

Chapitre 6

Les Propriétés des fonctions membres

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

112

1. Fonctions membres

1.1. Définitions

Les fonctions membres (y compris les constructeurs) peuvent:

- être surdéfinies,
- avoir des arguments par défaut,
- être `public` ou `private`,
- être `inline`.

Les fonctions membres ont accès à tous les membres de tous les objets de la classe.

La protection est au niveau de la classe, pas au niveau des instances.

1.2. Exemple de fonctions membres

```
#include <iostream>

class compte {
    char nom[20];
    double actif;
public:
    compte (char*,double); // constructeur
    ~compte(); // destructeur
    bool PlusRiche(const compte& C) { // qui est plus riche que l'autre?
        return actif > C.actif;
    }
    void affiche() { // affiche les membres données
        std::cout << "nom: " << nom << " " << "actif: " << actif << "$\n";
    }
};

compte::compte(char* name,double d) { // constructeur
    if ((name!=NULL)&&(strlen(name)<20)) { // on vérifie que la chaîne ne dépasse pas 20
        //\0 compris. On vérifie aussi qu'elle contient quelque chose!
        strcpy(nom,name); // si les tests sont bons, copie des données.
        actif = d;
    }else{
        nom[0] = '\0'; // sinon, nous avons préféré initialiser avec des valeurs nulles.
        d = 0.0; // c'est juste un choix, vous pouviez aussi sortir en signalant l'erreur.
    }
}
```

Chapitre 6 : Les propriétés des fonctions membres

```
// destructeur
compte::~compte() {
    // pas d'intérêt dans notre exemple
}

int main() {

    // création de deux comptes.
    compte A("John",280.67);
    compte B("Smith",89.78);
    // test de la richesse de l'un par rapport à l'autre.
    if (A.PlusRiche(B)){
        A.affiche(); // vrai => affiche A
    }else{
        B.affiche(); // faux => affiche B
    }
    // on peut tester aussi à partir de B
    if (B.PlusRiche(A)){
        B.affiche(); // vrai => affiche: B
    }else{
        A.affiche(); // faux => affiche: A
    }
    return 0;
}
```

En sortie:
nom: John actif: 280.67\$
nom: John actif: 280.67\$

2. Fonctions membres `inline`

Revoir la partie du cours à propos des fonctions `inline` en général

Défaut: - membres de données → une copie pour **chaque** objet

- fonctions membres → une copie pour **tous** les objets

Dans le cas des fonctions membres `inline`:

- fonctions membres → une copie pour **chaque** objet
définition de la méthode dans le même fichier que la classe.

Avantages: un peu plus rapide

Inconvénients: - code plus volumineux

- la définition de la fonction devient visible (elle doit toujours être définie dans le fichier utilisé) => problème d'encapsulation, car la fonction est visible partout. Ce qui est contraire au principe même de l'encapsulation.

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

116

2.1. Fonctions membres `inline` membres de la classe

Si on définit une fonction membre dans la classe, elle est automatiquement `inline`.

```
class compte {
public:
    void affiche() {
        cout << " fonction membre définie dans la classe => automatiquement inline\n";
    }
};
```

2.2. Fonctions `inline` à l'extérieur de la classe

Pour définir une fonction `inline` à l'extérieur de la classe, on utilise le mot clé `inline`.

```
class compte {
public:
    void affiche();
};

inline void compte::affiche(); {
    cout << " fonction membre définie à l'extérieur de la classe => \
    si inline, il faudra ajouter le mot clé inline\n";
}
```

© Mohamed N. Lokbani

v3.01

POO avec C++

3. Méthodes statiques

Méthode inline → une copie par objet
autres → une copie par classe

Méthodes statiques:

- une copie par classe,
- pour les fonctions associées aux classes, plutôt qu'à un objet particulier,
- accès seulement aux membres statiques,
- peut être appelée même si aucun objet n'a été créé.

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

118

```
#include <iostream>
using namespace std;
class compte{
    double actif;
    static int nbcomptes;

public:
    // une autre façon pour initialiser un membre données (voir actif) on calcule le nombre de comptes créés.
    compte(double d):actif(d){nbcomptes++;}

    // appel du destructeur, on décrémente le nombre de comptes
    ~compte(){nbcomptes--;}

    void affiche() {
        cout << "nbcomptes: " << nbcomptes << " " << "actif: " << actif << endl;
    }

    // une fonction statique
    static void fonction_statique () {
        cout << "nbcomptes: " << nbcomptes << endl;
        // cout "actif: " << actif << endl; ← erreur, car actif n'est pas statique
    };

    int compte::nbcomptes = 0;
```

© Mohamed N. Lokbani

v3.01

POO avec C++

```
int main() {  
    compte::fonction_static(); // correct.  
    // compte::affiche(); ← erreur, aucun objet n'a été créé.  
    // cout << compte::nbcomptes << endl; ← erreur nbcomptes étant private.  
    compte C(260.8);  
    compte::fonction_static(); // correct.  
    C.fonction_static(); // correct.  
    C.affiche(); // correct.  
}
```

En sortie:

```
nbcomptes: 0  
nbcomptes: 1  
nbcomptes: 1  
nbcomptes: 1      actif: 260.8
```

Réécrire le code précédent afin de mettre en évidence l'utilité du destructeur ainsi déclaré.

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

120

4. Objets comme arguments

Revoir la partie du cours à propos du passage des arguments par valeur, par pointeur et par référence.

4.1. Passage par valeur

Une copie locale du paramètre réel

```
#include <iostream>  
class point{  
    int x,y;  
public:  
    point(int v,int w):x(v),y(w){}  
    bool egal(const point);  
};  
bool point::egal(const point A) { // ← passage par valeur  
    return ((x==A.x) && (y==A.y));  
}  
int main() {  
    point a(1,2),b(3,4);  
    if (a.egal(b)){ // ← passage par valeur  
        std::cout << "idem \n";  
    } else {  
        std::cout << "différent \n";  
    }  
    return 0;  
}
```

© Mohamed N. Lokbani

v3.01

POO avec C++

Sortie:
différent

4.2. Passage par pointeur

Il n'y a pas de copie locale du paramètre réel. Il y a un lien avec un objet qui existe déjà

```
#include <iostream>

class point{
    int x,y;
public:
    point(int v,int w):x(v),y(w){}
    bool egal(const point*);
};

bool point::egal(const point* A) { // ← passage par pointeur
    return ((x==A->x) && (y==A->y));
}

int main() {
    point a(1,2),b(3,4);
    if (a.egal(&b)){ // ← passage par pointeur, on passe l'adresse
        std::cout << "idem \n";
    } else {
        std::cout << "différent \n";
    }
    return 0;
}
```

Sortie:
différent

4.3. Passage par référence

Il n'y a pas de copie locale du paramètre réel. Il y a un alias à un objet qui existe déjà

```
#include <iostream>

class point{
    int x,y;
public:
    point(int v,int w):x(v),y(w){}
    bool egal(const point&);
};

bool point::egal(const point& A) { // ← passage par référence
    return ((x==A.x) && (y==A.y));
}

int main() {
    point a(1,2),b(3,4);
    if (a.egal(b)){ // ← passage par référence
        std::cout << "idem \n";
    } else {
        std::cout << "différent \n";
    }
    return 0;
}
```

Sortie:
différent

5. Objet comme valeur de retour d'une fonction

Une fonction membre peut retourner un objet par:

- valeur
- référence
- pointeur

Cet objet pourra être

- de la même classe (accès à ses membres private)
- d'une classe différente (accès à ses membres public)

5.1. Retour par valeur

```
#include <iostream>

class compte {
    double actif;

public:
    compte(double d):actif(d){}
    compte copie(){
        compte temp(0.0); // ← création locale temp
        temp.actif = actif;
        return temp; // ← retourne la valeur temp
    } // ← destruction de temp
    void affiche() {
        std::cout << "actif: " << actif << std::endl;
    }
};

int main() {

    compte A(250.89);
    compte B(120.5);
    A.affiche();
    A = B.copie();
    A.affiche();

    return 0;
}
```

Sortie:

```
actif: 250.89
actif: 120.5
```

5.2. Retour par référence

- Si référence à un objet local, l'objet local est détruit à la fin de la fonction => référence vide. 🚫👉👎

```
compte& compte::copie() {
    compte temp(0.0); // ← création locale temp
    temp.actif = actif;
    return temp; // ← retourne une référence à temp
} // ← destruction de temp

int main() {
    compte A(250.89);
    compte& B = A.copie();
    return 0;
}
```

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

126

B est une référence à une variable locale (temp) qui n'existe plus. Aucune erreur à la compilation, si vous avez activé les bonnes options de compilation (-Wall en g++), le compilateur signale un warning (avertissement) du style:

```
In method 'class compte & compte::copie()':
warning: reference to local variable 'temp' returned
```

- Si la référence est à un objet qui n'est pas local à la fonction, pas de problème.

```
compte& compte::copie(compte& F) { // ← F n'est pas un objet local
    F.actif = actif;
    return F;
}
```

© Mohamed N. Lokbani

v3.01

POO avec C++

6. Autoréférence this (mot clé)

this dans une fonction membre donne l'adresse de l'objet qui fait l'appel (ou objet receveur).

- utilisable par les méthodes non statiques.
- utile lors de la surcharge des opérateurs, pour éviter l'affectation de l'objet à lui-même (voir un peu plus loin dans le cours).
- permet des appels en cascade de fonctions membres.

© Mohamed N. Lokbani

v3.01

POO avec C++

Chapitre 6 : Les propriétés des fonctions membres

128

```
#include <iostream>

class Test {

public:
    Test(int a = 0) : x(a) {} ;
    void affiche();

private:
    int x;

};

void Test::affiche() {
    // deux manières différentes pour accéder à la valeur de x.
    std::cout << "x= " << x << std::endl;
    // this pointe l'objet courant
    std::cout << "this->x= " << this->x << std::endl;
}

int main()
{
    Test testobjet(12);
    testobjet.affiche();

    return 0;
}

Sortie:
x= 12
this->x= 12
```

© Mohamed N. Lokbani

v3.01

POO avec C++

```
#include <iostream>

class compte
{
    double actif;

public:
    static compte* dernier;
    compte (double m) {
        actif = m;
        dernier = this; // on mémorise l'adresse du dernier objet créé.
    }
    void afficher() {
        std::cout << "actif = " << actif << std::endl;
    };

    compte* compte::dernier = NULL;

    int main()
    {
        compte c1(100.20);
        c1.afficher();
        compte::dernier->afficher();

        compte c2(200.45);
        c1.dernier->afficher();
        c2.dernier->afficher();
        compte::dernier->afficher();

        return 0;
    }
}
```

Sortie:

```
actif = 100.2
actif = 100.2
actif = 200.45
actif = 200.45
actif = 200.45
```

7. Objets constants

```
const compte compte_gele(250.8);

impossible de modifier le contenu.
```

Seules les méthodes const peuvent manipuler des objets const.

```
#include <iostream>

class compte
{
    double actif;
public:
    compte (double m): actif(m) {}
    void afficher() {
        std::cout << "actif = " << actif << std::endl;
    }
};

int main() {
    int a=10;
    const int b=20;
    a+=10; // correct.
    // b+=20; // incorrect, b étant une constante.
    compte C(250.8); // correct.
    C.afficher(); // correct.

    const compte compte_gele(200.80);

    // compte_gele.afficher(); // Erreur compte_gele n'est accessible que par une méthode constante.

    return 0;
}
```

8. Méthodes constantes

Les méthodes constantes sont utilisées pour manipuler des objets constants.

Indiquent au compilateur que l'objet receveur ne sera pas modifié.

```
type_de_retour nom_de_la_fonction(arguments) const {}

// exemple
#include <iostream>

class compte
{
    double actif;
public:
    compte (double m): actif(m) {}
    void afficher() const {std::cout << "actif = " << actif << std::endl;}
};

int main() {
    compte C(250.8); // correct.
    C.afficher(); // correct, même si afficher est const alors que C ne l'est pas.
    const compte compte_gele(200.80);
    compte_gele.afficher(); // correct, afficher est cette fois-ci const.
    return 0;
}
```

Si la classe `compte` avait la fonction membre suivante:

```
class compte {
public:
    void deposer(double d) const {
        actif = d;
    }
    //etc.
private: // etc.
};
```

la définition de la fonction `deposer` provoque une erreur de compilation, car une fonction constante est supposée ne pas modifier l'objet receveur ; or dans cet exemple, on tente d'affecter à `actif` le contenu de `d`, d'où erreur. L'objet receveur est en lecture seulement.

Solution du problème posé en page Error! Bookmark not defined.

Ce que nous voudrions faire: afficher le nombre de comptes quand ils sont créés et quand ils sont détruits. Il y a plusieurs techniques pour le faire:

- modifier le destructeur (la plus simple):

```
~compte() {
    nbcomptes--;
}

cout << "nbcomptes: " << nbcomptes << endl;
}

sinon

~compte() {
    nbcomptes--;

    fonction_static(); // contient la même ligne que dans l'exemple précédent.
}
// pour les deux cas précédents:
int main() {
    // etc.
    compte C(260.8); // ← création de l'objet
    // etc
} ← destruction de l'objet
```

- créer l'objet dynamiquement (histoire de se compliquer la vie ...) :

```
int main() {  
    compte::fonction_static();  
  
    compte* C = new compte(260.8); // ← création de l'objet.  
    compte::fonction_static();  
    C->fonction_static();  
    C->affiche();  
  
    delete C; // ← destruction de l'objet C.  
    compte::fonction_static();  
    return 0;  
}
```

En sortie:

```
nbcomptes: 0  
nbcomptes: 1  
nbcomptes: 1  
nbcomptes: 1    actif: 260.8  
nbcomptes: 0
```