2019-2020

Programmation C++

DR. AMINA JARRAYA
JARRAYA.AMINA.FST@GMAIL.COM

15/11/2019

Plan

- **Constructeurs et destructeurs**
- **La surcharge**
- **Pointeurs et objets**

Constructeurs et Destructeurs

Constructeur: Définition

- Un constructeur est une fonction-membre déclarée du même nom que la classe, et sans type :

```
Nomclasse(<paramètres>);
```

- A l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.
- Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils diffèrent par le nombre ou le type des paramètres (Surcharge de constructeurs).
- Un constructeur sans paramètre ou dont tous les paramètres ont des valeurs par défaut, s'appelle ***constructeur par défaut***.

Initialisation des attributs (1)

- Un constructeur peut contenir une liste d'initialisation des attributs.

Syntaxe générale:

```
NomClasse (liste_paramètres)
// liste d'initialisations
: attribut1(...), attribut2(...),...
{ // autres opérations }
```

- Cette section introduite par « : » est optionnelle mais recommandée.

Initialisation des attributs (2)

- Les attributs non initialisés dans cette section:
  - Prennent une valeur par défaut si ce sont des objets
  - Restent indéfinis s'ils sont de type de base;
- Les attributs initialisés dans cette section peuvent être changés dans le corps du constructeur.

Exemple:

```
Rectangle(double h, double L)
   : hauteur(h) //initialisation
{
   // largeur a une valeur indéfinie jusqu'ici
   largeur = 2.0 * L + h; // par exemple...
   // la valeur de largeur est définie à partir d'ici
}
```

Initialisation des attributs (3)


```
// ...
class Rectangle {
public:
  Rectangle(double h, double L)
    : hauteur(h), largeur(L)
  {}
  double surface() const
    { return hauteur * largeur; }
  // accesseurs/modificateurs
  // ...
private:
  double hauteur;
  double largeur;
};

int main()
{
  Rectangle rect1(3.0, 4.0);
  // ...
```

Initialisation des objets

- L'utilité principale du constructeur est d'effectuer des initialisations pour chaque objet nouvellement créé.

```
Rectangle r;
// appel automatique du constructeur par défaut
// équivaut à: Rectangle r = Rectangle ();
Rectangle r(1.0, 2.0);
// appel du constructeur paramétré
// équivaut à: Rectangle r = Rectangle(1.0, 2.0);
```

Constructeurs par défaut (1)

- Dans une classe, il est possible ne pas mettre de constructeur.
- Dans ce cas, lors de la déclaration d'une variable de cette classe, l'espace mémoire est réservé mais les attributs de l'objet ne reçoivent pas de valeur de départ : on dit qu'elles ne sont pas initialisées.
- Un constructeur par défaut est un constructeur qui n'as pas de paramètres ou dont tous les paramètres ont des valeurs par défaut.

Constructeurs par défaut (2)

Exemples:

A:

```
class Rectangle {
private:
  double h; double L;
  // suite ...
};
```

B:

```
class Rectangle {
private:
  double h; double L;
public:
  Rectangle()
  : h(0.0), L(0.0)
  {}
  // suite ...
};
```

C:

```
class Rectangle {
private:
  double h; double L;
public:
  Rectangle(double h=0.0,
            double L=0.0)
   : h(h), L(L)
  {}
  // suite ...
};
```

D:

```
class Rectangle {
private:
  double h; double L;
public:
  Rectangle(double h,
            double L)
    : h(h), L(L)
  { }
  // suite ...
};
```

Constructeurs par défaut (3)

	Constructeur par défaut	Rectangle r1;	Rectangle(1.0, 2.0)
A	Constructeur par défaut par défaut	? ?	Illicite!
B	Constructeur par défaut explicitement déclaré	0 0	Illicite!
C	Un des trois constructeurs est par défaut	0 0	1 2
D	Pas de constructeurs par défaut	Illicite!	1 2

Destructeur (1)

- Un destructeur est une fonction-membre déclarée du même nom que la classe mais précédé d'un tilda (~) et sans type de retour ni paramètre :

```
~Nom_classe();
```

- Le destructeur est automatiquement appelé pour chaque objet de la classe `Nom_classe` déclaré dans un bloc, à l'issue de l'exécution de ce bloc (à la fin du cycle de vie d'un objet)

- Il permet de libérer un espace mémoire alloué par l'objet

Destructeur (2)

- Exemple

```
class MaClasse {
// ...
~MaClasse();
};
```

- Est-il possible d'invoquer explicitement le destructeur d'une classe ?

```
class Test {
    public:
        ~Test() { std::cout << "Destruction\n"; }
};
int main() {
Test t;
t.~Test();    // appelle explicite du destructeur
} // ici, le compilateur appelle à nouveau le destructeur
```

Sortie du programme :

Destruction ➔ **appel explicite du destructeur**

Destruction ➔ **appel implicite du destructeur**

Destructeur (3)

- Besoin : SI l'initialisation des attributs d'une instance implique la mobilisation de ressources : fichiers, périphériques, portions de mémoire (pointeurs), etc.
  - Il est alors important de libérer ces ressources après usage !
- Donc, existe-il d'autres cas pour les quelles on aurait besoin d'un destructeur?
- Est-ce que le destructeur peut avoir d'autres rôles à part la libération de ressources?
- Comment faire si on veut compter le nombre des instances actives dans un programme?

Destructeur (3)


```
class Rectangle {
  //...
  Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
    ++compteur; }
  // ...
```

```
int main()
{
  // compteur = 0
  Rectangle r1;
  // compteur = 1
  {
    Rectangle r2;
    // compteur = 2
    // ...
  }
  // compteur = 2
  return 0;
} // compteur = 2
```

Destructeur (4)


```
class Rectangle {
  //...
  Rectangle(): hauteur(0.0), largeur(0.0) { //constructeur
    ++compteur; }
  ~Rectangle() { --compteur; } // destructeur
  // ...
```

```
int main()
{
  // compteur = 0
  Rectangle r1;
  // compteur = 1
  {
    Rectangle r2;
    // compteur = 2
    // ...
  }
  // compteur = 1
  return 0;
} // compteur = 0
```

La Surcharge

Surcharge des méthodes

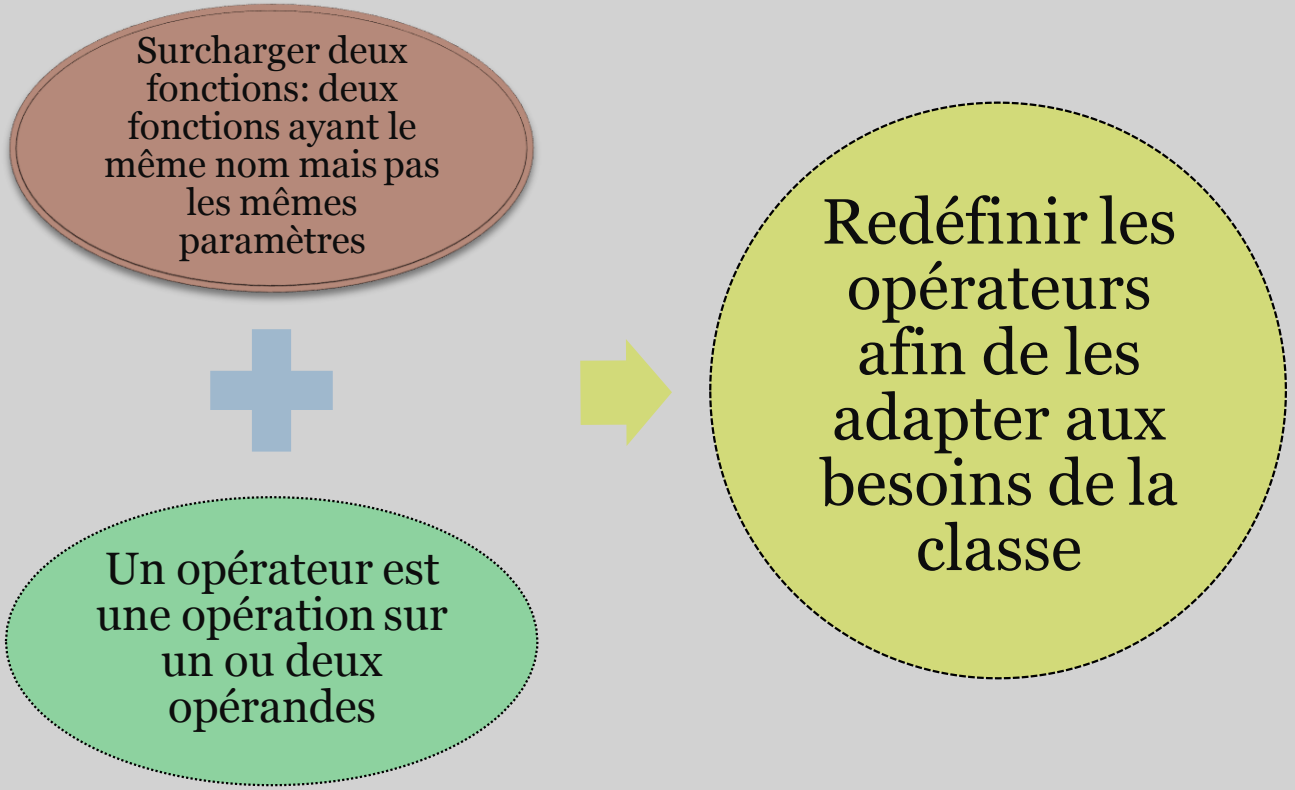
- Chaque méthode possède une signature :
  - nom de la méthode
  - paramètres admis en entrée (et ordre des paramètres)
  - résultat fourni en sortie (facultatif)

- Une même classe peut contenir la même méthode dotée de signatures différentes

```
float calculer();
int calculer(int);
```

Surcharge des opérateurs (1)

- C'est quoi la surcharge des opérateurs ?



Surcharge des opérateurs (2)

- La majorité des opérateurs est surchargeable à part quelques uns tels que:
  - ::
  - .
  - ?:
  - & (de prise de référence)

Remarque :

- On ne peut utiliser que les opérateurs déjà existants (c'est normal, c'est de la surcharge)
- Il faut conserver la pluralité (unaire, binaire) de l'opérateur initial.

Surcharge des opérateurs (3)

- Syntaxe:

```
<Type_de_retour> operatoroperateur(<operande>);
```

Exemple1: opérateur binaire

```
// Déclaration
Complexe operator+(Complexe g);

// Définition
Complexe Complexe::operator+(Complexe g)
{
  return Complexe(re + g.re, im + g.im);
}
// Utilisation
Complexe z1(0.0, 1.0);
Complexe z2;
Complexe z3 = z1 +z2;
```

appel du constructeur

Exemple2: opérateur unaire

```
// Déclaration
Complexe operator-();

// Définition
Complexe Complexe::operator-()
{
   return Complexe(-re, -im);
}
// Utilisation  Complexe
z1(0.0, 1.0);  Complexe
z2 = -z1;
```

- Avec :

```
class Complexe {
....
private:
float re, im; // parties réelle et imaginaire };
```

Surcharge des opérateurs (4)

- a + b correspond à operator+(a, b) ou a.operator+(b)
- b + a operator+(b, a) ou b.operator+(a)
- -a operator-(a) ou a.operator-()
- cout << a operator<<(cout, a) ou cout.operator<<(a)
- a = b a.operator=(b)
- a += b operator+=(a, b) ou a.operator+=(b)
- ++a operator++(a) ou a.operator++()
- not a operator not(a) ou a.operator not() ou operator!(a) ou a.operator!()

Pointeurs et objets

Pointeurs et objets

- On peut naturellement utiliser des pointeurs sur des types classes.
- Nous pourrons ainsi utiliser des objets dynamiques.
- Exemple de la classe Complexe

```
Complexe *pc = new Complexe; // etc...
```

- De la même manière que l'utilisation statique:
  - si new est utilisé avec un type classe, le constructeur par défaut (s'il existe) est appelé automatiquement,
  - il est possible de faire un appel explicite à un constructeur:

```
Complexe *pc = new Complexe(1.0, 2.0);
```

Accès à un objet pointé

- L'accès aux données et fonctions-membres d'un objet pointé peut se faire grâce à l'*opérateur flèche* ->.
- Par exemple:

```
pc -> re          \\ plutôt que (*pc).re
pc -> Affiche()   \\ plutôt que (*pc).Affiche()
```

Exercice 1

- Soient les deux classes Personne et Voiture définies comme suit: une personne possède une voiture

```
#include <iostream>

using namespace std;
class Voiture{

    public: string modele;
            Voiture (string m):modele(m){

            }
};
class Personne {
    public: int age;
            Voiture v;
            Personne(int a, string m):age(a),v(m){

            }
};
```

Création d'un objet d'une manière statique

Exercice 1 (suite)

Commentez le code suivant et donnez le résultat de son exécution:

```
int main()
{
    cout<<"Hello"<<endl;
    Personne p (30,"Citroen");
    Voiture vo("Renault");
    Personne q(p);
    q.v = vo;
    cout<<q.v.modele<<endl;
    cout<< p.v.modele<<endl;
    vo.modele="Peugeot";
    cout<<q.v.modele<<endl;
    cout<<vo.modele<<endl;
    cout<<"Bye";


    return 0;
}
```

Affectation de deux objets ➔ copie des valeurs des attributs de l'objet **p** vers les attributs de l'objet **q**

Modification du modèle de la voiture de l'objet q qui va prendre le même modèle de la voiture vo

➔ Ceci n'impactera pas la modification du modèle de voiture de l'objet p parce que l'objet v (attribut de la classe personne) est déclaré statiquement.

Exercice 1 (suite)

Commentez le code suivant et donnez le résultat de son exécution:

```
int main()
{
    cout<<"Hello"<<endl;
    Personne p (30,"Citroen");
    Voiture vo("Renault");
    Personne q(p);
    q.v = vo;
    cout<<q.v.modele<<endl;
    cout<< p.v.modele<<endl;
    vo.modele="Peugeot";
    cout<<q.v.modele<<endl;
    cout<<vo.modele<<endl;
    cout<<"Bye";


    return 0;
}
```

Sortie du programme

```
Hello
Renault
Citroen
Renault
Peugeot
Bye
```

Exercice 2

Modification de la classe Personne comme suit:

```
#include <iostream>

using namespace std;
class Voiture{

    public: string modele;
            Voiture (string m):modele(m){
            }
};
class Personne {
    public: int age;
            Voiture * v;
            Personne(int a, string m):age(a){
                v = new Voiture (m);
            }
};
```

Création d'un objet d'une manière dynamique

Exercice 1 (suite)

Commentez le code suivant et donnez le résultat de son exécution:

```
int main()
{
    cout<<"Hello"<<endl;
    Personne p (30,"Citroen");
    Personne q(p);

    cout<<q.v->modele<<endl;
    cout<< p.v->modele<<endl;

    q.v->modele="Peugeot";
    q.age=20;
    cout<<q.v->modele<<endl;
    cout<< p.v->modele<<endl;
    cout<<q.age<<endl;
    cout<<p.age<<endl;
    cout<<"Bye";

    return 0;
}
```

Affectation de deux objets ➔ copie des valeurs des attributs de l'objet **p** vers les attributs de l'objet **q**

```
Hello
Citroen
Citroen
Peugeot
Peugeot
20
30
Bye
```

Si le modèle de voiture de l'objet q se modifie ➔ ceci impactera automatiquement la modification du modèle de voiture de l'objet p parce que l'objet v (attribut de la classe personne) est déclaré dynamiquement.