

POO avec C++

Chapitre2

Des classes et des objets



2. Table des matières

1. Introduction

2. Définition d'une classe et de ses attributs
3. Implémentation d'une méthode
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Modules, membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé



2. Table des matières

1. Introduction

1. Paradigme de programmation
2. Classe et objet
3. Exemple de programme en C++
4. Les caractéristiques d'un objet

2.1 Introduction

1. Paradigme de programmation:

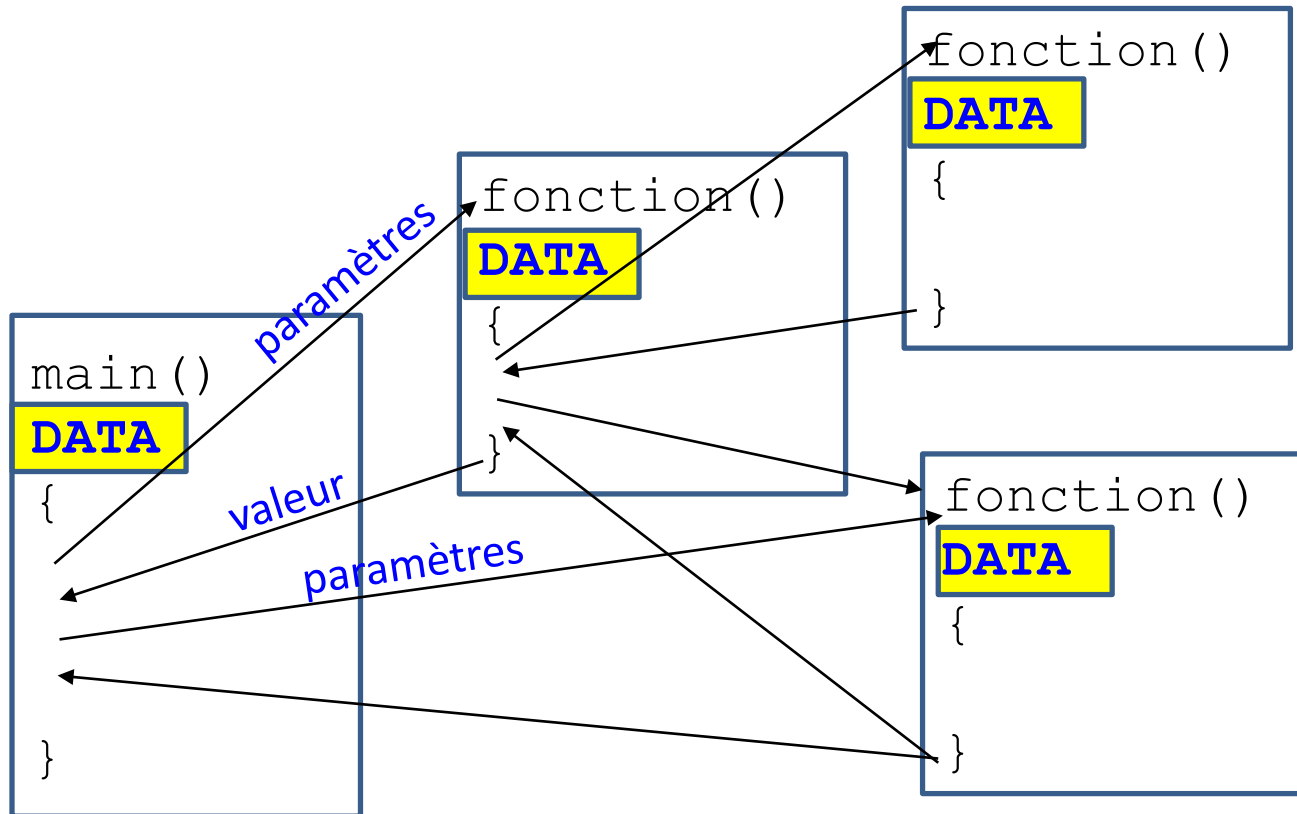
- Programmation **procédurale / impérative**
 - Séparation des données et des fonctions
 - Une fonction agit sur des données
- Programmation **objet**
 - Regroupement des fonctions et données dans un **objet**.
P.ex. : *voiture, pilote, compte bancaire, point, vecteur, ...*
 - Envoi de messages entre objets



2.1 Introduction

1. Paradigme de programmation:

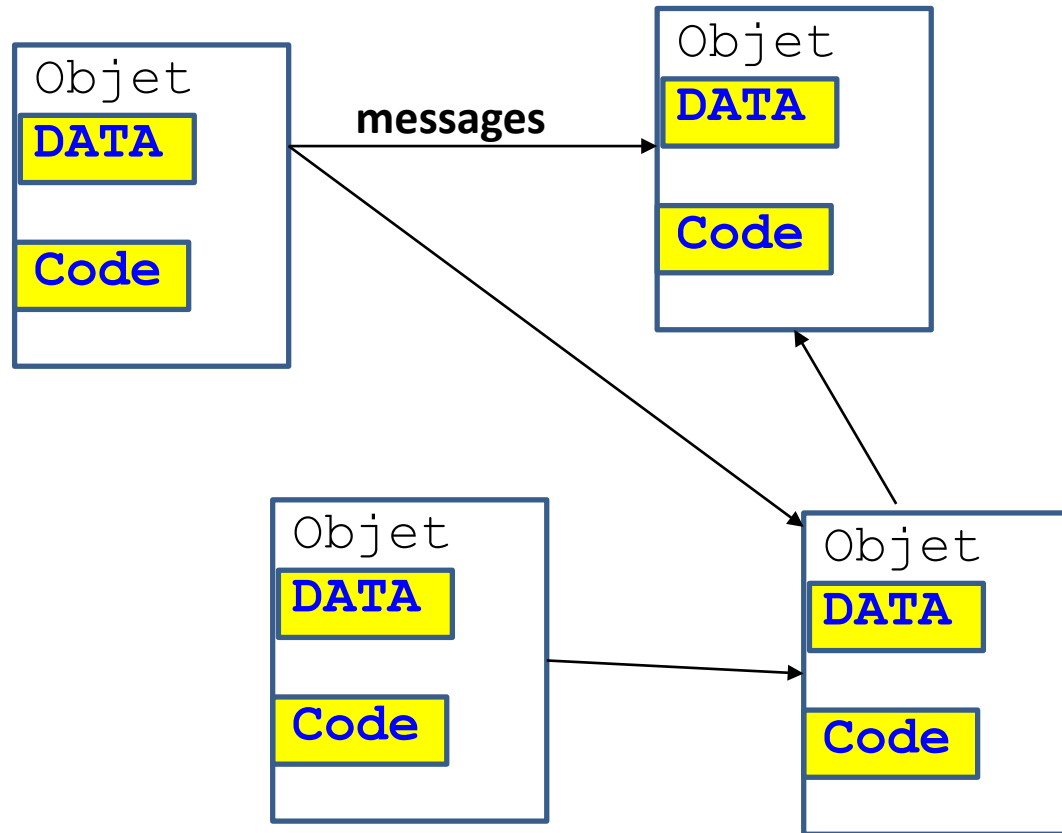
Programmation **procédurale**



2.1 Introduction

1. Paradigme de programmation:

Programmation **objet**





2.1 Introduction

1. Paradigme de programmation:

Principe de la POO

- Approche naturelle de regrouper **des objets en classes**
- Avantages :
 - Code réutilisable
 - Modulaire
 - Plus facile à maintenir / faire évoluer
 - Gestion de gros projet

Programmer en POO c'est...
décrire des objets et des actions sur ces objets



2. Table des matières

1. Introduction

1. Paradigme de programmation

2. Classe et objet

3. Exemple de programme en C++

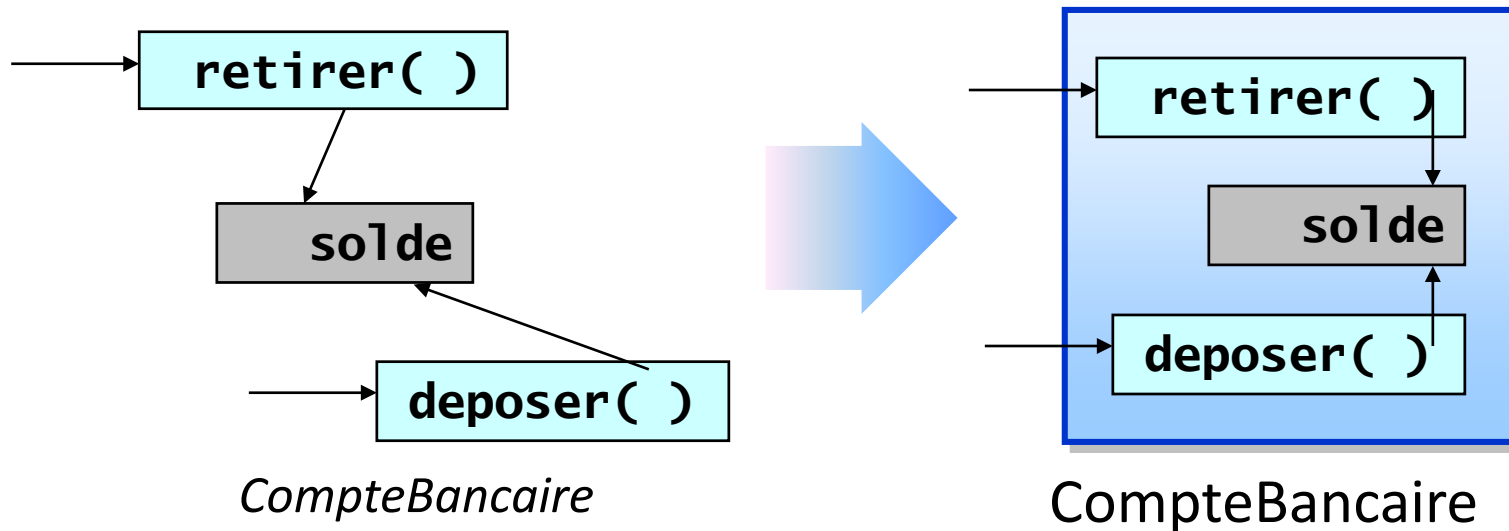
4. Les caractéristiques d'un objet

2.1 Introduction

2. Classe et objet

Notion d'objet

Un objet **regroupe** les données et les fonctions



Fonctions agissent sur des données séparées

POO = fonction + données

2.1 Notion de classe en C++

C

```
int solde;
```

```
deposer(int x)
{
    solde += x;
}
```

```
main()
{
    deposer(15);
}
```

C++

```
class CB
{
    int solde;
    void deposer(int x)
    {
        solde += x;
    }
};
```

Classe

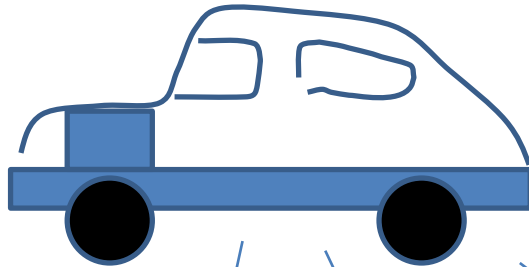
```
main()
{
    CB moncompte;
    CB unautreCompte;
    moncompte.deposer(15);
}
```

Objets

2.1 Classe versus objet

- Une **classe** est un “**type**” **défini** par le programmeur

Classe des voitures (générique)



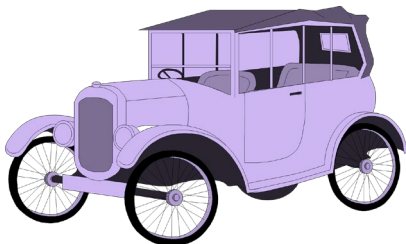
Roues, châssis,
moteur
carrosserie

Voiture

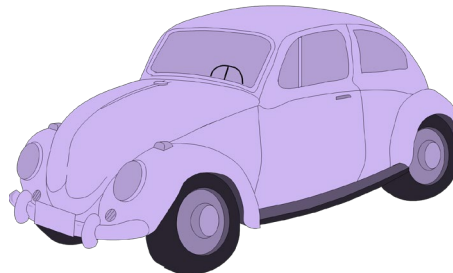
Opel Corsa

- Un **objet** est une **instance** d'une classe

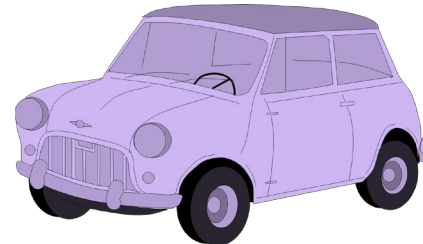
Voitures particulières



Ford T



VW Coccinelle



Mini Cooper

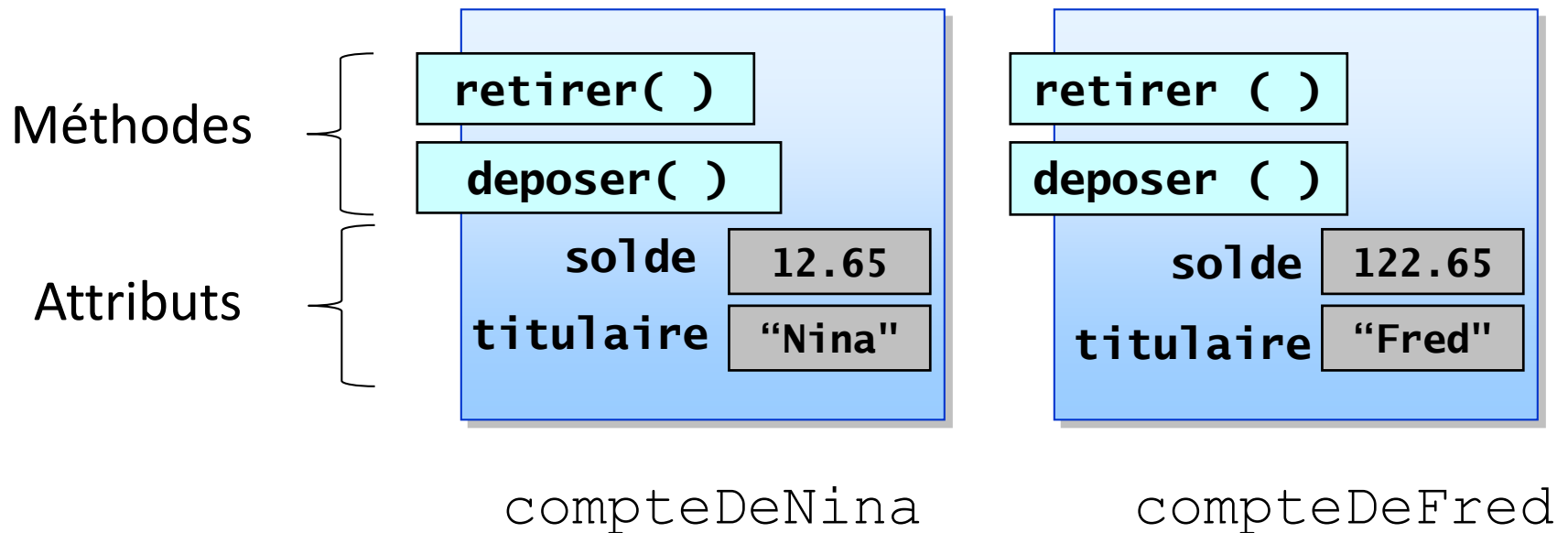
2.1 Principes de POO

Une classe est la description abstraite d'un objet. Elle regroupe:

- Des **attributs** décrivant l'état d'un objet
 - Ex véhicule: vitesse, poids, nombre de passagers, ...
 - Appelés aussi des **variables membres (de la classe)**
 - Type:
 - Simple: int, double
 - Complexe (tableau, objet).
- Des **méthodes** détaillant les actions qu'un objet peut effectuer
 - Ex véhicule : démarrer, accélérer, freiner
 - Appelées aussi des **fonctions membres (de la classe)**
 - Comme les méthodes représentent des actions, leurs noms sont souvent des **verbes**.

2.1 Déclaration d'une classe en C++

- Une classe contient 2 sortes de **membres** :
 - **Méthodes**: fonctions détaillant les actions possibles
 - **Attributs**: variables décrivant l'état de l'objet. Les **attributs d'un objet** définissent son état individuel (l'information spécifique)





2. Table des matières

1. Introduction

1. Paradigme de programmation
2. Classe et objet

3. Exemple de programme en C++

4. Les caractéristiques d'un objet

2.1 Exemple C++ d'une classe (1/3)

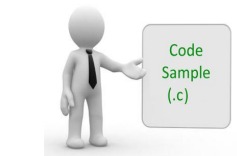
Définition de la classe Point

main.cpp

```
#include <iostream>
class Point
{
    public:
        void init(int, int);
        void translate(int, int);
        void print()
        {
            std::cout<<" ("<<x<<" , "<<y<<" ) ";
        }

        int x;
        int y;
};
```

classe



01_ClassePoint

méthodes

attributs

2.1 Exemple C++ d'une classe (2/3)

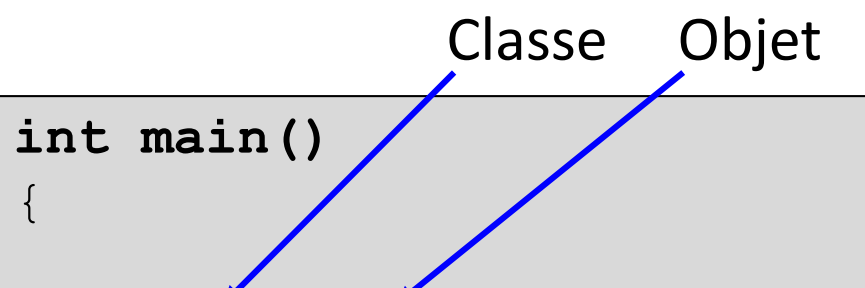
Définition des méthodes de la classe

main.cpp

```
...  
  
void Point::init(int _x, int _y)  
{  
    x=_x;  
    y=_y;  
}  
  
void Point::translate(int delta_x, int delta_y)  
{  
    x += delta_x;  
    y += delta_y;  
}  
  
...
```


2.1 Exemple C++ d'une classe (3/3)

Utilisation d'objets de la classe



```
int main()
{
    Point p1;

    p1.init(-1,2);
    p1.print();
    p1.translater(10,10);
    p1.print();

    return 0;
}
```

main.cpp



2. Table des matières

1. Introduction

1. Paradigme de programmation
2. Classe et objet
3. Exemple de programme en C++
4. **Les caractéristiques d'un objet**

2.1 Caractéristiques d'un objet

Un objet possède :

Un état interne

Contient l'ensemble des valeurs qui décrit à tout instant l'état de l'objet. En principe cet état n'est pas directement accessible à l'utilisateur, il n'est que partiellement accessible au travers de l'interface.

Une implémentation

C'est la façon dont l'objet est réalisé. L'implémentation n'est pas accessible à l'utilisateur. Deux classes ayant la même interface peuvent avoir une implémentation complètement différentes.

Il appartient à une classe

La classe est le "moule" à partir duquel on crée un objet. Un objet est une *instanciation* d'une classe.

Il est identifiable (nommable)

On peut nommer, identifier, distinguer un objet d'un autre objet, même s'il sont de la même classe.



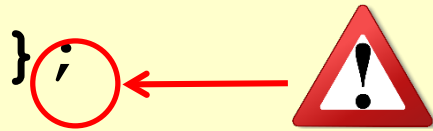
2. Table des matières

1. Introduction
- 2. Déclaration d'une classe et de ses attributs**
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Modules, membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé

2.2.1 Déclaration d'une classe en C++

Syntaxe pour la déclaration d'une classe

```
class NomDeLaClasse  
{  
    public:  
        int    uneFonctionMembre() ;  
        float unAttribut;  
    private:  
        char unAutreAttribut;
```





2.2.1 Déclaration d'une classe en C++





Contrôle d'accès aux membres

Membre **public** : Accessible par tous

Membre **privé** : Accessible par les objets de la classe (mode par défaut)

```
class XYZ{  
    public:  
        void fctPub() {..}  
        int varPub;  
    private:  
        void fctPriv() {..}  
        int varPriv;  
};
```

Accès **hors**
de la classe

```
int main()  
{  
    XYZ x;  
  
    x.fctPub();   
    x.varPub=3;   
  
    x.fctPriv();  Compile error  
    x.varPriv=6;  Compile error  
}
```

2.2.1 Déclaration d'une classe en C++

Contrôle d'accès aux membres

Membre **public** : Accessible par tous

Membre **privé** : Accessible par les objets de la classe (mode par défaut)

Accès **depuis la classe** (XYZ ::)

Accès **depuis la classe** (XYZ ::)

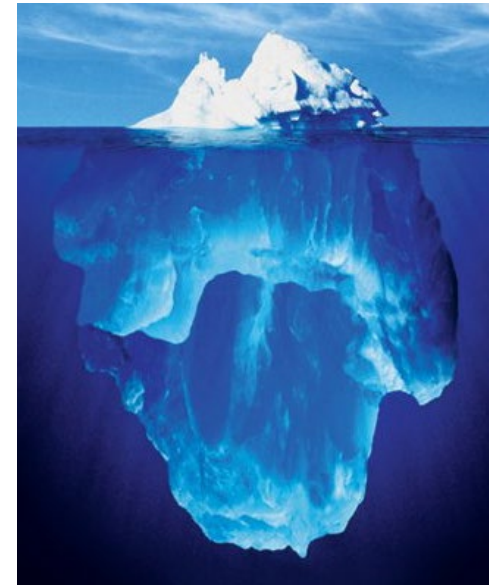
```
class XYZ
{
    public:
        void fctPub();
        int varPub;
    private:
        void fctPriv();
        int varPriv;
};
```

```
void XYZ::fctPub()
{
    varPub=3; ✓
    varPriv=6; ✓
}
void XYZ::fctPriv()
{
    varPub=3; ✓
    varPriv=6; ✓
}
```

2.2.2 Notion d'encapsulation

- 1) Permet **l'évolutivité des programmes**
l'implémentation et le type de données
sont cachés (peuvent évoluer)
 - Organisation du projet
- 2) Garantit la **cohérence/sécurité** des données
 - ⇒ Gestion de l'accès aux données
 - `private`, `public`
 - ⇒ Accès aux données par des méthodes
 - **Accesseur**: `getX()`
 - **Modificateur**: `setX()`

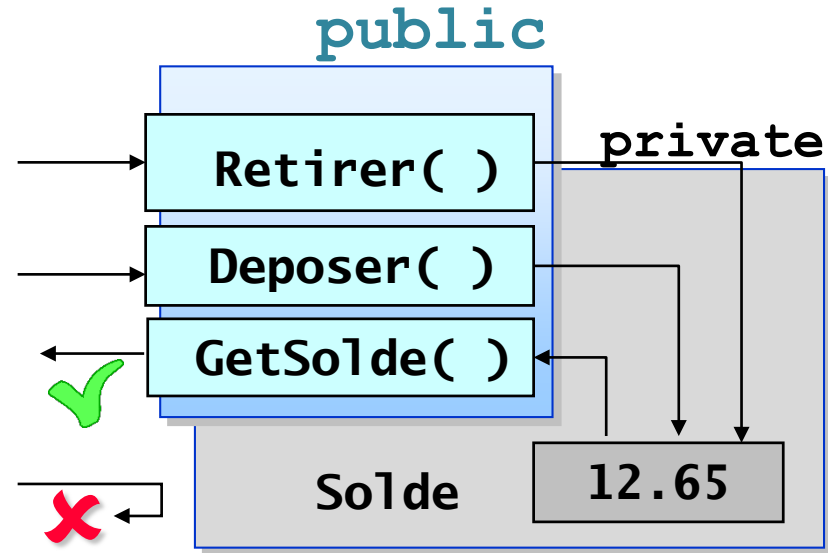
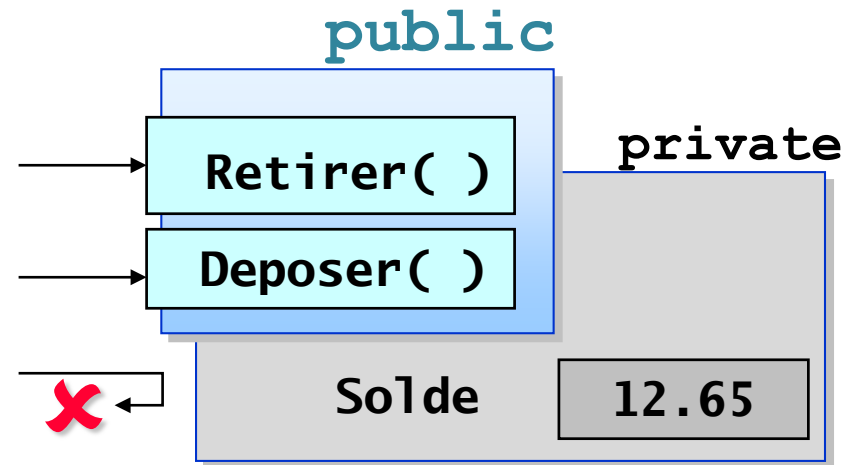
visible



invisible

2.2.2 Notion d'encapsulation

- **Contrôle d'accès** Principe:
 - Les fonctions sont publiques **public**
 - Les données sont privées **private**
- L'accès au **Solde** se fait par des méthodes.
 - Les **modificateurs**
 - `Retirer()`
 - `Deposer()`
 - Les **accesseurs**
 - `GetSolde()`



2.2.2 Portée des attributs et des méthodes

- Certaines méthodes doivent être publiques :
 - les constructeurs (en général),
 - les accesseurs,
 - les modificateurs.
- On **recommande** de
 - déclarer les attributs privés
 - les rendre accessibles par des méthodes publiques (accesseurs et modificateurs).

Note:

Un membre déclaré sans modificateur d'accès sera **private** par défaut (accessible uniquement depuis méthodes de la classe)

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
- 3. Définition des méthodes**
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Modules, membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé



2.3 Définition des méthodes

En-ligne

- Dans la déclaration de la classe
- Mot-clef `inline` (implicite)
- Pour des fonctions courtes
- Implémentation sous forme de MACRO

Séparée

- **Prototype** dans déclaration de la classe
- **Définition**: hors de la déclaration de la classe
- **Opérateur de portée** `::`

2.3 Définition des méthodes

Définition en-ligne `inline`

`inline`
facultatif

```
class Point
{
    public:
        int getX()
        {
            return x;
        }

```

```
        inline void setX( int _x)
        {
            x = _x;
        }

```

```
    private:
        int x, y;
};
```

2.3 Définition des méthodes

Définition séparée

```
class maClasse
```

```
{
```

```
    int uneFonctionMembre();
```

```
};
```

Déclaration de la classe

Prototype

Nom classe

Opérateur de portée ::

```
int maClasse::uneFonctionMembre()
```

```
{
```

```
    // implémentation
```

```
}
```

Définition
séparée: hors de
la déclaration de
la classe



2.3 Définition des méthodes

Définition séparée

```
class Point
{
    public:
        void display();
    private:
        int x, y;
};
```

Déclaration de la classe

```
void Point::display()
{
    cout << '(' << x << ',' << y << ')';
}
```

2.3 Diagramme de classe UML

```
class Point
{
    public:
        void display();
        char label;
    private:
        int x, y;
};
```

Point
-x : int -y : int +label : char
+display()



2.3 Définition des méthodes

Organisation d'un projet C++

Deux fichiers par classe

- Interface : **classe.h** (Déclaration classe et membres)
- Implémentation : **classe.cpp** (Définition méthodes)
- Directives de non-inclusion multiple dans les interfaces

```
#ifndef CLASSE_H  
#define CLASSE_H  
...  
#endif
```

classe.h

- Inclure la déclaration dans l'implémentation et le main.cpp

```
#include "classe.h"
```

classe.cpp

- Bonne habitude : mettre un commentaire explicatif en en-tête

Note : un fichier *peut* regrouper plusieurs classes (mais fortement déconseillé dans la pratique !)

2.3 Définition des méthodes

Organisation d'un projet C++

Classe Point

point.h

```
#ifndef POINT_H
#define POINT_H
class Point{
public:
    Point();
    int getX(){
        return x;
    };
#endif
```

Interface

point.cpp

```
#include "point.h"

Point::Point()
{
    x=0; y=0;
}
```

Implémentation

Objet de la classe Point

main.cpp

```
#include "point.h"

int main()
{
    Point ptA;
}
```

2.3 Définition des méthodes

Organisation d'un projet C++

- Directives de non-inclusion : alternative **#pragma once**
 - directive du préprocesseur non-standard mais largement supportée

point.h

```
#ifndef POINT_H
#define POINT_H
class Point{
public:
    Point();
    int getX(){
        return x;
    };
#endif
```



point.h

```
#pragma once

class Point{
public:
    Point();
    int getX(){
        return x;
    };
};
```

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
- 4. Instanciation d'un objet**
5. Les constructeurs
6. Le destructeur
7. Modules, membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé

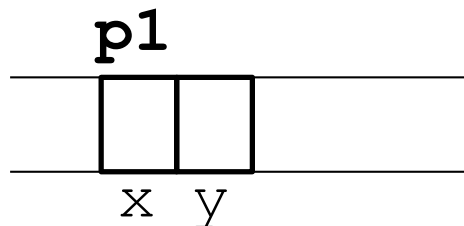
2.4 Instancier (déclarer) des objets

- **Instancier**: créer un objet à partir d'une classe
- Chaque objet dispose de son propre jeu d'attributs
- La syntaxe est la même que pour déclarer des variables.

Instanciation automatique

```
int main()
{
    Point p1;
    p1.afficher();
}
```

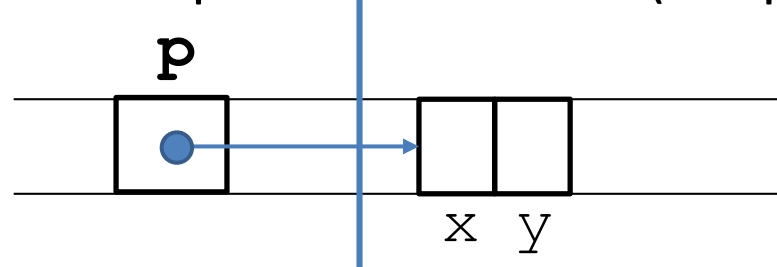
Allocation sur la pile (stack):



Instanciation dynamique

```
int main()
{
    Point *p = new Point;
    p->afficher();
    delete p;
}
```

Sur la pile Sur le tas (heap)



2.4 Commentaires

un **type** est utilisé pour « déclarer » des **variables**,
une **classe** est utilisée pour « instancier » des **objets**

- Les attributs et les méthodes sont relatifs à un objet: il est nécessaire de commencer par instancier un objet à partir d'une classe avant de pouvoir accéder à ses membres. C'est pour cette raison que les membres d'une classe sont parfois appelés **attributs d'instance** ou **méthodes d'instance**.
- Avantages de l'instanciation dynamique :
 - Portée globale
 - Rapidité de passage de paramètre par adresse (ou référence).



Série
2.1

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
- 5. Les constructeurs**
6. Le destructeur
7. Modules, membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé



2.5 Principe du constructeur

Problème: Entre le moment où un objet est instancié et son initialisation l'état interne est **indéterminé**

```
Point sommetA;           // instanciation d'un  
                           // Point non initialisé  
sommetA.afficher(); // -> (?, ?)  
sommetA.init(3, 5); // initialisation  
sommetA.afficher(); // -> (3, 5)
```

But du constructeur

Permettre d'initialiser un objet lors de son instanciation.

2.5 Principe du constructeur

On a vu :

```
Point pointA;  
pointA.init(3,5);
```

- 1) Instanciation d'un objet
- 2) Initialisation de l'objet

On peut initialiser l'état interne d'un objet lors de son instanciation grâce à un **constructeur** (méthode spéciale)

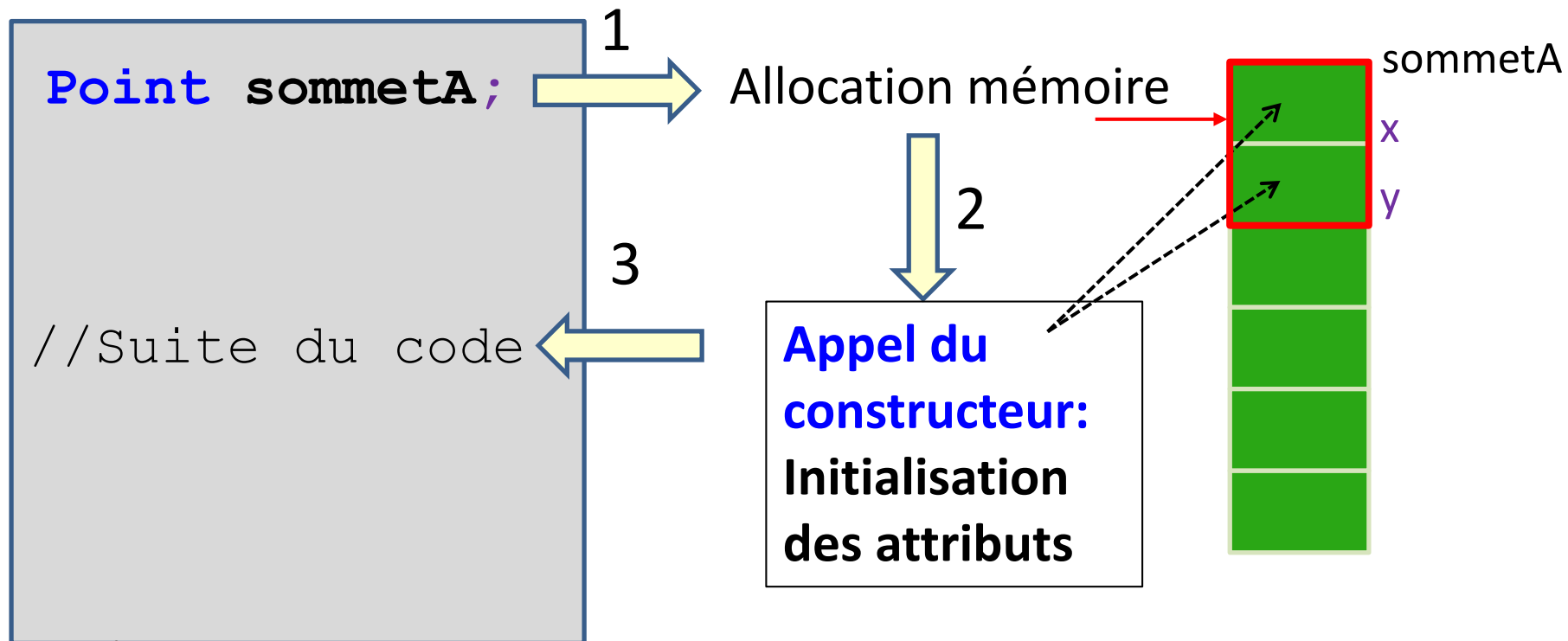
```
Point pointA(3,5);
```

Instanciation + initialisation
d'un objet

2.5 Constructeurs

Quand on **instancie** un objet:

1. La mémoire nécessaire est **allouée**
2. Le constructeur de la classe est **automatiquement appelé**.
Son rôle est d'initialiser les attributs de l'objet.

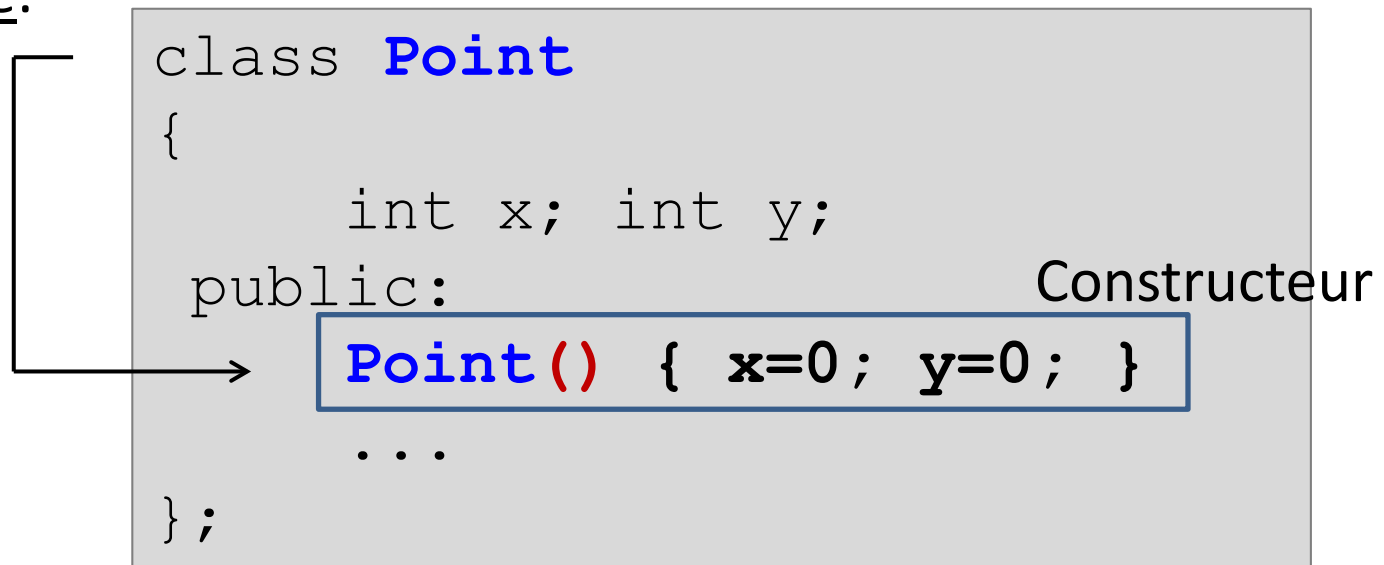


2.5 Définition d'un constructeur

Un **constructeur** est une **méthode** avec les caractéristiques suivantes:

- Son nom est celui de la classe
- Il ne contient pas d'instruction `return`
- Il ne possède pas de type de retour (même pas `void`)

Exemple:



```
class Point
{
    int x; int y;
public:
    Point() { x=0; y=0; }
    ...
};
```

Constructeur

2.5 Les différents constructeurs

Un constructeur peut avoir des **paramètres**, il **peut être surchargé**.
Selon sa signature, on l'appelle alors le constructeur:

1. **par défaut**^{*} : **Point()**

Pas de paramètre (**défaut** de paramètre)

2. **«standard»** : **Point(T1, T2, ...)**

Plusieurs paramètres de tout types.

3. **par recopie**^{*} : **Point(Point&)**

Un paramètre qui est une référence sur un objet de la classe

4. **de conversion** : **Point(T)**

Un seul paramètre de tout type sauf de la classe (ci-dessus).

^{*}Le compilateur le propose



2.5.1 Constructeurs par défaut

Deux façons de définir un constructeur par défaut (**sans paramètres**)

```
class Point
{
public:
    Point()
    {
        x=0; y=0;
    }
    ...
};
```

```
class Point
{
public:
    Point(int a=0, int b=0);
    ...
};

Point::Point(int a, int b)
{
    x=a; y=b;
}
```

Peut être défini en ligne ou séparé



2.5.1 Constructeurs par défaut (remarques)

- A. Le constructeur par défaut est appelé chaque fois qu'un objet doit être créé sans arguments.
- B. Le compilateur fournit un constructeur par défaut, **si le programmeur n'a écrit aucun autre constructeur!**
- C. Le constructeur par défaut proposé par le compilateur ne fait rien: { } (pour l'instant...).

Exemples d'appels du constructeur par défaut

```
Point x = Point();  
Point y;  
Point z(); Ne jamais mettre de parenthèses!!  
Ca correspond à la déclaration d'une fonction z  
Point t[10];           // 10 appels de Point()  
Point *p = new Point;   // équiv. à: p = new Point()  
Point *q = new Point[10]; // 10 appels de Point()
```



2.5.2 Constructeurs **standard**

Constructeur qui reçoit des arguments de tout type

```
class Point
{
    public:
        Point(int a, int b);
        ...
};
```

```
Point::Point(int a, int b)
{
    x=a; y=b;
}
```

```
class Point
{
    public:
        Point(int a=0, int b=0);
        ...
};
```

```
Point::Point(int a, int b)
{
    x=a; y=b;
}
```

Avec des valeurs d'arguments par défaut, fait aussi office de constructeur par défaut



2.5.2 Constructeurs **standard** (remarques)

Le constructeur standard est appelé chaque fois qu'un objet doit être créé avec des arguments.

Exemples d'appels du constructeur standard

```
Point p1(1, 2);  
Point p2 = Point(1, 2);  
Point p3[10];  
//Point p3[10] = {Point(5, 3)};  
Point *p4 = new Point(6, 3);
```

```
Point *p5 = new Point[10];  
for (int i=0; i<10; i++)  
    p5[i] = Point(2, 3);
```

!
!

Impossible en une
seule opération

2.5.2 Constructeur standard, pointeur **this**

Le pointeur **this** contient l'adresse de l'objet courant.

1) Éviter le masquage de l'attribut par un paramètre, p.ex:

```
Point::Point(int x, int y)
{
    this->x = x;
    this->y = y;
}
```

2) Améliorer la lisibilité du code source: on distingue mieux les méthodes de l'objet courant et les fonctions globales

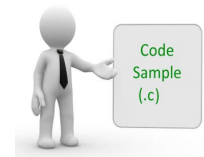
3) Comparer l'adresse de l'objet courant avec celle d'un autre objet.

2.5.3 Constructeurs de conversion

- Un seul argument obligatoire, d'un autre type simple.
- **Conversion**, car il convertit un type simple en un objet de la classe du constructeur.

Exemple: constructeur de conversion de `int` → `Point`

```
class Point
{
    public:
        Point(int a) {x = y = a;}
        ...
};
```



02_ConstrConversion

Exemples d'appels du constructeur standard

```
Point p = Point(2);
Point q(3);
Point q = 4;
q = 5;
```

} Conversion Implicite

2.5.3 Constructeur de conversion (explicit)

- Interdiction de la conversion implicite : **explicit**

Exemple :

```
class Point
{
    public:
        explicit Point(int a) {x = y = a;}
};
```



ADV_Explicit

```
Point p = Point(2); // OK conv. explicite
Point q(3);         // OK conv. explicite
Point q = 4;        // ERROR conv. implicite
q = 5;              // ERROR conv. implicite
```

2.5.4 Constructeurs **par copie** (clonage)

Constructeur dont le premier paramètre est une référence constante sur ce type, et sans autre paramètre obligatoire.

Exemple :

```
Point::Point (const Point& pt)
{
    x = pt.x;
    y = pt.y;
}
```

Utilisation :

```
Point p1 = Point(5, 10);
Point p2(p1);

Point p3 = p1;
```



2.5.4 Constructeurs **par copie** (clonage)

Ce constructeur est souvent appelé **implicitement**

- A l'**initialisation** d'un objet, avec l'opérateur =

```
MaClasse unObjet = unAutreObjet;
```

- Passage d'un objet en argument à une méthode

```
maFonction(monObjet);
```

- Lors du retour (`return`) d'un objet dans une méthode

```
return monObjet;
```



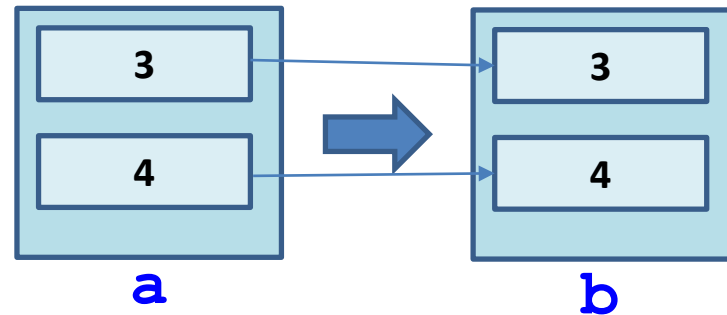
02_CopyConstructorCalls

2.5.4 Constructeurs **par copie** (clonage)

- Le compilateur **propose** automatiquement un **constructeur par copie trivial (bit à bit)**. On parle de "**copie en surface**".

```
Point a(3, 4);  
Point b = a;
```

Fait une copie 1 à 1 des tous les attributs



- S'il est insuffisant pour cloner des objets avec «des bouts dehors», le programmeur doit faire son constructeur avec "**copie en profondeur**".

2.5.4 Constructeurs par copie (1/3)

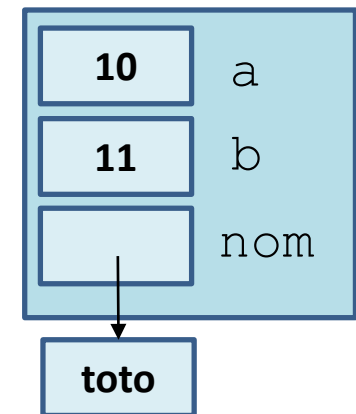
Problème lorsque des attributs sont des pointeurs !!

Exemple avec un `Point` qui contient un nom !

```
class PointN {  
public:  
    PointN (int a, int b, char *s = "")  
    {  
        x = a; y = b;  
        nom = new char[strlen(s) + 1];  
        strcpy(nom, s);  
    }  
private:  
    int x, y;  
    char *nom;  
};
```

Constructeur
standard

p1



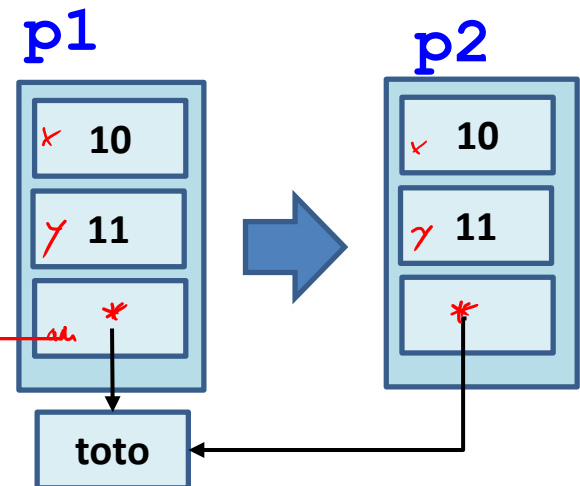
2.5.4 Constructeurs par copie (2/3)

Problème lié à la copie en surface **✗**

```
class PointN {  
public:  
    PointN (const PointN &p)  
    {  
        x = p.x; y = p.y;  
        nom = p.nom;  
    }  
    ...  
private:  
    int x, y;  
    char *nom;  
};
```

```
int main()  
{  
    PointN p1(10,11,"toto");  
    PointN p2(p1)  
}
```

Essai de
constructeur par
recopie



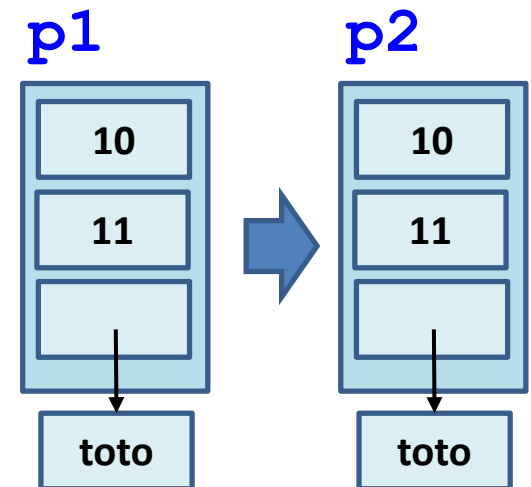
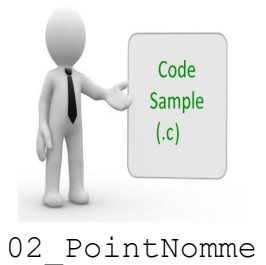
2.5.4 Constructeurs par copie (3/3)

Solution: copie en profondeur ✓

Constructeur par recopie

```
class PointN {  
public:  
...  
    PointN (const PointN &p) {  
        x = p.x; y = p.y;  
        nom = new char[strlen(p.nom)+1];  
        strcpy(nom, p.nom);  
    }  
...  
};
```

```
int main()  
{  
    PointN p1(10,11,"toto");  
    PointN p2(p1);  
}
```





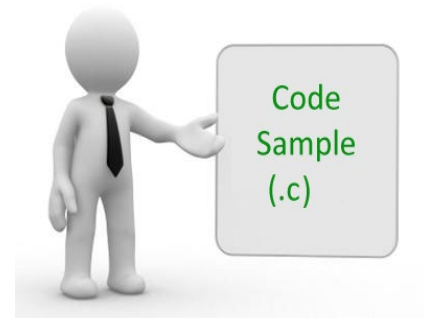
2.5 Appels de constructeurs (rappel)

- 1) `Point b = Point(5, 6);`
- 2) `Point a(3, 4);`
- 3) `Point e = 7; //équivalent à Point e = Point(7)`
- 4) `Point *pt;`
 `pt = new Point(1, 2);`
- 5) `Point apex;`
- 6) `Point hexagone[6];`

2.5 Appels de constructeurs (exemples)

- Initialisation d'**objets temporaires, anonymes**

```
cout << Point(10,0).distance(Point(3, 4));
```



02_CopyConstructorCalls

2.5 Appels de constructeurs (exemples)

- Deux interprétations différentes pour le = :

A. `Point p = Point(1, 2);`

B. `Point p;
p = Point(1, 2);`

A. Initialisation d'objets déclarés (signe = **initialisation**)

`Point p = Point(1, 2);` ➡ **constructeur standard**

- 1) Cette expression crée l'objet p et **l'initialise** en rangeant dans p.x et p.y les valeurs 1 et 2. (Appel du constructeur standard)

2.5 Appels de constructeurs (exemples)

B) **Affectation** d'objets déclarés (signe = **affectation**)

```
Point p;  
p = Point(1, 2);
```

⇒ **constructeur par défaut**
⇒ **constructeur «standard»**

Operations:

- 1) Crée un objet p (constructeur par défaut)
- 2) Crée un point anonyme de coordonnées (1,2)
- 3) Affectation (=) des valeurs des attributs du point anonyme aux attributs de p.

Risque si "=" n'est pas surchargé pour des copie en profondeur!

Bilan

- A. **Initialisation** : 1 appel constructeur (standard)
- B. **Affectation** : 2 appels constructeurs + 1 affectation

2.5 Liste d'initialisation des membres

Syntaxe pour l'initialisation de membres à l'appel du constructeur. Exemple:

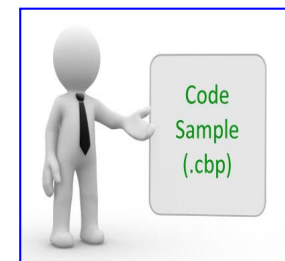
```
Point::Point(int a, int b) : x(a), y(b)
{
    // autres opérations
}
```

Une **liste d'initialisation** est **indispensable pour**:

- initialiser un **membre constant**
- initialiser un **membre de type référence**
- initialiser un **membre d'un type sans constructeur par défaut**

Valable pour tous les constructeurs

Bonne pratique!! 



2.5 Commentaire (rappel)

- Le constructeur par défaut fourni par le compilateur **n'est plus disponible** si le programmeur a défini un constructeur!
- Si un constructeur a des **valeurs par défaut** pour ses paramètres, celles-ci doivent figurer **dans la déclaration** de la classe (fichier d'interface : **classe.h**)
- **Bonne pratique** de redéfinir le constructeur par défaut, en profiter pour initialiser les membres avec des valeurs par défaut
- La **déclaration d'un tableau d'objets** génère autant d'appels au constructeur par défaut que d'éléments dans le tableau.



Série 2.2

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
- 6. Le destructeur**
7. Membres statiques, `this`, `friend`
8. Diagramme UML
9. Résumé





2.6 Le destructeur

- Appel implicite à la destruction d'un objet (auto et dynamique)
- Syntaxe :
 - Nom de la classe préfixé par: ~
 - Pas de type de retour (ni void)
 - Pas de paramètres
 - Un seul destructeur
- Destructeur par défaut généré par le compilateur (trivial)
- Permet au programmeur de détruire proprement un objet:
 - Libérer la mémoire (membres dynamiques)
 - Fermer les flux (fichiers...)
 - Stopper les threads

2.6 Le destructeur



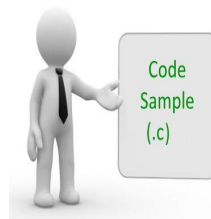
Un destructeur est appelé **AVANT** la destruction d'un objet en mémoire

```
class PointN
{
    public:
        ~PointN() ;
};

PointN::~~PointN()
{
    delete [] nom;
}
```

```
int main()
{
    PointN *p=new PointN;
    ...
    delete p;

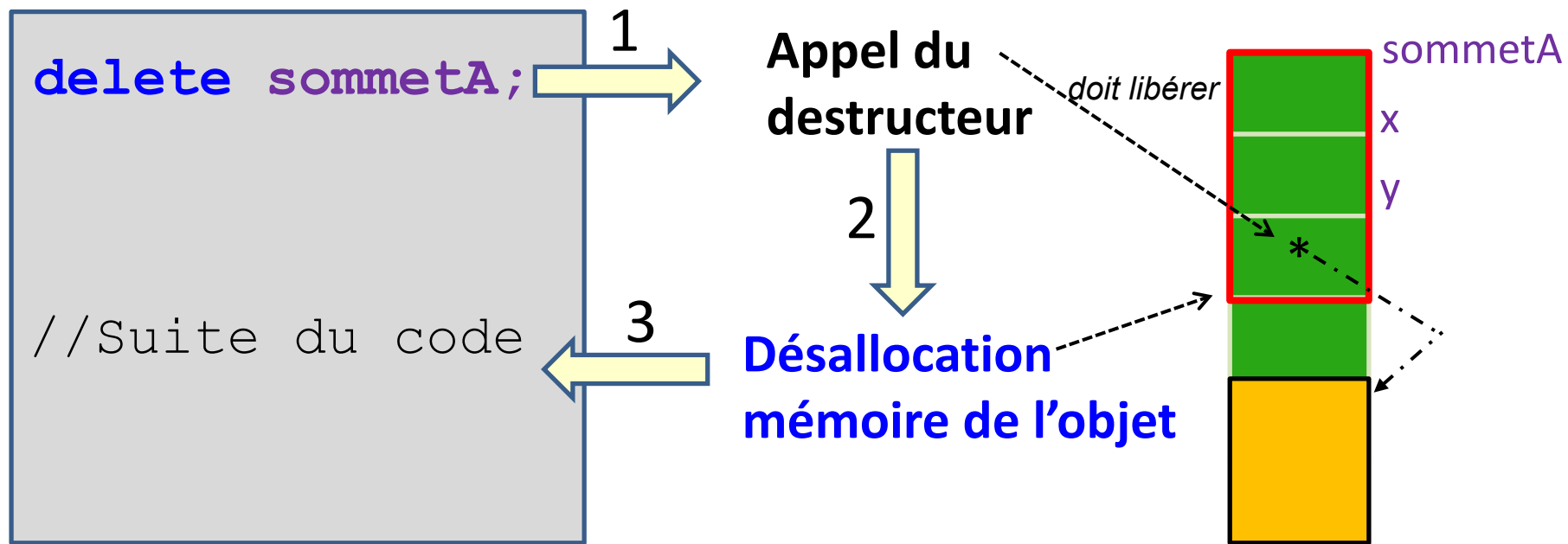
    PointN *q=new PointN[10];
    ...
    delete [] q;
}
```



2.6 Destructeur

Quand on **détruit** un objet:

1. le destructeur de la classe est **automatiquement appelé**.
Son rôle est terminer proprement.
2. la mémoire réservée pour l'objet est libérée (**désallouée**).



2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
- 7. Membres constants**
8. Membres statiques, `this`, `friend`
9. Diagramme UML
10. Résumé

2.7 Membres constants

1) Attributs constants:

- Un attribut d'une classe peut être qualifié **const**.
- Initialisé lors de la construction d'un objet.
- Plus modifiable par la suite.

```
class Segment {  
    ...  
    const int ID;  
public:  
    Segment(int x,int y, int num) ;  
};
```



Const.pdf

```
Segment::Segment (... ,int num) { ... , ID = num; }
```



Erreur de compilation car ID est constant!!

```
Segment::Segment (... ,int num) : ID(num) { ... }
```



Solution: Liste d'initialisation

2.7 Membres constants

2) Méthodes constantes:

le mot-clé **const** en fin d'en-tête d'une méthode indique que l'état de l'objet, par lequel la méthode est appelée, n'est pas changé du fait de l'appel.

C'est une manière de déclarer qu'il s'agit d'une **fonction de consultation de l'objet**, non d'une fonction de modification

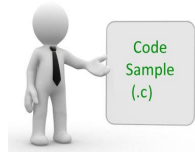
```
class Point
{
    // méthode qui peut modifier l'objet
    void setXY(float newX, float newY);

    // méthode qui ne peut pas modifier l'objet
    float getX( ) const;
};
```

2.7 Membres constants

2) Méthodes constantes avec objet constant

```
const Point pt;  
  
double x = pt.getX();
```



08_PointConstant

Le point **pt** est constant, il ne peut pas être modifié par `getX()`
=> la qualification `const` de la fonction `getX()` est **indispensable** pour que l'expression `pt.getX()` soit acceptée par le compilateur.

Il est **fortement conseillé** de qualifier `const` toute méthode qui peut l'être : cela élargit son champ d'application.

2.7 Membres constants

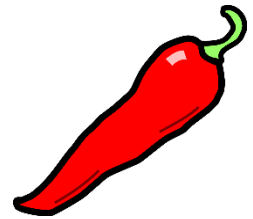
2) Méthodes constantes: Surcharge

La mot-clef **const** change la signature d'une méthode :

⇒ **On peut surcharger une fonction membre non constante par une fonction membre constante ayant, la même entête.**

- La fonction non constante sera appelée par les objets variables,
- la fonction `const` sur les objets constants.

2.7 Membres constants

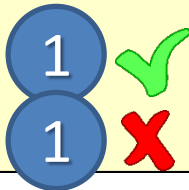


2) Méthodes **constantes**: Surcharge

```
class Point{  
    int x,y;  
    public:  
        int X() const {return x;}  
        int& X()      {return x;}  
};
```

```
const Point pC(2, 3);  
int r;
```

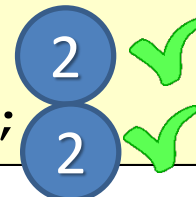
```
...  
r = pC.X();  
pC.X() = 15;
```



p.X() retourne la valeur x
=> **ne peut pas être** une Lvalue

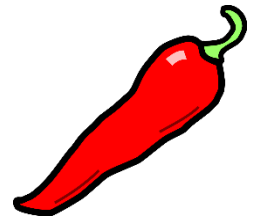
```
Point p(4,5);  
int r;
```

```
...  
r = p.X();  
p.X() = 15;
```



p.X() retourne une référence sur x => **peut être** une Lvalue

2.7 Membres constants



2) Méthodes constantes: Surcharge

```
class Point{  
    int x,y;  
    public:  
    int& X() const {return x;}  
};
```



Erreur de compilation

Comme X() retourne une référence (=> possibilité d'être une Lvalue) , le compilateur ne peut pas garantir que la méthode soit constante.



2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Membres constants
- 8. `this`, membres `static`, `friend`**
9. Diagramme UML
10. Résumé



2.8.1 Pointeur **this**

Le **pointeur this** contient l'adresse de l'objet courant.

Justification:

- Les méthodes sont toujours appelées **depuis un objet**
- Pour le compilateur, **l'adresse de cet objet est passé implicitement** à la méthode appelée.
- **this**: pointeur sur l'objet qui a servi à appeler la méthode

Le **déréférencement** de **this** donne l'objet courant!

```
return *this;
```

Retourne l'objet lui-même

```
Point temp(*this);
```

Crée une copie de l'objet courant
(constructeur par copie)

2.8.1 Pointeur **this**

Le **pointeur this** contient l'adresse de l'objet courant.

1) Éviter le masquage de l'**attribut** par un **paramètre**. p.ex:

```
Point::Point(int x, int y)  
{  
    this->x = x;  
    this->y = y;  
}
```



04_this

2) Améliorer la lisibilité du code source: **on distingue mieux les membres de l'objet courant** des paramètres et identifiants visible (portée globale)

3) **Comparer l'adresse** de l'objet courant avec celle d'un autre objet.

2.8.2 Membres statiques

Deux catégories de membres (attributs et méthodes):

A) Membres d'instance (cas connu)

1. Chaque objet d'une classe a ses propres **attributs**.
2. Les **méthodes** sont appelées à partir des objets d'une classe .

B) Membres de classe

1. Un **attribut** de classe est unique pour tous les objets d'une classe.
2. Une **méthode** de classe peut être appelée à partir de la classe.

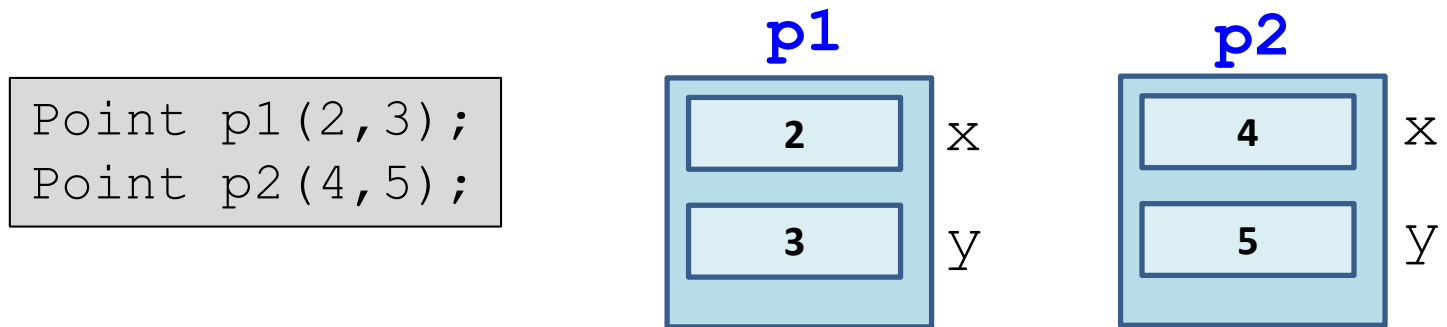
2.8.2 Membres statiques

A) Membres d'instance (cas connu):

Chaque objet (instance) a sa propre copie

1. Attributs d'instance des objets d'une classe

- chaque **objet possède les mêmes attributs** (x et y)
- chaque objet a ses propres valeurs pour ces attributs.



2. Méthodes d'instance

Nécessite d'instancier un objet pour appeler une méthode

```
p1.show();  
p2.show();
```

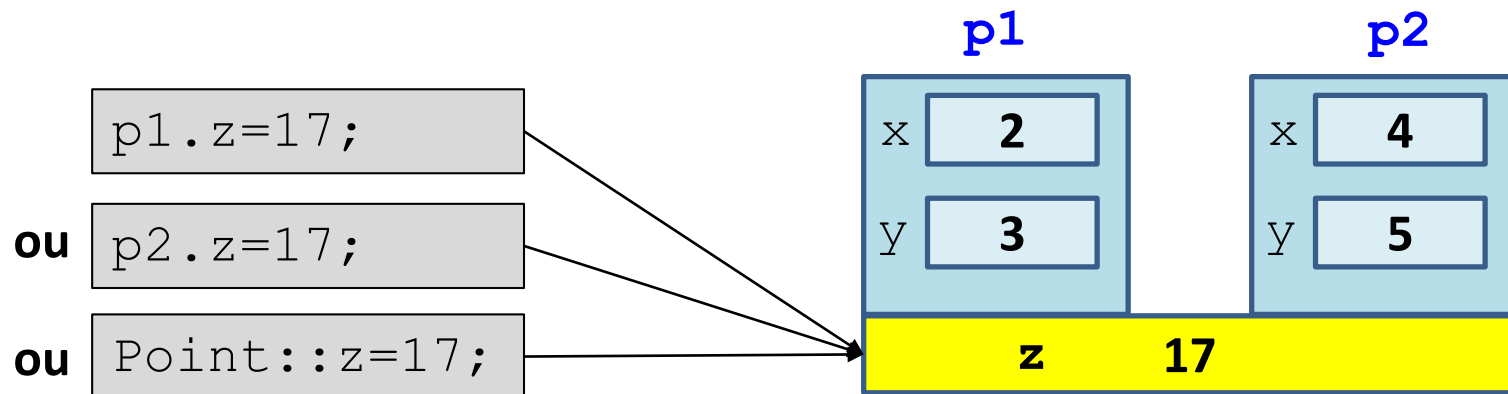
2.8.2 Membres statiques

B1. Variables statiques (attributs de classe)

- **Communs à tous les objets** d'une classe
- **Existent indépendamment de tout objet** de cette classe.
- associés à une classe et non à une instance

⇒ Le mot-clé **static** précède la déclaration de l'attribut.

Par exemple:



2.8.2 Membres statiques

B1. Variables statiques (attributs de classe)

Point.h

```
class Point
{
private:
    static int nbPoints;
public :
    Point(int x, int y)
    { ...
        nbPoints ++;
    }
};
```

← Déclaration

← Exemple d'application:
Compter le nombre d'objets
instanciés

Point.cpp

```
int Point::nbPoints = 0;
```

← Initialisation dans le
fichier .cpp



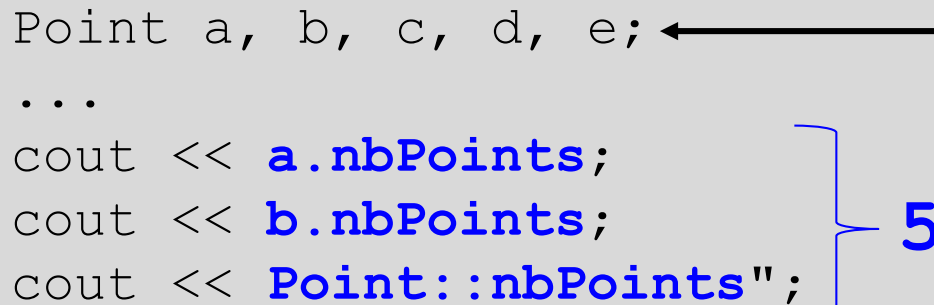
2.8.2 Membres statiques

B1. Attributs statiques (*de classe*)

La visibilité et les droits d'accès des membres statiques ont les mêmes que les membres ordinaires.

main.cpp

```
Point a, b, c, d, e;
...
cout << a.nbPoints;
cout << b.nbPoints;
cout << Point::nbPoints";
```



A chaque instantiation,
le compteur **nbPoints**
augmente de 1

On accède au même attribut

2.8.2 Membres statiques

B2. Méthodes statiques (*de classe*)

Une méthode statique n'est pas attachée à un objet.

⇒ de sa classe, elle ne peut référencer que les fonctions et les membres statiques.

⇒ ces fonctions de classe n'ont pas accès aux attributs d'instance (par exemple: x et y de Point)

⇒ elle ne dispose pas du pointeur `this`,

⇒ elles permettent de fournir des services à d'autres classes **sans passer obligatoirement par une instanciation.**

- Le mot clé **static** précède l'entête de la méthode.

Exemple

```
p1.getZ();
```

```
Point::getZ();
```

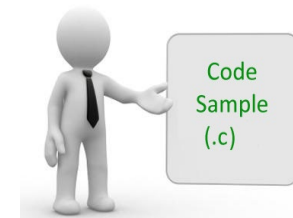
2.8.2 Membres statiques

B2. Méthode statique

```
class Point {  
    private:   
        int x, y;  
        static int nbPoints;  
  
    public:  
        static int getNbPoints() {  
            return nbPoints;  
        }  
  
    Point(int a, int b) {  
        x = a; y = b;  
        nbPoints++;  
    }  
};
```

Bonne pratique

Attribut privé, et accès
via un get...



05_static

```
cout << Point::getNbPoints();
```

```
cout << pt1.getNbPoints();
```

2.8.3 Fonctions et classes amies

Il est possible de **passer outre la protection** d'accès aux membres privés d'une classe grâce au mot-clé **friend**.

Ce «privilège» peut être accordé à :

1. Une fonction

Une fonction amie d'une classe est une fonction qui, sans être membre de cette classe, a le droit d'accéder à tous ses membres, aussi bien publics que privés.

2. Une classe

Une classe amie d'une classe C est une classe qui a le droit d'accéder à tous les membres de C .

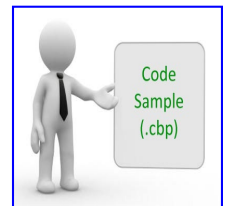
2.8.3 Fonctions et classes amies

1. Une **fonction amie** doit être déclarée dans la classe qui accorde le droit d'accès et est définie hors de la classe.

```
class Point
{ int x;
  public:
    friend void fctExt(Point &p) ;
};

void fctExt(Point &p)
{
    p.x = 0;           -> OK car fctExt() est amie
}

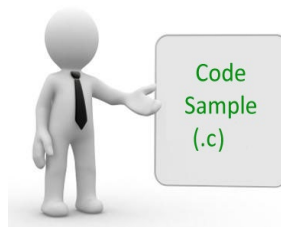
int main() {
    Point p;
    // p.x=0;          -> Impossible car x est privé
    fctExt(p) ;
}
```



2.8.3 Fonctions et classes amies

2. Une **classe amie** d'une classe *hôte* est une classe qui a le droit d'accéder à tous les membres de *hôte*.
- Une telle classe doit être déclarée dans la classe *hôte* précédée du mot **friend**, indifféremment parmi les membres privés ou parmi les membres publics de *hôte*.

```
class Hote {  
    friend class Amie;  
    int x;  
public:  
    Hote(int a):x(a) {}  
};
```



```
class Amie {  
public:  
    void affHote() {  
        Hote h(16);  
        cout<<h.x;  
    }  
};
```

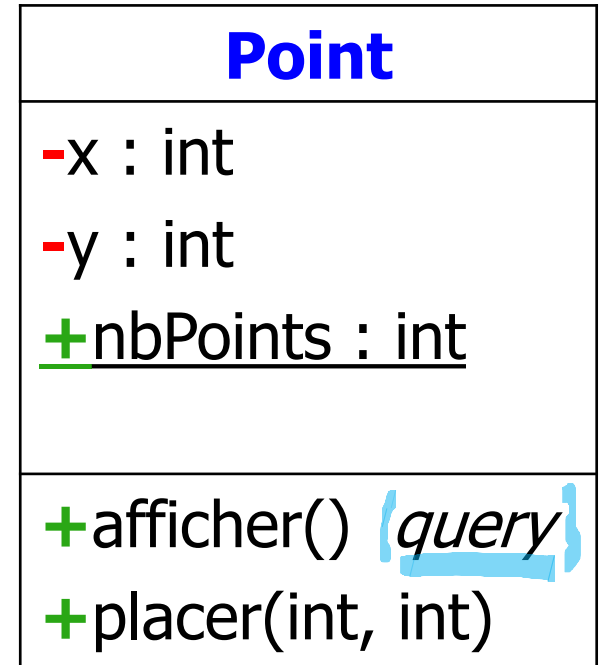
Accède directement à l'attribut

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Membres constants
8. Membres statiques, `this`, `friend`
- 9. Diagramme UML**
10. Résumé

2.9 Diagramme de classe UML

```
class Point
{
    public:
        static int nbPoints;
        void afficher() const
        {
            cout << ...;
        }
        void placer(int _x, int _y)
        {
            x=_x;
            y=_y;
        }
    private:
        int x, y;
};
```



2.9 Diagramme de classe UML

- Dans un diagramme UML, la portée d'un membre est indiquée en préfixant le membre par le signe
 - +** s'il est **public**
 - s'il est **privé**
- Le type d'une méthode est représenté de la même façon que le type d'une variable. Il est omis pour une fonction `void`.

Point
-x : int -y : int <u>+nbPoints</u> : int
+afficher() <i>query</i> +placer (int, int)

- Le **soulignement** indique qu'il s'agit d'un **membre statique**.
- L'indication « **query** » signifie qu'une méthode est qualifiée `const`. Elle ne peut pas modifier d'attribut de l'objet



Série 2.2

2. Table des matières

1. Introduction
2. Déclaration d'une classe et de ses attributs
3. Définition des méthodes
4. Instanciation d'un objet
5. Les constructeurs
6. Le destructeur
7. Membres constants
8. Membres statiques, `this`, `friend`
9. Diagramme UML
- 10. Résumé**

2.10 Objets et Classes en C++ (résumé)

- `class MaClasse { ... };` déclare une **classe**
- `MaClasse obj1;` déclare une **instance** (un **objet**) de classe `MaClasse`
- **Les attributs** d'une classe se déclarent comme les champs d'une structure

```
class MaClasse
{ ...
    type attribut;
};
```

- **Les méthodes** d'une classe se déclarent comme des fonctions, dans la déclaration de la classe

```
class MaClasse
{ ...
    type methode(type1 arg1, ...);
};
```

2.10 Objets et Classes en C++ (résumé)

- Encapsulation et interface :

```
class MaClasse
{
    private:
        // attributs et méthodes privées
        ...
    public:
        // interface : attributs et méthodes    publiques
        ...
};
```

- L'attribut particulier **this** est un pointeur sur l'instance courante de la classe. Exemple: d'utilisation : `this->monAttribut`

2.10 Construction (résumé)

- Méthode **constructeur** (initialisation des attributs):

```
NomClasse(liste args): attrib1(...), , attrib1()  
{  
    ...           // autres opérations  
}
```

optionnel

- Méthode **constructeur de copie** :

```
NomClasse(const NomClasse & obj) :    ...  
{ ... }
```

Des versions par défaut (minimales) de ces méthodes sont générées automatiquement par C++ si on ne les fournit pas.

Règle: si on en définit une explicitement, il faut penser définir toutes celles utiles

2.10 Destruction (résumé)

- Méthode **destructeur** (ne peut être surchargée):

```
~NomClasse()  
{  
    // opérations (de libération)  
}
```

Si on ne la fournit pas, une version par défaut (minimale) de ces méthodes est générée automatiquement par C++

Règle

Si on en définit une explicitement, il faut penser à définir toutes celles utiles.

C++11

- *Delegating constructors* : un constructeur peut appeler un autre constructeur avec un différent nombre de paramètres

```
Class Foo {  
public:  
    Foo() = default; // voir prochain slide  
    Foo(char x, int y) {}  
    Foo(int y) : Foo('a', y) {}  
};
```

- Initialisation d'attributs à la déclaration

```
Class Foo {  
private:  
    int a = 3;  
    int b(4);  
};
```




C++11

- "Defaulted" default constructor

```
ClassName() = default;
```

Un constructeur par défaut est créé par le compilateur même si un constructeur avec paramètres existe

- "Deleted" default constructor

```
ClassName() = delete;
```

Oblige le compilateur à ne pas créer un constructeur par défaut

Note: Le même principe s'applique au constructeur par *recopie* et au *destructeur*

C++11

- Exemple :

```
class A{  
public:  
    A() = default; // Inline explicitly defaulted  
                  // constructor definition  
  
    A(const A&);  
  
    ~A() = default; // Inline explicitly defaulted  
                  // destructor definition;  
};
```