

Chapitre 5

Classes et Objets

1. Structures

Une structure permet de regrouper des objets de types hétérogènes.

```
#include <iostream>

using namespace std;

struct compte {
    char nom[20];
    double actif;
    int limite; // limite de crédit
};

int main() {
    compte CA; // déclaration de la variable CA du type compte, CA n'est pas initialisé.

    // Initialisation du compte CA
    strcpy(CA.nom, "smith");
    CA.actif = 123.75;
    CA.limite = 1000;

    cout << "compte CA, nom: " << CA.nom << endl;
    cout << "compte CA, actif: " << CA.actif << endl;
    cout << "compte CA, la limite de credit: " << CA.limite << endl;

    return 0;
}
```

En C++, nous pouvons inclure des membres fonctions à l'intérieur d'une structure (ce qui n'est pas le cas en C). Ces fonctions ont le rôle de manipuler les membres de données de la structure.

```
#include <iostream>
using namespace std;

struct compte {
    char nom[20];
    double actif;
    int limite; // limite de crédit
    void initialise();
    double depot(double); } Fonctions membres
};

// on accède aux fonctions membres par l'intermédiaire de l'opérateur unaire de résolution de portée c.-à-d. ::

void compte::initialise() {
    nom[0] = '\0'; // un membre de données est accessible directement par
    actif = 0.0; // les fonctions membres
    limite = 2000;
}

double compte::depot(double argent) {
    actif += argent;
    return actif;
}
```

```
int main() {

    compte CA; // déclaration de la variable CA du type compte CA n'est pas initialisée.

    // Initialisation du compte CA

    CA.initialise(); // accès aux membres de données via la fonction membre: initialise()

    strcpy(CA.nom, "smith"); // accès direct
    CA.actif = 123.75;
    CA.limite = 1000;

    double actuel = CA.depot(250.88);
    cout << "compte CA, nom: " << CA.nom << endl;
    cout << "compte CA, actif: " << CA.actif << endl;
    cout << "compte CA, actif via la fonction dépôt: " << actuel << endl;
    cout << "compte CA, la limite de crédit: " << CA.limite << endl;

    return 0;
}
// -----

En sortie

compte CA, nom: smith
compte CA, actif: 374.63
compte CA, actif via la fonction dépôt: 374.63
compte CA, la limite de crédit: 1000
```

Écrire `depot (259 . 8)` va générer une erreur. La fonction membre `depot` n'est accessible qu'à travers un objet du type `compte` (dans l'exemple précédent `CA`).

On peut passer une structure comme argument:

```
int fonction( compte D ) {...}
```

Comme on peut recevoir une structure comme valeur de retour:

```
compte fonction( int , double ) {...}
```

On peut effectuer aussi une copie membre à membre de deux structures:

```
compte CA, CB ;  
CA = CB ;
```

Pour l'exemple précédent, écrire de deux manières différentes la fonction, `affiche (...)`, qui aura pour tâche d'afficher les membres données de la structure `compte`.

2. Classes

2.1. Définitions

(voir le chapitre 1 pour un rappel sur les notions de classes et objets)

```
struct compte {  
    char nom[20] ;  
    // etc.  
};  
  
class compte {  
    char nom[20] ;  
    // etc.  
};
```

Par défaut, tous les membres d'une structure sont **PUBLICS**, c.-à-d. accessibles du monde extérieur. Pour les classes, par défaut, tous les membres sont **PRIVÉS**.

Si on veut distinguer les membres publics des membres privés, il faut faire ce qui suit:

```
class compte {  
  
private: // mettre private ici est facultatif, puisque par défaut les membres sont privés.  
    char nom[20] ;  
    // etc.  
  
public: // obligatoire, si on veut déclarer des membres public.  
    void initialise() ;  
    // etc.  
};
```

Écrire `public` avant `private` ou l'inverse n'est pas très important. Ce qui l'est, c'est de réaliser la différence entre les membres privés et les membres publics.

```
class compte {  
  
    private: // facultatif  
        char nom[20];  
        // etc.  
  
    public:  
        void initialise();  
        // etc.  
};  
  
class compte {  
  
    public:  
        void initialise();  
        // etc.  
  
    private: //ici obligatoire, pour distinguer ce qui  
        //est public et ce qui est privé.  
        char nom[20];  
        double actif;  
        // etc.  
};
```

2.2. Principe d'encapsulation

Pour accéder aux membres de données `private`, il faut obligatoirement passer par les fonctions membres (déclarées `public`).

```
CA.initialise(); // OK, initialise étant public  
  
CA.actif = 250.87; // Erreur, actif est private, n'est donc accessible que par une fonction membre.
```

Parmi les rôles joués par les fonctions membres, celui de réaliser une vérification exhaustive sur les données en entrée.

```
double compte::depot(double argent) {  
    if (argent < 0) {  
        cerr << "dépôt d'argent négatif, opération illégale !\n";  
        exit (1); // on sort complètement du programme.  
    } else actif += argent;  
    return actif;  
}
```

Une fonction membre peut être déclarée aussi comme `private` si elle ne doit pas être rendue accessible de l'extérieur (au même titre que les données `private`), voir l'exemple complet sur les comptes, un peu plus loin dans le cours.

Dans tous les cas, une fonction membre (`public` ou `private`) a accès aux données `private` d'une classe.

On peut affecter les objets même si les membres sont privés:

```
compte CA,CB;  
CB = CA; // OK  
  
CB.actif = CA.actif; // Erreur, actif étant private
```

3. Constructeurs

3.1. Définitions

- méthode appelée à la création de l'objet.
- sert à initialiser l'objet

Comme il n'est pas possible d'initialiser les membres de données (non static, voir le cas de static plus loin dans le cours), dans l'exemple `compte`, si la fonction `initialise()` n'a pas été définie, en C++, le compilateur initialise les membres données de la classe avec des valeurs quelconques.

- ne retourne rien (ne pas mettre void)
- public
- porte le même nom que la classe

syntaxe: `nom_de_la_classe(arguments);`

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 5 : Classes et Objets

```
class compte {
private:
    // etc.
public:
    compte(char*, double, int); // ← constructeur
    // etc.
};

compte::compte(char* chaine, double argent, int max){
    // il faudra faire les vérifications pour s'assurer que la chaîne passée ne dépasse pas les 20 caractères ;
    // que les variables argent et max sont toutes les deux • 0, voir l'exemple complet.
    strcpy(nom, chaine);
    actif = argent;
    limite = max;
}

// etc.

int main() {
    compte CA("Fred", 100.50, 1000); // déclaration et initialisation en une seule opération
    // etc.
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

3.2. Constructeur par défaut

Chaque classe a un constructeur par défaut (il est caché, mais il existe),

```
compte CB; // appel le constructeur par défaut défini par le compilateur comme étant:
```

```
compte::compte() { // sans arguments.
```

```
    // initialisation quelconque, choisie par le compilateur
```

```
}
```

Il est appelé si aucun constructeur n'a été défini explicitement dans la classe,

Si, dans une classe, il a été défini un constructeur (comme vu précédemment dans le cas de la classe `compte`), ce dernier masque le constructeur par défaut. Ainsi, le constructeur par défaut cesse d'exister et devient donc inaccessible ; de ce fait, l'instruction suivante :

```
compte CB; // génère une erreur, car il n'existe aucun constructeur
```

3.3. Surdéfinition d'un constructeur et initialisation par défaut de ses arguments

Un constructeur a le même comportement qu'une fonction quelconque (aux exceptions citées précédemment, le fait qu'il ne retourne rien sans pour autant le déclarer par void, etc.)

Initialisation par défaut ...

```
compte(char* chaine, double argent = 390, int max=1000);
```

```
compte CA("Bob", 289.90); // 3e argument est par défaut
```

```
compte CA("Bob"); // 2e et 3e argument est par défaut
```

Surdéfinition

```
compte(char* chaine, double argent = 390, int max=1000);
```

```
compte CA("Bob", 289.90); // 3e argument est par défaut
```

```
compte(double argent, int max, char* chaine);
```

```
compte CA(290.98, 10000, "Santana");
```

3.4. La surdéfinition et le constructeur par défaut

Si au moins un constructeur a été défini dans la classe `compte` (ou autre), pour autoriser cette déclaration:

```
compte CB;
```

Il faut:

```
class compte {
private:
    // etc.
public:
    compte(char*, double, int); // ← constructeur
    compte(); // ← un autre constructeur
    // etc.
};

compte::compte(char* chaine, double argent, int max){// à faire}
compte::compte(){// à faire}

// etc.
int main() {
    compte CA("Fred",100.50,1000); // déclaration et initialisation en une seule opération
    compte CB; // cette fois-ci c'est ok! Appel du constructeur compte()
    // etc.
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 5 : Classes et Objets

4. Destructeur

Chaque classe a un et un seul destructeur (donc ne peut pas être surchargé).

Il est appelé automatiquement quand un objet disparaît.

```
{ // début d'un bloc
    // déclarations à l'intérieur d'un bloc
    int i;
    compte CA;
} // fin du bloc => mort de CA et i
```

pour CA, il y a appel du destructeur.

syntaxe : `~nom_de_la_classe () ;`

comme dans le cas du constructeur, le destructeur n'a pas de retour de type (donc pas de void).

```
class compte {
private:
    // etc.
public:
    compte() { //etc. } // peut être défini à l'intérieur ou à l'extérieur comme vu précédemment.
    ~compte() { // même remarque que précédemment, on va l'écrire cette fois-ci à l'extérieur.
    compte::~compte() { // destructeur
        // faire quelque chose
    }
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Très utilisé pour libérer la mémoire déjà allouée par le constructeur.

```
class tableau {
private:
    int * tab; // un pointeur sur un tableau
    int taille; // la taille du tableau
public:
    tableau(int); // constructeur
    ~tableau(); // destructeur
};

// plus d'autres fonctions ...

tableau::tableau(int Dim) { // constructeur
    if (Dim > 0) { // test si on n'a pas passé une taille négative
        tab = new int[taille = Dim]; // allocation de tab et taille = Dim
    } else { // sinon
        tab = new int[taille = 100]; // allocation d'un tableau de 100 int et taille = 100
    }
}

tableau::~tableau() { // destructeur
    delete [] tab; // libérer la mémoire allouée par le constructeur
}
```

Chapitre 5 : Classes et Objets

```
int main() {
    for (int i=0; i<10; i++){
        tableau x(10);

        // plus des choses ...
    } // ← Le programme ne se contente pas uniquement de détruire l'objet! D'abord il libère la mémoire allouée
        // par le constructeur pour cet objet ; puis par la suite, il détruit l'objet.

    // d'autres choses ...
}
```

```
    return 0;
}
```

Problème: Écrire la définition du constructeur et destructeur de la classe suivante:

```
class Id {
    char *nom; // nom de la personne.
public:
    Id(); // saisir la variable nom de l'entrée standard cin
    ~Id();
};
```

afin d'exécuter le programme suivant:

```
int main() {
    Id A,B;
    return 0;
}
```


5. Membres Statiques

Une classe est un moule à objets où chaque objet a sa propre copie des champs.

```
class compte {
    static int nbcomptes;
    char nom[20];
    double actif
    // etc.
public:
    void combien();
    // etc.
};
```

A.X	B.X
A.Y	B.Y

nbcomptes

Un champ static:
un champ de la classe et non des objets,
un champ partagé par tous les objets de la classe, un seul exemplaire pour toute la classe,
sert à stocker de l'info par tous les objets d'une même classe.

initialisation obligatoire sous cette forme:

```
int compte::nbcomptes = 0;
```

```
#include <iostream>
using namespace std;

class compte {
    // etc.
public:
    static int nbcomptes;
    compte();
    ~compte();
};

int compte::nbcomptes = 0;

compte::compte() {
    nbcomptes++;
}
compte::~compte(){
    nbcomptes--;
}

int main() {
    cout << compte::nbcomptes << endl; // 0
    compte A;
    cout << compte::nbcomptes << endl; // 1
    compte B;
    // 3 manières d'y accéder, les 3 donnent le même résultat
    cout << A.nbcomptes << endl; // 2
    cout << B.nbcomptes << endl; // 2
    cout << compte::nbcomptes << endl; //2
    return 0;
}
```

6. Un exemple complet

```
// Un autre programme sur la gestion d'un compte
#include <iostream>

using namespace std;

class compte {

private:
    char nom[21];
    double actif;
    int limite; // limite de crédit
    // Fonctions private car pas besoin qu'elles soient accessibles de l'extérieur. C'est à l'usage interne,
    // ou si vous préférez cuisine interne!
    void verif(char*);
    void verif(double); // on peut regrouper ces deux fonctions,
    void verif(int); // sera vu lors de l'étude des templates

public:
    // constructeur, même nom que la classe, les arguments sont ceux définis dans private. On peut ne pas
    // les mettre tous, or comme un constructeur est là pour initialiser alors autant initialiser toutes les données
    // de la classe.
    compte(char*, double, int);
    // on n'a pas besoin de définir un destructeur (pour faire quoi?), celui par défaut suffit.
    double depot(double);
    void affiche();
};
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 5 : Classes et Objets

```
compte::compte(char* name, double argent, int max) {

    // Vérifications exhaustives pour assurer que les données ont été correctement initialisées

    // Même nom de fonction, principe de surdéfinition de fonctions
    verif(name);
    verif(argent);
    verif(max);

    // Aucune sortie anormale, donc la copie des paramètres a lieu.
    strcpy(nom, name);
    actif = argent;
    limite = max;
}

// Définition de la fonction depot

double compte::depot(double argent) {
    verif(argent);
    actif += argent;
    return actif;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

```
// Définition de la fonction affiche
void compte::affiche() {
    cout << "nom: " << nom << endl;
    cout << "actif: " << actif << endl;
    cout << "la limite de crédit: " << limite << endl;
}

// Fonctions de vérification des données en entrée
void compte::verif(char* chaine) {
    // le +1 pour le caractère de fin de chaîne.
    if (strlen(chaine)+1 > 21) {
        cerr << "Erreur: la taille de la chaîne à copier est trop grande \n";
        cerr << "chaîne: " << chaine << " ; taille: " << strlen(chaine)+1 << " ; MAX: \
        autorisé 20\n";
        cerr << " sortie anormale du programme.\n";
        exit(1); // on quitte le programme! Ciao!
    }
}

void compte::verif(double d) {
    // Dans le cas où il y a eu un dépôt d'argent "négatif"! Ca existe de l'argent négatif? Plutôt prévenir contre
    // un bug du programmeur!
    if (d < 0) {
        cerr << "Erreur: dépôt négatif: " << d << " !!!!\n";
        cerr << " sortie anormale du programme.\n";
        exit(1); // on quitte le programme! Ciao!
    }
}
```

```
void compte::verif(int x) {
    // Idem dans le cas ou on décide de fixer une limite négative. Si c'était le cas, celui qui a ce type de compte,
    // n'est pas sorti de l'auberge!
    if (x < 0) {
        cerr << "Erreur: limite négative: " << x << " !!!!\n";
        cerr << " sortie anormale du programme.\n";
        exit(1); // on quitte le programme! Ciao!
    }
}

int main() {
    compte CA("smith", 250, 7000);
    cout << "affichage du compte CA\n";
    CA.affiche();
    cout << "affichage du compte de MLK, Jr. \n";
    compte CB("Martin Luther King, Jr.", 670, 9000);
    CB.affiche();
    return 0;
}
```

En sortie

```
affichage du compte CA
nom: smith
actif: 250
la limite de crédit: 7000
affichage du compte de MLK, Jr.
Erreur: la taille de la chaîne à copier est trop grande
chaîne: Martin Luther King, Jr. ; taille: 24 ; MAX autorisé: 20
sortie anormale du programme.
```

Solution du problème posé en page 86.

```
#include <iostream>

using namespace std;
struct compte {
    char nom[20];
    double actif;
    int limite; // limite de crédit
    void initialise();
    double depot(double);
    void affiche(); // affiche comme membre de la classe compte.
};

void compte::initialise() {
    nom[0] = '\0';
    actif = 0.0;
    limite = 2000;
}

double compte::depot(double argent) {
    actif += argent;
    return actif;
}

void compte::affiche() {
    cout << "compte CA, nom: " << nom << endl;
    cout << "compte CA, actif: " << actif << endl;
    cout << "compte CA, la limite de credit: " << limite << endl;
}
```

```
// affiche comme une fonction non membre de la classe compte
// Voir note de cours sur une des utilisation du passage par référence. Dans cet exemple on ne veut pas faire
// une recopie des éléments de la structure ; on ne touche pas aux données membres de compte, on se contente
// que d'afficher les membres d'où l'utilisation de constante.
void affiche(const compte& D) {
    cout << "compte CA, nom: " << D.nom << endl;
    cout << "compte CA, actif: " << D.actif << endl;
    cout << "compte CA, la limite de crédit: " << D.limite << endl;
}

int main() {
    compte CA; // déclaration de la variable CA du type compte CA n'est pas initialisée.

    // initialisation du compte CA
    CA.initialise(); // accès aux données membres via les fonctions membres.

    strcpy(CA.nom, "smith"); // accès direct.
    CA.actif = 123.75;
    CA.limite = 1000;
    double actuel = CA.dépot(250.88);

    cout << "actif du compte CA: " << actuel << endl;

    cout << "via la fonction affiche membre de la structure compte\n";
    CA.affiche();

    cout << "via une fonction affiche déclarée en dehors de la structure compte\n";
    affiche(CA);
    return 0;
}
```

En sortie, vous obtenez ce qui suit:

```
actif du compte CA: 374.63
via la fonction affiche membre de la structure compte
compte CA, nom: smith
compte CA, actif: 374.63
compte CA, la limite de crédit: 1000
via une fonction affiche déclarée en dehors de la structure compte
compte CA, nom: smith
compte CA, actif: 374.63
compte CA, la limite de crédit: 1000
```

Solution du problème posé en page 97

```
#include <iostream>
using namespace std;

class Id {
private:
    char *nom;
    void verif_alloc();
public:
    Id(); // constructeur
    ~Id(); // destructeur
    void affiche(); // affichage
};

Id::Id() {
    // Vérifications exhaustives pour assurer que les données ont été correctement initialisées,
    // puis copie de ces données.
    verif_alloc();
}

Id::~Id(){
    cout << "Appel du destructeur; nom: " << nom << endl;
    delete [] nom;
}

// Définition de la fonction affiche
void Id::affiche() {
    cout << "Nom: " << nom << endl;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Chapitre 5 : Classes et Objets

```
// Fonctions de vérification des données en entrée et lecture de ces données.

void Id::verif_alloc() {

    // Le code qui suit est à réécrire plus proprement en utilisant la classe string du C++. Donc définir un string, cin
    // vers le string, détecter sa taille, et finalement copier son contenu dans nom.

    char name[101];

    cout << "Entrez votre nom: MAX: 100 char\n";
    cin >> name;
    int taille = strlen(name);

    if (taille > 100) {
        cerr << "chaîne en entrée trop longue, MAX: 100 char\n";
        exit(1);
    }

    nom = new char[1+taille];
    strcpy(nom,name);
}

int main() {
    Id A,B;

    A.affiche();
    B.affiche();

    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

Résultat:

```
Entrez votre nom: MAX: 100 char
Paul
Entrez votre nom: MAX: 100 char
Marie
Nom: Paul
Nom: Marie
Appel du destructeur; nom: Marie
Appel du destructeur; nom: Paul
```