ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA
المدرسة العليا لأساتذة التعليم التقني المحمدية

Département Mathématique et Informatique

- Cycle Ingénieur -

Résumé de Chapitre 4 - POO :

Réaliser par :

KARROUM HOUSSAM /BDCC1

Av. Hassan II B.P. 159,
Mohammedia - Maroc
Tél. : +212 5 23 32 22 20
Fax : +212 5 23 32 25 46
www.enset.univh2c.ma

شارع الحسن الثاني ص.ب 159
المحمدية، المغرب
الهاتف : 212523322220+
الفاكس : 212523322546+
www.enset.univh2c.ma

Table de matiére

CONSTRUCTION ET DESTRUCTION DES OBJETS AUTOMATIQUES

Une variable locale automatique est une variable qui est déclarée à l'intérieur d'un bloc de code, tel qu'une fonction ou un bloc { }, sans utiliser le mot-clé static. Ces variables sont automatiquement créées et détruites en fonction de la portée du bloc dans lequel elles sont déclarées.

Chronologie de la création et de la destruction des objets :

Lorsqu'un objet est créé, son constructeur est appelé pour l'initialiser. Une fois construit, l'objet peut être utilisé. À la sortie de la portée de l'objet, son destructeur est appelé pour libérer les ressources qu'il utilisait. Ensuite, l'objet est détruit et l'espace mémoire est libéré.

```cpp
#include <iostream>

class point {
    int x, y;
public:
    point(int, int);
    ~point();
};

point::point(int abs, int ord) {
    x = abs; y = ord;
    std::cout << "Construction du point " << x << " " << y << "\n";
}

point::~point() {
    std::cout << "Destruction du point " << x << " " << y << "\n";
}

void test() {
    std::cout << "Debut de test()\n";
    point u(3, 7);
    std::cout << "Fin de test()\n";
}

int main() {
    std::cout << "Debut de main()\n";
    point a(1, 4);
```

```
    test();
    point b(5, 10);
    for (int i = 0; i < 3; i++) {
        std::cout << "Boucle tour numéro " << i << "\n";
        point(7 + i, 12 + i);
    }
    std::cout << "Fin de main()\n";
    return 0;
}
```

Resultat :

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4> cd "c:\Users\Adm
if ($?) { .\cours }
Debut de main()
Construction du point 1 4
Debut de test()
Construction du point 3 7
Fin de test()
Destruction du point 3 7
Construction du point 5 10
Boucle tour num├®ro 0
Construction du point 7 12
Destruction du point 7 12
Boucle tour num├®ro 1
Construction du point 8 13
Destruction du point 8 13
Boucle tour num├®ro 2
Construction du point 9 14
Destruction du point 9 14
Fin de main()
Destruction du point 5 10
Destruction du point 1 4
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

CONSTRUCTION ET DESTRUCTION DES OBJETS STATIQUES

- Les objets statiques sont créés par une déclaration située :

- En dehors de toute fonction.
- Dans une fonction, mais avec le qualificatif `static`.

- Les objets statiques sont créés avant le début de l'exécution de la fonction `main()` et sont détruits après la fin de son exécution.

En résumé, les objets statiques sont des variables qui existent pendant toute la durée d'exécution du programme. Ils sont initialisés avant le début de l'exécution de la fonction `main()` et détruits après la fin de son exécution.

```cpp
#include <iostream>

class point {
    int x, y;
public:
    point(int, int);
    ~point();
};

point::point(int abs, int ord) {
    x = abs;
    y = ord;
    std::cout << "Construction du point " << x << " " << y << "\n";
}

point::~point() {
    std::cout << "Destruction du point " << x << " " << y << "\n";
}

void test() {
    std::cout << "Debut de test()\n";
    static point u(3, 7);
    std::cout << "Fin de test()\n";
}

int main() {
    std::cout << "Debut de main()\n";
    point a(1, 4);
    test();
    point b(5, 10);
    std::cout << "Fin de main()\n";
    return 0;
}
```

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4> cd "c:\Us
if ($?) { .\cours }
Debut de main()
Construction du point 1 4
Debut de test()
Construction du point 3 7
Fin de test()
Construction du point 5 10
Fin de main()
Destruction du point 5 10
Destruction du point 1 4
Destruction du point 3 7
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

CONSTRUCTION ET DESTRUCTION DES OBJETS GLOBAUX

```cpp
#include <iostream>

#include <iostream>

class point {
    int x, y;
public:
    point(int, int);
    ~point();
};

point::point(int abs, int ord) {
    x = abs;
    y = ord;
    std::cout << "Construction du point " << x << " " << y << "\n";
}

point::~point() {
    std::cout << "Destruction du point " << x << " " << y << "\n";
}

point a(1, 4); // variable globale
```

```
int main() {
    std::cout << "Debut de main()\n";
    point b(5, 10);
    std::cout << "Fin de main()\n";
    return 0;
}
```

Resultat :

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4> cd "c:
if ($?) { .\cours }
Construction du point 1 4
Debut de main()
Construction du point 5 10
Fin de main()
Destruction du point 5 10
Destruction du point 1 4
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

CONSTRUCTION ET DESTRUCTION DES OBJETS TEMPORAIRES

Si a est un objet de type point, on peut écrire l'affectation : a = point(1,2) ; dont laquelle l'évaluation de l'expression point(1,2) conduit à :

- La déclaration d'un objet temporaire de type point,
- L'appel du constructeur point, pour cet objet temporaire, avec transmission des arguments spécifiés,
- La recopie de cet objet temporaire dans a.

```
#include <iostream>

class point {
    int x, y;
```

```
public:
    point(int, int);
    ~point();
};

point::point(int abs, int ord) {
    x = abs;
    y = ord;
    std::cout << "Construction du point " << x << " " << y << " à l'adresse : "
<< this << "\n";
}
```

```
point::~point() {
    std::cout << "Destruction du point " << x << " " << y << " à l'adresse : " <<
this << "\n";
}

int main() {
    std::cout << "Debut de main()\n";
    point a(0, 0);
    a = point(1, 2);
    a = point(3, 5);
    std::cout << "Fin de main()\n";
    return 0;
}
```

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4> cd "c:\User
if ($?) { .\cours }
Debut de main()
Construction du point 0 0 ├á lÔÇÖadresse : 0x61fef8
Construction du point 1 2 ├á lÔÇÖadresse : 0x61ff00
Destruction du point 1 2 ├á lÔÇÖadresse : 0x61ff00
Construction du point 3 5 ├á lÔÇÖadresse : 0x61ff08
Destruction du point 3 5 ├á lÔÇÖadresse : 0x61ff08
Fin de main()
Destruction du point 3 5 ├á lÔÇÖadresse : 0x61fef8
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

CONSTRUCTION ET DESTRUCTION DES OBJETS DYNAMIQUES

Voici un résumé des concepts concernant la création et l'utilisation d'objets dynamiques en C++ :

- La déclaration d'un pointeur vers un objet se fait comme suit :

```
  Point *adr;
```

- Pour créer dynamiquement un objet et affecter son adresse au pointeur, on utilise l'opérateur `new` :

```
   adr = new Point;
```

- L'accès aux fonctions membres de l'objet pointé par le pointeur se fait en utilisant l'opérateur `->` :

```
  adr->initialise(1, 3);
  adr->affiche();
```

- Pour supprimer l'objet dynamiquement alloué, on utilise l'opérateur `delete` :

```
  delete adr;
```

- Si la classe possède un constructeur prenant des arguments, on peut les spécifier lors de l'allocation dynamique :

```
adr = new Point(2, 5);
```

En résumé, l'allocation dynamique permet de créer des objets à la volée pendant l'exécution du programme, et leur libération se fait explicitement en utilisant l'opérateur `delete` pour éviter les fuites de mémoire. L'accès aux membres de l'objet se fait à travers le pointeur en utilisant l'opérateur `->`.

```
#include <iostream>

class point {
    int x, y;
public:
    point(int, int);
    ~point();
};

point::point(int abs, int ord) {
    x = abs;
```

```
    y = ord;
    std::cout << "Construction du point " << x << " " << y << "\n";
}

point::~point() {
    std::cout << "Destruction du point " << x << " " << y << "\n";
}

void fct(point *adp) {
    std::cout << "Debut de la fonction\n";
    delete adp; // libération de la place
    std::cout << "Fin de la fonction\n";
}

int main() {
    std::cout << "Debut de main()\n";
    void fct(point *);
    point *adr;
    adr = new point(3, 7); // réservation de la place en mémoire
    fct(adr);
    delete adr; // libération de la place
    std::cout << "Fin de main()\n";
    return 0;
}
```

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4> cd "c:\Us
-o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
Debut de main()
Construction du point 3 7
Debut de la fonction
Destruction du point 3 7
Fin de la fonction
Destruction du point 16455632 16449728
Fin de main()
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

Apres commenter « **delete adr** » de la fonction **fct**

```
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
if ($?) { .\cours }
Debut de main()
Construction du point 3 7
Debut de la fonction
Destruction du point 3 7
Fin de la fonction
Fin de main()
PS C:\Users\Admin\Desktop\cycle\1 annee\s2\c++\cours\td & resume 4>
```

en omettant l'instruction delete adp;, le programme fonctionnera toujours correctement en termes de syntaxe, mais il pourrait entraîner une fuite de mémoire si cela est répété à plusieurs reprises dans un programme plus complexe. Il est donc important de toujours libérer la mémoire allouée dynamiquement avec delete une fois que vous avez terminé d'utiliser l'objet.

INITIALISATION DES OBJETS

En C++, lorsque vous déclarez une variable et l'initialisez avec une autre variable du même type, deux situations peuvent se présenter :

1. **Initialisation directe avec un objet existant :**

```
   Point a(5, 6); // Crée un nouvel objet 'a' en utilisant le constructeur avec
arguments int (5, 6)
```

Lors de cette initialisation, le constructeur avec arguments de la classe `Point` est appelé pour créer l'objet `a` avec les valeurs spécifiées.

2. **Initialisation par copie avec un objet existant :

```
Point b = a; // Initialise 'b' avec 'a' (copie de 'a')
```

Lors de cette initialisation par copie, le constructeur de copie par défaut de la classe `Point` est appelé. Si vous ne définissez pas votre propre constructeur de copie, le compilateur fournira automatiquement un constructeur de copie par défaut qui effectue une copie superficielle des données membres de l'objet.

En résumé, lors de l'initialisation de `b` avec `a` en utilisant l'initialisation par copie, le constructeur de copie par défaut de la classe `Point` est appelé, sauf si vous avez défini explicitement votre propre constructeur de copie.

Exemple 1 :

```cpp
#include <iostream>

class liste {
    int taille;
    float *adr;
public:
    liste(int);
    ~liste();
};

liste::liste(int t) {
    taille = t;
    adr = new float[taille]; // Allocation dynamique de la mémoire pour adr
    std::cout << "Construction";
    std::cout << " Adresse de l'objet:" << this;
    std::cout << " Adresse de liste:" << adr << "\n";
}

liste::~liste() {
    std::cout << "Destruction Adresse de l'objet:" << this;
    std::cout << " Adresse de liste:" << adr << "\n";
    delete[] adr; // Libération de la mémoire allouée dynamiquement
}

int main() {
    std::cout << "Debut de main()\n";
    liste a(3);
    liste b = a; // Initialisation par copie
    std::cout << "Fin de main()\n";
    return 0;
}
```

Dans ce code, la classe `liste` alloue dynamiquement de la mémoire pour le pointeur `adr` dans son constructeur. Lorsque l'objet `b` est initialisé par copie avec l'objet `a`, le constructeur de copie par défaut est appelé, et les deux objets `a` et `b` finissent par avoir le même pointeur `adr`. Cela conduit à des problèmes lors de la libération de la mémoire dans les destructeurs, car la mémoire est libérée deux fois (une fois pour `a` et une fois pour `b`).

Exemple 2 :

Dans ce programme, une classe `liste` est définie avec un membre privé de type pointeur `adr`. Le constructeur de la classe alloue dynamiquement de la mémoire pour ce pointeur, et le destructeur libère cette mémoire. Lorsque deux objets de type `liste`, `a` et `b`, sont créés, l'initialisation par copie de `b` avec `a` entraîne une double libération de la mémoire, car les deux objets finissent par avoir le même pointeur.

```
#include <iostream>

class liste {
    int taille;
    float *adr;
public:
    liste(int);
    liste(const liste&); // Déclaration du constructeur de copie
    ~liste();
};

liste::liste(int t) {
    taille = t;
    adr = new float[taille]; // Allocation dynamique de la mémoire pour adr
    std::cout << "Construction ";
    std::cout << "Adresse de l'objet : " << this;
    std::cout << " Adresse de liste : " << adr << "\n";
}

// Définition du constructeur de copie
liste::liste(const liste& autre) {
    taille = autre.taille;
    adr = new float[taille]; // Allocation dynamique de la mémoire pour adr
    // Copie des données
    for (int i = 0; i < taille; ++i) {
        adr[i] = autre.adr[i];
    }
    std::cout << "Construction par copie ";
    std::cout << "Adresse de l'objet : " << this;
    std::cout << " Adresse de liste : " << adr << "\n";
}

liste::~liste() {
    std::cout << "Destruction Adresse de l'objet : " << this;
    std::cout << " Adresse de liste : " << adr << "\n";
```

```
    delete[] adr; // Libération de la mémoire allouée dynamiquement
}

int main() {
    std::cout << "Debut de main()\n";
    liste a(3); // Création de l'objet a
    liste b = a; // Initialisation de b par copie avec a
    std::cout << "Fin de main()\n";
    return 0;
}
```

ROLE DU CONSTRUCTEUR LORSQU'UNE FONCTION RETOURNE UN OBJET

Le rôle du constructeur lorsqu'une fonction retourne un objet est de créer cet objet et de l'initialiser correctement avant qu'il ne soit retourné à l'appelant.

Exemple 1 :

```
#include <iostream>

class point {
    int x, y;

public:
    point(int = 0, int = 0); // Constructeur par défaut avec des valeurs par
défaut
    point(const point&); // Constructeur de copie
    point symetrique() ; // Méthode pour calculer le point symétrique
    void affiche() ; // Méthode pour afficher les coordonnées du point
    ~point(); // Destructeur
};

point::point(int abs, int ord) : x(abs), y(ord) {
    std::cout << "Construction du point " << x << " " << y;
    std::cout << " d'adresse " << this << "\n";
}
```

```
point::point(const point& pt) : x(pt.x), y(pt.y) {
    std::cout << "Construction par recopie du point " << x << " " << y;
    std::cout << " d'adresse " << this << "\n";
}

point point::symetrique()  {
    point res(-x, -y); // Calcul du point symétrique par rapport à l'origine
    std::cout << "Construction du point symétrique " << res.x << " " << res.y;
    std::cout << " d'adresse " << &res << "\n";
    return res;
}

void point::affiche()  {
    std::cout << "x = " << x << ", y = " << y << "\n";
}

point::~point() {
    std::cout << "Destruction du point " << x << " " << y;
    std::cout << " d'adresse " << this << "\n";
}

int main() {
    std::cout << "Debut de main()\n";
    point a(1, 4), b;
    std::cout << "Avant appel à symetrique\n";
    b = a.symetrique();
    b.affiche();
    std::cout << "Après appel à symetrique et fin de main()\n";
    return 0;
}
```

Dans cet exemple, nous avons une classe `point` qui représente des points dans un espace bidimensionnel. La fonction membre `symetrique()` retourne un nouvel objet `point` qui est le point symétrique par rapport à l'origine de l'objet appelant.

Le programme manipule cette fonction et étudie à quel moment les constructeurs et destructeurs sont exécutés. Voici un résumé des étapes clés :

1. Lorsque la fonction `symetrique()` est appelée sur un objet `point`, un nouvel objet temporaire est créé pour stocker le résultat.
2. Le constructeur par défaut de la classe `point` est appelé pour initialiser cet objet temporaire avec les coordonnées du point symétrique.

3. Ensuite, le destructeur de cet objet temporaire est appelé une fois qu'il n'est plus nécessaire.

Il est mentionné qu'un constructeur par recopie est également utilisé lorsqu'un objet est assigné à un autre. Cela garantit que la copie est correctement effectuée, y compris pour les parties dynamiquement allouées de l'objet.

Il est important de prévoir un constructeur par recopie lorsque l'objet contient des parties dynamiques, comme des pointeurs alloués dynamiquement, pour éviter les problèmes de gestion de la mémoire et garantir le bon fonctionnement du programme.

Exemple 2 :

Nous reprenons la classe liste précédemment étudiée et ajoutons une fonction membre oppose(), qui renvoie une nouvelle liste contenant les coordonnées opposées de la liste actuelle.

```
#include <iostream>

class liste {
    int taille;
    float *adr;

public:
    liste(int);
    liste(const liste &);
    ~liste();
    void saisie();
    void affiche();
    liste oppose();
};

liste::liste(int t) {
    taille = t;
    adr = new float[taille];
    std::cout << "Construction de la liste de taille " << taille;
    std::cout << " à l'adresse : " << this << " avec la liste : " << adr << "\n";
}

liste::liste(const liste &v) {
    taille = v.taille;
    adr = new float[taille];
    for (int i = 0; i < taille; i++)
```

```
        adr[i] = v.adr[i];
    std::cout << "Construction par copie de la liste à l'adresse : " << this;
    std::cout << " avec la liste : " << adr << "\n";
}

liste::~liste() {
    std::cout << "Destruction de la liste à l'adresse : " << this;
    std::cout << " avec la liste : " << adr << "\n";
    delete[] adr;
}

void liste::saisie() {
    for (int i = 0; i < taille; i++) {
        std::cout << "Entrer un nombre pour l'élément " << i << " : ";
        std::cin >> adr[i];
    }
}

void liste::affiche() {
    std::cout << "Contenu de la liste à l'adresse : " << this << " avec la liste
: " << adr << " : ";
    for (int i = 0; i < taille; i++)
        std::cout << adr[i] << " ";
    std::cout << "\n";
}

liste liste::oppose() {
    liste res(taille); // Création d'une nouvelle liste avec la même taille
    for (int i = 0; i < taille; i++)
        res.adr[i] = -adr[i]; // Calcul des coordonnées opposées
    std::cout << "Liste opposée générée à l'adresse : " << &res << " avec la
liste : " << res.adr << " : ";
    for (int i = 0; i < taille; i++)
        std::cout << res.adr[i] << " ";
    std::cout << "\n";
    return res; // Retour de la nouvelle liste
}

int main() {
    std::cout << "Debut de main()\n";

    liste a(3), b(3);

    std::cout << "Saisie de la liste a :\n";
    a.saisie();
```

```
    std::cout << "Affichage de la liste a :\n";
    a.affiche();

    std::cout << "Liste opposée de la liste a :\n";
    b = a.oppose();
    b.affiche();

    std::cout << "Fin de main()\n";
    return 0;
}
```

La classe `liste` est conçue pour stocker un tableau de nombres flottants de taille variable. La fonction membre `oppose()` a été ajoutée pour retourner une nouvelle liste contenant les coordonnées opposées de la liste actuelle. Cependant, l'implémentation de cette fonction comporte des erreurs, notamment dans la création de la liste opposée.

LES TABLEAUX D'OBJETS

Les tableaux d'objets sont manipulés de manière similaire aux tableaux traditionnels en langage C. En reprenant la classe `point` étudiée précédemment, nous pouvons déclarer un tableau de 50 objets de type `point` en utilisant la syntaxe suivante :

```
point courbe[50]; // Déclaration d'un tableau de 50 points
```

Ainsi, si `i` est un entier, l'expression `courbe[i]` fait référence à un objet de type `point`. Ensuite, l'appel de la fonction membre `affiche()` pour le point `courbe[i]` peut être réalisé comme suit :

```
courbe[i].affiche();
```

Le programme suivant illustre l'affichage de tous les points de la courbe :

```
for (int i = 0; i < 50; i++)
    courbe[i].affiche();
```

Il est impératif que la classe `point` possède un constructeur sans argument (ou avec des arguments par défaut). Ce constructeur est appelé pour chaque élément du tableau. Une notation pratique pour le constructeur est la suivante :

```
class point {
    int x, y;
public:
    point(int abs = 0, int ord = 0) {
        x = abs;
        y = ord;
    }
};
```

L'exemple suivant illustre la création d'un tableau de 5 points avec des valeurs initiales :

```
void main() {
    point courbe[5] = { {7, 0}, {4, 0}, {2, 0} };
}
```

Cette initialisation entraîne les résultats suivants :

```
x  y
courbe[0]  7   0
courbe[1]  4   0
courbe[2]  2   0
courbe[3]  0   0
courbe[4]  0   0
```

De manière similaire, un tableau dynamique peut être créé comme suit :

```
point *adcourbe = new point[50];
```

Pour libérer la mémoire allouée pour ce tableau dynamique, l'instruction suivante est utilisée :

```
delete[] adcourbe;
```

Cela entraînera l'exécution du destructeur pour chaque élément du tableau.

OBJETS MEMBRES OU OBJETS D'OBJETS

Dans la programmation orientée objet, il est possible pour une classe d'avoir un membre qui est lui-même une instance d'une autre classe. Par exemple, considérons la classe `point` :

```
class point {
    int x, y;
public:
    point(int, int);
    void affiche();
};
```

Ensuite, nous pouvons définir une autre classe appelée `poincol` de la manière suivante :

```
class poincol {
    point p;
    int couleur;
public:
    void affcoul();
};
```

Lorsque nous déclarons un nouvel objet `a` de type `poincol`, il possède un membre `p` de type `point`. L'accès aux méthodes de la classe `point` se fait à travers cet objet membre. Par exemple, pour appeler la méthode `affiche()` de la classe `point`, nous utilisons `a.p.affiche()`.

Pour mettre en œuvre les constructeurs et destructeurs, supposons que la classe `point` ait un constructeur :

```
class point {
    int x, y;
public:
    point(int, int);
};
```

Pour la classe `poincol`, nous devons :

- Définir un constructeur prenant en compte les arguments nécessaires pour initialiser les membres, y compris le membre de type `point`.
- Spécifier les arguments qui seront transmis au constructeur de `point`. Ces arguments doivent être sélectionnés parmi ceux fournis au constructeur de `poincol`.

La classe `poincol` et son constructeur peuvent être définis de la manière suivante :

```
class poincol {
    point p;
    int couleur;
public:
    poincol(int, int, int);
};

poincol::poincol(int abs, int ord, int coul) : p(abs, ord) {
    couleur = coul;
}
```

Dans cet exemple, l'en-tête du constructeur de `poincol` spécifie les arguments à transmettre au constructeur de `point`. L'initialisation de `p` avec ces arguments est réalisée dans la liste d'initialisation du constructeur de `poincol`.

Les constructeurs sont appelés dans l'ordre suivant : `point`, puis `poincol`. Si des destructeurs existent, ils sont appelés dans l'ordre inverse.

Voici une version reformulée du code avec quelques ajustements :

```
#include <iostream>

class point {
    int x, y;
public:
    point(int abs = 0, int ord = 0) : x(abs), y(ord) {
        std::cout << "Constructeur point : " << x << " " << y << "\n";
    }
};

class poincol {
    point p;
    int couleur;
```

```
public:
    poincol(int, int, int);
};

poincol::poincol(int abs, int ord, int coul) : p(abs, ord), couleur(coul) {
    std::cout << "Constructeur poincol : " << couleur << "\n";
}

int main() {
    poincol a(1, 3, 9);
    return 0;
}
```

Conclusion

En conclusion, dans la programmation orientée objet, il est possible d'avoir des classes qui contiennent d'autres classes en tant que membres. Ces membres peuvent être des objets d'autres classes, ce qui permet une organisation modulaire et une meilleure gestion des données. Lors de la création de ces objets composés, les constructeurs sont appelés dans un ordre spécifique, généralement dans l'ordre de déclaration des membres dans la classe. De même, les destructeurs sont appelés dans l'ordre inverse lors de la destruction de ces objets. Cette approche permet de créer des structures complexes et bien encapsulées, favorisant la réutilisation du code et la maintenabilité du système.