

GL / C++

Chapitre 5

Destructeur d'objet

Destructeur d'objet

- Méthode particulière ayant pour nom le nom de la classe précédé du caractère ~
- Lorsque la classe d'un objet X possède un destructeur, ce destructeur est automatiquement appelé lorsque l'objet X est enlevé de la mémoire qu'il occupe
- Un destructeur d'objet sert à programmer les opérations qu'il est nécessaire d'effectuer lorsque l'objet est enlevé de la mémoire qu'il occupe.
- Pas besoin pour les classes des exemples précédents.

Destructeur d'objet

- Dans l'exemple suivant, on utilise une classe d'objets ayant un destructeur qui n'est pas nécessaire mais qui sert :
 - ❑ à afficher une trace chaque fois qu'il est appelé
 - ❑ et ainsi à montrer les instants où les objets sont détruits.
- **Seules classes qui ont besoin d'un destructeur :**
Les classes qui dans leurs attributs ont des pointeurs sur d'autres objets situés en mémoire dynamique.
- C'est pour ces classes que les destructeurs existent (nous verrons des exemples plus tard).

Classe avec destructeur : spécification

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

class Objet {
    private:
        string nom; // nom de l'objet

    public:
        Objet(string nom = "sans nom");
        virtual ~Objet(); // destructeur
};
```

Classe avec destructeur : implémentation

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

Objet::Objet(string nom) {
    this->nom = nom;
    cout << "construction de " << nom << endl ;
}

//-----
Objet::~Objet() { // implémentation du destructeur

    cout << "destruction de " << nom << endl ;
}
```

Programme exemple

```
Objet global("global");  
//-----
```

Que donnera la trace ?

```
void p1() {  
    Objet o("local p1");  
}
```

```
//-----
```

```
void p2() {  
    Objet o("local p2");  
    p1();  
}
```

```
//-----
```

```
int main() {  
    Objet o("local main");  
    p1(); p2();  
    return 0;  
}
```

Programme exemple

```
Objet global("global");  
//-----  
void p1() {  
    Objet o("local p1");  
}  
  
//-----  
void p2() {  
    Objet o("local p2");  
    p1();  
}  
  
//-----  
int main() {  
    Objet o("local main");  
    p1(); p2();  
    return 0;  
}
```

Trace

construction de global	<i>début d'exécution</i>
construction de local main	<i>début de main</i>
construction de local p1	<i>début de p1</i>
destruction de local p1	<i>fin de p1</i>
construction de local p2	<i>début de p2</i>
construction de local p1	<i>début de p1</i>
destruction de local p1	<i>fin de p1</i>
destruction de local p2	<i>fin de p2</i>
destruction de local main	<i>fin de main</i>
destruction de global	<i>fin du programme</i>

GL / C++

Chapitre 6

Classes dérivées, Héritage

Illustration par l'exemple

- Objets représentant des étudiants inscrits à un examen.
- Chaque objet est composé :
 - ❑ du nom de l'étudiant
 - ❑ de son adresse
 - ❑ de la note qu'il a obtenu à l'examen
- Un objet étudiant possède les méthodes suivantes :
 - ❑ un constructeur fournissant le nom et l'adresse et initialisant la note à zéro
 - ❑ une procédure qui affiche les valeurs d'attributs de l'étudiant sur l'écran
 - ❑ une procédure qui change l'adresse de l'étudiant
 - ❑ l'opérateur == qui compare le nom de l'étudiant à un nom donné en paramètre
 - ❑ une procédure qui fixe la note de l'étudiant.

1. Première définition d'une classe etudiant

spécification

```
class Etudiant {
```

```
    private:
```

```
        string    nom;  
        string    adresse;  
        float     note;
```

```
    public:
```

```
        Etudiant(string nom="",  
                  string adresse="" );  
        void afficher();  
        void setAdresse(string adresse);  
        bool egal(string nom);  
        void setNote(float note);
```

```
};
```

Première définition d'une classe **etudiant**

implémentation

- Très semblable à celle de la classe **personne**
- Différences :
 - ❑ le **constructeur**
 - ❑ la méthode **afficher**
 - ❑ la procédure **setNote**
- Pour **écrire rapidement** la classe **etudiant** il est possible :
 - ❑ de faire une copie de la classe **personne** (.h et .cc)
 - ❑ et dans cette copie d'effectuer les modifications nécessaires
- C'est peut-être ainsi que l'on procéderait avec un langage de programmation classique (sans héritage)

Rappel : la classe personne

```
class Personne {  
private:  
    string nom;  
    string adresse;  
  
public:  
    Personne(string nom="",  
              string adresse="");  
    void afficher();  
    void setAdresse(string adresse);  
    int  egal(string nom);  
};
```

```
class Etudiant {  
private:  
    string nom;  
    string adresse;  
    float note;  
  
public:  
    Etudiant(string nom="",  
              string adresse="");  
    void afficher();  
    void setAdresse(string adresse);  
    int  egal(string nom);  
    void setNote(float note);  
};
```

Intérêt de l'héritage

- **L'héritage** permet d'écrire la classe **etudiant** plus rapidement :
 - ❑ en écrivant seulement les différences par rapport à la classe **personne**
 - ❑ mais **sans faire de copie** de la classe **personne**
- La spécification de la classe **etudiant** dit **explicitement** qu'elle est **dérivée** de la classe **personne**, c'est-à-dire :
 - ❑ qu'elle prend tout ce qui existe dans la classe **personne**
 - ❑ qu'elle effectue des modifications
- La classe **etudiant** ainsi écrite **hérite** de la classe **personne**

2. Classe etudiant dérivée de la classe personne - Spécification

```
#include "Personne.h"

class Etudiant : public Personne {
// la classe Etudiant dérive de Personne
private:
    float note;

public:
    Etudiant (string nom="",
              string adresse="");
    void afficher();
    void setNote(float note);
};
```

Remarques

■ `class Etudiant : public Personne`

Sert à indiquer au compilateur C++ :

- ❑ que la classe `Etudiant` est dérivée de la classe `Personne`
- ❑ que l'on veut que les membres publics hérités de la classe `Personne` soient aussi publics dans la classe `Etudiant`.

■ Il existe d'autres modes d'héritage que le mode d'héritage `public` : `protected` et `private`

Classe Etudiant dérivée de la classe Personne - Implémentation

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

```
Etudiant::Etudiant(string nom, string adresse)  
    : Personne(nom, adresse) // appel constructeur classe mère  
{  
    note = 0; // construction supplémentaire  
}  
//-----  
void Etudiant::afficher()  
{  
    Personne::afficher(); // appel d'une méthode héritée  
    cout << "note : " << note << endl ;  
}  
//-----  
void Etudiant::setNote(float note)  
{  
    this->note = note;  
}
```

Remarques

- Le constructeur d'une classe dérivée doit :
 - appeler le constructeur de sa classe mère
 - lui donner les valeurs de paramètres nécessaires
- Dans la méthode `Etudiant::afficher()`, on se sert de la méthode `afficher` de la classe mère de `Etudiant` (`Personne::afficher`)
- On réutilise les parties de programmes déjà écrites... c'est un des objectifs de la programmation par objets.

3. Définition d'une classe salarié

- Objets représentant des **salariés** d'une entreprise.
- Un salarié est composé :
 - de son nom et de son adresse
 - du nom de l'entreprise qui l'emploie
 - du montant de son salaire
- Le constructeur donne une valeur pour chaque attribut
- Outre les méthodes de toute personne, un salarié possède une méthode pour changer son salaire.

classe Salarié : spécification

```
#include "Personne.h"
```

```
class Salarie : public Personne {
```

```
private:
```

```
    string employeur;
```

```
    float    salaire;
```

```
public:
```

```
    Salarie(string nom="", string adresse="",  
            string employeur="", float salaire=0.0);
```

```
    void afficher();
```

```
    void setSalaire(float salaire);
```

```
};
```

classe salarié : implémentation

```
#include "Salarie.h"

...
Salarie::Salarie(string nom,          string adresse,
                string employeur, float salaire)
    : Personne(nom, adresse) // appel constructeur classe mère
{
    this->employeur = employeur;
    this->salaire = salaire;
}
//-----
void Salarie::afficher()
{
    Personne::afficher(); // methode afficher de la classe mère
    cout << employeur << salaire << endl ;
}
//-----
void Salarie::setSalaire(float salaire)
{
    this->salaire = salaire;
}
```

4. Membres protégés d'une classe

- La méthode **Salarie::afficher** affiche :
 - le nom et l'adresse sur une ligne
 - l'employeur et le salaire sur une autre ligne
- Il serait plus agréable d'afficher les 4 valeurs sur une même ligne. Il faudrait pouvoir écrire la procédure afficher suivante :

```
void Salarie::afficher() {  
    cout << nom << adresse << employeur << salaire;  
}
```

- Mais cette procédure est incorrecte, car les attributs **nom** et **adresse** sont privés dans la classe **personne**.
- Il suffirait dans **personne**, de déclarer ces attributs publics... mais cette solution n'est pas bonne !

Notion de Membres Protégés

- C++ permet de déclarer des membres de classes :
 - Qui sont accessibles aux classes dérivées
 - Qui ne sont **pas** accessibles aux autres parties du programme.
- Ce sont les membres dits **protégés**

Classe Personne avec membres protégés

```
class Personne {
```

```
protected:    // membres protégés
```

```
    string nom;
```

```
    string adresse;
```

```
public:        // membres publics
```

```
    ...        // méthodes identiques
```

```
};
```

- L'implémentation est évidemment inchangée.

5. Représentation en mémoire - Arbre d'héritage

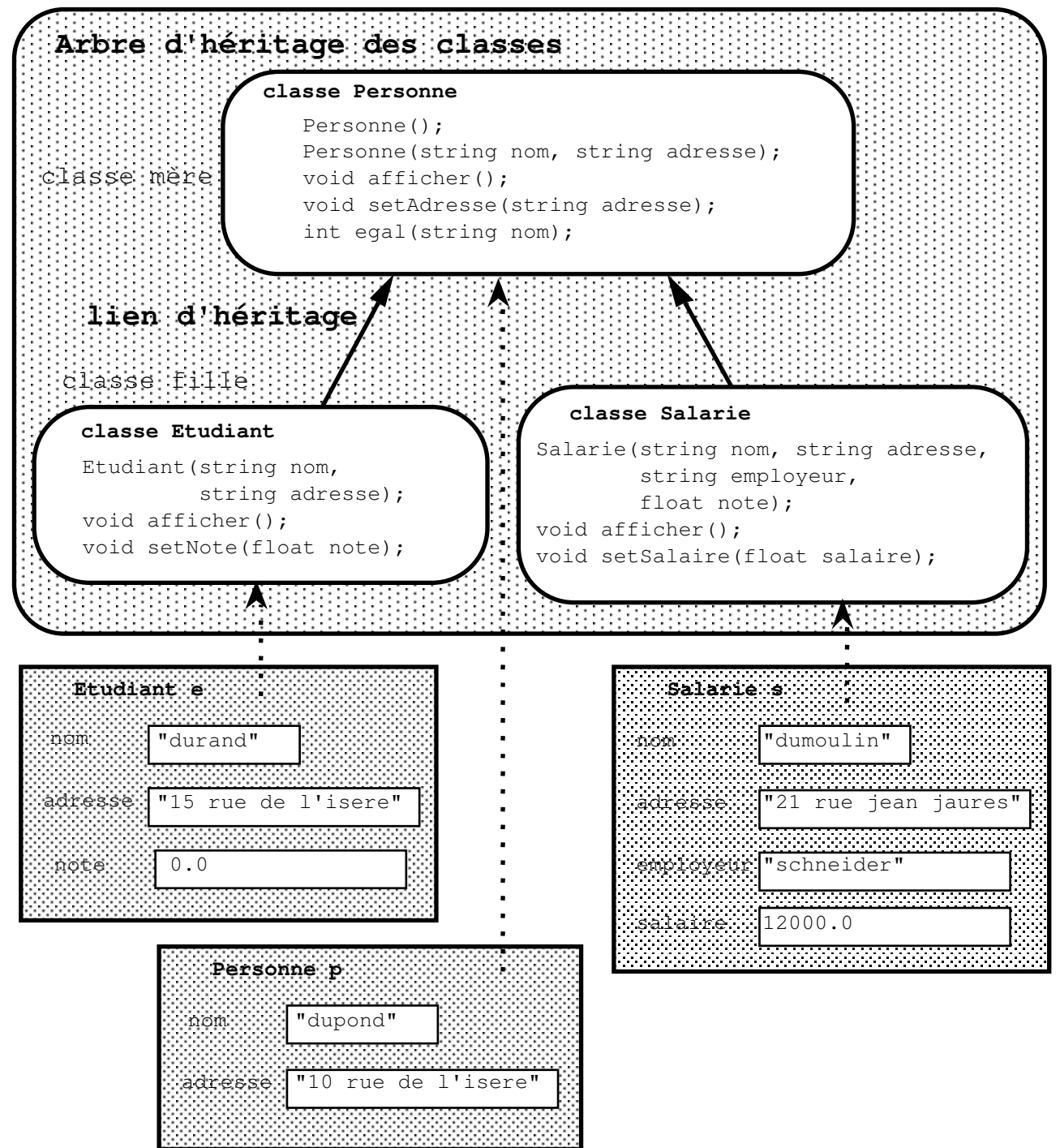
- Considérons les 3 objets suivants :

```
Personne p("dupond", "10 rue de l'isere");  
Etudiant e("durand", "15 rue de l'isere");  
Salarie s("dumoulin", "21 rue jean jaures",  
          "schneider", 12000.00);
```

- **Arbre d'héritage** des classes **Personne**, **Etudiant** et **Salarie**.

- **Personne** est la **classe mère** de **Etudiant** et **Salarie**.

- **Etudiant** et **Salarie** sont des **classes filles** de **Personne**



Remarques

- Un Etudiant ou un Salarie possède :
 - les membres de sa classe
 - et indirectement par le **lien d'héritage**, les membres de la classe **Personne**.
- Un objet de type Etudiant a donc 2 méthodes afficher :
 - celle de la classe **Etudiant** que l'on nomme :
 - **afficher**
 - **OU Etudiant::afficher** *//nom complet de la méthode*
 - celle de la classe **Personne** que l'on nomme
 - **Personne::afficher** *// nom complet de la méthode*

6. Appel d'une méthode héritée

- Etant donné la classe Etudiant, il est possible d'écrire la procédure suivante :

```
#include "Etudiant.h"
int main() {

    Etudiant e("durand", "15 rue de l'isere");
    e.afficher();
    // appelle la méthode afficher de etudiant
    e.Personne::afficher();
    // appel méthode afficher de personne
    // qui se nomme Personne::afficher
    return 0;
}
```

- Pour un objet de type Etudiant, l'appel de `Personne::afficher` peut être utile si on désire afficher seulement le nom et l'adresse de l'étudiant.

7. Modes d'héritage d'une classe dérivée

- Soit D une classe dérivée d'une classe C.
- Le mode d'héritage de D définit le statut dans D des membres hérités de C, c'est-à-dire si les membres hérités de C sont **privés**, **protégés** ou **publics** dans D
- Dans le mode d'héritage **public** :
 - les membres public de C restent publics dans D
 - les membres protégés de C restent protégés dans D
 - les membres privés de C restent privés à C
- Dans le mode d'héritage **protected** :
 - les membres publics et protégés de C sont protégés dans D
 - les membres privés de C restent privés à C
- Dans le mode d'héritage **private** :
 - Les membres publics et protégés de C sont privés dans D
 - les membres privés de C restent privés à C

exemple illustrant les modes d'héritage : classe mère C

```
class C {  
    private:  
        short i;    // i sera toujours privé à C  
    protected:  
        short j;  
    public:  
        short k;  
};
```

Exemples de modes d'héritage

```
class D1 : private C {  
  
    //      j et k privés à D1  
};  
//-----  
class D2 : protected C {  
  
    //      j et k protégés dans D2  
    //      (accessibles seulement aux classes dérivées)  
};  
//-----  
class D3 : public C {  
  
    //      j protégé dans D3 et k public dans D3  
};
```

8. Ordre d'Appel des constructeurs et destructeurs

- Dans le cas de **classes dérivées** :
 - ❑ Les Objets sont détruits dans **l'ordre inverse** de leur Construction
 - ❑ Les **Constructeurs** des classes mères sont appelés en **premier**
 - ❑ Les **Destructeurs** des classes mères sont appelés en **dernier**

Spécification des classes de l'exemple

```
class C1 {
```

```
protected:
```

```
    string nom;
```

```
public:
```

```
    C1 (string nom) ;
```

```
    ~C1 () ;
```

```
};
```

```
//-----
```

```
class C2 : public C1 {
```

```
public:
```

```
    C2 (string nom) ;
```

```
    ~C2 () ;
```

```
};
```

Implémentation des classes de l'exemple

```
C1::C1(string nom) {  
    this->nom=nom;  
    cout << "construction de C1(" << nom << ") " << endl ;  
}
```

```
C1::~~C1() {  
    cout << "destruction de C1(" << nom << ") " << endl ;  
}
```

```
//-----
```

```
C2::C2(string nom) : C1(nom) {  
    cout << "construction de C2(" << nom << ") " << endl ;  
}
```

```
C2::~~C2() {  
    cout << "destruction de C2(" << nom << ") " << endl ;  
}
```

Programme d'utilisation

```
int main()  
{  
    C1 o1("aaa");  
    C2 o2("bbb");  
  
    return 0;  
}
```

Que donnera la trace ?

Programme d'utilisation

```
int main()
```

```
{
```

```
    c1 o1( "aaa" );
```

```
    c2 o2( "bbb" );
```

```
    return 0;
```

```
}
```

Trace

construction de C1(aaa)

construction de C1(bbb) // le constructeur de C2

// appelle celui de C1

// avant de s'exécuter

construction de C2(bbb)

destruction de C2(bbb) // destructeur de C2 exécuté

// avant celui de sa classe

// mère C1

destruction de C1(bbb)

destruction de C1(aaa)

- Objets détruits dans l'ordre inverse de leur construction
- Constructeurs des classes mères appelés en premier
- Destructeurs des classes mères appelés en dernier

Exercice

- On considère la classe suivante :

```
class Point {  
    private :  
        float x, y ; // x et y sont privés  
    public :  
        void initialise (float abs=0.0, float ord=0.0) {  
            x = abs ; y = ord ;  
        }  
        void affiche () {  
            cout << "Point de coordonnées : " << x << " " << y << "\n" ;  
        }  
        float abs () { return x ; }  
        float ord () { return y ; }  
};
```

1. Créer une classe Pointb, dérivée de Point, comportant une nouvelle fonction distanceAuCarré qui calcule la distance au carré entre le point et l'origine
2. Même question en supposant que les membres x et y ont été déclarés protégés dans Point
3. Introduire un constructeur dans la classe Pointb
4. Quelles sont les fonctions membres utilisables pour un objet de type Pointb ?