

Chapitre 12

Classes

1. Définition

- La programmation orientée objet se fait à l'aide de classes.
- Une classe comporte à la fois des champs (appelés données membres) et des fonctions (appelées méthodes).
- Les membres de la classe possèdent une visibilité (public, protected, private).
- Cette visibilité permet de régir ce qui est invocable à l'extérieur de l'instance.
- La déclaration « class » est très proche de celle de « struct » :

```
struct compte {                class compte {
    char nom[20];                char nom[20];
    // etc.                       // 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 l'on veut distinguer les membres publics des membres privés, il faut faire ce qui suit:

```
class compte {
private:
    char nom[20];
    // etc.

public :
    void initialise();
    // etc.

};
```

Mettre « private » ici est facultatif puisque par défaut les membres sont privés.

Mettre « public » à ce niveau est obligatoire si on veut déclarer des membres publics.

- 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:
    char nom[20];
    // etc.

public:
    void initialise();
    // etc.

};
```

```
class compte {
public:
    void initialise();
    // etc.

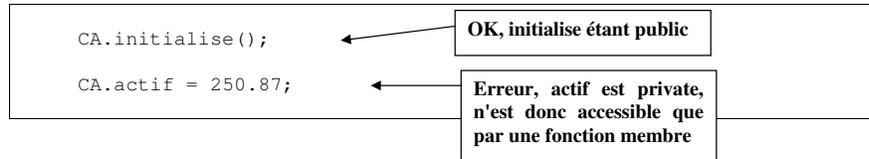
private:
    char nom[20];
    // etc.

};
```

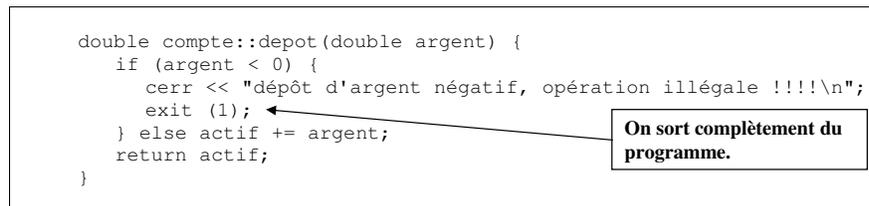
Ici c'est obligatoire pour distinguer ce qui est public de ce qui est privé.

2. Principe d'encapsulation

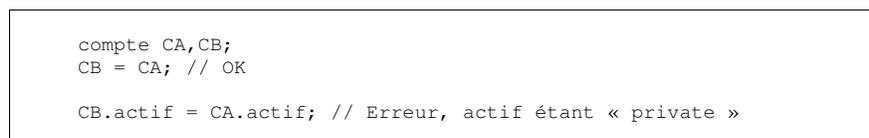
- Pour accéder aux membres de données `private`, il faut obligatoirement passer par les fonctions membres (déclarées `public`).



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



- Une fonction membre peut être déclarée aussi comme `private` si l'on ne veut pas qu'elle soit 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:



3. Constructeurs

3.1. Définitions

- La méthode appelée à la création de l'objet.
- Il 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.
- Il ne retourne rien (ne pas mettre `void`)
- Il est « public »
- Il porte le même nom que la classe
- Sa syntaxe: `nom_de_la_classe(arguments);`

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

compte::compte(char* chaine, double argent,int max){
    strcpy(nom,chaine);
    actif = argent;
    limite = max;
}

// etc.

int main() {
    compte CA("Fred",100.50,1000);
    // etc.
    return 0;
}
```

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 , à voir l'exemple complet.

Déclaration et initialisation en une seule opération

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;

```

Cette instruction génère une erreur car il n'existe aucun constructeur sans argument dans la classe « compte »

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 soit 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){ .....}
compte::compte(){..... }
// etc.
int main() {
    compte CA("Fred",100.50,1000);
    compte CB; //ok! Appel du constructeur compte()
    // etc.
    return 0;
}
```

4. Destructeur

- Chaque classe a un et, un seul destructeur (donc il 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
```

- Au moment de la « mort » de « CA », il y a eu appel du destructeur.
- Sa 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. }
    ~compte(); // a été défini cette fois-ci à l'extérieur.
};
compte::~compte() { // destructeur
    // faire quelque chose
}

```

- Le destructeur est 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) {
        tab = new int[taille = Dim];
    } else { // sinon
        tab = new int[taille = 100];
    }
}
tableau::~tableau() { // destructeur
    delete [] tab;
}

```

Un test pour vérifier si on n'a pas passé une taille négative

Allocation de tab et taille = Dim

Allocation d'un tableau de 100 int et taille = 100

Libération de la mémoire allouée par le constructeur

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

        // plus des choses ...
    } // ←
    // d'autres choses ...

    return 0;
}
```

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.

Exercice: 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();
};

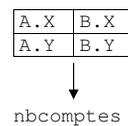
afin d'exécuter le programme suivant:

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.
};
```



- 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.
 - Il 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;
    cout << A.nbcomptes << endl; // 2
    cout << B.nbcomptes << endl; // 2
    cout << compte::nbcomptes << endl; // 2
    return 0;
}

```

**3 manières d'y accéder,
les 3 donnent le même
résultat**

6. Un exemple complet

```

#include <iostream>

using namespace std;

class compte {

private:
    char nom[21];
    double actif;
    int limite;
    void verif(char*);
    void verif(double);
    void verif(int);

public:
    compte(char*, double,int);
    double depot(double);
    void affiche();
};

```

**On n'a pas besoin de
définir un destructeur
(pour faire quoi?) celui
par défaut suffit.**

**Fonctions private car pas
besoin qu'elles soient
accessibles de l'extérieur.
C'est à l'usage interne ou
si vous préférez cuisine
interne!**

**Le 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::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 va avoir lieu.

    strcpy(nom,name);
    actif = argent;
    limite = max;
}
// Définition de la fonction depot

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

```

```

// 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 où l'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.
```