

Programmation en C/C++

Série 13

Programmation orientée objet

Surcharge de fonctions

Classes / Méthodes

Constructeurs / Destructeurs

Encapsulation de données

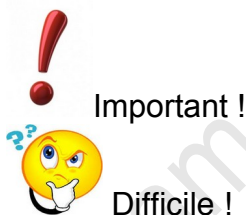
Getters/ Setters

Méthodes statiques et non statiques

Programmation modulaire

Headers / Librairies

Compilation séparée



Objectifs

Surcharger des fonctions.

Comprendre le principe de la programmation orientée objet.

Créer des classes ; des objets.

Utiliser un constructeur, un destructeur

Comprendre la notion d'encapsulation de données (*public*, *private*, *protected*)

Getters et Setters

Utiliser des méthodes statiques et non statiques

Programmer de façon modulaire

Utiliser des headers, des librairies

Réaliser une compilation séparée

1- Surcharge de fonctions en C++

Une fonction est caractérisée par son nom, son type (type de la valeur de renvoi ou *void*) et le nombre et le(s) type(s) de son (ses) paramètre(s).

Le nombre et le(s) type(s) de(s) paramètre(s) définissent la **signature** d'une fonction. Contrairement au langage C, il est possible de **surcharger** une fonction en C++ : il peut donc exister, dans un même programme, deux fonctions **de même nom** mais n'ayant pas la même signature. Le compilateur analysera les paramètres envoyés (type et nombre) et appellera la bonne fonction. Le type de valeur renvoyée doit-être identique.

Exemple :

- Une fonction ayant 2 paramètres de type entier,
- Une fonction de même nom ayant deux paramètres de type réel.

ou bien :

- Une fonction ayant 2 paramètres de type entier,
- Une fonction de même nom ayant trois paramètres de type entier.

Exercice 1

Écrire un programme en langage C++ qui dispose de deux fonctions de même nom : la première va renvoyer la somme de trois entiers qu'elle reçoit en paramètres. La seconde, de même nom, renvoie la somme de deux entiers qu'elle reçoit en paramètres. Le programme principal affiche les résultats donnés par l'appel de ces deux fonctions.

Corrigé

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

```
int Somme (int a, int b) {
    return (a+b);
}
```

```
int Somme (int a, int b, int c) {
    return (a+b+c);
}
```

```
int main() {
    cout <<"\nSomme : "<< Somme(2,3)<<endl;
    cout <<"Somme : "<< Somme(2,3,4)<<endl;
}
```

Remarque

On observe donc que les en-têtes des deux fonctions sont différents avec 2 ou 3 paramètres selon le cas. L'appel des fonctions est également différent avec 2 ou 3 paramètres envoyés selon le cas.

Exercice 2

Écrire un programme en langage C++ qui dispose de deux fonctions de même nom : la première va renvoyer la somme de deux entiers qu'elle reçoit en paramètres. La seconde, de même nom, renvoie la somme de deux réels qu'elle reçoit en paramètres. Les deux fonctions renvoient un réel (type double). Le programme principal affiche les résultats donnés par l'appel de ces deux fonctions.

Corrigé

```
#include <stdio>
#include <stdlib>
#include <iostream>

#include <string>
using namespace std;

double Somme (int a, int b) {
    return (double) a+b;
}
double Somme (double a, double b) {
    return a+b;
}

int main() {
    cout <<"\nSomme : "<< Somme(20 , 3)<<"\n"<<endl;
    cout <<"Somme : "<< Somme(20.5,3.0)<<"\n"<<endl;
}
```

Remarque

Ce programme peut ne pas fonctionner avec certains environnements ni avec des float.

2- Notions de programmation orientée objet

Au départ la programmation était purement procédurale : un programme était constitué de variables et d'une suite d'instructions ou de procédures cherchant à résoudre une tâche.

Afin de rendre le code plus lisible, on est passé à une programmation structurée utilisant des fonctions réalisant des tâches élémentaires. Le programme principal est devenu plus lisible ; chaque fonction, correspondant à un traitement spécifique, pouvant être identifiée et comprise plus facilement.

Néanmoins, dans le cas de réutilisation et d'évolution du programme, il faut comprendre l'architecture de celui-ci ; la séparation des données et des fonctions, la spécialisation de ces dernières ne sont pas forcément un atout. Il faudra réécrire, modifier profondément les fonctions, les variables dès que les données changeront...



D'où la notion de **programmation orientée objet** (P.O.O)...
La philosophie est la suivante : plutôt que d'écrire des traitements spécifiques permettant de résoudre un problème, on va créer des **objets** avec des **méthodes** travaillant sur ces objets.

Par exemple, on parlera d'un objet "Fenêtre", on peut poser, ouvrir, fermer, nettoyer, casser... une fenêtre. Tout est lié à l'objet. En programmation orientée objet, l'approche sera donc plus globale avec un **regroupement** des variables et des fonctions dans une structure appelée **classe**. A partir d'une classe, on créera des objets.

La programmation objet va permettre une programmation plus sécurisée en gérant qui peut accéder aux variables d'instance et méthodes.

La programmation orientée objet va également faciliter la réutilisation des programmes en facilitant l'évolution d'une classe.

Nous verrons ces notions un peu plus tard.

Les classes en C++

Une classe permettra donc de manipuler des objets.



Au niveau d'une classe on trouvera :

- Les **variables d'instance** (caractéristiques de l'objet)
- Les **méthodes** ou **fonctions membres** (actions pouvant être entreprises sur les objets et modifiant l'état de ces différents objets).



On dira qu'un **objet** est une **instance** d'une classe (les termes "**instanciation**" ou "**occurrence**" peuvent également être utilisés).

Ainsi, l'objet *Mésange* est une instance de la classe *Oiseau*.

La classe *Oiseau* possèdera, par exemple, les variables d'instance suivantes :

- Nom oiseau
- Taille
- Poids
- Envergure
- Couleur principale
- Durée de vie
- Type habitat
- Type nourriture

La classe *Oiseau* possèdera, par exemple, les méthodes ou fonctions membres suivantes :

- Voler
- Dormir
- Manger
- Chanter

Ces méthodes pourront s'appliquer sur toute instance de la classe et modifieront éventuellement ses variables d'instance.



Résumé

En programmation objet :

- Les **variables d'instance** correspondent à nos "anciennes" variables,
- Les **méthodes** ou **fonctions membres** correspondent à nos "anciennes" fonctions,
- Variables d'instance et méthodes sont regroupées ou **encapsulées** dans des classes,
- On créera des objets, instances de la classe.

Exemple concret en C++

Création de la classe

Créons la classe *Point* qui contient la description d'un point de coordonnées *x*, *y*, *z* (qui sont des entiers) et les méthodes ou fonctions membres *Saisie()* et *Affiche()* qui permettront respectivement de saisir et d'afficher les trois coordonnées d'un point de l'espace.

x est l'abscisse du point, *y* son ordonnée et *z* sa cote.

Voici l'implémentation de la classe :

```
class Point
{
```

```
/*Variables d'instance*/
```

```
int x;
```

```
int y;
```

```
int z;
```

```
/*méthodes*/
```

```
void Saisie ()
```

```
{
```

```
...
```

```
}
```

```

public void Affiche ()
{
...
}

}; /*fin de la classe point*/

```



Remarque

Les méthodes sont ici écrites dans la classe *Point*.
 Une classe n'a pas de fonction *main()* car ce n'est pas une application.
 Ne pas oublier le point virgule après la fermeture de la classe.

Création d'un objet

Maintenons, nous pouvons créer un objet *Omega*, au niveau du *main()*, qui est un point de l'espace avec ses trois coordonnées. Donc *Omega* est une variable ; c'est un objet de type *Point* : *Omega* est une instance de la classe *Point*.

Voici sa déclaration :
Point Omega;

Implémentation et utilisation de méthodes

On va écrire une méthode ou fonction membre *Affiche()* qui affiche les trois coordonnées du point créé et une méthode *Saisie()* qui gèrera la saisie des trois coordonnées du point.

Au niveau du programme principal, si l'on veut utiliser la méthode *Affiche()* sur l'objet *Omega* on écrira la ligne de code suivante : *Omega.Affiche()*;
 L'objet n'est pas passé en argument : on invoque la méthode *Affiche()* sur l'objet nommé *Omega*.

Code source de la méthode *Afficher()*

```

public : void Affiche ()
{
    cout << "Abscisse : " << this->x << "\n";
    cout << "Ordonnee : " << this->y << "\n";
    cout << "Cote : " << this->z << "\n";
}

```

Si *Affiche()* a été appelée par la ligne de code *Omega.Affiche()*; alors, *Affiche()* travaille sur l'objet *Omega* donc *this->x* correspond à *Omega.x* et ainsi de suite...

Au passage, on remarquera que la méthode *Affiche()* fait référence à l'objet en cours grâce au mot clé *this*.

Vous l'avez compris, *this* n'est autre qu'un pointeur sur l'objet courant !

On verra le rôle du mot clé "*public*" par la suite.

Code source de la méthode *Saisie()*

```
public : void Saisie () {  
    cout << "Saisir l'abscisse du point" << endl;  
    cin >> this->x;  
  
    cout << "Saisir l'ordonnee du point" << endl;  
    cin >> this->y;  
  
    cout << "Saisir la cote du point " << endl;  
    cin >> this->z;  
}
```

Remarque

this étant un **pointeur** sur l'objet Omega : *this* écrit directement en mémoire, donc les opérations effectuées par *Saisie()* seront visibles du *main ()*.

Exercice N°1 de cours

Implémenter la classe *Point* et le programme principal créant une instantiation de cette classe (objet de type *Point*).

Programme complet

```
#include <cstdio>  
#include <cstdlib>  
#include <iostream>  
using namespace std;  
  
class Point {  
  
    /*Variables d'instance*/  
    int x; int y; int z;  
  
    /*méthodes*/  
    public : void Saisie () {  
        cout << " Saisir l'abscisse du point " << endl;  
        cin >> this->x;  
  
        cout << " Saisir l'ordonnee du point " << endl;  
        cin >> this->y;  
  
        cout << " Saisir la cote du point " << endl;  
        cin >> this->z;  
    }  
}
```

```

public : void Affiche ()      {
cout <<"Abscisse : " << this->x << "\n";
cout <<"Ordonnee : " << this->y << "\n";
cout <<"Cote : " << this->z << "\n";
}

};    /*fin de la classe point*/

int main() {
//Déclaration d'une instance de la classe
Point Omega;

//appel des méthodes
Omega.Saisie();
Omega.Affiche();
}

```

Les constructeurs en C++

Lorsqu'on instancie une classe en créant un objet le compilateur fait appel à un constructeur qui a pour objet l'initialisation des variables d'instance de tout objet créé. Le constructeur est une méthode, parfois invisible alors appelée implicitement par le compilateur (on ne voit même pas son existence, ni l'appel au constructeur). Généralement, un constructeur est une méthode sans type de retour (même pas *void*) et de même nom que la classe. Si le constructeur ne possède pas d'arguments c'est un constructeur par défaut.

Comme les fonctions, il est possible de surcharger un constructeur en en créant un autre de même nom avec un nombre de paramètres différents. Le compilateur repèrera le bon constructeur à utiliser en fonction du nombre et du type des paramètres passés lors de l'instanciation.

Exemple de constructeur avec trois paramètres envoyés lors de l'instanciation.

```

Point (int a, int b , int c) {
this->x = a;
this->y = b;
this->z = c; }

```

Exemple d'instanciation d'un objet *Point1* :

```
Point Point1 (4,4,4);
```

Trois valeurs sont passées en paramètres au constructeur ; les trois variables d'instance de *Point1* vaudront chacune 4.

Exemple de constructeur sans paramètre lors de l'instanciation.

```

Point () {
this->x = 3;
this->y = 2;
this->z = 1; }

```


Exemple d'instanciation d'un objet *Point2* avec ce constructeur :

Point Point2 () ;

Les trois variables d'instance de *Point2* vaudront respectivement 3, 2 et 1.

Exercice N°2 de cours

Reprendre le programme précédent avec les 2 constructeurs différents puis créer 2 objets en faisant appel à ces constructeurs distincts puis les afficher.

Corrigé

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

class Point {
/*Variables d'instance*/
public : int x; int y; int z;

/*Les constructeurs*/
Point () {
this->x = 3;
this->y = 2;
this->z = 1; }

Point (int a, int b , int c) {
this->x = a;
this->y = b;
this->z = c; }

/*méthodes*/

void Saisie () {
cout << "Saisir l'abscisse du point " << endl;
cin >> this->x;
cout << "Saisir l'ordonnee du point " << endl;
cin >> this->y;
cout << "Saisir la cote du point " << endl;
cin >> this->z; }

void Affiche () {
cout <<"Abscisse : " << this->x << "\n";
cout <<"Ordonnee : " << this->y << "\n";
cout <<"Cote : " << this->z << "\n";
}

}; /*fin de la classe point*/

int main() {
```

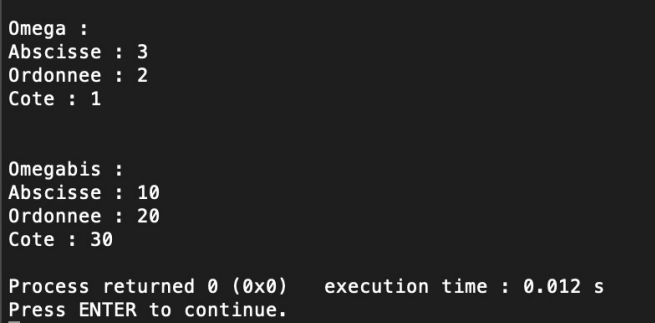
```
//Déclaration d'une instance de la classe avec le constructeur sans paramètre  
Point Omega ;
```

```
//Déclaration d'une instance de la classe avec le constructeur surchargé  
Point Omegabis (10,20,30);
```

```
//appel des méthodes //
```

```
cout <<"\n\nOmega : \n";  
Omega.Affiche();
```

```
cout <<"\n\nOmegabis : \n";  
Omegabis.Affiche();  
}
```



```
Omega :  
Abscisse : 3  
Ordonnee : 2  
Cote : 1  
  
Omegabis :  
Abscisse : 10  
Ordonnee : 20  
Cote : 30  
  
Process returned 0 (0x0)   execution time : 0.012 s  
Press ENTER to continue.
```

Affichage à l'exécution

Remarque

Dans ce programme on a créé 2 points Omega (3,2,1) et Omegabis (10,20,30).

Destructeur en C++ et allocation dynamique

La suppression d'une instance d'une classe (objet) fait appel à un destructeur.

En C++, il existe un destructeur par défaut.

Un destructeur est une méthode sans type de retour, commençant par un tilde (~). Le destructeur porte toujours le même nom que la classe et ne comporte pas de paramètre. L'absence de paramètre entraîne qu'un destructeur ne peut donc pas être surchargé : Une classe ne possèdera donc qu'un et un seul destructeur. Au niveau du programme principal ou d'une méthode, le destructeur est appelé par l'instruction *delete* suivi du nom de l'objet : on n'appelle jamais directement le destructeur.

L'objet, lorsqu'il est instancié possède sa propre zone de mémoire ; lorsqu'on le détruit il conviendra de libérer cette même zone de mémoire. Un destructeur n'est appelé (via l'instruction *delete*) que pour libérer la mémoire en détruisant un objet créé dynamiquement au départ, avec une instruction de type *new()*.

L'utilisation de destructeurs permet d'éviter des "fuites de mémoire".

Exemple de destructeur pour notre exemple

```
~ Point ()  
{ }
```

Cette ligne est à insérer éventuellement dans la classe *Point*, après le (ou les) constructeur(s).

Autre écriture externe à la classe *Point* :

```
Point :: ~ Point ()  
{ }
```

Destruction d'un objet Omega
Delete Omega ;

Autre exemple

On crée un tableau de 100 entiers :
*int * Tab = new int [100];*

On détruit un tableau de 100 entiers ; le destructeur est appelé 100 fois :
delete [] Tab;

3- Encapsulation de données en P.O.O

L'encapsulation de données en P.O.O est un terme qui signifie que les variables d'instance et méthodes sont bien encapsulées et donc protégées dans leur classe : si on les déclare en "*private*" seules des méthodes de la classe pourront appeler les autres méthodes et lire ou modifier les variables d'instance de ladite classe. Il s'agit donc d'un système de protection et de contrôle d'accès aux données (variables d'instances) et fonctions (méthodes ou fonctions membres).

C'est l'un des grands avantages de la P.O.O. :

- Une variable ne peut pas être modifiée par n'importe qui ;
- Une méthode ne peut pas être appelée par n'importe qui ;
- Une méthode ne peut pas modifier n'importe quoi !

3-1 Public / Private / Protected

Il existe trois catégories d'accès aux variables d'instance et méthodes :

- *Public*,
- *Private*,
- *Protected*.

Private

Les méthodes et variables d'instance "*private*", ne sont accessibles qu'à partir de méthodes appartenant à la même classe. Même le *main ()* ne peut y accéder ! Par défaut, les variables d'instance sont de type "*private*".

Public

Les méthodes et variables d'instance "*public*", sont accessibles à partir de méthodes de la classe ou extérieures à la classe.

Protected

Les méthodes et variables d'instance "*protected*" sont accessibles par toute méthode

de la classe ou d'une classe qui lui est dérivée (notion abordée lors du paragraphe sur l'héritage).



A retenir

- On utilise surtout les termes *private* et *public*,
- Par défaut, les variables d'instance sont de type "*private*",
- Par défaut, les fonctions membres ou méthodes sont "*public*",
- Une bonne pratique consiste donc à créer une classe avec des variables d'instances "*private*" afin de passer par les seules méthodes de cette classe pour y accéder,
- On écrira "*public* : " ou "*private* : " devant le nom des variables d'instance ou celui des méthodes pour définir leur mode.

On comprend donc maintenant le sens du " *public* : " devant les méthodes *Saisie()* et *Affiche()* : le programme principal, extérieur à la classe *Point* peut donc y accéder et les appeler !

Question 1

Que se passe-t-il si on insère "*private* : " devant les méthodes *Saisie()* et *Affiche()* ? Essayez d'observer et de prévoir le résultat.

Réponse

Les deux méthodes ou fonctions membres sont donc "*private*" et le compilateur signale une double erreur : '*Saisie*' is a private member of '*Point*' ; idem pour *Affiche()* : le programme principal ne peut y accéder : il est extérieur à la classe.

Question 2

Dans le programme de tout à l'heure, que se passera-t-il si on ajoute la ligne suivante à la fin du *main()* :
`cout << "Cote : " << Omega.z << "\n";`

Réponse

Une erreur de compilation va être signalée : on ne peut appeler la variable *z* de l'objet *Omega* de cette façon. En effet, les variables d'instances sont en mode "*private*" par défaut ; donc accessibles uniquement à partir de méthodes de la même classe donc, le programme principal ne peut pas y accéder. Pour y accéder il faudra modifier la classe pour rendre "*public*" la variable *z*.

3-2 Getters et setters

En programmation orientée objet (P.O.O.), les variables d'instance ou attributs sont par défaut en *private* ce qui signifie que l'on ne peut pas y accéder quand on est extérieur à la classe (encapsulation des données). Cela confère une certaine sécurité et l'accès aux données est réduit et contrôlé. Il est déconseillé de déclarer les variables d'instances ou attributs en *public*. Pour pouvoir manipuler ces attributs il

va donc falloir prévoir des fonctions membres *public* particulières au niveau de la classe : ce sont les accesseurs et mutateurs.

Des **getters** (accesseurs) vont permettre de "contourner" la sécurité de la P.O.O. en autorisant l'accès à une donnée *private*.

Des **setters** (mutateurs) vont également permettre de "contourner" la sécurité de la P.O.O. en autorisant la modification d'une donnée *private*.

Getters (accesseurs)

Un accesseur (getter) est un service proposé par la classe ; il permet par exemple d'accéder à des variables d'instance comme la référence, le nom et le prix d'un article malgré le fait que ces derniers soient *private*...

Un getter est une fonction extrêmement simple, placée généralement dans la classe, commençant par **get** (convention d'écriture) qui va permettre de récupérer les données attributs en *private* via un simple *return*. Les getters n'ont généralement pas d'arguments et ont pour type de retour celui de l'attribut ou variable d'instance.

Cela alourdit le code (méthodes supplémentaires) mais ne nuit pas à l'encapsulation de données.

Exemple

```
int Article :: getPrix() const
{
    return Prix;
}
```

Rq on a ajouté le mot clé *const* car le getter *getPrix* de la classe *Article* ne modifie rien elle ne fait qu'accéder à la variable *Prix* qui était en *private*.

Setters (Mutateurs)

Les mutateurs (setters) vont en quelque sorte à l'encontre de la P.O.O. qui propose une encapsulation des données c'est-à-dire une sécurité et un contrôle d'accès aux attributs ou variables d'instances déclarés en *private*.

Un setter est une fonction, placée généralement dans la classe, commençant par **set** (convention d'écriture) qui va permettre de modifier des attributs ou variables d'instances déclarés en *private*. Le setter ne renvoie rien (*void*) ; il a pour argument la valeur à assigner à la variable d'instance en *private* et donc du même type que celui de cette dernière.

Exemple

```
void Article :: set_Prix (int New_Prix) {
    Prix = New_Prix;
}
```

Rq Ici, le *setter* va modifier la valeur de la variable *Prix* d'un article.

Exemple d'implémentation de getters et setters

On crée ici une classe *Entreprise* avec un seul attribut mis en *private* qui est le chiffre d'affaire, un getter qui va accéder à cet attribut et un setter qui va permettre de modifier cet attribut.

Code source C++

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

```
class Entreprise {
```

```
private:
    int CA;
```

```
public:
```

```
// Setter Set_CA
```

```
void Set_CA(int New_CA) {
    CA = New_CA;
}
```

```
// Getter Get_CA
```

```
int Get_CA() {
    return CA;
}
```

```
}; // fin de classe
```

```
int main() {
```

```
    Entreprise Ma_Boite;
```

```
    Ma_Boite.Set_CA(1000000);
```

```
    cout << "Le chiffre d'affaire en euros est de : " << Ma_Boite.Get_CA() << endl;
```

```
    return 0;
```

```
}
```

Explications

On crée une instance de la classe *Entreprise* nommée *Ma_Boite* au niveau du *main()*. Le chiffre d'affaire de *Ma_Boite* est inaccessible à partir du *main()* car *CA* est en *private*. On modifie le chiffre d'affaire de *Ma_Boite* grâce au setter *Set_CA* à qui l'on envoie l'entier 1 000 000. Enfin, la valeur du chiffre d'affaire est récupérée par le getter *Get_CA* en vue d'être affichée. Ceci est rendu possible car getter et setter sont deux méthodes définies en *public*.

4- Méthodes statiques et non statiques

Les méthodes ou fonctions membres d'une classe sont soit **statiques** soit **non statiques**.

Les méthodes statiques ne font pas appel aux objets.

Les méthodes non-statiques font appel à des objets de la classe.

Les méthodes *Afficher()* et *Saisie()* sont des méthodes non statiques : elles font référence à un objet, ses variables d'environnement.

On appelle les méthodes non statiques en mettant un nom d'objet devant elles.

Exemple : *Omega.Afficher()*; Ici, on invoque la méthode *Afficher()* sur l'objet *Omega*.

Regardons maintenant la méthode *Compare()* qui pourrait être ajoutée à la classe *Point* :

```
public static bool Compare (int a , int b)
{
    return (a==b);
}
```

Compare (int a, int b) est une méthode statique : on repère le mot clé *static* dans son en-tête et on remarque que dans le corps de la fonction on n'utilise jamais le mot clé *this* et on ne fait jamais référence à un objet de la classe *Point*.

Au niveau du programme principal, l'appel de la fonction *Compare()* sera réalisé à l'aide d'une instruction du type :

```
Point::compare (12,15);
```

On écrit en premier le nom de la classe d'appartenance (*Point*) suivi du nom de la fonction *Compare()* puis deux nombres entiers car *Compare()* a deux arguments entiers.

L'oubli du mot clé *static* devant une fonction statique peut générer une erreur de compilation du type : "*call to non-static member function without an object argument*"

Si par exemple vous utilisez le mot clé "static" alors que la méthode travaille sur un objet, vous obtiendrez un message d'erreur lors de la compilation, du type : "*Invalid use of "static" outside of a non-static member function*".

Écriture dans des fichiers séparés

Programmation modulaire

La programmation par modules va être basée sur l'écriture d'un programme principal qui appelle des fonctions qui réalisent des tâches élémentaires, on l'a déjà vu. On parle souvent de sous-programme. Le projet est alors bien décomposé, plus lisible et donc plus facilement maintenable dans le temps.

Une programmation modulaire va également éviter la répétition des mêmes lignes de code à plusieurs endroits du programme. Imaginons que l'on doive afficher des résultats identiques plusieurs fois et à des moments différents : faire appel à une fonction d'affichage sera très utile et évitera une redondance de code.

Une programmation pas modules va entraîner des gains de temps de développement : en effet, plusieurs développeurs vont travailler en parallèle chacun

sur un module en respectant un cahier des charges précis, des noms de variables et de fonctions explicites, des paramètres...Il faudra ensuite écrire le programme principal qui va orchestrer le tout et utiliser ces différents modules.

Enfin, la programmation modulaire va permettre de générer des modules indépendants de la problématique, génériques et qui seront donc réutilisables dans d'autres programmes informatiques. Des gains de temps et de coûts seront alors observés.

Headers et programmes C++

Un fichier source C++ peut comporter un certain nombre de méthodes et un programme principal y faisant appel ; le tout dans un même fichier d'extension .cpp. Il est cependant possible de déclarer des fonctions, des variables dans un fichier d'en-tête (header) séparé (fichier d'extension .h) et d'écrire le programme source C++ dans un autre fichier d'extension .cpp. Ce programme principal devra néanmoins comporter au tout début une directive de compilation visant à inclure le fichier d'en-tête (`#include`) ; lors de la précompilation ces fichiers d'en-tête seront ajoutés au programme source et l'ensemble sera compilé pour générer un fichier objet. C'est ce qui est réalisé quand, en début de programme, on écrit des lignes comme : `#include <cstdio>` ; `#include <cstdlib>` ; `#include <iostream>`.

Librairies C++

Il existe en C++ des librairies qui proposent des fonctions ou méthodes qui permettent par exemple de gérer des affichages à l'écran, des aspects graphiques, des calculs mathématiques, des interfaces homme-machine, des traitements d'images, le son...

Il faut bien évidemment connaître leur contenu et leur description pour pouvoir pleinement en tirer parti lors de l'écriture d'un code source C++.

Exemple en C++

Dans le programme C++ ci-dessous on inclut la librairie standard *cmath* (header *math.h* hérité du C) afin de pouvoir utiliser la fonction prédéfinie *pow(a,b)* qui retournera a^b .

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

int main() {
    cout << pow(2,3)<<endl;
    return 0;
}
```

Remarque Le programme va retourner 2^3 c'est-à-dire 8.

Le même programme en C

```
#include <stdio.h>
#include <math.h>
```



```
int main() {

    printf ("%f", pow(2,3));
    return 0;
}
```

Remarque %f est utilisé car *pow* retourne un réel.

Classes et programme principal

Un fichier source C++ peut comporter par exemple une classe avec ses variables d'instance et fonctions membres puis un programme principal ; le tout dans un même fichier.

On prendra l'habitude de séparer le programme principal des classes :

- Un header avec la classe, les variables d'instance et les prototypes des méthodes (fichier d'extension *.h*).
- Le fichier source C++ avec le corps des fonctions membres et les constructeurs. Ce fichier d'extension *.cpp* aura le même nom que la classe et fera référence au *header* grâce à une directive de compilation du type : `#include "header.h"`, en début de programme.
- Le *main()* comportera également une directive de compilation du type : `#include "header.h"`, en début de programme.

Implémentation séparée d'une classe

Exercice de cours

On cherchera ici à implémenter un programme qui fait appel à une classe *Nombre* dont la variable d'instance est un nombre entier et possédant un constructeur et un getter permettant de récupérer plus tard la valeur du nombre entier.

Tout sera écrit dans le programme principal dans un premier temps puis nous séparerons l'implémentation en 3 fichiers :

- La classe *Nombre.h*
- Le programme *Nombre.cpp* implémentant constructeur et getter
- Le programme principal : *main()*

Corrigé 1 avec tout le code source dans le *main()*

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

```
class Nombre{
```

```
// variables d'instance
```

```
private:
```

```
    int num;
```

```
public:
```

```

// constructeur Nombre
Nombre(int n): num(n) {}

// Getter getNombre
int GetNombre()
{
    return num; }
};

int main() {

// Création de l'objet Nb et envoi de 44 au constructeur
Nombre Nb(44);

//Récupération et affichage de la valeur de Nb grâce au getter
cout << Nb.GetNombre() << endl;
return 0;
}

```

Corrigé 2 avec Implémentation en 3 parties **Classe Nombre.h**

```

class Nombre{

// variables d'instance
private:
    int num;

public:
// constructeur Nombre
    Nombre(int n);

// Getter getNombre
    int GetNombre();
};

```

Fichier Nombre.cpp

```

#include "Nombre.h"

//implémentation du constructeur
Nombre::Nombre(int n): num(n) {}

//implémentation du getter
int Nombre::GetNombre()
{
    return num; }

```

Programme principal (main())

```

#include <iostream>

```

```

#include "Nombre.h"
using namespace std;

int main() {

// Création de l'objet Nb et envoi de 44 au constructeur
Nombre Nb(44);

//Récupération et affichage de la valeur de Nb grâce au getter
cout << Nb.GetNombre() << endl;
return 0;
}

```

Compilation séparée

La compilation, on l'a déjà vu, permet de générer un fichier objet (.o) puis l'édition de liens génère un fichier (.out) exécutable par la machine ; sous réserve d'absence d'erreurs.

Lorsque l'on dispose de plusieurs fichiers sources en parallèle il est possible de les compiler de façon séparée pour générer de multiples fichiers objets. Il faudra ensuite expliciter à l'éditeur de liens quels sont les fichiers objet à lier. Cette technique peut faire gagner du temps en évitant de recompiler des codes sources existants et déjà compilés. Néanmoins, plusieurs difficultés vont apparaître comme connaître le bon ordre de compilation des codes sources (afin que les variables, fonctions soient bien reconnus par le compilateur) et connaître l'ensemble des fichiers dont aura besoin l'éditeur de liens pour générer l'exécutable.

Espace de noms

En C++, il est possible de créer des espaces de noms dans lesquels seront encapsulées des données. On pourra alors utiliser des variables, des constantes, des fonctions, des classes de même nom dans des espaces de noms différents ou à l'extérieur de ces derniers sans qu'il y ait pour autant homonymie. On utilisera l'opérateur de portée :: lorsqu'on parlera des variables ou constantes définies dans l'espace de noms.

Exemple : `NameSpace::V2 = 3;` on affecte 3 à la variable V2 de l'espace de noms `NameSpace`

Il est possible d'imbriquer des espaces de noms les uns dans les autres.

Exemple de cours 1

Qu'affiche le programme suivant ?

```

#include <iostream>
using namespace std;

// définition d'une variable globale
int Var1 = 1;

// définition d'un espace de noms
namespace Espace1 {

```

```

// définition de deux variables dans l'espace de noms
int Var1 = 2;
int Var2;
}

int main(){
// Définition d'une variable dans le main()
int Var2 = 4;

// affectation de valeur à la variable V2 de l'espace de nom
Espace1::Var2 = 3;

cout << "Var1 Globale : " << Var1 << endl;
cout<<"Var1 de l'espace de noms : " << Espace1::Var1 << endl;

cout<<"Var2 de l'espace de noms : " << Espace1::Var2 << endl;
cout<<"Var2 du main() : " << Var2 << endl;

return 0;
}

```

Réponse

```

Var1 Globale : 1
Var1 de l'espace de noms : 2
Var2 de l'espace de noms : 3
Var2 du main() : 4

Process returned 0 (0x0)   execution time : 0.008 s
Press ENTER to continue.

```

On a dans ce programme une variable *Var1* globale et une variable *Var1* de l'espace de noms qui sont différentes.

On a également une variable *Var2* de l'espace de noms et une variable *Var2* locale au *main()* qui sont différentes.

Exemple de cours 2

Écrire le programme C++ qui crée deux espaces de noms nommés *Espace1* avec une variable *Var* qui vaut 10 et une Fonction *Affiche()* et *Espace2* avec une variable *Var* qui vaut 20 et une Fonction *Affiche()*. Le programme affiche ensuite les deux variables et appelle les deux fonctions comme suit :

```

Var de l'Espace1 : 10
Fonction de l'Espace1

Var de l'Espace2 : 20
Fonction de l'Espace2

Process returned 0 (0x0)   execution time : 0.010 s
Press ENTER to continue.

```

Solution

```

#include <iostream>
using namespace std;

```

```

namespace Espace1 {
int Var = 10;

void F_Espace () {
cout<<"Fonction de l'Espace1"<<endl;
}
}

namespace Espace2 {
int Var = 20;

void F_Espace () {
cout<<"Fonction de l'Espace2"<<endl;
}

}

int main(){

cout<<"Var de l'Espace1 : " << Espace1::Var << endl;
Espace1::F_Espace();
cout<< endl;

cout<<"Var de l'Espace2 : " << Espace2::Var << endl;
Espace2::F_Espace();
cout<< endl;

return 0;
}

```

EXERCICES

Exercice 1

Énoncé

On travaillera ici dans le contexte d'une agence bancaire :

Un compte est caractérisé par un numéro de compte (entier) et un solde (réel).

On va créer une classe *Compte* avec une méthode *Affiche* qui affichera, sur une même ligne, le numéro de compte et son solde. On créera un constructeur pour le compte 234567 ayant pour solde 490.85 €. Aucune saisie ne sera effectuée au clavier.

Au niveau du programme principal on va créer le compte puis afficher ses caractéristiques.

Résultat obtenu

```
Affichage des comptes :  
Numero de compte : 234567          Solde : 490.85
```

Corrigé

```
#include <stdio>  
#include <stdlib>  
#include <iostream>  
  
using namespace std;  
  
class Compte {  
    /*Variables d'instance*/  
  
    int NumCompte;  
    float Solde;  
  
    // Constructeur  
    public : Compte (int a, float b) {  
        this->NumCompte = a;  
        this->Solde = b;  
    }  
  
    //méthodes  
  
    public : void Affiche () {  
        cout << "Numero de compte : " << this->NumCompte << "\t";  
        cout << "Solde : " << this->Solde << "\n";  
    }  
  
}; //fin de la classe Compte
```

```

int main() {
//Création d'une instance de la classe
Compte Cpt1(234567,490.85);

//appel des méthodes
cout <<"\nAffichage des comptes : \n";
Cpt1.Affiche();
cout <<"\n";
}

```

Remarque

Le constructeur est rendu *public* pour être accessible à partir du *main()*. Ce n'est pas la meilleure façon de sécuriser les données et nous allons améliorer cela par la suite. La méthode *Affiche()* est "*public*" pour être accessible du *main()*.

Remarque 2

Le programme principal fait appel à la méthode *Affiche()* qui est située dans la classe *Compte* et qui peut donc accéder aux variables d'instance des objets même si ceux-ci sont "*private*".

Si l'on souhaite accéder aux variables d'instance directement à partir du *main()* il faudrait les définir comme "*public*" en rajoutant "*public* : " devant leur ligne de déclaration.

Exercice 2

Énoncé

On va améliorer le programme précédent (d'un point de vue de la sécurité) en ajoutant une nouvelle méthode à la classe *Compte* : une méthode *Saisie* qui va autoriser la saisie d'un nouveau compte et de son solde.

Le programme principal va créer un compte et afficher ses caractéristiques. Attention au constructeur !

Corrigé

```

#include <stdio>
#include <stdlib>
#include <iostream>

using namespace std;

class Compte {
/*Variables d'instance*/

int NumCompte;
float Solde;

//méthodes

public : void Saisie () {
cout << "Saisir le numero de compte : \t" << endl;

```

```

cin >> this->NumCompte;

cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde;
}

public : void Affiche () {
cout << "Numero de compte : " << this->NumCompte << "\t";
cout << "Solde : " << this->Solde << "\n";
}

}; //fin de la classe Compte

int main() {
//Création d'une instance de la classe
Compte Cpt1;

//appel des méthodes
Cpt1.Saisie();
cout << "\nAffichage des comptes : \n";
Cpt1.Affiche();
cout << "\n";
}

```

Remarque

- On n'utilise ici que le constructeur par défaut !
- La méthode *Saisie()* est "*public*" pour pouvoir être appelée par le programme principal.
- Les variables d'environnement sont par défaut *private* car seules les méthodes *Saisie()* et *Affiche()*, de la même classe, les utilisent : les données sont bien protégées.

Exercice 3

Énoncé

On va continuer à améliorer le programme principal en créant deux nouvelles méthodes *Crediter* et *Debiter* qui vont respectivement créditer et débiter une somme à chaque compte. Écrire un programme principal qui va utiliser ces deux nouvelles méthodes.

```

Saisir le numero de compte :
123456
Saisir le solde du compte :
100.0
Numero de compte : 123456      Solde : 100

Creditons un compte
Saisir la somme a crediter :
20.45
Numero de compte : 123456      Solde : 120.45

Debitons un compte
Saisir la somme a debiter :
100.99
Numero de compte : 123456      Solde : 19.46

```


Exemple d'utilisation

Corrigé

```
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

class Compte {
/*Variables d'instance*/
int NumCompte;
float Solde;

//méthodes
public : void Saisie () {
cout << "Saisir le numero de compte : \t" << endl;
cin >> this->NumCompte;
cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde;
}

public : void Crediter () {
float somme;
cout << "\nCrediter un compte \n";
cout << "Saisir la somme a crediter : \t" << endl;
cin >> somme;
this->Solde+=somme;
}

public : void Debiter () {
float somme;
cout << "\nDebiter un compte \n";
cout << "Saisir la somme a debiter : \t" << endl;
cin >> somme;
this->Solde-=somme;
}

public : void Affiche () {
cout << "Numero de compte : " << this->NumCompte << "\t";
cout << "Solde : " << this->Solde << "\n";
}

}; //fin de la classe Compte

int main() {
Compte Cpt1;

Cpt1.Saisie();
Cpt1.Affiche();
```

```

Cpt1.Crediter();
Cpt1.Affiche();

Cpt1.Debiter();
Cpt1.Affiche();
cout << "\n";
}

```

Exercice 4

Énoncé

Créer une méthode *Red* qui va vérifier si le compte est dans le rouge (solde ≤ 0). On crée un compte par saisie, on effectue un crédit puis un débit. A chaque fois, le solde s'affiche et, à la fin, le programme affiche une alerte uniquement si le compte est dans le rouge.

```

Saisir le numero de compte :
123456
Saisir le solde du compte :
1000
Numero de compte : 123456          Solde : 1000

Creditons un compte
Saisir la somme a crediter :
500
Numero de compte : 123456          Solde : 1500

Debitons un compte
Saisir la somme a debiter :
2000
Numero de compte : 123456          Solde : -500
Compte dans le rouge !!

```

Exemple d'utilisation de la méthode EstRed

Corrigé

```

#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

class Compte {
/*Variables d'instance*/
int NumCompte;
float Solde;

//méthodes
public : void Saisie () {
cout << "Saisir le numero de compte : \t" << endl;
cin >> this->NumCompte;
cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde;
}
}

```

```
}
```

```
public : void Crediter () {  
    float somme;  
    cout << "\nCreditions un compte \n";  
    cout << "Saisir la somme a crediter : \t" << endl;  
    cin >> somme;  
    this->Solde+=somme;  
}
```

```
public : void Debiter () {  
    float somme;  
    cout << "\nDebitons un compte \n";  
    cout << "Saisir la somme a debiter : \t" << endl;  
    cin >> somme;  
    this->Solde-=somme;  
}
```

```
public : void Affiche () {  
    cout << "Numero de compte : " << this->NumCompte << "\t";  
    cout << "Solde : " << this->Solde << "\n";  
}
```

```
public : bool EstRed () {  
    return (this->Solde<=0);  
}  
}; //fin de la classe Compte
```

```
int main() {  
    Compte Cpt1;
```

```
    Cpt1.Saisie();  
    Cpt1.Affiche();
```

```
    Cpt1.Crediter();  
    Cpt1.Affiche();
```

```
    Cpt1.Debiter();  
    Cpt1.Affiche();
```

```
    if (Cpt1.EstRed() )  
        cout << "Compte dans le rouge !!\n";
```

```
    cout << "\n";  
}
```

Remarque

if (Cpt1.EstRed())... est équivalent à if (Cpt1.EstRed()==true)...

Exercice 5

Énoncé

Modifier la classe *Compte* afin que chaque compte ait un montant (réel) d'autorisation de découvert de 500 euros. Modifier la méthode *Debiter* afin qu'elle tienne compte de cette autorisation de découvert et interdise tout débit qui entraînerait un solde au delà du découvert autorisé. Modifier le programme principal afin qu'il utilise ces nouvelles fonctionnalités.

Exemple Un compte avec un solde de 300 € et une autorisation de découvert de 500 € pourra donc être débité de 800 € au maximum ; il restera néanmoins dans le rouge !

Corrigé

```
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;

class Compte {
/*Variables d'instance*/
int NumCompte;
float Solde;
float Decouvert;

//méthodes
public : void Saisie () {
cout << "Saisir le numero de compte : \t" << endl;
cin >> this->NumCompte;
cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde;
this->Decouvert = 500;
}

public : void Crediter () {
float somme;
cout << "\nCrediter un compte \n";
cout << "Saisir la somme a crediter : \t" << endl;
cin >> somme;
this->Solde+=somme;
}

public : void Debiter () {
float somme;
cout << "\nDebitons un compte \n";
cout << "Saisir la somme a debiter : \t" << endl;
cin >> somme;
if (this->Solde-somme>=this->Decouvert)
this->Solde -= somme;
}
```

```

else
    cout<<"Debit non autorise ! \n";
}

public : void Affiche () {
    cout <<"Numero de compte : " << this->NumCompte << "\t";
    cout <<"Solde : " << this->Solde << "\n";
}

public : bool EstRed () {
    return (this->Solde<=0);
}
}; //fin de la classe Compte

int main() {
    Compte Cpt1;

    Cpt1.Saisie();
    Cpt1.Affiche();

    Cpt1.Crediter();
    Cpt1.Affiche();

    Cpt1.Debiter();
    Cpt1.Affiche();

    if (Cpt1.EstRed() )
        cout <<"Compte dans le rouge !!\n";

    cout <<"\n";
}

```

Remarque

On a ajouté une variable d'environnement de type *float* correspondant au découvert. Le découvert est fixé à 500 euros dans la méthode *Saisie()*. Il suffit de comparer (Solde-Somme à débiter) à (- Decouvert) pour autoriser le débit ou non.

Exercice 6

Énoncé

Exercice pédagogique.

On va maintenant créer successivement deux comptes, c'est-à-dire deux instantiations de la classe *Compte*.

Modifier la méthode *Saisie()* qui vérifie si le compte existe déjà.

Contraintes

Saisie () renvoie un booléen (type *bool*)

Si la situation est favorable le second compte est enregistré.

Le programme principal va créer maintenant deux comptes (différents !) et afficher leurs caractéristiques s'ils sont différents.

Corrigé

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

class Compte {
/*Variables d'instance*/
public :
int NumCompte;
float Solde;

//méthodes
public : bool Saisie (int num1) {
int num2; cout << "Saisir le numero du nouveau compte : \t" << endl;
cin >> num2;

if (num2 == num1) {
cout << "Compte deja existant\n";
return 0;
}
else {
this->NumCompte = num2;
cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde; return 1;
}
}

public : void Affiche () {
cout << "Numero du compte : " << this->NumCompte << "\t";
cout << "Solde du compte: " << this->Solde << "\n";
}

}; //fin de la classe Compte

int main() {
//Création de deux instances de la classe
Compte Cpt1, Cpt2;
Cpt1.NumCompte = 0;

cout << "\nAffichage des comptes : \n";
if (Cpt1.Saisie(Cpt1.NumCompte))
Cpt1.Affiche();
if (Cpt2.Saisie(Cpt1.NumCompte))
Cpt2.Affiche();

cout << "\n";
```

}

Remarque

Pour des raisons pratiques on initialise *Cpt1.NumCompte* à 0.

Tout est repassé en "*public*". En effet, *Saisie()* doit recevoir un numéro de compte en paramètre à partir du programme principal donc les variables d'instance sont en "*public*". Nous verrons comment améliorer, automatiser et sécuriser le programme lorsqu'on travaillera sur un tableau de comptes.

Exercice 7

Énoncé

Écrire le programme principal qui crée 6 comptes différents puis affiche les caractéristiques de ces comptes.

Contraintes

Créer un tableau de 6 pointeurs sur des comptes au niveau du *main ()*.

On utilisera deux méthodes *Saisie()* et *Affiche()*.

On ne vérifiera pas ici si deux numéros de comptes sont identiques.

Creation des comptes :

Saisie des valeurs :

```
Saisir le numero du nouveau compte :  
1  
Saisir le solde du compte :  
11  
Saisir le numero du nouveau compte :  
2  
Saisir le solde du compte :  
22  
Saisir le numero du nouveau compte :  
3  
Saisir le solde du compte :  
33  
Saisir le numero du nouveau compte :  
4  
Saisir le solde du compte :  
44.44  
Saisir le numero du nouveau compte :  
5  
Saisir le solde du compte :  
55  
Saisir le numero du nouveau compte :  
6  
Saisir le solde du compte :  
66.66
```

Affichage des comptes :

```
Numero du compte : 1      Solde du compte: 11  
Numero du compte : 2      Solde du compte: 22  
Numero du compte : 3      Solde du compte: 33  
Numero du compte : 4      Solde du compte: 44.44  
Numero du compte : 5      Solde du compte: 55  
Numero du compte : 6      Solde du compte: 66.66
```

Corrigé

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

```
class Compte {  
int NumCompte;  
float Solde;
```

```
public : void Saisie () {  
    cout << "Saisir le numero du nouveau compte : \t" << endl;  
    cin >>this->NumCompte;  
    cout << "Saisir le solde du compte : \t" << endl;
```



```

        cin >> this->Solde;
    }

public : void Affiche () {
    cout <<"Numero du compte : " << this->NumCompte << "\t";
    cout <<"Solde du compte: " << this->Solde << "\n";
}

};

int main() {
int i,Taille;
Compte * T[6];

cout <<"\nSaisie des valeurs : \n";

for (i=0; i<6; i++) {
    T[i]= new Compte;
    T[i]->Saisie();
}

cout <<"\nAffichage des comptes : \n";

for (i=0 ; i<6;i++)
    T[i]->Affiche();
cout <<"\n";
}

```

Remarque

`Compte * T[6];` on a créé ici un tableau de 6 pointeurs sur des variables de type `Compte`.

`T[i]= new Compte;` à chaque tour de boucle, on alloue de l'espace en mémoire pour créer un nouveau `Compte` et on enregistre son adresse dans le pointeur `T[i]`. On dira que `T[i]` pointe sur ce nouveau compte.

`T[i]->Saisie();` on invoque la méthode `Saisie()` à l'aide de l'adresse d'un compte (objet).

Si l'on avait saisi `T[i].Saisie()` ; l'erreur ci-après aurait été signalée : **"member reference type 'Compte' is a pointer; maybe you meant to use '->'** ; rappelant que `T[i]` est un pointeur.

Exercice 8

Énoncé

Ajouter une méthode statique dans la classe `Compte`, qui retourne la valeur du solde le plus élevé de tous les comptes.

Corrigé

```

#include <cstdio>
#include <cstdlib>
#include <iostream>

```

```

using namespace std;

class Compte {
int NumCompte;
float Solde;

public : static float PlusGrand (Compte *T[], int nbelem) {
int i;
float Max =T[0]->Solde;

for (i=0; i<nbelem; i++)
{
if (T[i]->Solde>Max)
Max = T[i]->Solde ;
}
return Max;
}

public : void Saisie () {
cout << "Saisir le numero du nouveau compte : \t" << endl;
cin >>this->NumCompte;
cout << "Saisir le solde du compte : \t" << endl;
cin >> this->Solde;
}

public : void Affiche () {
cout <<"Numero du compte : " << this->NumCompte << "\t";
cout <<"Solde du compte: " << this->Solde << "\n";
}

}; //fermeture de la classe

int main() {
int i,Taille;
Compte *T[6];

cout <<"\nSaisie des valeurs : \n";
for (i=0; i<6; i++)
{
T[i]= new Compte;
T[i]->Saisie();
}

cout <<"\nAffichage des comptes : \n";
for (i=0 ; i<6;i++)
T[i]->Affiche();

cout <<"\n";

```

```
cout << "Solde le plus eleve : " << Compte:: PlusGrand(T,6) << "\n";
}
```

Remarque

La fonction *PlusGrand* est bien statique ; en effet, on l'appelle sans référence à un objet : on lui envoie tout simplement le tableau de pointeurs et le nombre de pointeurs.

Exercice 9

Énoncé

Améliorer le programme principal en demandant le nombre de comptes que l'on souhaite créer ; afficher tous les comptes puis tous les comptes dans le rouge en faisant appel à la méthode publique non statique *EstRed()*.

```
Saisissez le nombre de nombre de comptes voulu : 3

Saisie des valeurs :
Saisir le numero du nouveau compte :
123456
Saisir le solde du compte :
-23
Saisir le numero du nouveau compte :
345
Saisir le solde du compte :
5000.78
Saisir le numero du nouveau compte :
3555
Saisir le solde du compte :
-400.56

Affichage des comptes :
Numero du compte : 123456      Solde du compte: -23
Numero du compte : 345      Solde du compte: 5000.78
Numero du compte : 3555      Solde du compte: -400.56

Affichage des comptes dans le rouge:
Numero du compte : 123456      Solde du compte: -23
Numero du compte : 3555      Solde du compte: -400.56
```

Exemple à l'exécution

Corrigé

```
#include <cstdio>
```

```
#include <cstdlib>
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Compte {
int NumCompte;
float Solde;    /
```

```
public : void Saisie () {
cout << "Saisir le numero du nouveau compte : " << endl;
cin >>this->NumCompte;
```

```

cout << "Saisir le solde du compte : " << endl;
cin >> this->Solde;
}

```

```

public : bool EstRed() {
return (this->Solde<=0);
}

```

```

public : void Affiche () {
cout <<"Numero du compte : " << this->NumCompte << "\t";
cout <<"Solde du compte: " << this->Solde << "\n";
}
};

```

```

int main() {
int i,Taille;
cout<<"\nSaisissez le nombre de comptes voulus : ";
cin>>Taille;

```

```

//on crée un tableau avec "Taille" comptes bancaires
Compte * T[Taille];

```

```

cout <<"\nSaisie des valeurs : \n";
for (i=0; i<Taille; i++)
{
T[i]= new Compte;
T[i]->Saisie();
}

```

```

cout <<"\nAffichage des comptes : \n";
for (i=0 ; i<Taille;i++)
T[i]->Affiche();
cout <<"\n";

```

```

cout <<"\nAffichage des comptes dans le rouge: \n";
for (i=0 ; i<Taille;i++)
if (T[i]->EstRed())
T[i]->Affiche();
cout <<"\n";
}

```