Module R3.04

Qualité de Développement

Cours 1 – Notions de Base en C++

Objectifs

- (Re)parler un peu de conception objet
- Voir les bases du langage C++
- Acquérir de bonnes pratiques de base en C++
- (Re)découvrir quelques (2 ou 3 !) Design Patterns
- (Re)découvrir la notion de tests unitaires (Google Test)

Modalités

- 6 séances de cours (1 x 1h)
- 6 semaines de TP (2 x 1h30)
- Evaluations :
  - QCM « papier » (lundi 30 septembre)
  - Examen Machine (semaine 21-27 octobre)

Pourquoi C++ ? C'est un standard industriel

(source : http://www.lextrait.com/vincent/implementations.html)

- **OS** (C et C++) : windows, Google Chrome OS, MacOS
- **OS mobiles** : Symbian OS, RIM Blacberry OS 4.x
- **Graphics** : Windows UI, MacOS UI, KDE
- **Bureautique** : MS office, Sun Open Office, Adobe Acrobat
- **SGBD** : Oracle, MYSQL, MS SQL server
- **Web navigateurs** : MS IE, Firefox, Safari, Chrome, Opera
- **Web serveurs** : MS IIS
- **Java Virtual Machine** !
- **Photo** : Photoshop, Gimp
- **Web** : Google, Paypal, Amazon, facebook
- **Compilateurs** : MS Visual C++, VB, C#, Perl
- **Moteurs 3D** : DirectX, Ogre 3D
- **CD/DVD** : Nero
- **Xmedia** : Winamp, Windows media player, iPod software
- **Peer 2 Peer** : eMule, µtorrent
- **GPS** : TomTom, Garmin
- **Jeux vidéos !**

C++ - Historique

- Conçu en 1982 par **Bjarne Stroustrup** (AT&T Bell Labs, berceau d'Unix et C)
- **Thèse** : ajouter au langage C le concept de classes, comme en **Simula 67, Smalltalk**
- **Intérêt** : bénéficier des avantages de la Programmation Orientée Objet tout en conservant les performances du langage C et ses bibliothèques nombreuses et variées
- Langage ancien mais toujours utilisé, normalisé (ISO) et évoluant régulièrement
  Versions « officielles » : C++98, C++03, C++11, C++14, C++17, C++20, (C++23)

« C makes it easy to shoot yourself in the foot…
C++ makes it harder, but when you do, it blows your whole leg off ! »

Module R3.04
Chapitre 1

Différences entre C et C++

1.1 Les types références (1)

- **En C**, il n'y a qu'un seul mode de passage des paramètres : le **passage par valeur**.
- **Conséquence** : si une procédure a besoin de modifier la valeur de l'un de ses paramètres, il faut lui passer un pointeur sur l'objet à modifier,
- Exemple : permuter le contenu de deux variables entières en C :

```c
/* en langage C */
void permuteEntiers(int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}

int main() {
    int i = 5;
    int j = 6;
    permuteEntiers(&i, &j);
    return 0;
}
```

L'utilisation des « * » et des « & » n'est vraiment pas intuitive ☹

1.1 Les types références (2)

- **En C++**, on va pouvoir utiliser des paramètres référence « & »
- **Attention** : la syntaxe reste trompeuse car le signe de référence « & » est le même que celui de **l'opérateur de calcul d'adresse**

```
// Et maintenant en C++
void permuteEntiers(int &a, int &b) {
    int c = a;
    a = b;
    b = c;
}

int main() {
    int i = 5;
    int j = 6;
    permuteEntiers(i, j);
    return 0;
}
```

Une référence peut être considérée comme étant un alias :
- a est un alias de i
- b est un alias de j

1.1 Les types références (3)

- On peut dire qu'une **variable / un paramètre référence** est un **pointeur automatiquement déréférencé** quand on le manipule
- Une **variable / un paramètre référence** « pointe » toujours sur la même variable au cours de sa vie :
  - Sur la variable qu'on doit **obligatoirement** lui affecter au moment de sa déclaration, dans le cas d'une **variable référence** :

```
int i = 5;
int & ref_i = i;  // ref_i sera toujours une référence à i !
```

  - Sur la variable qu'on lui associe au moment de l'appel de la fonction dans le cas d'un **paramètre référence** :

```
void permuteEntiers(int & a, int & b) {
  // Lors de l'appel de cette procédure ci-dessous
  // a référencera toujours i et b référencera toujours j
  ...
}
...
int i = 5;  int j = 6;
permuteEntiers(i,j);
```

1.2 Surcharge de fonctions (1)

- **Objectif** : pouvoir nommer de la même façon des fonctions de prototypes différents.
- **Exemple** : un système de gestion d'interface à base de fenêtres.
  Chaque fenêtre est repérée par 2 systèmes différents : Un identifiant (un entier) et son nom (une chaîne).
  On veut écrire 2 fonctions qui permettent de récupérer une fenêtre soit par son identifiant, soit par son nom.
- **Dans un langage sans surcharge**, il faut utiliser 2 noms de fonctions différents, par exemple :

```
Window* getWindowParIdent(unsigned int id);
Window* getWindowParNom  (const string & nom);
```

- **En C++,** on peut utiliser le même nom pour les 2 fonctions si elles ont des paramètres différents (en nombre et/ou en type) :

```
Window* getWindow (unsigned int id);
Window* getWindow (const string & nom);
```

1.2 Surcharge de fonctions (2)

- **Comment le compilateur fait-il pour s'y retrouver ?**
  Le nom « interne » de la fonction, aussi appelé **signature**, est composé du nom de la fonction et de la liste des types des paramètres. Le type du résultat d'une fonction n'est pas pris en compte
- Ainsi, à la compilation, lors d'un appel de fonction, le compilateur établit quelle est la bonne fonction à appeler en regardant les types des paramètres effectifs
- **Attention : il est impossible d'avoir des fonctions qui ne diffèrent que par le type de leur résultat** puisque le compilateur ne prend pas en compte le type du résultat d'une fonction dans sa signature

```
// Ceci serait refusé par le compilateur car les 2 fonctions
//  ont la même signature : somme(int,int)
int   somme (int x, int y) { return x+y;         }
float somme (int x, int y) { return (float)x+y; }
```

1.3 Valeurs par défaut des paramètres (1)

- Lorsqu'une fonction est appelée souvent et que certains de paramètres ont, la plupart du temps, la même valeur, il est agréable de pouvoir définir une **valeur par défaut** pour ces paramètres afin de ne pas avoir besoin de la passer systématiquement lors d'un appel.
- La valeur par défaut est définie dans la spécification de la fonction (fichier **.h**) mais ne doit pas être rappelée dans implémentation (fichier **.cpp**).

```
int f1(int x, int y = 4) {
// y vaudra 4 par défaut
  return x + y;
}

int f2(int x = 3, int y = 4) {
// x vaudra 3 par défaut
// y vaudra 4 par défaut
  return x + y;
}

int main() {
  // appels possibles
  int a;
  a = f1(10, 23); // a = 33
  a = f1(10);     // a = 14
  a = f2(10, 23); // a = 33
  a = f2(10);     // a = 14
  a = f2();       // a = 7
  return 0;
}
```

1. Valeurs par défaut des paramètres (2)

- Le nombre des paramètres avec des valeurs par défaut n'est pas limité : tous les paramètres peuvent prendre une valeur par défaut.
- **Attention**, seuls les derniers paramètres de la fonction peuvent avoir une valeur par défaut.
- Vous ne pouvez omettre que les valeurs des derniers paramètres. Cela tient au mécanisme de passage des paramètres sur la pile lors de l'appel des fonctions…
- Par exemple, le code suivant est illégal :

```
int f3 (int x = 3, int y) {
  // x ayant une valeur par défaut,
  // les paramètres suivants
  // (selon l'ordre de lecture gauche->droite)
  // devraient aussi avoir une valeur par défaut
  return x + y;
}
```

1.4 Les constantes de C++ (1)

- En C, on ne disposait pas de « vraies » constantes, il fallait utiliser le préprocesseur du compilateur (sorte de traitement de texte qui fait des rechercher/remplacer sur votre code avant de le compiler). Exemple :

```
#define TAILLE 5
```

- En C++, on utilisera plutôt le qualificateur **`const`** :

```
const int    TAILLE = 5;
const double PI = 3.14;
const char   MESSAGE[] = "Hello World";
```

- La règle à retenir : **`const`** s'applique à ce qui se trouve à sa gauche et s'il n'y a rien à sa gauche, il s'applique à ce qui est à sa droite

1.4 Les constantes de C++ (2)

- **const** peut être utilisé pour qualifier des **pointeurs** :

Déclaration	«Décodage»	Pointeur modifiable ? (p++)	Valeur pointée modifiable ? ((*p)++)
`const int *p`	p est un pointeur sur un entier constant	oui	non
`int * const p`	p est un pointeur constant sur un entier	non	oui
`const int * const p`	p est un pointeur constant sur un entier constant	non	non

- **const** peut être utilisé pour qualifier une **référence** :
  `const int & p` // La variable entière référencée par p ne peut être modifiée
- **const** peut enfin être utilisé pour qualifier une **méthode** afin de préciser au compilateur que cette méthode ne modifiera pas les attributs de l'objet auquel elle s'appliquera.

1.5 Lecture et écriture sur le terminal

- En C : bibliothèque **stdio**, utilisable en C++
  (mais à éviter autant que possible…)

- En C++ : bibliothèque **iostream,** orientée objets (description fournie plus tard)

- **iostream** fournit 2 objets particuliers :
  - `cout` pour écrire sur l'écran du terminal,
  - `cin` pour lire sur le clavier du terminal.

- Ces deux objets, comme tout ce que propose la bibliothèque standard (std), sont déclarés dans un **espace de nom** appelé **std**

1.5 Objet `stream`

- **`cin`** et **`cout`** sont des objets de la classe **`stream`**
- Le mot **`stream`** peut être traduit par **flot (flux) de données**.
- Un **`stream`** est un objet :
  - qui représente un **support d'informations** sur lequel on peut **lire** ou bien **écrire** ou bien **lire et écrire**.
  - qui possède toutes les méthodes pour lire ou écrire sur ce support.
- Un support d'informations peut être:
  - l'**écran** ou le **clavier** d'un terminal ,
  - un **fichier**
  - une zone de **mémoire** (chaîne de caractères).

1.5 Utilisation de **cout** , Opérateur **<<**

```
#include <iostream>
#include <iomanip>
using namespace std; // pour ne pas avoir à préfixer avec std::
int main() {
  int    code          = 147;
  char   numTel[]      = "0476284528";
  float  prix          = 12.5;

  cout << code << numTel << prix << endl;

  // sans "using namespace"  on aurait écrit :
  // std::cout << code << numTel << prix << std::endl;
  return 0;
}
```

- L'opérateur **<<** **envoie** une valeur de **type prédéfini** (**int**, **short**, **float**, **char**) sur un flux d'écriture, ici **cout** (la console)
- Le manipulateur **endl** représente une fin de ligne (‘\n’)
- Le résultat de l’opérateur **<<** est son opérande gauche, ici **cout**

1.5 Utilisation de `cin`, Opérateur `>>`

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

int main() {
  int          code;
  char         numTel[11];
  float        prix;

  cin >> code >> numTel >> prix;

  return 0;
}
```

- L'opérateur `>>` **extrait (lit)** une valeur de type prédéfini (`int`, `short`, `float`, `char`) d'un flux de lecture, ici `cin`
- Le résultat de l'opérateur `>>` est son opérande gauche, ici `cin`

1.5 Manipulateurs

- Fonctions ou opérateurs qui permettent de modifier le statut d'un **flux**, par exemple le **format** et la **base** de lecture ou d'écriture des nombres.

- Le statut d'un ***stream*** est défini par un mot d'état (une suite de bits). Ce mot est découpé en plusieurs champs, chaque champ étant composé d'un certain nombre de bits.

- Les manipulateurs permettent de modifier ces bits en utilisant la syntaxe d'une entrée ou d'une sortie, c'est-à-dire en utilisant les opérateurs **>>** ou **<<**

Manipulateur	Utilisation	Action
`endl`	Sortie	insère un saut de ligne et vide le tampon
`ends`	Sortie	insère une fin de chaîne (caractère '\0')
`flush`	Sortie	vide le tampon
`left/right/internal`	Sortie	cadrage gauche, droite, remplissage après signe ou base
`scientific/fixed`	Sortie	notation scientifique ou fixe
`showbase/noshowbase`	Sortie	montre/cache la base
`showpoint/noshowpoint`	Sortie	montre/cache le point décimal
`showpos/noshowpos`	Sortie	montre/cache le signe + devant les nombres positifs
`uppercase/nouppercase`	Sortie	affichages en majusules/minuscules
`boolalpha/noboolalpha`	Entrée/Sortie	représentation des booléens (0/1 ou false/true)
`dec`	Entrée/Sortie	conversion décimale
`hex`	Entrée/Sortie	conversion hexadécimale
`oct`	Entrée/Sortie	conversion octale
`resetiosflags(long)`	Entrée/Sortie	remet à zéro les bits de formattage désignés
`setbase(int)`	Entrée/Sortie	fixe la base de conversion
`setfill(int)`	Entrée/Sortie	définit le caractère de remplissage
`setiosflags(long)`	Entrée/Sortie	met à un les bites de formattage désignés
`setprecision(int)`	Entrée/Sortie	définit la précision des flottants
`setw(int)`	Entrée/Sortie	définit la largeur d'affichage (gabarit)
`skipws/noskipws`	Entrée	saute ou pas les espaces
`ws`	Entrée	saute les espaces

1.5 Exemple : manipulateurs en sortie

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

int main() {
  cout << setprecision(3) << 110.55555 << endl;
  // 110.556

  cout << scientific << 110.5 << endl;
  // 1.105e+02

  cout << setw(10) << fixed << setfill((int)'-') << 110.5 << endl;
  // -----110.5

  cout << setfill((int)'*') << "abc" << endl;
  // *******abc

  return 0;
}
```

1.6. La classe `string`

- En C++ on utilisera de préférence la classe **`string`** pour représenter des chaînes de caractères.
- L'utilisation de cette classe est en effet bien plus simple que les tableaux de caractères du langage C et leur bibliothèque **<string.h>**

```
#include <string>
#include <iostream>
// …

std::string s;          // Déclaration d'une chaîne s, vide pour l'instant ("")
s="Hello";              // s prend pour valeur "Hello"
s=s+" World";           // Concaténation : s vaut "Hello World"
s[s.size()-1]='D';      // Modification du dernier caractère : s vaut "Hello WorlD"
s[0]='h';               // Modification du premier caractère : s vaut "hello WorlD"
std::cin >> s;          // Lecture d'une chaîne sur un flux d'entrée :
                        //   la lecture s'arrête dès qu'un séparateur est rencontré
getline(std::cin, s); // Lecture d'une chaîne sur un flux d'entrée :
                        //   la lecture s'arrête en fin de ligne
std::cout << s;         // Ecriture de la chaîne s sur un flux de sortie
if (s=="oui") {         // On peut utiliser tous les opérateurs de comparaison usuels
:
                        //   == != > >= < <=
// …
```

Module R3.04
Chapitre 2

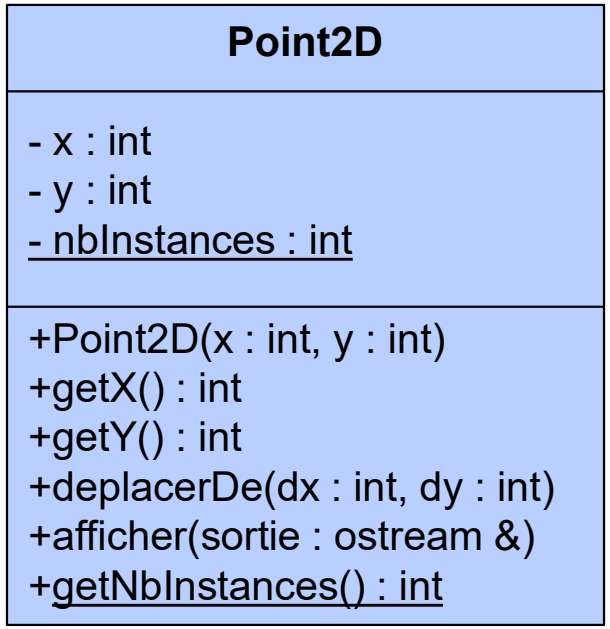
Classe, Objet

2.1 Représentation des classes (1)

- En C++, une classe est représentée par une forme particulière de structure (**struct**) qui rassemble à la fois les attributs et les méthodes.

- Le transparent suivant illustre la « traduction » d'un diagramme de classe UML en C++.
  - On veut représenter la notion de **Point2D**, caractérisé par ses coordonnées **x** et **y** qui seront donc 2 **attributs d'instance**
  - On aura aussi un **attribut de classe** « à vocation pédagogique », **nbInstances**, qui servira à compter le nombre d'objets instanciés dans cette classe…

2.1 Représentation des classes (2)



UML -> C++ : Non trivial ! Injection nécessaire de vos connaissances en C++

```cpp
// Fichier Point2D.h
#ifndef POINT2D_H
#define POINT2D_H
#include <iostream>

class Point2D {
public: // membres publics
    // méthodes d'instance
    Point2D(int x=X_DEFAUT, int y=Y_DEFAUT); //constructeur
    ~Point2D(); // destructeur
    int getX() const;
    int getY() const;
    void deplacerDe(int dx, int dy);
    void afficher(std::ostream & sortie = std::cout) const;
    // méthode de classe
    static int getNbInstances();
private: // membres privés
    // attributs d'instance, préfixés par m_ par convention
    int m_x;
    int m_y;
    // attributs de classe
    static int nbInstances;
    // constantes de classe
    static const int X_DEFAUT;
    static const int Y_DEFAUT;
};
#endif
```

2.1 Représentation des classes (3)

```cpp
// Fichier Point2D.cpp
#include "Point2D.h"
#include <iostream>
using namespace std;

// Méthodes d'instance
Point2D::Point2D(int x, int y)
: m_x(x), m_y(y) {
    // une instance de plus
    nbInstances++;
}

Point2D::~Point2D() {
   // une instance de moins
    nbInstances--;
}

int Point2D::getX() const {
    // "this->" est facultatif
    return this->m_x;
}

int Point2D::getY() const {
    return this->m_y;
}
```

```cpp
void Point2D::deplacerDe(int dx, int dy) {
  this->m_x += dx;
  this->m_y += dy;
}

void Point2D::afficher(ostream & sortie) const {
    sortie << "x=" << this->getX()
           << " y=" << this->getY()
           << endl;
}

// Méthodes de classe
int Point2D::getNbInstances() {
    // pas de this ici
    // c'est une méthode de classe
    return nbInstances;
}

// Attributs de classe
int Point2D::nbInstances = 0;

// Constantes de classe
const int Point2D::X_DEFAUT = 0;
const int Point2D::Y_DEFAUT = 0;
```

2.1 Représentation des classes (4)

- La déclaration commence par `class` et est encadrée par une paire d'accolades.
  L'accolade finale doit **impérativement** être suivie d'un point virgule.

- Les membres déclarés après le mot clef `public` forment l'interface de la classe alors que ceux suivant `private` sont invisibles en dehors des méthodes de la classe

- Les membres précédés du mot clef `static` sont des **membres de classe**. Les autres sont des **membres d'instance**.

- L'ordre de déclaration des méthodes et des attributs est libre, mais on met en général la section `public` en premier

- Il peut y avoir plusieurs sections `public`, `private`, `protected`, …

2.1 Représentation des classes (5)

- La déclaration des attributs est semblable à la déclaration d'une variable (**mais sans initialisation possible**)
- La déclaration d'une méthode est analogue au prototype d'une fonction C
- Chaque méthode d'instance d'une classe **MaClasse** possède un argument **implicite** : l'objet sur lequel elle sera invoquée. Il est accessible *via* un **pointeur** particulier appelé `this`, de type **MaClasse ***
- Lors de l'implémentation des méthodes, il est nécessaire de préfixer le nom de la méthode par le nom de la classe suivi de `::`
- Le **constructeur** est une méthode particulière qui porte le même nom que la classe et dont le but est d'initialiser TOUS les attributs d'instance. Elle est appelée juste après l'allocation mémoire d'un nouvel objet.
- Certaines méthodes d'instance (getX(), getY(), …) sont déclarées constantes à l'aide du mot clé `const`. Cela signifie que leur code n'affecte en aucune manière la valeur des attributs d'instance de l'objet sur lequel elles sont invoquées. Le pointeur implicite `this`, pour ces méthodes, est de type `const` **MaClasse ***

2.2 Organisation du code

- Il est fortement conseillé d'avoir 2 fichiers sources pour chaque classe :
  - un fichier header (extension .h, .H, .hxx, .hh ou .hpp) où l'on place la **déclaration** (ou **spécification**) de la classe
  - un fichier de **définition** (ou **implémentation**) des méthodes et des variables de classe (extension .C, .cpp, .cc ou .cxx).
- Afin d'éviter les inclusions multiples, on place des directives de compilation « `#ifndef/#ifdef` » comme dans l'exemple suivant

Fichier Point2D.h

```
#ifndef POINT2D_H
#define POINT2D_H
class Point2D {
   // Spécification de la classe
}; // Ne pas oublier le ";"  !!!
#endif
```

Fichier Point2D.cpp

```
#include "Point2D.h"
// autres inclusions nécessaires

// Définitions des var. de classe

// Définitions des méthodes
```

2.3 Modificateurs d'accès

- Les mots clés `public` et `private` sont des **modificateurs d'accès**. Leur portée s'étend jusqu'au prochain modificateur.
- Le modificateur par défaut est `private`.
- Les membres déclarés `private` ne sont visibles que par les méthodes de la classe elle même. En revanche, tout membre déclaré `public` aura une visibilité « universelle ».
- **Le respect du principe d'encapsulation** impose donc que :
  - Tout attribut d'instance ou de classe sera déclaré `private` (abstraction des données)
  - Toute méthode non nécessaire à l'utilisateur sera déclarée `private`
  - Toute méthode que vous souhaitez rendre disponible à l'utilisateur, et qui définit donc l'interface de la classe sera déclarée `public`
- Un troisième modificateur d'accès, `protected`, sera utilisé dans le cadre de l'héritage

2.4 Déclaration et définition des membres de classes

- Les **membres de classe** sont déclarés avec le mot clé `static`. Contrairement aux modificateurs d'accès, `static` n'a d'effet que sur une seule ligne de déclaration.
- Particularité du C++ : la déclaration d'un attribut `static` ne lui alloue pas de mémoire. Il faudra redéclarer cet attribut ailleurs, dans le fichier de définitions (.cpp), pour que de la mémoire lui soit allouée.
  - C'est le rôle de la définition « `int Point2D::nbInstances=0;` » qui déclare l'attribut de classe `nbInstances` et lui affecte la valeur 0.

- Rappelons que :
  - **Les méthodes d'instance** peuvent très bien utiliser les attributs de classe en plus de leurs attributs d'instance
  - **Les méthodes de classe** ne peuvent par contre utiliser que les attributs de classe (*d'ailleurs les attributs de quelle instance pourraient-elles bien utiliser ??* `this` *n'est* ***pas disponible*** *dans une méthode de classe…*)

2.5 Résolution de portée

- L'opérateur particulier `::` est appelé « **opérateur de résolution de portée** ». Il sert à désigner à quelle classe appartient une méthode ou un attribut.
- Cet opérateur est nécessaire car le langage C++ n'oblige pas à respecter la règle de séparation de l'implémentation dans différents fichiers.
- Aussi, la définition de chaque méthode doit être précédée du nom de la classe à laquelle elle se rattache et de l'opérateur de résolution de portée.
- Par exemple, supposons que dans le même fichier, vous vouliez implémenter la méthode **methodeA** de la classe **A** ainsi que la méthode **methodeB** de la classe **B**, alors, vous auriez à spécifier :

```
void A::methodeA(…) { // code… }
void B::methodeB(…) { // code… }
```

2.6 Les méthodes `inline` (1)

- Les méthodes **`inline`** sont traitées par le compilateur comme les macro du langage C (effets de bord en moins) : le compilateur remplacera chaque appel à une fonction **`inline`** par le « développement » du code de cette méthode.

- Il en résulte de meilleurs temps d'exécution mais aussi un accroissement de la taille du code produit.

- Aussi, ces méthodes doivent être limitées à quelques instructions (sous peine d'accroître excessivement la taille de l'exécutable). Ce sera typiquement le cas des accesseurs et des constructeurs.

- Les méthodes **`inline`** doivent impérativement être implémentées dans le fichier de spécification de la classe (.h).

2.6 Les méthodes `inline` (2)

Fichier Point2D.h

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D {
public:
  inline Point2D(int x, int y);         // Constructeur inline (cf plus bas)
  inline int getX() const;              // Getter inline (cf plus bas)
  inline int getY() const;              // Getter inline (cf plus bas)
  void deplacerDe(int dX, int dY);      // Méthode NON inline
                                        //   => implémentée dans Point2D.cpp
private:
  int m_x;
  int m_y;
  static int nbInstances;
};

// Définitions inline du constructeur et des accesseurs
inline Point2D::Point2D(int x, int y) : m_x(x), m_y(y) { nbInstances++; }
inline int Point2D::getX() const { return m_x; }
inline int Point2D::getY() const { return m_y; }

#endif
```

Module R3.04
Chapitre 3

Cycle de vie des Objets

3 Cycle de vie des objets

- La vie d'un objet se résume à trois épisodes fondamentaux : sa création, son utilisation puis sa destruction.
- Nous allons voir cela à travers les points suivants :
  - 3.1 Création d'objets : les constructeurs
  - 3.2 Instanciation
    - 3.2.1 Les classes d'allocation
    - 3.2.2 Création d'instances simples
  - 3.3 Invocation de méthodes
    - 3.3.1 Appel de méthodes d'instance
    - 3.3.2 Appel de méthodes de classe
    - 3.3.3 Accès aux attributs
  - 3.4 Appels de méthodes et objets constants
  - 3.5 Mort des objets
  - 3.6 Création et destruction de tableaux

3.1 Création d'objets : constructeurs (1)

- L'initialisation des objets repose sur des méthodes spéciales nommées *constructeurs* qui sont appelées lors de « l'**instanciation** » d'un objet (quand de la mémoire lui est allouée)
- Les constructeurs portent le même nom que la classe.
- Pour la classe **Point2D** déjà vue, le constructeur était :
  `Point2D::Point2D(int x=0, int y=0);`
- Notez au passage qu'on utilise la possibilité offerte par le C++ de donner des valeurs par défaut aux arguments des fonctions.

- Ce constructeur pourrait s'implémenter de la façon suivante :

```
Point2D::Point2D(int x, int y) {
  this->m_x = x;
  this->m_y = y;
  nbInstances++; // L'attribut de classe nbInstance sert à
                 // compter le nombre d'instances de la classe
}
```

3.1 Création d'objets : constructeurs (2)

- Ce code peut être réécrit en utilisant une **liste d'initialisation** :

```
Point2D::Point2D(int x = 0, int y = 0)
: m_x(x), m_y(y) {
  nbInstances++;
}
```

- La syntaxe de cette liste d'initialisation est :
  : attribut(valeur_initialisation) {, attribut(valeur_initialisation)}^{n}
- Les attributs doivent être initialisés dans l'ordre de leur déclaration
- **valeur_initialisation** peut être n'importe quelle expression C++ valide comprenant des opérateurs, des appels de fonctions ou de méthodes sur tout autre objet que celui en cours de création.
- **Attention : certains attributs ne peuvent pas être initialisés dans le corps du constructeur et devront donc obligatoirement l'être dans la liste d'initialisation** : il s'agit des attributs qui sont eux-mêmes des objets ou des références sur objets, dans le cadre des relations **d'agrégation** et **d'association**.

3.2 Instanciation des objets (1)

- La création d'objets se fait en appelant le constructeur
- La syntaxe diffère selon la méthode d'allocation de l'objet.
- Il y a 3 méthodes d'allocation :

  - **Allocation statique : dans la zone données (data)**

  - Objets déclarés en variable globale, c'est à dire en dehors de toute fonction ou méthode.

  - Objets explicitement mis en allocation statique dans une méthode ou fonction à l'aide du modificateur d'allocation `static`.
    Rappelons que dans ce cas, ils sont initialisés une seule fois et gardent leur valeur lors des appels successifs de la fonction / méthode.
    Créés dès le début du programme, ils sont détruits automatiquement à la fin de l'exécution de celui-ci (sans que vous ayez à vous en occuper).
    Il n'existe qu'un seul cas où ces objets statiques ne sont pas détruits : l'arrêt du programme par la fonction `abort`.

3.2 Instanciation des objets (2)

- **Allocation automatique : dans la pile (stack)**

  Tout objet créé sur la pile est dit à **allocation automatique**.
  Cela concerne donc les variables temporaires dans une méthode ainsi que les objets renvoyés par une méthode qui sont momentanément stockés sur la pile.
  Un objet automatique est créé à chaque lancement de la méthode à l'intérieur de laquelle il est déclaré et détruit automatiquement, sans que vous ayez à vous en soucier, dès la sortie de la méthode.

- **Allocation dynamique : dans le tas (heap)**

  Il s'agit des objets créés sur le **tas** et auxquels vous ne pouvez accéder qu'à travers un pointeur.

  Ils sont créés par un appel à l'opérateur `new` et doivent être détruits *explicitement* par un appel à l'opérateur `delete`.

3.2 Instanciation des objets (3)

- La création d'une instance **automatique** ou **statique** est simple et répond à la syntaxe suivante :

```
T nomObjet(<paramètres d'un constructeur>);
```

ou

```
T nomObjet; // appel du const. par défaut s'il existe
```

- Dans le cas d'une instance dynamique, il faut commencer par déclarer un pointeur, puis appeler `new` suivi du nom de la classe
  Cela peut se faire sur une ou plusieurs lignes de code.
  En une ligne :

```
T* pointeurObjet = new T(<paramètres d'un constructeur>);
```

ou

```
T* pointeurObjet = new T; // const. par défaut s'il existe
```

3.2 Instanciation des objets (4)

- Exemple

```
#include "Point2D.h"
Point2D p1; // Objet statique avec constructeur par défaut

int main() {
  Point2D p1; // Utilise le constructeur Poin2Dt::Point2D(int, int)
              //   avec arguments constr. par défaut
  Point2D p2(10, 20); // Crée un nouveau Point2D avec x=10 et y=20
  Point2D *p3;        // Déclaration d'un pointeur sur objet Point2D

  p3 = new Point2D(20, 30); // Instanciation de l'objet dynamique
                            //   avec arguments explicites
  Point2D * p4 = new Point2D; // Instanciation de l'objet dynamique
                              //    avec arguments constr. par défaut

  delete p3; // On n'oublie pas de libérer la mémoire !
  delete p4; // Idem
  return 0;
}
```

3.3 Appel de méthodes (2)

- **3.3.1 Appel de méthodes d'instance**

Pour les objets à classe d'allocation statique ou automatique, la syntaxe d'appel est la suivante :

```
objet.methodeInstance(<paramètres>);
```

Par exemple, appliquons les opérations suivantes à l'objet de classe **Point2D** nommé p1 tel qu'il a été instancié précédemment :

- afficher son abscisse
- le déplacer de 10 unités sur x et 20 unités sur y relativement à sa position actuelle
- afficher son ordonnée

```
cout << p1.getX() << endl;
p1.deplacerDe(10,20);
cout << p1.getY() << endl;
```

3.3 Appel de méthodes (2)

- Pour un objet dynamique manipulé *via* un pointeur, Il faut remplacer le "`.`" par une flèche "`->`".
- Ainsi, si l'on reprend l'exemple précédent avec l'objet alloué dynamiquement et pointé par `p4`, on obtient :

```
cout << p4->getX() << endl;
p4->deplacerDe(10,20);
cout << p4->getY() << endl;
```

- On peut également **déréférencer** le pointeur avec l'opérateur « `*` » et utiliser ensuite le « `.` » :

```
cout << (*p4).getX() << endl;
(*p4).deplacerDe(10,20);
cout << (*p4).getY() << endl;
```

3.3 Appel de méthodes (2)

- **3.3.2 Appel de méthodes de classe**

Les méthodes de classe ne sont évidemment pas invoquées à travers un objet mais à l'aide du nom de la classe.
La syntaxe générale est la suivante :

```
T::methodeDeClasse(<paramètres>);
```

Ainsi, si vous souhaitez afficher le nombre d'instances de **`Point2D`** dans votre programme en invoquant la méthode de classe **`getNbInstance()`**, vous devrez écrire :

```
cout << "Nombre de points : "
     << Point2D::getNbInstances()
     << endl;
```

3.4 Appel de méthodes et objets constants (1)

- Les règles suivantes s'appliquent aux objets constants :
  - On déclare un objet constant avec le modificateur `const`
  - **On ne peut appliquer que des *méthodes constantes* sur un objet *constant***
  - **Un objet passé en paramètre sous forme de référence constante est évidemment considéré comme constant**

- **Qu'est-ce qu'est une méthode constante ?**
  C'est une méthode qui ne modifie aucun des attributs de l'instance à laquelle elle s'applique.
- Dans une classe **T**, le paramètre implicite `this` d'une méthode d'instance constante est de type `const T *`
- Il est impératif de déclarer constante toute méthode qui ne modifie pas l'état des objets auxquels elle s'applique. Vous pourrez ainsi passer en paramètre à une procédure, par référence constante, les objets qui ne doivent pas être modifiés dans cette procédure.
- Le prototype d'une méthode constante est suivi du mot clé `const` qui fait partie de la signature.

3.4 Appel de méthodes et objets constants (2)

```
class Chose { // Une classe avec des méthodes constantes ou pas
  // …
          void methConst(…) const;
          void methNonConst(…);
};
```

```
void uneProcedure(const Chose & uneChose) {

  uneChose.methConst (...);    // OK, methConst est une méthode
                               //   constante qui peut être
                               //   invoquée sur un objet constant

  uneChose.methNonConst(...); // NON ! methNonConst est une
                               //   méthode non constante qui ne
                               //   peut pas être invoquée sur un
                               //   objet constant

}
```

3.5 Destruction des objets (1)

- La « mort » des objets s'accompagne de l'appel automatique d'une autre méthode spéciale : le **destructeur**.
- Contrairement au constructeur, le destructeur ne peut être surchargé car son appel est toujours implicite.
- Le destructeur de la classe **MaClasse** est :
  `MaClasse::~MaClasse()`
  On reconnait ici le symbole **~** associe à l'opérateur binaire NON
- Le destructeur ne renvoie rien ; il n'admet aucun paramètre
- Le but du destructeur est de réaliser toutes les opérations de « nettoyage » nécessaires à la destruction correcte d'un objet.
  Par exemple, si le constructeur ou une autre méthode de la classe réalise des allocations dynamiques, le destructeur veillera à libérer la mémoire afin d'éviter que celle-ci ne soit perdue (fuite mémoire).
  Autre cas : si un objet détient des ressources systèmes critiques, le destructeur devra « rendre » ces ressources avant la libération de l'objet.

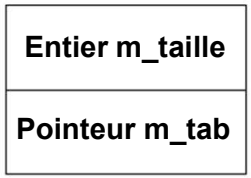
3.5 Destruction des objets (2)

- Le processus de destruction se passe en deux fois :
  - Appel du destructeur
  - Libération de l'espace mémoire occupé par les données membres de l'objet

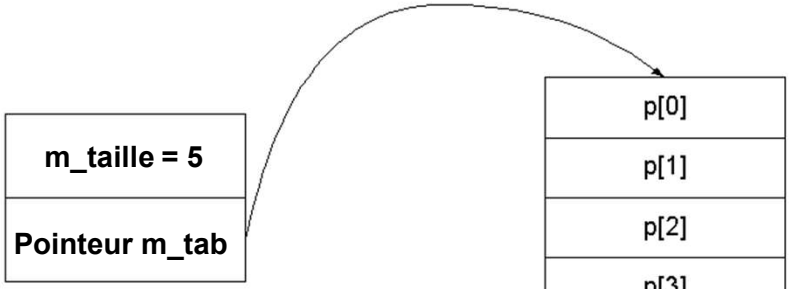
```
class Tableau { // Tableau d'entier « dynamique »
public:
  Tableau(unsigned int laTailleDuTableau = 5)
    : m_taille(laTailleDuTableau) {
    if (m_taille) m_tab = new int [m_taille];
    else m_tab = nullptr;
  }
  ~Tableau(void) { // le destructeur libère la mémoire dynamique
     if (m_taille) delete [] m_tab; //delete [] libère un tableau
  }
private:
  unsigned int m_taille; // taille du tableau
  int* m_tab; // pointeur sur le tableau alloué dynamiquement
};
```

3.5 Destruction des objets (3)

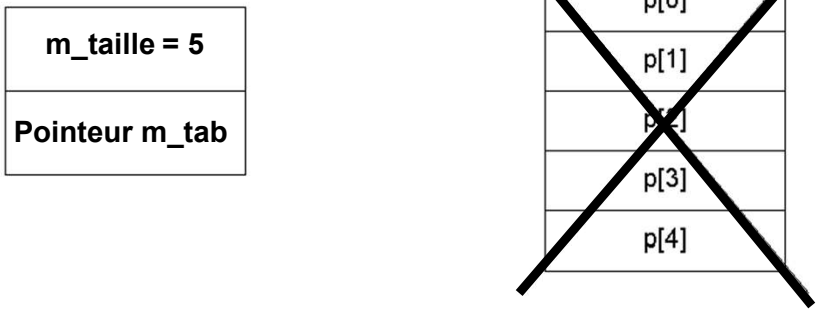
Construction phase 1 : Allocation de l'espace attribué aux données membres de l'objet ; ces dernières sont non initialisées



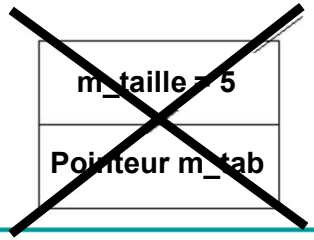
Construction phase 2 : Constructeur => initialisation des données membres et allocation du tableau



Destruction phase 1 : Destructeur => le tableau est désalloué



Destruction phase 2 : Désallocation de l'espace alloué aux données membres



3.6 Création/Dest. de tableaux (1)

- L'utilisation de **tableaux d'objets** nécessite **un constructeur par défaut dans la classe des objets**, c'est-à-dire un constructeur pouvant ne prendre aucun paramètre lors de son invocation.

  Un tel constructeur peut :
  - Ne prendre effectivement aucun paramètre
  - Ou avoir des paramètres qui ont tous une valeur par défaut

- Cette contrainte découle de la syntaxe de déclaration des tableaux qui ne permet pas de passer des paramètres au constructeur :

```
T tableau[tailleTableau];
```

- Un constructeur par défaut est donc indispensable dans la classe T.

3.6 Création/Dest. de tableaux (2)

Type de tableau	Construction des objets	Destruction	Remarques
Tableau statique d'instances statiques	`T tableau[TAILLE];`	`Automatique`	Constr. par défaut obligatoire. Tout est détruit automatiquement

Tableau statique d'instances dynamiques	`T* tableau[TAILLE];`
	`for(int i=0; i<TAILLE;i++)`
	`  tableau[i]=new T(params);`
		`for(int i=0;i<TAILLE;i++)`
		`  delete tableau[i];`
			Le constr. étant appelé explicitement, n'importe lequel convient.
			Attention, chaque objet devra être détruit individuellement

Tableau dynamique d'instances statiques	`T* tableau;` `tableau=new T[TAILLE]`	`delete [] tableau;`	Constr. par défaut obligatoire Seul le tableau doit être détruit explicitement

Tableau dynamique d'instances dynamiques	`T** tableau;`
	`tableau = new T*[TAILLE];`
	`for(int i=0;i<TAILLE; i++)`
	`  tableau[i]=new T(params);`
		`for(int i=0;i<TAILLE;i++)`
		`  delete *tableau[i];`
		`delete [] tableau;`
			Les éléments doivent être détruits individuellement, avant de détruire le tableau

Module R3.04
Chapitre 4

Références sur objets
Constructeur de recopie
Affectation

4.1 Références à des objets (1)

- **Les références** à des objets sont à considérer dans plusieurs cas :
  - Passage d'objets en paramètres
  - Fonctions ou méthodes qui renvoient des objets
  - Agrégation « externe » (vu plus tard)
- **Passage d'objets par référence**
  - **Dans le cas du passage de paramètre par valeur**, il y a recopie du paramètre sur la pile. Dans le cas d'un objet, cette recopie nécessite un appel à un constructeur particulier, le **constructeur de recopie** (vu plus tard). Cette opération peut être coûteuse. L'objet temporaire créé sur la pile sera détruit à la fin, ce qui implique un appel à un destructeur.
  - **Avec un passage par référence**, il n'y a pas de recopie. Du point de vue du code généré, passer un objet par référence ou un pointeur sur cet objet (par valeur) revient au même.
    Les références sont une facilité apportée au programmeur afin de limiter les erreurs. En outre, une référence est toujours associée à une variable, elle ne peut pas être non initialisée à la manière d'un pointeur qui peut ne pointer sur rien…

4.1 Références à des objets (2)

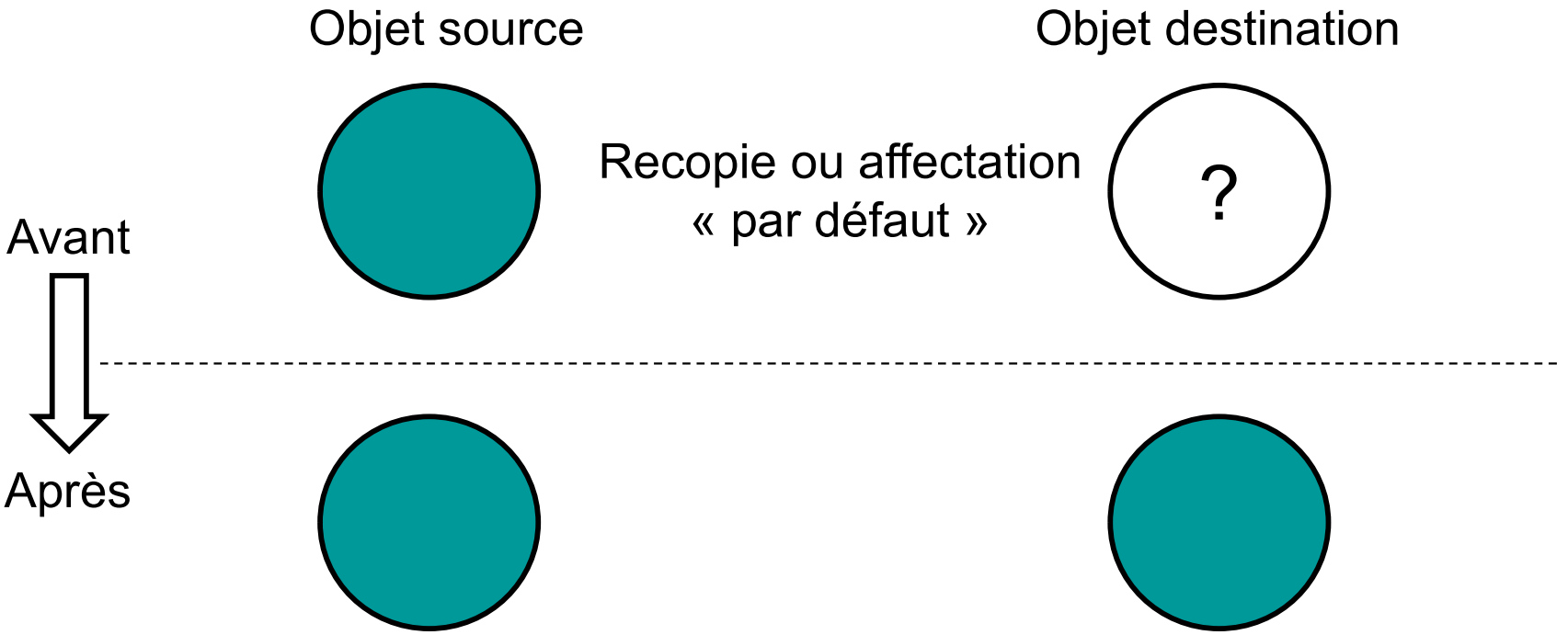
- **Il faut privilégier les passages par référence pour les objets :**

```
class Fenetre {
  // …
  void afficheImage( const Image & image ){
    // …
}
// …
int main() {
  Fenetre uneFenetre(...);
  Image uneImage(...);
  uneFenetre.afficheImage(uneImage); // passage par référence
  return 0;
}
```

- Au moment de l'appel, rien ne permet de distinguer que l'on a passé l'objet par référence plutôt que par valeur
- A l'intérieur de la fonction, on manipule la référence comme on manipulerait un objet

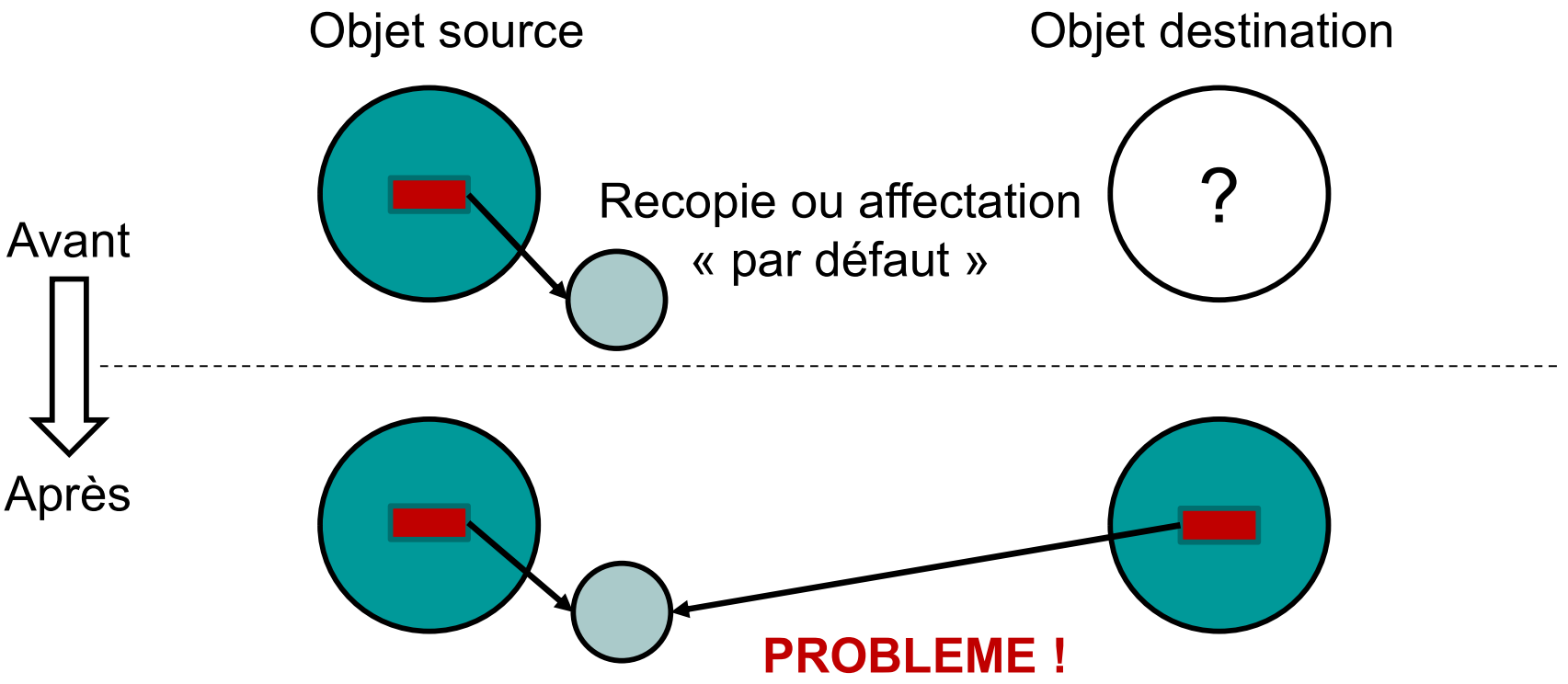
4.2 Clonage d'objets : recopie et affectation (1)

- Lorsqu'un objet est passé en paramètre par valeur (avec recopie donc), ou lorsque vous réalisez une affectation avec deux objets de même classe, un **clonage** d'objet est réalisé
- Par défaut, le clonage va consister à dupliquer toute la zone mémoire allouée à l'objet source dans celle de l'objet destination



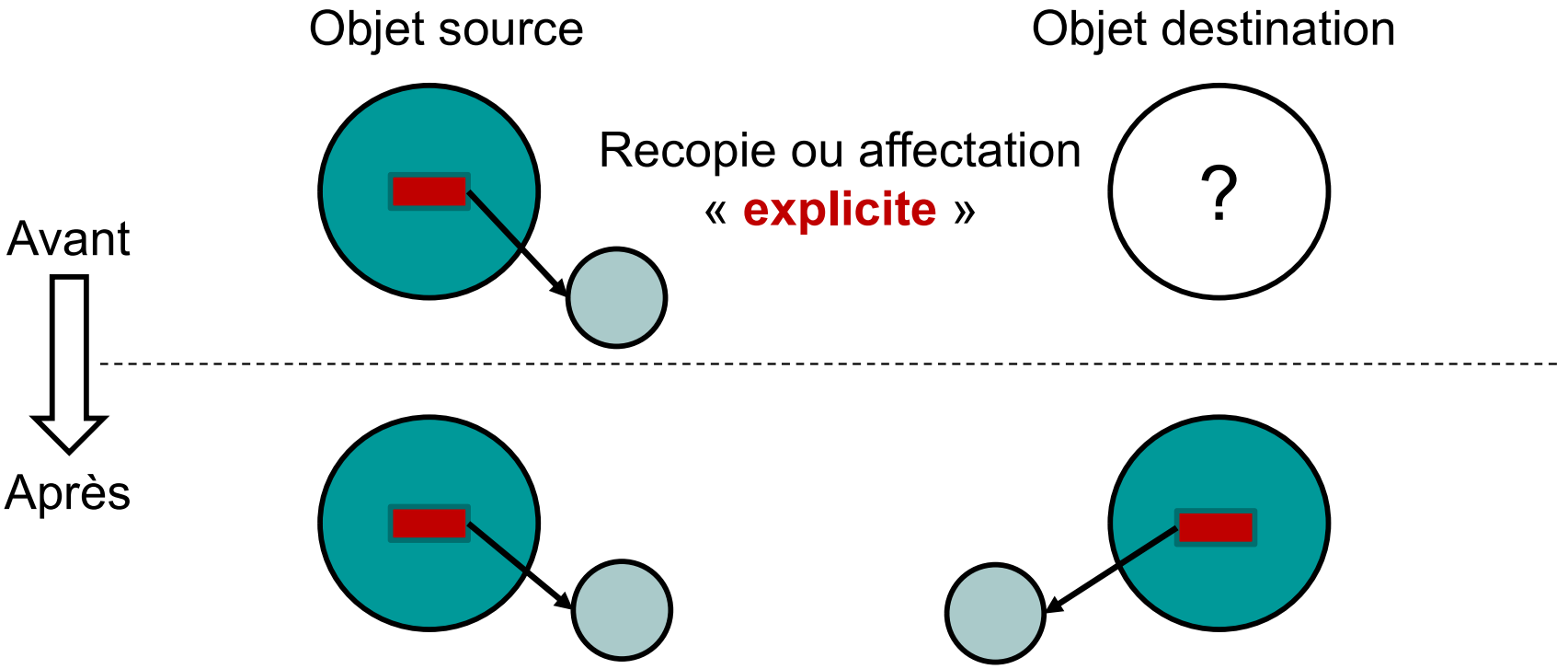
4.2 Clonage d'objets : recopie et affectation (2)

- Si l'objet source contient des pointeurs sur des données extérieures à sa zone mémoire et qui lui sont propres, **ces données ne seront pas dupliquées mais partagées avec l'objet destination, ce qui peut conduire à des incohérences :**



4.2 Clonage d'objets : recopie et affectation (3)

- => Lorsque l'objet source contient de pointeurs sur des données extérieures à sa zone mémoire et qui lui sont propres, **il faut programmer explicitement la recopie et l'affectation pour dupliquer les données extérieures**
- **Ce clonage correct se programme dans le constructeur de recopie et l'opérateur = d'une classe**



4.2 Clonage d'objets : recopie et affectation (2)

4.2.1 Le constructeur par recopie

- Le constructeur par recopie est très important car il permet d'initialiser un objet par clonage d'un autre. En particulier, le constructeur par recopie est invoqué dès lors que l'on passe un **objet en paramètre par valeur**
- Le prototype du constructeur par recopie d'une classe T est le suivant :

```
T::T(const T & o);
```

- Il est extrêmement important de passer l'objet recopié par référence sous peine d'entrainer une récursivité infinie !
- Si vous ne fournissez pas explicitement de constructeur par recopie, le compilateur en génère automatiquement un pour vous.
  Vous pouvez aussi déclarer : `T::T(const T & o) = default;`
- Il faut par impérativement fournir un constructeur de recopie dès lors que le clonage d'un objet par recopie binaire brute peut entraîner un dysfonctionnement de votre classe, en particulier si votre classe :
  - Utilise de la mémoire dynamique
  - Utilise de ressources systèmes (fichiers, sockets, etc ...)

4.2 Clonage d'objets : recopie et affectation (3)

4.2.2 Opérateur d'affectation

- L'opérateur d'affectation et le constructeur de recopie sont très proches dans le sens où ils sont requis dans les mêmes circonstances et qu'ils effectuent la même opération : cloner un objet dans un autre.
- **Différence fondamentale** : L'opérateur d'affectation écrase le contenu d'un objet déjà existant et donc déjà construit auparavant.
  Cela signifie que, dans la majorité des cas, il faudra commencer par "nettoyer" l'objet, à la manière du destructeur, avant d'effectuer l'opération de clonage.
- Le prototype de l'opérateur d'affectation d'une classe **T** est le suivant :

```
T & operator = (const T & objetT);
```

  - C'est une méthode de la classe **T** dont le nom est **`operator =`**
  - Elle renvoie en résultat une référence sur l'objet qui a été affecté (**`*this`**)
  - Elle reçoit en paramètre par **référence constante** un objet de classe **T**
- On peut aussi « garder » le comportement par défaut en écrivant :

```
T & operator = (const T & objetT) = default;
```

4.2 Clonage d'objets : recopie et affectation (4)

4.2.3 Initialisation ou Affectation

Il est parfois délicat de savoir si l'on a affaire à une affectation ou une construction par recopie car la syntaxe du signe "=" peut être trompeuse.

Règle simple :

Toute opération d'initialisation ou d'affectation dans une déclaration est l'affaire d'un constructeur

Instruction	Description	Méthode mise en jeu
`T t1;`	Initialisation par le constructeur par défaut	`T::T(void);`
`T t2(params);`	Initialisation par un constructeur quelconque	`T::T(liste params);`
`T t3(t1);`	Initialisation par le constructeur de recopie	`T::T(const T&);`
`T t5=t1`	Initialisation par le constructeur de recopie Cette ligne est à remplacer par « `T t5(t1);` » qui fait exactement la même chose mais est moins ambigüe du point de vue de la syntaxe.	`T::T(const T&);`
`t5=t2`	Affectation à l'aide de l'opérateur d'affectation	`T & T::operator=(const T&);`

4.3. Forme Canonique de Coplien

- Une classe T est dite sous **forme canonique de Coplien** si elle fournit les éléments suivants :

Prototype	Fonctionnalité
`T::T()`	Constructeur par défaut
`T::T(const T&)`	Constructeur par recopie
`T& T::operator=(const T&)`	Opérateur d'affectation
`T::~T()`	Destructeur

- Si ces éléments sont codés correctement, alors l'utilisation de cette classe vis à vis de la mémoire est « sûr ».
- **Dès qu'une classe utilise de la mémoire dynamique ou des ressources critiques, il est indispensable de la mettre sous forme canonique de Coplien.**

4.3. Exemple Coplien(1) : classe Image

- Exemple : classe pour représenter une image de taille `hauteurxlargeur` (en niveaux de gris). Le contenu est un tableau alloué dynamiquement.

```
#ifndef IMAGE_H
#define IMAGE_H
#include <string>

class Image {
public:
    Image(const std::string & nom = "une image",
          int hauteur=0, int largeur=0);    // Const. par défaut
    Image(const Image & image);             // Constr. de recopie
    Image & operator=(const Image & image); // Opérateur =
    ~Image();                               // Destructeur
private:
    std::string m_nom;
    int m_hauteur;
    int m_largeur;
    unsigned char * m_contenu; // pointeur sur un tableau alloué dynamiquement
};

#endif //IMAGE_H
```

4.3. Exemple Coplien(2) : Constructeur

- Si l'image n'est pas vide, le constructeur alloue dynamiquement (new) un tableau de taille `hauteur x largeur`
- L'attribut `m_contenu` pointe sur le tableau alloué

```
#include "Image.h"
#include <cstring>
#include <string>
using namespace std;

Image::Image(const std::string& nom, int hauteur, int largeur)
  : m_nom(nom),
    m_hauteur(hauteur),
    m_largeur(largeur) {

  if (this-> m_largeur > 0 && this-> m_hauteur > 0) {
    // Allocation dynamique du contenu de *this
    this->m_contenu = new unsigned char[this-> m_hauteur * this-> m_largeur];
  }
  else {
    this->m_contenu=nullptr;
  }
}
```

4.3. Exemple Coplien(3) : Constr. Recopie

- Le constructeur de recopie doit :
  - Allouer dynamiquement un tableau de même taille que celui de l'image à copier
  - Recopier le contenu de l'image à copier dans le tableau alloué précédemment

```
Image::Image(const Image & imageCopiee)
    : m_nom(imageCopiee.m_nom),
      m_hauteur(imageCopiee.m_hauteur),
      m_largeur(imageCopiee.m_largeur) {

    // Si le contenu de imageCopiee n'est pas vide, on le clone
    if (imageCopiee.m_contenu != nullptr) {
      // Allocation dynamique du contenu de *this
      this->m_contenu = new unsigned char[this->m_hauteur * this->m_largeur];
      // recopie du contenu de imageCopiee dans le contenu de *this
      memcpy(this->m_contenu, imageCopiee.m_contenu,
             this->m_hauteur * this->m_largeur);
    } else {
        this->m_contenu=nullptr;
    }
}
```

4.3. Exemple Coplien(4) : Opérateur =

- L'opérateur = doit :
  - Libérer l'espace occupé par le contenu actuel de l'image (*this)
  - Allouer un nouveau tableau de même taille que celui de l'image à copier
  - Recopier le contenu de l'image à copier dans le nouveau tableau
  - Renvoyer l'image (*this) qui vient d'être modifée par l'affectation

```
Image& Image::operator=(const Image& imageCopiee) {
    // On libère l'espace occupé dans le tas par le contenu de *this
    delete [] this->m_contenu; // [] nécessaire car m_contenu pointe sur un TABLEAU
    // Puis on recopie le contenu de imageCopiee dans *this
    this->m_nom     = imageCopiee.m_nom;
    this->m_hauteur = imageCopiee.m_hauteur;
    this->m_largeur = imageCopiee.m_largeur;
    if (imageCopiee.m_contenu != nullptr) {
        // Allocation dynamique du contenu de *this
        this->m_contenu = new unsigned char[m_hauteur * m_largeur];
        // recopie du contenu de imageCopiee dans le contenu de *this
        memcpy(this->m_contenu, imageCopiee.m_contenu,
               this->m_hauteur * this->m_largeur);
    } else {
        this->m_contenu=nullptr;
    }
    return *this; // On renvoie l'objet (*this), celui à gauche de l'opérateur =
}
```

4.3. Exemple Coplien(5) : Destructeur

- Le destructeur - qui est automatiquement exécuté juste avant qu'un objet `Image` soit supprimé - doit libérer le tableau (pointé par `m_contenu`) qui a été alloué dynamiquement (`new`) pendant le cycle de vie de l'image.

- Sans ce `delete`, il y aurait une fuite de mémoire (memory leak) !

```
Image::~Image() {

    // On libère l'espace occupé dans le tas par le contenu de *this

    delete [] this->m_contenu; // delete [] nécessaire ici car m_contenu pointe
                               //  sur un TABLEAU alloué dynamiquement
}
```