C++: Cours n° 4
Programmation modulaire
Héritage
Membres de classe statiques

Module Info3/M3105C

Sebastien.Kramm@univ-rouen.fr

IUT GEII Rouen

2018-2019

Préambule

Compilation en ligne : Intéressant pour tester rapidement un petit *snippet* de code.
Quelques services :

- `https://ideone.com/`
- `http://cpp.sh/`
- `http://coliru.stacked-crooked.com/`

Sommaire

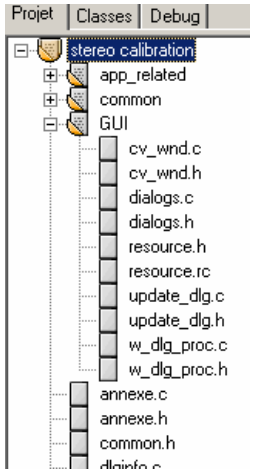
Programmation modulaire
- Principes
- Inclusions multiples
- Déclaration d'existence

Héritage
- Création de classe dérivée
- Conversions de type
- Niveaux d'encapsulation
- Heritage et constructeurs

Membres de classe statiques
- Attributs
- Méthodes

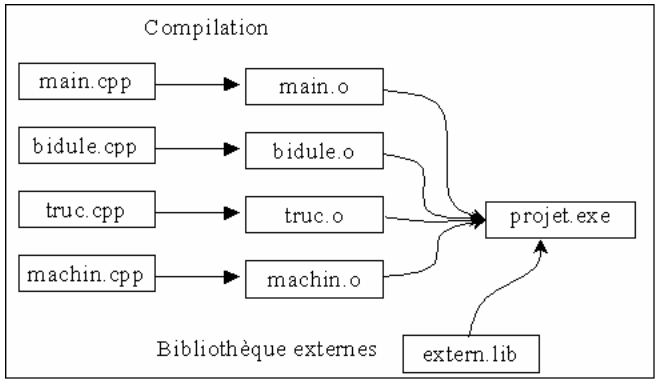
Pourquoi ?

- Un projet d'une certaine envergure peut aller de 1000 lignes à plus de 100 000 lignes.
- Le code est réparti entre plusieurs dizaines ou centaines de fichiers.
- Approche modulaire : chaque classe est décrite par 2 fichiers :
  - un fichier d'en tête (.h ou .hpp) : déclaration. (*interface* de la classe).
  - un fichier source (.c ou .cpp) : implémentation.
- Intérêt
  - Facilité d'édition.
  - Réutilisation possible dans une autre application.
  - Diminution temps de compilation : on peut pré-compiler les fichiers, et les lier lors de l'édition de liens.



Construction de l'exécutable

- La construction de l'exécutable nécessite :
  - La compilation de chacun des fichiers sources (.c ou .cpp), si besoin (= si pas déjà compilé).
  - L'édition des liens (assemblage des différents binaires compilés + bibliothèques externes).

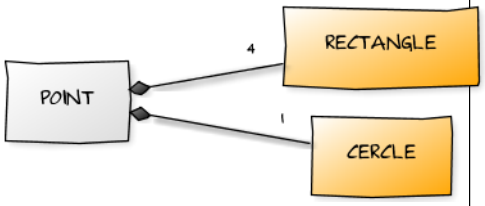


Construction de l'exécutable

- Gros projets : plusieurs centaines de fichiers à compiler, autant de binaires à assembler ensemble → parfois plusieurs heures... (notion de "Nightly Build")
- On utilise un outil qui va s'occuper de cette construction : `make`
  - S'occupe de ne "reconstruire" (compiler) que ce qui a besoin de l'être.
  - Vérifie les **dépendances** entre fichiers
  - Prends en entrée un fichier de description du projet (*makefile*).
  - Inconvénient : syntaxe difficile.
- Avec une IDE, utilisation transparente : le fichier makefile est généré automatiquement par l'IDE, puis exécuté par un simple clic.

Problème : Inclusions Multiples

- Problème : il arrive fréquemment que plusieurs objets utilisent les mêmes objets de base dans leur construction.
- Or, les classes ne peuvent être déclarées qu'**une seule fois** lors de la compilation d'un fichier.
- Exemple :
  - Soit 3 objets, POINT, RECTANGLE et CERCLE, déclarés dans des fichiers séparés.
  - RECTANGLE et CERCLE utilisent l'objet POINT (relation de _____)



⇒ *un* **rectangle** *est* _____

- Pour un programme utilisant des points : `#include "point.h"`
- Pour un programme utilisant des rectangles : `#include "rectangle.h"`
- Comme RECTANGLE utilise POINT, on doit prévoir dans "rectangle.h" : la ligne : `#include "point.h"`

Quel est le problème ?

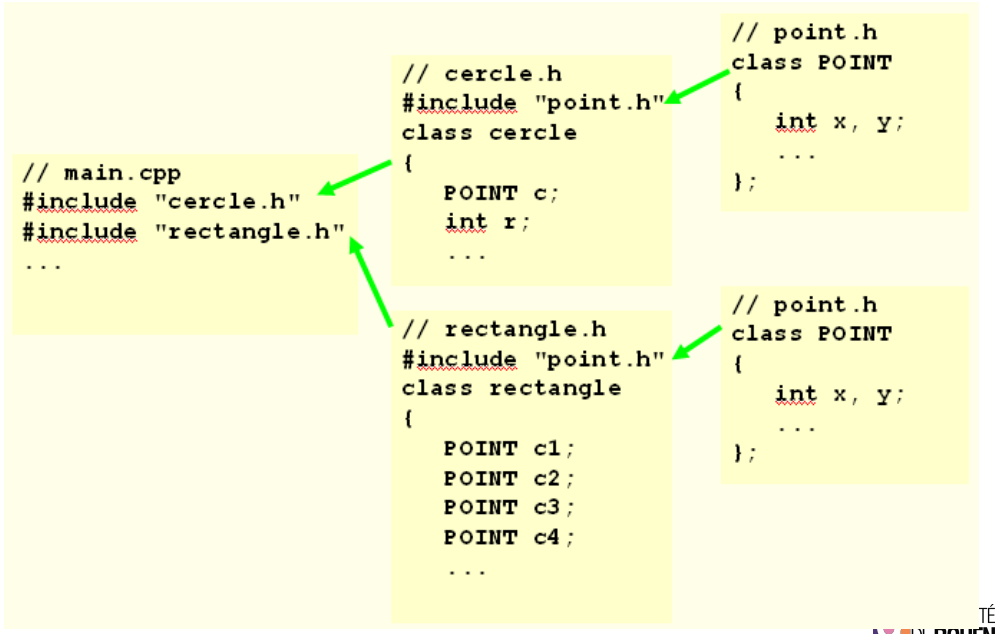
- Pour un programme utilisant des cercles ET des rectangles, il faudra avoir :

```
#include "cercle.h"
#include "rectangle.h"
int main()
{ ...
```

⇒ Le compilateur va générer une erreur !

- Les fichiers "cercles.h" et "rectangle.h" ont tous les deux l'inclusion suivante : `#include "point.h"`
  ⇒ La classe POINT est déclarée **2 fois** lors de la compilation de main() !

Inclusions multiples : illustration



Solution ?

- Il faut que la déclaration de l'objet ne soit incluse qu'une seule fois.
- On réalise ceci via des **directives** du préprocesseur (#) :

`#define SIMPLET`	définit l'existence d'un symbole
`#define PI 3.14`	assigne une chaîne de caractères à un symbole
`#ifndef PROF`	ne lit la suite que si le symbole N'EST PAS défini
`#ifdef PROF`	ne lit la suite que si le symbole EST défini

- Exemples :

```
#ifdef ATCHOUM
   int a;
#endif
```

Déclare a si le symbole ATCHOUM est défini.

```
#ifndef GRINCHEUX
   a += b;
#endif
```

Additionne b dans a si le symbole GRINCHEUX n'est **PAS** défini.

Solution pratique

- On rajoute dans le fichier d'en-tête (.h) des lignes pour que la compilation ne soit effectuée **QUE** lors de la **première** lecture du fichier :

```
// bidule.h
#ifndef _BIDULE_H_
#define _BIDULE_H_

class BIDULE
{
   ...
};

#endif    // fin du test, et fin du fichier
```

- Ceci est à faire dans **chaque** fichier d'en-tête,
- Le symbole **doit** être unique pour tout le projet, la convention est de construire un symbole basé sur le nom du fichier.
- Les IDE ont des assistants qui génèrent ces symboles **automatiquement** lors de la création d'une classe.

Illustration

- Les 3 fichiers d'en-tête deviennent :

```
// point.h
#ifndef _POINT_H_
#define _POINT_H_

class POINT
{
   int x, y;
   ...
};

#endif
```

```
// cercle.h
#ifndef _CERCLE_H_
#define _CERCLE_H_

#include "point.h"
class CERCLE
{
   POINT c;
   int r;
   ...
};
#endif
```

```
// rectangle.h
#ifndef _RECT_H_
#define _RECT_H_

#include "point.h"
class RECTANGLE
{
   POINT c1;
   POINT c2;
   POINT c3;
   POINT c4;
   ...
};
#endif
```

Compilation conditionnelle

- Ces directives pré-processeur peuvent aussi servir à faire de la **compilation conditionnelle**.
- Exemple : Ne - compiler / pas compiler - une partie du code que sur un OS donné (programmation multi-plateforme) :

```
#ifdef _WIN32
  // ici , le code à compiler uniquement si "Windows"
  ...
#else
  // ici , le code à compiler uniquement si pas "Windows"
  ...
#endif
```


Intermède : syntaxe de `#include`

```
#include "fichier"
#include "fichier.h"
#include "fichier.hpp"
#include <fichier>
#include <fichier.h>
```

?

- Bibliothèque standard : < ... >
- Fichiers d'en-tête du projet : "...", avec l'extension (.h en général, parfois .hpp ou .hxx)
- Les fichiers d'en-tête C++ sont fournis en 2 versions :
  - une version avec extension .h : obsolètes, fournis pour la compatibilité ascendante,
  - une version sans extension, dont les identificateurs sont inclus dans l'espace de nommage `std` : il faut déclarer qu'on utilise cet "espace de noms" :
- Exemple :

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


Sommaire

Programmation modulaire
- Principes
- Inclusions multiples
- Déclaration d'existence

Héritage
- Création de classe dérivée
- Conversions de type
- Niveaux d'encapsulation
- Heritage et constructeurs

Membres de classe statiques
- Attributs
- Méthodes

Introduction - motivation

- Le concept d'héritage est au cœur même de la P.O.O.

Rappel

"La classe dérivée est une version spécialisée de la classe de base"

- Ex : Si `Avion` est une classe dérivée de `Aeronef`, on peut dire que `Avion` est "une sorte" d'`Aeronef`.
  ⇒ Tout ce qu'on peut faire avec un `Aeronef`, on doit pouvoir le faire avec un `Avion` (inverse pas vrai).
- Ex : si la classe `Aeronef` est dotée d'une méthode `Decoller()`, et que la classe `Avion` n'en a pas, on pourra écrire :

```
Avion a;
a.Decoller();
```

- L'héritage permettra des conversions de type automatiques.

Création d'une classe dérivée

- Soit une classe ANIMAL :

```
class ANIMAL
{
   string nom;
   COULEUR couleur;
   void courir();
};
```

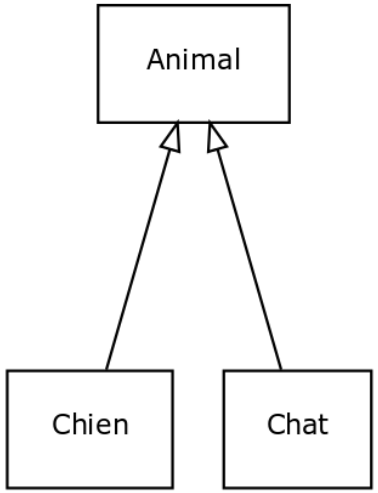
- On dérive cette classe en deux nouvelle classes CHIEN et CHAT :

```
class CHIEN : public ANIMAL
{
   void aboyer();
};
```

```
class CHAT : public ANIMAL
{
   int  moustaches;
   void miauler();
};
```

- Tous les objets crées à partir de CHIEN ou CHAT seront munis d'un attribut `couleur` : ils **héritent** de cet attribut.
- Tous les objets crées à partir de CHIEN ou CHAT seront munis d'une méthode `courir()` : ils **héritent** de cette méthode.

UML : relation d'héritage



Rappel : conversions de type C

- En C, on peut dire que le type `float`, représente un entier auquel on ajoute une partie fractionnelle
  $\Rightarrow$ vu ainsi, le type `float` "*dérive*" du type `int`
- On peut réaliser des conversions de type automatiques :

```
float pi = 3.1415;
int a = 55;
a = pi;  // ok, a = 3
```

Conversion `float` $\rightarrow$ `int` : dégradation de l'information, on perd la partie fractionnelle.

```
float pi = 3.1415;
int a = 55;
pi = a;  // ok, pi = 55.0
```

Conversion `int` $\rightarrow$ `float` : OK, on ajoute une partie fractionnelle nulle.

Conversions de type C++

- De la même façon, C++ peut convertir implicitement des objets d'un type dans un autre, mais **uniquement** :
  - d'un objet d'une classe dérivée
  - vers un objet d'une classe de base (*upcasting*).
- Impossible de créer un objet sans fournir la **totalité** de l'information.

```
CHAT   chat;
ANIMAL animal;
animal = chat;
```

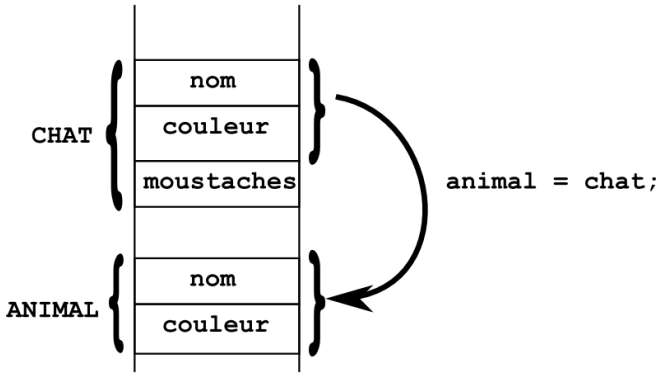
Ok, mais perte des informations spécifiques au chat (dégradation de l'information).

```
CHAT   chat;
ANIMAL animal;
chat = animal;
```

Rejeté par le compilateur : il manque des informations, un animal n'est pas forcement un chat !

Vu de la mémoire

- Se souvenir que le symbole = signifie :
  "copier ce qu'il y a à droite **dans** la variable à gauche"



- *Upcasting* ⇒ perte des attributs spécifiques à la classe dérivée.

Protection des membres hérités

- Question : comment sont considérés les membres de la classe de base dans les objets instanciés à partir d'une classe dérivée ?
- 2 niveaux de réponse :
  - Type d'héritage (public/private).
  - Type d'origine des membres de la classe de base (public/private).

Si les membres sont déclarés dans la *classe de base* comme `private`
⇒ ils seront inaccessibles dans les méthodes de la classe dérivée.

Si les membres sont déclarés dans la *classe de base* comme `public`
⇒ ils seront... publics ! (et donc librement accessible par tout le monde)
(**violation** du principe d'encapsulation).

Attributs protégés (`protected`)

- Il est souvent souhaitable de rendre accessible aux classes dérivées le contenu des classes de base, tout en maintenant la protection vis-à-vis de l'exterieur.
- Il existe pour ça un 3e niveau de protection, intermédiaire entre `public` et `private` : `protected` ("protégé")
- Il peut être utilisé :
  - pour caractériser un niveau de protection d'un membre de classe (attribut ou méthode),
  - pour caractériser un **type d'héritage** (voir plus loin).
- Les attributs et méthodes dotés de ce niveau de protection seront :
  - **inaccessibles** de l'extérieur des objets de cette classe de base,
  - **inaccessibles** de l'extérieur des objets d'une classe dérivée,
  - **accessibles** aux méthodes des classes dérivées.

Exemple

```
// enginvolant.hpp

class ENGINVOLANT
{
  private:
    int NbPlaces;
  protected:
    int altitude;
};
```

- `NbPlaces` et `altitude` seront **inaccessibles** de l'extérieur.
- `altitude` sera **accessible** depuis les classes dérivées de celle-ci.
- `NbPlaces` sera **inaccessible** depuis les classes dérivées de celle-ci.

Attributs protégés : exemple d'héritage

- Exemple de classe dérivée :

```
// avion.hpp
#include "enginvolant.hpp"

class AVION : public ENGINVOLANT
{
  private:
    int NbMoteurs;
  public:
    AVION()   // constructeur
      { altitude = 0; }
    ...
};
```

(Note : pour les fonctions courtes, il est possible de les définir directement dans la déclaration de la classe).

- `altitude` est accessible au constructeur de AVION, bien qu'il appartiennent à la classe de base ENGINVOLANT, parce qu'il est déclaré comme `protected` dans celle-ci.
- Pour acceder aux attributs privés de la classe de base, il faudra passer par leurs accesseurs et mutateurs.

Type d'héritage

- On peut en fait réaliser l'héritage de 3 façons :

```
class FILLE : public MERE
{
 ...
};
```

```
class FILLE : protected MERE
{
 ...
};
```

```
class FILLE : private MERE
{
 ...
};
```

- Le plus courant est l'héritage public :
  $\Rightarrow$ c'est le seul qui permette à un objet d'une classe dérivée de jouer le rôle d'un objet d'une classe de base.

Conversions de type : application

- Dotons nos classes CHIEN et CHAT de moyens de défense (ou d'attaque...)

```
class CHIEN:public ANIMAL {
   ...
  void mordre( ANIMAL a );
};
```

```
class CHAT:public ANIMAL {
   ...
  void griffer( ANIMAL a );
};
```

- On pourra ainsi écrire dans un programme :

```
CHIEN medor;
CHAT  matou;
medor.mordre( matou );
matou.griffer( medor );
```

- Question : l'argument de la méthode `mordre()` est un ANIMAL, et on passe un CHAT ???
- Réponse : il y **automatiquement** dégradation de l'objet dans son type de base : le CHAT se transforme en objet de type ANIMAL lors de l'appel de la méthode.

Conversions de type : application

- Les méthodes `mordre()` et `griffer()` pourront s'écrire :

```
// chien.cpp

void CHIEN::mordre( ANIMAL a )
{
  cout << nom << ":je mors ";
  cout << a.nom << endl;
}
```

```
// chat.cpp

void CHAT::griffer( ANIMAL a )
{
  cout << nom << ":je griffe ";
  cout << a.nom << endl;
}
```

Remarque : transmission des arguments

- On utiliserai ici de façon plus judicieuse la transmission d'une **référence** sur la variable originelle :

- Déclaration :

```
// chien.h
#include "animal.h"

class CHIEN:public ANIMAL {
  ...
  void mordre( ANIMAL& a );
};
```

- Implémentation : (aucun changement à part la signature de la méthode)

```
// chien.cpp

#include "chien.h"

void CHIEN::mordre( ANIMAL& a )
{
  cout << nom << ":je mors ";
  cout << a.nom << endl;
}
```


Heritage et constructeurs

- Lors de la création d'un objet d'une classe dérivée, le constructeur de la classe de base est **toujours** appelé d'abord.
- S'il existe dans les deux classes des constructeurs sans arguments (implicites ou explicites), c'est simple :
  - le constructeur de la classe de base est exécuté en premier,
  - puis le constructeur de la classe dérivée.
- Exemple : une classe POINTCOL, qui dérive d'une classe POINT :

```
class POINTCOL : public POINT
{
  private:
    int color;
  public:
    POINTCOL() { color = 0; }
};
```

```
class POINT
{
  private:
    int x, y;
  public:
    POINT() { x = y = 0; }
};
```

⇒ La création d'un objet de type POINTCOL entrainera automatiquement l'exécution du constructeur de POINT, puis du constructeur de POINTCOL.

Heritage et constructeurs : transmission d'arguments

- Si les constructeurs réclament des arguments :

```
class POINTCOL : public POINT
{
  private:
    int color;
  public:
    POINTCOL( int x0, int y0, int
        col );
};
```

```
class POINT
{
  private:
    int x, y;
  public:
    POINT( int x0, int y0 );
};
```

- On va créer un objet de type POINTCOL avec (par exemple) la déclaration suivante :

```
POINTCOL pc( 3, 4, ma_couleur );
```

- Comment le constructeur de POINTCOL va-t-il transmettre les valeurs de x et y à POINT ?

Heritage et constructeurs : solution 2

- Le C++ permet de répondre à ce problème d'une façon élégante, en permettant l'appel **explicite** d'un constructeur d'une classe de base par le constructeur d'une classe dérivée :

```
// pointcol.cpp
#include "pointcol.h"

POINTCOL::POINTCOL( int x0, int y0, int coul ) : POINT(x0, y0)
{
  color = coul;
}
```

- Préférable à la solution 1 (initialisation des attributs de la classe de base dans le constructeur de la classe dérivée).

Exercice

Soit une classe VEHICULE, dotée d'un attribut `NbRoues`, et d'un constructeur permettant de le fixer :

```
class VEHICULE {
   public:
      VEHICULE( int nbr )
      { NbRoues = nbr; }
   private:
      int NbRoues;
};
```

Elle est dérivée en deux classes, MOTO, et AUTO. En écrire les constructeurs (sans arguments), appelant celui de VEHICULE

```
AUTO::AUTO(
{

```

```
MOTO::MOTO(
{

```


Sommaire

Programmation modulaire
- Principes
- Inclusions multiples
- Déclaration d'existence

Héritage
- Création de classe dérivée
- Conversions de type
- Niveaux d'encapsulation
- Heritage et constructeurs

Membres de classe statiques
- Attributs
- Méthodes

Rappel : `static` dans un bloc

- Le mot clé `static` dans un bloc donne à l'objet une durée de vie permanente.
- L'objet est crée via son constructeur lors de sa déclaration (avant l'exécution du `main()` pour les variable globales) et détruit à la fin du programme.
- Les variables globales étant obligatoirement statique, le mot clé `static` n'est pas nécessaire dans ce cas.

```
AUTOBUS abus; // var. globale , et de ce fait statique

void ma_fonction()
{
   static SOUSMARIN sm; // var. locale statique
    ...
}
```


Attribut `static`

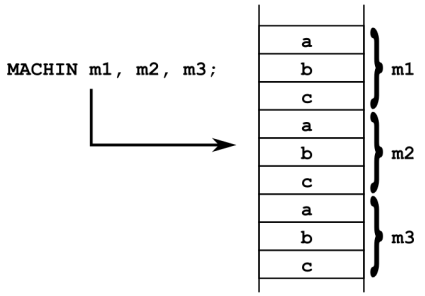
- Associé à un **attribut**, ce mot clé rattache ce dernier **à la classe**, et non plus à l'objet crée à partir de la classe.
- L'attribut devient **unique** pour toutes les instances de cette classe, $\forall$ le nombre d'instances crées.
- La déclaration d'un objet de ce type ne réservera **pas** de memoire pour cet attribut.
  $\Rightarrow$ Il sera nécessaire d'instancier et d'initialiser **explicitement** cet attribut.
- Il peut être privé ou public (de préférence privé).

```
// machin.h

class MACHIN
{
  private:
   static int Nb; // statique
   int a,b,c;     // pas statique
};
```

Attribut static

- Exemple : si on crée 3 variables de type MACHIN :



- Pas d'allocation mémoire pour les attributs statiques.
- L'instanciation et l'initialisation des attributs statiques se fait via une simple **déclaration** dans le fichier d'implémentation :

```
// machin.cpp
#include "machin.h"

int MACHIN::Nb = 0;
 ...
```


Exemple d'application

- Application la plus courante : pouvoir connaître à tout moment combien d'objets d'un type donné ont été instanciés :
  - on dote la classe d'un entier statique,
  - tout les constructeurs doivent incrémenter ce membre,
  - le destructeur doit le décrémenter. (*)

```
// machin.h
class MACHIN
{
  private:
    static int Nb;
  public:
    int GetNb()
      { return Nb; }

   MACHIN() { Nb++; }
   ~MACHIN() { Nb--; }
};
```

```
// machin.cpp
#include "machin.h"

// instanciation et initialisation
// du membre statique
int MACHIN::Nb = 0;

 ...
```

(*) Attention, ceci ne fonctionne que si on définit aussi correctement un **constructeur de recopie** (non-traité dans ce cours).

Exemple d'application

- L'exécution du programme ci-dessous :

```
// main.cpp
#include "machin.h"

int main()
{
  MACHIN m1, m2;

  std::cout << "nb de machins = ";
  std::cout << m1.GetNb() << std::endl;
}
```

Affichera :

```
nb de machins = 2
```


Méthodes statiques

- Problème du programme précédent : on ne peut pas connaître le nombre d'objets ... s'il n'y en a pas ! ⇒ la méthode `GetNb()` s'applique à un objet, qui doit exister, **et** être accessible (problème de la **portée** : les variables peuvent exister ailleurs que dans le main()).
- Solution : **méthodes statiques**.
- Elles peuvent être exécutées sans qu'il y ait d'objet instancié, en utilisant l'opérateur de résolution de portée (`::`)
- Exemple :

```
class MACHIN
{
  private:
    static int Nb;
  public:
    static int GetNb()
      { return Nb; }
};
```

Méthodes statiques : exemple d'utilisation

- Avec cette méthode, le programme suivant :

```
// main.cpp
#include "machin.h"

int main()
{
  cout << "nb = " << MACHIN::GetNb()<<endl;

  MACHIN m1, m2;

  cout << "nb = " << MACHIN::GetNb()<<endl;
}
```

affichera :

```
nb = 0
nb = 2
```