()
void heure::augmente()
{}
```

```cpp
int main()
{
    heure letemps;
    //instanciation de la classe heure: L
        //partir d'un modèle.
    // l'objet letemps est donc créé!!
    letemps.raz();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();
    letemps.augmente();
    letemps.affiche();

    /*letemps.nb_heures=21;
    letemps.nb_minutes=59;
    letemps.nb_secondes=59;
    */
    heure maintenent(21,59,59);
    maintenent.affiche();
    maintenent.augmente();
    maintenent.affiche();
    return 0;
}
```

POO : Les constr

```
1  #include <iostream>
2
3  class heure
4  {
5  //Déclaration des attributs
6  private:
7  int nb_heures;
8  int nb_minutes;
9  int nb_secondes;
10
11 public:
12     heure(int h, int m, int s):nb_heures(h), nb_minutes(m),nb_secondes(s){}
13     heure():nb_h
14
15 //Déclaration de
16 public:
17 void affiche() c
18 void augmente(); // la méthode permet de modifier
19 void raz()
20 {}
23 }; // attention, le ; est obligatoire en fin de classe
24
25 //écriture de la fonction affiche()
26 void heure::affiche() const
27 {}
30 //écriture de la fonctione augmente()
31 void heure::augmente()
32 {}
```

```
46 int main()
47 {
48     heure letemps;
49     //instanciation de la classe heure: L
           //partir d'un modèle.
50     // l'objet letemps est donc créé!!
51     letemps.raz();
52     letemps.affiche();
53     letemps.augmente();
54     letemps.affiche();
                          );
                          ;
                          );
                          ;
60     /*letemps.nb_heures=21;
61     letemps.nb_minutes=59;
62     letemps.nb_secondes=59;
63     */
64     heure maintenent(21,59,59);
65     maintenent.affiche();
66     maintenent.augmente();
67     maintenent.affiche();
68     return 0;
69 }
```

J'ai été obligé d'instancier à nouveau car l'objet a été construit.
Pour utiliser le même objet « letemps », il aurait fallu détruire.
On construit puis on détruit...

POO : Les destructeurs

Le destructeur n’est pas obligatoire

Le destructeur est dans la définition de la classe ou en dehors de la classe (selon sa taille)

Le destructeur a le même nom que la classe avec un ~ avant le nom

Le destructeur est vide de paramètres : il nettoie...

POO : Les destructeurs

```
74  int main()
75  {
76      heure1();
77
78      heure2(21,59,59);
79
80      return 0;
81  }
```

```
51  void heure1()
52  {
53      heure letemps;
54      //instanciation de la classe heure: L'instanciation est l'action d'instancier, de créer un objet
            à //partir d'un modèle.
55      // l'objet letemps est donc créé!!
56      letemps.raz();
57      letemps.affiche();
58      letemps.augmente();
59      letemps.affiche();
60      letemps.augmente();
61      letemps.affiche();
62      letemps.augmente();
63      letemps.affiche();
64      //l'objet letemps est en fin de vie donc, il sera détruit.  Il y a un nettoyage de la mémoire.
65  }
66  void heure2(int h, int m, int s)
67  {
68      heure letemps(h,m,s);
69      letemps.affiche();
70      letemps.augmente();
71      letemps.affiche();
72      //l'objet letemps est en fin de vie donc, il sera détruit. Il y a un nettoyage de la mémoire.
73  }
```

POO : Le

```
51  void heure1()
52  {
53      heure letemps;
54      //instanciation de la classe heure: à //partir d'un modèle.
55      // l'objet letemps est donc créé!!
56      letemps.raz();
57      letemps.affiche();
58      letemps.augmente();
59      letemps.affiche();
60      letemps.augmente();
61      letemps.affiche();
62      letemps.augmente();
63      letemps.affiche();
64      //l'objet letemps est en fin de vie
65  }
66  void heure2(int h, int m, int s)
67  {
68      heure letemps(h,m,s);
69      letemps.affiche();
70      letemps.augmente();
71      letemps.affiche();
72      //l'objet letemps est en fin de vie
73  }
```

```
74  int main()
75  {
76      heure1();
77
78      heure2(21,59,59);
79
80      return 0;
81  }
```

```
1   #include <iostream>
2
3   class heure
4   {
5   //Déclaration des attrributs
6   private:
7   int nb_heures;
8   int nb_minutes;
9   int nb_secondes;
10
11  public:
12      heure(int h, int m, int s):nb_heures(h), nb_minutes(m),nb_secondes(s)
13      {std::cout << "Hi :-)"<< std::endl;}
14      heure():nb_heures(0), nb_minutes(0),nb_secondes(0)
15      {std::cout << "Hi ;-)"<< std::endl;}
16      ~heure()
17      {
18          std::cout << "Kiss"<< std::endl;
19      }
20
21  //Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
22  public:
23  void affiche() const; // la méthode ne permet que d'afficher donc, const.
24  void augmente(); // la méthode permet de modifier
25  void raz()
26  {⟺}
29  }; // attention, le ; est obligatoire en fin de classe
30
31  //écriture de la fonction affiche()
32  void heure::affiche() const
33  {⟺}
36  //écriture de la fonctione augmente()
37  void heure::augmente()
38  {⟺}
```

```cpp
#include <iostream>

class heure
{
//Déclaration des attrributs
private:
int nb_heures;
int nb_minutes;
int nb_secondes;

public:
    heure(int h, int m, int s):nb_heures(h), nb_minutes(m),nb_secondes(s)
    {std::cout << "Hi :-)"<< std::endl;}
    heure():nb_heures(0), nb_minutes(0),nb_secondes(0)
    {std::cout << "Hi ;-)"<< std::endl;}
    ~heure()
    {
        std::cout << "Kiss"<< std::endl;
    }

//Déclaration de 2 méthodes et déclaration mais écriture d'une fonction car très courte
public:
void affiche() const; // la méthode ne permet que d'afficher donc, const.
void augmente(); // la méthode permet de modifier
void raz()
{}
}; // attention, le ; est obligatoire en fin de classe

//écriture de la fonction affiche()
void heure::affiche() const
{}
//écriture de la fonctione augmente()
void heure::augmente()
{}
```

```cpp
int main()
{
    heure1();

    heure2(21,59,59);

    return 0;
}
```

Output

```
/tmp/2Pu1zVcrVT.o
Hi ;-)
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
0 : 0 : 3
Kiss
Hi :-)
21 : 59 : 59
22 : 0 : 0
Kiss
```

POO : L’encapsulation

Dans la déclaration d’une classe, il est possible de protéger certaines données-membres ou fonctions-membres en les rendant invisibles de l’extérieur : c’est ce qu’on appelle l’encapsulation.

A quoi cela sert-il ?
Supposons qu’on veuille programmer une classe Cercle avec comme données-membres :
– un point représentant le centre,
– un nombre r présentant le rayon,
– un nombre représentant la surface du cercle.
Permettre l’accès direct à la variable surface, c’est s’exposer à ce qu’elle soit modifiée depuis l’extérieur,et cela serait catastrophique puisque l’objet risquerait alors de perdre sa cohérence (la surface dépend en fait du rayon). Il est donc indispensable d’interdire cet accès,ou au moins permettre à l’objet de le contrôler

Exemple

```
#include <iostream>

class LETTER
{
    private:
        char lettre;
    public:
        LETTER (char c):lettre(c)
        {
            std::cout << "LETTER ("<<c<<")"<<std::endl;
        }
        ~LETTER()
        {
            std::cout<<"FIN_LETTER("<<lettre<<")";
        }
};

class MOT
{
    private:
        LETTER debut;
        LETTER fin;
    public:
        MOT(char x):debut(0), fin(x)
        {
            std::cout<<"MOT"<<std::endl;
            std::cout<<x<<std::endl;
        }
        ~MOT()
        {
            std::cout<<"~MOT"<<std::endl;
        }
        void affiche()
        {
            std::cout <<"test" <<std::endl;
        }
};

int main()
{
    LETTER a('A'), b('B');
    MOT m('C'), n('D');
    return 0;
}
```

Qu’affiche ce programme ?

Exemple

```
LETTER (A)
LETTER (B)
LETTER (·)
LETTER (C)
MOT
C
LETTER (·)
LETTER (D)
MOT
D
~MOT
FIN_LETTER(D)FIN_LETTER(·)~MOT
FIN_LETTER(C)FIN_LETTER(·)FIN_LETTER(B)FIN_LETTER(A)
```

TD

Nous souhaitons développer une classe nommée ListData simplifiant les calculs liés aux vecteurs de données et assurant les services suivants :
➜ Ajout de nouvelles données par l'utilisateur de la classe,
➜ Lecture du nombre de données fournies par l'utilisateur,
➜ Calcul de la somme des données fournies par l'utilisateur,
➜ Calcul de la moyenne des données fournies par l'utilisateur,
➜ Calcul de la valeur minimum contenue dans les données fournies,
➜ Calcul de la valeur maximum contenue dans les données fournies,
➜ Affichage des données présentes dans la liste à l'écran.

Hypothèse :
➜ L'utilisateur ne peut pas fournir plus de 10 données. Chaque donnée est rentrée une par une dans le tableau (ajoutée à la fin).

Classe, méthode, constructeur, destructeur obligatoires

Accesseurs et Mutateurs

La protection des données membres :
→ L'un des aspects les plus essentiels du concept « orienté objet » est l'encapsulation, qui consiste à définir des étiquettes pour les données membres et les fonctions membres afin de préciser si celles-ci sont accessibles à partir d'autres classes ou non...

De cette manière, des données membres portant l'étiquette private ne peuvent pas être manipulées directement par les fonctions membres des autres classes. Ainsi, pour pouvoir manipuler ces données membres, le créateur de la classe (vous en l'occurrence) doit prévoir des fonctions membres spéciales portant l'étiquette public, permettant de manipuler ces données.

- Les fonctions membres permettant d'accéder aux données membres sont appelées accesseurs, parfois **getter** (appellation d'origine anglophone)
- Les fonctions membres permettant de modifier les données membres sont appelées mutateurs, parfois **setter** (appellation d'origine anglophone)

L’accesseur

Un accesseur est une fonction membre permettant de récupérer le contenu d'une donnée membre protégée. Un accesseur, pour accomplir sa fonction :

- doit avoir comme type de retour le type de la variable à renvoyer
- ne doit pas nécessairement posséder d'arguments

Une convention de nommage veut que l'on fasse commencer de façon préferrentielle le nom de l'accesseur par le préfixe Get, afin de faire ressortir sa fonction première.

La syntaxe d'un accesseur réduit à sa plus simple expression ressemble donc à ceci :

```
class MaClasse{
	private :
		TypeDeMaVariable MaVariable;
	public :
		TypeDeMaVariable GetMaVariable();
};

TypeDeMaVariable MaClasse::GetMaVariable(){
	return MaVariable;
}
```

Le mutateur

Un mutateur est une fonction membre permettant de modifier le contenu d'une donnée membre protégée. Un mutateur, pour accomplir sa fonction :

- doit avoir comme paramètre la valeur à assigner à la donnée membre. Le paramètre doit donc être du type de la donnée membre
- ne doit pas nécessairement renvoyer de valeur (il possède dans sa plus simple expression le type void)

Une convention de nommage veut que l'on fasse commencer de façon préferrentielle le nom du mutateur par le préfixe Set.

La syntaxe d'un mutateur réduit à sa plus simple expression ressemble donc à ceci :

```
class MaClasse{
	private :
		TypeDeMaVariable MaVariable;
	public :
		void SetMaVariable(TypeDeMaVariable);
};
MaClasse::SetMaVariable(TypeDeMaVariable MaValeur){
	MaVariable = MaValeur;
}
```

Exercice 1

Accesseurs (on ajoute get)

Mutateurs (on ajoute set)

```cpp
#include<iostream>
#include <cstdlib>

using namespace std;

class Rectangle
{
public:
  Rectangle(unsigned int initLargeur, unsigned int initHauteur);
  ~Rectangle();
  unsigned int getLargeur() const { return largeur; };
  unsigned int getHauteur() const { return hauteur; };
  unsigned int perimetre() const { return 2*(largeur+hauteur); };
  unsigned int surface() const { return largeur * hauteur; };
  void setLargeur(unsigned int newLargeur) { largeur = newLargeur; };
  void setHauteur(unsigned int newHauteur) { hauteur = newHauteur; };
  void afficher();

private:
  unsigned int largeur;
  unsigned int hauteur;
};

Rectangle::Rectangle(unsigned int initLargeur, unsigned int initHauteur)
{
  largeur = initLargeur;
  hauteur = initHauteur;
}

Rectangle::~Rectangle()
{
}
```

```cpp
void Rectangle::afficher()
{
  for(unsigned int i=0; i < hauteur; i++)
  {
    for(unsigned int j=0; j < largeur; j++)
      cout << "*";
    cout << endl;
  }
}
```

Exercice 1

Prenons largeur = 10 et longueur = 20
Que réalise ce programme ?

Mutateurs (on ajoute set)

```
44  int main()
45  {
46    Rectangle monRectangle(0,0);
47    char choix = '0';
48    unsigned int value;
49
50    while(true)
51    {
52      do
53      {
54        cout << " Rectangle - Menu" << endl;
55        cout << "1 - Modifier largeur du rectangle" << endl;
56        cout << "2 - Modifier hauteur du rectangle" << endl;
57        cout << "3 - Calculer les propriétés du rectangle" << endl;
58        cout << "4 - Afficher le rectangle" << endl;
59        cout << "5 - Quitter" << endl;
60
61        cin >> choix;
62      }while(choix < '1' || choix > '5');
63
64      switch(choix)
65      {
66        case '1':
67          cout << "Nouvelle largeur : ";
68          cin >> value;
69          monRectangle.setLargeur(value);
70          break;
71        case '2':
72          cout << "Nouvelle hauteur : ";
73          cin >> value;
74          monRectangle.setHauteur(value);
75          break;
76        case '3':
77          cout << "Périmètre : " << monRectangle.perimetre() << endl;
78          cout << "Surface : " << monRectangle.surface() << endl;
79          break;
80        case '4':
81          monRectangle.afficher();
82          break;
83        case '5':
84          exit(0);
85          break;
86        default:
87          cout << "Erreur ! Choix invalide." << endl;
88          exit(1);
89      }
90    }
91
92    return 0;
93  }
```

Exercice 2

Source

1/ On voudrait gérer les étudiants d'une institution à l'aide d'une classe Etudiant définie par :les attributs suivants :

- nom : nom d'un étudiant
- prénom: prénom d'un étudiant
- tabnotes : tableau contenant les notes d'un étudiant, sachant qu'un étudiant a au total 10 notes.

les méthodes suivantes :
- void saisie (), permettant la saisie d'un étudiant
- void affichage (), permettant l'affichage d'un étudiant
- float moyenne (), retourne comme résultat la moyenne des notes d'un étudiant.
- int admis (), retourne comme résultat la valeur 1, si un étudiant est admis et la valeur 0, sinon. Un étudiant est considéré comme étant

admis lorsque la moyenne de ses notes est supérieure ou égale à 10.
- int exae_quo (Etudiant E), retourne comme résultat la valeur 1, si deux étudiants ont la même moyenne et la valeur 0, sinon.

Ecrire la classe Etudiant dans le langage C++.

Exercice 2

Source

2/ On voudrait maintenant représenter, à l'aide d'une nouvelle classe Etudiant_en_Maitrise, certains étudiants particuliers dans cette institution qui sont les étudiants en dernière année d'études. Ces étudiants possèdent en effet un attribut supplémentaire : note_memoire, qui représente la note de leur mémoire de fin d'études.

Les méthodes à associer à cette classe sont les suivantes :
- void saisie (), permettant la saisie d'un étudiant en maîtrise
- void affichage (), permettant l'affichage d'un étudiant en maîtrise
- float moyenne (), retourne comme résultat la moyenne des notes d'un étudiant en maîtrise
- int admis (), retourne comme résultat la valeur 1, si un étudiant est admis et la valeur 0, sinon. Un étudiant en maîtrise est considéré comme étant admis lorsque, d'une part, la moyenne de ses notes est supérieure ou égale à 10 et d'autre part la note obtenue pour son mémoire de fin d'études est supérieure ou égale à 10.

- int exae_quo (Etudiant_en_Maitrise E), retourne comme résultat la valeur 1, si deux étudiants ont d'une part la même moyenne et d'autre part, la même note de mémoire et retourne la valeur 0, sinon.
a) Quelles sont les méthodes qui sont à redéfinir dans la classe Etudiant_en_Maitrise ?
b) Ecrire la classe Etudiant_en_Maitrise dans le langage C++.

Exercice 2 (correction 1ère partie)

Source

```
class Etudiant
{ private:
        char nom[50], prenom[50];
        float tabnotes[10] ;

  public :
        void saisie () ;
        void affichage () ;
        float moyenne() ;
        int admis() ;
        int exae_quo (Etudiant E) ;

} ;

void Etudiant ::saisie ()
{       int i ;
        cout << "Donner le nom :" ;
        cin >> nom ;
        cout << "Donner le prénom :" ;
        cin >> prenom ;
        cout << "Saisie des notes \n" ;
        for (i = 0 ; i < 10 ; i++)
        {
                cout << "Donner la note N°" << i<< " : " ;
                cin >> tabnotes[i] ;
        }
}
```

```
void Etudiant ::affichage ()
{       int i ;
        cout << "Le nom :"<<nom<< endl ;
        cout << "Le prénom :" <<prenom<< endl ;
        for (i = 0 ; i < 10 ; i++)
                cout << "La note N°" << i << "est " << tabnotes[i]<< endl ;
}

float Etudiant ::moyenne()
{ int i ;
  float som = 0;
        for (i = 0 ; i < 10 ; i++)
                som + = tabnotes[i] ;
        return (som/10)
}

int Etudiant ::admis()
{ if (moyenne() >= 10) return (1); else return (0);}
int Etudiant ::Exae_quo(Etudiant E)
{ if (moyenne() == E.moyenne()) return (1); else return (0);}
```

Exercice 2 (correction 2nde partie)

Source

a) Les méthodes qui sont à redéfinir dans la classe Etudiant_en_Maitrise sont : saisie, affichage, admis et esae_quo.

Exercice 2 (correction 2nde partie)

Source

```
class Etudiant_en_Maitrise : public Etudiant
{ private:
          float note_memoire ;
  public :
          void saisie () ;
          void affichage () ;
          int admis() ;
          int exae_quo (Etudiant_en_Maitrise E) ;
} ;

void Etudiant_en_Maitrise ::saisie ()
{         Etudiant ::saisie () ;
         cout << "Donner la note du mémoire :" ;
          cin >> note_memoire ;
}

void Etudiant_en_Maitrise ::affichage ()
{         Etudiant :: affichage () ;
         cout << "La note du mémoire :" << note_memoire<< endl ;
}

int Etudiant_en_Maitrise ::admis()
{ if ((moyenne() >= 10) && (note_memoire >=10))return (1); else return (0);}

int Etudiant_en_Maitrise ::Exae_quo(Etudiant E)
{ if ((moyenne() == E.moyenne()) && (note_memoire == E.note_memoire)) return (1); else return (0);}
```

L'opérateur *this* en C++

L'opérateur *this* en C++ est un pointeur spécial qui est automatiquement défini dans toutes les fonctions membres d'une classe.

L’opérateur *this* pointe vers l'instance courante de la classe qui appelle la fonction.

L'opérateur *this* en C++

```
#include <iostream>

class Exemple {
private:
    int valeur;

public:
    Exemple(int val) {
        this->valeur = val;  // Utilisation de "this"
    }

    void afficher() {
        std::cout << "Valeur : " << this->valeur << std::endl;
    }
};
```

```
int main() {
    Exemple obj(10);
    obj.afficher();
    return 0;
}
```

this est un pointeur implicite qui contient l'adresse de l'objet courant.
Il est disponible dans toutes les fonctions membres non statiques d'une classe.
this ne peut pas être modifié (c'est un pointeur constant).

L'opérateur *this* en C++

```
#include <iostream>

class Exemple {
private:
    int valeur;

public:
    Exemple(int val) {
        this->valeur = val;  // Utilisation de "this"
    }

    void afficher() {
        std::cout << "Valeur : " << this->valeur << std::endl;
    }
};
```

```
int main() {
    Exemple obj(10);
    obj.afficher();
    return 0;
}
```

this est un pointeur implicite qui contient l'adresse de l'objet courant.
Il est disponible dans toutes les fonctions membres non statiques d'une classe.
this ne peut pas être modifié (c'est un pointeur constant).

Ici : this->valeur = val;
Cela indique que valeur de l'instance courante est assigné à val.
Cela est utile pour éviter les conflits de noms entre l'attribut de classe et le paramètre du constructeur.

this pour retourner l'objet courant

```
class Chaine {
private:
    std::string texte;

public:
    Chaine(std::string t) { this->texte = t; }

    Chaine& ajouter(std::string t) {
        this->texte += " " + t;
        return *this;  // Retourne l'objet courant par référence
    }

    void afficher() {
        std::cout << "Texte : " << this->texte << std::endl;
    }
};
```

```
int main() {
    Chaine c("Bonjour");
    c.ajouter("tout").ajouter("le monde").afficher();

    return 0;
}
```

La fonction ajouter() retourne *this, c'est-à-dire l'objet lui-même, ce qui permet d'enchaîner plusieurs appels (c.ajouter("tout").ajouter("le monde").afficher();).

Les membres statiques

Les membres constants (attributs & méthodes)

```
class …
{
        const int i ;
        void f(…) const ;
} ;
```

Exemple :

```
class math
{
        const double PI;
        public :
                math() : PI(3.141459){}
} ;
```

const pour signaler au compilateur que l'attribut ne changera pas
const pour signaler au compilateur qu'une méthode ne modifiera pas d'attributs

Les membres statiques

Les membres mutables (attributs)

Une méthode peut être constante mais permettre à 1 attribut d'être modifié...

```
1  #include <iostream>
2
3  int main()
4  {
5  class X
6  {
7      public:
8          X(int a=0, int b=0): i(a),j(b){}
9          int i;
10         mutable int j;
11 };
12
13 const X x1(5,2);
14 X x2;
15 std::cout << x1.i << " " <<x1.j; // affiche 5 2
16
17 x1.j = 3; // ok, valide puisque j est mutable alors que la classe est déclarée constante
18 x1.i = 4; // Créé une erreur: i n'est pas mutable alors que la classe est déclarée constante
19 return 0;
20 }
```

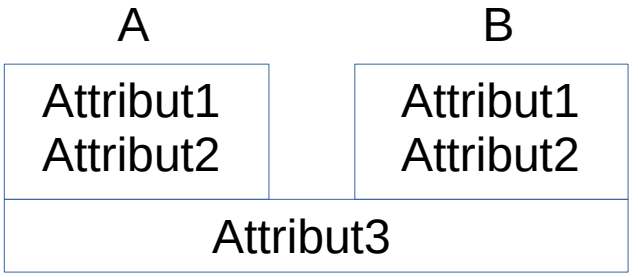
```
18 | x1.i = 4; // Créé une erreur: i n'est pas mutable alors que la classe est déclarée constante
```

Les membres statiques

Les membres statiques (attributs)

Il s’agit d’un attribut commun pour plusieurs instanciations de classe

```
1  class C
2  {
3      int attribut1;
4      char attribut2;
5      static double attribut3;
6  };
7  C A,B;
```



L’encapsulation

Pour quelle raison utiliser l’encapsulation dans une classe ?

- Rendre tous les membres des données privés.
- Créez des fonctions publiques setter et getter pour chaque membre de données de manière à ce que la fonction **set** définisse la valeur de membre de données, et la fonction **get** récupère les données.

Objectif : sécuriser l’accès aux données en utilisant des membres « privé »

L’encapsulation

Source

```
#include<iostream>

using namespace std;

class Encapsulation
{
    private:
        // Données cachées de l'extérieur
        int var;

    public:
        // fonction pour définir la valeur de la variable var
        void set(int v)
        { var = v;     }

        // fonction pour retourner la valeur de la variable var
        int get()
        {    return var;        }
};
```

```
int main()
{
    Encapsulation data;
    data.set(10);
    cout << data.get();
    return 0;
}
```

mutateur

accesseur

L’encapsulation

L’amitié

Il est possible de rendre les membres privés accessibles pour un objet seulement (fonction ou classe) : c’est l’amitié
Dans la classe qui doit « ouvrir » l’accès à ses membres privés, on déclare une fonction ou une classe comme étant amie.

```
class X
{
    friend class Y ;
    friend void f(int ...) ;
}
```

L’encapsulation

L’amitié

```
#include <stdio.h>

class Hote
{
    friend class Amie;  // Toutes les méthodes de Amie sont amies.
    int i;              // Donnée privée de la classe Hote.

public:
    Hote(void)
    {
        i=0;
        return ;
    }
};

Hote h;

class Amie
{
public:
    void print_hote(void)
    {
        printf("%d\n", h.i); // Accède à la donnée privée de h.
        return ;
    }
};
```

```
int main(void)
{
    Amie a;
    a.print_hote();
    return 0;
}
```

Alors que i est privée

Résultat : 0

POO

Les assertions

Les assertions

Prenons cette méthode :

```
int Divise(int numerateur, int denominateur)
{
    return numerateur / denominateur ;
}
```

On peut éviter le problème du dénominateur nul grâce à cette solution

Les assertions

Prenons cette méthode :

```
int Divise(int numerateur, int denominateur)
{
	return numerateur / denominateur ;
}
```

On peut éviter le problème du dénominateur nul grâce à cette solution

```
int Divise(int numerateur, int denominateur)
{
	int quotient = 0 ;
	if (denominateur!= 0)
		quotient = numerateur / denominateur
	return quotient;
}
```

Les assertions

Prenons cette méthode :

```
int Divise(int numerateur, int denominateur)
{
    return numerateur / denominateur ;
}
```

On peut éviter le problème du dénominateur nul grâce à cette solution

```
int Divise(int numerateur, int denominateur)
{
    int quotient = 0 ;
    if (denominateur!= 0)
        quotient = numerateur / denominateur
    return quotient;
}
```

→ si quotient = 0, est-ce que denominateur infini ou une erreur car nul ?

Les assertions

Prenons cette méthode :

```
int Divise(int numerateur, int denominateur)
{
    return numerateur / denominateur ;
}
```

On peut éviter le problème du dénominateur nul grâce à cette solution

```
#include <cassert>
int Divise(int numerateur, int denominateur)
{
    assert (denominateur!= 0) ;
    return numerateur / denominateur;
}
```

Assert est un « contrat »….

Les assertions

= un contrat

main.cpp

Run

```cpp
#include <iostream>
#include <cassert>

int Divise(int numerateur, int denominateur)
{
    assert (denominateur!= 0);
    return numerateur / denominateur;
}

int main()
{
    int num, denum;
    std::cout << "entrez la valeur de a et de b de l'équation y=a/b" << std::endl;
    std::cin >> num;
    std::cout << "entrez la valeur de a et de b de l'équation y=a/b" << std::endl;
    std::cin >> denum ;

    std::cout << "le résultat est " << Divise(num,denum);
}
```

Output

```
/tmp/miLHsbmjgb.o
entrez la valeur de a et de b de l'équation y=a/b
10
entrez la valeur de a et de b de l'équation y=a/b
0
miLHsbmjgb.o: /tmp/miLHsbmjgb.cpp:6: int Divise(int, int): Assertion `denominateur!= 0' failed.
Aborted
```

Les assertions

= un contrat

Exercice :
Créez une assertion dans une méthode pour un calcul de racine carrée.
std::sqrt

POO

Les exceptions

Les exceptions

Une assertion est très utile en mode de prototypage, en mode de développement.
Une exception est très intéressante en production : l'ordinateur essai (try) le code.

Les mots clefs sont :
Try{…} pour tester une portion de programme
Throw qui signale une erreur en lançant un objet
Catch(...){…} qui récupère le code et gère l'erreur

Les exceptions

```
#include <iostream>

using namespace std;

class erreur   // Première exception possible, associée
               // à l'objet erreur.
{
public:
    int cause;  // Entier spécifiant la cause de l'exception.
    // Le constructeur. Il appelle le constructeur de cause.
    erreur(int c) : cause(c) {}
    // Le constructeur de copie. Il est utilisé par le
mécanisme
    // des exceptions :
    erreur(const erreur &source) : cause(source.cause) {}
};

class other {};   // Objet correspondant à toutes
                  // les autres exceptions.
```

Exemple complet

```
int main(void)
{
    int i;          // Type de l'exception à générer.
    cout << "Tapez 0 pour générer une exception Erreur, "
       "1 pour une Entière :";
    cin >> i;        // On va générer une des trois exceptions
                     // possibles.
    cout << endl;
    try              // Bloc où les exceptions sont prises en charge
    {
       switch (i)    // Selon le type d'exception désirée,
       {
       case 0:
          {
             erreur a(0);
             throw (a);   // on lance l'objet correspondant
                          // (ici, de classe erreur).
                          // Cela interrompt le code. break est
                          // donc inutile ici.
          }
       case 1:
          {
             int a=1;
             throw (a);   // Exception de type entier.
          }…..
```

Les exceptions

Exemple complet

```
int main(void)
{
    int i;            // Type de l'exception à générer.
    cout << "Tapez 0 pour générer une exception Erreur, "
        "1 pour une Entière :";
    cin >> i;         // On va générer une des trois exceptions
                      // possibles.
    cout << endl;
    try               // Bloc où les exceptions sont prises en charge.
    {
        switch (i)    // Selon le type d'exception désirée,
        {
        case 0:
            {
                erreur a(0);
                throw (a);   // on lance l'objet correspondant
                             // (ici, de classe erreur).
                             // Cela interrompt le code. break est
                             // donc inutile ici.
            }
        case 1:
            {
                int a=1;
                throw (a);   // Exception de type entier.
            }
        default:             // Si l'utilisateur n'a pas tapé 0 ou 1,
            {
                other c;     // on crée l'objet c (type d'exception
                throw (c);   // other) et on le lance.
            }
        }
    }                 // fin du bloc try. Les blocs catch suivent :
    catch (erreur &tmp) // Traitement de l'exception erreur ...
    {                 // (avec récupération de la cause).
        cout << "Erreur erreur ! (cause " << tmp.cause << ")" << endl;
    }
    catch (int tmp)   // Traitement de l'exception int...
    {
        cout << "Erreur int ! (cause " << tmp << ")" << endl;
    }
    catch (...)       // Traitement de toutes les autres
    {                 // exceptions (...).
                      // On ne peut pas récupérer l'objet ici.
        cout << "Exception inattendue !" << endl;
    }
    return 0;
}
```

TD

1) Écrire un programme dans lequel on demande à l'utilisateur de saisir un entier en gérant l'exception dans le cas où il ne saisit pas un entier correctement.
2) Écrire un programme dans lequel on demande à l'utilisateur de saisir un entier en gérant l'exception dans le cas où il ne saisit pas un entier correctement en lui demandant de refaire la saisie.
3) Écrire un programme dans lequel on demande à l'utilisateur de saisir son date de naissance en gérant l'exception dans le cas où il ne saisit pas une date valide en lui demandant de refaire la saisie.
4) Écrire un programme qui demande à l'utilisateur une date de départ et une date d'arrivée, générer une exception si la date d'arrivée et inférieure à la date de départ.
5) Créer une classe Élèves caractérisée par nom, âge et moyenne.

- L'âge doit être entre 18 et 26 sinon l'exception InvalidAgeException (elle affiche le message "L'âge doit être entre 18 et 26") est générée.
- La note doit être entre 0 et 20 sinon l'exception InvalidNoteException est générée (elle affiche le message "La note doit être entre 0 et 20").

→ Définir les constructeurs de la classe, les accesseurs et les méthodes ToString.
//Source : www.exelib.net : merci à eux