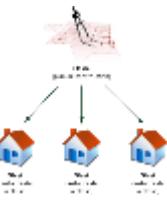


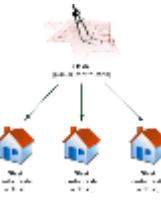
Chapitre 7

Classes et objets

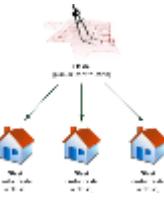


Plan du chapitre 7

1. Programmation orientée objet et encapsulation [3-12]
2. Classes [13-24]
3. Constructeurs [25-38]
4. Compilation séparée [39-44]
5. Surcharge des opérateurs [45-75]
6. Membres constants et membres statiques [76-81]
7. Membres particuliers [82-89]
8. Conclusion [90-92]

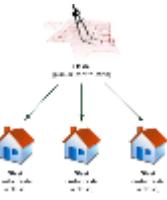


1. Programmation orientée objet et « encapsulation »



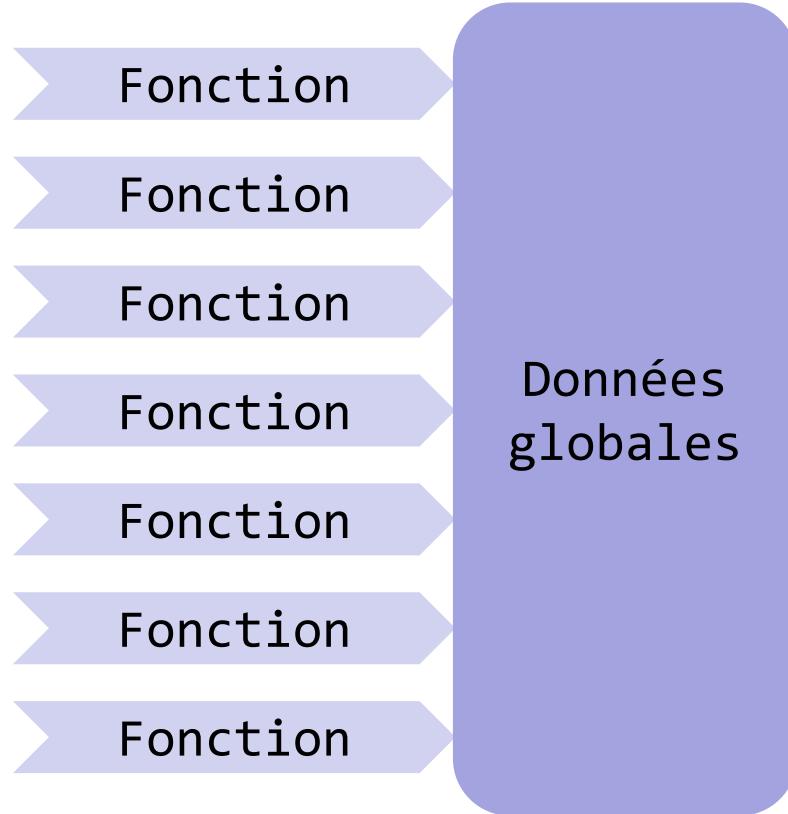
Pourquoi la POO ?

- Au tout début de PRG1, vous écriviez tout votre code dans la fonction `main()`
- Quand vos programmes sont devenus trop grands pour cela, vous avez appris à les organiser en les divisant en fonctions qui résolvent des sous-problèmes
- S'ils grandissent encore, il devient difficile de maintenir une collection de fonctions de plus en plus grande
- Il devient tentant, voire nécessaire, d'utiliser des variables globales... 🤷



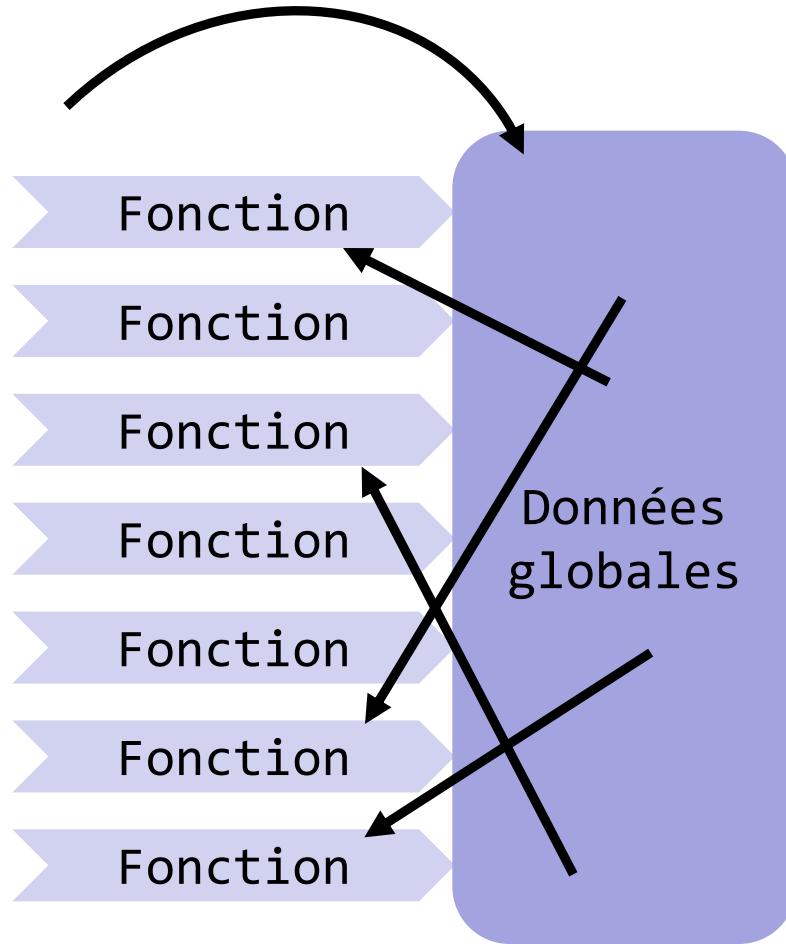
Pourquoi la POO ?

- Les variables globales sont celles qui sont définies hors de toute fonction.
- Cela permet à toutes les fonctions d'y accéder
- Mais...

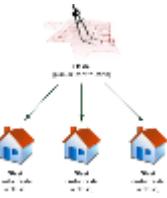


Pourquoi la POO ?

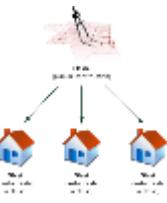
- Quand il est nécessaire de **modifier** une partie des variables globales
- Un grand nombre de fonctions sont potentiellement **affectées**
- Vous devrez toutes les **réécrire**
- Et **espérer** que tout fonctionne toujours



Pourquoi la POO ?

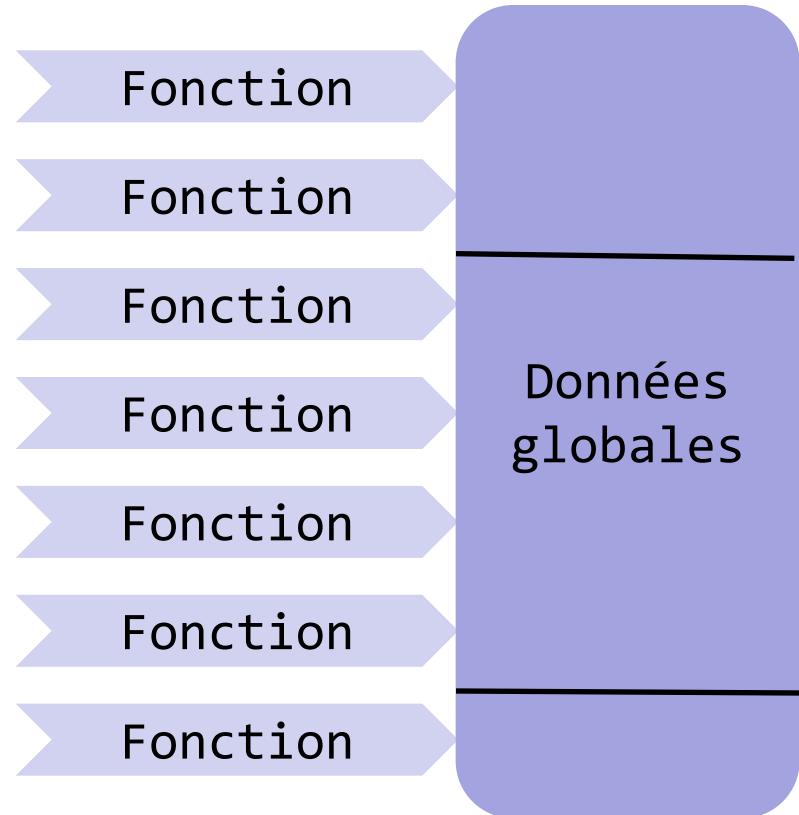


- En général, on remarque que l'on peut **regrouper** certaines **données** avec certaines **fonctions**
- Un **objet** est la conjonction de
 - **Données membres**
 - **Fonctions membres** qui traitent ces données membres



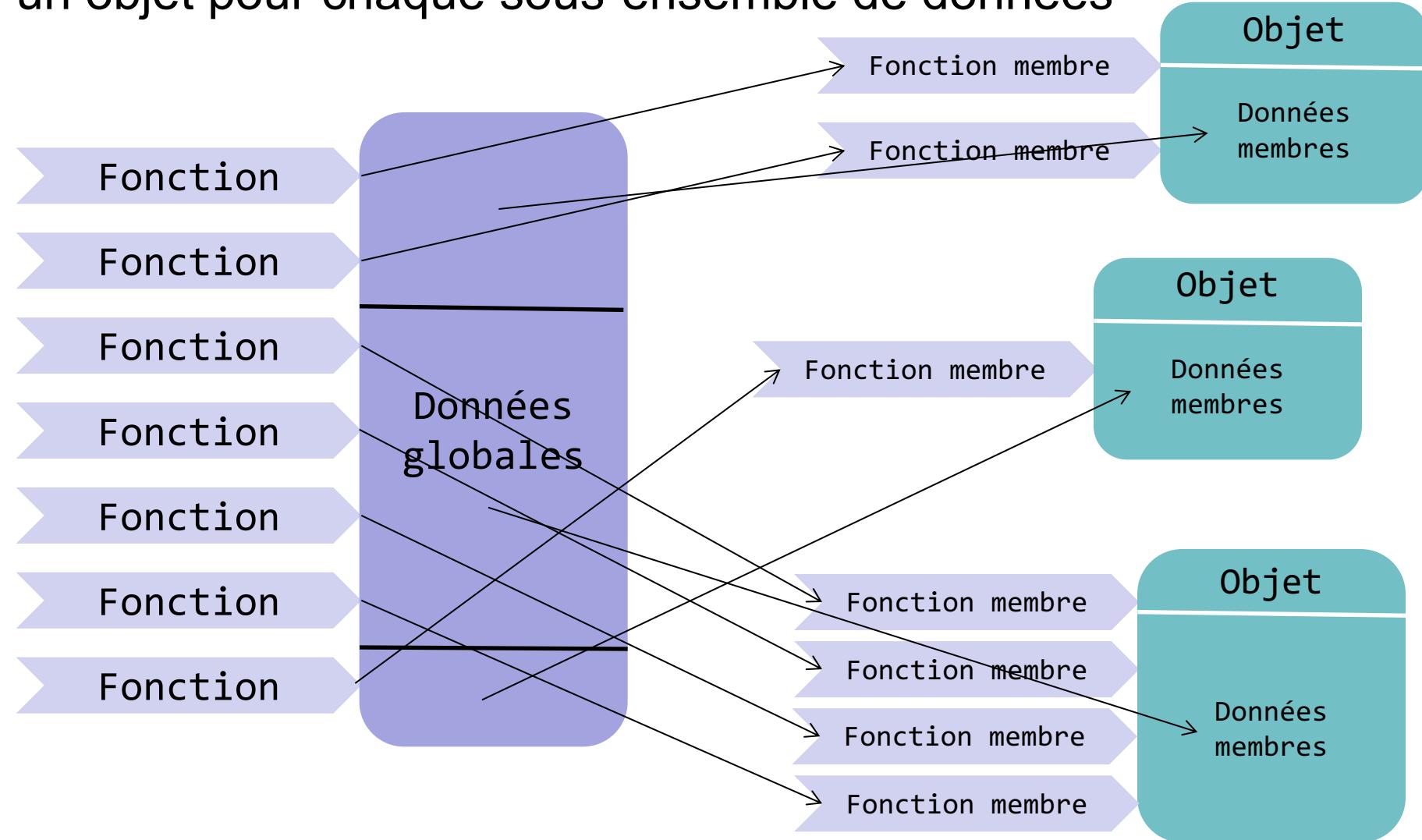
Programmation fonctionnelle -> POO

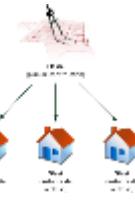
Déterminer quelles données vont avec quelles fonctions



Programmation fonctionnelle -> POO

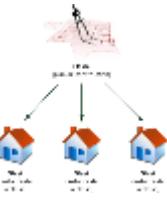
Créer un objet pour chaque sous-ensemble de données





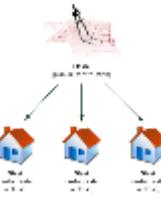
Encapsulation et classes

- **L'encapsulation** consiste à ne pas permettre d'accéder directement aux données, mais uniquement d'interagir avec l'objet via les fonctions membres
- Il devient possible de **changer la mise en œuvre** d'un objet **sans en changer l'interface**. Toutes les modifications seront locales à l'objet
- **Maintenir** et faire **évoluer** le programme est plus simple
- En C++, on définit des **types**, qui permettront de créer un ou plusieurs objets de chaque type
- Ce type s'appelle une **classe**



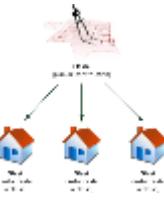
Éléments de POO déjà vus

- Vous avez déjà pratiqué la programmation orientée objet
- Les variables de type `string`, `vector`, `array`, ainsi que les flux `cin`, `cout`, `cerr`, ... sont tous des objets
 - vous les avez initialisés « par constructeur », avec divers paramètres, par exemple : `vector<int> v(3, 7);`
 - vous avez appelé leurs méthodes avec la notation `objet.methode(...)`, par exemple : `v.size();`
- Le but de ce chapitre est de vous apprendre à créer vos propres objets, donc à définir des classes, à savoir des types composés définis par l'utilisateur

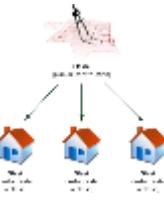


Autres concepts de POO après PRG1

- La programmation orientée objet est un paradigme bien plus riche que la simple encapsulation. Elle inclut :
 - la composition
 - l'héritage
 - la délégation
 - le polymorphisme
- Ces concepts - et la syntaxe C++ nécessaire à leur mise en œuvre - seront étudiés aux cours de POO1 et POO2

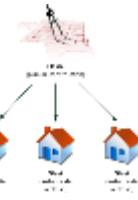


2. Classes



Que contient une classe en C++ ?

- Une **classe** est
 - un type complexe défini par le programmeur
 - qui permet de créer des **instances** = des **objets**
 - autant que nécessaire au fil de l'exécution du programme
- La définition d'une classe indique
 - les **données membres** ≈ des « **variables** »
 - en général, prendront des valeurs différentes selon l'objet
 - des **fonctions membres** = des **méthodes**
 - ont la même définition pour tous les objets de la classe



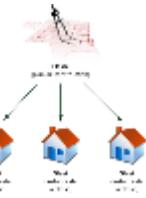
Déclaration

- Pour déclarer une classe, il suffit d'écrire

```
class NomDeLaClasse {  
public:  
    // déclaration de l'interface publique de  
    // la classe. Uniquement des fonctions  
    // membres pour une bonne encapsulation  
private:  
    // déclaration des données membres et  
    // éventuellement de fonctions privées.  
    // Non accessibles depuis l'extérieur de la classe  
};
```

Attention
au point-
virgule
final !

- Plusieurs sections **public:** et **private:** peuvent être présentes
- Par défaut (avant le premier **public:**), les membres sont privés

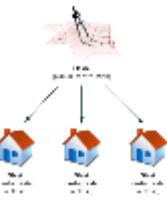


Exemple – Rectangle

- Par exemple, déclarons une classe Rectangle

```
class Rectangle {  
public:  
    // spécifie les dimensions du rectangle  
    void setDims(double la, double lo);  
    // calcule et renvoie la surface du rectangle  
    double surface() const;  
private:  
    // stocke les dimensions  
    double largeur;  
    double longueur;  
};
```

- ❖ On commence en général par les membres publics
- ❖ On évite de rendre publiques les données membres



Exemple – Rectangle

- Les déclarations suivantes sont équivalentes

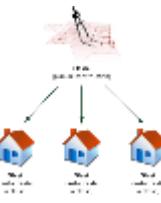
```
class Rectangle {  
public:  
    void setDims(double la, double lo);  
    double surface() const;  
private:  
    double largeur;  
    double longueur;  
};
```

```
class Rectangle {  
public: void setDims(double la, doubl );  
private: double largeur;  
public: double surface() const;  
private: double longueur;  
};
```

Déconseillé!

```
class Rectangle {  
double largeur;  
double longueur;  
public:  
    void setDims(double la, double lo);  
    double surface() const;  
};
```

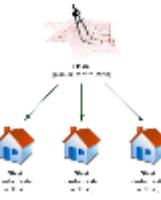
```
class Rectangle {  
private:  
    double largeur;  
    double longueur;  
public:  
    void setDims(double la, double lo);  
    double surface() const;  
};
```



Client = code utilisant la classe

- Une fois la classe `Rectangle` définie (avec `class`), on peut utiliser son nom comme n'importe quel nom de type
- Une variable (objet) de type `Rectangle` s'utilise comme n'importe quelle autre variable du C++
 - peut-être variable ou constante
 - allocation statique, automatique ou dynamique
 - passage par valeur, référence ou référence constante
- On accède aux membres (données ou fonctions) avec la `notation pointée`

```
Rectangle r;  
r.setDims(2.0, 3.0);  
cout << r.surface();
```



Accès aux membres privés

- Essayer d'accéder aux membres privés depuis le code extérieur à la classe donne une erreur à la compilation

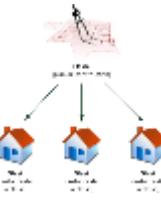
```
Rectangle r;  
cout << r.largeur; → 'largeur' is a private member of 'Rectangle'
```

- S'il est nécessaire d'y accéder, il faut définir des **accesseurs**, i.e. des fonctions membres permettant de

- lire la donnée privée
(un « **sélecteur** » ou « *getter* »)
- modifier la donnée privée
(un « **modificateur** » ou « *setter* »)

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

```
void setLargeur(double val) {  
    largeur = val;  
}
```



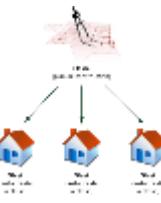
Accès aux membres privés : solution

Depuis une fonction membre, on a accès à toutes les données et fonctions membres, y compris privées :

- soit directement via leur nom
- soit en utilisant le mot-clé **this** qui est un **pointeur** vers l'objet, ou ***this** : l'objet lui-même (voir transparents suivants sur this)
- ❖ **this** (parfois appelé **autoréférence**) ne peut être utilisé qu'au sein d'une fonction membre et représente un pointeur sur l'objet ayant appelé la fonction.

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

```
double Rectangle::surface() const {  
    return this->largeur * (*this).longueur;  
}
```

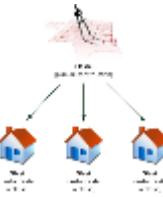


this

- Une variable **pointeur** (ou simplement un *pointeur*) est une variable contenant l'**adresse mémoire** d'une autre variable, appelée **variable pointée**.
- Pour pouvoir accéder (en lecture ou en écriture) à un champ donné d'un objet pointé par un pointeur, il faut réaliser deux opérations :
 1. Déréférencer le pointeur (c'est-à-dire accéder à l'objet pointé) en utilisant l'opérateur de déréférencement (appelé aussi opérateur d'indirection) : *****
 2. Utiliser la notation pointée (opérateur **.**) pour accéder au champ voulu.

```
double Rectangle::surface() const {
    return (*this).largeur * (*this).longueur; // Notez les () autour de *this
                                                // nécessaires en vertu de la
                                                // priorité des opérateurs * .
}
```

this

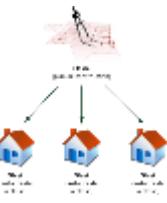


- Ecrire `(*this).champ` s'avère peu pratique.
- L'**opérateur ->** permet de simplifier les choses. En effet :

`this->champ` est équivalent à `(*this).champ`

- Remarques :
 1. N'abusez pas du mot-clé `this`.
Sur le slide 20, par exemple, il suffit d'écrire :
`return largeur * longueur;`
Il n'est pas nécessaire d'écrire :
`return this->largeur * this->longueur;`
 2. Utiliser `this` permet d'accéder aux membres même si une variable locale ou un paramètre porte le même nom (voir exemple ci-contre)

```
class Rectangle {  
public:  
    void setDims(double largeur, double longueur) {  
        this->largeur = largeur;  
        this->longueur = longueur;  
    }  
private:  
    double largeur;  
    double longueur;  
};
```



Définition des fonctions membres

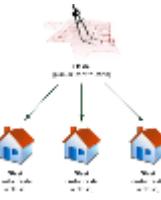
- Les fonctions membres sont définies

- soit en ligne dans la déclaration

```
class Rectangle {  
public:  
    double surface() const {  
        return largeur * longueur;  
    }  
    ...  
};
```

- soit séparément (cas le plus fréquent), avec l'opérateur de résolution de portée ::

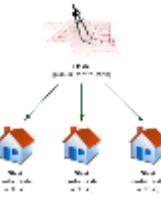
```
double Rectangle::surface() const {  
    return largeur * longueur;  
}
```



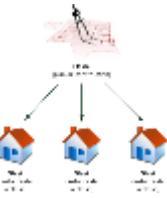
Fonctions membres const

- Notons la présence du mot-clé **const** à la fin de la déclaration de la fonction membre `surface()`
- Elle indique qu'aucune donnée membre (non static) de `Rectangle` n'est modifiée par cette fonction
- La fonction `setDims()`, qui n'est pas déclarée **const**, ne peut être appelée que pour une variable de type `Rectangle`
- La fonction `surface()`, déclarée **const**, peut aussi être appelée pour une constante (ou une référence constante) de type `Rectangle`

```
class Rectangle {  
public:  
    void setDims(double la, double lo);  
    double surface() const;  
private:  
    double largeur;  
    double longueur;  
};
```



3. Constructeurs

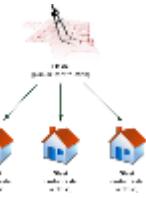


Motivation

- L'exemple suivant n'est pas satisfaisant

```
Rectangle r;  
r.setDims(2.0, 3.0);  
cout << r.surface();
```

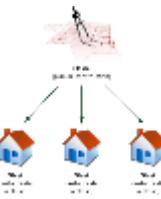
- La première ligne crée l'objet mais les valeurs des données membres sont indéterminées
 - si on appelle à la 2^e ligne `r.surface()` alors on obtient un résultat indéterminé
 - il faudrait pouvoir initialiser l'objet lors de sa création
- C'est possible ! Il suffit de définir une ou plusieurs fonctions membres particulières, appelées **constructeurs**



Constructeur

- Un constructeur est une **fonction membre** particulière qui
 - a le **même nom** que la classe
 - ne **retourne pas** de valeur, pas même void
 - ne comporte pas d'instruction **return**
- On améliore la classe Rectangle en lui ajoutant un constructeur qui initialise les deux dimensions (données membres)

```
Rectangle::Rectangle(double la, double lo) {  
    largeur = la;  
    longueur = lo;  
}
```



Constructeur

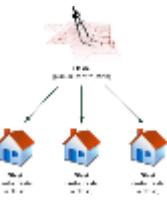
- Le code client se réécrit alors plus proprement

```
Rectangle r(2.0, 3.0);
cout << r.surface();
```

- On peut créer autant d'instances du même type qu'on souhaite (leurs données sont indépendantes)

```
Rectangle r1(2.0, 3.0);
Rectangle r2(5.0, 7.0);
cout << r1.surface() << ' ';
cout << r2.surface();
```

Affiche : 6 35



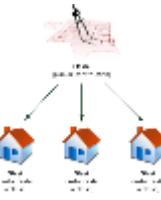
Surcharge de constructeurs

- Comme pour toutes les fonctions, on peut surcharger les constructeurs.
Par exemple :

```
Rectangle::Rectangle(double la, double lo) {  
    largeur = la;  
    longueur = lo;  
}
```

```
Rectangle::Rectangle(double cote) {  
    largeur = longueur = cote;  
}
```

```
Rectangle::Rectangle() {  
    largeur = longueur = 0.;  
}
```

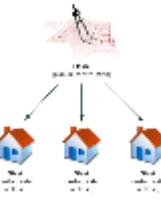


Surcharge de constructeurs

- On peut alors écrire le code client suivant :

```
Rectangle r1(2.0, 3.0);
Rectangle r2(5.0);
Rectangle r3;
cout << r1.surface() << ' ';
cout << r2.surface() << ' ';
cout << r3.surface();
```

Affiche : 6 25 0



Constructeur par défaut

- Rectangle() est appelé **constructeur par défaut** ou **constructeur sans arguments**

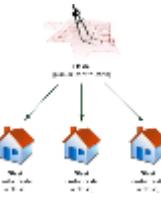
```
Rectangle::Rectangle() { largeur = longueur = 0.; }
```

- Si aucun constructeur n'est déclaré explicitement, le compilateur ajoute un constructeur par défaut, vide
- Le client peut l'appeler via

```
Rectangle r1; // sans parenthèses  
Rectangle r2 {};// initialisation uniforme (C++11)
```

mais attention, pas via

```
Rectangle r3(); // déclare une fonction r3 retournant un Rectangle
```



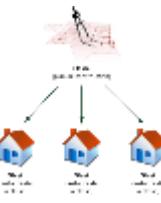
Ajout par le compilateur ou pas ?

- Si la classe déclare explicitement un autre constructeur, alors le constructeur vide par défaut n'est plus ajouté par le compilateur.
- Pour la classe C exemplifiée ci-contre, il n'est pas possible de créer un objet c2 sans paramètre comme ci-dessous

```
class C {  
    int data;  
public:  
    C(int val) {data = val;}  
};
```

```
C c1(100);  
C c2; // No matching constructor for initialization of 'C'
```

- ❖ Il faudrait écrire `C(){ }` dans la zone publique de la déclaration de C pour définir le constructeur vide, ou écrire plus explicitement : `C() = default;`
- ❖ Inversement, on peut aussi l'interdire en écrivant `C() = delete;`



Appels des constructeurs

- Pour appeler le constructeur à un argument

```
Rectangle::Rectangle(double cote) {  
    largeur = longueur = cote;  
}
```

- Il y a 4 syntaxes possibles

```
Rectangle r1(1.0);      // fonctionnelle  
Rectangle r2 = 2.0;     // affectation  
Rectangle r3{3.0};      // init. uniforme (C++11)  
Rectangle r4 = {4.0};    // init. uniforme (C++11)
```

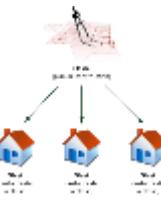
- Constructeurs à deux arguments ou plus

- possible : syntaxe fonctionnelle ou uniforme | impossible : par affectation

Parfois, il peut s'avérer nécessaire d'interdire la construction d'un objet par affectation (avec Rectangle r2 = 2.0, par exemple)

Ceci peut se faire en munissant le constructeur du mot-clé **explicit** :

```
explicit Rectangle(double cote);
```



Initialisation des membres

Quand un constructeur sert à initialiser des données membres de l'objet, cela peut être fait sans recourir à des affectations, mais avec une **liste d'initialisations de membres**, p.ex.

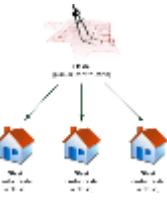
```
Rectangle::Rectangle(double la, double lo) {  
    largeur = la; longueur = lo;  
}
```

peut s'écrire aussi comme ceci avec une liste

```
Rectangle::Rectangle(double la, double lo) : largeur(la), longueur(lo) {}
```

et cela est possible aussi avec une partie des arguments seulement

```
Rectangle::Rectangle(double la, double lo) : largeur(la) {  
    longueur = lo;  
}
```



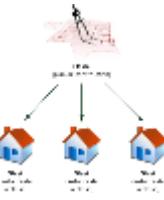
Initialisation des membres

- Il peut paraître plus simple d'initialiser les membres avec une valeur par défaut ainsi :

```
class Rectangle {  
public:  
    void setDims(double la, double lo);  
    double surface() const;  
private:  
    double largeur = 0;  
    double longueur = 0;  
};
```

- Mais...**

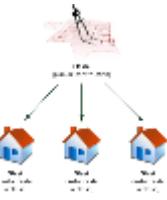
- Ce n'est possible que depuis C++11
- L'initialisation par un constructeur est plus flexible
- Elle est prioritaire si les deux sont présentes



Initialisation des membres : liste ou pas ?

- Pour les **membres variables de type simple**, cela ne change rien
- Pour les **membres qui sont eux-mêmes des objets** (variables de type composé) de type classe C, cela change la méthode de leur propre création
 - **Affectation**
 - **appel du constructeur par défaut de C, puis de l'opérateur d'affectation¹ de C**
 - si la classe C n'a pas de constructeur par défaut, alors erreur !
 - **Initialisation (liste)**
 - **appel du constructeur de (re)copie¹ de C** qui reçoit la valeur initiale en paramètre.
- Pour les **données membres constantes**, *seule l'utilisation de la liste d'initialisation est possible*, l'opérateur d'affectation n'existant pas.

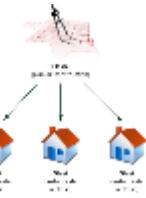
¹ L'opérateur d'affectation et le constructeur de (re)copie seront étudiés en détail plus loin dans ce chapitre



Appel des constructeurs : exemple 1

- Supposons qu'une classe A inclut parmi ses données membres un objet d'une classe B
- Lorsqu'on appelle un constructeur de la classe A, est-ce que cela appelle aussi un constructeur de B, et quand ?
- Réponse : oui !
Il est appelé avant l'exécution du bloc {} du constructeur de A, en d'autres mots, à l'endroit où doit se trouver la liste d'initialisation
- Si la liste manque ou est incomplète, on appelle les constructeurs par défaut

```
class B {  
    int n;  
};  
  
class A {  
    B b;  
public:  
    A() {};  
};  
  
int main() {  
    A a;  
    return 0;  
}
```

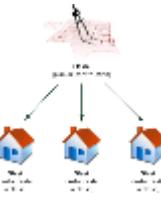


Appel des constructeurs : exemple 2

- Grâce au concept de liste d'initialisation, un constructeur (dans l'exemple ci-dessous le constructeur sans paramètre) peut facilement appeler un autre constructeur de la même classe

```
class Rectangle {  
public:  
    Rectangle() : Rectangle(0, 0) {}  
  
    Rectangle(double la, double lo) : largeur(la), longueur(lo) {}  
    ...  
private:  
    ...  
};
```

- Cela permet de **factoriser** :
 - le code d'initialisation des diverses données membres
 - (s'il existe) le code vérifiant la validité des paramètres passés au constructeur



4. Compilation séparée

Organisation du code par classe

- Typiquement, chaque classe a ses propres fichiers header (.h) et .cpp, souvent nommés du nom de la classe
- La **déclaration**, dans le fichier **header**, inclut
 - les **déclarations** des données et des fonctions membres et amies
 - la **définition** de certaines fonctions en ligne (les opérateurs sont des fonctions comme les autres)
- Le fichier **.cpp** inclut la **définition** des autres fonctions membres, ainsi que l'initialisation des variables et constantes statiques

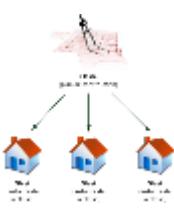
maClasse.h

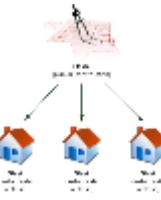
```
#ifndef MACLASSE_H
#define MACLASSE_H

class MaClasse {
    friend void fonctionAmie(MaClasse& c);
    friend void fonctionAmieEnLigne(MaClasse& c) { /*...*/ }
public:
    // constructeurs
    MaClasse() : constante(0) { /* ... */ }
    MaClasse(int n, double x); // defini dans maClasse.cpp
    // constructeur de copie
    MaClasse(const MaClasse& obj)
        : variable(obj.variable), constante(obj.constante) { /* ... */ }
    // destructeur
    ~MaClasse() { /* ... */ }

    ...
}
```

Nombre d'éléments apparaissant dans la déclaration de MaClasse vous sont, pour l'heure, inconnus. Rassurez-vous, ils seront tous étudiés plus loin dans ce chapitre !



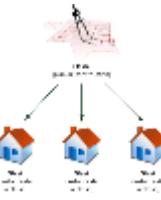


maClasse.h (suite)

```
...
void fonctionMembre(int n);
void fonctionMembreEnLigne() { /*...*/ }
static void fonctionStatique(int n);
static void fonctionStatiqueEnLigne() { /*...*/ }
private:
    int variable;
    const double constante;
    static char variableStatique;
    static const short CONSTANTE_STATIQUE;
};

#endif /* MACLASSE_H */
```

À noter que seules les données membres `static const` s'écrivent entièrement en majuscules.



maClasse.cpp

```
#include "maClasse.h"

// membres statiques
char MaClasse::variableStatique = 'A';
const short MaClasse::CONSTANTE_STATIQUE = 0;

void fonctionAmie(MaClasse& c) { /* ... */ }

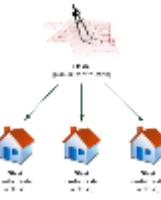
// un des constructeurs
MaClasse::MaClasse(int n, double x) : variable(n), constante(x) { /* ... */ }

void MaClasse::fonctionMembre(int n) { /* ... */ }

void MaClasse::fonctionStatique(int n) { /* ... */ }
```

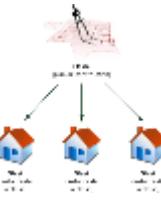
A noter que dans `maClasse.cpp` :

- 1) on ne répète pas les mots `static` et `friend`
- 2) la fonction amie n'est pas préfixée par `maClasse::`

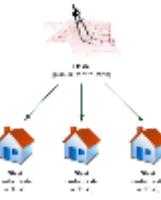


Où placer les #include ?

- **Le fichier header** (`maClasse.h`)
 - inclut les autres headers dont il a besoin pour ses déclarations et pour les définitions *inline* des fonctions membres ou amies
- **Le fichier source cpp** (`maClasse.cpp`)
 - inclut le header "`maClasse.h`"
 - inclut les autres headers (hormis ceux déjà inclus dans le header "`maClasse.h`") dont il a besoin pour ses définitions des fonctions membres ou amies
- **Le fichier client** (qui utilisera la classe)
 - inclut le header "`maClasse.h`" et tout autre header nécessaire à son implémentation (sans prendre pour acquise l'inclusion des headers déjà inclus dans le header "`maClasse.h`")
- **Le test** `#ifndef MACLASSE_H #define MACLASSE_H ... #endif` assurera qu'on n'inclut qu'une fois chaque header, même s'il est utilisé à beaucoup d'endroits (le nom est unique)
- **Et les « using namespace » ?**
Il vaut mieux ne pas les mettre dans les headers, car ils s'appliqueront partout ailleurs.



5. Surcharge des opérateurs

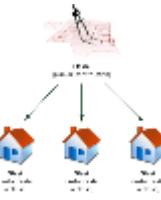


Nécessité de la surcharge d'opérateurs

- Comment réaliser une opération entre deux objets ?

```
bool Robot::plusADroiteQue(const Robot& robot) const {  
    return this->position > robot.position;  
}
```

- Si robot1 est en 5 et robot2 est en 3, alors :
 - robot1.plusADroiteQue(robot2) vaut **true**
 - robot2.plusADroiteQue(robot1) vaut **false**
- Mais peut-on écrire robot1 > robot2 ?
Non, car l'opérateur > est défini seulement pour des nombres,
et pas pour des objets de la classe Robot

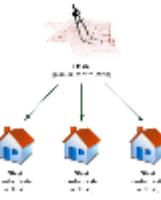


Opérateurs qu'on peut surcharger

- Les classes définissent de nouveaux types C++
- Pour les types simples, nous utilisons souvent des opérateurs tels que =, +, -, *, ++, <, ==, <<, ...
- Pourrait-on les appliquer aussi à nos classes ?
Oui, on peut **surcharger** pour nos classes les opérateurs suivants :

```
+ - * / = < > += -= *= /= << >>
<<= >>= == != <= >= ++ -- % & ^ !
~ &= ^= |= && || %= [] () , ->* -> new
delete new[] delete[]
```

- Note 1 : on ne peut pas changer leur priorité ni le nombre d'arguments, et on ne peut pas définir de nouveaux opérateurs
- Note 2 : l'opérateur d'affectation (=) est défini par défaut pour les classes

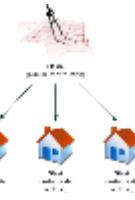


Syntaxe de la surcharge

- Pour surcharger un opérateur, il faut définir une fonction membre `operator@` où `@` est le symbole de l'opérateur que l'on veut définir.
La syntaxe générale est :

```
typeRetour operator @ (paramètres) { /*... corps ...*/ }
```

- Les espaces autour du symbole sont facultatifs
- Les paramètres et le type de retour dépendent de l'opérateur que l'on veut surcharger
- On doit considérer quels paramètres sont modifiés ou pas (mot clé `const`) et penser au passage par référence`&` (si la copie de l'objet passé en paramètre est coûteuse)

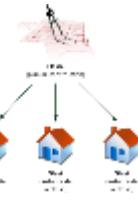


Exemple

- Classe CVector qui représente des vecteurs du plan

```
class CVector {  
    double x, y; // données membres  
public:  
    CVector() {} // constructeur par défaut  
    CVector(double a, double b) : x(a), y(b) {} // constructeur d'initialisation  
};
```

- Note : on fournit ici deux constructeurs
 - 1. celui par défaut (zéro arguments)
 - 2. celui qui initialise x et y (deux arguments)
grâce ici à une liste d'initialisation

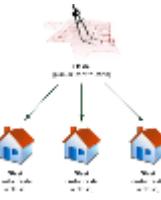


operator+

- Pour utiliser un opérateur + permettant de sommer deux CVector, on **déclare** d'abord la fonction membre **operator+**

```
class CVector {  
    double x, y;  
public:  
    CVector() {};  
    CVector(double a, double b) : x(a), y(b) {}  
    CVector operator+(const CVector& v) const;  
};
```

- Elle reçoit en paramètre une référence constante à un autre CVector (la référence évite la copie de l'objet)
- Elle retourne un CVector qui est la somme (*à définir*) de l'objet courant et du CVector reçu en paramètre



operator+

- Il faut évidemment aussi **définir** cette fonction membre

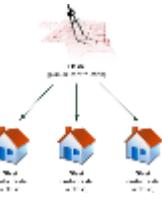
```
CVector CVector::operator+(const CVector& v) const {  
    CVector temp;  
    temp.x = this->x + v.x; // this est facultatif  
    temp.y = this->y + v.y; // dans les 2 lignes  
    return temp;  
}
```

À noter que le code ci-dessus peut s'écrire en une ligne :

`return CVector(x + v.x, y + v.y);` ou encore `return CVector{x + v.x, y + v.y};`

- Comme pour toute fonction membre, on a accès aux membres privés de `v` (argument de la fonction), même s'il s'agit d'un autre objet que `*this`

En C++, la **notion de privé/public** s'applique au niveau de la **classe** (le type), pas au niveau des **objets** (les variables)



Utilisation de CVector::operator+

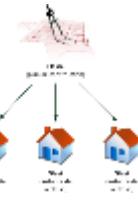
- Un client peut maintenant utiliser l'opérateur + pour le type CVector

```
int main() {
    CVector v1(3, 1);
    CVector v2(1, 2);
    CVector v3 = v1 + v2; // v3.x vaut 4, v3.y vaut 3
}
```

- Note : on pourrait aussi écrire (avec ou sans espace)

```
CVector v3 = v1.operator+(v2);
```

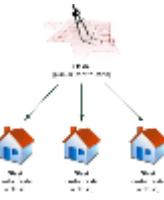
... mais ce n'est pas intuitif et personne ne le fait !



operator*

Rien n'oblige à ce que les deux opérandes d'un opérateur soient du même type. Exemple : multiplication d'un CVector par un réel

```
class CVector {  
    double x, y;  
public:  
    ... // idem exemple précédent  
    CVector operator*(double d) const;  
};  
  
CVector CVector::operator*(double d) const {  
    CVector temp;  
    temp.x = x * d;  
    temp.y = y * d;  
    return temp;  
}
```



operator*

- On peut alors écrire le client

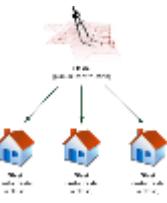
```
int main() {
    CVector v1(3, 1);
    CVector v2 = v1 * 2; // v2.x vaut 6, v2.y vaut 2
}
```

- Par contre, si on change l'ordre, le code ne compile pas

```
CVector v2 = 2 * v1;
```

Invalid operands to binary
expression ('int' and 'CVector')

- En effet, il aurait fallu surcharger **operator*** pour le type **int**... ce qui est impossible, **int** étant un type simple en C++, pas une classe.
Y a-t-il une solution ?



Surcharge par fonction non membre

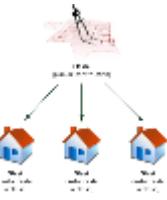
- Il est aussi possible de surcharger des opérateurs par une fonction « simple », et non pas une fonction membre
- Par exemple, la fonction membre

```
CVector CVector::operator*(double d) const { ... }
```

peut être remplacée par la fonction

```
CVector operator*(const CVector& lhs, double rhs) {  
    CVector temp;  
    temp.x = lhs.x * rhs; // lhs signifie left hand side  
    temp.y = lhs.y * rhs;  
    return temp;  
}
```

- Le premier paramètre de la fonction simple correspond au paramètre implicite `*this` pour les fonctions membres



Surcharge par fonction non membre

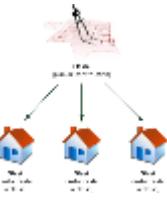
- Il suffit alors d'écrire les deux fonctions

```
CVector operator*(const CVector& lhs, double rhs);  
CVector operator*(double lhs, const CVector& rhs);
```

pour pouvoir effectuer la multiplication entre un `CVector` et un `double` dans les deux ordres possibles

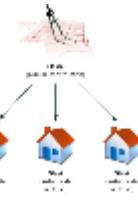
- Mais les fonctions externes n'ont pas accès aux données privées, d'où l'erreur de compilation suivante qui demeure :

```
CVector operator*(const CVector& lhs, double rhs) {  
    CVector temp;  
    temp.x = lhs.x * rhs;  'x' is a private member of 'CVector'  
    temp.y = lhs.y * rhs;  'y' is a private member of 'CVector'  
    return temp;  
}
```



Solution : les fonctions amies (friend)

- La fonction **operator*** a besoin d'accéder aux membres privés des objets de type **CVector**
- Mais on ne peut rendre publics ces membres sans violer le principe d'encapsulation
- Une solution serait de donner accès aux membres x et y via des **accesseurs** (sélecteurs et modificateurs) mais ce n'est pas satisfaisant si x et y sont des détails de mise en œuvre qui ne doivent pas être **exposés à tous les clients**
- La solution qui **respecte l'encapsulation** consiste à déclarer dans la classe **CVector** que la fonction **operator*** est une **amie**, ce qui lui donne accès aux membres privés

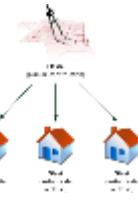


friend

```
class CVector {  
    friend CVector operator*(double lhs, const CVector& rhs);  
public:  
    ... // idem exemple précédent  
private:  
    double x, y;  
};  
  
CVector operator*(double lhs, const CVector& rhs) {  
    CVector temp;  
    temp.x = lhs * rhs.x;  
    temp.y = lhs * rhs.y;  
    return temp;  
}
```

Cela nous permet d'écrire le code client suivant :

```
CVector v1(3, 1);  
CVector v2 = 2 * v1;
```



friend

Une fonction amie peut aussi être définie en ligne, dans le code de la classe.
Elle reste une fonction non membre de la classe.

```
class CVector {  
    friend CVector operator*(double lhs, const CVector& rhs) {  
        CVector temp;  
        temp.x = lhs * rhs.x;  
        temp.y = lhs * rhs.y;  
        return temp;  
    }  
public:  
    ... // idem exemple précédent  
private:  
    ... // idem exemple précédent  
};
```

friend

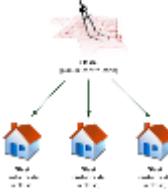
- L'amitié n'est pas réservée aux opérateurs
- On peut déclarer comme amie d'une classe
 - une **fonction**
 - une **fonction membre d'une autre classe** (voir exemple ci-contre)
 - une **autre classe** dans son ensemble
- Les fonctions ou classes peuvent être les amies de plusieurs classes

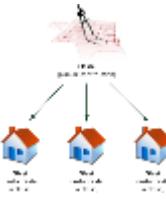
```
class A;

class B {
    int n;
public:
    B(int n) : n(n) {}
    void changer(const A& a);
};

class A {
    friend void B::changer(const A& a);
    int n;
public:
    A(int n) : n(n) {}
};

void B::changer(const A& a) {
    n = a.n; // a.n est un membre
              // privé de la classe A
}
```

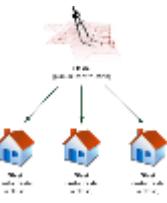




Commutativité

- L'exemple précédent a permis d'introduire les notions de surcharge d'opérateur par fonction amie
- Mais la solution obtenue n'est pas satisfaisante : il y a **duplication de code** entre les deux versions d'`operator*`
- Pour un **opérateur commutatif**, il est plus propre qu'une version de l'opérateur appelle l'autre

```
class CVector {  
    friend CVector operator*(double lhs, const CVector& rhs);  
public:  
    CVector operator*(double d) const;  
    ... // idem exemples précédents  
};  
  
CVector operator*(double lhs, const CVector& rhs) {  
    return rhs * lhs;  
}  
  
CVector CVector::operator*(double d) const {  
    CVector temp;  
    temp.x = x * d;  
    temp.y = y * d;  
    return temp;  
}
```

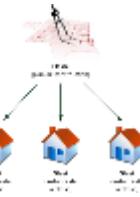


Affichage d'un objet : operator<<

- Une application importante du principe d'amitié est la surcharge des opérateurs de flux << et >>

```
class CVector {  
    friend ostream& operator<<(ostream& lhs, const CVector& rhs);  
    ... // idem exemples précédents  
};  
  
ostream& operator<<(ostream& lhs, const CVector& rhs) {  
    lhs << rhs.x << ',' << rhs.y;  
    return lhs;  
}
```

- Les deux paramètres sont
 - une référence au flux de sortie (cout, cerr) dans lequel écrire
 - une référence constante à l'objet à afficher
- On retourne la référence au même flux



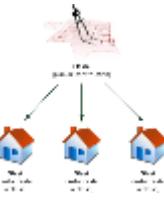
operator<<

- La surcharge de cet opérateur nous permet d'écrire le client suivant

```
int main() {
    CVector v1(3, 1);
    const CVector v2(1, 2);
    cout << v1 << endl
        << v2 << endl
        << v1 + v2 << endl;
}
```

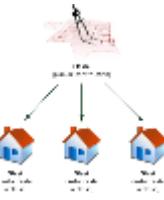
3,1
1,2
4,3

- Passer le deuxième paramètre en référence constante (`const CVector& rhs`) permet d'afficher tant la variable `v1` que la constante `v2` ou l'expression `v1 + V2`
- Retourner la référence au flux permet d'enchaîner les <<



Opérateurs d'affectation composée

- Quand on définit un opérateur binaire tel que `+`, `-`, `*`, ..., il est mieux de définir aussi l'opérateur d'affectation composée correspondant: `+=`, `-=`, `*=`, ...
- Pour s'assurer que les deux opérateurs sont cohérents, on implémente typiquement l'opérateur binaire en utilisant l'opérateur d'affectation composée
- **L'opérateur d'affectation est obligatoirement une fonction membre**; il ne peut pas être déclaré en tant que fonction amie : *c'est une contrainte de C++*
- L'opérateur d'affectation retourne typiquement une référence vers l'objet qu'il affecte, alors que l'opérateur binaire retourne typiquement l'objet par valeur

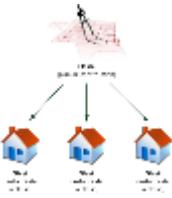


Opérateurs d'affectation composée

Les **formes canoniques** d'un opérateur arithmétique et de l'affectation composée correspondante sont donc :

```
class X {  
    // ...  
    friend X operator+(X lhs, const X& rhs) {  
        lhs += rhs; // appel à +=  
        return lhs;  
    }  
  
public:  
    X& operator+=(const X& rhs) {  
        // ici modifier les données en y ajoutant rhs  
        return *this;  
    }  
};
```

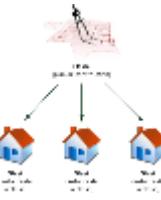
À noter le passage par valeur



Précisions sur la forme canonique

- Surcharge de `+` grâce à `+=`
 - pour ne pas dupliquer le code (surtout s'il est complexe)
 - ce serait faux (et inutile) de passer `lhs` par référence, car la somme doit être un nouvel objet
 - pourrait aussi être implémentée en tant que fonction membre constante, mais en tant que fonction amie permet de traiter `lhs` et `rhs` de manière symétrique
- Surcharge de `+=` comme fonction membre pour retourner une référence à l'objet (`X&`)
 - `(a += 2) += 3; // possible`
 - `X&` n'est pas `const` car il doit être modifié (affecté)
 - permet certaines optimisations ...

```
class X {  
    // ...  
    friend X operator+(X lhs, const X& rhs) {  
        lhs += rhs; // appel à +=  
        return lhs;  
    }  
  
public:  
    X& operator+=(const X& rhs) {  
        // ici modifier les données  
        // en y ajoutant rhs  
        return *this;  
    }  
}; // À UTILISER DONC COMME MODÈLE !
```



Opérateurs relationnels

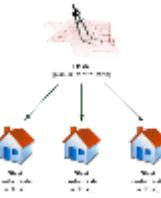
Pour les opérateurs de comparaison et d'égalité, on met typiquement en œuvre `operator<` et `operator==`, et on implémente les autres grâce à ces deux.

Pour une classe X :

```
friend bool operator<(const X& lhs, const X& rhs) /* comparaison < à écrire ici */
friend bool operator>(const X& lhs, const X& rhs) {return rhs < lhs;}
friend bool operator<=(const X& lhs, const X& rhs) {return !(rhs < lhs);}
friend bool operator>=(const X& lhs, const X& rhs) {return !(lhs < rhs);}

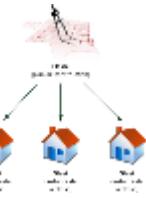
friend bool operator==(const X& lhs, const X& rhs) /* comparaison == à écrire ici */
friend bool operator!=(const X& lhs, const X& rhs) {return !(lhs == rhs);}
```

operator++



- Pour surcharger l'opérateur ++ (ou --), il faut écrire deux fonctions. L'une pour l'opérateur préfixe, l'autre pour le postfixe. Comment les distinguer ?
- **L'opérateur préfixe** ne prend pas de paramètre et retourne une référence vers l'objet lui-même
- **L'opérateur postfixe** prend formellement un paramètre entier (dont la valeur n'est pas utilisée) et retourne une copie de l'objet avant incrémentation

```
class A {  
    // ...  
public:  
    A& operator++() {    // préfixe  
        // incrémenter les données  
        return *this;  
    }  
  
    A operator++(int) { // postfixe  
        A temp = *this;  
        // incrémenter les données  
        return temp;  
    }  
};
```



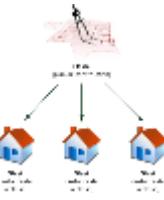
operator++

- On peut aussi écrire la version postfixe comme ceci :

```
class A {  
    // ...  
public:  
    A& operator++() {    // préfixe (p.ex. ++n)  
        // incrémenter les données  
        return *this;  
    }  
  
    A operator++(int) { // postfixe (p.ex. n++)  
        A temp = *this;  
        ++*this; // appel de la version préfixe  
        return temp;  
    }  
};
```

Fonction membre ou non pour la surcharge d'opérateurs ?

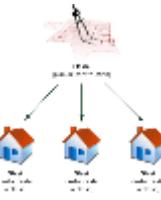
- Utiliser plutôt les **fonctions membres** : bonne encapsulation, intégration à la classe, accès à toutes les données des objets.
- Quand est-ce qu'on doit utiliser une **fonction non membre** ?
 - Notamment lorsque le premier argument de l'opérateur est un objet d'une classe qu'on ne peut modifier (on ne peut lui ajouter une fonction membre), par exemple `iostream`
- Comment procède le compilateur lorsqu'il rencontre `obj1 + obj2` ?
 1. Il cherche d'abord une **fonction membre** de la classe O1 de `obj1`, définie comme `operator+(O2)` où O2 est la classe de `obj2`, et s'il la trouve, il l'applique à `obj1`, comme `obj1.operator+(obj2)`.
 2. Sinon, il cherche ensuite une **fonction non membre** de la forme `operator+(O1, O2)`, et s'il la trouve, il l'applique comme `operator+(obj1, obj2)` – *sinon, erreur.*
- ❖ Si on surcharge par fonction non membre, on n'est pas obligé de la déclarer comme *friend* si elle n'a pas besoin des données membres privées des objets (ou peut utiliser les accesseurs publics)



Autres opérateurs

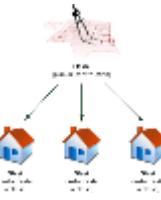
Pour un objet **a** de classe **A**, et un objet **b** de classe **B**, on peut surcharger l'opérateur **@ ...** (à remplacer par le symbole approprié) de la façon suivante :
(Note : A et B peuvent être des classes identiques ou distinctes)

| Expr. | Opérateurs @ | Membre | Non membre |
|-----------|--|-------------------------|------------------|
| @a | + - * & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - * / % ^ & , < > == != <= >= << >> && | A::operator@(B) | operator@(A,B) |
| a@b | = += -= *= /= %= ^= &= = <<= >>= [] | A::operator@(B) | Non disponible |
| a(b,c...) | () | A::operator@()(B,C,...) | |
| a->b | -> | A::operator->() | |
| (TYPE)a | TYPE | A::operator TYPE() | |



Classe foncteur

- Une classe surchargeant l'**opérateur ()** est appelée **classe foncteur**.
- Les instances d'une classe foncteur sont appelés **objets fonctions** (ou **foncteurs**) car ils peuvent être utilisés de la même manière qu'une fonction ordinaire.
- Les foncteurs s'avèrent très utiles (voire indispensables) lorsqu'il s'agit de transmettre une fonction en paramètre d'une autre fonction.
- Le slide 73 propose un exemple simple de mise en œuvre d'une classe foncteur ainsi qu'un exemple de surcharge de l'**opérateur de cast** (appelé TYPE dans le tableau du slide précédent).
- Les slides 74 et 75 illustrent l'utilité des foncteurs.

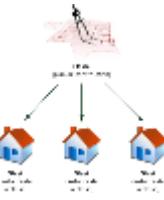


Exemple de classe foncteur

```
class Double { // classe foncteur
    friend ostream& operator<<(ostream& os, const Double& d) {return os << d.x;}
public:
    explicit Double(double x) : x(x) {} // constructeur
    explicit operator int() {return (int) x;} // opérateur de cast
    double operator()(int n) {return pow(x, n);} // opérateur (), ici pow(x, n)
private:
    double x;
};

int main() {
    Double d(1.5);
    cout << d << endl; // affiche 1.5
    int n = (int)d; // ou int(d)... mais pas d.int()
                     // cast Double -> int
    cout << n << endl; // affiche 1
    cout << d(2) << endl; // l'objet d peut s'utiliser comme une fonction ordinaire
                          // affiche 2.25 (= pow(1.5, 2))
}
```

Noter la forme particulière des surcharges des opérateurs () et de cast, ainsi que la présence du mot-clé explicit pour se prémunir (ici) contre les conversions implicites Double \leftrightarrow int



Utilité des foncteurs

- On veut compter le nombre d'entiers pairs et le nombre d'entiers multiples de 3 dans un vecteur avec `std::count_if` de la librairie `<algorithm>`
- Pour cela, il faut écrire 2 fonctions : `est_pair` et `est_multiple_de_3`

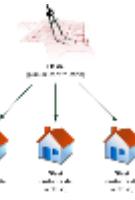
```
bool est_pair(int i) {return (i % 2) == 0;}
bool est_multiple_de_3(int i) {return (i % 3) == 0;}

int main() {
    vector<int> v{1, 4, 5, 2, 9, 5, 6, 8};
    cout << count_if(v.begin(), v.end(), est_pair) << endl;           // 4
    cout << count_if(v.begin(), v.end(), est_multiple_de_3) << endl; // 2
}
```

- Pour éviter de dupliquer du code, on voudrait écrire :

```
bool est_multiple_de_n(int i, int n) {return (i % n) == 0;}
```

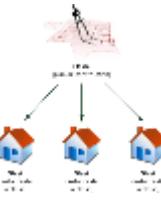
- Mais `est_multiple_de_n` a trop d'arguments pour être utilisée par `std::count_if`



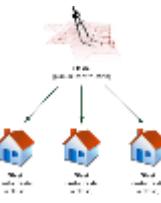
Utilité des foncteurs

- Par contre, avec un foncteur ...

```
class Est_multiple_de {  
    int n;  
public:  
    Est_multiple_de(int n) : n(n) {}  
    bool operator() (int i) {return (i % n) == 0;}  
};  
  
int main() {  
    vector<int> v{1, 4, 5, 2, 9, 5, 6, 8};  
    cout << count_if(v.begin(), v.end(), Est_multiple_de(2)) << endl; // 4  
    cout << count_if(v.begin(), v.end(), Est_multiple_de(3)) << endl; // 2  
}
```



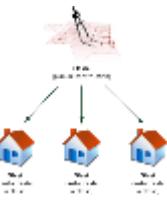
6. Membres constants et membres statiques



Membres constants

- Un **membre** d'une classe peut être déclaré **const**. Il ne peut donc **pas** subir **d'affectation** après son initialisation.
- La valeur initiale peut éventuellement **varier d'un objet à l'autre** d'une même classe. **L'initialisation** se fait donc lors de la construction de l'objet, via **la liste d'initialisation du constructeur** :

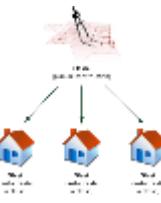
```
class V {  
public:  
    V(int c) : cste(c) {};  
private:  
    const int cste;  
};  
  
V v(3);
```



Membres constants

- Depuis C++11, on peut également fournir une valeur initiale à un membre constant dans sa déclaration
- C'est une valeur par défaut : elle est utilisée pour les objets créés avec des constructeurs qui n'initialisent pas explicitement la constante dans leurs listes d'initialisation

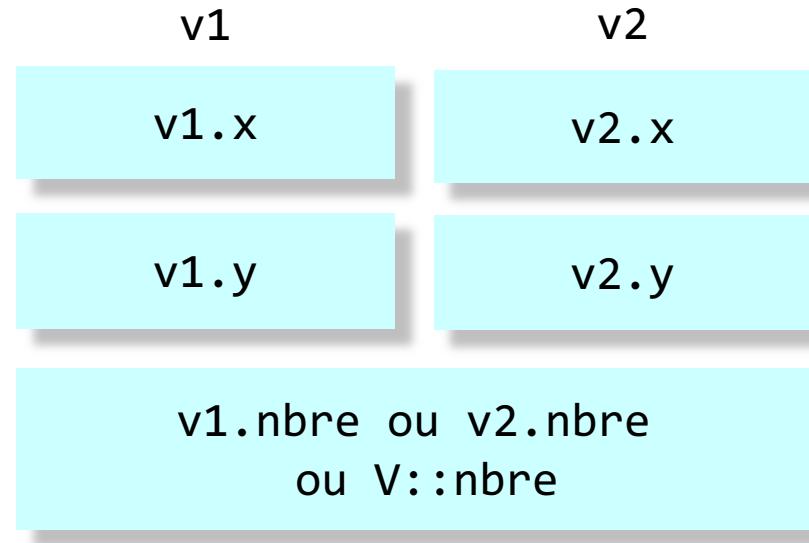
```
class V {  
public:  
    V(int n) : n(n) {};  
private:  
    int n;  
    const int cste = 1;  
};  
  
V v(3); // cste vaut 1
```



Membres statiques

- Un **membre** d'une classe peut être déclaré **static**.
Il est unique pour la classe et **commun à tous les objets de la classe**

```
class V {  
public:  
    double x, y;  
    static int nbre;  
    // ...  
};  
  
V v1, v2;
```



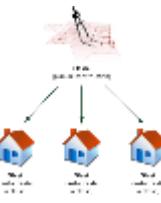
- S'il est public, on y accède soit sans référence à un objet en précisant la classe via l'**opérateur de portée ::**, soit comme membre d'un objet avec la notation pointée

Membres statiques

- Contrairement à une variable statique **locale** déclarée dans le **corps** de sa fonction, un **membre statique** d'une classe est déclaré dans la **déclaration** de la classe
- Dans un contexte de **compilation séparée**, sa déclaration est donc potentiellement incluse dans plusieurs fichiers source. On ne peut donc **initialiser ce membre** à l'endroit de sa déclaration.
- On l'**initialise** donc **explicitement** à **l'extérieur de la déclaration**, typiquement avec les définitions des fonctions membres
 - ❖ Attention, il n'y a pas d'initialisation à zéro par défaut
 - ❖ Un membre static constant peut être initialisé lors de sa déclaration mais uniquement s'il est de type entier (char, short, int, ...)

```
class V {  
    static int n;  
public:  
    double x, y;  
    static int m;  
    // ...  
};  
  
int V::n = 1;  
int V::m = 2;
```

On ne répète pas le mot **static** dans la partie **définition**



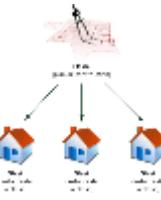
Fonctions membres statiques

- Une **fonction membre** peut aussi être déclarée **static**
 - Elle ne s'applique pas à un objet spécifique
 - Elle n'a pas accès aux membres non statiques
 - Si elle est publique, on y accède via l'opérateur de portée, p.ex.
MaClasse::uneMethodeStatique()
 - Si on la définit hors-ligne, on ne répète pas le mot-clé **static** lors de la définition

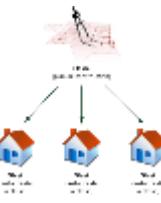
```
class V {  
public:  
    static void f();  
};
```

```
void V::f() {  
    // ...  
}
```

```
V::f();  
  
V v;  
v.f();
```



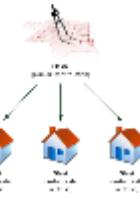
7. Membres particuliers



Membres particuliers

Les fonctions membres suivantes ont la particularité d'être **définies implicitement** par le compilateur (dans certaines circonstances)

| | |
|-----------------------------|--|
| Constructeur par défaut | <code>C();</code> |
| Destructeur | <code>~C();</code> |
| Constructeur de copie | <code>C(const C&);</code> |
| Opérateur d'affectation | <code>C& operator=(const C&);</code> |
| Constructeur de déplacement | <code>C(C&&);</code> |
| Opérateur de déplacement | <code>C& operator=(C&&);</code> |



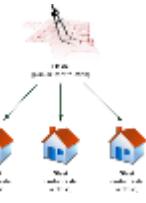
Constructeur par défaut (rappel)

- Si aucun constructeur n'est déclaré explicitement, le compilateur ajoute le constructeur par défaut **implicitement**. Ainsi, pour la classe

```
class C {  
    int data;  
};
```

on peut créer un objet en appelant ce constructeur

```
C c;
```



Constructeur par défaut (rappel)

- Si la classe déclare explicitement un autre constructeur, le constructeur par défaut n'est pas ajouté implicitement.

```
class C {  
    int data;  
public:  
    C(int val) : data(val) {}  
};
```

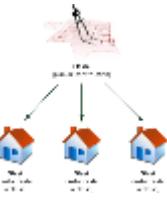
- Il n'est pas possible de créer l'objet c2 sans paramètre

```
C c1(100);  
C c2; // No matching constructor for initialization of 'C'
```

- Il faudrait explicitement ajouter `c() {}` ou `c() = default;` dans la zone publique de la déclaration de C ci-dessus

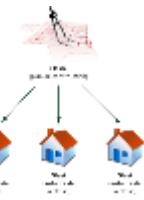
Destructeur

- Le destructeur est une fonction sans type de retour, sans paramètre, de même nom que la classe mais précédée d'un tilde : `~`
- Il est appelé quand un objet est détruit, ce qui arrive quand
 - le programme s'arrête – pour une variable créée **statiquement** (i.e. *globale* ou *statique*)
 - le programme sort de la fonction où l'objet variable *locale* a été créé **automatiquement**
 - le programmeur l'efface explicitement (**delete**) pour un objet créé **dynamiquement** (**new**)
- Typiquement, on définit explicitement le destructeur pour **libérer la mémoire allouée dynamiquement** par l'objet
- Un destructeur vide `~C() {}` est ajouté par le compilateur



Constructeur de copie

- Le constructeur de copie est un constructeur dont le seul paramètre est un objet du même type.
Sa signature est typiquement `C(const C&);`
- Si aucun constructeur de copie (ni constructeur de déplacement) n'est défini explicitement, le compilateur crée implicitement un constructeur de copie qui effectue une **copie superficielle** membre à membre, ce qui est souvent suffisant
- Il faut écrire explicitement un constructeur de copie si cette **copie superficielle** (*shallow copy*) ne suffit pas, typiquement quand certaines données membres de la classe sont des **pointeurs**, ce qui peut nécessiter une **copie profonde** (*deep copy*) des valeurs pointées (car recopier les adresses ne crée pas des copies des valeurs)



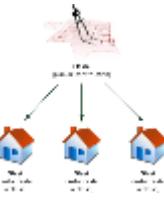
Opérateur d'affectation

- Les mêmes considérations s'appliquent à l'affectation
- Par défaut, l'opérateur

```
C& operator=(const C&);
```

est défini implicitement par le compilateur et effectue une **copie superficielle**

- En présence de membres pointeurs, il convient souvent de surcharger explicitement cet opérateur pour qu'il effectue une **copie profonde**

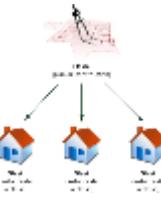


Déplacement

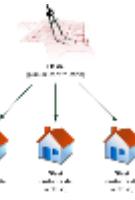
- Pour être complet, notons que depuis C++11, il est également possible de définir des **constructeurs et opérateurs de déplacement**, notés

```
C(C&&);           // move-constructor  
C& operator=(C&&); // move-assignment
```

- Il s'agit d'une **optimisation** permettant d'éviter d'effectuer une copie inutile d'un **objet temporaire** – valeur de retour d'une fonction, résultat d'une conversion de type, etc.
 - qui disparaîtrait juste après. Au lieu de cela, on déplace (**move**) tous ses membres. L'étude des constructeurs et opérateurs de déplacement sort du cadre de PRG1. Ceci sera étudié dans le cours ASD.
- Si aucun des éléments suivants n'est défini **explicitement** - *destructeur, constructeur de copie, opérateur d'affectation, constructeur et opérateur de déplacement* - le compilateur **définit implicitement** le constructeur et l'opérateur de déplacement

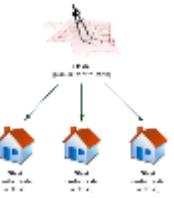


8. Conclusion



Conclusion

- Les principales notions vues dans ce chapitre ont été :
 - déclarations, données membres, fonctions membres
 - constructeurs (y.c. ceux ajoutés par le compilateur)
 - surcharge d'opérateurs
 - fonctions amies
 - membres constants et/ou statiques



En résumé

