

Où en est-on ?

Dans les vidéos précédentes, nous avons vu comment déclarer des classes et des objets.

On peut par exemple déclarer une instance de la classe `Rectangle` :

```
Rectangle rect;
```

Une fois que l'on a fait cette déclaration, comment faire pour *initialiser les attributs* de `rect` ?

```
class Rectangle {
public:
    double surface() const;
    double getHauteur() const;
    double getLargeur() const;
    void setHauteur(double h);
    void setLargeur(double l);
private:
    double hauteur;
    double largeur;
};
```

Initialisation des attributs

Première solution : **affecter individuellement une valeur** à chaque attribut

```
Rectangle rect;
double lu;
cout << "Quelle hauteur? "; cin >> lu;
rect.setHauteur(lu);
cout << "Quelle largeur? "; cin >> lu;
rect.setLargeur(lu);
```

Ceci est une *mauvaise solution* dans le cas général :

- ▶ elle implique que tous les attributs fassent partie de l'interface (`public`) ou soient assortis d'un manipulateur
 - ☞ casse l'encapsulation
- ▶ oblige le programmeur-utilisateur de la classe à initialiser explicitement tous les attributs
 - ☞ risque d'oubli

Initialisation des attributs (2)

Deuxième solution : définir une **méthode dédiée à l'initialisation** des attributs

```
class Rectangle {
public:
    void init(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    ...
private:
    double hauteur;
    double largeur;
};
```

Pour faire ces initialisations, il existe en C++ des méthodes particulières appelées **constructeurs**.

Les constructeurs

Un constructeur est une méthode :

- ▶ invoquée *automatiquement* lors de la déclaration d'un objet
- ▶ chargée d'effectuer toutes les opérations requises en « début de vie » de l'objet (dont *l'initialisation des attributs*)

Syntaxe de base :

```
NomClasse(liste_paramètres)
{
    /* initialisation des attributs
       en utilisant liste_paramètres */
}
```

Exemple (à améliorer par la suite) :

```
Rectangle(double h, double L)
{
    hauteur = h;
    largeur = L;
}
```

Les constructeurs (2)

Les constructeurs sont des méthodes presque comme les autres. Les différences sont :

- ▶ *pas* de type de retour (pas même `void`)
- ▶ *même nom* que la classe
- ▶ invoqués *automatiquement* à chaque fois qu'une instance est créée.

```
Rectangle(double h, double L)
{
    hauteur = h;
    largeur = L;
}
```

Comme les autres méthodes :

- ▶ les constructeurs peuvent être surchargés
- ▶ on peut donner des valeurs par défaut à leurs paramètres

(exemples dans la suite)

Une classe peut donc avoir **plusieurs constructeurs**, pour peu que leur liste de paramètres soit différente.

Notre programme (1/3)

```
class Rectangle {
public:
    Rectangle(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    double surface() const
    { return hauteur * largeur; }
    // accesseurs/modificateurs si nécessaire
    // ...
private:
    double hauteur;
    double largeur;
};
```

Initialisation par constructeur

La *déclaration avec initialisation* d'un objet se fait comme pour une variable ordinaire.

Syntaxe :

```
NomClasse instance(valarg1, ..., valargN);
```

où *valarg1*, ..., *valargN* sont les valeurs des arguments passés au constructeur.

Exemple :

```
Rectangle r1(18.0, 5.3); // invocation du constructeur à 2 paramètres
```

Notre programme (2/3)

```
// ...
class Rectangle {
public:
    Rectangle(double h, double L)
    {
        hauteur = h;
        largeur = L;
    }
    // ...
private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1(3.0, 4.0);
    // ...
}
```

Construction des attributs

Que se passe-t-il si les attributs sont eux-mêmes des objets ?

Exemple :
(à améliorer dans
une prochaine leçon)

```
class RectangleCouleur {
private:
    Rectangle rectangle;
    Couleur couleur;
    //...
};
```

mauvaise solution :

```
RectangleCouleur(double h, double L, Couleur c)
{
    rectangle = Rectangle(h, L);
    couleur = c;
}
```

☞ Il faut initialiser *directement* les attributs en faisant appel à *leurs propres* constructeurs !

Appel aux constructeurs des attributs

Un constructeur devrait normalement contenir une section d'appel aux constructeurs des attributs....

...ainsi que l'initialisation des attributs de type de base.

C'est ce qu'on appelle la « **liste d'initialisation** » du constructeur.

Syntaxe générale :

```
NomClasse(liste_paramètres)
// liste d'initialisation
: attribut1(...), // appel au constructeur de attribut1
...
    attributN(...) // appel au constructeur de attributN
{ // autres opérations }
```

Appel aux constructeurs des attributs

Exemple :

```
class Rectangle {
    Rectangle(double h, double L);
    // ...
};

class RectangleCouleur {

    RectangleCouleur(double h, double L, Couleur c)
    : rectangle(h, L), couleur(c)
    {}

private:
    Rectangle rectangle;
    Couleur couleur;
};
```

Liste d'initialisation (2)

Cette section introduite par « : » est optionnelle mais **recommandée**.

Par ailleurs :

- ▶ les attributs non-initialisés dans cette section
 - ▶ prennent une valeur par défaut si ce sont des objets ;
 - ▶ restent indéfinis s'ils sont de type de base ;
- ▶ les attributs initialisés dans cette section peuvent (bien sûr) être changés dans le corps du constructeur.

Exemple :

```
Rectangle(double h, double L)
: hauteur(h) //initialisation
{
    // largeur a une valeur indéfinie jusqu'ici
    largeur = 2.0 * L + h; // par exemple...
    // la valeur de largeur est définie à partir d'ici
}
```

Notre programme (3/3)

```
// ...
class Rectangle {
public:
    Rectangle(double h, double L)
        : hauteur(h), largeur(L)
    {}
    double surface() const
    { return hauteur * largeur; }
    // accesseurs/modificateurs
    // ...
private:
    double hauteur;
    double largeur;
};

int main()
{
    Rectangle rect1(3.0, 4.0);
    // ...
}
```

Constructeur par défaut

Le constructeur par défaut est un constructeur qui **n'a pas de paramètre** ou dont **tous** les paramètres ont des **valeurs par défaut**.

Exemple :

```
// Le constructeur par défaut
Rectangle() : hauteur(1.0), largeur(2.0)
{}

// 2ème constructeur
Rectangle(double c) : hauteur(c), largeur(2.0*c)
{}

// 3ème constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L)
{}
```

Constructeur par défaut : autre exemple

Autre façon de faire : regrouper les 2 premiers constructeurs en utilisant les valeurs par défaut des paramètres :

```
// DEUX constructeurs dont le constructeur par défaut
Rectangle(double c = 1.0) : hauteur(c), largeur(2.0*c)
{}

// 3ème constructeur
Rectangle(double h, double L) : hauteur(h), largeur(L)
{}
```


Constructeur par défaut par défaut

Si aucun constructeur n'est spécifié, le compilateur *génère automatiquement* une **version minimale du constructeur par défaut** qui :

- ▶ appelle le constructeur par défaut des attributs objets.
- ▶ laisse non initialisés les attributs de type de base.

Dès qu'**au moins un constructeur a été spécifié**, ce constructeur par défaut par défaut *n'est plus fourni*.

Si donc on spécifie *un* constructeur sans spécifier de constructeur par défaut, on ne peut plus construire d'objet de cette classe sans les initialiser (ce qui est voulu !) puisqu'il n'y a plus de constructeur par défaut.

 ...mais on peut le rajouter si on veut (voir plus loin).

A :

```
class Rectangle {
private:
    double h; double L;
    // suite ...
};
```

B :

```
class Rectangle {
private:
    double h; double L;
public:
    Rectangle()
        : h(0.0), L(0.0)
    {}
    // suite ...
};
```

C :

```
class Rectangle {
private:
    double h; double L;
public:
    Rectangle(double h=0.0,
               double L=0.0)
        : h(h), L(L)
    {}
    // suite ...
};
```

D :

```
class Rectangle {
private:
    double h; double L;
public:
    Rectangle(double h,
               double L)
        : h(h), L(L)
    {}
    // suite ...
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);
(A)	constructeur par défaut par défaut	<input type="text" value="?"/> <input data-bbox="555 236 649 274" type="text" value="?"/>	Illicite !

```
class Rectangle {
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);
(A)	constructeur par défaut par défaut	<input type="text" value="?"/> <input data-bbox="1675 236 1769 274" type="text" value="?"/>	Illicite !
(B)	constructeur par défaut explicitement déclaré	<input type="text" value="0"/> <input data-bbox="1675 316 1769 354" type="text" value="0"/>	Illicite !

```
class Rectangle {
    Rectangle()
        : h(0.0), L(0.0)
    {}
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);
(A)	constructeur par défaut par défaut	<input type="text" value="?"/> <input data-bbox="555 1034 649 1072" type="text" value="?"/>	Illicite !
(B)	constructeur par défaut explicitement déclaré	<input type="text" value="0"/> <input data-bbox="555 1114 649 1152" type="text" value="0"/>	Illicite !
(C)	un des trois constructeurs est par défaut	<input type="text" value="0"/> <input data-bbox="555 1193 649 1232" type="text" value="0"/>	<input type="text" value="1"/> <input data-bbox="790 1193 884 1232" type="text" value="2"/>

```
class Rectangle {
    Rectangle(double h=0.0,
               double L=0.0)
        : h(h), L(L)
    {}
};
```

Constructeur par défaut : exemples

	constructeur par défaut	Rectangle r1;	Rectangle r2(1.0, 2.0);
(A)	constructeur par défaut par défaut	<input type="text" value="?"/> <input data-bbox="1675 1034 1769 1072" type="text" value="?"/>	Illicite !
(B)	constructeur par défaut explicitement déclaré	<input type="text" value="0"/> <input data-bbox="1675 1114 1769 1152" type="text" value="0"/>	Illicite !
(C)	un des trois constructeurs est par défaut	<input type="text" value="0"/> <input data-bbox="1675 1193 1769 1232" type="text" value="0"/>	<input type="text" value="1"/> <input data-bbox="1910 1193 2004 1232" type="text" value="2"/>
(D)	pas de constructeur par défaut	Illicite !	<input type="text" value="1"/> <input data-bbox="1910 1257 2004 1295" type="text" value="2"/>

```
class Rectangle {
    Rectangle(double h,
               double L)
        : h(h), L(L)
    {}
};
```

C++11 Remettre le constructeur par défaut par défaut

Dès qu'au moins un constructeur a été spécifié, ce constructeur par défaut par défaut n'est plus fourni.

C'est très bien si c'est vraiment ce que l'on veut

(c'est-à-dire forcer les utilisateurs de la classe à utiliser nos constructeurs).

Mais si l'on veut quand même avoir le constructeur par défaut par défaut, on peut le re-demander en écrivant dans la définition de la classe :

```
NomClasse() = default;
```

Exemple (modification du cas (D) précédent) :

```
class Rectangle {
public:
    Rectangle() = default; // mais peu pertinent ici
    Rectangle(double h, double L) : h(h), L(L) {}
    // suite ...
};
```

C++11 méthodes default et delete

Ce que l'on a fait précédemment (= default) pour le constructeur par défaut se généralise :

- ▶ à toute méthode (pour laquelle cela est pertinent)
- ▶ à la suppression de méthode, via la syntaxe « = delete »

Exemple :

```
class Demo {
public:
    double pas_d_int(double x) { ... }
    double pas_d_int(int) = delete;
};
```

Un autre exemple sera donné dans la séquence suivante.

C++11 Appel aux autres constructeurs

C++11 autorise les constructeurs d'une classe à appeler n'importe quel autre constructeur de cette même classe

Exemple :

```
class Rectangle {
private:
    double hauteur; double largeur;
public:
    Rectangle(double h, double L) : hauteur(h), largeur(L) {}

    Rectangle() : Rectangle(0.0, 0.0) {}
    // bien mieux que le =default précédent

    // suite ...
};
```

C++11 Initialisation par défaut des attributs

C++11 permet de donner directement une valeur par défaut aux attributs.

Si le constructeur appelé ne modifie pas la valeur de cet attribut, ce dernier aura alors la valeur indiquée.

Exemple :

```
class Rectangle {
    // ...
private:
    double hauteur = 0.0;
    double largeur = 0.0;
    // ...
};
```

Conseil : préférez l'utilisation des constructeurs.

Constructeur de copie

C++ offre un moyen de créer la **copie** d'une instance :
le *constructeur de copie*

```
Rectangle r1(12.3, 24.5);  
Rectangle r2(r1);
```

`r1` et `r2` sont deux *instances* **distinctes**
mais ayant des mêmes valeurs pour leurs attributs
(au moins juste après la copie).

Autre exemple de copie (invocation du constructeur de copie) :

```
double f(Rectangle r);  
...  
x = f(r1);
```

Constructeur de copie (2)

Le constructeur de copie permet d'initialiser une instance
en *copiant* les attributs d'une *autre instance* du même type.

Syntaxe :

```
NomClasse(NomClasse const& autre) { ... }
```

Exemple :

```
Rectangle(Rectangle const& autre)  
: hauteur(autre.hauteur), largeur(autre.largeur)  
{}
```

Constructeur de copie (3)

- ▶ Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini
(constructeur de copie par défaut)
- ▶ Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)
☞ **copie de surface**

Tout se passe comme si le constructeur précédent avait été écrit :

```
Rectangle(Rectangle const& autre)  
: hauteur(autre.hauteur), largeur(autre.largeur)  
{}
```

mais il n'est *pas nécessaire* de l'écrire !

Constructeur de copie (3)

- ▶ Un constructeur de copie est *automatiquement généré* par le compilateur s'il n'est pas explicitement défini
(constructeur de copie par défaut)
- ▶ Ce constructeur opère une initialisation *membre à membre* des attributs (si l'attribut est un objet le constructeur de cet objet est invoqué)
☞ **copie de surface**
- ▶ Cette copie de surface suffit dans la plupart des cas.

Cependant, il est parfois nécessaire de redéfinir le constructeur de copie, en particulier lorsque certains attributs sont des *pointeurs* (des exemples arriveront plus tard dans le cours) !

Mais je vous donne déjà la **règle d'or** : si vous touchez à l'un des trois parmi « constructeur de copie », « destructeur » et « opérateur d'affectation » (`operator=`), alors pensez aux deux autres.

C++11 Suppression du constructeur de copie

C++11 Par ailleurs, si l'on souhaite *interdire* la copie, il suffit de **supprimer** le constructeur de copie par défaut avec la commande « `= delete` » vue dans la séquence précédente.

Exemple :

```
class PasCopiable {  
    /* ... */  
    PasCopiable(PasCopiable const&) = delete;  
};
```

Destructeur

SI l'initialisation des attributs d'une instance implique la mobilisation de ressources : *fichiers, périphériques, portions de mémoire (pointeurs), etc.*

☞ il est alors important de **libérer ces ressources** après usage !

Comme pour l'initialisation, l'*invocation explicite* de méthodes de libération n'est pas satisfaisante (fastidieuse, source d'erreur, affaiblissement de l'encapsulation).

☞ C++ offre une méthode appelée *destructeur* invoquée automatiquement en fin de vie de l'instance.

Destructeur (2)

La syntaxe de déclaration d'un destructeur pour une classe `NomClasse` est :

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

- ▶ Le destructeur d'une classe est une méthode *sans paramètre*
☞ **pas de surcharge possible**
- ▶ Son nom est celui de la classe, précédé du signe `~` (tilda).
- ▶ Si le destructeur n'est pas défini explicitement par le programmeur, le compilateur en génère automatiquement une version minimale.

Exemple

Supposons que l'on souhaite compter le nombre d'instances d'une classe actives à un moment donné dans un programme.

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2
        // ...
    }
    // compteur = 1
    return 0;
} // compteur = 0
```

Exemple

Supposons que l'on souhaite compter le nombre d'instances d'une classe actives à un moment donné dans un programme.

Utilisons comme compteur une variable globale de type entier :

- ▶ le constructeur incrémente le compteur

```
long compteur(0); /* Hmm... On y reviendra
                  dans une autre séquence vidéo ! */

class Rectangle {
    //...
    Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
        ++compteur; }
    // ...
}
```

Exemple

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2
        // ...
    }
    // compteur = 2
    return 0;
} // compteur = 2
```

👉 on est obligé ici de **définir explicitement le destructeur**

Exemple

- ▶ le constructeur incrémente le compteur
- ▶ le destructeur le décrémente

```
long compteur(0); /* Hmm... On y reviendra
                  dans une autre séquence vidéo ! */

class Rectangle {
    //...
    Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
        ++compteur; }
    ~Rectangle() { --compteur; } // destructeur
    // ...
}
```

Exemple

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2
        // ...
    }
    // compteur = 1
    return 0;
} // compteur = 0
```

Exemple

Que se passe-t-il si l'on souhaite utiliser la copie d'objet ?

```
int main()
{
    // compteur = 0
    Rectangle r1;
    // compteur = 1
    {
        Rectangle r2;
        // compteur = 2

        Rectangle r3(r2);
        // compteur = ??
    }
    // compteur =
    return 0;
} // compteur =
```

👉 oops... la copie d'un rectangle échappe au compteur d'instances car il n'y a pas de définition explicite du constructeur de copie

Exemple

Il faudrait donc encore ajouter au code précédent, la définition *explicite* du constructeur de copie :

```
Rectangle(Rectangle const& r)
: hauteur(r.hauteur), largeur(r.largeur)
{ ++compteur; }
```

Règle générale : si on doit toucher à l'un des trois parmi destructeur, constructeur de copie et opérateur d'affectation (=), alors on doit certainement également toucher aux deux autres (ou alors au moins se poser la question !).

(En C++11, on peut ajouter le constructeur de déplacement et l'opérateur de déplacement à cette liste....
...quand on s'en préoccupe [avancé])