Chapitre 7: classes

Programmation orientée objet

- 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 savez donc utiliser des objets. En particulier
  - les initialiser par constructeur, avec divers paramètres
  - appeler leurs méthodes avec la notation `objet.methode(...)`
- Le but de ce chapitre est d'apprendre à créer vos propres objets, ou plus précisément à définir des classes, les types composés dont les instances (variables et constantes) sont les objets

Programmation orientée objet et encapsulation

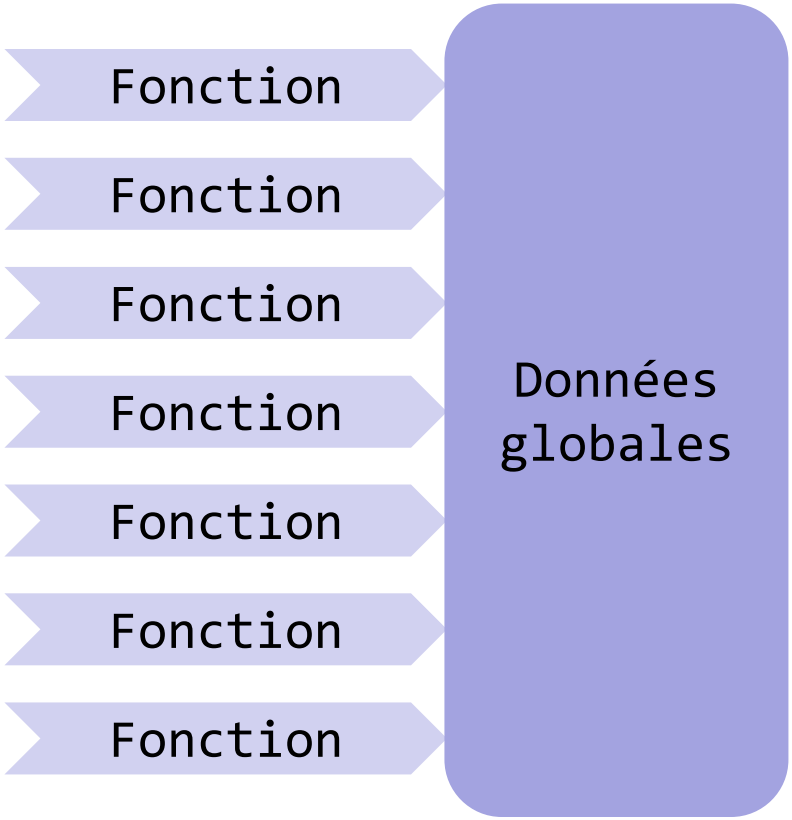
Pourquoi la POO ?

- Au tout début d'INF1, 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.

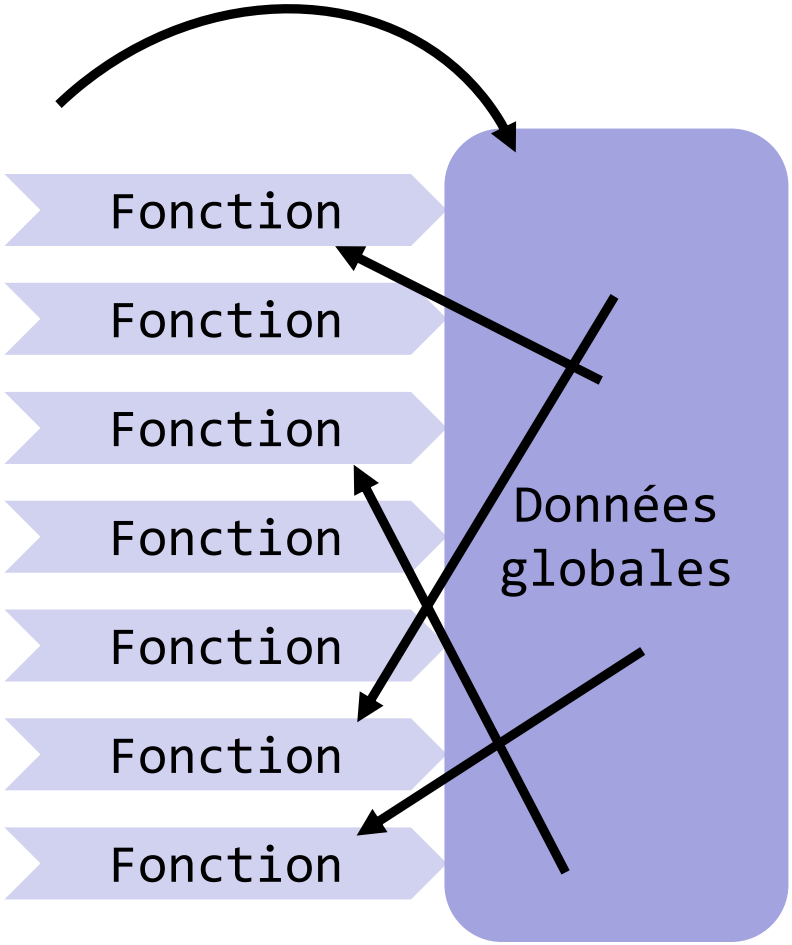
Cela permet à toutes les fonctions d'y accéder



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

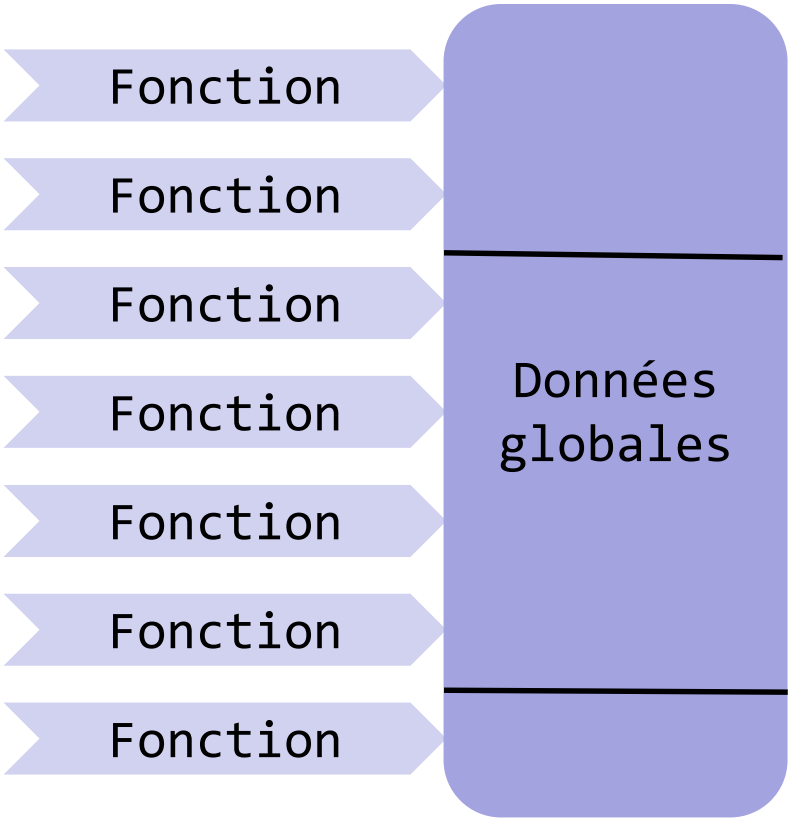


Pourquoi la POO ?

- 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

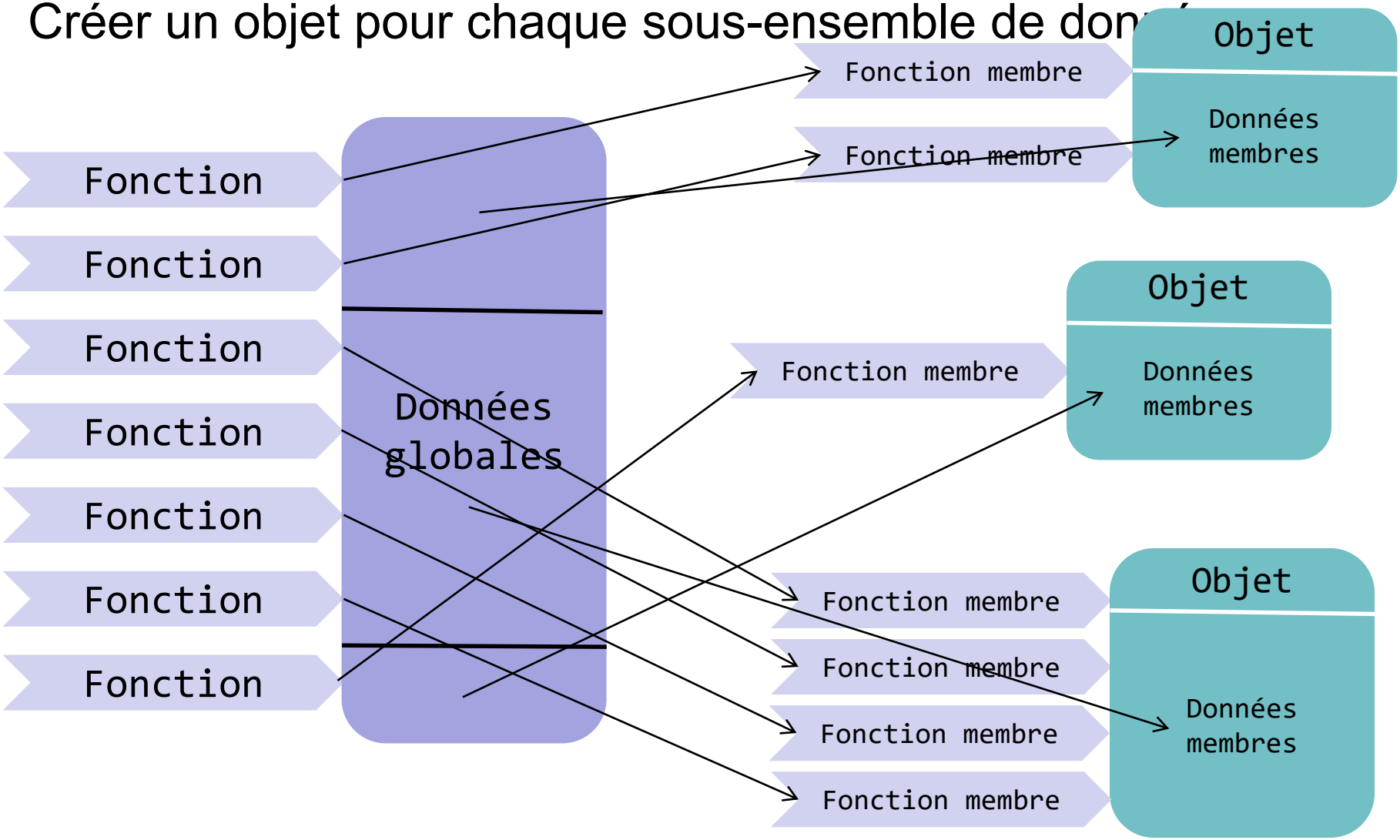
Programmation fonctionnelle -> POO

Déterminer quelles données vont avec quelles fonctions



Programmation fonctionelle -> POO

Créer un objet pour chaque sous-ensemble de don



Encapsulation / 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 ne met pas en œuvre un objet directement, mais on définit son type, qui permettra d'instancier un ou plusieurs objets de ce type
- Ce type s'appelle une classe

Autres concepts de POO

- 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

Classes

Déclaration

- Pour déclarer une classe, il suffit d'écrire

```
class NomDeLaClasse
{
public:
  // déclaration de l'interface public 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, double);
    // calcule et renvoie la surface du rectangle
    double surface() const;
private:
    // stocke les dimensions
    double largeur;
    double longueur;
};
```

Exemple – Rectangle

- Les déclarations suivantes sont équivalentes

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

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

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

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

Fonctions membres const

- Notons la présence du mot-clé `const` à la fin de la déclaration de la fonction membre `surface()`

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

- Elle indique qu'aucune donnée membre 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`

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;
}
```

Accès aux autres membres

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

- Soit directement via leur nom

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

- Soit en utilisant le mot-clé `this`: un pointeur vers l'objet; ou `(*this)`: l'objet lui-même.

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

- Utiliser `this` permet d'accéder aux membres même si une variable locale ou un paramètre porte le même nom

Client

- Une variable de type `class` s'utilise comme n'importe quelle autre variable
  - Variable ou constante
  - Allocation statique, automatique ou dynamique
  - Passage en paramètre par valeur, référence ou référence constante
- On accède aux membres en utilisant la notation pointée

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

Accès aux membres privés

- Essayer d'accéder aux membres privés 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)

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

  - modifier la donnée privée

    (un modificateur)

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

Constructeurs

Motivation

- Le code suivant n'est pas satisfaisant

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

- La première ligne crée l'objet `rect` mais les valeurs de `rect.largeur` et `rect.longueur` sont indéterminées. Un appel à `rect.surface()` donnerait un résultat indéterminé
- Il faudrait pouvoir initialiser l'objet. C'est possible en écrivant un ou plusieurs 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 aucune instruction `return`
- On améliore notre classe `Rectangle` en lui fournissant un constructeur initialisant les dimensions

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

Constructeur

- Le code client se réécrit alors plus proprement

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

- Notons que l'on peut aussi créer plusieurs instances du même type. Les données stockées dans chaque instance sont indépendantes et donc ce code

```
Rectangle recta(2.0, 3.0);
Rectangle rectb(5.0, 7.0);
cout << recta.surface() << ' ';
cout << rectb.surface();
```

affiche `6 35`

Surcharge de constructeur

- 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 constructeur

- Le code client est alors par exemple

```
Rectangle recta(2.0, 3.0);
Rectangle rectb(5.0);
Rectangle rectc;
cout << recta.surface() << ' ';
cout << rectb.surface() << ' ';
cout << rectc.surface();
```

```
6 25 0
```

- Cela demande quelques précisions…

Constructeur par défaut

- Le constructeur sans argument est appelé constructeur par défaut. Ici,

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

- Si aucun constructeur n'est déclaré, le compilateur en ajoute un d'office qui ne fait rien (mais pas si un autre constructeur existe)
- Le client appelle ce constructeur par défaut via

```
Rectangle recta;   // sans parenthèses
Rectangle rectb{}; // initialisation uniforme (C++11)
```

- Mais attention, pas via

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

Constructeur à un argument ou plus

- Pour appeler le constructeur à un argument

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

- Il y a 4 syntaxes possibles

```
Rectangle recta(1.0);     // fonctionnelle
Rectangle rectb = 2.0;    // affectation
Rectangle rectc{3.0};     // init. uniforme (C++11)
Rectangle rectd = {4.0}; // init. uniforme (C++11)
```

- Dans le cas général à 2 arguments ou plus, les syntaxes fonctionnelles et uniformes sont disponibles, mais pas celle par affectation

Initialisation des membres

Quand un constructeur sert à initialiser des membres de la classe, cela peut être fait sans recourir à des affectations, mais avec une liste d'initialisations de membres

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

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

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

Initialisation des membres

- Pour les membres variables de type simple, cela ne change rien
- Pour les membres variables de type composé (objets), cela change la méthode de création de l'objet
  - Affectation: Appel du constructeur par défaut puis de l'opérateur d'affectation. Si la classe de l'objet à créer n'a pas de constructeur par défaut, cette approche ne fonctionne pas.
  - Initialisation: Appel d'un constructeur recevant la valeur initiale en paramètre
- Pour les membres constants, seule l'utilisation de la liste d'initialisation est possible, l'opérateur d'affectation n'existant pas

Initialisation des membres

- Enfin, notons qu'il aurait pu paraître plus simple d'initialiser les membres ainsi

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

- Mais…
  - Ce n'est possible que depuis C++11
  - L'initialisation via un constructeur offre plus de flexibilité

Surcharge des opérateurs

Opérateurs sur nos classes

- Les classes définissent de nouveaux types utilisables dans notre code C++
- Pour les types simples, nous interagissions principalement avec ces types via des opérateurs tels que `=, +, -, *, ++, <, ==, <<, …`
- Parmi ces opérateurs, seul =, l'opérateur d'affectation, est défini par défaut pour une classe que vous créez.
- Mais il est possible de surcharger les opérateurs suivants

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

Syntaxe

- 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

```
type operator @ (parameters) { /*... body ...*/ }
```

- Les paramètres et le type de retour dépendent de l'opérateur que l'on veut surcharger

Exemple

- Prenons l'exemple d'une classe `CVector` qui représente des vecteurs cartésiens, i.e. des paires de coordonnées x et y.

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

- On fournit 2 constructeurs. L'un permettant d'initialiser x et y, l'autre étant le constructeur par défaut

operator +

- Pour ajouter un opérateur + permettant de sommer deux `CVector` à cette classe, il suffit de déclarer la fonction membre `operator +`

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

- Elle reçoit en paramètre une référence constante à un autre `CVector`
- Elle retourne un `CVector` qui est la somme 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 = x + v.x;
    temp.y = y + v.y;
    return temp;
}
```

- Notons que la fonction a accès aux membres privés de `param`, même s'il s'agit d'un autre objet que `*this`
- La notion de privé/public s'applique au niveau de la classe (un type), pas à celui des objets (des variables)

Client de CVector::operator +

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

```
int main() {
    CVector foo(3,1);
    CVector bar(1,2);
    CVector result = foo + bar;
    // result.x vaut 4, result.y vaut 3
}
```

- Notons que l'on pourrait aussi écrire

```
CVector result = foo.operator + (bar);
```

(mais personne ne le fait)

operator *

Rien n'oblige à ce que les deux opérandes d'un opérateur soient du même type. Ajoutons par exemple la multiplication d'un `CVector` par un réel

```
class CVector {
    double x, y;
public:
    … // idem exemple précédent
    CVector operator * (double) 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 foo(3,1);
    CVector result = foo * 2;
    // result.x vaut 6, result.y vaut 2
}
```

- Par contre, le code qui suit ne compile pas

```
CVector result = 2 * foo;
```

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

- En effet, il aurait fallu surcharger `operator *` dans le type `int` … ce qui est impossible, `int` étant un type simple en C++, pas une classe. Mais il y a une alternative...

Surcharge par fonction non membre

- Il est aussi possible de surcharger certains opérateurs par une fonction simple plutôt que par 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;
    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 suffirait 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 d'un `CVector` et d'un `double` dans les deux sens

- Malheureusement, le corps de notre fonction ne compile pas…

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

```
'x' is a private member of 'CVector'
'y' is a private member of 'CVector'
```

Amitié

- Le fonction `operator *` a besoin d'accéder aux membres privés de `CVector`
- Mais on ne peut rendre public 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 œuvre qui ne doivent pas être exposés aux 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);
  double x, y;
public:
     … // idem exemple précédent
};

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

Ce qui nous permet d'écrire dans le client

```
CVector foo(3,1);
CVector result = 2 * foo;
```

friend

Notons que la 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;
  }
  double x, y;
public:
     … // idem exemple précédent
};
```

friend

- Notons que l'amitié n'est pas réservée aux opérateurs. Une classe peut aussi déclarer comme amie
  - une fonction
  - une fonction membre d'une autre classe
  - une autre classe dans son ensemble
- Et chacune de ces fonctions/classes peut-être l'amie de plusieurs classes

```
class A;

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

class A {
  friend void B::changer(const 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 {
public:
  CVector operator *(double d) const;
  … // idem exemples précédents
};

CVector CVector::operator *
(double d) const
{
    CVector temp;
    temp.x = x * d;
    temp.y = y * d;
    return temp;
}

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

operator <<

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

```
class CVector {
  friend ostream& operator << (ostream&, const CVector&);
  … // idem exemples précédents
};

ostream& operator << (ostream& lhs, const CVector& rhs)
{
  lhs << rhs.x << ',' << rhs.y ;
  return lhs;
}
```

- Les paramètres sont
  - une référence au flux de sortie dans lequel écrire
  - une référence constante à l'objet à afficher
- Et on retourne une référence au flux

operator <<

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

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

```
3,1
1,2
4,3
```

- Passer le deuxième paramètre en référence constante permet d'afficher tant la variable `foo` que la constante `BAR` ou l'expression `foo + BAR`
- Retourner une référence au flux permet d'enchainer les `<<`

Opérateurs d'affectation composée

- Quand on définit un opérateur binaire telle 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 pas peut être déclaré en tant que fonction amie
- L'opérateur d'affectation retourne typiquement une référence vers l'objet qu'il affecte
- Par contre, l'opérateur binaire retourne typiquement l'objet par valeur

Opérateurs d'affectation composée

La forme canonique d'un opérateur arithmétique et de l'affectation composée correspondante est donc

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

  friend X operator + (X lhs, const X& rhs) {
    lhs += rhs; // appel à +=
    return lhs;
  }
};
```

Opérateurs relationnels

Pour les opérateurs de comparaison et d'égalité, on met typiquement en œuvre `operator<` et `operator==`, les autres les utilisant de manière standard. 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 !(lhs > rhs); }
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.
- 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 et retourne une copie de l'objet avant incrémentation

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

operator ++

- On peut aussi l'écrire avec des fonctions non membres

```
class A {
    // ...
    friend A& operator ++ (A&);
    friend A  operator ++ (A&, int);
};
// ++A
A& operator ++ (A& a) {
    // incrémenter les données de a
    return a;
}
// A++
A operator ++ (A& a, int) {
    A temp = a;
    // incrémenter les données de a
    return temp;
}
```

Résumé + autres opérateurs

Pour un objet **a** de classe **A**, **b** de classe **B**, … on surcharge l'opérateur **@ …** (à remplacer par le symbole approprié)
(**NB** 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()	

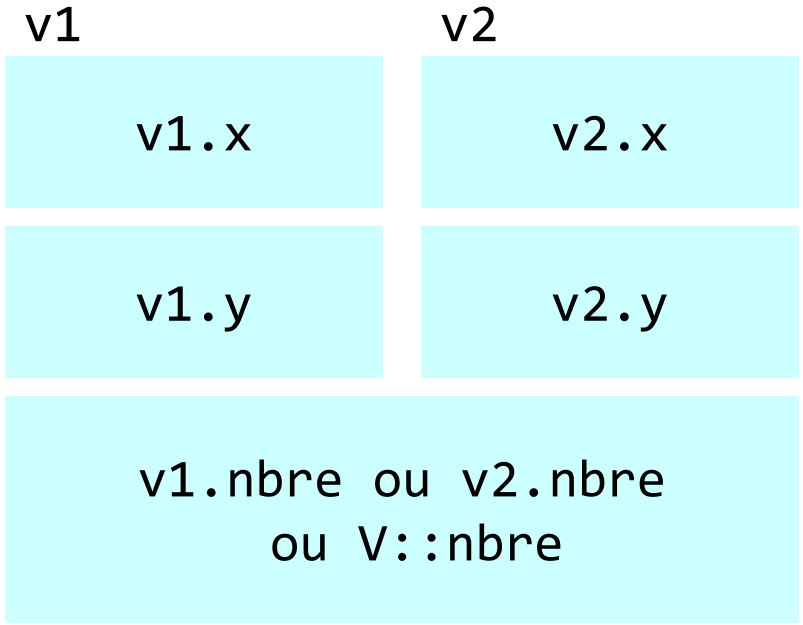
Membres constants et statiques

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

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

int V::n = 1;
int V::m = 2;
```

- Attention, il n'y a pas d'initialisation à zéro par défaut

Fonctions membres statiques

- Une fonction membre peut également 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 `laClasse::laMethodeStatique()`
  - 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();
```

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 {
private:
  const int TOUS = 1;
};
```

```
class V {
public:
  V(int c) : CSTE(c) {};
private:
  const int CSTE;
};

V v(3);
```

Membres constants

- Depuis C++11, on peut également initialiser une constante à sa déclaration. Cette valeur d'initialisation à la déclaration est une valeur par défaut, utilisée uniquement pour les constructeurs n'initialisant pas explicitement cette 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 particuliers

Membres particuliers

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

Constructeur par défaut	`C();`
Destructeur	`~C();`
Constructeur de copie	`C(const C&);`
Opérateur d'affectation	`C& operator=(const C&);`
Constructeur de déplacement	`C(C&&);`
Opérateur de déplacement	`C& operator=(C&&);`

Constructeur par défaut

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

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

on peut créer un objet en appelant ce constructeur implicite

```
C c;
```

Constructeur par défaut

- Par contre, si la classe déclare explicitement un autre constructeur, le constructeur par défaut n'est pas ajouté implicitement. Ainsi, pour cette classe

```
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() { }` 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 le déclare explicitement pour libérer la mémoire allouée dynamiquement par l'objet

- A défaut, 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 certains membres de la classe sont des pointeurs, ce qui peut nécessiter une copie profonde (deep copy)

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, … - qui disparaitrait juste après

- Si aucun des éléments suivants n'est défini explicitement - *destructeur, constructeur de copie, opérateur d'affectation, constructeur et opérateur de déplacement* - le compilateur défini implicitement les constructeur et opérateur de déplacement

Résumé

Résumé

- Pour résumer, voyons ces éléments tous ensembles dans le cadre de la compilation séparée
- Typiquement, chaque classe a ses propres fichiers header 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
{
private:
  int variable;
  const double constante;
  static char variableStatique;
  static const short CONSTANTE_STATIQUE;

public:
  // constructeurs
  MaClasse() : constante(0) { /* ... */ }
  MaClasse(int v, double c); // defini dans maClasse.cpp
…
```

maClasse.h

```
…
  // constructeur de copie
  MaClasse(const MaClasse& obj)
  : variable(obj.variable), constante(obj.constante)
    { /* ... */ }
  // destructeur
  ~MaClasse() { /* ... */ }

  static void fonctionStatique(int);
  void fonctionMembre(int);
  friend void fonctionAmie(MaClasse& c);

  static void fonctionStatiqueEnLigne() { /*...*/ }
  void fonctionMembreEnLigne() { /*...*/ }
  friend void fonctionAmieEnLigne(MaClasse& c) { /*...*/ }
};

#endif /* MACLASSE_H*/
```

maClasse.cpp

```cpp
#include "maClasse.h"

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

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

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

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

void fonctionAmie(MaClasse& obj)
{ /* ... */ }
```