MOOC semaine 2

Constructeur (de copie) et destructeur

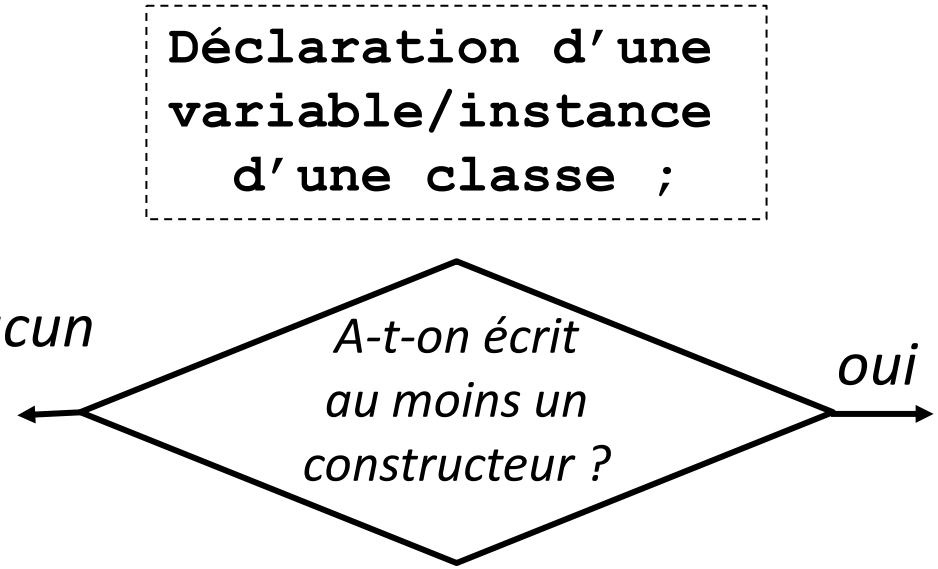
Buts:

- Gérer l'initialisation des instances
- Quand faut-il un constructeur de copie ?
- Quand faut-il s'intéresser au destructeur ?

Plan:

- Les bonnes pratiques en matière de constructeur
- Exemple pour lequel il faut un constructeur de copie
- Exemple pour lequel il faut un destructeur

Le(s) constructeur(s) => initialisation



Le compilateur C++ met en place un ***constructeur par défaut par défaut***

Un *attribut* défini par une **classe** est initialisé avec son **constructeur par défaut** (**string** est initialisé avec la chaîne vide).

Un attribut ayant un type de base (int, char, bool,double …) obtient **un motif binaire quelconque** => source de bugs.

c'est une mauvaise pratique
de ne pas définir de constructeur

Bonne pratique !

Le compilateur C++ exploite le constructeur approprié pour la déclaration :

Le ***constructeur par défaut*** est utilisé pour une déclaration sans aucune transmission de valeur d'initialisation

La **surcharge** permet de définir autant de constructeurs qu'on a défini de scénarios d'utilisation de ce type de variable

Un constructeur peut en appeler un autre dans la **liste d'initialisation** = section deux-points après les paramètres et avant le bloc

rect.h

```
class Rect
{
public:
     Rect(double width,double height);
     double surf() const;
private:
      double width;
      double height;
};
```

rect.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

Rect::Rect(double width, double height)
       :width(width),height(height) {}

double Rect::surf() const{
       return width*height;
}
```

prog.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

int main()
{
     Rect r;
     cout << r.surf() << endl;
     return 0;
}
```

Quizz1 : ce code

A: s'execute et affiche une valeur quelconque à cause du constructeur par défaut par défaut

B: s'execute et affiche 0.

C: ne compile pas à cause de la liste d'initialisation

D: s'execute et affiche une valeur quelconque car les paramètres masquent les attributs

E: ne compile pas car pas de constructeur par défaut

rect.h

```
class Rect
{
public:
    Rect(double width =0.,
         double height=0.);
    double surf() const;
private:
    double width  =1.;
    double height =1.;
};
```

rect.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

Rect::Rect(double width, double height)
      :width(width),height(height) {}

double Rect::surf() const{
      return width*height;
}
```

prog.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

int main()
{
    Rect r;
    cout << r.surf() << endl;
    return 0;
}
```

Quizz2 : ce code

A: ne compile pas car pas de constructeur par défaut

B: s'execute et affiche 0.

C: s'execute et affiche 1.

Constructeurs et manipulateurs : même combat

Ces deux familles de méthodes ont accès aux attributs et peuvent les modifier.

Il est donc particulièrement important ***d'effectuer les vérifications de domaine autorisé*** des paramètres avant d'affecter des valeurs aux attributs.

Pourquoi faudrait-il faire ces vérifications en double ?

Le principe de ré-utilisation recommande d'éviter de dupliquer le code ;
Il suffit de mettre au point une unique fonction qui est ré-utilisée plusieurs fois.

C'est pourquoi le C++ autorise d'appeler des **manipulateurs** dans le corps des **constructeur** (il n'y a pas de contraintes sur l'ordre des déclarations des méthodes, les manipulateurs peuvent être déclarées après les constructeurs).

Le constructeur de copie
= initialiser en copiant la valeur d'une autre variable

Le compilateur C++ met en place un **constructeur de copie par défaut**
qui copie la valeur des attributs terme à terme

Inutile d'écrire soi-même un constructeur de copie si c'est pour réaliser ce type de copie appelée **copie superficielle**

Par contre, dès qu'on effectue de **l'allocation dynamique de mémoire** (sem1 Topic 11)
la copie superficielle n'est généralement pas ce qu'on veut obtenir pour la copie;
elle duplique la valeur d'un pointeur vers un bloc alloué dynamiquement qui lui n'est pas dupliqués (slide suivant).

Le constructeur de copie

Limitation de la copie superficielle dans un contexte d'allocation dynamique

```
Class_avec_alloc_dyn x(50); // attribut pointeur vers bloc de 50 octets
```

Une allocation dynamique de 50 octets est effectuée pour x
Ce qui mémorise l'adresse du bloc de 50 octets dans l'attribut p


```
Class_avec_alloc_dyn x(50); // attribut pointeur vers bloc de 50 octets
Class_avec_alloc_dyn y(x);  // copie superficielle de l'attribut
```

Le bloc de 50 octets est partagé par x et par y !

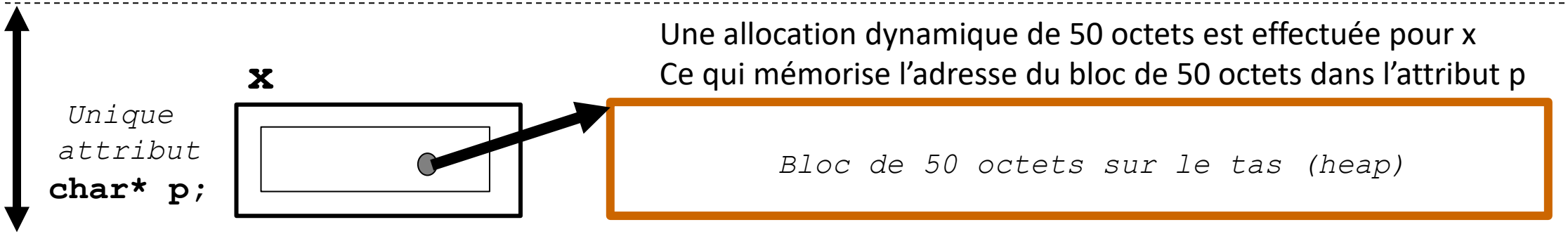
Le destructeur (il n'y en a qu'un au maximum par classe)

En général pas nécessaire

Exception: à écrire en cas de ressource partagée / d'allocation dynamique

Le compilateur C++ met en place un **destructeur par défaut** qui ne gère pas la mémoire allouée dynamiquement => risque de fuite de mémoire (*memory leak*)

```
{ Class_avec_alloc_dyn x(50); // attribut pointeur vers bloc de 50 octets
```




```
} // fin de portée
```

Le destructeur par défaut est appelé mais il ne libère PAS le bloc pointé par p

Règle à mémoriser

Si on modifie l'une des trois méthodes ci-dessous,
il FAUT vérifier si les deux autres doivent être aussi adaptées

Constructeur de copie

Destructeur

Opérateur d'affectation (leçon ultérieure)