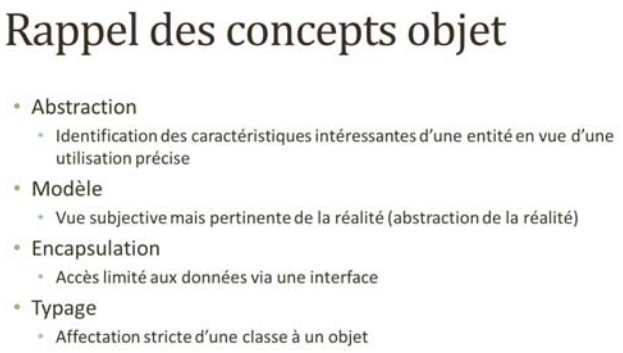
Chapitre 3. Programmation Orientée Objet en C++

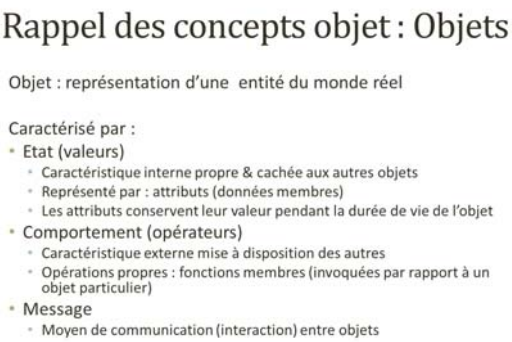
Quelques rappels sur les concepts objets



Slide 2

L'abstraction est un concept consistant à ignorer les détails pour ne conserver qu'une notion générale
L'encapsulation consiste à proposer le code et les données dans une même entité, mais à garder les détails d'implémentation cachés.
Le typage permet de préciser les comportements communs à un ensemble d'entités.

Notion d'objet



Slide 3

Un objet est la représentation d'une entité du monde réel.
Son état est représenté par les valeurs prises par une ou plusieurs variables
Son comportement est représenté par les opérateurs applicables à l'objet (i.e. aux variables représentant son état).

Un message peut, par exemple, être une requête envoyée à un objet pour demander l'exécution d'une méthode.

Notion de Classe

Rappel des concepts objet : Classes

- Modèle pour une catégorie d'objets structurellement identiques
- Chaque objet est une instance d'une classe
- Une classe peut contrôler l'accès à ses membres
- Différents types de méthodes :
  - Accesseur (consultent l'état)
  - Mutateur (modifient l'état)
  - Constructeur (initialisent, appel automatique à la création)
  - Destructeur (libération des ressources, appel automatique à la destruction)

Slide 4

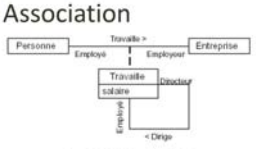
Une classe est un modèle de données ET un mécanisme pour la création d'objets basés sur ce modèle.
Une classe peut contrôler l'accès à ses membres qui sont des données et/ou des fonctions.

Les différents types de méthodes :

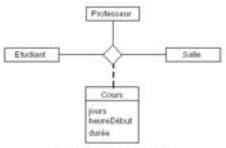
- Les méthodes de type *accesseur* consultent l'état d'un objet
- Les méthodes de type *mutateur* modifient l'état d'un objet
- Les méthodes de type *constructeur* initialisent un objet (instance de classe), fixent l'invariant de la classe et sont appelées automatiquement à la création d'une instance de l'objet.
- Les méthodes de type *destructeur* libèrent les ressources mobilisées lors de l'instanciation d'un objet et sont appelées automatiquement à la destruction de l'objet.

Rappel des concepts objet : Classes

- Association



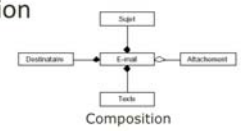
Association binaire



Association n-aire

- Agrégation et Composition

Agrégation

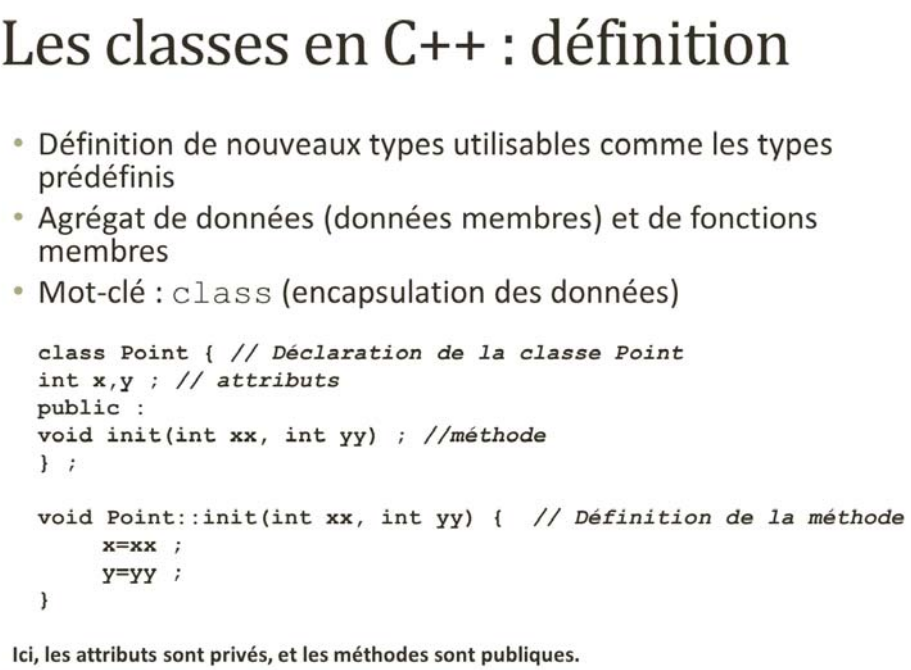


Composition

Slide 5

Les associations matérialisent les interactions possibles entre les classes, tandis que les agrégations et compositions représentent une modélisation d'objets possible.
Les premières vont se matérialiser par des méthodes incluant des paramètres issus des classes concernées, tandis que les secondes pourront se traduire dans le type des paramètres des classes.

Définition des classes en C++



Slide 6

Les données membres d'une classe sont aussi appelées des attributs et les fonctions membres sont aussi appelées des méthodes membres.
Déclarer une classe en utilisant *class* au lieu de *struct* permet l'encapsulation des données : l'accès est limité aux méthodes, par défaut.
NB : on peut définir des attributs privés dans une structure, mais cela reste un type plus limité que la classe.

Dans l'exemple proposé ici (slide 6), les attributs sont privés (donc non accessibles directement) et les en-têtes des méthodes sont publiques, afin de permettre leur utilisation.

Déclaration dans fichier en-tête (header) .h ou .hpp :

```
#ifndef Point_H // Si la constante n'a pas été définie, le fichier n'a jamais été inclus
#define Point_H // On définit la constante pour que la prochaine fois le fichier ne soit plus inclus
class Point { // Déclaration de la classe Point
   int x,y ; // attributs (private par défaut)
public :
   void init(int xx, int yy) ; // en-tête de méthode de classe
} ;
```

```
#endif
```

Définition dans fichier cpp :

```
#include "Point.hpp"
void Point::init(int xx, int yy) {  // Définition de la méthode
    x=xx ;
    y=yy ;
}
```

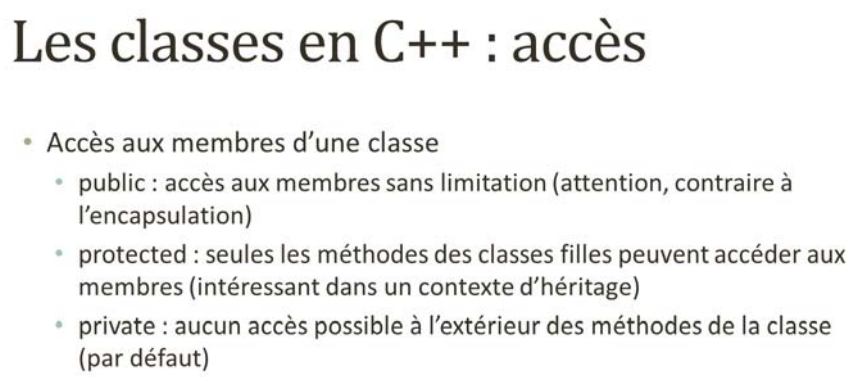
Avantages :
- distribuable : fichier en-tête.hpp distribué avec son binaire (le source .cpp reste secret, donc encapsulation)
- factorisation de code, organisation

Les classes en C++ : bonnes pratiques

- Déclaration de la classe (en-tête) : dans un fichier d'en-tête (header) d'extension .hpp
- Définition de la classe (corps) : dans un fichier du même nom, mais d'extension .cpp

Avantages
- distribuable : fichier d'en-tête distribué avec son binaire (le source .cpp reste caché)
- factorisation de code, organisation

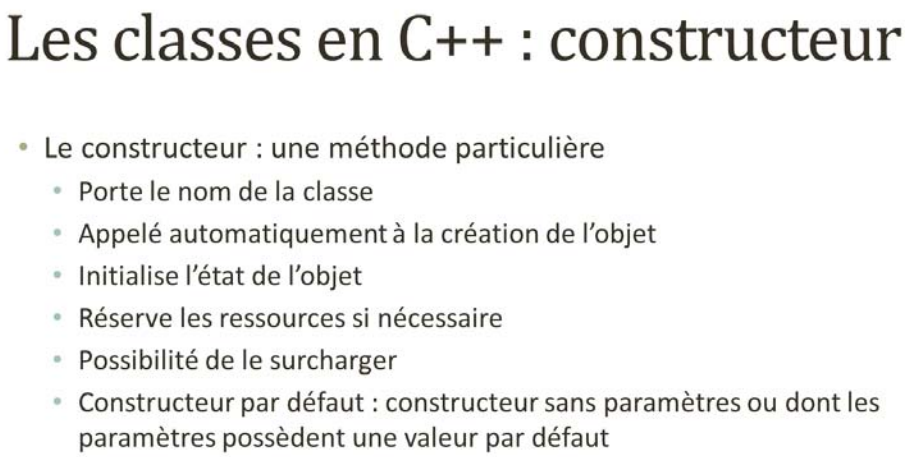
Slide 7



Slide 8

Quelques précisions pour l'accès aux membres d'une classe...
Il vaut mieux éviter d'avoir des attributs publics, ce qui est contraire à l'encapsulation.
La partie protégée est intéressante dans le contexte de l'héritage (les descendants d'une classe et la classe elle-même accèdent à ses membres).
« private » est le mode par défaut lors de la déclaration d'une classe : tout est protégé à l'extérieur de la classe.

Méthode de classe particulière : le constructeur



Slide 9

Le constructeur ne retourne pas de résultat, à proprement parler.
On peut le surcharger (et donc initialiser un objet à partir de différents paramètres)
Le constructeur par défaut est utilisé lorsqu'un objet est créé sans appel explicite à un constructeur.

Les classes en C++ : constructeur bis

Exemple :

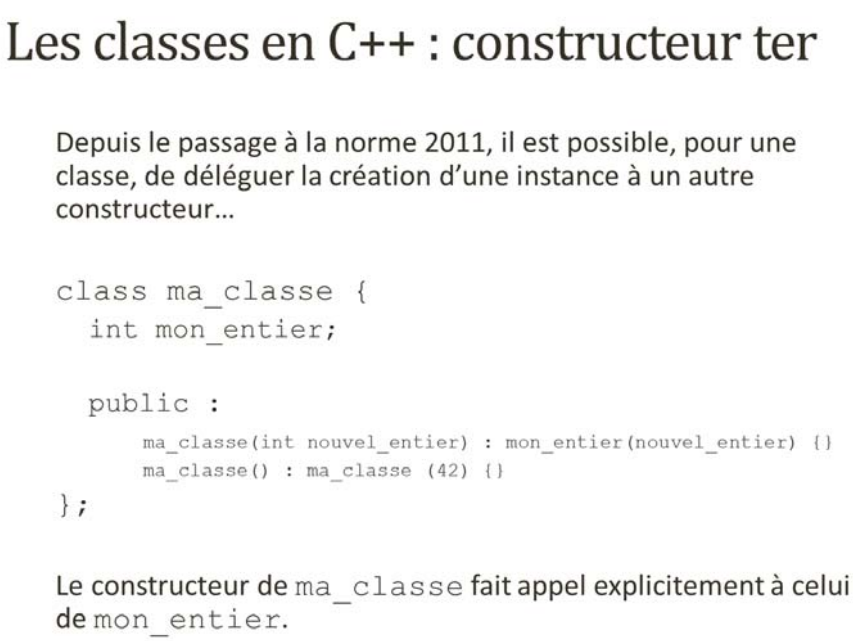
```
class Point{//Déclaration de la classe Point
  int x,y ;
public :
  Point(int xx,int yy) ; //Constructeur
  Point(int xx) ; //Constructeur
};
```

- Nécessité de préciser le code du constructeur si la classe contient des membres de type pointeur
- Remplace une méthode de type `init`

```
Point ::Point(int xx, int yy):x(xx), y(yy) {}
// initialisation de x avec xx et de y avec yy
```

Slide 10

Constructeur
Par défaut : signature sans argument, ssi classe n'en possède pas
Par copie : signature avec un argument unique de type référence à une instance de la classe, ssi classe n'en possède pas (voir plus loin)
De transtypage : un seul argument de type différent de la classe elle-même (jamais par défaut)
Autres : plusieurs argument (jamais)
=> toute définition explicite "écrase" la génération automatique



Slide 11

C++11 permet de déléguer la création d'une instance à un autre constructeur, évitant la duplication de code
Il est possible d'écrire (si on a un compilo 2011 et au-delà) :

```
class ma_classe {
        int mon_entier;

        public :
        ma_classe(int nouvel_entier) : mon_entier(nouvel_entier) {}
        ma_classe() : ma_classe (42) {}
};
```

Méthode de classe particulière bis : le destructeur

Les classes en C++ : destructeur

- Appelé automatiquement quand un objet sort de sa portée, est détruit (`delete`),...
- Libère les ressources mémoire mobilisées par le constructeur
- Porte le même nom que la classe préfixé par ~
- Il est très rare de l'appeler explicitement
- Ni surcharge, ni redéfinition

```
class A{
  int *p ;
public :
  A(){p=new int;} //Constructeur
  ~A(){delete p;} //Destructeur
};
```

Slide 12

Du bon usage des constructeurs et destructeurs

Rule of 3/5/0 ou... la « Sainte Trinité »

- **Rule of 3**

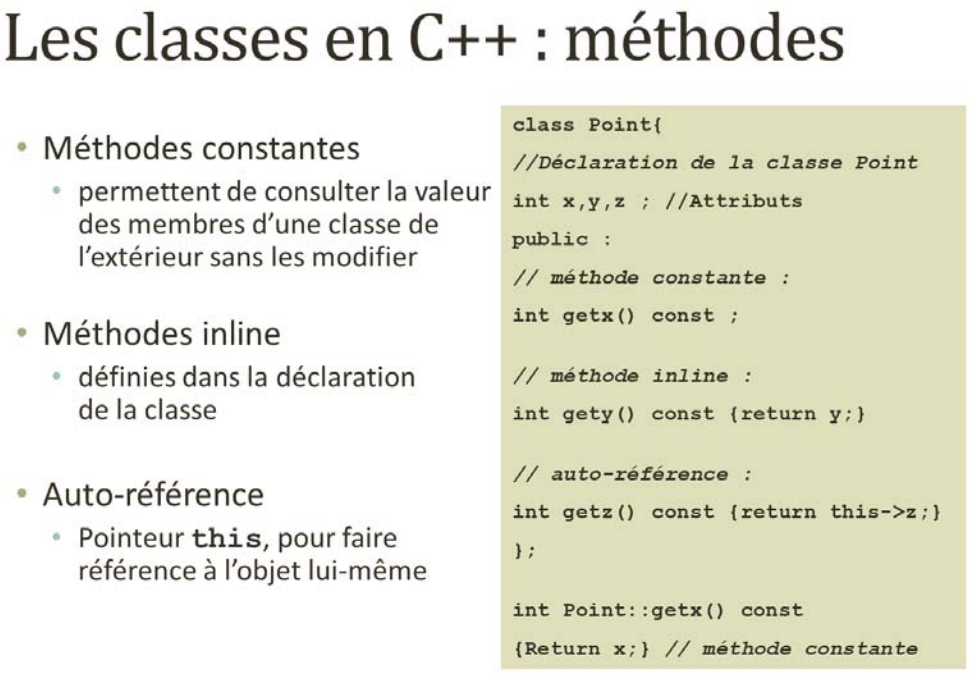
  Si une classe nécessite un destructeur défini par l'utilisateur, un constructeur de copie défini par l' utilisateur , ou un opérateur d'affectation défini par l'utilisateur , il faut très probablement les trois.
- **Rule of 5**

  Élargissement de la règle de 3 liée à la norme 2011, incluant les opérateurs de mouvement (cf. l'an prochain).
- **Rule of 0**

  Pour une classe qui n'est pas explicitement responsable de ressources, mieux vaut ne coder aucune des opérations de la règle de cinq (constructeur de copie, affectation, constructeur de mouvement, affectation par mouvement, destructeur) car le compilateur fera mieux que nous.

Slide 13

Autres méthodes de classe en C++



Slide 14

L'auto-référence, pour accéder à l'objet lui-même

Slide 15

Pour que les fonctions membres (non statiques) puissent accéder aux données membres d'un objet spécifique, c'est-à-dire d'une instance de leur classe, le compilateur leur transmet implicitement comme premier argument un pointeur sur ces données : il s'agit du pointeur this. Cela signifie que *this représente l'objet lui-même.
Le pointeur this est accessible à l'intérieur de la fonction membre et c'est un pointeur constant (vous ne pouvez pas le modifier).

Le type du pointeur this d'un objet de classe Date est Date *const.

Exemple avec une classe Date :

```
//déclaration de classe
class Date {
public :
	void ajoute_mois (int n) {m+=n} // incrémente le mois de la date
	// …
private :
	int jour,mois,annee;
};
```

prototype et définition de la fonction AugmenteAnnee() :
Date& AugmenteAnnee(int n); //prototype à inclure dans l'en-tête de la classe.

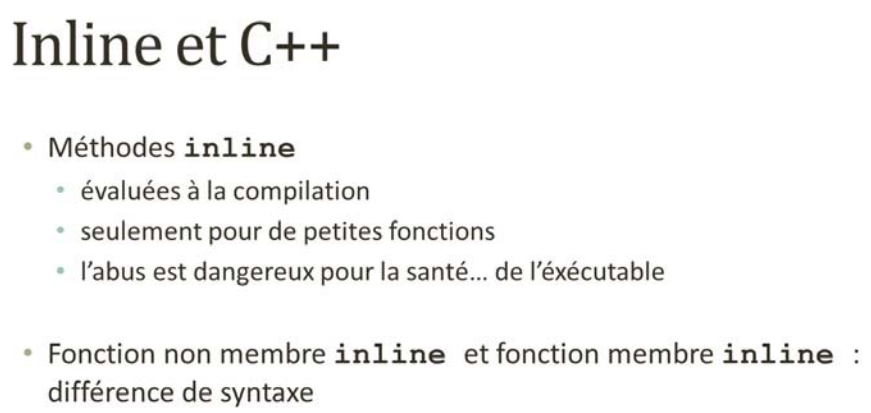
```
/******Définition de AugmenteAnnee()******/

Date& Date::AugmenteAnnee(int n)
{ //Cette fonction permettra d'ajouter n années à l'objet Date concerné.
	if(jour==29 && mois==2 && !bissextile(annee+n){
		//s'il s'agit du 29 février et que annee+n n'est
		// pas bissextile
		jour=1; //on modifie aussi le jour et le mois
		mois=3;
	}
	annee = annee + n;
	return *this; //on retourne une référence sur l'objet courant
}
```

L'intérêt de retourner une référence sur l'objet mis à jour est que si l'on ajoute d'autres fonctions de mise à jour en relation avec celle-ci (pour ajouter des jours ou des mois à la date, par exemple), il sera possible d'enchaîner les opérations de la façon suivante :

```
void fonction(Date& d)
{
	d.AugmenteJour(1).AugmenteMois(1).AugmenteAnnee(1);
}
```

Méthodes Inline



Slide 16

Les fonctions inline sont évaluées à la compilation : l'appel dans le programme sera remplacé par la valeur évaluée. En C, on a l'équivalent avec #define, sauf qu'on remplaçait par une partie de code et non par sa valeur et que l'on faisait donc le calcul plusieurs fois...

Inline est utile surtout pour les petites fonctions, où la séquence appel de fonction+retour de fonction prend une part non négligeable de l'opération effectuée.

NB : trop de fonction inline peut gonfler le code et aboutir au résultat inverse de celui souhaité (impact négatif sur l'exécution et un défaut d'encapsulation).

Fonction inline non membre

Déclaration : void f(int i, char c)

Définition **dans le fichier d'en-tête** (hpp) :
inline void f(int i, char c) {
/* corps…*/
}

-> la définition doit être impérativement dans le .hpp, sauf si f est utilisée seulement dans le .cpp où elle figure

Fonction inline membre

Déclaration :

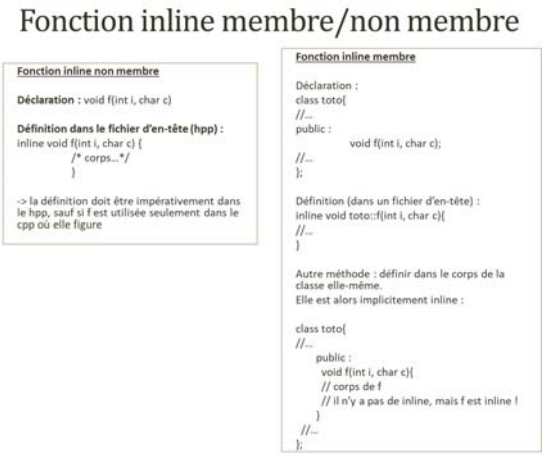
```
class toto{
//…
public :
        void f(int i, char c);
//…
};
```

Définition (dans un fichier d'en-tête) :

```
inline void toto::f(int i, char c){
//…
}
```

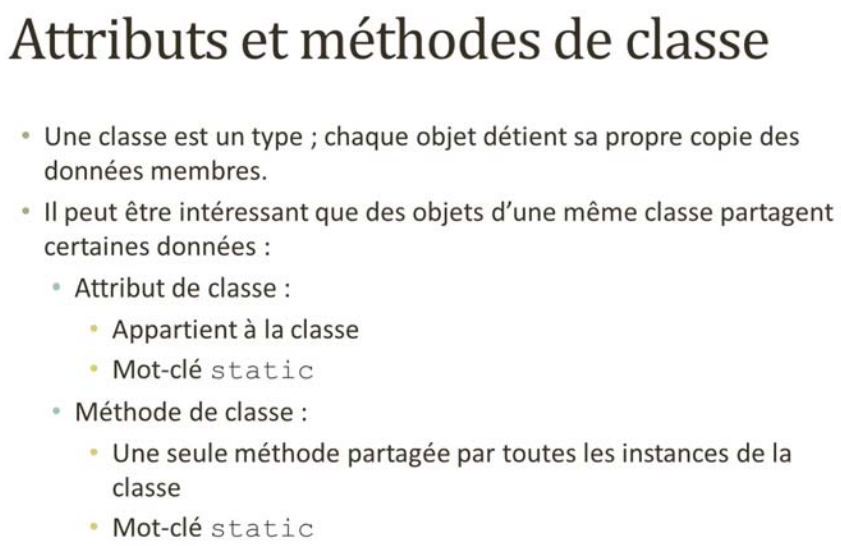
Un autre moyen de définir une fonction inline membre d'une classe est de la définir dans l'en-tête de la classe elle-même. Elle est alors **implicitement inline** (le mot-clé est absent) :

```
class toto{
//…
public :
        void f(int i, char c){
        // corps de f                    => il n'y a pas de inline, mais f est inline !
        }
//…
};
```



Slide 17

Les attributs et méthodes de classes



Slide 18

Une classe définit un type et chaque objet (instance) de la classe détient sa propre copie des données membres.
Il peut être intéressant que des objets d'une même classe partagent certaines données (*i.e.* une seule donnée dont la valeur et l'adresse sont communes à toutes les instances de la classe).
On peut utiliser à cette fin des variables globales. Cependant il est généralement plus judicieux et plus élégant que ces données soient déclarées comme faisant partie de la classe : ce sont les attributs de classe.
On utilise le mot clé static pour déclarer un attribut de classe.

```
#include <iostream>
class Point{
	private :
		int x,y,z;
		static int compteur; // initialisation interdite ici (sauf si const)
	public :
		static int getNbPoints();
		Point(){compteur++} ; // constructeur;
		~Point(){compteur--};// destructeur
};
	int Point::getNbPoints(){cout << compteur << endl;};
	int Point::compteur =0; // pas d'initialisation par défaut

int main() {
	Point a; Point::getNbPoints();
	Point b; Point::getNbPoints();
}
```

Différence Java : en c++, l'initialisation d'un attribut static ne peut être un bloc exécutable contenant des instructions exécutables.

Exemple

```
#include <iostream>
class Point{
      private :
            int x,y,z;
            static int compteur;
            // initialisation interdite ici (sauf si const)
      public :
            static int getNbPoints();
            Point();
            ~Point();
};

// dans le fichier principal...
int Point::compteur =0; // pas d'initialisation par défaut
Point::Point(){compteur++;} // constructeur
Point::~Point(){compteur--;} // destructeur
int Point::getNbPoints(){cout << compteur << endl;};

int main() {
      Point a; Point::getNbPoints();
      Point b; Point::getNbPoints();
}
```

Slide 19

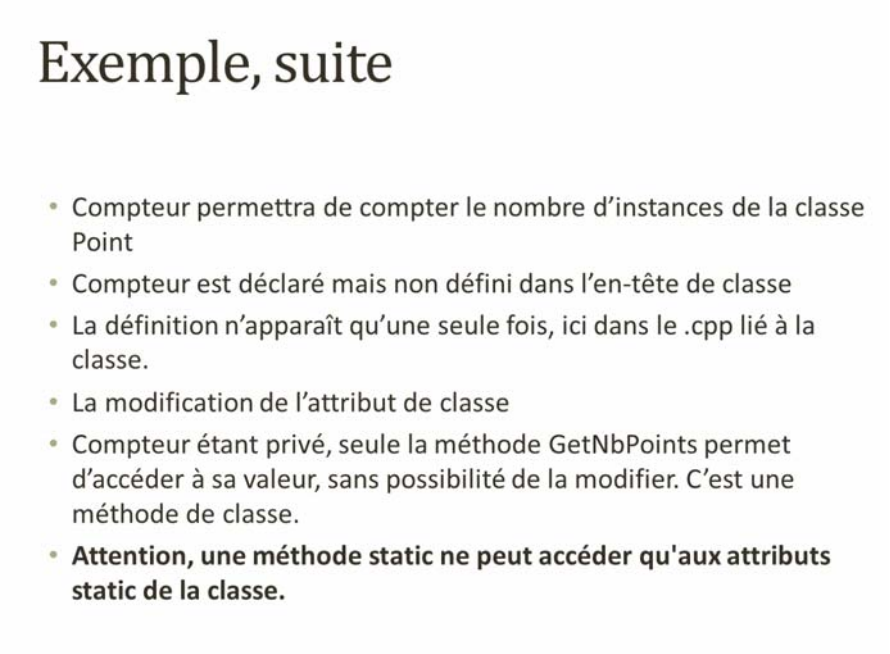
Revenons sur la classe Point précédemment déclarée.
On désire compter le nombre d'instances de classe (*i.e.* le nombre d'objets de cette classe).
Pour cela, on va déclarer un attribut de classe compteur à la classe Point qui sera incrémenté à la création d'un objet et décrémenté à la destruction.
Cet attribut de classe doit pouvoir être récupéré depuis l'extérieur mais il ne faut surtout pas qu'on puisse le modifier : on le déclare donc en *private* et on définit une méthode d'accès getNbPoints() qui renvoie la valeur du compteur.
Cette méthode ne doit pas être détenue par chaque instance de la classe Point, c'est une méthode partagée par tous les objets de la classe Point : c'est une méthode de classe, déclarée elle aussi avec le mot clé static.

Dans cette déclaration de la classe Point, le membre statique compteur est seulement déclaré et non défini.
Il faut que la définition (et en particulier son initialisation qui est ici essentielle) apparaissent quelque part dans le programme (en général dans le .cpp lié à la classe), et de façon unique :

```
// Définition de l'attribut de classe
int Point::compteur = 0;
// Définition la méthode de classe
int Point::getNbPoints() { return compteur; }
```

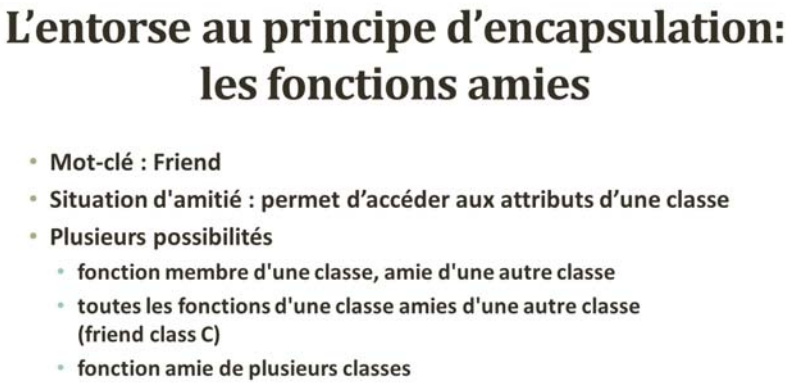
Pour accéder à la méthode de classe getNbPoints dans le programme, on la désignera par Point::getNbPoints() :

Attention, une méthode static ne peut accéder qu'aux attributs static de la classe.



Slide 20

Les Fonctions amies



Slide 21

Une déclaration de fonction membre ordinaire spécifie logiquement 3 choses :
1- La fonction peut accéder aux parties privées de la déclaration de classe
2- La fonction est dans la portée de la classe
3- La fonction doit être appelée via une instance de la classe

Déclarer la fonction membre comme *static* lui donne les deux premières propriétés.

Déclarer une fonction non-membre comme amie (mot-clé *friend*) lui donne la première propriété seulement. Une fonction déclarée comme amie a accès à l'implémentation complète de la classe dont elle est amie, comme une fonction membre, mais elle est indépendante de la classe elle-même.

```
class Joueur; // ceci est un exemple pédagogique : il y a un moyen d'éviter
                // d'utiliser la fonction amie joueurEstMembreEquipe, lequel ?
class Equipe {
   vector<Joueur> effectif;
public:
Equipe();
   friend bool joueurEstMembreEquipe(Joueur j, Equipe e);
};
class Joueur {
   string nom;
   Equipe employeur;
public:
   Joueur();
   friend bool joueurEstMembreEquipe(Joueur j, Equipe e);
};
bool joueurEstMembreEquipe(Joueur j, Equipe e) {}
```

On peut avoir une fonction amie de plusieurs classes.

Par exemple, si l'on veut permettre la multiplication entre vecteurs et matrices, il faudra définir l'opérateur * comme étant ami à la fois de la classe Vecteur et de la classe Matrice.

Ici, la méthode JoueurEstMembreEquipe accède aux attributs privés des classes Equipe et Joueur, ce qui lui permet de comparer ces derniers. Cette fonction amie n'est membre d'aucune des deux classes.

Fonction amie : exemple

```
class Joueur;
      // ceci est un exemple pédagogique : il y a un moyen d'éviter
      // d'utiliser la fonction amie joueurEstMembreEquipe, lequel ?

class Equipe {
     vector<Joueur> effectif;
public:
Equipe();
     friend bool joueurEstMembreEquipe(Joueur j, Equipe e);
};

class Joueur {
     string nom;
     Equipe employeur;
public:
     Joueur();
     friend bool joueurEstMembreEquipe(Joueur j, Equipe e);
};

bool joueurEstMembreEquipe(Joueur j, Equipe e) {}
```

Slide 22