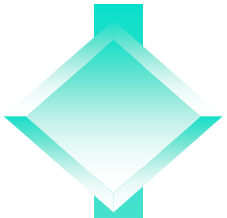
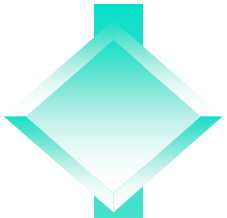


UNIVERSITÉ PARIS SORBONNE PARIS IV

Cours n° 3
Polymorphisme I



Sommaire

Résolution de problèmes

Méthodologies de résolution de problèmes

Problème spécifique (définition d'une unité logicielle spécifique)
multiplication des unités logicielles spécifiques
augmentation des erreurs et du temps de développement

Cas particulier d'un problème plus général (spécialisation d'une unité logicielle générique ou réutilisation)

Unité logicielle générique

Indépendance de la définition de l'unité logicielle (e.g., fonction, classe, …) au contexte d'exécution (e.g., type des paramètres ou des fonctions)

Instanciation de l'unité générique avec des types et/ou des fonctions

Polymorphisme

Possibilité pour un opérateur ou une fonction d'être utilisable dans des contextes différents et d'avoir un comportement adapté à ces paramètres.

Polymorphisme ad-hoc = surcharge de méthodes

Polymorphisme d'inclusion = héritage ou redéfinition

Polymorphisme paramétrique = types génériques ou généricité

Programmation modulaire

Définition de deux fonctions de même nom avec des paramètres différents (type ou nombre).

Par exemple, définition d'une fonction Maximum pour deux entiers, pour deux réels

```
int Maximum(int i, int j){return i > j ? i : j;}
float Maximum(float x, float y){return x > y ? x : y;}

int main (int argc, char *argv[]) {
float x, y; cout << "entrez deux réels : "; cin >> x >> y;
cout << Maximum(x, y) << endl;

int i, j; cout << "entrez deux entiers : "; cin >> i >> j;
cout << Maximum(i, j) << endl;
}
```

Définition de deux méthodes de même nom dans deux classes indépendantes.

Par exemple, définition d'une méthode afficher pour les deux classes Image et Texte

```
class Image {
public :
Image ();
void afficher() const;
};

class Texte {
public :
Texte ();
void afficher() const;
};
```

Principe

Extension de la notion de la surcharge de fonction

• Interprétation d'un opérateur comme d'un appel de fonction

Ex: Equivalence pour le compilateur entre l'expression a+b et l'appel de add(a, b)

Opérateurs surchargables

• Binaire	()	[]	->					
• Unaire	+	-	++	--	!	~	*	&
• Unaire	new	new[]	delete					
• Binaire	*	/	%	+	-			
• Binaire	<<	>>						
• Binaire	<	<=	>	>=	=	!=		
• Binaire	&	^	||	&&	!			
• Binaire	=	+=	-=	*=	/=	%=		
• Binaire	&=	^=	|=	<<=	>>=			
• Binaire	,							

Syntaxes

Surcharge externe d'opérateur

- Redéfinition par une fonction externe à la classe
- Accès aux attributs de classe
  Accesseurs (fonctions membres),
  Fonctions amies de la classe (qualificatif friend)

Syntaxe type operator Op (type 1 var1 [,type2 var2])

- type1 ou type2 doivent des classes
- « var1 Op var2 » interprétée comme « operator op (var1, var2)

Surcharge interne d'opérateur

- Redéfinition par une fonction membre

- **Syntaxe** type operator Op([type 2 var2])
- « var1 Op var2 » interprétée comme « var1.operator op (var2)

Classe Identite

Deux attributs privés de type String

nom et prénom

Addition de deux Identite

Surcharge interne de l'opérateur +

Test d'égalité de deux Identite

Surcharge interne de l'opérateur ==

Envoi sur un flot de sortie d'une Identite

Surcharge externe de l'opérateur <<

méthode d'une autre classe (iostream)

Lecture sur un flot d'entrée d'une Identite

Surcharge externe de l'opérateur >>

méthode d'une autre classe (iostream)

Identite.h

Définition de la classe Identite

```
class Identite {
private :
string nom_, prenom_;

public :
Identite (string n, string p); // constructeurs
Complexe ();

string nom() const; // accesseurs
string prenom() const;

Identite operator + (const Identite & y); // redéfinition de +
bool operator == (const Identite & y); // redéfinition de ==

// redéfinition par une fonction externe amie de la réception sur le
flot d'entrée
friend istream & operator >> (istream & S, Identite & y);
// redéfinition de l'envoi sur le flot de sortie
friend ostream & operator << (ostream & S, const Identite & y);
} ;
```

Identite.cpp

Constructeurs, accesseurs et surcharge interne

```
#include « Identite.h"
// constructeurs
Identite::Identite (string n, string p) {nom_ = n; prenom = p;}
Identite::Identite () {nom_ = ""; prenom_ = "";}

// accesseurs
string Identite::nom () const {return nom_;}
string Identite::prenom () const {return prenom_;}

// surcharge interne de l'opérateur +
Identite Identite::operator + (const Identite & y) {
      Identite c(nom() + "-" + y.nom(), prenom());
return c; }

// surcharge interne de l'opérateur ==
bool Identite::operator == (const Identite & y) {
      if (nom().compare(y.nom()) != 0) return false;
      if (prenom().compare(y.prenom()) != 0) return false;
      return true;
}
```

Identite.cpp

Surcharge externe

```
// surcharge externe de l'envoi sur le flot de sortie (fonction amie de la classe Identite)
ostream & operator << (ostream & S, const Identite & y) {
S << y.nom() << " " << y.prenom() << endl;
return S;
}

// surcharge externe de la réception sur le flot d'entrée (fonction amie de la classe Identite)
istream & operator >> (istream & S, Identite & y) {
S >> y.nom_ >> y.prenom_;
return S;
}
```

testIdentite.cpp

Test de la classe Identite

```
#include "Identite.h"
int main (int argc, char *argv[]) {
Identite Id1("Ducrut","Lucie"), Id2 ("Herbrant","Claude"), Id;
cout << "Entrez une identite : ";
cin >> Id;
cout << (Id1==Id2) << "  " << (Id1==Id1) << endl;
cout << Id1+Id2 << Id;
return 0;
}
```

```
Entrez une identite : Herbrand Paul
0  1
Ducrut-Herbrant Lucie
Herbrand Paul
```

Principe de l'héritage en C++

Définition d'un nouvelle classe héritant des caractéristiques (attributs) et du comportement (méthodes) d'une classe existante

Amélioration de la réutilisation (héritage de module) et de l'extensibilité (héritage de sous-typage) d'une application.

Héritage de module (ou privée)

Simple réutilisation du code de la classe parente dans la classe dérivée

Classe d'analyse de texte par héritage d'un flot de texte en lecture

Héritage de sous-typage (ou publique)

Relation « est-un » entre les deux classes (« est un cas particulier »)
Classe dérivée est un cas particulier de la classe parente

dérivation d'un rectangle en carré

Syntaxe de l'héritage en C++

Class Fille : <modificateur> Mere

Dérivation de la classe Mere en classe fille

Droits d'accès aux attributs et aux méthodes accessibles (modificateur)

pas de modifications des droits d'accès (**public**),

restriction des droits d'accès aux seules classes dérivées (**protected**),

pas de droits d'accès (**private**)

Appels des constructeurs

Appel par défaut d'un constructeur vide de la classe parente

Transmission d'une liste d'arguments à un constructeur non-vide de la classe parente

```
Fille::Fille (type1 arg1, type2 arg2,..) : Mere(arg1, arg2, …)
```

Définition des méthodes

Appel par défaut des méthodes de la classe parente

Redéfinition possible de méthodes

Appel explicite à la méthode de la classe parente (opérateur de portée)

```
Mère::nom de la méthode(paramètres)
```

Animal.h

Syntaxe de l'héritage en C++

```
class Animal {
private:
string espece;
int nb_pattes;
public:
    /** création d'une nouvelle instance de la classe Animal
     * @param type nom de l'espèce
     * @param pattes nombre de pattes */
    Animal(string type, int pattes);
    /** présentation des caractéristiques de l'animal */
    void presente();
    /** cri de l'animal */
    void crie();
};
```

Syntaxe de l'héritage en C++

Animal.cpp

```
#include "Animal.h"
/** création d'une nouvelle instance de la classe Animal
 * @param type nom de l'espèce
 * @param pattes nombre de pattes */
Animal::Animal(string type, int pattes) {
espece=type;
    nb_pattes = pattes;  }
/** présentation des caractéristiques de l'animal */
void Animal::presente() {
    cout << "je suis un représentant de l'espèce des " << espece
<< " et j'ai "
<< nb_pattes << " pattes" << endl;}
/** cri de l'animal */
void Animal::crie() {
    cout << "j'existe, donc je crie ..." << endl; }
```

Syntaxe de l'héritage en C++

Felin.h

```
#include "Animal.h"

class Felin : public Animal {
protected:
    /** est-ce un animal domestique ? */
    bool domestique;
public:
    /** Création d'une nouvelle instance de la classe Félin
     * @param type espèce de félins */
    Felin::Felin(string type);
    /** présentation des caractéristiques du félin */
    void presente();
    /** cri du félin */
    void crie();
};
```

Syntaxe de l'héritage en C++

Felin.cpp

```
#include "Felin.h"

Felin::Felin(string type) : Animal (type, 4) {
    domestique = false;
}
/** présentation des caractéristiques du félin */
void Felin::presente() {
Animal::presente();
    string etat = (domestique) ? "domestique" : "sauvage";
    cout << "je suis vraiment un animal " << etat << endl;
}
/** cri du félin */
void Felin::crie() {
cout << "je rugis, et  j'ai faim  ..." << endl;
}
```

Principes

Dérivation possible de plusieurs classes

Class Fille : <modificateur> Mere1, <modificateur> Mere2

Dérivation des classes Mere1 et Mere2 en classe fille

Appel de constructeurs

Appel par défaut d'un constructeur vide de la classe parente

Résolution des noms ambigus par l'utilisation de l'opérateur de portée

Mère1::nom de la méthode(paramètres)

Mère2::nom de la méthode(paramètres)

Syntaxe de l'héritage multiple en C++

Herbivore.h
Carnivore.h

```
class Herbivore : public Animal {
protected:
bool ruminant;
public:
Herbivore::Herbivore(string type, int pattes, bool ruminant);
void presente();};

class Carnivore : public Animal {
protected:
bool doux;
public:
Carnivore::Carnivore(string type, int pattes, bool doux);
void presente();
};
```

Herbivore.cpp

Syntaxe de l'héritage multiple en C++

```
#include "Herbivore.h"
/** Création d'une nouvelle instance de la classe Herbivore
 * @param type espèce de Herbivore */
Herbivore::Herbivore(string type, int pattes, bool r) : Animal
(type, pattes) {
ruminant = r;
}
/** présentation des caractéristiques de l'herbivore */
void Herbivore::presente() {
Animal::presente();
    cout << "j'aime les légumes";
    if (ruminant == true) cout << " et je rumine";
    else cout << " et je ne rumine pas";
    cout << endl;
}
```

Carnivore.cpp

Syntaxe de l'héritage multiple en C++

```
#include "Carnivore.h"
/** Création d'une nouvelle instance de la classe Carnivore
 * @param type espèce de carnivore */
Carnivore::Carnivore(string type, int pattes, bool d) : Animal
(type, pattes) {
doux = d;
}
/** présentation des caractéristiques du carnivore */
void Carnivore::presente() {
Animal::presente();
    cout << "j'aime la viande";
    if (doux == false) cout << " et je suis cruel";
    else cout << " et je suis doux";
    cout << endl;
}
```

Syntaxe de l'héritage multiple en C++

Omnivore.h
Omnivore.cpp

```
#include "Herbivore.h"
#include "Carnivore.h"
class Omnivore : public Herbivore, public Carnivore {
public:
Omnivore::Omnivore(string type, int pattes);
void presente();
void crie(); };

#include "Omnivore.h"
Omnivore::Omnivore(string type, int pattes) : Carnivore (type,
pattes, true), Herbivore(type, pattes, false) {}
void Omnivore::presente() {
      Herbivore::presente();Carnivore::presente();}
void Omnivore::crie() { Herbivore::crie(); }
```

testOmnivore.cpp

Syntaxe de l'héritage multiple en C++

```
#include "Omnivore.h"

int main() {
Omnivore a("blaireaux", 4);
    a.presente();
    a.crie();
}
```

```
je suis un représentant de l'espèce des blaireaux et j'ai 4 pattes
j'aime les légumes et je ne rumine pas
je suis un représentant de l'espèce des blaireaux et j'ai 4 pattes
j'aime la viande et je suis doux
j'existe, donc je crie ...
```