Département Mathématique et Informatique

- Cycle Ingénieur –

POO

Cours - 2 - résumé

Réaliser par :

KARROUM Houssam – BDCC 1

Contents

PROGRAMMATION ORIENTEE OBJET: PROPRIETES DES FONCTIONS MEMBRES

1. SURDEFINITION DES FONCTIONS MEMBRES:

En utilisant la propriété de surdéfinition des fonctions du C++, on peut définir plusieurs constructeurs, ou bien plusieurs fonctions membres, différentes, mais portant le même nom.

```
point::point() // constructeur 1
{ x=0;
 y=0;}
point::point(int abs) // constructeur 2
{x = abs;
 y = abs;}
point::point(int abs,int ord) // constructeur 3
{x = abs;
 y = ord;}
```

2. FONCTIONS MEMBRES EN LIGNE :

En C++, il est possible de définir des fonctions membres directement lors de leur déclaration dans la classe, ce qui est appelé une fonction "inline". Cette approche permet de créer une sorte de "macrofonction", où le code de la fonction est généré à chaque appel plutôt que d'appeler un sous-programme distinct. Cela peut conduire à une meilleure optimisation du code en évitant les appels de fonction et en réduisant la surcharge d'exécution associée.

```cpp
#include <iostream>

class Calculatrice {
public:
    // Déclaration d'une fonction membre inline pour ajouter deux nombres
    inline int addition(int a, int b) {
        return a + b;
    }
};

int main() {
    Calculatrice calc;

    // Appel de la fonction addition directement à partir de l'objet calc
    int resultat = calc.addition(5, 3);

    std::cout << "Résultat de l'addition : " << resultat << std::endl;

    return 0;
}
```

NOTA :

Pour comparer la taille des fichiers objet générés par ces exemples, nous devrions d'abord compiler les programmes en utilisant un compilateur C++ tel que g++. Ensuite, nous pouvons examiner les tailles des fichiers objets résultants pour voir s'il y a une différence significative entre les deux approches.

Cependant, en théorie, l'utilisation de fonctions inline peut réduire la taille du fichier objet, car le code de la fonction est directement intégré à chaque appel plutôt que d'être appelé comme une fonction séparée. Cela peut conduire à une optimisation du code en réduisant la surcharge associée aux appels de fonction.

3. OBJETS TRANSMIS EN ARGUMENT D'UNE FONCTION MEMBRE :

Une fonction membre peut re

cevoir un ou plusieurs arguments du type de sa classe.

```
#include <iostream>

class MaClasse {
private:
    int valeur;

public:
    // Constructeur
    MaClasse(int v) : valeur(v) {}

    // Fonction membre prenant un argument de type MaClasse
    void afficherValeur(MaClasse autreObjet) {
        std::cout << "La valeur de cet objet est : " << autreObjet.valeur <<
std::endl;
    }
};

int main() {
    MaClasse objet1(10);
    MaClasse objet2(20);

    objet1.afficherValeur(objet2); // Appel de la fonction membre avec objet2
comme argument

    return 0;
}
```

Dans cet exemple, la fonction membre afficherValeur de la classe MaClasse prend un argument de type MaClasse. Lorsque nous appelons cette fonction à partir d'un objet objet1 et passons un objet objet2 comme argument, elle peut accéder aux données de objet2 de la même manière qu'elle le ferait avec ses propres données. Cela montre que les fonctions membres peuvent recevoir des objets de leur propre classe comme arguments.

4. OBJETS RETOURNE PAR UNE FONCTION MEMBRE :

Dans cette partie, nous explorons les différents modes de retour d'une fonction membre qui retourne elle-même un objet.

Retour par valeur : Dans l'exemple donné, la fonction membre **symetrique** retourne un objet de type **point** par valeur. Elle crée un nouvel objet **res** qui est la symétrique de l'objet appelant et le retourne. L'exécution montre que cette approche fonctionne comme prévu, produisant la symétrique attendue.

```
#include <iostream>

class Point {
private:
    int x, y;

public:
    Point(int abs = 0, int ord = 0) : x(abs), y(ord) {}

    // Fonction membre retournant un objet par valeur
    Point symetrique() {
        Point res(-x, -y); // Création de l'objet symétrique
        return res; // Retour de l'objet
    }

    void affiche() {
        std::cout << "Le point est en " << x << " et " << y << "\n";
    }
};

int main() {
    Point a(1, 6);
    Point b = a.symetrique(); // Appel de la fonction membre retournant un objet
par valeur
    b.affiche(); // Affichage du point symétrique

    return 0;
}
```

Retour par adresse : Dans cet exemple, la fonction membre **symetrique** retourne un pointeur vers un objet **point** alloué dynamiquement. Cette fois, la fonction crée un nouvel objet sur le tas à l'aide de **new**. L'objet est ensuite retourné via un pointeur. L'exécution montre le même résultat que le précédent, mais l'objet est créé sur le tas et doit être libéré manuellement pour éviter les fuites de mémoire.

```cpp
#include <iostream>

class Point {
private:
    int x, y;

public:
    Point(int abs = 0, int ord = 0) : x(abs), y(ord) {}

    // Fonction membre retournant un objet par adresse
    Point* symetrique() {
        Point* res = new Point(-x, -y); // Création de l'objet symétrique sur le tas
        return res; // Retour de l'adresse de l'objet
    }

    void affiche() {
        std::cout << "Le point est en " << x << " et " << y << "\n";
    }
};

int main() {
    Point a(1, 6);
    Point* b = a.symetrique(); // Appel de la fonction membre retournant un objet par adresse
    b->affiche(); // Affichage du point symétrique

    delete b; // Libération de la mémoire allouée dynamiquement

    return 0;
}
```

Retour par référence : Dans cet exemple, la fonction membre **symetrique** retourne une référence à un objet **point**. L'objet **res** est déclaré statiquement à l'intérieur de la fonction, ce qui signifie qu'il conserve sa valeur entre les appels de la fonction. L'exécution produit encore une fois le même résultat, mais cette fois l'objet retourné est toujours le même et n'est pas créé dynamiquement.

```
#include <iostream>

class Point {
private:
    int x, y;

public:
    Point(int abs = 0, int ord = 0) : x(abs), y(ord) {}

    // Fonction membre retournant un objet par référence
    Point& symetrique() {
        static Point res; // Objet statique pour conserver sa valeur entre les
appels
        res.x = -x; // Calcul des coordonnées symétriques
        res.y = -y;
        return res; // Retour de l'objet par référence
    }

    void affiche() {
        std::cout << "Le point est en " << x << " et " << y << "\n";
    }
};

int main() {
    Point a(1, 6);
    Point& b = a.symetrique(); // Appel de la fonction membre retournant un
objet par référence
    b.affiche(); // Affichage du point symétrique

    return 0;
}
```

5. FONCTIONS MEMBRES STATIQUES :

en C++, il est possible de déclarer et d'utiliser des fonctions membres statiques dans une classe. Les fonctions membres statiques sont des fonctions associées à la classe elle-même plutôt qu'à des instances individuelles de cette classe. Elles peuvent être appelées sans avoir besoin d'instancier un objet de la classe.

Voici un exemple simple pour illustrer l'utilisation de fonctions membres statiques :

```
#include <iostream>

class MaClasse {
public:
    // Fonction membre statique
    static void afficherMessage() {
        std::cout << "Bonjour depuis la fonction membre statique !" <<
std::endl;
    }
};

int main() {
    // Appel de la fonction membre statique sans instancier un objet
    MaClasse::afficherMessage();

    return 0;
}
```

Dans cet exemple, la fonction membre **afficherMessage** est déclarée comme statique à l'intérieur de la classe **MaClasse**. Elle peut être appelée directement en utilisant le nom de la classe suivi de l'opérateur de résolution de portée **::** et le nom de la fonction membre statique. Il n'est pas nécessaire de créer une instance de la

classe pour appeler cette fonction.

6. LE MOT CLE « THIS » :

Le mot clé "this" en C++ fait référence à l'adresse mémoire de l'objet qui invoque la fonction membre. Il est utilisable uniquement à l'intérieur des fonctions membres d'une classe.

À chaque appel d'une fonction membre, le compilateur passe implicitement un pointeur sur les données de l'objet en tant que paramètre. Ce pointeur sur l'objet est accessible à l'intérieur de la fonction membre et est désigné par le mot clé "this". Il est important de noter que "this" est un pointeur constant, ce qui signifie qu'on ne peut pas le modifier.

Voici un exemple illustrant l'utilisation de "this" en C++ :

```cpp
#include <iostream>

class MaClasse {
private:
    int x;

public:
    MaClasse(int val) : x(val) {}

    void afficherAdresse() {
        std::cout << "L'adresse de l'objet est : " << this << std::endl;
    }

    void afficherValeur() {
        std::cout << "La valeur de x pour cet objet est : " << this->x << std::endl;
    }
};
```

```
int main() {
    MaClasse objet(5);

    objet.afficherAdresse(); // Affiche l'adresse de l'objet
    objet.afficherValeur();  // Affiche la valeur de x pour cet objet

    return 0;}
```

Dans cet exemple, "this" est utilisé pour accéder à l'adresse de l'objet et à ses données membres à l'intérieur des fonctions membres de la classe.

FIN