Programmation Orientée Objet en C++

Emanuel Aldea <emanuel.aldea@u-psud.fr>
http://hebergement.u-psud.fr/emi

M1-E3A

Deuxième chapitre

Les objets

Objets complexes

La construction

- **objet complexe** : un objet qui a au moins un sous-objet obtenu par agrégation hybride (déclaré avec `class` ou `struct`)

Objets complexes

La construction

- **objet complexe** : un objet qui a au moins un sous-objet obtenu par agrégation hybride (déclaré avec `class` ou `struct`)
- **sous-objet** : un objet déclaré comme attribut ou obtenu par héritage

Objets complexes

La construction

- **objet complexe** : un objet qui a au moins un sous-objet obtenu par agrégation hybride (déclaré avec `class` ou `struct`)
- **sous-objet** : un objet déclaré comme attribut ou obtenu par héritage
- avant d'exécuter le constructeur de l'objet, on construit tout sous-objet qui le compose :
  - les objets attributs (déclarés dans la liste)
  - les objets héritées (éventuellement)

Objets complexes

La construction

- **objet complexe** : un objet qui a au moins un sous-objet obtenu par agrégation hybride (déclaré avec `class` ou `struct`)
- **sous-objet** : un objet déclaré comme attribut ou obtenu par héritage
- avant d'exécuter le constructeur de l'objet, on construit tout sous-objet qui le compose :
  - les objets attributs (déclarés dans la liste)
  - les objets héritées (éventuellement)
- à son tour, chaque sous-objet est construit de la même manière

Objets complexes

La construction - les étapes

- allocation de mémoire pour l'objet complexe

Objets complexes

La construction - les étapes

- allocation de mémoire pour l'objet complexe
- appel des constructeurs des objets les plus petites (de type natif)

Objets complexes

La construction - les étapes

- allocation de mémoire pour l'objet complexe
- appel des constructeurs des objets les plus petites (de type natif)
- appel des constructeurs des objets intermédiaires

Objets complexes

La construction - les étapes

- allocation de mémoire pour l'objet complexe
- appel des constructeurs des objets les plus petites (de type natif)
- appel des constructeurs des objets intermédiaires
- dernière étape : exécution du constructeur de l'objet complexe

Objets complexes

La construction - les étapes

- allocation de mémoire pour l'objet complexe
- appel des constructeurs des objets les plus petites (de type natif)
- appel des constructeurs des objets intermédiaires
- dernière étape : exécution du constructeur de l'objet complexe
- **important :** avant l'exécution du constructeur courant, tous les sous-objets ont été déjà instanciés entièrement !

Objets complexes

Constructeurs multiples : comment le compilateur sait lequel appeler ?

- soit l'appel au constructeur du sous-objet figure entre le prototype et le corps du constructeur

Objets complexes

Constructeurs multiples : comment le compilateur sait lequel appeler ?

- soit l'appel au constructeur du sous-objet figure entre le prototype et le corps du constructeur
- soit il n'y a pas d'appel explicite et alors le compilateur appelle pour le sous - objet en question le constructeur sans paramètres. S'il ne trouve pas de constructeur sans paramètres il affiche une erreur de compilation.

Objets complexes

Constructeurs multiples : comment le compilateur sait lequel appeler ?

- soit l'appel au constructeur du sous-objet figure entre le prototype et le corps du constructeur
- soit il n'y a pas d'appel explicite et alors le compilateur appelle pour le sous - objet en question le constructeur sans paramètres. S'il ne trouve pas de constructeur sans paramètres il affiche une erreur de compilation.
- **important :** ne jamais préjuger sur l'ordre d'appel des constructeurs des sous-objets !

Objets complexes

```
class ClasseA
{
  ClasseB b;
  ClasseC c; //appel implicite du constructeur
  ClasseD d;

public:
  ClasseA (params): b(params1), d(params2) //liste d'appels explicites
  {
    // corps du constructeur
  }
};
```

- **Appel explicite obligatoire** : si le sous-objet n'a pas de constructeur sans paramètres

Objets complexes

```
class ClasseA
{
  ClasseB b;
  ClasseC c; //appel implicite du constructeur
  ClasseD d;

public:
  ClasseA (params): b(params1), d(params2) //liste d'appels explicites
  {
    // corps du constructeur
  }
};
```

- **Appel explicite obligatoire** : si le sous-objet n'a pas de constructeur sans paramètres
- **Appel explicite optionnel** : si le sous-objet a un constructeur sans paramètres

Objets complexes

Constructeur de copie et opérateur d'attribution

- le même principe de l'appel itératif des constructeurs (de l'objet le plus simple jusqu' à l'objet le plus complexe) s'applique

Objets complexes

Constructeur de copie et opérateur d'attribution

- le même principe de l'appel itératif des constructeurs (de l'objet le plus simple jusqu' à l'objet le plus complexe) s'applique
- Mais il n'y a plus d'ambiguï té possible pour les appels donc le compilateur le fait de manière implicite

Objets complexes

Constructeur de copie et opérateur d'attribution

- le même principe de l'appel itératif des constructeurs (de l'objet le plus simple jusqu' à l'objet le plus complexe) s'applique
- Mais il n'y a plus d'ambiguï té possible pour les appels donc le compilateur le fait de manière implicite
- **important :** un seul constructeur ou opérateur de copie inaccessible (privé ou autre) peut empêcher la copie du "grand" objet !

Objets complexes

Destructeurs

- à la destruction l'ordre est inversé

Objets complexes

Destructeurs

- à la destruction l'ordre est inversé
- après l'exécution du destructeur de l'objet, le compilateur appelle les destructeurs de tous les sous-objets qui le compose :
  - les objets attributs (déclarés dans la liste)
  - les objets hérités (éventuellement)

Objets complexes

Destructeurs

- à la destruction l'ordre est inversé
- après l'exécution du destructeur de l'objet, le compilateur appelle les destructeurs de tous les sous-objets qui le compose :
  - les objets attributs (déclarés dans la liste)
  - les objets hérités (éventuellement)
- il n'y a pas d'ambiguïté car les destructeurs sont uniques

Conversions implicites

Qui les provoque ? Vers quel type/classe ?

- une attribution ou une opération suivie d'attribution : vers le type de l'opérande gauche (conteneur)

Conversions implicites

Qui les provoque ? Vers quel type/classe ?

- une attribution ou une opération suivie d'attribution : vers le type de l'opérande gauche (conteneur)
- un champ d'instruction (for, while, if etc.) : vers le type `bool`

Conversions implicites

Qui les provoque ? Vers quel type/classe ?

- une attribution ou une opération suivie d'attribution : vers le type de l'opérande gauche (conteneur)
- un champ d'instruction (for, while, if etc.) : vers le type `bool`
- un appel de fonction / méthode / constructeur : vers le type du paramètre déclaré dans le prototype

Conversions implicites

Qui les provoque ? Vers quel type/classe ?

- une attribution ou une opération suivie d'attribution : vers le type de l'opérande gauche (conteneur)
- un champ d'instruction (for, while, if etc.) : vers le type `bool`
- un appel de fonction / méthode / constructeur : vers le type du paramètre déclaré dans le prototype
- une instruction de retour : vers le type de retour de la fonction / méthode

Conversions implicites

Qui les provoque ? Vers quel type/classe ?

- une attribution ou une opération suivie d'attribution : vers le type de l'opérande gauche (conteneur)
- un champ d'instruction (for, while, if etc.) : vers le type `bool`
- un appel de fonction / méthode / constructeur : vers le type du paramètre déclaré dans le prototype
- une instruction de retour : vers le type de retour de la fonction / méthode
- un opérateur non défini pour le type en question : vers le type pour lequel l'opérateur est défini

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`
- opérateurs surchargés de typecast de `class1` vers `class2`

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`
- opérateurs surchargés de typecast de `class1` vers `class2`
- extraction de l'ancêtre si `class2` est un ancêtre de `class1`

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`
- opérateurs surchargés de typecast de `class1` vers `class2`
- extraction de l'ancêtre si `class2` est un ancêtre de `class1`
- s'il n'y a pas de règle : erreur de conversion

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`
- opérateurs surchargés de typecast de `class1` vers `class2`
- extraction de l'ancêtre si `class2` est un ancêtre de `class1`
- s'il n'y a pas de règle : erreur de conversion
- s'il y a une seule règle : conversion automatique

Conversions implicites

Quelles règles applique le compilateur pour convertir le type `class1` vers le type `class2` ?

- règles de conversion implicite des types natifs
- constructeur de `class2` avec un paramètre : `class1`
- opérateurs surchargés de typecast de `class1` vers `class2`
- extraction de l'ancêtre si `class2` est un ancêtre de `class1`
- s'il n'y a pas de règle : erreur de conversion
- s'il y a une seule règle : conversion automatique
- s'il y a plusieurs règles : erreur d'ambiguïté

Conversions explicites

Spécifiques au C++

- `const_cast<type>` : peut enlever la propriété `const` d'un objet qui n'est pas déclaré initialement `const` (sinon sa modification engendra un résultat indéfini)

Conversions explicites

Spécifiques au C++

- `const_cast<type>` : peut enlever la propriété `const` d'un objet qui n'est pas déclaré initialement `const` (sinon sa modification engendra un résultat indéfini)
- `static_cast<type>` : demande explicitement la conversion de l'objet reçu en paramètre vers le type spécifiée ; si le type est un pointeur, pas de vérification de consistance effectuée

Conversions explicites

Spécifiques au C++

- `const_cast<type>` : peut enlever la propriété `const` d'un objet qui n'est pas déclaré initialement `const` (sinon sa modification engendra un résultat indéfini)
- `static_cast<type>` : demande explicitement la conversion de l'objet reçu en paramètre vers le type spécifiée ; si le type est un pointeur, pas de vérification de consistance effectuée
- `dynamic_cast<type>` : similaire au précédent mais si le type est un pointeur, vérification de consistance effectuée ; coûteux

Conversions explicites

Spécifiques au C++

- `const_cast<type>` : peut enlever la propriété `const` d'un objet qui n'est pas déclaré initialement `const` (sinon sa modification engendra un résultat indéfini)
- `static_cast<type>` : demande explicitement la conversion de l'objet reçu en paramètre vers le type spécifiée ; si le type est un pointeur, pas de vérification de consistance effectuée
- `dynamic_cast<type>` : similaire au précédent mais si le type est un pointeur, vérification de consistance effectuée ; coûteux
- `reinterpret_cast<type>` : cast de bas-niveau, aux niveau des bits

Surcharge de conversion

```
class A
{
private:
    int memberInt;
public:
    ...
    // Overloaded int cast
    operator int() { return memberInt; }
    ...
};
...
A myobj;
int val = static_cast<int>(myobj);
```

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`
- les fichiers, déclarés dans `<fstream.h>`

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`
- les fichiers, déclarés dans `<fstream.h>`
- la mémoire, déclarés dans `<strstream.h>`

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`
- les fichiers, déclarés dans `<fstream.h>`
- la mémoire, déclarés dans `<strstream.h>`
- Pour toute classe type flot de données, l'entrée est faite par l'opérateur polymorphe `> >`

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`
- les fichiers, déclarés dans `<fstream.h>`
- la mémoire, déclarés dans `<strstream.h>`
- Pour toute classe type flot de données, l'entrée est faite par l'opérateur polymorphe `> >`
- Pour toute classe type flot de données, la sortie est faite par l'opérateur polymorphe `< <`

Les flots d'entrée/sortie

Spécification

- C++ prévoit des entrées/sorties à l'aide des objets, appelés flots, vers trois médias différents
- la console/le clavier, déclarés dans `<iostream.h>`
- les fichiers, déclarés dans `<fstream.h>`
- la mémoire, déclarés dans `<strstream.h>`
- Pour toute classe type flot de données, l'entrée est faite par l'opérateur polymorphe `> >`
- Pour toute classe type flot de données, la sortie est faite par l'opérateur polymorphe `< <`
- Les manipulateurs s'insèrent dans les flots et modifient leur comportement (formatage etc.) ; déclarés en `<iomanip>`

Les flots d'entrée/sortie

Extension aux nouvelles classes

Extension à n'importe quel objet (surcharge par fonction friend car opérande droit) !

```
class CCercle
{
  float x, y, r;
  public:
  ...
  friend ostream & operator <<( ostream & os, const CCercle & c);
};
...
ostream & operator <<( ostream & os, const CCercle & c)
{
  return os<<"Cercle de rayon "<<c.r << " et centre (" <<
      c.x <<","<< c.y <<") \ n";
}
...
CCercle monobj;
cout << monobj; //affichage par surcharge de <<
```

Les flots d'entrée/sortie

Les fichiers

Il n'y a pas de différence à l'utilisation entre cout et un fichier stream ouvert en mode sortie

```
#include <iostream>
#include <fstream>
using namespace std;
...
CCercle c(1.0f, 3.0f, 4.0f);
cout << c;
fstream fs ("cercle.txt", ios ::out);
fs << c;
```