Programmation Orientée Objet

Constructeurs, destructeurs et gestion dynamique de la mémoire

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Plan du cours

Plan du cours

Qu'est ce qu'un constructeur ?

- Un constructeur $\rightsquigarrow$ méthode spéciale utilisée pour initialiser les instances d'une classe, affecter une valeur à chacun de leurs attributs
  - certaines actions d'initialisation : allocation mémoire, ouverture de fichier, ...
  - appelé immédiatement à chaque fois qu'un espace est alloué pour un objet
  - appelé automatiquement ou par l'utilisation de l'opérateur `new`
- Un constructeur a le même nom que la classe, et sans aucun type de retour
  - ne contient aucune instruction `return`
  - déclaré comme une méthode publique
- La surcharge de constructeurs permet d'initialiser des instances de classes de plusieurs manières
  - initialisation par défaut, par copie, ou spécialisée et définie par l'utilisateur

Plusieurs type de constructeurs

- Les constructeurs “classiques” $\rightsquigarrow$ définis par le développeur, ils peuvent prendre plusieurs paramètres
  - possibilité de définir plusieurs constructeurs “classiques”

Plusieurs type de constructeurs

- Les constructeurs “classiques” $\rightsquigarrow$ définis par le développeur, ils peuvent prendre plusieurs paramètres
  - possibilité de définir plusieurs constructeurs “classiques”
- Le constructeur par copie $\rightsquigarrow$ constructeur spécial qui prend en paramètre une référence vers une instance de la classe
  - si l’utilisateur n’en fournit pas, le compilateur en fournit un $\rightsquigarrow$ copie membre à membre
  - le constructeur par copie est appelé, lorsqu’une instance de classe :
    - est passée par valeur à une fonction
    - est retournée par une fonction
    - est initialisée avec une autre instance ($\rightsquigarrow$ initialisation : affectation à la déclaration)
    - est passée explicitement comme seul paramètre du constructeur

Plusieurs type de constructeurs

- Les constructeurs “classiques” $\rightsquigarrow$ définis par le développeur, ils peuvent prendre plusieurs paramètres
  - possibilité de définir plusieurs constructeurs “classiques”
- Le constructeur par copie $\rightsquigarrow$ constructeur spécial qui prend en paramètre une référence vers une instance de la classe
  - si l’utilisateur n’en fournit pas, le compilateur en fournit un $\rightsquigarrow$ copie membre à membre
  - le constructeur par copie est appelé, lorsqu’une instance de classe :
    - est passée par valeur à une fonction
    - est retournée par une fonction
    - est initialisée avec une autre instance ($\rightsquigarrow$ initialisation : affectation à la déclaration)
    - est passée explicitement comme seul paramètre du constructeur
- Le constructeur par défaut $\rightsquigarrow$ ne prend pas de paramètre, ou pour lequel chaque paramètre a une valeur par défaut
  - si l’utilisateur ne fournit aucun constructeur, le compilateur en fournit un sans argument

Exemples de constructeurs pour la classe Point2D

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;

public:

  // constructeur par defaut, sans parametre
  Point2D();

  // constructeur prenant deux parametres
  Point2D(float, float);

  // constructeur par copie
  Point2D(Point2D const&);

  // ...

};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

- Attention : un constructeur ne prend pas de type de retour ! !

Exemples de constructeurs pour la classe Point2D

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;

public:

  // constructeur par defaut, sans parametre
  Point2D();

  // constructeur prenant deux parametres
  Point2D(float, float);

  // constructeur par copie
  Point2D(Point2D const&);

  // ...

};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

- Attention : un constructeur ne prend pas de type de retour ! !

```
// Point2D.cpp

#include <iostream>
#include "Point2D.hpp"

Point2D::Point2D()
{
  x = 0;
  y = 0;
}

Point2D::Point2D(float _x, float _y)
{
  x = _x;
  y = _y;
}

Point2D::Point2D(Point2D const& pt)
{
  x = pt.x;
  y = pt.y;
} // ...
```

Appels des constructeurs

```
// c2-expl1.cpp
#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D a;                              // --> constructeur par defaut
  Point2D b = Point2D();                  // --> constructeur par defaut
  Point2D* pt1 = new Point2D;             // --> constructeur par defaut

  Point2D c(17.3,15.1);                   // --> constructeur specialise
  Point2D* pt2 = new Point2D(-1,-1);      // --> constructeur specialise

  Point2D d(c);                           // --> constructeur par copie
  Point2D e = *pt2;                       // --> constructeur par copie

  std::cout << "> a : "; a.print();   // > a :  --> Point2D (0,0)
  std::cout << "> b : "; b.print();   // > b :  --> Point2D (0,0)
  std::cout << "> c : "; c.print();   // > c :  --> Point2D (17.3,15.1)
  std::cout << "> d : "; d.print();   // > d :  --> Point2D (17.3,15.1)
  std::cout << "> e : "; e.print();   // > e :  --> Point2D (-1,-1)

  std::cout << "> pt1 : "; pt1->print(); // > pt1 :  --> Point2D (0,0)
  std::cout << "> pt2 : "; pt2->print(); // > pt2 :  --> Point2D (-1,-1)

  delete pt1; delete pt2;
  return 0;
}
```

Différence entre initialisation et affectation ?

```
// c2-expl1-bis.cpp
#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D tmp(17.3,15.1);    // initialisation

  Point2D tmp1(tmp);         // initialisation

  Point2D tmp2;
  tmp2 = tmp;                // affectation

  std::cout << "> tmp  : "; tmp.print();    // > tmp  :  --> Point2D (17.3,15.1)
  std::cout << "> tmp1 : "; tmp1.print();   // > tmp1 :  --> Point2D (17.3,15.1)
  std::cout << "> tmp2 : "; tmp2.print();   // > tmp2 :  --> Point2D (17.3,15.1)

  return 0;
}
```

Plan du cours

Qu'est ce qu'un destructeur ?

- Un destructeur $\rightsquigarrow$ méthode spéciale automatiquement appelée lors de la destruction d'une instance de classe
  - juste avant que la mémoire utilisée par l'objet ne soit récupérée par le système
  - certaines actions de nettoyages : libération mémoire, fermeture de fichier, ...
  - il y a au plus un destructeur par classe $\rightsquigarrow$ aucun argument

- Un destructeur a le même nom que la classe, précédé du caractère "~", et sans aucun type de retour
  - ne contient aucun paramètre et aucune instruction instruction `return`
  - déclaré comme une méthode publique

Définition d'un desctructeur pour la classe Point2D

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;

public:

  // destructeur
  ~Point2D();

  // ...

};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

- Attention : un destructeur ne prend pas de type de retour ! !

```
// Point2D.cpp

#include <iostream>
#include "Point2D.hpp"

Point2D::~Point2D()
{
  std::cout << "Destruction de : "
            << x << "," << y
            << std::endl;
}

// ...
```

Appels des destructeurs

```
// c2-expl2.cpp
#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D a;                              // --> constructeur par defaut
  Point2D b = Point2D();                  // --> constructeur par defaut
  Point2D* pt1 = new Point2D;             // --> constructeur par defaut

  Point2D c(17.3,15.1);                   // --> constructeur specialise
  Point2D* pt2 = new Point2D(-1,-1);      // --> constructeur specialise

  Point2D d(c);                           // --> constructeur par copie
  Point2D e = *pt2;                       // --> constructeur par copie

  std::cout << "> a : "; a.print();   // > a :  --> Point2D (0,0)
  std::cout << "> b : "; b.print();   // > b :  --> Point2D (0,0)
  std::cout << "> c : "; c.print();   // > c :  --> Point2D (17.3,15.1)
  std::cout << "> d : "; d.print();   // > d :  --> Point2D (17.3,15.1)
  std::cout << "> e : "; e.print();   // > e :  --> Point2D (-1,-1)

  std::cout << "> pt1 : "; pt1->print(); // > pt1 :  --> Point2D (0,0)
  std::cout << "> pt2 : "; pt2->print(); // > pt2 :  --> Point2D (-1,-1)

  delete pt1; delete pt2;                 // --> destructeur de pt1 et pt2
  return 0;
}                                         // --> destructeur de a, b, c, d, et e
```

Plan du cours

Le pointeur this

- Le pointeur this est un pointeur spécial qui référence l'instance de la classe elle-même
  - sa définition est transparente à l'utilisateur

- Remarque 1 : il peut être utilisé dans une classe lorsqu'un attribut a le même nom qu'un paramètre d'une méthode

```
// Point2D.cpp
void Point2D::setX(float x)
{
  this->x = x; // Remarque : la classe Point2D possede un attribut x (float x)
}
```

Le pointeur this

- Le pointeur this est un pointeur spécial qui référence l'instance de la classe elle-même
  - sa définition est transparente à l'utilisateur

- Remarque 2 : il peut également être utilisé dans une classe pour tester si le paramètre d'une méthode est l'instance de la classe elle-même

```
// Classe.cpp
void Classe::methode(const Classe& p)
{
  if(this != p){
    // le parametre p n'est pas l'instance de la classe elle-meme
  }
}
```

Attributs et méthodes statiques

- Habituellement, chaque instance d'une classe possède son propre exemplaire de chaque membre (attributs et méthodes)
  - attributs $\rightsquigarrow$ de la mémoire leurs est allouée à chaque instanciation de classe
  - méthodes $\rightsquigarrow$ elles ne peuvent être appelée que sur un objet particulier

Attributs et méthodes statiques

- Habituellement, chaque instance d'une classe possède son propre exemplaire de chaque membre (attributs et méthodes)
  - attributs $\rightsquigarrow$ de la mémoire leurs est allouée à chaque instanciation de classe
  - méthodes $\rightsquigarrow$ elles ne peuvent être appelée que sur un objet particulier

- Les membres statiques sont des membres partagés par toutes les instances d'une même classe
  - la définition d'un objet statique se fait en utilisant le mot clé `static`
  - attributs $\rightsquigarrow$ un seul espace mémoire alloué pour toutes les instances d'une classe
  - méthodes $\rightsquigarrow$ ne peuvent manipuler que des attributs et méthodes statiques, et n'ont pas accès au pointeur `this`

Exemple d'utilisation d'attribut statique

- L'utilisation classique d'une attribut statique permet de compter le nombre d'instances de classe créées

- Exemple : pour notre classe Point2D, on souhaite compter le nombre de points créés $\rightsquigarrow$ utilisation d'un attribut statique `nbPoints2D`
  - l'attribut statique est déclaré dans la définition de la structure de la classe
  - par contre, il est initialisé en dehors de la classe

Exemple d'utilisation d'attribut statique

```hpp
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;

public:

  static int nbPoints2D;

  // ...

};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

```cpp
// Point2D.cpp

#include <iostream>
#include "Point2D.hpp"

int Point2D::nbPoints2D = 0;

Point2D::Point2D()
{
  x = 0;
  y = 0;
  nbPoints2D ++;     // <-- incrementation
}

Point2D::~Point2D()
{
  std::cout << "Destruction de : "
            << x << "," << y
            << std::endl;
  nbPoints2D --;    // <-- decrementation
}
```

Exemple d'utilisation d'attribut statique

```cpp
// c2-expl3.cpp
#include <iostream>
#include "Point2D.hpp"

int
main( void )
{
  Point2D a;
  Point2D b = Point2D();

  // ...

  std::cout << "> Nombre de points : " << Point2D::nbPoints2D << std::endl;

  return 0;
}
```

```
$> ./c2-expl3
> Nombre de points : 2
Destruction de (0,0)
Destruction de (0,0)
```

L'opérateur de résolution de portée

- L'opérateur de résolution de portée "::" permet d'indiquer explicitement une portée ($\rightsquigarrow$ bloc d'instructions, classe, espace de noms, ...)

- Exemple 1 : utilisation pour spécifier l'espace de nom `std`

```
// Point2D.cpp

Point2D::~Point2D()
{
  std::cout << "Destruction de (" << x << "," << y << ")" << std::endl;
  nbPoints2D --;
}
```

L'opérateur de résolution de portée

- L'opérateur de résolution de portée "::" permet d'indiquer explicitement une portée ($\rightsquigarrow$ bloc d'instructions, classe, espace de noms, ...)

- Exemple 2 : accès à une méthode statique

```
// BoiteDialogue.hpp

class Classe
{
public:
  static int dialogue();
  // ...
};
```

```
int d = BoiteDialogue::dialogue();
```

L'opérateur de résolution de portée

- L'opérateur de résolution de portée "::" permet d'indiquer explicitement une portée ($\rightsquigarrow$ bloc d'instructions, classe, espace de noms, ...)

- Exemple 3 : résolution de cas de conflits de noms de variables

```
// Date.hpp

int mois;                       // variable globale
class Date
{
private:
  int mois;
  // ...
public:
  Date(int);
  // ...
};
```

```
// Date.cpp

Date::Date(int mois){
  Date::mois = mois;   // Date:mois --> attribut mois de l'instance courante
  ::mois = mois - 1;   // ::mois    --> variable globale mois
}
```

Attributs et méthodes constants

- Les attributs constants d'une classe ne peuvent être modifiés à aucun moment de leur existence
  - on les déclare en utilisant le mot clé `const`
  - ils doivent être initialisés par le constructeur de la classe ⇝ en fait, avant l'exécution du constructeur

```
// Point2D.hpp

#ifndef __POINT2D_HPP__
#define __POINT2D_HPP__

class Point2D
{

private:

  float x, y;
  const float origineX;
  const float origineY;

public:
  // ...
};      // <---- A NE PAS OUBLIER !!

#endif // __POINT2D_HPP__
```

```
// Point2D.cpp

#include <iostream>
#include "Point2D.hpp"

int Point2D::nbPoints2D = 0;

Point2D::Point2D():
  origineX(0),
  origineY(0)
{
  x = 0;
  y = 0;
  nbPoints2D ++;
}
```

Attributs et méthodes constants

- Les attributs constants d'une classe ne peuvent être modifiés à aucun moment de leur existence
  - on les déclare en utilisant le mot clé `const`
  - ils doivent être initialisés par le constructeur de la classe $\rightsquigarrow$ en fait, avant l'exécution du constructeur

- Les méthodes constantes sont les seules méthodes non-statiques qui soit possible d'appeler sur des objets constants
  - on les déclare également en utilisant le mot clé `const` (après leur signature)
  - elles ne peuvent pas modifier des attributs non statiques, sauf si ils sont déclarés avec le mot clé `mutable` ($\rightsquigarrow$ on verra ça plus tard)
  - elles ne peuvent pas appeler d'autres méthodes qui ne sont pas constantes

Plan du cours

Données dynamiques

- Les données dynamiques sont créées pendant l'exécution, et manipulées à travers des pointeurs
  - en opposition aux données statiques (déterminées à la compilation)

- En C++, on utilise deux opérateurs :
  - `new` $\rightsquigarrow$ pour allouer un espace mémoire (assimilable à la fonction `malloc` en C)
  - `delete` $\rightsquigarrow$ pour libérer un espace mémoire (assimilable à la fonction `free` en C)

Les tableaux en C++

```
// c2-expl4.cpp
#include "Point2D.hpp"

int
main( void )
{
  const int nbElts = 10;

  Point2D tabPoints[nbElts];    // = tableau de 10 objets Point2D
  //   --> 10 appels au constructeur par defaut

  Point2D* tab2Points = new Point2D[nbElts];
  // = tableau dynamique de 10 objets Point2D
  //   --> 10 appels au constructeur par defaut

  Point2D tab3Points[2] = { Point2D(-1,-1) , Point2D(tab2Points[0]) };
  // = tableau de 2 objets Point2D
  //   --> 1 appel au constructeur classique
  //   --> 1 appel au constructeur par copie

  Point2D tab4Points[2][2];    // = tableau de 2 x 2 objets Point2D
  //   --> 4 appels au constructeur par defaut

  delete[] tab2Points;  //   --> 10 appels au destructeur de Point2D (tab2Points)

  return 0;             //   --> 16 appels au destructeur de Point2D
}
```

Questions ?