Cours 2 : Les classes

Rabii EL GHORFI

Module : Technique de programmation avancées

Département : Mathématiques, informatique et géomatique (MIG)

EHTP 2017-2018

Principaux axes du cours

- Classes et objets
- Champs et méthodes
- Constructeurs et destructeurs
- Constructeurs par copie
- Fonctions amies
- Surcharge d'opérateurs

Les classes et les objets

Définition : **Classe**

- Sert à regrouper les données
- Sert à associer les fonctions aux objets de la classe
- Permet de restreindre l'accès à certaines données

Définition : **Objet**

- Est une instance de la classe
- On peut créer plusieurs objets à partir d'une même classe

```
Rectangle R1;
Rectangle R2;
```

Principe (1)

- Les détails d'implémentation n'ont pas à être connus par l'utilisateur

Exemple : On crée un objet de type rectangle

```
int main() {
	Rectangle R1; // R1 est un objet de type Rectangle
	int a = R1.calculer_Perimetre();
	return 0;
}
```

- La fonction Calculer_perimetre() renvoie le périmètre
- Les détails de cette fonction se trouve dans la classe rectangle

Principe (2)

- Pour pouvoir créer des objets de type Rectangle, il va falloir coder la classe Rectangle

Exemple : On crée la classe rectangle

```
class Rectangle {
public:
    double longueur;
    double largeur;
    double calculer_Perimetre();
};
```

- La classe Rectangle contient uniquement le corps de la classe

Principe (3)

- Maintenant, il reste à implémenter la fonction Calculer_perimetre();

Exemple : On implémente la fonction

```
double Rectangle::calculer_Perimetre(){
        return 2 * longueur + 2 * largeur;
}
```

- Dans la fonction calculer_Perimetre(), on peut utiliser toutes les variables propres à la classe Rectangle
- L'implémentation de la fonction se fait à l'extérieur de la classe

Exemple basique d'une classe

```
class Rectangle {
public:
    double longueur;
    double largeur;
    double calculer_Perimetre();
};
double Rectangle::calculer_Perimetre(){
        return 2 * longueur + 2 * largeur;
}
int main() {
        Rectangle R1; // R1 est un objet de type Rectangle
        return 0;
}
```

Champs et méthodes (1)

Définition : **Champ**

- Est une variable associée à l'objet
- Aussi appelé attribut

Définition : **Méthode**

- Est une fonction membre de l'objet
- Comme une fonction classique, elle peut contenir des paramètres

```
Rectangle R1;
R1.longeur = 40;            // longueur est un champ
R1.calculer_Perimetre();    // calculer_Perimetre() est une méthode
```

Champs et méthodes (2)

- Les méthodes sont déclarées dans une classe
- Le nom complet de la méthode Test de la classe T est T::Test
- Les méthodes peuvent accéder aux champs des objets de la classe. Ce sont les seules à pouvoir le faire (ou presque) !
- L'objet sur lequel elle sont appelées, n'apparait pas explicitement dans la définition des méthodes
- Par défaut, les paramètres apparaissant dans la définition d'une méthode sont ceux avec lesquelles la méthode sera appelée

Déclaration des champs et méthodes

- Les champs et les méthodes sont déclarés à l'intérieur de la classe
- Les méthodes peuvent être codées à l'intérieur ou à l'extérieur de la classe

Exemple : 2 façons d'implémenter la méthode `calculer_Perimetre()`

```
class Rectangle {
public:
    double longueur;
    double largeur;
    double calculer_Perimetre();
};

double Rectangle::calculer_Perimetre(){
    return 2 * longueur + 2 * largeur;
} // Codée à l'extérieur
```

```
class Rectangle {
public:
    double longueur;
    double largeur;
    double calculer_Perimetre(){
    return 2 * longueur + 2 * largeur;
    } // Codée à l'intérieur
};
```

Accès aux champs et méthodes

- Pour accéder aux attributs et méthodes, on utilise la notation :
  - « . » pour les objets
  - « -> » pour les pointeurs

Exemple :

```
int main() {
    Rectangle R1; // R1 est une instance statique
    Rectangle *R2 = new Rectangle; // R2 est une instance dynamique
    R1.calculer_Perimetre();
    R2->calculer_Perimetre();
    return 0;
}
```

Visibilité des champs et méthodes (1)

- La visibilité des champs et des méthodes est définie dans l'interface de la classe. Trois mots-clés permettent de la préciser :
  - Public : autorise l'accès pour tous
  - Private : restreint l'accès aux méthodes de cette classe
  - Protected : comme private, restreint l'accès aux méthodes de cette classe, sauf que l'accès est aussi autorisé aux méthodes des classes qui héritent (directement ou indirectement) de cette classe.

Visibilité des champs et méthodes (2)

Exemple :

```
class Personne {
public:
    string nom;
    string prenom;
private:
    int age;
protected:
    string adr;
};
```

```
int main() {
    Personne P1; // Déclaration d'une personne P1
    cout << "Le nom de P1 : " << P1.nom << endl;
    cout << "L'age de P1 : " << P1.age << endl;
    cout << "L'adresse de P1 : " << P1.adr << endl;
    return 0;
}
```

- L'affichage de l'âge donne une erreur car le champ age est private
- L'affichage de l'adresse donne une erreur car le champ adr est protected

Constructeurs (1)

Définition : **Constructeur**

- Méthode particulière appelée automatiquement à chaque création d'un objet

Principe :

- Pour appeler un constructeur de la classe MaClasse, il suffit de faire suivre le nom de l'objet O par la liste des arguments

```
MaClasse O (arg1, arg2, … );
```

- Lorsqu'aucun argument n'est donné, le constructeur sans argument est appelé

```
MaClasse O;
```

Constructeurs (2)

Exemple : Un constructeur qui initialise la longueur et la largeur du rectangle

```
class Rectangle {
    public:
    Rectangle(double init_longeur, double init_largeur);
    double longueur = 0;
    double largeur = 0;
};

Rectangle::Rectangle(double init_longeur, double init_largeur) {
    longueur = init_longeur;
    largeur = init_largeur;
}
```

Constructeurs (3)

Exemple : Un constructeur qui initialise la longueur et la largeur du rectangle

```
int main() {
    Rectangle R1; // Appel au constructeur par default
    Rectangle R2(10, 20); // Appel au constructeur Rectangle(double, double)
    cout << "Longueur : " << R1.longueur << "Largeur : " << R1.largeur << endl;
    cout << "Longueur : " << R2.longueur << "Largeur : " << R2.largeur << endl;
    return 0;
}
```

- La longueur et la largeur affichées de R1 sont (0, 0)
- La longueur et la largeur affichées de R2 sont (10, 20)

Destructeurs (1)

Définition : **Destructeur**

- Méthode particulière appelée automatiquement à la destruction d'un objet

Principe :

- Son nom est de la forme :

```
~MaClasse()
```

- Par défaut, le destructeur ne fait rien. Toutefois, on peut lui donner un comportement spécifique
- Il est indispensable lorsque l'on a besoin de faire de l'allocation dynamique (quand il y'a des pointeurs dans la classe)

Destructeurs (2)

Exemple : Déclaration d'un destructeur

```
class Rectangle {
    public:
    ~Rectangle();
    double longueur = 0;
    double largeur = 0;
    MaStructure * ptr;
};

Rectangle::~Rectangle() {
    delete [] ptr; // Objectif du destructeur : libérer tous les pointeurs
}
```

Constructeurs par copie (1)

Définition : **Constructeur par copie**

- Méthode particulière appelée lors de l'instanciation d'un objet avec en argument un objet du même type

Principe :

- Son nom est de la forme :

```
~MaClasse(Const MaClasse &s)
```

- Le rôle d'un constructeur par copie est de permettre l'instanciation d'un nouvel objet dans le même état qu'un objet existant (clonage)
- Le constructeur copieur par défaut fait une initialisation champ à champ
- Comme pour le destructeur, il est utile d'implémenter un constructeur copieur uniquement dans les allocations dynamiques

Constructeurs par copie (2)

Exemple : Déclaration d'un constructeur par copie

```
class Rectangle {
    public:
    Rectangle(const Rectangle &s);
    double longueur = 0, largeur = 0;
    MaStructure * ptr;
};

Rectangle::Rectangle(const Rectangle &s) {
    longueur = s.longueur; largeur = s.largeur;
    ptr = new MaStructure[100];
    for (int i=0; i<100; i++)
        ptr[i] = s.ptr[i];
};
```

Constructeurs par copie (3)

Exemple : Un constructeur par copie

```
int main() {
    Rectangle R1;
    Rectangle R2(R1); // Appel au constructeur par copie
    Rectangle R3=R1; // Appel au constructeur par copie
    cout << "Longueur : " << R2.longueur << "Largeur : " << R3.largeur << endl;
    fonction(R1);
    return 0;
}
void fonction(Rectangle R4) {…} // Appel au constructeur par copie
```

- La longueur et la largeur affichées sont celles de R1

Exemple : Tableau d'étudiants

- Soit une classe Tab_etud qui permet de gérer un tableau de structures de type `Etudiant`
  - Chaque structure `Etudiant` contient le nom et la moyenne d'un étudiant

- Cette classe contient :

  - 2 champs : taille et un pointeur `ptr` qui pointe sur la structure `Etudiant`
  - 5 méthodes : element, supprimer, affiche, existe et ajout
  - 3 méthodes spéciales : constructeur, constructeur copieur et destructeur

Remarque : `ptr` joue le rôle d'un tableau de type Etudiant : `ptr[0]` pointe vers le premier étudiant `ptr[1]` le second etc …

La classe Tab_etud

```
class Tab_etud {
private:
	Etudiant * ptr;

protected:
	int taille;

public:
	Tab_etud(int);
	Tab_etud(const Tab_etud &s);
	~Tab_etud();

	bool existe(char * );
	void ajout(Etudiant ) ;
	Etudiant element(int );
	void supprimer(int );
	void affiche();
	};
```

```
struct Etudiant {
		char nom[20];
		float moyenne;
};
```

Les méthodes de Tab_etud

```
Etudiant Tab_etud::element(int position) {
	Etudiant e = ptr[position];
	return e;
}

void Tab_etud::supprimer(int position) {
for(int i = position; i < taille - 1; i++)
	{ptr[i] = ptr[i+1];}
taille = taille -1;
}

void Tab_etud::affiche() {
cout << " Voici la liste :\n";
for (int i = 0; i < taille ; i++)
	cout << " Nom : " << ptr[i].nom <<
	 " Moyenne : " << ptr[i].moyenne ;
}
```

```
bool Tab_etud::existe(char * nom) {
for(int i =0; i< taille; i++)
	{//strcmp renvoie 0 si identique
	if( strcmp(ptr[i].nom, nom) == 0)
		return true;
	}
return false;
}

void Tab_etud::ajout(Etudiant e) {
if( !existe(e.nom) ) {
	// Ajout dans la dernière position
	ptr[taille] = e;
	taille = taille + 1;
	}
}
```

Les méthodes spéciales de Tab_etud

```
Tab_etud::Tab_etud(int n) {
	ptr = new Etudiant[n];
	taille = 0;
}

Tab_etud::Tab_etud(const Tab_etud &s) {
	taille = s.taille;
	ptr = new Etudiant[100];
	for(int i=0;i<taille;i++)
		{ptr[i] = s.ptr[i];}
}

Tab_etud::~Tab_etud() {
	// suppression du pointeur
	delete [] ptr;
}
```

Fonctions amies

Définition : **Fonctions amies**

- Si une fonction F est amie « friend » d'une classe C1, alors F peut accéder aux champs privés de C1.
- Si une classe C2 est amie de C1, alors toutes les fonctions membres de C2 peuvent accéder aux champs privés de C1.

Exemple :

```
class C1 {
        friend void F() ;
        friend class C2 ;
        …
};
```

Surcharge d'opérateurs (1)

- Il est possible de surcharger les operateurs (+, -, [ ], =, ==, …) de C++
- Il existe 2 façons de faire :

Fonction globale : A op B qui est vue comme l'application d'une fonction op à deux arguments A et B

```
MaClasse operator + ( MaClasse A, MaClasse B )
```

Fonction membre : A op B qui est vue comme l'application d'une fonction op à un argument B de l'objet A

```
MaClasse MaClasse :: operator + ( MaClasse B )
```

- Réalisons la surcharge de (+) de la classe Tab_op (identique à Tab_etudiants)

Surcharge d'opérateurs (2)

- Soit une classe Tab_op qui dérive de la classe Tab_etud (voir Héritage)
  - Cette classe contient les mêmes champs et méthodes que Tab_etud
- Dans cette classe, on définit :
  - Un opérateur + permettant de concaténer deux tableaux. Ça consiste à mettre les tableaux, l'un à la suite de l'autre
  - Un opérateur = permettant de copier un tableau dans un autre

Objectif : Implémenter les opérateurs + et = et réaliser des opérations du genre :

```
Tab_op A(100), B(100);
Tab_op C = A + B; // Utilisation du constructeur de l'objet C
Tab_op D(100);
D = A + B; // Utilisation de l'opérateur = de l'objet D
```

Surcharge d'opérateurs (3)

- Par fonction globale :

```
//  Opérateur + :
Tab_op operator + ( Tab_op a, Tab_op b ) {
        // Principe : on retourne a + b
        Tab_op result(a); // Copie de a
        for( int i = 0; i < b.taille; i++ )
                {result.ajout(b.element(i));} // Ajout des éléments de b
        return result;
}
```

Remarque : Dans les fonctions globales on prend 2 paramètres. Les fonctions globales sont définies amies et accèdent aux champs privés de a et b

Surcharge d'opérateurs (4)

- Par fonction membre :

```
//  Opérateur + :
Tab_op Tab_op::operator + ( Tab_op  b ) {
      // Principe : on retourne *this + b
      Tab_op result(* this); // Copie this
      for( int i = 0; i < b.taille; i++ )
            {result.ajout(b.element(i));}
            // Ajout des éléments de b
      return result;
}
```

```
//  Opérateur = :
void Tab_op::operator = (Tab_op b)
{
taille = 0;
for (int i = 0; i < b.taille; i++)
      this->ajout(b.element(i));
      // Ajout des éléments de b
}
```

Remarque : Dans les fonctions membres a = this. Le retour de a + b représente le résultat. Le retour de a = b n'est pas important, il faut toutefois mettre b dans a