Constructeurs et destructeurs

Contexte

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

```
Rectangle rect;
```

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

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);
cout << "surface = " << rect.surface();
```

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

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

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

Les constructeurs

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

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

- pas de type de retour (pas même void)
- même nom que la classe

Exemple

```
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

Une classe peut donc avoir plusieurs constructeurs (listes de paramètres différentes).

Initialisation par constructeur

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

Exemple

```
Rectangle rect(4.2,8.4);
```

Exemple

```
#include <iostream>
using namespace std;
class Rectangle {
    public:
        Rectangle(double h, double L) {
            hauteur = h;
            largeur = L;
        }
        double surface() const {
            return hauteur * largeur;
        }
    private:
        double hauteur;
        double largeur;
};
```

```
int main() {
    double luH, luL;
    cout << "Quelle hauteur ? : ";
    cin >> luH;
    cout << "Quelle largeur ? : ";
    cin >> luL;
    Rectangle rect(luH,luL);
    cout << "surface = " << rect.surface();
    return 0;
}
```

```
Quelle hauteur ? : 4.2
Quelle largeur ? : 8.6
surface = 36.12
Process returned 0 (0x0)
Press any key to continue.
```

Liste d'initialisation

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

```
class RectangleColore {
    // ...
    private:
        Rectangle rectangle;
        int couleur;
};
```

Mauvaise solution :

```
RectangleColore(double h, double L, int c){
    rectangle = Rectangle(h, L);
    couleur = c;
}
```

=> Il faut initialiser directement les attributs en faisant directement appel à leurs propres constructeurs !

Syntaxe générale

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

Exemple

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

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

Exemple

```
#include <iostream>
using namespace std;
class Rectangle {
    public:
        Rectangle(double h, double L)
        :hauteur(h), largeur(L)
        {}
        double surface() const {
            return hauteur * largeur;
        }
    private:
        double hauteur;
        double largeur;
};
```

```
int main() {
    double luH, luL;
    cout << "Quelle hauteur ? : ";
    cin >> luH;
    cout << "Quelle largeur ? : ";
    cin >> luL;
    Rectangle rect(luH,luL);
    cout << "surface = " << rect.surface();
    return 0;
}
```

Remarque :

Les attributs non-initialisés :

- prennent une valeur par défaut si ce sont des objets
- restent indéfinis s'ils sont de type de base

Les attributs initialisés dans la liste d'initialisation peuvent être changés dans le corps du constructeur.

```
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
}
```

Remarque

```
public:
    Rectangle(double hauteur, double largeur)
    : hauteur(hauteur), largeur(largeur) // plus d'ambiguité
    {}
```

Instanciation d'objets et appels aux constructeur

```
class Rectangle {
    public:
        // Le constructeur par defaut
        Rectangle() : hauteur(1.2), largeur(2.6)
        {}
        // 2ème constructeur
        Rectangle(double c) : hauteur(c), largeur(2.0*c)
        {}
        // 3ème constructeur
        Rectangle(double h, double L) : hauteur(h), largeur(L)
        {}
        double surface() const {
            return hauteur * largeur;
        }
    private:
        double hauteur;
        double largeur;
};
```

```
surface 1 = 3.12
surface 2 = 11.52
surface 3 = 50.88

Process returned 0 (0x0)
Press any key to continue.
```

```
int main() {
    Rectangle rect1;
    cout << "surface 1 = " << rect1.surface() << endl;
    Rectangle rect2(2.4);
    cout << "surface 2 = " << rect2.surface() << endl;
    Rectangle rect3(4.8,10.6);
    cout << "surface 3 = " << rect3.surface() << endl;
    return 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**.

Exemples

```
// avec liste d'initialisation
Rectangle(): hauteur(2.4), largeur(8.4)
{}

// sans liste d'initialisation
Rectangle(){
    hauteur = 0.0;
    largeur = 0.0;
}
```

1. 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.

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

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	`Rectangle r1;`	`Rectangle r2(1.0, 2.0);`
(A)	constructeur par défaut par défaut	? ?	Illicite !
(B)	constructeur par défaut explicitement déclaré	0 0	Illicite !
(C)	un des **trois** construc-teurs est par défaut	0 0	1 2
(D)	pas de constructeur par défaut	Illicite !	1 2

Réactiver le constructeur par défaut généré par le compilateur

- 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).

C++11

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

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

Exemple

```
class Rectangle {
    public:
        Rectangle() = default; // mais peu pertinent ici
        Rectangle(double h, double L)
        : hauteur(h), largeur(L)
        {}
        double surface() const {
            return hauteur * largeur;
        }
    private:
        double hauteur;
        double largeur;
};
```

Un constructeur appel un autre constructeur

C++11

```
class Rectangle {
    public:
        Rectangle(double h, double L)
        : hauteur(h), largeur(L)
        {}
        Rectangle()
        : Rectangle(0.0, 0.0)
        {}
        double surface() const {
            return hauteur * largeur;
        }
    private:
        double hauteur;
        double largeur;
};
```

Initialisation par défaut des attributs

C++11

```
class Rectangle {
    public:
        //...
    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 rect1(2.4, 4.6);
Rectangle rect2(rect1);
```

- rect1 et rect2 sont deux instances distinctes
- mais ayant des mêmes valeurs pour leurs attributs

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

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

Passage des paramètres par valeur (copie de r1 dans r)

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

- 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

Syntaxe

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

- Passage par référence (sinon boucle infinie)
- Pas de modification de l'instance autre

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

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

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

Interdire la copie d'un objet

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

C++11

Les destructeurs

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 !

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

Syntaxe

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

- Le destructeur d'une classe est une méthode sans paramètre => pas de surcharge possible
- 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 à un moment donné dans un programme.

Utilisons comme compteur une variable globale de type entier :

- le constructeur incrémente le compteur
- le destructeur le décrémente

```
long compteur(0); // variable globale
class Badenya {
    public:
        Badenya() {
            ++compteur;
            numero = compteur;
            cout << "Constructeur objet " << numero << endl;
        }
        ~Badenya() {
            --compteur;
            cout << "Destructeur objet " << numero << endl;
        }
    private:
        int numero;
};
```

```
int main() {
    Badenya b1;
    {
        Badenya b2;
        {
            Badenya b3;
        }
    }
    return 0;
}
```

```
Constructeur objet 1
Constructeur objet 2
Constructeur objet 3
Destructeur objet 3
Destructeur objet 2
Destructeur objet 1

Process returned 0 (0x0)
Press any key to continue.
```

Exercice

Que se passe-il si l'on souhaite utiliser la copie d'objet ?

```
int main() {
    Badenya b1;
    {
        Badenya b2(b1);
        {
            Badenya b3;
        }
    }
    return 0;
}
```

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

Il faudrait donc ajouter la classe Badenya la définition explicite du constructeur de copie

```
Badenya(Badenya const& r) {
    ++compteur;
    numero = compteur;
    cout << "Constructeur de copie " << numero << endl;
}
```

Programme complet

```
long compteur(0); // variable globale
class Badenya {
    public:
        Badenya() {
            ++compteur;
            numero = compteur;
            cout << "Constructeur objet " << numero << endl;
        }
        ~Badenya() {
            --compteur;
            cout << "Destructeur objet " << numero << endl;
        }
        Badenya(Badenya const& r) {
            ++compteur;
            numero = compteur;
            cout << "Constructeur de copie " << numero << endl;
        }
    private:
        int numero;
};
```

```
int main() {
    Badenya b1;
    {
        Badenya b2(b1);
        {
            Badenya b3;
        }
    }
    return 0;
}
```

```
Constructeur objet 1
Constructeur objet 2
Destructeur objet 2
Destructeur objet 1
Destructeur objet 1

Process returned 0 (0x0)
Press any key to continue.
```

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)

Les objets Dynamiques

Mode de création d'un objet

Un objet (comme une variable) peut être déclarer de deux façons :

- par une déclaration : l'objet est alors de classe automatique (ou statique, voire plus tard), sa durée de vie est parfaitement définie par la nature et l'emplacement de sa déclaration ;

```
Rectangle rect1(2.2,4.6);
Rectangle rect2();
```

- en utilisant l'opérateur new ; l'objet est alors de classe dite dynamique ; sa durée de vie est contrôlée par le programme (opérateurs new et delete).

```
int main()
{
    Rectangle *rect1= new Rectangle(2.2,4.6);
    Rectangle *rect2= new Rectangle;
    cout << "surface 1 = " << (*rect1).surface() << endl;
    cout << "surface 2 = " << rect2->surface() << endl;
    delete rect1;
    delete rect2;
    return 0;
}
```

Exemple

```
class Rectangle {
    public:
        Rectangle(double h, double L) : hauteur(h), largeur(L){}
        ~Rectangle(){
            cout << "Destructeur" << endl;
        }
    private:
        double hauteur;
        double largeur;
};
```

```
int main()
{
    Rectangle *rect = nullptr;
    cout << "** Debut main \n" ;
    rect = new Rectangle(2.2,4.6);
    fct(rect);
    cout << "** Fin main \n" ;
    return 0;
}
void fct (Rectangle* r){
    cout << "** Debut fct \n" ;
    delete r; // destruction objet
    cout << "** Fin fct \n" ;
}
```

```
** Debut main
** Debut fct
Destructeur
** Fin fct
** Fin main

Process returned 0 (0x0)
```