Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://clarolineconnect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Utiliser l'opérateur new avec un constructeur

Avec les structures, pour faire une allocation dynamique on procède comme suit :

```
Type * pT = new Type ;
```

Avec les classes, on utilise un constructeur :

```
CType * pT = new CType ; : appel au constructeur sans argument
CType * pT = new CType() ; : appel au constructeur sans argument
CType * pT = new CType(5) ; : appel à un constructeur avec 1 argument de type int
```

Utiliser l'opérateur new avec un constructeur

Avec les structures, pour faire une allocation dynamique on procède comme suit :
Type * pT = **new** Type ;

Avec les classes, on utilise un constructeur :
CType * pT = **new** CType ; : appel au constructeur sans argument
CType * pT = **new** CType() ; : appel au constructeur sans argument
CType * pT = **new** CType(5) ; : appel à un constructeur avec 1 argument de type **int**

Application

```
class CPersonne {
    string m_nom, m_prenom ; // attributs privés (par défaut private)
    int m_age ;
  public :
    CPersonne(string nom="", string prenom="", int age=-1) ;
    ~CPersonne() ;
};

CPersonne::CPersonne(string nom, string prenom, int age){
    cout << " Constructeur de CPersonne " << endl ;
    m_nom = nom; m_prenom = prenom; m_age = age ;
}
CPersonne::~CPersonne(){ cout << " Destructeur de CPersonne " << endl ; }

int main() {
    CPersonne* pPers = new CPersonne("A", "Bob", 36);// le constructeur est appelé
                                                     // après l'allocation effective
    ...

    delete pPers;// le destructeur de la classe CPersonne est
                 // est appelé avant la libération effective
                 // de la mémoire
    return EXIT_SUCCESS;
}
```

Nouvelle syntaxe pour les constructeurs

```
class CPoint2D
{
    float m_x, m_y;
  public :
              // initialisation des membres dans leur ordre de déclaration
              // il n'est pas obligatoire d'initialiser tous les membres
    CPoint2D() : m_x(0), m_y(0) {}

    CPoint2D(float x, float y) : m_x(x), m_y(y) {}
};
class CSegment2D
{
    CPoint2D m_p1, m_p2; // composition (on aurait du faire de une agrégation)
  public :
    CSegment2D(float p1_x, float p1_y, float p2_x, float p2_y) : m_p1(p1_x, p1_y), m_p2(p2_x,
      p2_y) {}
};
```

Syntaxe type scalaire : : m_membre(*exp*)
Syntaxe type classe : : m_membre([param effectifs constructeur]).

Nouvelle syntaxe pour les constructeurs

Cette syntaxe est l'équivalent d'une initialisation à la déclaration d'une variable ou d'un objet. *A quoi sert cette possibilité ?*

Elle permet de pouvoir initialiser :

- un *objet attribut* de la classe via son constructeur ;
- un *membre donnée constant* (e.g. `const int m_mon_id ;`, `const float m_PI ;` etc.) ;
- un membre donnée qui est une *référence sur un type* (e.g. `int& m_ma_ref ;`).

Nouvelle syntaxe pour les constructeurs

Cette syntaxe est l'équivalent d'une initialisation à la déclaration d'une variable ou d'un objet. *A quoi sert cette possibilité ?*

Elle permet de pouvoir initialiser :

- un *objet attribut* de la classe via son constructeur ;
- un *membre donnée constant* (e.g. `const int m_mon_id ;`, `const float m_PI ;` etc.) ;
- un membre donnée qui est une *référence sur un type* (e.g. `int& m_ma_ref ;`).

Nouvelle syntaxe pour les constructeurs

Cette syntaxe est l'équivalent d'une initialisation à la déclaration d'une variable ou d'un objet. *A quoi sert cette possibilité ?*

Elle permet de pouvoir initialiser :

- un *objet attribut* de la classe via son constructeur ;
- un *membre donnée constant* (e.g. **const int** m_mon_id ;, **const float** m_PI ; etc.) ;
- un membre donnée qui est une *référence sur un type* (e.g. **int&** m_ma_ref ;).

Nouvelle syntaxe pour les constructeurs

Cette syntaxe est l'équivalent d'une initialisation à la déclaration d'une variable ou d'un objet. *A quoi sert cette possibilité ?*

Elle permet de pouvoir initialiser :

- un *objet attribut* de la classe via son constructeur ;
- un *membre donnée constant* (e.g. **const int** m_mon_id ;, **const float** m_PI ; etc.) ;
- un membre donnée qui est une *référence sur un type* (e.g. **int&** m_ma_ref ;).

Nouvelle syntaxe pour les constructeurs

```
#include <iostream>
class CMath
{
    const float m_PI;
    double& m_ref_sur_var_exter;
  public :
    CMath(double& d) : m_PI(3.14159), m_ref_sur_var_exter(d) {}

    void met_au_carre() { m_ref_sur_var_exter = m_ref_sur_var_exter*m_ref_sur_var_exter; }
};

int main()
{
    double val_tmp = 2.;
    CMath m(val_tmp);
    m.met_au_carre();
    std::cout << " Valeur courante = " << val_tmp << std::endl;
    return EXIT_SUCCESS;
}
```

Notations

```
#include <iostream>
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D() : m_x(0.f), m_y(0.f) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};

int main()
{
    CPoint2D courbe2D[15]; // tableau de 15 points 2D
                           // Quel constructeur est appelé pour
                           // construire chaque CPoint2D?
    ...
    courbe2D[4].affiche();

    return EXIT_SUCCESS;
}
```

Remarque : en C++, un tableau d'objets n'est pas un objet.

Notations

```
#include <iostream>
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D() : m_x(0.f), m_y(0.f) {}
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y) {}
    void affiche() { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};

int main()
{
    CPoint2D courbe2D[15]; // tableau de 15 points 2D
                           // Quel constructeur est appelé pour
                           // construire chaque CPoint2D?
    ...
    courbe2D[4].affiche();

    return EXIT_SUCCESS;
}
```

Remarque : en C++, un tableau d'objets n'est pas un objet.

Constructeurs et initialisation

Si la classe CPoint2D comporte un constructeur sans argument, celui-ci est appelé sur les 15 éléments du tableau courbe2D. Sinon il y aura une erreur de compilation.

Comment initialiser une case d'un tableau avec un autre constructeur que le constructeur sans argument ?

Constructeurs et initialisation

Si la classe CPoint2D comporte un constructeur sans argument, celui-ci est appelé sur les 15 éléments du tableau courbe2D. Sinon il y aura une erreur de compilation.

Comment initialiser une case d'un tableau avec un autre constructeur que le constructeur sans argument ?

Constructeurs et initialisation

```
#include <iostream>
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D() : m_x(0.f), m_y(0.f)
    { std::cout << " Appel construct sans argument " << std::endl; }
    CPoint2D(float x, float y=0) : m_x(x), m_y(y)
    { std::cout << " Appel construct avec au min 1 arg " << std::endl; }
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y)
    { std::cout << " Appel construct par recopie " << std::endl; }
    void affiche() { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};

int main()
{
    const int TAILLE = 5;
    CPoint2D courbe2D[TAILLE] = {2, CPoint2D(3, 4), 5};
    for(int i=0; i<TAILLE; i++)
    {
        courbe2D[i].affiche();
        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

Constructeurs et initialisation : résultat de l'exécution

```
 Appel construct avec au min 1 arg
 Appel construct avec au min 1 arg
 Appel construct avec au min 1 arg
 Appel construct sans argument
 Appel construct sans argument
(2, 0)

(3, 4)

(5, 0)

(0, 0)

(0, 0)
```

Tableaux dynamiques d'objets

```
#include <iostream>
class CPoint2D
{
    float m_x, m_y;
  public :
    CPoint2D() : m_x(0.f), m_y(0.f)
    { std::cout << " Appel construct sans argument " << std::endl; }
    CPoint2D(float x, float y=0) : m_x(x), m_y(y)
    { std::cout << " Appel construct avec au min 1 arg " << std::endl; }
    CPoint2D(const CPoint2D& p) : m_x(p.m_x), m_y(p.m_y)
    { std::cout << " Appel construct par recopie " << std::endl; }
    void affiche() { std::cout << "(" << m_x << ", " << m_y << ")" << std::endl; }
};
int main()
{
    const int TAILLE = 5 ;
    CPoint2D* dynCourbe2D = new CPoint2D[TAILLE] ; // init possible SEULEMENT par constr. défaut
    for(int i=0; i<TAILLE; i++)
    {
        dynCourbe2D[i].affiche();
        std::cout << std::endl;
    }
    delete [] dynCourbe2D ; // Attention au []
    return EXIT_SUCCESS;
}
```

Plan

Les propriétés du C++ qui s'appliquent aux fonctions membres d'une classe

- **Surdéfinition/surcharge** : OK si même règle d'accessibilité (public ou private) ; que se passe-t-il si l'accessibilité est différente ? *Le statut privé ou public d'une fonction n'intervient pas dans le choix du compilateur* ;
- **Arguments par défauts** : permet de diminuer le nombre de surdéfinitions d'une fonction ;
- **Transmission par référence** ;
- **Fonction en ligne** (inline).

Les propriétés du C++ qui s'appliquent aux fonctions membres d'une classe

- **Surdéfinition/surcharge** : OK si même règle d'accessibilité (public ou private) ; que se passe-t-il si l'accessibilité est différente ? *Le statut privé ou public d'une fonction n'intervient pas dans le choix du compilateur* ;
- **Arguments par défauts** : permet de diminuer le nombre de surdéfinitions d'une fonction ;
- **Transmission par référence** ;
- **Fonction en ligne** (inline).

Les propriétés du C++ qui s'appliquent aux fonctions membres d'une classe

- **Surdéfinition/surcharge** : OK si même règle d'accessibilité (public ou private) ; que se passe-t-il si l'accessibilité est différente ? *Le statut privé ou public d'une fonction n'intervient pas dans le choix du compilateur* ;
- **Arguments par défauts** : permet de diminuer le nombre de surdéfinitions d'une fonction ;
- **Transmission par référence** ;
- **Fonction en ligne** (inline).

Les propriétés du C++ qui s'appliquent aux fonctions membres d'une classe

- **Surdéfinition/surcharge** : OK si même règle d'accessibilité (public ou private) ; que se passe-t-il si l'accessibilité est différente ? *Le statut privé ou public d'une fonction n'intervient pas dans le choix du compilateur* ;
- **Arguments par défauts** : permet de diminuer le nombre de surdéfinitions d'une fonction ;
- **Transmission par référence** ;
- **Fonction en ligne** (inline).

Les propriétés du C++ qui s'appliquent aux fonctions membres d'une classe

- **Surdéfinition/surcharge** : OK si même règle d'accessibilité (public ou private) ; que se passe-t-il si l'accessibilité est différente ? *Le statut privé ou public d'une fonction n'intervient pas dans le choix du compilateur* ;
- **Arguments par défauts** : permet de diminuer le nombre de surdéfinitions d'une fonction ;
- **Transmission par référence** ;
- **Fonction en ligne** (inline).

Les méthodes en ligne

Il y a deux façons de rendre "en ligne" une fonction membre ou méthode d'une classe :

- **Définir directement** la méthode dans la déclaration de la classe.
- **Précéder la *déclaration* et la *définition* de la méthode du qualificatif** `inline` ; **attention**, *la définition d'une fonction en ligne doit être connue du compilateur à la compilation*, ce qui nécessite la définition de la fonction en ligne dans le .h/.hpp à la suite de la déclaration de la classe.

Généralement la première approche est utilisée, la définition étant écrite sur une seule ligne.

Les méthodes en ligne

Il y a deux façons de rendre "en ligne" une fonction membre ou méthode d'une classe :

- **Définir directement** la méthode dans la déclaration de la classe.
- **Précéder la *déclaration* et la *définition* de la méthode du qualificatif** `inline` ; **attention**, *la définition d'une fonction en ligne doit être connue du compilateur à la compilation*, ce qui nécessite la définition de la fonction en ligne dans le .h/.hpp à la suite de la déclaration de la classe.

Généralement la première approche est utilisée, la définition étant écrite sur une seule ligne.

Les méthodes en ligne

Il y a deux façons de rendre "en ligne" une fonction membre ou méthode d'une classe :

- **Définir directement** la méthode dans la déclaration de la classe.
- **Précéder la *déclaration* et la *définition* de la méthode du qualificatif** `inline` ; **attention**, *la définition d'une fonction en ligne doit être connue du compilateur à la compilation*, ce qui nécessite la définition de la fonction en ligne dans le .h/.hpp à la suite de la déclaration de la classe.

Généralement la première approche est utilisée, la définition étant écrite sur une seule ligne.

Les méthodes en ligne

Il y a deux façons de rendre "en ligne" une fonction membre ou méthode d'une classe :

- **Définir directement** la méthode dans la déclaration de la classe.
- **Précéder la *déclaration* et la *définition* de la méthode du qualificatif** `inline` ; **attention**, *la définition d'une fonction en ligne doit être connue du compilateur à la compilation*, ce qui nécessite la définition de la fonction en ligne dans le .h/.hpp à la suite de la déclaration de la classe.

Généralement la première approche est utilisée, la définition étant écrite sur une seule ligne.

Exemple de méthodes en ligne

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"
    point2D(float x, float y) { m_x = x; m_y = y; } // constructeur "en ligne"

    // pas de destructeur

    // méthodes
    inline void affiche() ;
};

inline void point2D::affiche(){
   std::cout << " (" << m_x << ", " << m_y << ")" << std::endl;
}
```

L'unité de l'encapsulation est la classe et non l'objet

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"
    point2D(float x, float y) { m_x = x; m_y = y; } // constructeur "en ligne"

    // pas de destructeur

    // méthodes
    inline void affiche() ;

    // j'ai le droit d'accéder aux champs m_x et m_y de p car je suis dans une méthode
    // de la classe point2D
    bool est_identique(point2D p){ return (m_x==p.m_x && m_y==p.m_y); }
};

inline void point2D::affiche(){
   std::cout << " (" << m_x << ", " << m_y << ")" << std::endl;
}
```

3 modes de transmission des objets

- Transmission **par valeur** :
  `void maFonct(TypeObjet o);` : *tous les champs sont copiés* : attention aux données membres dynamiques : aucune allocation dynamique ne sera réalisée par le contructeur de recopie par défaut ! (attributs dynamiques croisés)
- Transmission **de l'adresse d'un objet** :
  `void maFonct(TypeObjet* pt_o);` : *copie de l'adresse de l'emplacement mémoire de l'objet*, la seule contrainte est de passer l'adresse de l'objet à l'appel de maFonct : `maFonct(&o);` l'objet o pourra être modifié par maFonct.
- Transmission **par référence** : `void maFonct(TypeObjet& o);` : transmission par adresse "cachée à l'utilisateur" ; l'appel est simplifié : `maFonct(o);` l'objet o pourra être modifié par maFonct.

3 modes de transmission des objets

- Transmission **par valeur** :
  `void maFonct(TypeObjet o);` : *tous les champs sont copiés* : attention aux données membres dynamiques : aucune allocation dynamique ne sera réalisée par le contructeur de recopie par défaut ! (attributs dynamiques croisés)
- Transmission **de l'adresse d'un objet** :
  `void maFonct(TypeObjet* pt_o);` : *copie de l'adresse de l'emplacement mémoire de l'objet*, la seule contrainte est de passer l'adresse de l'objet à l'appel de maFonct : `maFonct(&o);` l'objet o pourra être modifié par maFonct.
- Transmission **par référence** : `void maFonct(TypeObjet& o);` : transmission par adresse "cachée à l'utilisateur" ; l'appel est simplifié : `maFonct(o);` l'objet o pourra être modifié par maFonct.

3 modes de transmission des objets

- Transmission **par valeur** :
  `void maFonct(TypeObjet o);` : *tous les champs sont copiés* : attention aux données membres dynamiques : aucune allocation dynamique ne sera réalisée par le contructeur de recopie par défaut ! (attributs dynamiques croisés)
- Transmission **de l'adresse d'un objet** :
  `void maFonct(TypeObjet* pt_o);` : *copie de l'adresse de l'emplacement mémoire de l'objet*, la seule contrainte est de passer l'adresse de l'objet à l'appel de maFonct : `maFonct(& o);` l'objet o pourra être modifié par maFonct.
- Transmission **par référence** : `void maFonct(TypeObjet& o);` : transmission par adresse "cachée à l'utilisateur" ; l'appel est simplifié : `maFonct(o);` l'objet o pourra être modifié par maFonct.

3 modes de transmission des objets

On peut **interdire la possibilité de la modification de l'objet dans les cas de la transmission de son adresse ou du passage par référence** en précédant le type de l'argument formel de la fonction du qualificatif **const** :

- `void maFonct(const Objet* pt_o);`
- `void maFonct(const Objet& o);`

2 modes de transmission des objets à préférer

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"
    point2D(float x, float y) { m_x = x; m_y = y; } // constructeur "en ligne"

    // méthodes
    inline void affiche() ;

    // solutions à retenir si l'objet argument ne doit pas être modifié :
    bool est_identique(const point2D& p){ return (m_x==p.m_x && m_y==p.m_y); }
    bool est_identique(const point2D* pt_p){ return (m_x==pt_p->m_x && m_y==pt_p->m_y); }

    // solutions à retenir si l'objet argument doit être modifié :
    void modifie(point2D& p){ p.m_x = 2*m_x; p.m_y = 2*m_y; }
    void modifie(point2D* p){ p->m_x = 2*m_x; p->m_y = 2*m_y; }
};

inline void point2D::affiche(){
   std::cout << " (" << m_x << ", " << m_y << ")" << std::endl;
}
```

Et lorsqu'une fonction retourne un objet ?

- Ne jamais retourner une référence sur un objet local à la méthode ;
- Ne jamais retourner une adresse d'un objet local à la méthode ;
- Ne jamais retourner une copie (transmission par valeur) d'un objet ayant des membres dynamiques, sauf si vous avez réécrit vous même le constructeur par recopie de la forme `maclasse(const maclasse&)` ou `maclasse(maclasse&)`.

Et lorsqu'une fonction retourne un objet ?

- Ne jamais retourner une référence sur un objet local à la méthode ;
- Ne jamais retourner une adresse d'un objet local à la méthode ;
- Ne jamais retourner une copie (transmission par valeur) d'un objet ayant des membres dynamiques, sauf si vous avez réécrit vous même le constructeur par recopie de la forme `maclasse(const maclasse&)` ou `maclasse(maclasse&)`.

Et lorsqu'une fonction retourne un objet ?

- Ne jamais retourner une référence sur un objet local à la méthode ;
- Ne jamais retourner une adresse d'un objet local à la méthode ;
- Ne jamais retourner une copie (transmission par valeur) d'un objet ayant des membres dynamiques, sauf si vous avez réécrit vous même le constructeur par recopie de la forme maclasse(**const** maclasse&) ou maclasse(maclasse&).

3 types de fonction membre

- Les **fonctions membres d'instance** (sans mot clef particulier) : leur appel est *dépendant* de la création d'un objet.
- Les **fonctions membres statiques** (méthodes de classe) : leur déclaration est précédée du mot clef **static** : leur appel est *indépendant* de la création d'un objet et **elles ne peuvent agir que sur des membres données statiques**.
- Les **fonctions membres constantes** : leur déclaration et l'en-tête de leur définition **sont suivies** du qualificatif **const** : elles ne peuvent pas modifier les attributs non-statiques (sinon erreur de compilation) ; elles peuvent modifier des attributs statiques ; **les fonctions membres constantes sont appelables par des objets constants**.

3 types de fonction membre

- Les **fonctions membres d'instance** (sans mot clef particulier) : leur appel est *dépendant* de la création d'un objet.
- Les **fonctions membres statiques** (méthodes de classe) : leur déclaration est précédée du mot clef **static** : leur appel est *indépendant* de la création d'un objet et elles ne peuvent agir que sur des membres données statiques.
- Les **fonctions membres constantes** : leur déclaration et l'en-tête de leur définition **sont suivies** du qualificatif **const** : elles ne peuvent pas modifier les attributs non-statiques (sinon erreur de compilation) ; elles peuvent modifier des attributs statiques ; **les fonctions membres constantes sont appelables par des objets constants**.

3 types de fonction membre

- Les **fonctions membres d'instance** (sans mot clef particulier) : leur appel est *dépendant* de la création d'un objet.
- Les **fonctions membres statiques** (méthodes de classe) : leur déclaration est précédée du mot clef **static** : leur appel est *indépendant* de la création d'un objet et elles ne peuvent agir que sur des membres données statiques.
- Les **fonctions membres constantes** : leur déclaration et l'en-tête de leur définition **sont suivies** du qualificatif **const** : elles ne peuvent pas modifier les attributs non-statiques (sinon erreur de compilation) ; elles peuvent modifier des attributs statiques ; **les fonctions membres constantes sont appelables par des objets constants**.

Revenons sur un membre static d'une classe

```
class compte_obj
{
  private :
    static int m_cpt ; // compteur d'objets crées
  public :
    compte_obj() ;     // constructeur
    ~compte_obj() ;    // destructeur
    void affiche() ;
};

int compte_obj::m_cpt = 0; // initialisation globale du compteur
compte_obj::compte_obj() { m_cpt++; }
compte_obj::~compte_obj() { m_cpt--; }

// la méthode affiche nécessite l'existance d'un objet pour être invoquée
void compte_obj::affiche() {
   std::cout << " Il y a actuellement " << m_cpt << " instances de la classe." << std::endl;
}
```

Comment accéder au membre static m_cpt s'il n'existe pas d'instance de la classe compte_obj ?

Revenons sur un membre static d'une classe

```
class compte_obj
{
  private :
    static int m_cpt ; // compteur d'objets crées
  public :
    compte_obj() ;     // constructeur
    ~compte_obj() ;    // destructeur
    void affiche() ;
};

int compte_obj::m_cpt = 0; // initialisation globale du compteur
compte_obj::compte_obj() { m_cpt++; }
compte_obj::~compte_obj() { m_cpt--; }

// la méthode affiche nécessite l'existance d'un objet pour être invoquée
void compte_obj::affiche() {
   std::cout << " Il y a actuellement " << m_cpt << " instances de la classe." << std::endl;
}
```

Comment accéder au membre static m_cpt **s'il n'existe pas d'instance de la classe** compte_obj **?**

2 solutions à notre problème

- On accède directement au membre static via l'opérateur résolution de portée : : : compte_obj : :m_cpt.
- On accède au membre static grâce à une méthode static via l'opérateur résolution de portée : : : compte_obj : :affiche()

```
class compte_obj
{
  private :
    static int m_cpt ; // compteur d'objets crées
  public :
    compte_obj() ;      // constructeur
    ~compte_obj() ;     // destructeur
    static void affiche() ;
};

int compte_obj::m_cpt = 0; // initialisation globale du compteur
compte_obj::compte_obj() { m_cpt++; }
compte_obj::~compte_obj() { m_cpt--; }
// la méthode affiche ne nécessite plus l'existance d'un objet pour être invoquée
void compte_obj::affiche() {
   std::cout << " Il y a actuellement " << m_cpt << " instances de la classe." << std::endl;
}
```

2 solutions à notre problème

- On accède directement au membre static via l'opérateur résolution de portée : : : compte_obj : :m_cpt.
- On accède au membre static grâce à une méthode static via l'opérateur résolution de portée : : : compte_obj : :affiche()

```
class compte_obj
{
  private :
    static int m_cpt ; // compteur d'objets crées
  public :
    compte_obj() ;     // constructeur
    ~compte_obj() ;    // destructeur
    static void affiche() ;
};

int compte_obj::m_cpt = 0; // initialisation globale du compteur
compte_obj::compte_obj() { m_cpt++; }
compte_obj::~compte_obj() { m_cpt--; }
// la méthode affiche ne nécessite plus l'existance d'un objet pour être invoquée
void compte_obj::affiche() {
   std::cout << " Il y a actuellement " << m_cpt << " instances de la classe." << std::endl;
}
```

Exemple

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"
    point2D(float x, float y) { m_x = x; m_y = y; } // constructeur "en ligne"

    // méthodes
    void affiche() const; // les attributs propres à l'instance ne peuvent
                          // pas être modifiés par affiche()
                          // intérêt? affiche peut être appelée par un objet constant
    bool est_identique(const point2D& p) const { return (m_x==p.m_x && m_y==p.m_y); }
    bool est_identique(const point2D* pt_p) const { return (m_x==pt_p->m_x && m_y==pt_p->m_y); }
};
void point2D::affiche() const {
   std::cout << " (" << m_x << ", " << m_y << ")" << std::endl;
}
int main(){
    const point2D p(5,7);
    p.affiche(); // OK car affiche est une méthode constante
    return 0;
}
```

Le mot clé this

Chaque méthode d'instance (non-statique) peut accéder à l'adresse de l'objet courant grâce au mot clef *this*.

Notez bien la différence entre le *this* du Java (référence) et le *this* du C++ (adresse/pointeur).

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"

    // méthodes
    inline void affiche() const;
};

inline void point2D::affiche() const {
   std::cout << " ad = " << this << " ; (" << this->m_x << ", " << this->m_y << ")" << std::endl;
}
```

Le mot clé this

Chaque méthode d'instance (non-statique) peut accéder à l'adresse de l'objet courant grâce au mot clef *this*.
Notez bien la différence entre le *this* du Java (référence) et le *this* du C++ (adresse/pointeur).

```
class point2D
{
  private :
    float m_x, m_y ;

  public :
    // constructeurs
    point2D() { m_x = m_y = 0.0 ; } // constructeur sans argument "en ligne"

    // méthodes
    inline void affiche() const;
};

inline void point2D::affiche() const {
   std::cout << " ad = " << this << " ; (" << this->m_x << ", " << this->m_y << ")" << std::endl;
}
```