
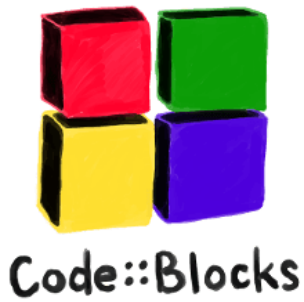


| Extrait du référentiel : BTS CIEL option A (Informatique et Réseaux) | | Niveau(x) |
|---|---|---|
| C08 CODER | Langages de développement, de description, de création d'API et les IDE associés | 4 |

## Objectifs du cours :

- Création de classes et d'objets :
  - déclaration d'une classe
  - instance d'une classe
- 1er programme
- Déclaration des méthodes
- Surdéfinition des méthodes
  - règle de recherche d'une fonction surdéfinie
  - les arguments par défaut
- Les constructeurs
- Le destructeur
- Exercices

Dans ce cours, la plupart des codes sont donnés sans modularité.

# CRÉATION DE CLASSES ET D'OBJETS

## DÉCLARATION D'UNE CLASSE

Nous allons créer une classe qui permet de caractériser des points sur un repère orthonormé. Les points auront des coordonnées en (x, y) et il sera possible de modifier leurs valeurs.

Pour déclarer une classe, on utilise le mot clé **class**.

```
class Point
{
```

```
 private : // Membres privés (ici les attributs)
 int x ;
 int y ;

 public : // Membres publics (ici les méthodes)
 void initialise(int, int) ; // Pour initialiser les coordonnées du point
 void afficher() ;
} ;
```

Toute classe sera déclarée dans un fichier d'entête « **.h** » portant le même nom.
La déclaration d'une classe n'occupe aucun espace mémoire (comme pour la structure en langage C), tant qu'il n'y a pas de variable déclarée.
Une classe ressemble de très près à une structure en langage C.
Si aucune visibilité n'est indiquée, alors par défaut tous les membres seront de visibilité « private ».
La classe fait partie des types hétérogènes.

## INSTANCE D'UNE CLASSE

**Exemple :**

```
int main()
{
 point a ; // Création d'un objet a de type point
 a.initialise(3,7) ; /* Appel de la méthode pour initialiser les coordonnées
 du point */
 return 0 ;
}
```

## 1er PROGRAMME

Sous CodeBlocks, il vous faudra :

Créez un fichier d'entête (**.h**) par classe pour la déclaration du contenu de la classe, le fichier portera le même nom que la classe ;
Créez un fichier source (**.cpp**) par classe pour la définition de la classe, le fichier portera le même nom que la classe ;
Créez un fichier source (**.cpp**) pour la fonction `main()`, exemples : main.cpp, source.cpp, appli.cpp.

**Le fichier : Point.h**

```
#ifndef POINT_H
#define POINT_H
class Point
{
 private :
 int x, y ;
 public :
```

```
 void initialise(int, int) ; // Une méthode de classe
 void affiche() ;
 } ;
#endif
```

**Le fichier : Point.cpp**

```
#include <iostream>
#include "Point.h"

using namespace std ;

void point::initialise(int abs, int ord)
{
 x = abs ;
 y = ord ;
}
 void point::affiche()
{
 cout << "Les coordonnees du point sont : " << x << ',' << y << " \n " ;
}
```

Le fichier d'E/S standard **stdio.h** n'est plus utilisé en C++ (sauf si besoin).
Il est remplacé par le fichier **iostream** (Input Output STREAM), il utilise de nouvelles méthodes du langage C++ : **cout** (qui remplace le **printf( )**) par exemple.
Utilisation de l'espace de nom nommé **std**.
Chaque méthode de classe est précédée du nom de sa classe puis de deux points « **::** »
Le symbole « **::** » est **l'opérateur de résolution de portée**. Il sert à modifier la portée d'un identificateur.

**Le fichier : main.cpp**

```
#include "Point.h"

int main()
{
 point c ; // Un objet automatique, c'est une variable locale
 c.initialise(3,7) ;
 c.affiche() ; // Utilisation d'une méthode par l'objet
 return 0 ;
}
```

L'utilisation (l'appel) des méthodes **initialise( )** et **affiche( )** s'effectue à l'aide du symbole « **.** » qui fait le lien entre l'objet et la méthode de classe appelée (dans le cas d'un objet de type automatique).

Résultat :

```
Les coordonnees du point sont : 3,7
```

```
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

## DÉCLARATION DES MÉTHODES

Toute **fonction membre** (**méthode de classe**) utilisée dans un fichier source doit obligatoirement avoir fait l'objet d'un prototypage dans sa classe, sauf pour la fonction `main()`.

Les fonctions ordinaires sont déclarées en dehors de toute déclaration de classe. On utilisera un fichier d'en-tête à part pour déclarer ces fonctions (ou alors on peut les déclarer dans le fichier source contenant le `main()`). La définition des fonctions ordinaires pourra se faire dans un fichier source à part ou alors dans le fichier source contenant le `main()`.

## SURDÉFINITION DES MÉTHODES

Le langage C++ permet de **surdéfinir** ou encore **surcharger** les méthodes de classes.

Qu'est-ce que c'est ?

C'est tout simplement permettre que deux (voir plus) méthodes portent le même nom.

C'est quoi l'intérêt ?

Le principal intérêt est de pouvoir « multiplier » les constructeurs de classes afin de personnaliser la création et l'initialisation des objets. Une méthode de classe (sauf le destructeur) ou une fonction ordinaire peut être surdéfinie en langage C++.

Comment différencier ces méthodes portant un même nom ?

Par sa liste des arguments reçus en paramètres.
Une méthode surdéfinie porte le même nom qu'une autre méthode mais avec une liste de paramètres différents. C'est simplement la liste et la nature des paramètres qui permettent de différencier ces différentes méthodes surdéfinies lors d'un appel.
Dans ce cas, lors de l'appel, le compilateur effectue le choix de la « bonne fonction » en tenant compte de la nature des arguments effectifs.

## RÈGLE DE RECHERCHE D'UNE FONCTION SURDÉFINIE

Le compilateur regarde en premier, sur le niveau « exact » si une fonction correspond. Sinon, il change « le niveau de recherche ».

Le niveau « exact » correspond exactement au même type entre les arguments passés et les types des paramètres attendus. Sinon, le compilateur peut effectuer des conversions implicites de type (int → float) par exemple.

Exemple :

```
float LeCalcul(int, float, char*) ; // Prototypage méthode 1
float LeCalcul(float, float, char*) ; // Prototypage méthode 2
int a = 5 ;
```

```
float x, f = 3.2 ;
char *mess = "Bonjour" ;
x = LeCalcul(a, f, mess) ;
```

Ici, c'est la méthode 1 qui convient pour l'appel (niveau exact). Si la méthode 1 n'existait pas, la méthode 2 pourrait convenir (correspondance inexacte).

Dès qu'une correspondance est trouvée, la recherche sur un même niveau s'arrête. Si plusieurs fonctions conviennent au même niveau de correspondance (exacte ou inexacte), il y a erreur de compilation due à l'ambiguïté rencontrée. Si aucune fonction ne convient à aucun niveau, il y a également erreur de compilation. Les ambiguïtés peuvent essentiellement survenir lors de l'utilisation d'arguments par défaut.

## LES ARGUMENTS PAR DÉFAUT

En langage C, il est indispensable que l'appel d'une fonction contienne autant d'arguments que la fonction en attend effectivement.
Dans **une déclaration** d'une fonction (prototype), il est possible en C++ de prévoir pour un ou plusieurs arguments des valeurs par défaut. Un argument par défaut est une valeur par défaut attribuée à une variable si cette dernière n'est pas spécifiée lors de l'appel de la méthode.

Exemple de déclaration d'une méthode :

```
float fct(char, int = 10, float = 0.0) ; // Deux arguments par défaut
```

Les valeurs par défaut seront utilisées dans le cas d'un appel avec un nombre d'arguments inférieur à celui prévu.

**fct('a') ;** sera équivalent à **fct('a', 10, 0.0)**
**fct('x', 12) ;** sera équivalent à **fct('x', 12, 0.0)** /* il faut respecter l'ordre */
**fct( ) ;** // appel illégal car il n'y a pas de valeur par défaut pour le premier argument.

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.
Les valeurs par défaut peuvent être constituées de n'importe quelle expression, pourvu que cette dernière soit déclarée avant son utilisation.

Exemple :

```
float x ;
int n ;
........
void fct(float = n * x + 1.5) ;
```

# LES CONSTRUCTEURS

## PRÉSENTATION

Toute instance ou objet d'une classe possède une durée de vie. On peut distinguer trois phases :

- la naissance de l'objet ;
- la vie de l'objet ;
- la mort de l'objet.

Lors de la **création d'un objet**, il peut s'avérer utile d'affecter des valeurs aux attributs de ce dernier. Il faut initialiser les données liées à l'objet. N'initialiser que les données nécessaires, celles dont on connaît la valeur lors de la création de l'objet. C'est le rôle du **constructeur de classe**.

Intérêt : plus besoin d'utiliser et d'appeler une méthode **`initialise()`**.

Il s'agit d'une méthode qui est appelée automatiquement à chaque création d'un objet.
**L'existence d'un constructeur (ou de plusieurs constructeurs) nous garantit que l'objet sera toujours initialisé.**

Le constructeur se reconnaît car il porte le même nom que la classe où les instances seront créées.
Il se peut que des objets, instances d'une même classe, n'aient pas forcément les mêmes données à initialiser. Dans ce cas, le concepteur du programme écrira autant de fonctions membre constructeurs que nécessaire.

## RÈGLES DU CONSTRUCTEUR

Toute classe possède un constructeur, c'est **le constructeur implicite**. Il permet de construire un objet sans attributs. **Il n'est pas codé explicitement, donc non visible.**
Un constructeur ne renvoie pas de valeur (non typé). **Le mot clé `void` ne doit pas apparaître.**
Un constructeur porte le même nom que sa classe. C'est simplement la liste et la nature des paramètres qui permettent de différencier les différents constructeurs. **Surdéfinition de constructeurs dans ce cas.**
À partir du moment où une classe possède un constructeur explicite avec arguments, il n'est plus possible de créer un objet sans fournir les arguments requis. **Dans ce cas, le constructeur par défaut ou encore appelé constructeur « vide » (sans paramètre) doit figurer dans la déclaration de la classe pour la création d'un objet sans attribut.**
**Le constructeur est appelé après la création de l'objet pour initialiser ce dernier.**

## INITIALISATION D'OBJETS

Lors de la création de l'objet, il est possible également d'affecter des valeurs aux attributs de ce dernier en utilisant une seule et même instruction.
Il existe plusieurs possibilités.

Exemple : constructeurs paramétrés

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

class point
{
private :
int x, y ;
```

```
public :
point(int) ; // Constructeur 1
point(int, int) ; // Constructeur 2
} ;
point::point(int abs)
{
x = abs ;
y = 0 ;
}
point::point(int abs, int ord)
{
x = abs ;
y = ord ;
}
int main( )
{
point a(3) ; // Appel constructeur 1
point b = 3 ; // Identique pour l'objet b
point c = a ; // Création de l'objet c puis initialisé avec l'objet a
point d(a) ; // Identique pour l'objet d
point e(1, 5) ; // Constructeur 2 appelé
point f = point(1, 5) ; // Démarche identique
return 0 ;
}
```

***Question***

Quel est le constructeur appelé pour les objets c et d ?

……

Exemple : sans constructeur

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

class point
{
private :
int x, y ;

public :
void affiche() ;
} ;
void point::affiche()
{
cout << "Je suis un point de coordonnees : " << x << '\t' << y << endl ;
}

int main()
{
```

```
point a ;
a.affiche() ;
return 0 ;
}
```

***Question***

Quel sera l'affichage en sortie console ?

……

Exemple : constructeur par défaut

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

class point
{
private :
int x, y ;

public :
point() ;
point(int) ;
point(int, int) ;
void affiche() ;
} ;
point::point()
{
cout << "Je suis le constructeur par defaut" << endl ;
}
point::point(int abs)
{
cout << "Je suis le constructeur 2" << endl ;
x = y = abs ;
}
point::point(int abs, int ord)
{
cout << "Je suis le constructeur 3" << endl ;
x = abs ;
y = ord ;
}
void point::affiche()
{
cout << "Je suis un point de coordonnees :" << x << '\t' << y << endl ;
}

int main()
{
point a ;
point b(2) ;
```

```
point c(3,7) ;
a.affiche() ;
b.affiche() ;
c.affiche() ;
b = c ;
b.affiche() ;
return 0 ;
}
```

Résultat :

Le fait qu'il y ait la présence des constructeurs 2 et 3 (deux constructeurs explicites avec paramètres), la création d'un objet sans attributs **a**, nécessite la présence d'un constructeur explicite vide (le constructeur par défaut) sinon une erreur de compilation survient.

## EXERCICES

Soit le code ci-dessous :

**Le fichier : Csosie.h**

```
#ifndef CSOSIE_H
#define CSOSIE_H
#include <iostream>

class CSosie
{
private :

public :
CSosie(int) ;
CSosie(double) ;
CSosie(int, int = 15) ;
} ;
#endif
```

**Le fichier : main.cpp**

```
#include «CSosie.h»

void fct(int, int = 12 ) ;

int main()
{
```

```
int n = 5 ;
double x = 2.5 ;
int m = 10, p = 20 ;

CSosie objet1(n) ;
CSosie objet2(x) ;
CSosie objet3(m, n) ;
fct(m , p) ;
fct(m) ;
fct() ;
return 0 ;
}
```

**Le fichier : CSosie.cpp**

```
#include «CSosie.h»

using namespace std ;

CSosie::CSosie(int a)
{
cout << "sosie numero 1 a = " << a << "\n" ;
}
CSosie::CSosie(double a)
{
cout << "sosie numero 2 a = " << a << "\n" ;
}
CSosie::CSosie(int a, int b)
{
cout << "sosie numero 3 a = " << a << "\tb = " << b << "\n" ;
}
void fct(int a, int b)
{
cout << " premier argument : " << a << "\n" ;
cout << " second argument : " << b << "\n" ;
}
```

***Question 1***

***Commentez*** les lignes du code.

***Question 2***

Quelles sont les lignes de code qui ne permettent pas la compilation du programme ?

***Question 3***

Les lignes de code interrompant la compilation du programme sont supprimées. Quel est le résultat de l'exécution du programme ?

Résultat :

# LE DESTRUCTEUR

## PRÉSENTATION

Lors de la mort (ou destruction) d'un objet, une fonction membre particulière est automatiquement appelée : c'est le **destructeur**. **Elle présentera surtout un intérêt dans le cas d'objets effectuant des allocations dynamiques d'emplacements. Ce destructeur permettra la destruction de l'allocation mémoire allouée (destruction des composantes de l'objet).**

## RÈGLES D'UTILISATION

Une classe ne peut disposer que d'**un seul destructeur**, alors qu'elle peut avoir à sa disposition plusieurs constructeurs. Donc un destructeur ne peut pas être surdéfini.
Le destructeur explicite porte le même nom que sa classe, précédé du symbole tilde (**~**).
Le destructeur a pour rôle de détruire les zones mémoires dynamiques allouées par le constructeur lors de la création d'un objet.
Le destructeur peut être implicite ou explicite, comme pour le constructeur.
Le destructeur n'accepte pas d'arguments en paramètre (contrairement au constructeur), et ne renvoie pas d'arguments. Le mot clé **`void`** n'apparaît pas.

Un destructeur ne détruit pas d'objet. Il est appelé avant la destruction de l'objet dans une fonction (pour un objet dynamique par exemple) ou à la fin d'une boucle ou d'une fonction pour un objet automatique.

Exemple :

```
#include <iostream>
#include <string.h>

using namespace std ;

class point
{
private :
int x, y ;
char *couleurPoint ; // Attribut dynamique
```

```
public :
point(int, int, char*) ;
~point() ;
void affiche() ;
} ;

point::point(int abs, int ord, char *couleur)
{
int nbreCar ;
x = abs ;
y = ord ;

cout << "Je suis le constructeur du point" << endl ;
nbreCar = strlen(couleur) ;
couleurPoint = new char[nbreCar+1] ; // Allocation dynamique
strcpy(couleurPoint, couleur) ;
}

point::~point()
{
cout << "Je suis le destructeur du point" ;
delete[] couleurPoint ; // Libération zone mémoire de la couleur
}
void point::affiche()
{
cout << "Je suis un point de coordonnees :" << x << '\t' << y << '\t' <<
"et de couleur : " << couleurPoint << endl ;
}

int main()
{
point a(3, 7, "Bleu") ;
a.affiche() ;
return 0 ;
}
```

***Question***

Quel est le résultat de l'exécution du programme ?