

UE D - Introduction au C++

Définition des objets et des classes

1. Les bases du langage C++

1.1. Apports du C++ par rapport au C

C'est en 1983 que [Bjarne Stroustrup](#) des laboratoires Bell crée le C++ sur la base du langage C. Le langage C++ a été normalisé par l'ISO la première fois en 1998 (C++98), puis une seconde en 2003. Le standard actuel a été publié par ISO en septembre 2011 (C++11).

Le C++ a apporté par rapport au langage C les notions suivantes :

- les [concepts orientés objet](#) (encapsulation, héritage),
- les [références](#), la vérification stricte des types,
- les valeurs par défaut des paramètres de fonctions,
- la [surcharge de fonctions](#) (plusieurs fonctions portant le même nom se distinguent par le nombre et/ou le type de leurs paramètres),
- la [surcharge des opérateurs](#) (pour utiliser les opérateurs avec les objets), les constantes typées.

L'extension par défaut des fichiers écrits en langage C++ est **.cpp** ou **.cc**. Pour passer de C à C++, il suffit de changer de compilateur (**par exemple gcc vers g++**).

1.2. Un premier programme en C++

```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Donnez un entier : " << std::endl;
    std::cin >> n;

    for(int i = 0; i < n; i++)
        std::cout << "Hello world !" << std::endl;

    return 0;
}
```

Examinons chaque partie de ce code en détail:

- **#include <iostream>** est une **directive de préprocesseur** qui inclut le contenu du fichier d'en-tête C ++ standard **iostream**.
- **iostream** est un **fichier d'en-tête de bibliothèque standard** qui contient les définitions des flux d'entrée et de sortie standard. Ces définitions sont incluses dans l'espace de noms **std** , expliqué ci-dessous.
- **std::cout << "Hello World!" << std::endl;** imprime "Hello World!" au flux de sortie standard ;
- **std** est un espace de noms , et **::** est l' opérateur de résolution de portée qui permet de rechercher des objets par nom dans un espace de noms.
- **std::cout** est l'objet de flux de sortie standard , défini dans **iostream** , et il imprime sur la sortie standard (**stdout**) ;
- **<<** est, dans ce contexte, l'opérateur d'insertion de flux, ainsi appelé car il insère un objet dans l'objet flux.

1.3. Entrée/sortie basique en C++

Un flux est un canal recevant (flut « d'entrée ») ou fournissant (flot de « sortie ») de l'information. Ce canal est associé à un périphérique ou à un fichier. Un flux d'entrée est un objet de type **istream** tandis qu'un flux de sortie est un objet de type **ostream**. La bibliothèque standard **<iostream>** définit peu de flux pour l'entrée et la sortie:

stream	description
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

- **cin** est un flux d'entrée prédéfini connecté à l'entrée standard **stdin** d'un programme (le clavier par défaut).
- **cout** est un flux de sortie prédéfini connecté à la sortie standard **stdout** d'un programme (l'écran par défaut).
- **cerr** est un flux de sortie prédéfini (redirigée vers l'écran par défaut) connecté à la sortie standard des erreurs **stderr** d'un programme.

Le flux **cin** est essentiellement utilisé pour la saisie par l'utilisateur, tandis que le flux **cout** est utilisé pour l'affichage à la sortie standard ou au moniteur.

Les quatre flux sont situés dans l'espace de noms standard **std**.

Il y a aussi un manipulateur **std::endl** dans le code. Il ne peut être utilisé qu'avec des flux de sortie. Il insère le caractère '**\n**' de fin de ligne dans le flux et le vide.

1.4. Les espaces de noms en C++

Le terme **espace de noms** (**namespace**) désigne un lieu abstrait conçu pour accueillir (encapsuler) des ensembles d'entités (constantes, fonctions, classes, variables, ...) appartenant à un même domaine.

- En C++, un espace de nom (**namespace**) est une notion permettant de lever une ambiguïté sur des termes qui pourraient être homonymes sans cela.
- Il est matérialisé par un préfixe identifiant de manière unique la signification d'un terme. On utilise alors **l'opérateur de résolution de portée ::**

```
#include <iostream>

int global;

namespace nsA // crée un espace de noms nsA
{
    const int x = 2 ;
    int y;
}

int main(int argc, char **argv)
{
    const int x = 1;
    int y = 3;
    global = nsA::x;
    nsA::y = 4;

    std::cout << "nsA::x = " << nsA::x << "x = " << x << std::endl;
    std::cout << "nsA::y = " << nsA::y << "y = " << y << std::endl;
    std::cout << "global = " << global << std::endl;

    return 0;
}
```

Affichage :

nsA::x = 2 ; x = 1 ;

nsA::y = 4 ; y = 3 ;

global = 2

Pour utiliser les entités d'un espace de noms en dehors de celui-ci (voire dans un autre espace de noms), vous pouvez utiliser le nom précédé de l'**espace_de_nom::**. Ainsi, dans l'exemple précédent, l'utilisation de l'espace de noms **nsA::** avant les entités **x** et **y** permet de les distinguer des variables **x** et **y** de la fonction **main()**.

a) Alias d'espace de noms

Une autre manière très pratique d'utiliser un espace de noms surtout lorsque celui-ci est imbriqué dans d'autres espaces de noms sur plusieurs niveaux est l'emploi d'alias. Un alias permet de ramener une portée à un seul nom.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
        int f();
        int g();
        void bar(int x, int y);
    }
}
void foo() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::bar(N::f(), N::g());
}
```

La fonction **foo()** déclare localement un alias **N**. Les membres de cet espace de noms peuvent alors être accédés simplement en utilisant **N::**.

b) Directive Using

La directive **using** permet d'utiliser, sans spécification d'espace de nommage, toutes les entités de cet espace de nommage. La syntaxe de la directive **using** est la suivante :

using namespace nom;

où **nom** est le nom de l'espace de nommage dont les entités doivent être utilisées sans qualification complète.

```
namespace A
{
    int i;           // Déclare A::i.
    int j;           // Déclare A::j.
}

void f(void)
{
    using namespace A; // On utilise les entités de A.
    i=1;               // Équivalent à A::i=1.
    j=1;               // Équivalent à A::j=1.
    return ;
}
```

Si un espace de nommage est étendu après une directive **using**, les entités définies dans l'extension de l'espace de nommage peuvent être utilisées sans qualification complète de leurs noms.

```
namespace A
{
    int i;
}

using namespace A;

namespace A // extension de A
{
    int j;
}

int main(int argc, char **argv)
{
    i=0;    // Initialise A::i.
    j=0;    // Initialise A::j.
    return 0 ;
}
```

► L'introduction des entités d'un espace de nommage par une directive using peut faire apparaître des conflits de noms. Dans ce cas, aucune erreur n'est signalée lors de la directive using. En revanche, une erreur se produit si une des entités pour lesquels il y a conflit est utilisée.

```
namespace A
{
    int i;
}

namespace B
{
    int i;    // Définit B::i.
    using namespace A; // A::i et B::i sont en conflit.
                  // Cependant, aucune erreur n'est signalée.
}

int main(int argc, char **argv)
{
    using namespace B;
    i=1;    // Erreur : il y a ambiguïté.
    return 0 ;
}
```

1.5. Les références en C++

En C++ (pas en C), il est possible d'utiliser des **références** sur des objets (variables) définis ailleurs: cela permet de créer un **nouveau nom** qui sera un **synonyme** de cette variable (comme un **alias** sur la variable référencée).

La syntaxe de la déclaration d'une référence est la suivante : **type &référence(identificateur);**

- On pourra modifier le contenu d'une variable en utilisant une référence sur celle-ci.
- Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Elle doit absolument être initialisée vers un objet existant.
- Une référence ne peut référencer qu'une seule variable tout au long de sa durée de vie.

```
#include <iostream>

int main (int argc, char **argv)
{
    int i = 10; // i est un entier valant 10
    int j(20); // j est un entier valant 20
    int &ri = i; // ri est une référence sur un entier qui est i
    int &rj(j) ; // rj est une référence sur un entier qui est j
    int &k = 44; // illégal : la référence doit être liée à un objet
    int &k(ri) ; // illégal : on ne peut pas référencer une référence.
                // une référence n'est pas un objet en mémoire !!

    // A partir d'ici ri est synonyme de i, ainsi :
    ri = ri + 1; // est équivalent à i = i + 1 !
    ri = j;      // ne veut pas dire que ri est maintenant un alias de j

    return 0;
}
```

Les références **doivent** être initialisées correctement au moment de la définition et ne peuvent plus être modifiées par la suite. Le morceau de code suivant provoque une erreur de compilation:

```
int &i; // error: declaration of reference variable 'i' requires an
initializer
```

On ne peut pas non plus lier directement une référence à **nullptr**, contrairement aux pointeurs:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: invalid initialization of non-const
reference of type 'int&' from an rvalue of type 'std::nullptr_t'
```

Intérêt des références :

- Comme une référence établit un lien entre deux noms, leur utilisation est efficace dans le cas de variable de grosse taille car cela évitera toute copie.
- Les références sont (systématiquement) utilisées dans le passage des paramètres d'une fonction (ou d'une méthode).

```
#include <iostream>
using namespace std;
void affectation (int &x ) {
    int b ;
    cout << "valeur de b : ";
    cin >> b ;
    x = b;
}

int main (int argc, char **argv)
{
    int i;
    cout << "Adresse à l'intérieur de Main => " << &i <<endl;

    affectation(i);
    cout << "Valeur de i après l'appel => " << i <<endl;

    return 0;
}
```

1.6. Opérateurs pour la gestion de la mémoire

a) L'opérateur new

Pour allouer dynamiquement en C++, on utilisera l'opérateur **new**. Celui-ci admet comme opérande une spécification de type et retourne un pointeur vers ce type. Plus besoin, comme avec **malloc**, de convertir le résultat avec l'opérateur de cast. La définition dynamique d'une variable simple peut s'accompagner d'une valeur d'initialisation, mise entre parenthèses.

```
int *ip1;
int *ip2;
int taille;
ip1 = new int(0); // création d'une variable int, initialisée à 0

taille = 10 ;
ip2 = new int[taille] ; // création d'un tableau d'entiers ;
```

Avec *malloc*, la création d'un tableau d'entiers demande à ce qu'on passe comme paramètre la taille totale (en octets) du tableau. **new** est plus simple à utiliser, puisqu'il suffit d'indiquer la taille.

► Comme nous le verrons plus loin, la création d'objets dynamiques par **new** s'accompagne d'un appel au constructeur.

b) L'opérateur **delete**

Une zone mémoire allouée par **new** peut être libérée via l'opérateur **delete**. Il admet comme unique opérande l'adresse de la zone à libérer.

```
int *ip1;
int *ip2;
int taille;
ip1 = new int(0); // création d'une variable int, initialisée à 0

taille = 10 ;
ip2 = new int[taille] ; // création d'un tableau d'entiers

delete ip1 ; // désallocation de l'espace alloué à ip1
delete ip2 ; // désallocation incorrecte d'un tableau
delete [] ip2 ; // désallocation valide d'un tableau
```

La libération d'un tableau dynamique se fait avec la syntaxe : **delete [] nomTableau;**

► En C ++, **delete []** est un opérateur avec un comportement très spécifique: une expression avec l'opérateur **delete []**, appelle d'abord les destructeurs appropriés pour chaque élément du tableau (si ceux-ci sont de type classe), puis appelle une fonction de désallocation de tableau.

1.7. Surcharge de fonctions

La **surcharge de fonctions** fait en sorte que plusieurs fonctions déclarées dans la même étendue avec exactement le **même nom** existent au même endroit et ne diffèrent que par leur **signature**, c'est à dire les arguments qu'elles acceptent.

En C++, il est possible de donner le même nom à plusieurs fonctions. Considérons le code ci-dessous qui montre un exemple de surcharge de la fonction **print**.

```
void print(const std::string &str)
{
    std::cout << "C'est un string: " << str << std::endl;
}

void print(int num)
{
    std::cout << "C'est un int:  " << num << std::endl;
}
```

Les deux fonctions ont le même nom mais des signatures différentes. L'un accepte **std::string**, l'autre un **int**, Lors de l'appel d'une des deux fonctions, il n'aura aucune ambiguïté sur la fonction appelée.

Affichage :

```
print("Hello world!"); //affiche "C'est un string: Hello world!"
print(1337);           //affiche "C'est un int: 1337"
```

Lorsque vous avez des fonctions surchargées, le compilateur déduit quelles fonctions appeler à partir des paramètres que vous lui avez fournis. Des précautions doivent être prises lors de l'écriture de surcharges de fonctions. La seule contrainte : **pas d'ambiguïté sur l'identité de la fonction à partir des arguments**.

Dans l'exemple ci-dessous, si vous appelez la fonction avec **print(5)**, il n'est pas clair quelle surcharge de **print** est appelée.

```
void print(double num)
{
    std::cout << "C'est un double: " << num << std::endl;
}

void print_int(int num)
{
    std::cout << "C'est un int:  " << num << std::endl;
}
```

1.8. L'opérateur Const et les variables

Le mot réservé « **const** » se retrouve dans plusieurs contextes, en particulier lorsqu'on a affaire à des classes ; ce qui est parfois source de confusions. Essayons de clarifier ses emplois.

Avant tout, commençant par quelques rappels :

1. le mot réservé « **const** » qualifie un nom de variable pour indiquer qu'*au travers de ce nom*, la valeur ne peut pas être modifiée ;
2. **const** s'applique toujours à ce qui le précède ; sauf s'il n'y a rien devant, auquel cas il s'applique à ce qui suit.

Pour le point 1. :

```
int const i(3);
```

empêche de modifier la valeur de **i** : le code suivant ne compilera pas : **i** = 5;

Pour l'aspect « *au travers de ce nom* » du point 1. :

```
int const i(3);  
int* ptr(&i);
```

ptr pointe sur **i**, mais sans être lui-même **const**. (Attention ! C'est justement de la mauvaise programmation !... mais c'est possible. Alors,

```
i = 5;
```

ne compilera toujours pas, mais par contre

```
*ptr = 5;
```

est tout à fait possible et pourrait donc changer la valeur de **i** ! Le **const** sur **i** ne veut donc pas dire que la valeur de **i** ne peut pas changer dans l'absolu, mais bien qu'elle ne peut pas changer au travers du nom **i**.

Pour le second point : on peut tout aussi bien écrire

```
int const i(3);
```

que

```
Const int i(3);
```

Dans les deux cas, c'est bien l'int qui est **const**.

La situation se complique avec les pointeurs/références :

```
const int* ptr1;  
int const* ptr2;  
int* const ptr3(&i);
```

Qui est quoi ?

- **ptr1** est un pointeur sur un « **const int** », exactement ce qu'aurait du être **ptr** dans l'exemple ci-dessus pour ne pas pouvoir modifier la valeur de **i** au travers de **ptr1** : on ne peut pas modifier la valeur (de type int) pointée.
- **ptr2** est exactement de même type que **ptr1**.
- **Ptr3**, par contre, est un « **pointeur const** » sur un **int** : ici c'est l'adresse pointée par **ptr3** qui ne peut pas être modifiée (au travers de **ptr3**) : **ptr3** pointe toujours au même endroit (sur **i** dans cet exemple). Par contre, la valeur pointée par **ptr3** peut tout à fait être modifiée.
- Et on peut bien sûr vouloir protéger les deux, l'adresse et la valeur pointée :

```
int const * const ptr3(&i);
```

2. Classes, instances, attributs et Méthodes en C++

2.1. Notion de classe

Une **classe** déclare des propriétés communes à un ensemble d'objets. Elle permet donc de définir un **type d'objet** à partir duquel il sera possible de créer des objets.

En C++, une classe se déclare avec le mot-clé **class** , qui définit un nouveau type. On utilise la syntaxe suivante : **class NomClasse {...};**

Une fois le nouveau type défini, on peut l'utiliser pour déclarer des variables, **instances** de cette classe. La syntaxe est la suivante : **NomClasse NomVariable ;**

L'exemple ci-dessus donne la déclaration d'une classe **Rectangle**, et d'une variable **rect1** de ce type.

2.2. Attributs et méthodes de classe

Le processus de déclaration des **attributs**, au sein d'une classe, est similaire à celui des champs d'une structure : on écrit le type puis le nom de chaque attribut, le tout suivi d'un point-virgule. Pour utiliser ces attributs, on emploie également la même syntaxe que pour manipuler les champs d'une structure : le nom de l'instance, suivi d'un point et du nom de l'attribut permettent de le désigner.

Les **méthodes** sont des fonctions qui sont déclarées au sein de la classe. Le prototype d'une méthode indique donc le type de retour, le nom de la méthode puis entre parenthèses rondes, la liste des éventuels paramètres. L'entête de la méthode est complété par son corps, ou sa définition, entre accolades. La syntaxe sera donc la suivante :

```
type_retour Nom_Methode (type1 nom_parametre1,...){ // Corps de la méthode ... }
```

► Les paramètres d'une méthode sont des variables externes à la classe : chaque méthode d'une classe a accès aux attributs de celle-ci ; ils ne doivent donc pas être passés en arguments.

Règles de bonne pratique :

- Pour une meilleure **séparation** entre l'**interface** et l'**implémentation**, on peut choisir de définir les méthodes à l'**extérieur** de la classe. Le prototype de la méthode devra cependant rester à l'intérieur de la déclaration de la classe.
- Pour indiquer au compilateur que l'on définit une méthode et non pas une fonction, il faut indiquer à quelle classe elle appartient grâce à l'opérateur de résolution de portée ::, de la forme suivante :
`type_retour Nom_Classe :: Nom_Methode (type1 nom_parametre1,...) { // Corps de la méthode ...}`, comme c'est le cas pour la fonction `surface()`.
- Il est utile de distinguer les méthodes qui **modifient** les attributs de l'instance manipulée de celles qui ne le font pas. On parle alors de **méthodes constantes**. Une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet.
 - L'usage du mot **const** dans le prototype d'une fonction, après la liste de ses paramètres, indique que la fonction en question ne modifie pas l'instance à laquelle elle s'applique, comme dans l'entête de la méthode `getHauteur()`.

```
#include <iostream>

using namespace std;

class Rectangle {
    private :
        double largeur, hauteur;

    public:
        double surface() const ;

        // getters ou accesseurs
        double getHauteur() const {return hauteur; }
        double getLargeur() const {return largeur; }

        // setters
        void setHauteur(double h) {hauteur = h; }
        void setLargeur(double largeur) {largeur = largeur; }
};
```

```
Double Rectangle::surface() {  
    return largeur * hauteur;  
}  
  
int main(int argc, char **argv)  
{  
    Rectangle rect1;  
    rect1.setHauteur(3.0)  
    rect1.setLargeur(4.0)  
    cout << "hauteur: " << rect1.getHauteur() << endl;  
    cout << "largeur: " << rect1.getLargeur() << endl;  
    return 0;  
}
```

2.3. const et les classes

En POO, il y a quatre utilisations possibles de **const** : trois pour les méthodes et une pour les variables/attributs.

`const int i;` veut simplement dire qu'une fois initialisée, **la valeur de cet attribut ne peut plus être modifiée** (au travers de ce nom `i`), **même par une méthode de la classe** (autre qu'un constructeur).

C'est exactement le même rôle que dans le rappel précédent, à la seule différence que les attributs sont initialisés par les constructeurs et non pas directement.

Pour les méthodes, on peut trouver **const** à trois endroits différents :

```
CONST1 type_de_retour methode(CONST2 parametre) CONST3;
```

Par exemple : `const int & method(const Point & p) const;`

- Le premier **const** (« CONST1 »), plutôt rare car pas souvent nécessaire, sert à dire que la **valeur de retour ne peut pas être affectée/modifiée** ; on ne peut pas écrire quelque chose comme : `int x = obj.method(p1)` ;
- Le second **const** (« CONST2 ») est tout à fait « classique » et n'est pas liée aux méthodes ; il existe aussi de la même façon pour les fonctions usuelles. Il sert à dire que **l'argument reçu au travers de ce paramètre ne sera pas modifié**, et donc que
 1. si l'on fait « `obj.method(p1)` », `p1` n'est pas modifié. Dans un passage par valeur, ceci est bien sûr totalement inutile ! Cela ne prend vraiment de sens que dans un passage par référence.
- Le troisième **const** enfin (« CONST3 ») est spécifique aux méthodes. Il sert à dire que **l'appel à cette méthode ne modifie pas l'instance courante**, i.e. lorsque l'on fait « `obj.method(p1)` », on est sûr que `obj` ne sera pas modifié.

Conséquence : si l'on a la déclaration suivante :

```
const Objet obj(b1, b2);
```

alors, bien sûr, on ne peut appeler que des « méthodes **const** » (au sens du CONST3 ci-dessus) sur cet objet !

Sinon l'aspect « **const** » de la déclaration de cet objet ne peut pas être garanti et le compilateur nous dira que l'appel à une méthode non-const « discards qualifier » : cet appel annulerait (« discard ») la qualification **const** de l'objet obj.

2.4. Notion de visibilité

Le C ++ permet de préciser le **type d'accès des membres** (attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets :

- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et non par ceux d'une autre classe.

Dans l'exemple de la classe Rectangle, les attributs largeur et longueur sont privés ; ils sont donc pas inaccessibles depuis l'extérieur de la classe. Par conséquent, tout appel à ses éléments depuis le main provoquera une erreur de compilation.

```
Rectangle rect1;           // je suis une instance (un objet) de la classe
rect1.hauteur = 3.0;       // erreur d'accès : 'double Rectangle::hauteur' is private
rect1.largeur = 4.0;       // idem
```

► A l'intérieur de la méthode surface() de la classe Rectangle, on peut accéder directement à largeur du rectangle en utilisant la donnée membre **largeur**. De la même manière, on pourra accéder directement à la hauteur du rectangle en utilisant la donnée membre **hauteur**.

2.5. Accesseurs et Manipulateurs

Tous les attributs de la classe **Rectangle** étant privés par respect du principe d'encapsulation, on veut néanmoins pouvoir les manipuler, notamment pour modifier ou connaître leur valeur depuis l'extérieur de la classe.

Pour permettre un accès extérieur en lecture et/ou écriture aux attributs privés d'une classe, il est alors nécessaire d'utiliser des **méthodes publiques**, appelés **accesseurs** et **mutateurs**, pour établir un accès contrôlé à ces attributs :

1. **accesseur** (méthode «get » ou « getters») : permet l'accès en lecture,

2. **mutateur** (méthode «set» ou « setteur ») : permet l'accès en écriture.

Dans l'exemple de la classe Rectangle, la méthode publique **getHauteur()** est un accesseur et la méthodes **setHauteur()** et un mutateur (ou manipulateur) de l'attribut **hauteur**.

On utilisera ces méthodes pour accéder en lecture ou écriture aux membres privés d'un rectangle.

```
// on peut modifier le membre hauteur de rect1
rect1.setHauteur(3.0)
// on peut modifier le membre largeur de rect1
rect1.setLargeur(4.0)
// on peut accéder au membre hauteur de rect1
cout << "hauteur: " << rect1.getHauteur() << endl;
// on peut accéder au membre largeur de rect1
cout << "largeur: " << rect1.getLargeur() << endl;
return 0;
```

3. Constructeurs et destructeurs d'objets

3.1. Constructeurs d'objets

Lors de la déclaration d'un nouvel objet, il existe différentes façons de donner une valeur à ses attributs. On pourrait opter pour des mutateurs qui modifient individuellement chaque valeur, mais il s'agirait d'une mauvaise solution car elle implique que tous les attributs soient assortis d'un mutateur, ce qui ferait dépendre l'interface de l'implémentation et nuirait donc à l'encapsulation. De plus, cette solution oblige l'utilisateur à initialiser explicitement tous les attributs, au risque que certains soient oubliés.

Une meilleure solution consiste à définir une méthode dédiée à l'initialisation des attributs. Elle prend en paramètres les valeurs que prendront les attributs. Une telle méthode est appelée un **constructeur**.

- Un constructeur est méthode invoquée **systématiquement lors de la création d'un objet** d'une classe.
- Il est chargé **d'initialiser un objet de la classe**, notamment **ses attributs**.

- Un constructeur est une **méthode qui porte le même nom que la classe** et n'a pas de type de retour.
- On peut également faire des **surcharges de constructeurs** et donner **des valeurs par défaut** à leurs paramètres.

La syntaxe de déclaration d'un constructeur est la suivante :

Nom_Classe instance (liste_paramètres) ;

```
class Rectangle {  
    private :  
        double largeur, hauteur;  
  
    public:  
  
    // constructeur de classe  
    Rectangle (double hauteur, double largeur);  
    double surface () const;  
};
```

Il faut maintenant **définir** le constructeur de la classe **Rectangle** afin qu'il **initialise tous les attributs de l'objet au moment de sa création** :

```
Rectangle::Rectangle(double hauteur, double largeur) {  
    this->largeur = largeur;  
    this->hauteur = hauteur ;  
}
```

Les méthodes d'instance ont toutes un argument supplémentaire, nommé **this**, qui désigne l'objet sur lequel la méthode est appelée (appelé l'**objet receveur** car c'est lui qui reçoit le message d'invocation de la méthode). Nous n'avons pas à le préciser explicitement parmi les arguments car **this** est un paramètre implicite. Toutes les méthodes d'instance l'ont.

L'opérateur "**this**" permet donc de désigner l'**adresse de l'instance en cours de la classe** sur laquelle la fonction membre a été appelée.

► Dans notre exemple, les noms des paramètres du constructeur sont identiques aux noms des attributs de la classe. L'opérateur **this** permet d'indiquer qu'il s'agit d'un nom de l'attribut et non pas d'un nom de paramètre.

La syntaxe de déclaration avec initialisation d'un objet est la suivante :

Nom_Classe instance (valeur_arg1, ..., valeur_argN) ;

où **valeur_arg1**, ..., **valeur_argN** sont les valeurs passées aux paramètres du constructeur.

```
// invocation du constructeur Rectangle avec 2 paramètres
Rectangle rect1(3.0,4.0);
```

3.2. Liste d'initialisation

Un meilleur moyen d'initialiser les données membres de la classe lors de la construction est la [liste d'initialisation](#). La syntaxe est la suivante :

Nom_Classe (liste_paramètres) :

```
    attribut1(...), // appel au constructeur de attribut1
    ...,
    attributN(...) ; // appel au constructeur de attributN
{ // autres opérations }
```

Pour les attributs de type de base, il suffit de spécifier la valeur entre parenthèses, après le nom de l'attribut, pour l'initialiser. On va utiliser cette technique pour définir le constructeur de la classe **Rectangle** :

```
class Rectangle {
    private :
        double largeur, hauteur;

    public:
        Rectangle (double H, double L)
        : hauteur(H) // initialisation
        {
            largeur = 2.0 * L + H; // définition de la valeur de largeur
        }

        double surface () const;
};
```

Dans l'exemple ci-dessus, les initialisations peuvent être modifiées dans le corps du constructeur. Ainsi, en entrant dans le corps de la méthode, la valeur de l'attribut largeur est encore indéterminée, puis elle est modifiée par une affectation.

Considérons le cas où les attributs d'une classe sont eux-mêmes des objets. Soit une classe **Rectangle_Colore** ayant un attribut de type **Rectangle** et un autre attribut couleur de type énuméré :

```
typedef enum { BLUE, BLAND, ROUGE} Couleur;

class Rectangle {
private :
    double largeur, hauteur;

public:
    Rectangle (double h, double l);
};

class Rectangle_Colore {
private :
    Rectangle rectangle;
    Couleur couleur;

public:
    Rectangle_Colore (double H, double L, Couleur c);
} ;
```

Un exemple de constructeur pour **Rectangle_Colore** peut être défini de la façon suivante :

```
Rectangle_Colore::Rectangle_Colore(double H, double L, Couleur c) {
    rectangle = Rectangle(l,h);
    couleur = c;
}
```

Mauvaise solution

Cette solution **est mauvaise** car l'appel du constructeur **Rectangle** sans nom d'objet crée une **instance anonyme**¹ de la classe **Rectangle** qui sera copiée dans l'attribut rectangle. Du coup, on aura deux rectangles, le premier anonyme initialisé par l'appel au constructeur et le second l'attribut rectangle de l'instance que nous sommes entrain d'initialiser.

Il faudrait donc initialiser directement les attributs-objets en faisant appel à leurs propres constructeurs, comme illustré ci-dessous :

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

3.3. Constructeur par défaut

Le **constructeur par défaut** est un constructeur qui **n'a pas de paramètres** ou dont **tous** les paramètres ont des **valeurs par défaut**. Il est appelé lors de la déclaration d'un objet sans valeurs d'initialisation.

1 Une instance anonyme est un objet sans nom.

Le constructeur par défaut de la classe Rectangle sera :

```
class Rectangle {  
    private :  
        double largeur, hauteur;  
  
    // Le constructeur par défaut  
    public:  
        Rectangle () : hauteur(3.0), largeur(4.0) {}  
};
```

L'appel à un constructeur par défaut se fait en déclarant une instance par la syntaxe :

Nom_Classe Nom_Instance ;

```
// invocation du constructeur par défaut  
Rectangle rect1;
```

► L'ajout de parenthèses vides après le nom de l'instance est une erreur à éviter, interprétée par le compilateur comme le prototype d'une fonction.

3.4. Paramètres par défaut

Le langage C ++ offre la possibilité d'avoir des **valeurs par défaut pour les paramètres d'une fonction** (ou d'une méthode), qui peuvent alors être sous-entendus au moment de l'appel. Cette possibilité, permet d'écrire qu'un seul constructeur profitant du mécanisme de valeur par défaut :

```
class Rectangle {  
    private :  
        double largeur, hauteur;  
  
    // Le constructeur par défaut avec paramètres par défaut  
    public:  
        Rectangle (double hauteur = 3.0, double largeur = 4.0) {}  
};
```

Lorsqu'une instance de cette classe est initialisée, si le constructeur appelé ne modifie par les valeurs par défaut pour les attributs **largeur** et **hauteur**, ces derniers prennent alors les valeurs par défaut.

```
// invocation du constructeur par défaut  
Rectangle rect1; // hauteur = 3.0 et largeur = 4.0
```

Avec cette façon de faire, il est donc possible de regrouper un constructeur par défaut et un constructeur avec paramètres en utilisant les valeurs par défaut des paramètres :

```
class Rectangle {
    private :
        double largeur, hauteur;

    // Le constructeur par défaut
    public:
        Rectangle (double c = 3.0) : hauteur(c), largeur(2.0 * c) {}
};

// invocation du constructeur par défaut
Rectangle rect1;    // hauteur = 3.0 et largeur = 4.0

// invocation du constructeur de classe
Rectangle rect1(4.0) // hauteur = 4.0 et largeur = 8.0
```

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

- Si ces attributs sont des objets, alors pour les initialiser, il va utiliser les constructeurs par défaut associés à ces objets.
- Si les attributs sont de type de base, ils resteront non initialisés.

Considérons la classe **Rectangle** qui, en plus des attributs **hauteur** et **largeur**, de type de base, possède un attribut de type **Position** permettant de modéliser la position du rectangle sur l'écran.

```
class Rectangle {
    private :
        double largeur, hauteur;
        Position pos;
};
```

Lorsqu'une instance de cette classe est initialisée, comme il n'y a aucun constructeur explicite dans la classe, le compilateur va donc invoquer le **constructeur par défaut par défaut**. Ce constructeur va construire un objet où les champs largeur et les champs hauteur demeurent non initialisés car ils sont de type de base et où le champ position aurait une valeur donnée par le constructeur par défaut de la classe Position s'il y en a un.

3.5. Constructeur de copie

Nous allons regarder à présent ce qui se passe lorsqu'un objet est initialisé avec une copie d'un autre objet de la même classe. C++ offre le moyen de faire des copies d'instances.

```
// invocation du constructeur usuel
Rectangle rect1(2.5,4.0);
// invocation du constructeur de copie
Rectangle rect2(rect1)
```

Dans l'exemple ci-dessus, nous avons une première instance **rect1** de la classe **Rectangle**, initialisée avec un constructeur usuel, et une seconde instance **rect2**, de la même classe **Rectangle**, mais qui est initialisée avec une copie de **rect1**, c'est à dire initialisée avec une copie des valeurs, donc **2.5** pour la hauteur et **4.0** pour la largeur.

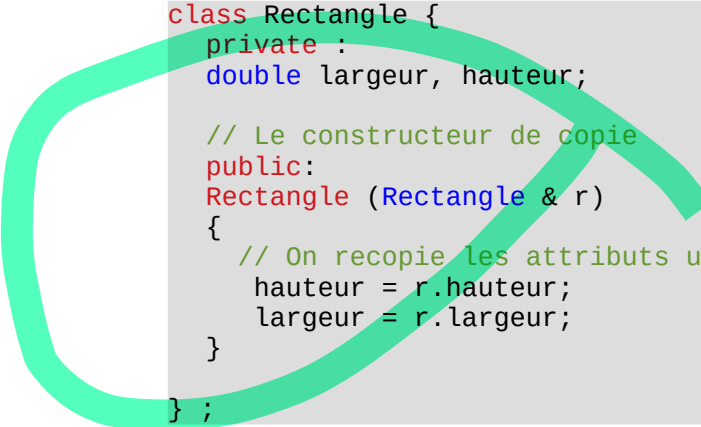
► Notez que **rect1** et **rect2** sont des instances distinctes qui vivent séparément dans la mémoire mais simplement **rect2** est initialisé lors de sa construction avec une copie des valeurs de **rect1**.

Le prototype du constructeur de copie est le suivant :

Nom_Classe(Nom_Classe const& autre) {...}

Le constructeur de copie reçoit un seul paramètre qui est donc une autre instance de la même classe. Cette instance est passée par **référence constante** pour éviter de faire un passage par valeur qui appellerait de nouveau une copie, auquel cas, on ferait des copies de copies.

Dans le cas de la classe **Rectangle**, le constructeur de copie sera :



```
class Rectangle {  
    private :  
        double largeur, hauteur;  
  
    // Le constructeur de copie  
    public:  
    Rectangle (Rectangle & r)  
    {  
        // On recopie les attributs un par un  
        hauteur = r.hauteur;  
        largeur = r.largeur;  
    }  
};
```

► Remarque : Il n'est souvent pas nécessaire de définir explicitement le constructeur de copie. En effet, le compilateur C++ en fournit une version par défaut souvent suffisante.

3.6. Destructeurs

Le **destructeur** est la **méthode membre appelée automatiquement** lorsqu'une instance (objet) de classe cesse d'exister en mémoire. Son rôle est de libérer toute la mémoire allouée à l'objet.

Un destructeur est une méthode qui porte toujours le même nom que la classe, précédé du signe "~". Une telle fonction

- ne possède aucun paramètre,
- n'admet aucune valeur de retour.
- Il n'y a qu'un seul destructeur par classe, c'est à dire **pas de surcharge possible**.

La forme habituelle d'un destructeur est la suivante :

```
class T {  
    public:  
    ~T(); // destructeur  
};
```

► Si le destructeur n'est pas défini explicitement par le programmeur, le compilateur en génère automatiquement une version minimale.

3.7. Membres statiques

Une classe peut contenir des attributs et/ou des méthodes statiques. Ces données membres appartiennent à la classe, et non pas aux objets de cette classe. Elles sont donc communes à tous ces objets. L'emploi du mot clé **static** permet de caractériser les données membres statiques des classes, les fonctions membres statiques des classes, et les données statiques des fonctions membres.

3.7.1. Les attributs statiques

Un membre donnée déclaré avec l'attribut **static** est **partagé par tous les objets de la même classe**. Il est impossible d'initialiser les attributs d'une classe dans le constructeur de la classe, car le constructeur n'initialise que les données des nouveaux objets.

Les données statiques ne sont pas spécifiques à un objet particulier et ne peuvent donc pas être initialisées dans le constructeur. Un attribut membre statique doit être initialisé en dehors de la déclaration de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::).

Continuons avec notre exemple de la classe **Rectangle** et supposons que l'on veuille compter le nombre d'instances actives de la classe **Rectangle**.

pour cela, on va utiliser un **membre statique** **cpt** pour **compter le nombre d'objets Rectangle créés à un instant donné**.

Il faut maintenant **initialiser** ce membre statique et **compter/décompter** le nombre d'objets créés et détruits.

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

class Rectangle {
private :
    double largeur, hauteur;
    static int cpt;

public:
    // constructeur par défaut
    Rectangle () : hauteur(0.0), largeur(0.0) {
        ++cpt;
        cout << "c'est le constructeur par default" << endl;
    }

    // destructeur par défaut
    ~Rectangle () {
        --cpt;
        cout << "c'est le destructeur par default" << endl;
    }
};
```

Imaginons maintenant la déclaration de l'instance **r3** par le moyen d'une instance de copie **r2**. Quand est-il du comptage dans ce cas ?

► Sans la définition explicite du constructeur de copie, la copie d'un rectangle échappera au compteur d'instances.

Il faudrait donc encore ajouter au code précédent, la définition explicite du constructeur de copie qui se chargera du comptage d'instances.

► Tous les constructeurs de la classe Rectangle doivent incrémenter le compteur cpt.

Note : Imaginons le cas ci-dessous où les attributs de la classe sont des pointeurs et le constructeur de la classe alloue de la mémoire pour les deux attributs lors de la création de l'objet. Imaginons un programme qui utiliserait la classe **Rectangle** : **Rectangle rect** ;

Question : qu'advient-il de toutes les ressources allouées après la fin du programme (mémoire allouée aux deux attributs de la classe) ? memory leak

Dans notre cas, il faudra explicitement désallouer toute la mémoire avec l'instruction **delete**. La prise en charge de cette opération se fera via le destructeur.

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

class Rectangle {
private :
    double *largeur;
    double *hauteur;
    static int cpt;

public:
    // constructeur par défaut
    Rectangle (): hauteur(new double(0.0)), largeur(new double(0.0)) {
        ++cpt;
        cout << "c'est le constructeur par default" << endl;
    }

    // destructeur par défaut
    ~Rectangle () {
        delete largeur;
        delete hauteur;
        --cpt;
        cout << "c'est le destructeur par default" << endl;
    }
};
```

3.7.2. Les fonctions membres statiques

Lorsqu'une fonction membre a **une action indépendante d'un quelconque objet de sa classe**, on peut la déclarer avec l'attribut **static**.

Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Les fonctions membre statiques :

- ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.

- peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

On va utiliser une fonction membre statique **compte()** pour connaître le nombre d'objets Point existants à un instant donné.

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

class Rectangle {
private :
    double largeur, hauteur;
    static int cpt;

public:
    // constructeur par défaut
    Rectangle () : hauteur(0.0), largeur(0.0) {
        ++cpt;
        cout << "c'est le constructeur par default" << endl;
    }

    // constructeur de copie
    Rectangle (Rectangle & r) : largeur(r.largeur), hauteur(r.hauteur){
        ++cpt;
        cout << "c'est le constructeur de copie" << endl;
    }

    // destructeur par défaut
    ~Rectangle () {
        --cpt;
        cout << "c'est le destructeur par default" << endl;
    }

    // retourne le nombre d'objets Rectangle
    static int compte()
} ;

// Il faut maintenant définir cette méthode statique :
int Rectangle::compte() {
    return cpt;
}
```

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

class Rectangle {
private :
    double largeur, hauteur;
    static int cpt;

public:
    // constructeur par défaut
    Rectangle () : hauteur(0.0), largeur(0.0) {
        ++cpt;
        cout << "c'est le constructeur par default" << endl;
    }

    // constructeur de copie
    Rectangle (Rectangle & r) : largeur(r.largeur), hauteur(r.hauteur){
        ++cpt;
        cout << "c'est le constructeur de copie" << endl;
    }

    // destructeur par défaut
    ~Rectangle () {
        --cpt;
        cout << "c'est le destructeur par default" << endl;
    }
} ;

int Rectangle::cpt(0); // attribut de classe

int main(int argc, char **argv)
{
    Rectangle r1;
    cout << "cpt (r1): " << Rectangle::cpt << endl;
    {
        Rectangle r2;
        cout << "cpt (r2): " << Rectangle::cpt << endl;
        Rectangle r3(r2);
        cout << "cpt (r3): " << Rectangle::cpt << endl;
    }
    cout << "compteur: " << Rectangle::cpt << endl;

    Rectangle r4;
    cout << "cpt: " << Rectangle::cpt << endl;
    return 0;
}
```

4. Gestion des exceptions en C++

4.1. Principe général

Une **exception** est l'interruption de l'exécution du programme à la suite d'un événement particulier et le transfert du contrôle à des fonctions spéciales appelées **gestionnaires**.

Le but des exceptions est de réaliser des traitements spécifiques lorsqu'une situation anormale est détectée. C'est le cas par exemple de la division par zéro qui peut créer une erreur. Dans ce cas, on peut simplement arrêter le programme.

La **gestion d'une exception** est découpée en deux parties distinctes :

- le déclenchement : instruction **throw**
 - **throw** signale l'erreur en lançant un objet ;
- le traitement : l'inspection et la capture : deux instructions inséparables **try** et **catch**
 - **try{ ... }** signale une portion de code où une erreur peut survenir ;
 - **catch(...) { ... }** introduit la portion de code qui récupère l'objet et gère l'erreur.

Bloc de code protégé (unique)	{	<pre>try { // Code susceptible de lever une exception }</pre>	}	
Gestionnaires d'exceptions (multiples)	{	<pre>catch (TypeException &identificateur) { // Code de gestion d'une exception }</pre>	}	ⁿ

4.2. Lancement et récupération d'une exception

En C++, lorsqu'il faut lancer une exception, on doit créer un objet dont la classe caractérise cette exception, et utiliser le mot clé **throw**. Sa syntaxe est la suivante :

throw objet;

où *objet* est l'objet correspondant à l'exception. Cet objet peut être de n'importe quel type, et pourra ainsi caractériser l'exception.

L'exception doit alors être traitée par le gestionnaire d'exception correspondant. On doit donc placer le code susceptible de lancer une exception d'un bloc d'instructions particulier.

Ce bloc est introduit avec le mot clé **try**, suivi par le bloc **catch** qui reçoit en paramètres la référence sur l'objet de l'exception, comme le montre la syntaxe indiquée ci-dessous.

```
try
{
    // instructions susceptible de générer des exceptions...
} catch (TypeException & e) {
    // Instructions de traitement de l'exception
}
```

La levée d'une exception court-circuite le fil d'exécution « normal » en sautant au bloc **catch** de traitement de l'exception. Les instructions du bloc **try** situées après celle ayant levé l'exception ne sont donc pas exécutées.

Il est bien souvent impossible de retourner un résultat. Lever une exception est un moyen de signaler le problème au code appelant, charge à lui de gérer la situation.

Un sous-programme susceptible de lever une exception doit l'indiquer via l'instruction « **throw std::exception** ("explication des raisons de la levée d'exception") ; ».

Dans l'exemple ci-dessous, la fonction division doit retourner le résultat de la division de **a** par **b** passés en paramètre, ce qui peut provoquer une erreur si la valeur de **b** vaut zéro.

Le code lève par conséquent une exception si **b == 0**. Un appel à la fonction division ne pourra se faire que dans une bloc try...catch vu que cette fonction peut lever des exceptions.

```
#include <iostream>

using namespace std;

int division(int a,int b) // division entière de a par b.
{
    if(b==0)
        throw range_error("Division par zéro");
    else
        return a/b;
}

int main(int argc, char **argv)
{
    try
    {
        Division(1,0); // Division par zéro
    }
    catch( range_error const& e) //On rattrape l'exception standard
    {
        Cerr << "ERREUR:" << e.what()<< endl;//affiche la description de l'erreur
    }
    return 0;
}
```

4.3. La classe exception

C++ comprend un ensemble de classes d'exception pour gérer automatiquement les erreurs de division par zéro, les erreurs d'E/S, les trans-typages incorrects et de nombreuses autres conditions d'exception. Toutes les classes d'exception dérivent d'une classe mère appelée **exception**.

En outre, cette classe propose une méthode nommée **what** qui permet de fournir à l'utilisateur un message le renseignant sur la nature de l'erreur levée.

Pour l'utiliser, il faut inclure le fichier d'en-tête **exception**. Pour lever l'exception, il suffit alors d'utiliser le code générique ci-dessus :

```
catch (std::exception &e) {
    cerr << "ERREUR :" << e.what() << endl;
}
```

ci-dessous quelques exemples de classes dérivées de la classe de base **std::exception** :

Logic_error	Description
Domain_error	Erreur de domaine mathématique
Invalid_argument	Argument invalide passé à une fonction
Length_error	Taille invalide
Out_of_range	Erreur d'indice de tableau
Runtime_error	Description
range_error	Erreur de domaine.
overflow_error	Erreur d'overflow
underflow_error	Erreur d'underflow
Bad_alloc //échec d'allocation mémoire par new	
Bad_cast //échec de l'opérateur dynamic_cast	

5. Références

- MOC EPLF : Introduction à la programmation orientée objet (en C++) de Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit.
- Programmation C/C++ : Initiation à la POO de Thierry VAIRA.

