

Programmation orienté objet en C++

IUT Lyon 1

Yosra ZGUIRA

yosra.zguira@insa-lyon.fr

2016 - 2017

Introduction

- **La programmation orientée objet (POO)** permet de rassembler dans une même entité appelée **objet** les données et les traitements (méthodes) qui s'y appliquent.

Méthodes + Données = Objet

- Un programme est alors constitué d'un ensemble d'objets communiquant par des requêtes (ou messages).
- La POO permet une modélisation plus naturelle des systèmes considérés et propose un ensemble de concepts indépendants de tout langage de programmation facilitant les représentations des objets :
 - ✓ du monde réel (personne, voiture, ...)
 - ✓ d'un domaine particulier (vecteur, matrice, complexe, ...)

Les strings en C++

Le type string

- La gestion du texte en mémoire est en fait complexe.
- Pour simplifier les choses, le langage C++ propose le type **string** qui peut créer des objets de type string et manipuler du texte.
- Chaque objet possède des attributs et des méthodes.
- Pour appeler la méthode d'un objet, on utilise cette écriture:

```
objet.methode();
```

Quelques méthodes utiles du type string (1/6)

❑ La méthode `size()`:

- La méthode `size()` permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string.
- Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne.

```
maChaine.size();
```

Quelques méthodes utiles du type string (2/6)

❑ La méthode erase():

- Cette méthode supprime tout le contenu de la chaîne de caractères.

```
maChaine.erase();
```

Quelques méthodes utiles du type string (3/6)

❑ La méthode `substr()`:

- Cette méthode permet d'extraire une partie de la chaîne stockée dans un string.
- Son prototype:

```
string substr( size_type index, size_type num = npos );
```

- ✓ **index** permet d'indiquer à partir de quel caractère on doit couper.
- ✓ **num** permet d'indiquer le nombre de caractères que l'on prend.
Par défaut, la valeur est `npos`, ce qui revient à prendre tous les caractères qui restent.


Quelques méthodes utiles du type string (4/6)

❑ La méthode substr():

- Exemple:

```
int main()
{
    string chaine("Bonjour !");
    cout << chaine.substr(3) << endl;

    return 0;
}
```



jour !

Quelques méthodes utiles du type string (5/6)

❑ La méthode `c_str()`:

- Cette méthode permet de renvoyer un pointeur vers le tableau de char que contient l'objet de type string.
- Il peut arriver que vous deviez envoyer à une fonction un tableau de char.
 - ➔ Dans ce cas, la méthode `c_str()` vous permet de récupérer l'adresse du tableau de char qui se trouve à l'intérieur de l'objet string.
- L'utilisation de `c_str()` reste assez rare puisque on préfère largement manipuler un objet string plutôt qu'un tableau de char c'est plus simple et plus sûr.

Quelques méthodes utiles du type string (6/6)

❑ La méthode `c_str()`:

- Exemple: Dans le chapitre précédent, nous en avons eu besoin pour indiquer le nom du fichier à ouvrir:

```
string const nomFichier("C:/C++/Fichiers/monFichier.txt");  
  
ofstream monFlux(nomFichier.c_str());
```

Les classes

La notion de classe

- **Une classe** permet de regrouper dans une même entité:
 - ✓ de variables, ici appelées **attributs** (on parle aussi de **variables membres**) ;
 - ✓ de fonctions, ici appelées **méthodes** (on parle aussi de **fonctions membres**).
- La classe est la notion de base de la programmation orientée objet.
- Il s'agit d'une évolution de la notion de **structure**, qui apporte de nouvelles notions orientées objet absolument fondamentales.
- Un **objet** est un élément d'une certaine classe : on parle de **l'instance d'une classe**.

Squelette d'une classe

```
class Nom_class {  
  
    // Définition des attributs  
    ...  
  
    // Définition des méthodes  
    ...  
  
};
```



Exemple

```
class Point {  
  
    int coord_x;  
    int coord_y;  
  
    void SetCoord(const int & x, const int & y);  
    void Afficher();  
  
};
```



- Il faut que le nom des classes commence toujours par une lettre majuscule !

➔ C'est utile pour différencier les noms des classes des noms des objets.

Droits d'accès (1/2)

- Chaque attribut et chaque méthode d'une classe doit définir un droit d'accès par l'un des mots-clés **public**, **private** ou **protected**.
 - ✓ **L'accès public:** un membre(attribut / méthode) public est accessible depuis n'importe quelle fonction.
 - ✓ **L'accès private:** un membre private est accessible uniquement au sein de la classe dans laquelle il est déclaré.
 - ✓ **L'accès protected:** un membre protected est accessible uniquement aux classes d'un package et à ses classes filles.

➔ **Par défaut, tous les éléments d'une classe sont private.**

Droits d'accès (2/2)

- Exemple:

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

class Point {

private:
    int coord_x;
    int coord_y;

public:
    void SetCoord(const int & x, const int & y) {
    ...
    }
```

```
void Afficher() {
...
}

};

int main()
{
    Point p1;
    //Création d'un objet de type Point

    p1.SetCoord(4,8);
    p1.Afficher();

    return 0;
}
```

L'encapsulation (1/2)

- **L'encapsulation** est un mécanisme qui interdit d'accéder à certaines données depuis l'extérieur de la classe.
 - ➔ **tous les attributs d'une classe doivent toujours être privés !**
- Un utilisateur de la classe ne pourra pas accéder à tous les éléments de celle-ci.
 - ➔ Il sera obligé d'utiliser certaines fonctions membres de la classe (celles qui sont publiques).
- L'avantage de cette restriction est qu'il empêche par exemple un utilisateur de la classe de mettre les données dans un état incohérent.

L'encapsulation (2/2)

- Vue de l'extérieur, la classe apparaît comme une boîte noire, qui a un certain comportement à laquelle on ne peut accéder que par les méthodes publiques.
- Cette notion est extrêmement **puissante** et permet *d'éviter de nombreux effets de bord*.



- En C++, on peut utiliser les structures (**struct**) pour créer des classes.
 - ➔ La seule différence avec le mot-clé **class** est que, par défaut, les méthodes et attributs sont publics au lieu de privés.

Séparation des méthodes (1/4)

- Il s'agit de séparer les *méthodes* en **prototypes** et **définitions** dans deux fichiers différents pour avoir un code plus modulaire.
- Il faut séparer le **main()** (qui se trouve dans `main.cpp`) des classes.
- Pour chaque classe, on crée:
 - ✓ **un header** (fichier `*.h`) qui contiendra les attributs et les prototypes de la classe ;
 - ✓ **un fichier source** (fichier `*.cpp`) qui contiendra la définition des méthodes et leur implémentation.

Séparation des méthodes (2/4)

Point.h

```
class Point {  
  
    private:  
        int coord_x;  
        int coord_y;  
  
    public:  
        void SetCoord(const int & x, const int & y);  
        void Afficher();  
  
};
```

- Le fichier **Point.h** contient la déclaration de la classe avec les attributs et les prototypes des méthodes.

Séparation des méthodes (3/4)

Point.cpp

```
#include "Point.h"

void Point::SetCoord(const int & x, const int & y) {
    coord_x = x;
    coord_y = y;
}

void Point::Afficher() {

    cout << "x = " << coord_x << " y = " << coord_y << endl;
}
```

- Le fichier **Point.cpp** implémente le code des méthodes.
- On utilise l'opérateur **::** de **réalisation de domaine** pour préfixer le nom de la classe au nom de la méthode.

Séparation des méthodes (4/4)

main.cpp

```
#include "Point.h"

int main()
{
    Point p1;
    //Création d'un objet de type Point

    p1.SetCoord(4,8);
    p1.Afficher();

    return 0;
}
```

- Il faut inclure **#include "Point.h"** pour pouvoir créer des objets de type Point.

Le constructeur (1/8)

- Lorsqu'on crée une nouvelle instance d'une classe, les données membres de cette instance ne sont pas initialisées par défaut.
- **Un constructeur** est une fonction membre qui sera appelée au moment de la création d'une nouvelle instance d'une classe.
- Il peut y avoir plusieurs constructeurs d'une classe avec différents paramètres qui serviront à initialiser notre classe.
- Le rôle principal du constructeur est d'initialiser les attributs.

Le constructeur (2/8)

- Exemple:

```
Point.h

class Point {

private:
    int coord_x;
    int coord_y;

public:
    Point();    //Constructeur
    void SetCoord(const int & x, const int & y);
    void Afficher();

};
```

- Il faut que la méthode ait le même nom que la classe.
- La méthode ne doit rien renvoyer, pas même void !
➔ C'est une méthode sans aucun type de retour.

Point.cpp

```
#include "Point.h "
```

```
Point::Point(){  
    coord_x = 0;  
    coord_y = 0;  
}
```

```
void Point::SetCoord(const int & x, const int & y) {  
    coord_x = x;  
    coord_y = y;  
}
```

```
void Point::Afficher() {  
  
    cout << "x = " << coord_x << " y = " << coord_y << endl;  
}  
  
};
```


Le constructeur (3/8)

❑ La liste d'initialisation:

- Le C++ permet d'initialiser les attributs de la classe d'une autre manière appelée **liste d'initialisation**.

```
Point::Point() : x(0), y(0) {  
  
    //Rien à mettre dans le corps du constructeur !  
  
}
```

Le constructeur (4/8)

❑ Surcharge du constructeur:

- Surcharger un constructeur permet de créer un objet de **plusieurs façons différentes**.

Point.h

```
class Point {  
  
    private:  
        int coord_x;  
        int coord_y;  
  
    public:  
        Point(); //Constructeur  
        void SetCoord(const int & x, const int & y);  
        void Afficher();  
  
};
```

Point() est un **constructeur par défaut**.

➔ c'est le constructeur sans argument.

➔ Si aucun constructeur n'est présent explicitement dans la classe, il est généré par le compilateur.

Le constructeur (5/8)

- Supposons que l'on souhaite créer un point avec des coordonnées précises, on utilise la technique de **surcharge de constructeur**.
- On crée un deuxième constructeur qui prendra en paramètre les coordonnées exactes.
- On **rajoute** dans **Point.h** ce prototype:

```
Point(int abscisse, int ordonne);
```

- L'implémentation dans **Point.cpp** sera :

```
Point::Point(int abscisse, int ordonne) : x(0), y(0) {  
  
}
```

Le constructeur (6/8)

- Un exemple montrant la création d'un point ayant des coordonnées précises dès le début dans la fonction **main()**:

```
Point p1, p2(5, 7);
```

Le constructeur (7/8)

❑ Le constructeur de copie:

- **Le constructeur de copie** est un constructeur qui prend en paramètre un objet de la même classe et qui construit une copie de cet objet.
- Si le constructeur de copie n'est pas déclaré, le compilateur le crée par défaut qui fera appel au constructeur de copie de chacun des attributs qui composent l'objet.
- Par exemple si l'on souhaite que **p1** soit une copie conforme de **p2**:

```
Point p2(5, 7);    //création de p2 en utilisant un constructeur normal
```

```
Point p1(p2);      //création de p1 en copiant tous les attributs de p2
```

Le constructeur (8/8)

- Le constructeur de copie est un constructeur à **un argument** dont le type est celui de la classe.
- En général, l'argument est passé par **référence constante**:
 - ✓ le déclarer dans la classe de la manière suivante:

```
Point(Point const& autre);
```

- ✓ et définir son implémentation:

```
Point::Point(Point const& autre): x(autre.x), y(autre.y) {  
  
}
```

➔ accès direct aux attributs de l'objet à copier (autre) dans la liste d'initialisation.

Le destructeur (1/2)

- Lorsqu'un objet devient inutile, il doit être détruit.
 - ➔ Cette opération est réalisée par **le destructeur**.
- Caractéristiques :
 - ✓ Il est unique (un seul destructeur pour une même classe).
 - ✓ Un destructeur est une méthode qui commence par un tilde (~) suivi du nom de la classe.
 - ✓ Il n'a pas de paramètre. Il ne retourne aucun résultat, pas même void (il ne peut pas être surchargé).
 - ✓ Il doit restituer la place mémoire allouée dynamiquement par l'objet.

Le destructeur (2/2)

- Dans **Point.h**, le prototype du destructeur sera:

```
~Point();
```

- Dans **Point.cpp**, l'implémentation sera :

```
Point::~~Point()
{
    /* Rien à mettre ici car on ne fait pas d'allocation dynamique dans la classe Point.
    Le destructeur est donc inutile dans ce cas.
    En temps normal, un destructeur fait souvent des delete et quelques autres vérifications si
    nécessaire avant la destruction de l'objet. */
}
```


Les méthodes constantes

- **Les méthodes constantes** sont des méthodes de **lecture seule**.
- Elles possèdent le mot-clé **const** à la fin de leur prototype et de leur déclaration.
- Une méthode constante ne modifie la valeur d'aucun de ses attributs, comme les méthodes d'affichage et de vérification.

```
void maMethode(int parametre) const;           //Prototype de la méthode (dans le .h)
```

```
void MaClasse::maMethode(int parametre) const {   //Déclaration de la méthode (dans le .cpp)  
...  
}
```

Association des classes

- La programmation orientée objet devient vraiment intéressante et puissante lorsqu'on se met à combiner plusieurs objets entre eux.
- Un programme objet est un programme constitué d'une multitude d'objets différents.
- Si une classe A veut utiliser la classe B, elle doit ajouter dans **A.h**:

```
#include "B.h"
```

La surcharge d'opérateurs

Introduction (1/3)

- Le C++ permet de surcharger les opérateurs, c'est-à-dire de créer des méthodes pour modifier le comportement des symboles $+$, $-$, $*$, $/$, \geq , etc.
 - ➔ **La surcharge d'opérateurs** est une technique qui permet de réaliser des opérations mathématiques intelligentes *entre les objets* lors de l'utilisation dans votre code des symboles tels que $+$, $-$, $*$, $==$, $<$, etc.
- La surcharge d'opérateurs permet d'écrire directement **$U+V$** , par exemple, lorsqu'on veut additionner deux instances U et V d'une même classe A.

Introduction (2/3)

- Pour surcharger un opérateur, on doit donner un nom précis à sa méthode (**operator+** pour le symbole + par exemple).
- Ces méthodes doivent prendre des paramètres et renvoyer parfois des valeurs d'un certain type précis.
 - ➔ Cela dépend de l'opérateur que l'on surcharge.
- Il ne faut pas abuser de la surcharge d'opérateur car elle peut avoir l'effet inverse du but recherché, qui est de rendre la lecture du code plus simple.

Introduction (3/3)

- La signature générique est la suivante:

- `type_retour operator @(Arg);` // pour les méthodes non const
- `type_retour operator @(Arg) const;` // pour les méthodes const
- `type_retour operator @(Arg1, Arg2) const;` // pour les fonctions

où @ est l'opérateur à surcharger: + - * / ...

Opérateurs surchargeables (1/4)

- ❑ **Les opérateurs unaires:** ce sont des opérateurs qui s'appliquent sans argument supplémentaire

| Opérateur | Définition |
|-----------|-----------------------------------|
| + | opérateur signe + |
| - | opérateur négation |
| * | opérateur dérérérencement |
| & | opérateur adresse |
| -> | opérateur indirection |
| ~ | opérateur négation binaire |
| ++ | opérateur post/pré-incrémentation |
| -- | opérateur post/pré-décrément |

Opérateurs surchargeables (2/4)

- ❑ **Les opérateurs binaires:** ce sont des opérateurs qui s'appliquent moyennant un argument supplémentaire

| Opérateur | Définition |
|-----------|--------------------------------------|
| = | opérateur assignation |
| + | opérateur addition |
| += | opérateur addition/assignation |
| - | opérateur soustraction |
| -= | opérateur soustraction/assignation |
| * | opérateur multiplication |
| *= | opérateur multiplication/assignation |
| / | opérateur division |

Opérateurs surchargeables (3/4)

| Opérateur | Définition |
|-----------|-----------------------------------|
| /= | opérateur division/assignation |
| % | opérateur modulo |
| %= | opérateur modulo/assignation |
| ^ | opérateur ou exclusif |
| ^= | opérateur ou exclusif/assignation |
| & | opérateur et binaire |
| &= | opérateur et binaire/assignation |
| | opérateur ou binaire |
| = | opérateur ou binaire/assignation |
| < | opérateur plus petit que |
| | opérateur ou logique |
| [] | opérateur indexation |

Opérateurs surchargeables (4/4)

| Opérateur | Définition |
|-----------|--|
| <= | opérateur plus petit ou égal |
| > | opérateur plus grand que |
| >= | opérateur plus grand ou égal |
| << | opérateur de décalage à gauche |
| <<= | opérateur de décalage à gauche/assignation |
| >> | opérateur de décalage à droite |
| >>= | opérateur de décalage à droite/assignation |
| == | opérateur d'égalité |
| != | opérateur d'inégalité |
| && | opérateur et logique |
| , | opérateur enchaînement |
| () | opérateur fonction |

Exemple de l'opérateur de comparaison == (1/8)

Duree.h

```
#ifndef DEF_DUREE
#define DEF_DUREE

class Duree
{
    public:
        Duree(int heures = 0, int minutes = 0, int secondes = 0);

    private:
        int m_heures;
        int m_minutes;
        int m_secondes;
};

#endif
```

➤ Exemple:

- Création d'une classe pour stocker une durée.
- Instanciation de deux objets de type Duree qu'on veut vérifier s'ils sont égaux.

Exemple de l'opérateur de comparaison == (2/8)

Duree.cpp

```
#include "Duree.h"
```

```
Duree::Duree(int heures, int minutes, int secondes) : m_heures(heures), m_minutes(minutes),  
m_secondes(secondes) {  
  
}
```

main.cpp

```
int main()  
{  
    Duree duree1(0, 10, 28), duree2(0, 15, 2);  
  
    return 0;  
}
```

Exemple de l'opérateur de comparaison == (3/8)

- Pour être capable d'utiliser le symbole « == » entre deux objets, il faut créer une fonction ayant précisément pour nom **operator==** et dotée du prototype:

```
bool operator==(Objet const& a, Objet const& b);
```

➔ La fonction reçoit deux références sur les objets (**références constantes**, qu'on ne peut donc pas modifier) à comparer et va renvoyer un booléen indiquant si les deux objets sont identiques ou non.

- On doit rajouter cette fonction dans **Duree.h**:

```
bool operator==(Duree const& a, Duree const& b);
```

Exemple de l'opérateur de comparaison == (4/8)

- Le but de la création de la fonction **operator==** est de comparer deux objets de type **Duree**:

```
if (duree1 == duree2) {  
    std::cout << "Les deux durees sont egales !" << std::endl;  
}
```

Le compilateur traduit cela par



```
if (operator==(duree1, duree2)) {  
    std::cout << "Les deux durees sont egales !" << std::endl;  
}
```

Exemple de l'opérateur de comparaison == (5/8)

- L'implémentation de l'opérateur == :

Dans notre exemple, deux Duree sont égales si elles contiennent le même nombre d'heures, de minutes et de secondes.

```
bool operator==(Duree const& a, Duree const& b)
{
    if (a.m_heures == b.m_heures && a.m_minutes == b.m_minutes && a.m_secondes == b.m_secondes)
        return true;
    else
        return false;
}
```

Exemple de l'opérateur de comparaison == (6/8)

- Le problème dans l'implémentation précédente est qu'il faudrait lire les attributs des objets **a** et **b** qui sont privés et inaccessibles depuis l'extérieur de la classe.
- Il existe trois solutions à ce problème:
 - ✓ créer des accesseurs (avec les méthodes `getHeures()`, `getMinutes()`, ...).
 - ➔ Cela marche bien mais c'est un peu ennuyeux à écrire.
 - ✓ utiliser le concept d'amitié.
 - ✓ Ou bien utiliser la technique suivante:

Exemple de l'opérateur de comparaison == (7/8)

- Le problème est que l'**opérateur ==** est situé en-dehors de la classe (ce n'est pas une méthode) et ne peut donc accéder aux attributs privés.
- Donc, on crée une méthode dans la classe qui fera la comparaison et on demande à l'opérateur d'appeler cette fonction.

```
bool Duree::estEgal(Duree const& b) const
{
    if (m_heures == b.m_heures && m_minutes == b.m_minutes && m_secondes == b.m_secondes)
        return true;
    else
        return false;
}
```

Exemple de l'opérateur de comparaison == (8/8)

- Utiliser cette méthode dans l'opérateur de comparaison:

```
bool operator==(Duree const& a, Duree const& b)
{
    return a.estEgal(b);
}
```

- Dans la fonction main.cpp:

```
int main()
{
    Duree duree1(1, 16, 41), duree2(1, 16, 41);

    if (duree1 == duree2)
        cout << "Les durees sont identiques";
    else
        cout << "Les durees sont differentes";

    return 0;
}
```

Exemple de l'opérateur != (1/2)

- Cet opérateur permet de tester si deux objets sont différents.
- Pour tester si deux objets sont différents, il suffit de tester s'ils ne sont pas égaux !
- On utilise l'opérateur == défini dans l'exemple précédent.

```
bool operator!=(Duree const& a, Duree const& b)
{
    if(a == b)
        return false;           //ils ne sont pas différents
    else
        return true;             //ils sont différents
}
```

ou

```
bool operator!=(Duree const& a, Duree const& b)
{
    return !(a==b);
}
```

Exemple de l'opérateur != (2/2)

- Dans la fonction main.cpp:

```
int main()
{
    Duree duree1(1, 16, 41), duree2(1, 16, 41);

    if (duree1 != duree2)
        cout << "Les durees sont differentes";
    else
        cout << "Les durees sont identiques";

    return 0;
}
```

Exercice

- Ecrire la surcharge de l'opérateur < qui permet de vérifier si un objet est **strictement inférieur à** un autre objet.

Exemple de l'opérateur < (1/3)

- L'opérateur < permet de vérifier si un objet est **strictement inférieur à** un autre objet.
- On crée une méthode publique dans la classe et on l'appelle dans le corps de l'opérateur afin de résoudre le problème d'accès aux attributs des objets.

```
bool operator<(Duree const &a, Duree const& b)
{
    return a.estPlusPetitQue(b);
}
```

Exemple de l'opérateur < (2/3)

- La méthode **estPlusPetitQue()** de la classe `Duree` :

```
bool Duree::estPlusPetitQue(Duree const& b) const
{
    if (m_heures < b.m_heures)
        return true;
    else if (m_heures == b.m_heures && m_minutes < b.m_minutes)
        return true;
    else if (m_heures == b.m_heures && m_minutes == b.m_minutes && m_secondes < b.m_secondes)
        return true;
    else
        return false;
}
```

Exemple de l'opérateur < (3/3)

- Dans la fonction main.cpp:

```
int main()
{
    Duree duree1(1, 16, 41), duree2(1, 16, 41);

    if (duree1 < duree2)
        cout << "La premiere duree est plus petite";
    else
        cout << "La premiere duree n'est pas plus petite";

    return 0;
}
```


Exemple de l'opérateur arithmétique + (1/2)

- Le prototype de l'opérateur d'addition:

```
Duree operator+(Duree const& a, Duree const& b);
```

➔ Cette fonction doit être définie *à l'extérieur* de la classe. Ce n'est pas une méthode.

- Cet opérateur doit créer un nouvel objet de type `Duree` dont la valeur des attributs est la somme des valeurs des attributs de `a` et `b`.
- On crée une méthode publique dans la classe et on l'appelle dans le corps de l'opérateur afin de résoudre le problème d'accès aux attributs des objets.

Exemple de l'opérateur arithmétique + (2/2)

```
Duree operator+(Duree const& a, Duree const& b)
{
    Duree resultat;
    resultat = a.calculerAddition(b);

    return resultat;
}
```

```
int main()
{
    Duree resultat, duree1, duree2;

    resultat = duree1 + duree2;

    return 0;
}
```

Exemple de l'opérateur de flux << (1/7)

- Les opérateurs de flux permettent l'injection dans les flux d'entrée-sortie.

```
cout << "Bonjour";
```

```
cin >> variable;
```



Ce code revient à
faire appel aux fonctions:

```
operator<<(cout, "Bonjour");
```

```
operator>>(cin, variable);
```

- Problème:** `cout` ne connaît pas la classe `Duree` et il ne possède donc pas de fonction surchargée pour les objets de ce type.
- La fonction `operator<<` utilise un objet de la classe `ostream` dont `cout` est une instance.

Exemple de l'opérateur de flux << (2/7)

- Donc, on doit inclure la classe `<iostream>`.
- Lorsqu'on inclut `<iostream>`, un objet `cout` est automatiquement déclaré:

```
ostream cout;
```

➔ On ne peut pas modifier la classe `ostream` mais ce n'est pas important puisque tout ce qu'on veut faire c'est écrire une fonction qui reçoit un de ces objets en argument.

- La fonction `operator<<`:

```
ostream& operator<<( ostream &flux, Duree const& duree )  
{  
    //Affichage des attributs  
    return flux;  
}
```

Exemple de l'opérateur de flux << (3/7)

- Le type de retour de la fonction **operator<<** est une **référence** et non un objet.
- Le premier paramètre (référence sur un objet de type ostream) qui sera automatiquement passé est en fait l'objet **cout** (que l'on appelle ici **flux** dans la fonction pour éviter les conflits de nom).
- Le second paramètre est une référence constante vers l'objet de type **Duree** qu'on veut l'afficher en utilisant l'**opérateur <<**.

```
ostream& operator<<( ostream &flux, Duree const& duree )  
{  
    //Affichage des attributs  
    return flux;  
}
```

Exemple de l'opérateur de flux << (3/7)

- Le type de retour de la fonction **operator<<** est une **référence** et non un objet.
- Le premier paramètre (référence sur un objet de type ostream) qui sera automatiquement passé est en fait l'objet **cout** (que l'on appelle ici **flux** dans la fonction pour éviter les conflits de nom).
- Le second paramètre est une référence constante vers l'objet de type **Duree** qu'on veut l'afficher en utilisant l'**opérateur <<**.
- La fonction doit récupérer les attributs qui l'intéresse dans l'objet et les envoyer à l'objet flux (qui est cout).
- Ensuite, elle renvoie une référence sur cet objet:

```
cout << duree1 << duree2;
```

Exemple de l'opérateur de flux << (4/7)



Les types de retour des opérateurs ne sont pas toujours facile à déduire, mieux vaut les apprendre par cœur.

- On définit une méthode dans la classe `Duree` pour pouvoir accéder aux attributs et les afficher.

```
ostream& operator<<( ostream &flux, Duree const& duree)
{
    duree.afficher(flux) ;
    return flux;
}
```

Exemple de l'opérateur de flux << (5/7)

- Dans **Duree.h**:

```
void afficher(std::ostream &flux) const;
```

- Dans **Duree.cpp**:

```
void Duree::afficher(ostream &flux) const  
{  
    flux << m_heures << "h" << m_minutes << "m" << m_secondes << "s";  
}
```

➔ La méthode prend en paramètre la référence vers l'objet flux pour pouvoir lui envoyer les valeurs qu'on veut les afficher.

Exemple de l'opérateur de flux << (6/7)

- Dans la fonction main.cpp:

```
int main()
{
    Duree duree1(22, 03, 14), duree2(21, 15, 33);

    cout << duree1 << " et " << duree2 << endl;

    return 0;
}
```

Exemple de l'opérateur de flux << (7/7)

- On peut même combiner les opérateurs dans une seule expression: Faire une addition et afficher le résultat directement.

```
int main()
{
    Duree duree1(2, 25, 28), duree2(0, 16, 33);

    cout << duree1 + duree2 << endl;

    return 0;
}
```

Comme pour les int, double, etc, les objets deviennent réellement simples à utiliser.