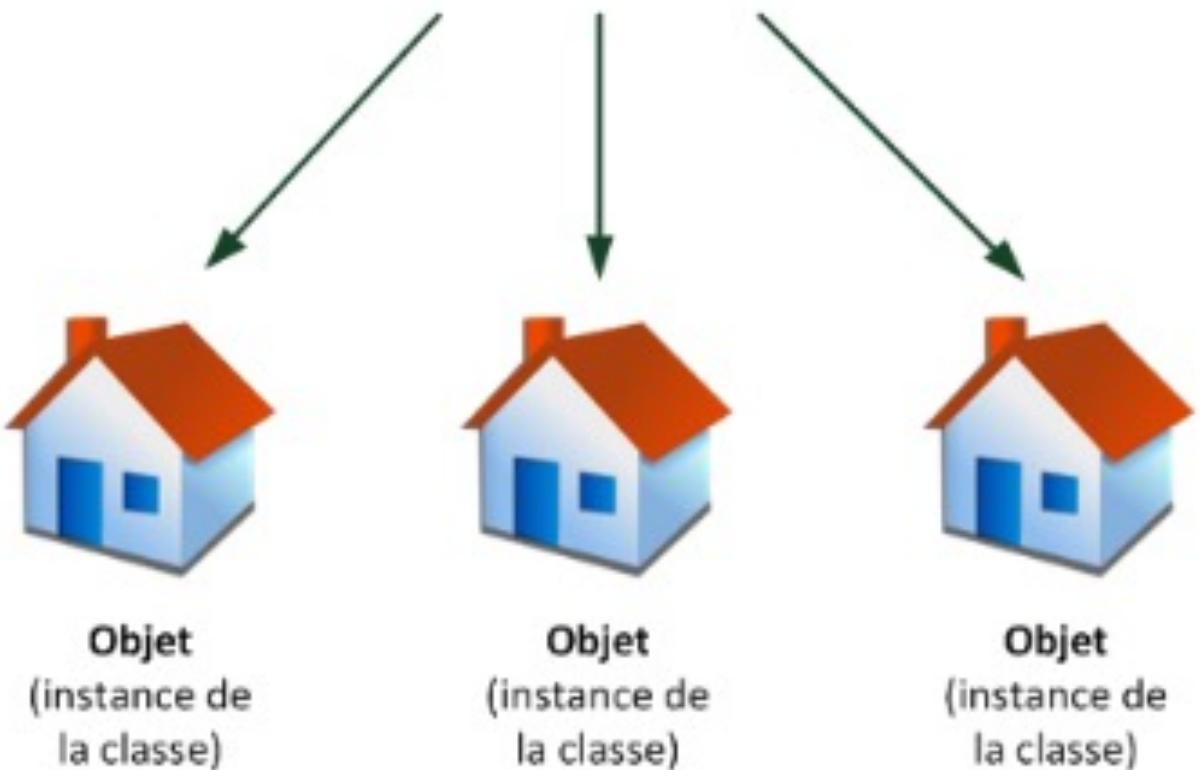


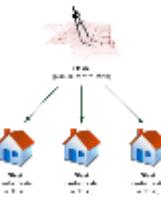
Chap. 11

Classes

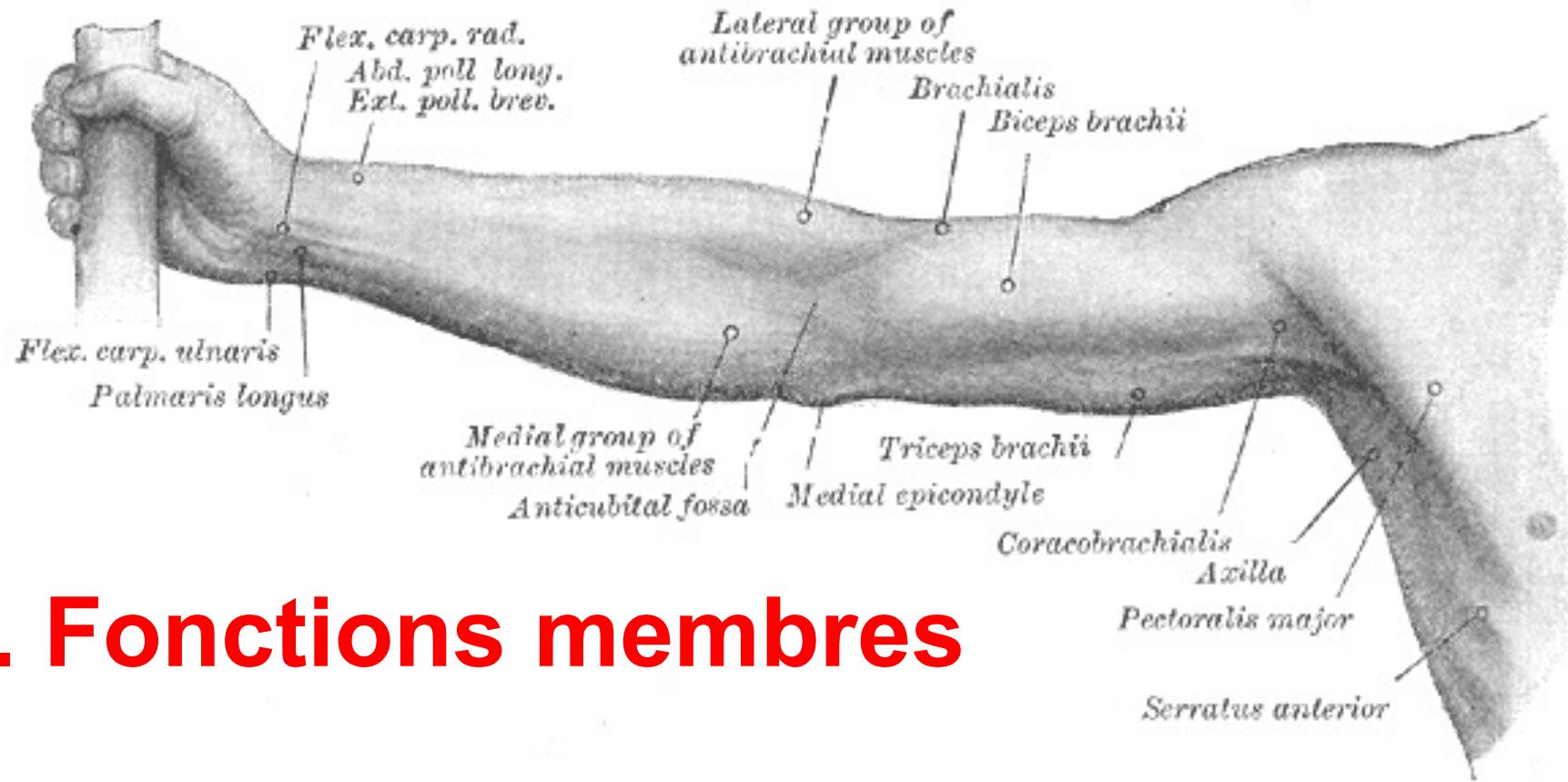


Classe
(plan de construction)

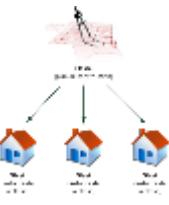




- Fonctions membres
- private vs. Public
 - class vs. struct
- Constructeurs
- Surcharge des opérateurs
 - operator=, operator[], operator(), conversions
- Attributs et fonctions membres statiques
- Compilation séparée



1. Fonctions membres

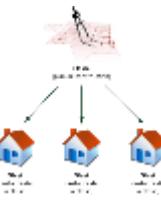


struct et fonctions externes

- En travaillant avec des **struct**, on crée de nombreuses fonctions prenant une **struct** en paramètre, soit par **&**, soit par **const&**

```
struct Date {  
    int jour; int mois; int annee;  
};  
  
void afficher(Date const& d);  
bool est_bissextile(Date const& d);  
int jours_du_mois(Date const& d);  
void incrementer(Date& d, unsigned n);  
  
int main() {  
    Date date{24, 1, 1798};  
    incrementer(date, 1905);  
    afficher(date);  
}
```

```
void afficher(Date const& d) {  
    cout << d.jour << '/'  
        << d.mois << '/' << d.annee;  
}  
  
void incrementer(Date & d, unsigned n) {  
    d.jour += n;  
    while (d.jour > jours_du_mois(d)) {  
        d.jour -= jours_du_mois(d);  
        ++d.mois;  
        if(d.mois > 12) {  
            d.mois = 1;  
            ++d.annee;  
        } } }
```

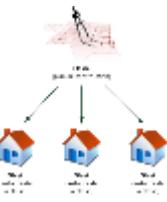


struct et fonctions membres

- En C++, on peut clarifier le code en transformant ces fonctions externes en fonctions membres

```
struct Date {  
    int jour; int mois; int annee;  
  
    void afficher() const;  
    bool est_bissextile() const;  
    int jours_du_mois() const;  
    void incrementer(unsigned n);  
};  
  
int main() {  
    Date date{24, 1, 1798};  
    date.incrementer(1905);  
    date.afficher();  
}
```

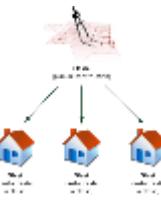
```
void Date::afficher() const {  
    cout << jour << '/'  
        << mois << '/' << annee;  
}  
  
void Date::incrementer(unsigned n) {  
    jour += n;  
    while (jour > jours_du_mois()) {  
        jour -= jours_du_mois();  
        ++mois;  
        if(mois > 12) {  
            mois = 1;  
            ++annee;  
        } } }
```



Comparons les 2 approches

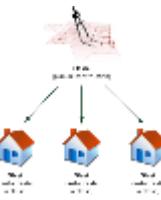
```
struct Date {  
    int jour; int mois; int annee;  
};  
  
void afficher(Date const & d);  
void incrementer(Date & d, unsigned n);  
  
void afficher(Date const & d) {  
    cout << d.jour << '/'  
        << d.mois << '/' << d.annee;  
}  
  
void incrementer(Date & d, unsigned n) {...}  
  
int main() {  
    Date date{24, 1, 1798};  
    incrementer(date, 1905);  
    afficher(date);  
}
```

```
struct Date {  
    int jour; int mois; int annee;  
    void afficher() const;  
    void incrementer(unsigned n);  
};  
  
void Date::afficher() const {  
    cout << jour << '/'  
        << mois << '/' << annee;  
}  
  
void Date::incrementer(unsigned n) {...}  
  
int main() {  
    Date date{24, 1, 1798};  
    date.incrementer(1905);  
    date.afficher();  
}
```



Un fonction membre ...

- Est déclarée dans la structure dont elle est membre
- Est déclarée **const** si elle ne modifie pas le contenu de la structure, sans qualification **const** si elle le modifie
- Utilise l'opérateur de résolution de portée `::` quand on la définit en dehors de la structure. On peut aussi la définir en ligne là où elle est déclarée
- Est appelée depuis l'extérieur en utilisant l'opérateur `.`, comme pour les attributs. Ou via l'opérateur `->` si l'on a un pointeur sur la structure
- Reçoit implicitement la (variable de type) structure qui l'appelle en paramètre caché
- Peut accéder aux attributs de cette structure par leur nom, sans devoir préciser à quelle variable de type struct on se réfère
- Peut accéder aux autres fonctions membres par leur nom

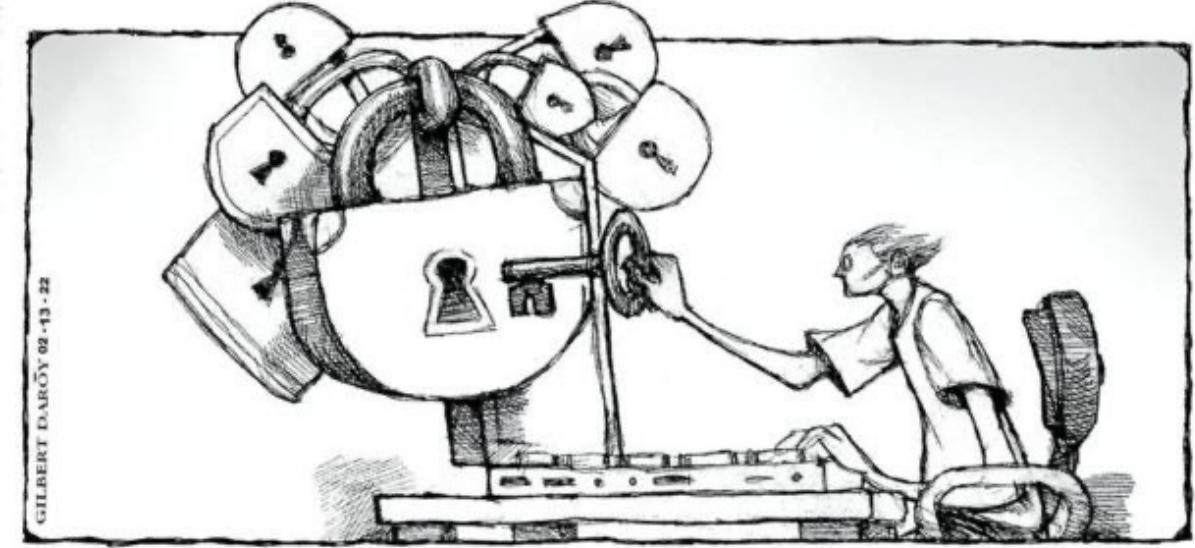


this

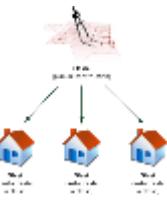
- Dans le code d'une fonction membre, le mot clé **this** donne accès à l'adresse de la structure courante.
- ***this** est donc la structure courante
- Il y a donc 3 syntaxes pour accéder à un attribut ou à une fonction membre.
- Utiliser **this** est nécessaire si une variable locale cache l'attribut

```
void Date::afficher() const {  
    cout << jour << '/'  
        << (*this).mois << '/'  
        << this->annee;  
}
```

```
void Date::modifier_jour(int jour) {  
    this->jour = jour;  
}
```



2. public vs. private



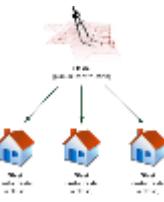
Limites des structures publiques

- Une structure telle que `Coord3d` est appropriée car toutes les valeurs de `x`, `y`, `z` sont pertinentes
- Une structure telle que `Date` est problématique car `jour` et `mois` doivent respecter des règles pour faire sens
 - `mois` entre 1 et 12
 - `jour` entre 1 et `jour_du_mois()`
- La syntaxe des structures permet d'écrire `Date date{55,12,1797};`, ou de donner la valeur `13` à `mois`, ce qui n'a pas de sens

```
struct Coord3d {  
    double x, y, z;  
};
```

```
struct Date {  
    int jour, mois, annee;  
};
```

```
int main() {  
    Date date{55,12,1797};  
    date.mois = 13;  
}
```



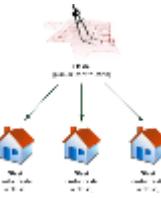
Sections private: et public:

- Au sein de la déclaration d'une **struct** (ou d'une **class**), on peut spécifier des sections privées ou publiques avec les mots clés **private:** et **public:**
- Les attributs et fonctions membres déclarés privés sont inaccessibles depuis l'extérieur.

```
Date date;  
date.mois = 13;
```

error: 'mois' is a
private member of 'Date'

```
class Date {  
private:  
    int jour, mois, annee;  
public:  
    void setDate(int j, int m, int a);  
    int getJour() const { return jour; }  
    int getMois() const { return mois; }  
    int getAnnee() const { return annee; }  
private:  
    bool estValide(int j, int m, int a) const;  
};  
  
void Date::setDate(int j, int m, int a) {  
    if (estValide(j, m, a)) {  
        jour = j; mois = m; annee = a;  
    }  
}  
  
bool Date::estValide(int j, int m, int a) const  
{ ... }
```

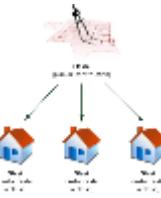


struct vs. class

- En C++, la seule différence entre une **struct** et une **class**, est qu'avant la première indication de section **private:** ou **public:**, les attributs / fonctions membres d'une **struct** sont publics, ceux d'une **class** privés.
- En pratique,
 - On utilise toujours **struct** si toutes les données sont publiques
 - On utiliser toujours **class** si toutes les données sont privées
 - On évite les structures hybrides mélangeant données publiques et privées

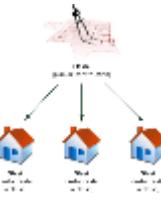
```
class Date {  
    int jour, mois, annee; // privés  
  
public:  
    void setDate(int j, int m, int a);  
    int getJour() const { return jour; }  
    int getMois() const { return mois; }
```

```
struct Date {  
    int jour, mois, annee; // publics  
  
public:  
    void setDate(int j, int m, int a);  
    int getJour() const { return jour; }  
    int getMois() const { return mois; }
```



- On appelle « objet » les variables de type « class ... »
- date1 et date2 sont deux objets de type Date

```
class Date {  
private:  
    int jour, mois, annee;  
public:  
    ...  
};  
  
int main() {  
    Date date1;  
    date1.setDate(24,1,1798);  
  
    Date date2;  
    ...  
}
```



Portée de la notion « privé »

- La notion de « privé » vs. « public » est définie au niveau de la `class`, pas des objets
- La fonction `Date::avant` de l'objet `*this` a accès aux données privées de l'objet `autre` parce qu'elles sont de la même classe
- Une fonction externe ou une autre classe n'y a pas accès

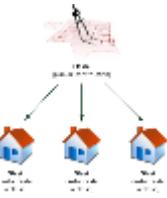
```
bool operator<(Date const& lhs, Date const& rhs) {
    return lhs.annee < rhs.annee ? true :
           lhs.mois < rhs.mois ? true :
           lhs.jour < rhs.jour;
}
```

```
class Date {
private:
    int jour, mois, annee;
public :
    bool avant(Date const& autre) const;
    ...
};

bool Date::avant(const Date& autre) const {
    return annee < autre.annee ? true :
           mois < autre.mois ? true :
           jour < autre.jour;
}
```

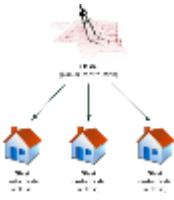
error: 'annee' is a private member of 'Date'

Fonctions amies



- Une classe peut donner explicitement accès à ses attributs / fonctions membre privées en déclarant une fonction amie avec le mot clé `friend`
- La déclaration d'amitié peut être placée indifféremment dans une section publique ou privée
- La déclaration d'amitié tient lieu de déclaration. Il n'est pas nécessaire de la répéter hors de la classe.

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
...  
    friend bool operator< (Date const& lhs,  
                           Date const& rhs);  
};  
  
bool operator< (Date const& lhs,  
                Date const& rhs) {  
    return lhs.annee < rhs.annee ? true :  
           lhs.mois < rhs.mois ? true :  
           lhs.jour < rhs.jour;  
}
```



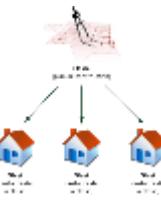
- Quand B déclare la classe A amie, toutes les fonctions membres de A ont accès aux attributs / fonctions membres privés de B
- Quand deux classes se réfèrent l'une à l'autre, il est nécessaire d'en pré-déclarer une pour pouvoir écrire la déclaration de l'autre

```
class A; // pré-déclaration

class B {
    int a;
    friend A;
};

class A {
public:
    void f(B& b) const;
};

void A::f(B& b) const {
    b.a = 42; // ok, même
    // si B::a est un membre
    // privé de B
}
```



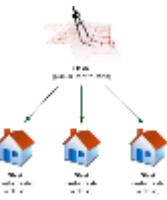
Sélecteurs et modificateurs

- On donne parfois accès aux attributs privés via des fonctions publiques
- Un **sélecteur** retourne la valeur d'un attribut privé. Il est typiquement déclaré **const**
- Un **modificateur** écrit dans un attribut privé. Il n'est jamais déclaré **const**
- Normalement, les modificateurs ne sont pas tous triviaux / pas tous présents, sinon il serait plus pertinent d'utiliser une **struct** avec des attributs publics.

```
class Date {  
private:  
    int jour;  
    int mois;  
    int annee;  
  
public :  
    // sélecteurs  
    int getJour() const { return jour; }  
    int getMois() const { return mois; }  
    int getAnnee() const { return annee; }  
  
    // modificateurs  
    void setJour(int j);  
    void setMois(int m);  
    void setAnnee(int a);  
    ...  
}
```



3. Constructeurs Destructeurs



Initialisation ?

- En déclarant les attributs de `Date` privés, on perd la possibilité d'initialiser l'objet date avec un agrégat
- Mais l'alternative

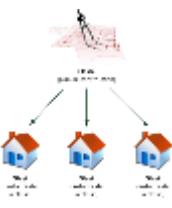
```
Date date;  
date.setDate(24, 1, 1798);
```

n'est pas acceptable, l'objet date étant dans un état indéterminé avant l'appel à `setDate`

- Il faut un mécanisme pour initialiser les attributs au moment de la déclaration de l'objet

```
class Date {  
private:  
    int jour;  
    int mois;  
    int annee;  
  
...  
};  
  
int main() {  
    Date date{24, 1, 1798};  
}
```

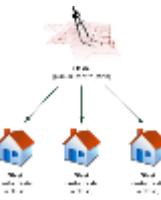
error: no matching
constructor for
initialization of 'Date'



Initialiser via un 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
- Les attributs sont initialisés via la **liste d'initialisation**, précédée de :, entre la liste des paramètres et le corps de la fonction constructeur
- On peut aussi écrire dans les attributs en y affectant une valeur dans le corps du constructeur

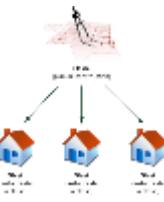
```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    Date(int j, int m, int a);  
};  
  
Date::Date(int j, int m, int a)  
: jour(j), mois(m), annee(a)  
{  
    if (mois < 1) mois = 1;  
    else if (mois > 12) mois = 12;  
    // + autres vérifications  
}  
  
int main()  
{  
    Date date(24, 1, 1798);  
}
```



Surcharge des constructeurs

- Plusieurs constructeurs peuvent être définis selon les règles usuelles de surcharge des fonctions (nombres de paramètres, types, ...)

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    Date();                      // valeur par défaut: Date(1, 1, 1970)  
    Date(int j, int m, int a);   // jour, mois, année  
    Date(Date const& d);        // copie de la date d  
    Date(std::chrono::year_month_day const&);  
                                // conversion depuis le type year_month_day de la STL  
    ...  
}
```



Constructeur par défaut

- Un constructeur sans paramètre est appelé **constructeur par défaut**. Il est appelé en cas
 - d'initialisation par défaut

```
Date d1; // sans parenthèses
```

- d'initialisation par valeur

```
Date d2 {}; // depuis C++11  
Date d3 = Date();
```

- Mais attention, la syntaxe ci-dessous ne déclare pas un objet d4 par valeur mais une fonction d4 retournant une Date

```
Date d4(); // déclare une fonction
```

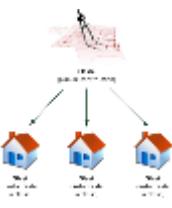
- Notons qu'un constructeur peut aussi en appeler un autre via la liste d'initialisation. Une mise en œuvre alternative serait ...

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    Date();  
    ...  
}
```

```
Date::Date() : jour(1),  
mois(1), annee(1970)  
{ }
```

```
int main() {  
    Date date;  
    // contient {1,1,1970}  
}
```

```
Date::Date()  
: Date(1, 1, 1970) { }
```



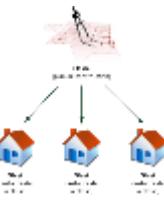
Constructeur par défaut ... par défaut

- Revenons à notre code utilisant setDate plutôt qu'un constructeur ... il ne compile plus !
- La ligne `Date date;` utilisait le constructeur par défaut, qui était défini implicitement en l'absence d'autre constructeur
 - Sa version par défaut ne fait rien. Elle est équivalente à `Date::Date() {}`
- En présence d'un autre constructeur, ici `Date(int j, int m, int a);`, le constructeur par défaut n'est plus créé implicitement
- Si nécessaire, on faut l'écrire explicitement, ou utiliser sa version par défaut

```
Date::Date() = default;
```

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    Date(int j, int m, int a);  
    void setDate(int j, int m, int a);  
};  
  
int main() {  
    Date date;  
    date.setDate(24, 1, 1798);  
}
```

```
error: no matching constructor for  
initialization of 'Date'
```

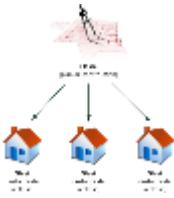


Constructeur à un paramètre

- Déclarer un constructeur à un paramètre crée une **conversion implicite** du type du paramètre vers celui de la classe.
- On peut empêcher la conversion implicite en qualifiant le constructeur **explicit**. Seule une **conversion explicite** est alors possible.
- Les règles de surcharge des fonctions s'appliquent toujours quand plusieurs constructeurs sont appelables.
 - c1 utilise le 1^{er} constructeur.
 - c2 ne peut pas appeler implicitement le 2^{ème} constructeur. Il utilise le 1^{er} constructeur via une conversion implicite préalable de **int** vers **double**
 - c3 utilise le 2^{ème} constructeur car la conversion est explicite

```
class C {  
    double d;  
public:  
    C(double _d) : d(_d) {  
        cout << "d";  
    }  
    explicit C(int i) : d(i) {  
        cout << "i";  
    }  
};  
  
int main() {  
    C c1 = 2.3; // affiche d  
    C c2 = 3; // affiche d  
    C c3 = C(4); // affiche i  
}
```

Constructeur de copie



- Dans le code ci-contre, quel constructeur est appelé pour initialiser date2 ?
- Le constructeur par copie prend en paramètre un objet du même type, typiquement par référence constante
- S'il n'est pas défini explicitement, le compilateur en crée une version par défaut qui copie les attributs, i.e.

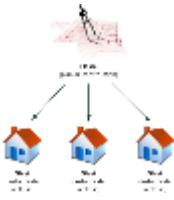
```
Date(Date const& d)
: jour(d.jour), mois(d.mois), annee(d.annee)
{ }
```

```
class Date {
private:
    int jour, mois, annee;

public :
    Date(int j, int m, int a)
        : jour(j), mois(m), annee(a) {}

    Date() : Date(1,1,1970) {}

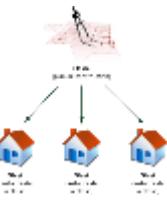
int main() {
    Date date1(24, 1, 1798);
    Date date2 = date1;
}
```



- 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 `~Date() {}` est ajouté par le compilateur



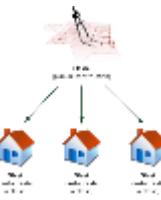
4. Surcharge des opérateurs



Surcharge par fonction externe

- Comme au chapitre 10, on peut surcharger un opérateur via une fonction externe
- Inconvénient:
 - l'opérateur doit être écrit en n'utilisant que l'interface public de la classe
 - sans accès aux attributs / fonctions privées

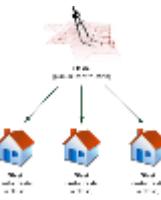
```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    ...  
    int getJour() const { return jour; }  
    int getMois() const { return mois; }  
    int getAnnee() const { return annee; }  
};  
  
bool operator< (Date const& lhs,  
                 Date const& rhs) {  
    return lhs.getAnnee() < rhs.getAnnee() ? true :  
           lhs.getMois() < rhs.getMois() ? true :  
           lhs.getJour() < rhs.getJour();  
}
```



Surcharge par fonction amie

- En faisant de l'opérateur une fonction amie, il a accès aux attributs / fonctions privées de la classe

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    ...  
  
    friend bool operator<(Date const& lhs,  
                           Date const& rhs);  
};  
  
bool operator< (Date const& lhs,  
                Date const& rhs) {  
    return lhs.annee < rhs.annee ? true :  
           lhs.mois < rhs.mois ? true :  
           lhs.jour < rhs.jour;  
}
```

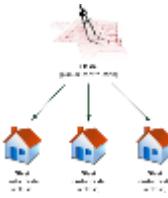


Surcharge par fonction membre

- Quand le premier paramètre de l'opérateur est du type de la classe, on peut l'écrire comme fonction membre de cette classe.
- Avantage : accès aux données privées
- Inconvénient : possible uniquement pour les opérateurs unaires ou binaires dont l'opérande gauche est du type de la classe. Par exemple, les opérateurs ci-dessous ne peuvent être mis en œuvre comme fonction membre:

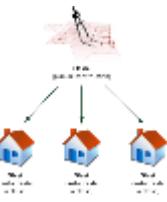
```
Date operator+(int n, Date const& d);  
istream& operator>> (istream & in, Date & rhs);
```

```
class Date {  
private:  
    int jour, mois, annee;  
public :  
    ...  
    bool operator< (Date const& rhs) const;  
};  
  
bool Date::operator< (Date const& rhs) const {  
    return this->annee < rhs.annee ? true :  
        this->mois < rhs.mois ? true :  
        this->jour < rhs.jour;  
}
```



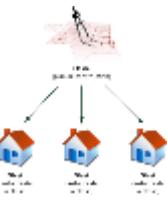
- Quand la fonction externe reçoit le 1^{er} paramètre par `const&` ou par valeur, la fonction membre est déclarée `const`.
- Quand la fonction externe reçoit le 1^{er} paramètre par `&`, la fonction membre n'est pas déclarée `const`.

```
class C {  
    ...  
public:  
    C operator- (C const& rhs) const;  
    C& operator-= (C const& rhs);  
    C& operator-- ();      // prefix  
    C operator-- (int);   // postfix  
};  
C operator+ (C const& lhs, C const& rhs);  
C& operator+= (C& lhs, C const& rhs);  
C& operator++ (C& c);      // prefix  
C operator++ (C& c, int); // postfix
```



Fonction membre uniquement

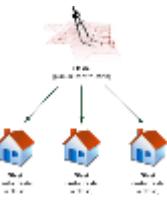
- Certains opérateurs ne peuvent être surchargés que sous forme de fonction membre
 - `operator=`, l'opérateur d'affectation
 - `operator[]`, l'opérateur d'accès aux éléments d'un tableau
 - `operator()`, l'opérateur d'appel de fonction qui permet de créer des foncteurs
 - `operator To()`, conversion implicite vers le type To
 - `explicit operator To()`, conversion explicite vers le type To



Affectation : operator=

- Doit bien se comporter en cas d'auto-affectation
- Retourne `*this`
- Est fourni par défaut pour la copie d'un objet du même type. N'est écrit explicitement que lorsque l'on gère des ressources (voir chap. 15)
- Peut affecter depuis d'autres types que celui de la classe par surcharge

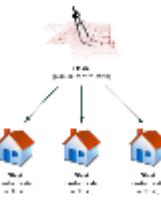
```
class C {  
public:  
    C& operator= (C const& other);  
};  
  
C& C::operator=(const C& other) {  
    if (this == &other) return *this;  
    // libérer les ressources liées à *this  
    // copier le contenu de other  
    return *this;  
}
```



Accès aux éléments : operator[]

- Typiquement surchargé deux fois
 - Retourne une `const&` si l'objet est constant
 - Retourne une `&` une si l'objet n'est pas constant

```
class C {  
    vector<int> v;  
public:  
    int& operator[] (size_t i);  
    int const& operator[] (size_t i) const;  
};  
  
int& C::operator[] (size_t i) {  
    return v[i];  
}  
  
const int& C::operator[] (size_t i) const {  
    return v[i];  
}
```



Foncteurs : operator()

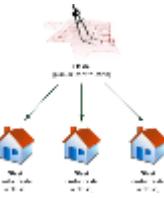
- L'opérateur () permet de créer des objets qui se comportent comme des fonctions
- **operator()** est
 - Précédé du type de retour de la fonction créée
 - Suivi des paramètres de cette fonction

```
struct Est_multiple_de
{
    int n;
    bool operator()(int a) const
    {
        return a % n == 0;
    }
};

int main() {
    auto est_pair = Est_multiple_de{2};
    auto est_rond = Est_multiple_de{10};

    int n; cin >> n;
    if (est_pair(n) and not est_rond(n)) {
        cout << n <<
            " est pair mais pas multiple de 10";
    }
}
```

Conversions



- Un constructeur à un paramètre permet de convertir vers la classe que l'on écrit
- Pour convertir d'une classe que l'on écrit vers une autre / vers un simple, seul l'opérateur de conversion est utilisable

```
int main()
{
    int i = 42;
    X x(i);                      // 1 implicite
    X y = static_cast<X>(i);      // 1 explicite
    X z = X(&i);                  // 2 explicite
    // X w = &i;      // ne compile pas : implicite
    int m = x;                    // 3 implicite
    int n = (int)x;                // 3 explicite
    int* p = static_cast<int*>(x); // 4 explicite
    // int* q = x;  // ne compile pas : implicite
}
```

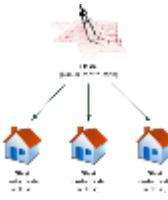
```
struct X
{
    // 1: int -> X implicite
    X(int) { }

    // 2: int* -> X explicite
    explicit X(int*) {}

    // 3: X -> int implicite
    operator int() const { return 7; }

    // 4: X -> int* explicite
    explicit operator int*() const {
        return nullptr;
    }
};
```

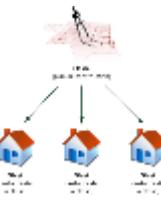
Membre, amie, ou externe ?



- On n'a pas toujours le choix ...
 - operator=, operator[], operator(), conversion seulement en fonction membre
 - Opérateurs dont l'opérande gauche n'est pas une classe modifiable seulement en fonction externe
- Si vous avez le choix, nous recommandons ...
 - Opérateur unaire (ex: `++`) ou binaire qui modifie son opérande gauche (ex: `+=`) :
Fonction membre
 - Opérateur binaire (ex: `+`) qui ne modifie pas son opérande gauche :
Fonction non membre ou amie si nécessaire

5. static



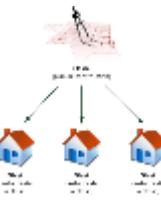


Membres constants

- Un membre d'une classe peut être déclaré `const`. Sa valeur non modifiable peut être initialisée
 - par la valeur donnée à sa déclaration
 - via la liste d'initialisation, ce qui remplace la valeur déclarée
- C'est une **mauvaise pratique** qui rend la classe incopiable et n'améliore pas la sécurité (il suffit de ne pas fournir de modificateur pour obtenir le même effet)

[Cpp Core guideline C.12](#): Don't make data members const or references in a copyable or movable type

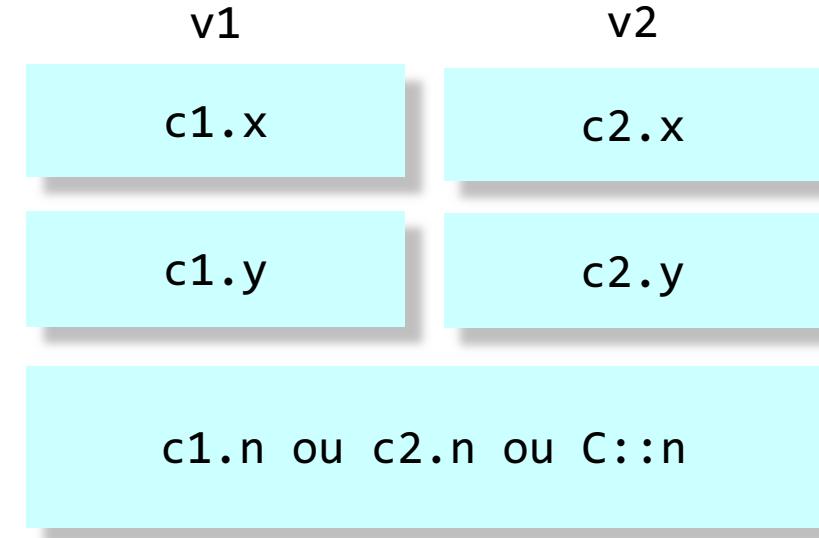
```
class C {  
public:  
    C() {}  
    C(int c) : cste(c) {}  
private:  
    const int cste = 1;  
};  
  
int main() {  
    C c1; // c1.cste = 1;  
    C c2(2); // c2.cste = 2;  
}
```



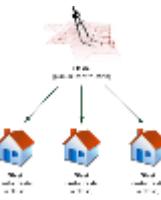
Membres statiques

- 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 C {  
public:  
    double x, y;  
    static int n;  
    // ...  
};  
  
C c1, c2;
```



- 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



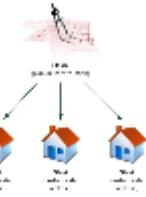
Membres statiques

- 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
- On l'initialise donc **explicitement à l'extérieur de la déclaration**, typiquement avec les définitions des fonctions membres

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

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

- ❖ 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, ...)



Fonctions membres statiques

Une fonction membre déclarée **static**

- ne s'applique pas à un objet spécifique
- n'a pas accès aux membres non statiques
- Si elle est publique, on y accède via l'opérateur de portée, p.ex.

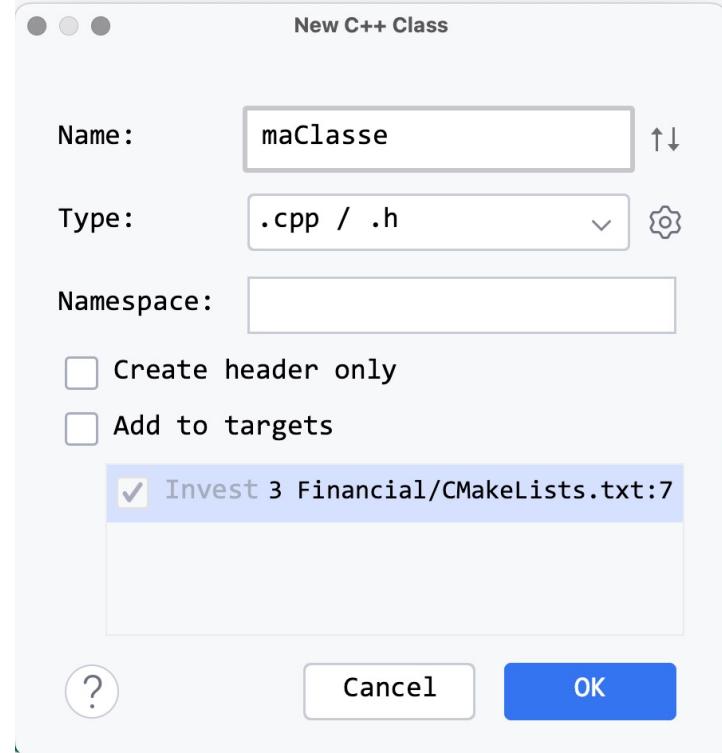
MaClasse::uneMethodeStatique()

- Si on la définit hors-ligne, on ne répète pas le mot-clé **static** lors de la définition

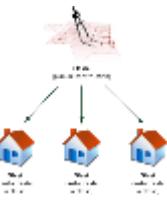
```
class C {  
public:  
    static void f();  
};
```

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

```
int main() {  
    C::f();  
  
    C c;  
    c.f();  
}
```

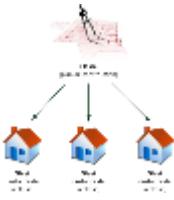


6. Compilation séparée



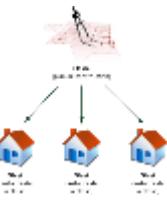
Organisation du code par classe

- 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 et opérateurs en ligne
- Le fichier **.cpp** inclut la **définition** des autres fonctions membres, ainsi que l'initialisation des variables et constantes statiques



```
#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) { /* ... */ }
    ...
}
```

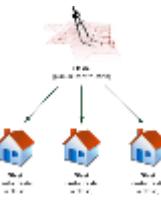


maClasse.h (suite)

```
...
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 */
```

- Dans un contexte de compilation séparée, la déclaration d'un membre statique est potentiellement incluse dans plusieurs fichiers source. On ne peut donc initialiser ce membre à l'endroit de sa déclaration. Un **membre statique** doit être **initialisé** dans **le fichier cpp**.



maClasse.cpp

```
#include "maClasse.h"

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

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

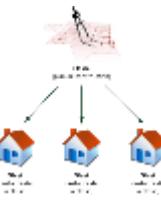
// un des constructeurs
MaClasse::MaClasse(int n, double x) : variable(n), constante(x) { /* ... */ }

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 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.