2019-2020

Programmation C++

DR. AMINA JARRAYA
AMINA.JARRAYA.FST@GMAIL.COM

Plan

- **Objets : concepts de base**
- **L a stratégie D.D.U**
- **Gestion des membres de classe**
- **C onstructeurs et destructeurs**
- **L a surcharge**
- **P ointeurs et objets**

Objets : concepts de base

Concepts de base

Un des objectifs principaux de la notion d'objet :

- organiser des programmes complexes grâce aux notions :
  - d'encapsulation
  - d'abstraction
  - d'héritage
  - et de polymorphisme

Notion d'encapsulation

Principe:

- regrouper dans le même objet informatique («concept»), les données et les traitements qui lui sont spécifiques :
- attributs : les données incluses dans un objet
- méthodes : les fonctions définies dans un objet

- ✓ Les objets sont définis par leurs attributs et leurs méthodes.
- ✓ Les attributs et les méthodes sont définis avec le contrôle d'accès (private, public et protected)

Rectangle

Largeur
Hauteur

Surface

Notion d'abstraction (1)

- Pour être véritablement intéressant, un objet doit permettre un certain degré d'abstraction.
- Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :
  - des caractéristiques communes à tous les éléments
  - des mécanismes communs à tous les éléments

✓ description générique de l'ensemble considéré : Se focaliser sur l'essentiel, cacher les détails.

Notion d'abstraction (2)


```
// Dans la programation procédurale :
// On définit autant de données que de rectangles

        double largeur1=2.5;
        double hauteur1=4.0;

        double largeur2=4.5;
        double hauteur2=5.0;

//Pour calculer la surface :
        Surface(largeur1,hauteur1);
        Surface(largeur2,hauteur2);
```

2 → Source d'erreur

Notion d'abstraction (3)

Exemple: Rectangles

- la notion d'"objet rectangle" n'est intéressante que si l'on peut lui associer des propriétés et/ou mécanismes généraux (valables pour l'ensemble des rectangles)
- Les notions de largeur et hauteur sont des propriétés générales des rectangles (attributs),
- Le mécanisme permettant de calculer la surface d'un rectangle (surface = largeur × hauteur) est commun à tous les rectangles (méthodes)

Notion d'abstraction (4)


```
// Dans la programation procédurale :
// On définit autant de données que de rectangles

    double largeur1=2.5;
    double hauteur1=4.0;          rect1

    double largeur2=4.5;
    double hauteur2=5.0;          rect2

//Pour calculer la surface :
    Surface(largeur1,hauteur1);
    Surface(largeur2,hauteur2);
```

Rectangle

hauteur
largeur
Surface

Rect1.Surface();

Rect2.Surface();

La stratégie D.D.U

La stratégie D.D.U (1)

- En C++, la programmation d'une classe se fait en trois phases :
  - ***déclaration, définition, utilisation***

- En abrégé: D.D.U

La stratégie D.D.U (2)

- **Déclaration:** c'est la partie interface de la classe (la partie visible). Elle se fait dans un fichier dont le nom se termine par .h qui va permettre l'utilisation de la classe (et donc sa réutilisation)
- Ce fichier se présente de la façon suivante :

```
class Maclasse
{
  public:
     déclarations des données et fonctions-membres publiques
  private:
     déclarations des données et fonctions-membres privées
};
```

La stratégie D.D.U (3)

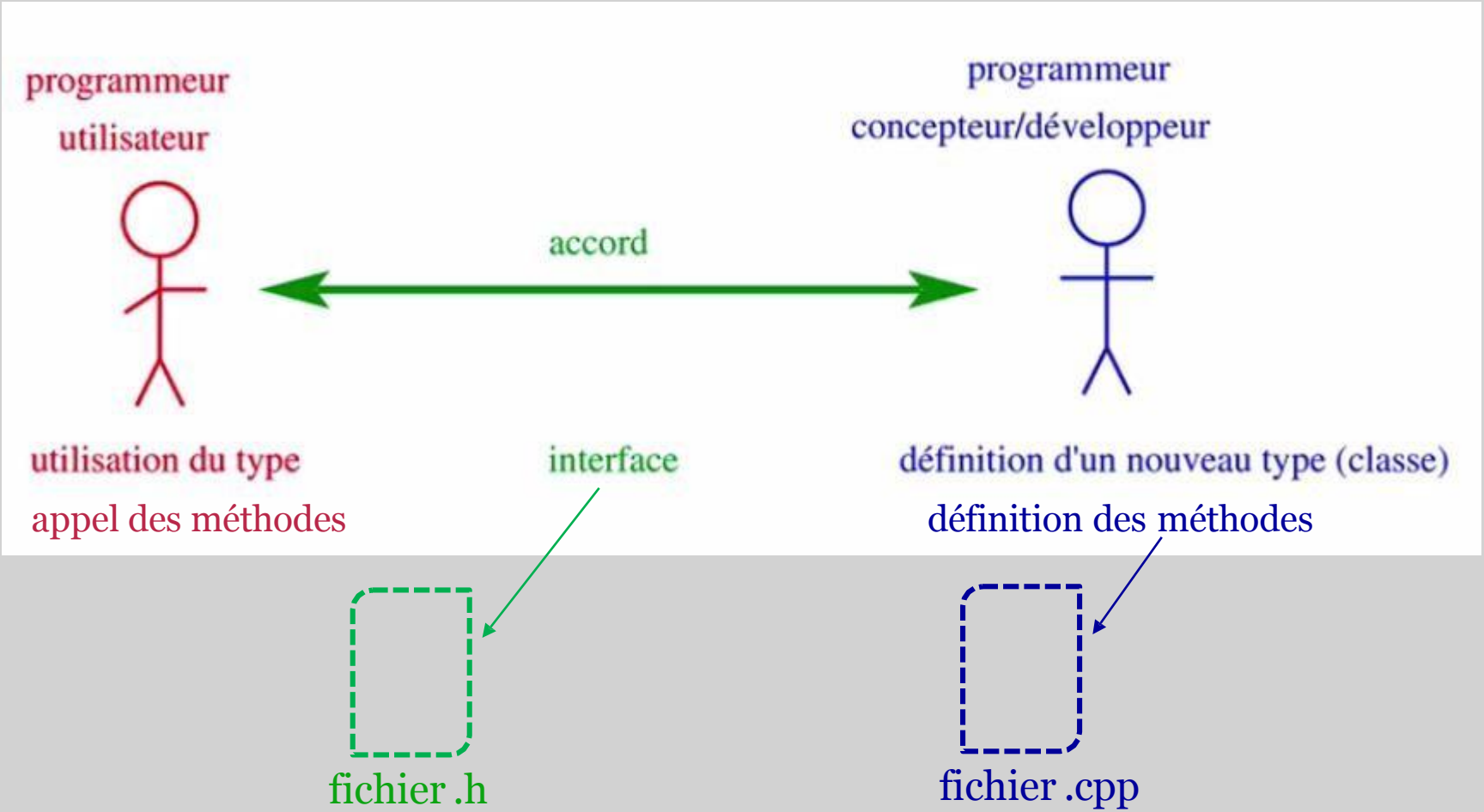
- **Définition:** c'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par .cpp

  Ce fichier contient les définitions des fonctions-membres de la classe, c'est-à-dire le code complet de chaque fonction.

  Elle n'est pas directement nécessaire pour l'utilisateur de la classe. Elle peut être cachée (aux autres).

- **Utilisation:** elle se fait dans un fichier dont le nom se termine par .cpp

La stratégie D.D.U (4)



Structure d'un programme en C++ (1)

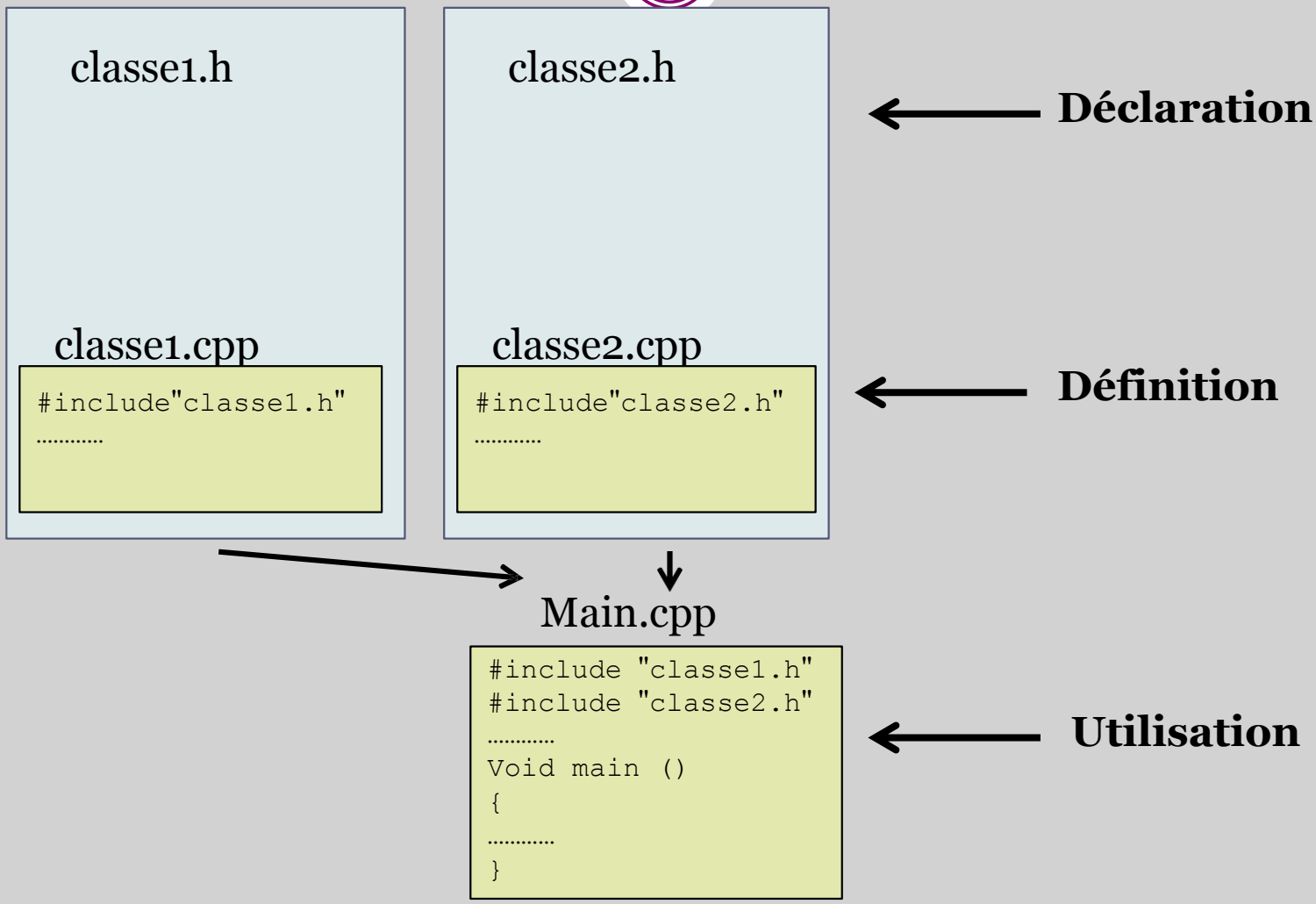
- Nos programmes seront généralement composés d'un nombre impair de fichiers :

  - pour chaque classe:
    - ✓ un fichier **.h** contenant sa déclaration
    - ✓ un fichier **.cpp** contenant sa définition

  - un fichier **.cpp** contenant le traitement principal.

Ce dernier fichier contient la fonction **main**, et c'est par cette fonction que commence l'exécution du programme.

Structure d'un programme en C++ (2)



Mise en œuvre

- Nous donnons ici un programme complet afin d'illustrer les principes exposés.

- Ce programme simule le fonctionnement d'un parcmètre.

Mise en œuvre (2)


```
// ---------------------------- parcmetre.h ------------------------------
  #ifndef PARCMETRE_H
  #define PARCMETRE_H

// ce fichier contient la déclaration de la classe Parcmetre

class parcmetre
{
  public:
    parcmetre(); //constructeur de la classe
    void affiche(); //affichage du temps de stationnement
    void prendsPiece(float valeur);//introduction d'une pièce
  private:
     int heures, minutes; //chiffre des heures et des minutes
};

#endif
```

Mise en œuvre (3)


```
// ------------------ parcmetre.ccp ------------------
  -


// ce fichier contient la définition de la classe Parcmetre

#include <iostream> //pour les entrées-sorties
#include "parcmetre.h"//déclaration de la classe Parcmetre
using namespace std;
parcmetre::parcmetre()//initialisation d'un nouveau
  parcmètre
{
    heures = minutes = 0;
}
```

Mise en œuvre (4)


```
 // Cette fonction permet l'affichage du temps de stationnement
 // restant et du mode d'emploi du parcmètre

void parcmetre::affiche()
{
  cout << "\n\n\tTEMPS DE STATIONNEMENT :";
  cout << heures << "heures" << minutes << "minutes";
  cout << "\n\n\nMode d'emploi du parcmètre:";
  cout << "\n\tPour mettre une pièce de 10 centimes: tapez A";
  cout << "\n\tPour mettre une pièce de 20 centimes: tapez B";
  cout << "\n\tPour mettre une pièce de 50 centimes: tapez C";
  cout << "\n\tPour mettre une pièce de 1 euro: tapez D";
  cout << "\n\tPour quitter le programme: tapez Q";
}
```

Mise en œuvre (5)


```
// Cette fonction permet l'introduction d'une pièce de monnaie
void parcmetre::prendsPiece(float valeur)
{
  minutes += valeur * 10; // 1 euro = 100 minutes de stationnement
  while (minutes >= 60)
    {
      heures += 1;
      minutes -= 60;
    }
  if (heures >= 3) //on ne peut dépasser 3 heures
    {
       heures = 3;
       minutes = 0;
    }
}
```

Mise en œuvre (6)


```
// ----------- simul.cpp ---------
// ce fichier contient l'utilisation
// de la classe Parcmetre

#include <iostream>
#include "parcmetre.h"
using namespace std;

void main() //traitement principal
{
parcmetre p;//déclaration parcmètre p
char choix = 'X';
while (choix != 'Q')
 // boucle principale d'évènements
 {
  p.affiche();
  cout << "\nchoix ? --> ";
  cin >> choix;//lecture d'une lettre
  switch (choix)
  //action correspondante
  {
    case 'A' :
      p.prendsPiece(1);
      break;
    case 'B':
      p.prendsPiece(2);
      break;
    case 'C' :
      p.prendsPiece(5);
      break;
    case 'D' :
      p.prendsPiece(10);
   }
  }
}
```

Quizz (1)

Pour la classe suivante :

```
class Personne { double taille; double poids; };
```

comment définir la méthode calculant l'indice de masse corporelle (IMC) ?

- [ ]
```
double imc(double poids, double taille)
{ return poids / (taille * taille) ; }
```

- [ ]
```
double imc()
{ return poids / (taille * taille) ; }
```

- [ ]
```
double imc(double taille, double poids)
{ return poids / (taille * taille) ; }
```

- [ ]
```
double imc(double taille)
{ return poids / (taille * taille) ; }
```

- [ ]
```
double imc(double poids)
{ return poids / (taille * taille) ; }
```

Quizz(2)

Pour la classe suivante :

```
class Contribuable { double fortune; };
```

comment définir la méthode calculant l'impôt à payer pour un certain taux d'imposition donné ?

- [ ] `double impot(double taux) { return taux * fortune; }`
- [ ] `double impot(double taux, double fortune) { return taux * fortune; }`
- [ ] `double impot(double fortune, double taux) { return taux * fortune; }`
- [ ] `double impot(double fortune) { return taux * fortune; }`
- [ ] `double impot() { return taux * fortune; }`

Gestion des membres de classe

Les opérateurs . et ::

- Instantier une classe ➔ `Ma_classe Mon_obj;`
- Dans une expression, on accède aux données et fonctions-membres d'un objet grâce à la notation pointée
  - Si *donnee* figure dans la déclaration de *Ma_classe* et si mon objet est une instance de *Ma_classe*, on écrit mon `Mon_obj.donnee;`
- Dans la définition d'une fonction-membre, on doit ajouter *<nom de la classe>::* devant le nom de la fonction.
  - Exemple: la définition d'une fonction-membre *truc()* de la Ma_classe aura la forme suivante :

```
<type> Ma_classe::truc(<déclaration de paramètres formels>)
<instruction-bloc>
```

  - L'appel se fait avec la notation pointée :

```
Mon_obj.truc() ;
```

Actions et Prédicats

- En C++, on peut distinguer les méthodes qui modifient l'état de l'objet (« actions ») de celles qui ne changent rien à l'objet (« prédicats »).
- On peut pour cela ajouter le mot **const** après la liste des paramètres de la méthode :

```
Type_retour nom_methode (typ_par1 nom_par1,...) const
```

```
class Rectangle {
  // ...
  double surface() const;
};

double Rectangle::surface() const
{
    return hauteur * largeur;
}
```

- Si on déclare une action en tant que prédicat (const), on aura un message d'erreur à la compilation!

La portée des membres (1)

- Tout ce qui n'est pas nécessaire de connaître à l'extérieur d'un objet devrait être dans le corps de l'objet et identifié par le mot clé private : c'est la notion de portée
- Donc les données et fonctions private ne sont pas accessibles à l'extérieur de la classe.
- Note : Si aucun droit d'accès n'est précisé, c'est private par défaut.
- Donc, dans notre exemple quelle est la portée de la fonction surface() ?

```
class Rectangle {
  double surface() const;
private:
  double hauteur;
  double largeur;
};
```

La portée des membres (2)

- À l'inverse, l'interface, qui est accessible de l'extérieur, se déclare avec le mot-clé public.
- public : Données et fonctions utilisables par d'autres objets et fonctions.

```
class Rectangle {
public:
  double surface() const;
private:
  // ...
};
```

- Dans la plupart des cas :
  - Privé :
    - Tous les attributs
    - La plupart des méthodes
  - Public :
    - Quelques méthodes bien choisies (interface)

Accesseurs et manipulateurs (1)

- Si le programmeur le juge utile, il inclut les méthodes publiques:
  - Accesseurs (getters):
    - Consultation (prédicat)
    - Retour de la valeur d'une variable d'instance précise
  - Mutateurs (setters):
    - Modification (Action)
    - Affectation de l'argument à une variable d'instance précise

Accesseurs et manipulateurs (2)

- Accesseurs

```
double getHauteur() const {return hauteur};
double getLargeur() const {return largeur};
```

- Manipulateurs

```
void setHauteur(double h) {hauteur = h};
void setLargeur(double l) {largeur = l};
```

Le pointeur this

- comment les fonctions membres, qui appartiennent à la classe, peuvent accéder aux données d'un objet, qui est une instance de cette classe ?

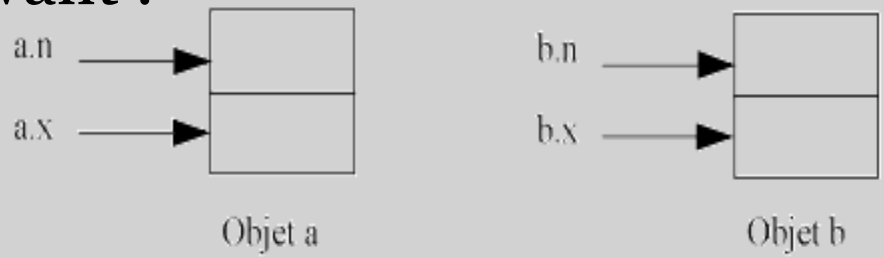
  - À chaque appel d'une fonction membre, le compilateur passe implicitement un pointeur sur les données de l'objet en paramètre.
  - Le pointeur sur l'objet est accessible à l'intérieur de la fonction membre. Il porte le nom « *this* »
  - **this* représente l'objet lui-même. Fait référence à l'objet pour lequel une fonction membre a été appelé
  - Dans une fonction non-static, le mot clé *this* est un pointeur sur l'objet pour lequel la fonction a été appelée.

Les membres données statiques (1)

- A chaque déclaration d'une instance, celle-ci possède ses propres membres données.
- Exemple :

```
class exple1
{      int n ;
      float x ;
      .....
} ;
```

- une déclaration telle que: exple1 a, b ; Conduira au schéma suivant :

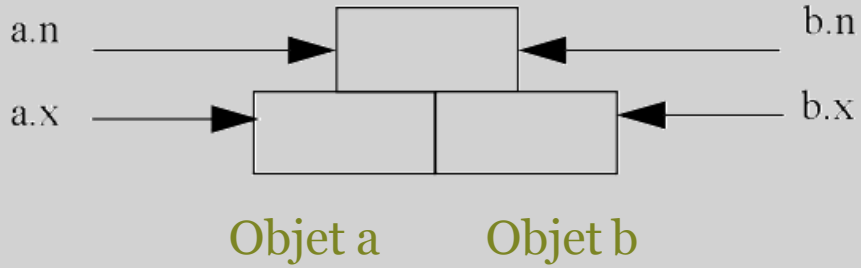


Les membres données statiques (2)

Question : Est-ce qu'il existe des cas ou on a besoin d'une variable membre commune à tous les objets ?

- Un membre static est un membre commun à tous les objets de la classe.
- Le qualificatif static permet de **définir** un membre de donnée static:
- Exemple :

```
class exple2{
   public:
        static  int n;
   private:
        float x;
     ..... } ;
```



- Avec la déclaration: exple2 a, b ;

Un membre statique est accessible via la classe : class::varstat;

- Exemple : `exple2::n`

Initialisation des membres données statiques

- Un membre statique doit être initialisé explicitement (à l'extérieur de la déclaration de la classe) par une instruction telle que :

```
int exple2::n = 5;
```

- Cette démarche est utilisable aussi bien pour les membres statiques privés que publics.

Les méthodes statiques

- Similairement, si on ajoute static à une méthode, on peut accéder aussi à la méthode sans objet, à partir le nom de la classe et l'opérateur de résolution de portée ::

```
class A {
public:
  static void methode1() { cout << "Méthode 1" << endl; }
         void methode2() { cout << "Méthode 2" << endl; }
};

int main () {
    A::methode1(); // OK
    A::methode2(); // ERREUR
    A x;
    x.methode1();  // OK
    x.methode2();  // OK
}
```

Les membres statiques

Il est préférable de toujours désigner les variables membre statiques par leurs noms complets, pour éviter tout risque de malentendu quant à la nature de ces variables!

Quiz

Quelle est la bonne sortie :

a)0 0

b) 5 0

c) 0 5

d) 5 5

```
#include <iostream>
using namespace std;
class Test
{
    static int x;
  public:
    Test() { x++; }
    static int getX() {return x;}
};
int Test::x = 0;
int main()
{
    cout << Test::getX() << " ";
    Test t[5];
    cout << Test::getX();
}
```

Quiz

Quelle est la bonne sortie :

a) 1 2 3

b) 2 2 2

c) 1 3 1

d) 1 1 1

```
#include <iostream>
using namespace std;
class Player
{
  private:
    int id;
    static int next_id;
  public:
    int getID() { return id; }
    Player()  {  id = next_id++; }
};
int Player::next_id = 1;
int main()
{
  Player p1;
  Player p2;
  Player p3;
  cout << p1.getID() << " ";
  cout << p2.getID() << " ";
  cout << p3.getID();
  return 0;
}
```