

# Introduction à la programmation en C++

## Les objets

---

Nicolas Audebert

Vendredi 16 novembre 2018



## Rendus de TP et des exercices

Les rendus se font sur [Educnet](#), même en cas de retard. **Pas par mail.**

1. Le code rendu **doit compiler**.
2. Le code rendu doit **être propre** (indentation, noms de variables clairs).
3. Le code rendu doit **être commenté** (réponses aux questions, fonctionnement du code).
4. Rassembler le code dans une seule archive (**.zip, .rar, .tar.gz**, etc.).
5. Le code doit contenir **les noms des deux binômes** le cas échéant.

Un exercice ou un TP rendu en retard ou ne respectant pas une des consignes ci-dessus sera pénalisé.

## Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

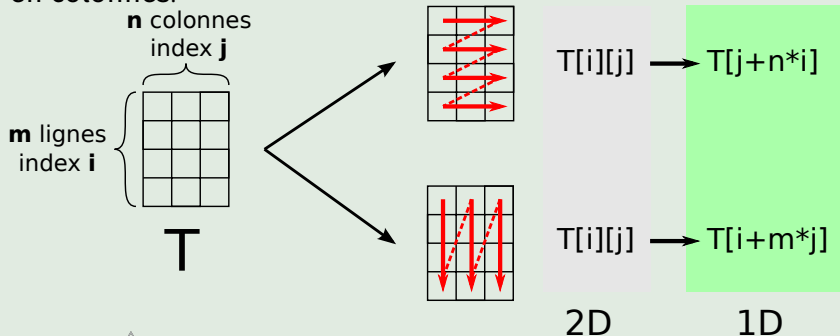
TP

## Tableaux 2D en C++

Les tableaux 2D de taille constante sont autorisés en C++, mais peu pratiques.

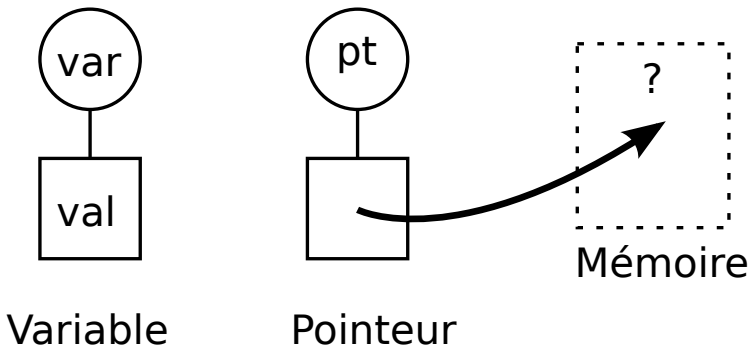
## En pratique

En pratique, on utilise des tableaux 1D que l'on parcourt **en lignes** ou **en colonnes**.



## Définition

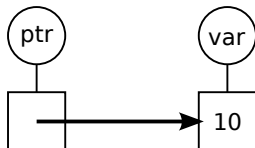
Un **pointeur** est une variable qui stocke une adresse vers une zone mémoire (tableau ou variable) dans la pile ou dans le **tas**.



# Utilisation des pointeurs

Les pointeurs sont caractérisés par le symbole `*`.

```
double* ptr; // un pointeur vers un double
int* ptr; // un pointeur vers un entier
int test = 10;
ptr = &test; // le pointeur redirige vers test
int val = *ptr; // val contient 10
```



Pour récupérer l'adresse d'une variable on utilise le `&`. Pour récupérer la valeur pointée par une adresse, on utilise `*`.

L'intérêt d'utiliser des pointeurs avec des variables classiques est limité.

## Des pointeurs pour le tas

Les pointeurs sont la porte d'entrée vers le tas (la mémoire de l'ordinateur).

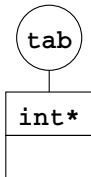
- ▶ Créer une variable dans le tas : **new**
- ▶ Supprimer une variable dans le tas : **delete**

## Gestion de la mémoire

Chaque appel à **new** doit être suivi par un unique appel à **delete**.

```
double* tab;  
int n=5;  
tab = new double[n];  
  
for(int i=0; i<n; i++){  
    tab[i] = 2*i;  
}  
  
delete[] tab;
```

## La pile

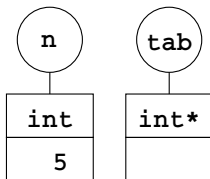


## Le tas



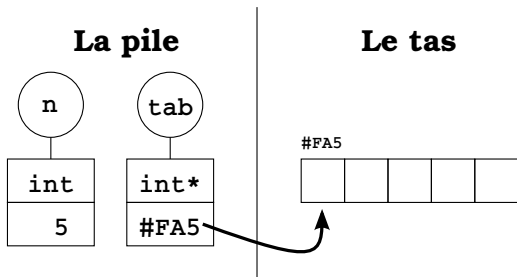
```
double* tab;  
int n=5;  
tab = new double[n];  
  
for(int i=0; i<n; i++){  
    tab[i] = 2*i;  
}  
  
delete[] tab;
```

## La pile



## Le tas

```
double* tab;  
int n=5;  
tab = new double[n];  
  
for(int i=0; i<n; i++){  
    tab[i] = 2*i;  
}  
  
delete[] tab;
```

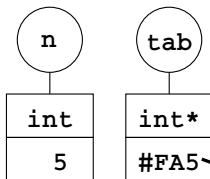


```
double* tab;  
int n=5;  
tab = new double[n];
```

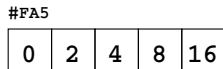
```
for(int i=0; i<n; i++){  
    tab[i] = 2*i;  
}
```

```
delete[] tab;
```

## La pile

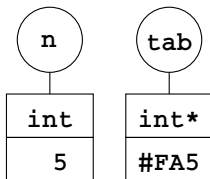


## Le tas



```
double* tab;  
int n=5;  
tab = new double[n];  
  
for(int i=0; i<n; i++){  
    tab[i] = 2*i;  
}  
  
delete[] tab;
```

## La pile



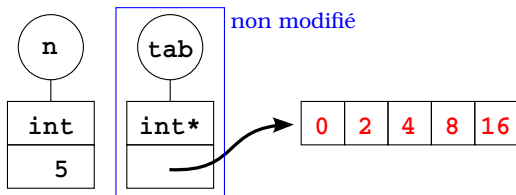
## Le tas

## Modifier la variable / tableau désigné par le pointeurs

- ▶ Pas besoin de passage par référence : on ne modifie pas le pointeur (l'adresse), seulement les valeurs stockées dans la zone de la mémoire désignées par le pointeur.
- ▶ On peut utiliser les fonctions créées pour les tableaux statiques.

```
void fill(double* tab, int n){  
    for(int i=0; i<n; i++)  
        tab[i] = 2*i;  
}
```

```
double* t;  
int taille=5;  
t = new double[taille];  
fill(t,taille);  
delete[] tab;
```

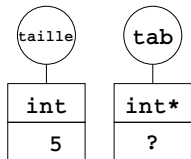


## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```

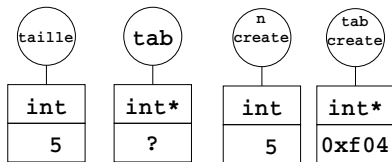


## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

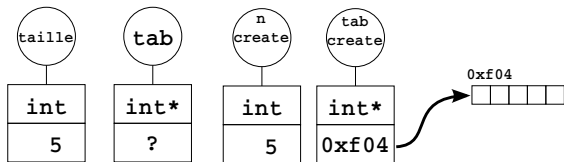
```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```



## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}  
  
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```





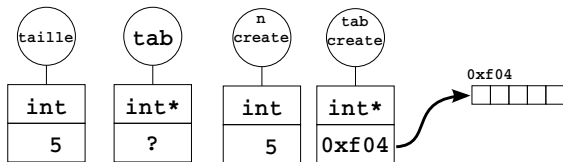
## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```



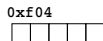
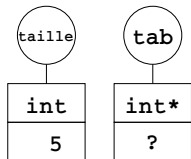
## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```



## Modifier le pointeur

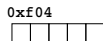
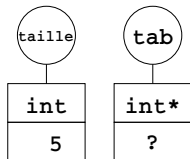
- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```

--> ?



## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

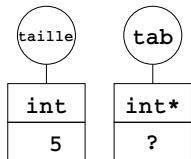
```
void create(double* tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;
```

--> ?

```
delete[] tab; --> ERREUR non alloué
```



0xf04  
□□□□

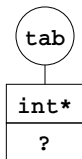
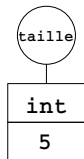
**ERREUR FUITE DE MÉMOIRE**

## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

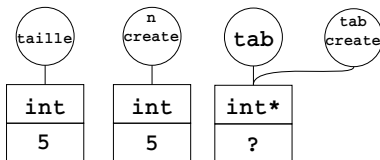
```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```



## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n) {  
    tab = new double[n];  
    cout << tab << endl;  
}  
  
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```

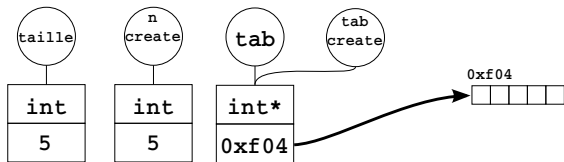


## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```



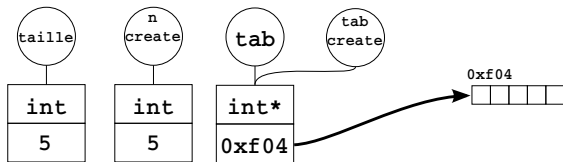
## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```





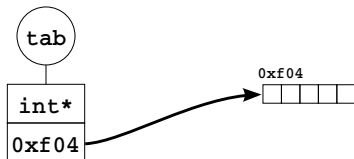
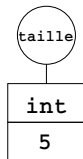
## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

--> 0xf04

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl;  
delete[] tab;
```

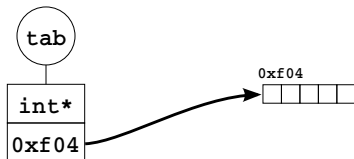
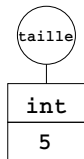


## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl; --> 0xf04  
delete[] tab;
```

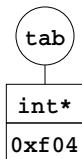
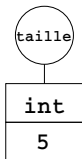


## Modifier le pointeur

- ▶ Il faut faire un passage par référence : on modifie l'adresse stockée par le pointeur.

```
void create(double* &tab, int n){  
    tab = new double[n];  
    cout << tab << endl;  
}
```

```
double* t;  
int taille=5;  
create(t,taille);  
cout << t << endl; --> 0xf04  
delete[] tab;
```



L'égalité de pointeurs est autorisée.

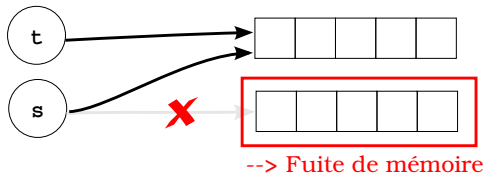
## Attention

- ▶ Il y a des risques de fuite de mémoire
- ▶ Deux pointeurs égaux renvoient au même espace mémoire
- ▶ Il n'y a pas création d'un nouveau tableau

```
double* t, s;  
int n=5;  
t = new double[n];  
s = new double[n];
```

```
s = t;
```

```
... --> Fuite de mémoire
```



L'égalité de pointeurs est autorisée.

## Attention

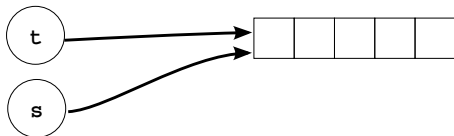
- ▶ Il y a des risques de fuite de mémoire
- ▶ Deux pointeurs égaux renvoient au même espace mémoire
- ▶ Il n'y a pas création d'un nouveau tableau

```
double* t,s;  
int n=5;  
t = new double[n];
```

```
s = t;
```

```
delete[] t;  
delete[] s;
```

--> double délétion



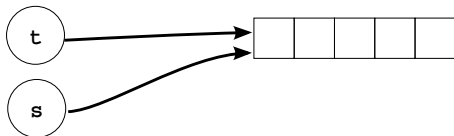
L'égalité de pointeurs est autorisée.

## Attention

- ▶ Il y a des risques de fuite de mémoire
- ▶ Deux pointeurs égaux renvoient au même espace mémoire
- ▶ Il n'y a pas création d'un nouveau tableau

```
double* t,s;  
int n=5;  
t = new double[n];  
  
s = t;  
  
delete[] t; // ou delete[] s
```

--> OK



Il est possible d'utiliser des tableaux dynamiques dans les structures.

## Attention

Surtout pas de tableaux statiques.

```
struct Vect{
    int taille; // la taille
    double* t; // le tableau, ne pas allouer
                // dans la declaration de la structure
};
```

L'instruction **break** permet de sortir d'une boucle.

```
for(int i=0; i<n; i++){
    bool b = f(i);
    if(!b) break; // sort de la boucle si b est faux
}
```

L'instruction **continue** permet de passer à l'itération suivante dans une boucle (sans exécuter ce qui se trouve après le **continue**).

```
int i=1;
while(i< 1000){
    i++;
    if(i%2 == 1)
        continue;
    cout << i << " est pair" << endl;
}
```



Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

TP

Jusqu'à présent :

- ▶ Factoriser du code : **fonctions, fichiers**
- ▶ Regrouper des données cohérentes : **tableaux, structures**

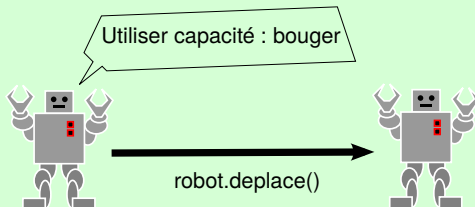
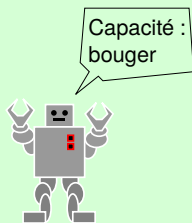
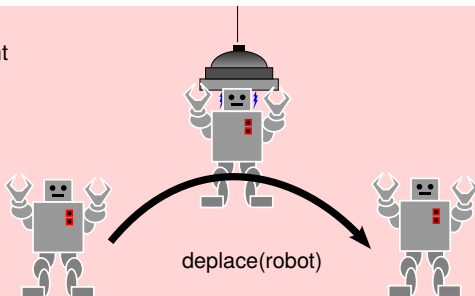
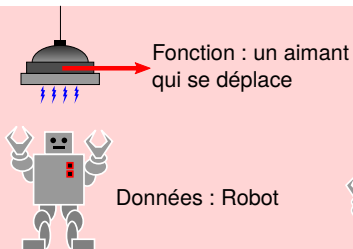
Les fonctions sont des opérations qui transforment des valeurs.

## Les objets

**OBJET = STRUCTURE + MÉTHODES (fonctions)**

**Idée** : les objets ont des fonctionnalités.

# Les objets



## Attention

Il ne faut pas voir des objets partout :

- ▶ Les données et les fonctions ne sont pas toujours liées.
- ▶ Il faut bien penser à l'organisation des données.
- ▶ Les fonctions sont souvent plus adaptées lorsqu'elles concernent plusieurs objets.

```
Obj1 a;  
Obj2 b;  
int i = f(a,b) // fonction f sur a et b
```

```
Obj1 a;  
Obj2 b;  
int i = a.f(b); // méthode f de a appliquée a b  
// ou b.f(a) ?
```

# Exemple d'objet

```
// Structure + fonctions
struct Obj1{
    int x;
};
int f(Obj1 &x);
int g(Obj1 &x, int y);

...
Obj1 a;
cout << f(a) << endl;
int i = g(a,10);
```

```
// Objet
struct Obj1{
    int x;
    int f();
    int g(int y);
};

...
Obj1 a;
cout << a.f() << endl;
int i = a.g(10);
```

On met simplement les déclarations **dans** la structure. **On ne met plus en argument** l'objet en question.

## Exemple d'objet

Dans la définition de la structure précédente, on **déclare** les méthodes, on ne les **définit** pas. Pour définir les méthodes, on utilise les **::**

```
// OBJET
struct Obj1{
    int x;
    int f();
    int g(int y);
};

// SOURCE Obj1.cpp
int Obj1::f(){
    ...
}
int Obj1::g(int y){
    ...
}

int main(){
    Obj1 a;
    // initialisation
    Obj1 b = {5};

    a.x = 2;

    cout << a.f() << endl;
    cout << b.g(a.f()) << endl;

    ...
}
```

## Fichiers d'en-tête (.h)

Ils reçoivent les **déclarations des structures**.

Ex : `struct Obj1{...};` dans `obj1.h`

## Fichiers sources (.cpp)

On y place les **déclarations des méthodes**.

Ex : `int Obj1::f(){...}` dans `obj1.cpp`

Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

TP



Les espaces de nom (**namespace**) définissent un conteneur pour les fonctions ou des objets.

Nous en avons déjà rencontré deux : **std** et **Imagine**.

Pour se placer à l'intérieur du namespace on utilise **using namespace xxx**; . De l'extérieur on utilise **::**

```
// De l'intérieur du namespace
#include <iostream>
using namespace std;
```

```
#include <Imagine/Graphics.h>
using namespace Imagine;
```

```
...
cout << i << endl;
click();
...
```

```
// De l'extérieur du namespace
#include <iostream>
```

```
#include <Imagine/Graphics.h>
```

```
...
std::cout << i << std::endl;
Imagine::click();
...
```

## Et pour les objets ?

C'est le même principe : lorsqu'on est dans l'objet, on a accès à ses champs et à ses méthodes.

```
struct Obj1{
    int x;
    void double_x();
    int renvoie_4x();
};

int main(){
    // En dehors de l'objet, on
    // utilise . pour accéder
    // aux champs et méthodes
    Obj1 a;
    a.x = 3;
    cout << a.renvoie_4x() << endl;
}
```

```
void Obj1::double_x(){
    // À l'intérieur du namespace
    // Obj1, on peut modifier ses
    // ↪ champs
    x = 2*x;
}

int Obj1::renvoie_4x(){
    // À l'intérieur du namespace
    // Obj1, on peut utiliser ses
    // ↪ méthodes
    double_x();
    return x;
}
```

Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

TP

# Création d'un objet Matrice

```
// Matrice.h
struct Matrice{
    // Champs
    int m, n;
    double* t;

    // Méthodes
    void cree(int m_, int n_);
    void detruit();
    void get(int i, int j);
    void set(int i, int j,
        ↪ double x);
    void affiche();
};
```

Matrice **operator\***(Matrice A,  
↪ Matrice B);

```
// Matrice.cpp
void Matrice::cree(int m_, int n_){
    m = m_; n = n_;
    t = new double[m_*n_];
}
void Matrice::detruit()
{ delete[] t; }
double Matrice::get(int i, int j)
{ return t[i+j*m]; }
void Matrice::set(int i,int j,double
    ↪ x){
    t[i+j*m] = x;
}
void Matrice::affiche(){
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            cout << get(i,j) << " ";
            cout << endl;
        }
    }
}
```

# Création d'un objet Matrice

```
// Matrice.h
struct Matrice{
    // Champs
    int m, n;
    double* t;

    // Méthodes
    void cree(int m_, int n_);
    void detruit();
    void get(int i, int j);
    void set(int i, int j,
        ↪ double x);
    void affiche();
};
```

```
Matrice operator*(Matrice A,
    ↪ Matrice B);
```

```
Matrice operator*(Matrice A,Matrice
    ↪ B){
    assert(A.n == B.m);
    Matrice C;
    C.cree(A.m, B.n);
    for(int i=0; i<A.m; i++){
        for(int j=0; j<B.n; j++){
            double d=0;
            for(int k=0; k<A.n; k++){
                d+=
                ↪ A.get(i,k)*B.get(k,j);
            }
            C.set(i,j,d);
        }
    }
    return C;
}
```

# Création d'un objet Matrice

```
// Matrice.h
struct Matrice{
    // Champs
    int m, n;
    double* t;

    // Méthodes
    void cree(int m_, int n_);
    void detruit();
    void get(int i, int j);
    void set(int i, int j,
        ↪ double x);
    void affiche();
};
```

Matrice **operator\***(Matrice A,  
↪ Matrice B);

```
int main(){
    Matrice M1;
    M1.cree(2,3);
    for(int i=0; i<2; i++)
        for(int j=0; j<3; j++)
            M1.set(i,j,i+j);
    M1.affiche();
    Matrice M2;
    M2.cree(3,5);
    for(int i=0; i<3; i++)
        for(int j=0; j<5; j++)
            M1.set(i,j,i*j);
    M2.affiche();
    Matrice M3 = M1 * M2;
    M3.affiche();
    M1.detrui();
    M2.detrui();
    M3.detrui();
}
```

Il est possible de mettre les opérateurs dans les objets. Par convention l'opérateur méthode d'un objet **A** de type Obj1 :

**operatorOp(Obj2 B)**

définit l'opération **A Op B** (dans cet ordre)

## Attention

Pour définir **B Op A**, il faut définir l'opérateur dans l'objet de type Obj2.

```
// Matrice.h
struct Matrice{
    ...

    // Opérateurs
    Matrice operator+(Matrice
        ↪ B);
    Matrice operator*(double
        ↪ l);
};

// l * A se définit
// à l'extérieur
Matrice operator*(double l,
    ↪ Matrice A);
```

```
Matrice Matrice::operator+(Matrice B){
    Matrice C;
    C.cree(m,n);
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
            C.set(i,j,get(i,j) + B.get(i,j));
    return C;
}

Matrice Matrice::operator*(double l){
    Matrice C;
    C.cree(m,n);
    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
            C.set(i,j,l*get(i,j));
    return C;
}

Matrice operator*(double l, Matrice A){
    return A*l;
}
```



```
int main(){
    Matrice M1;
    M1.cree(2,3);
    for(int i=0; i<2; i++)
        for(int j=0; j<3; j++)
            M1.set(i,j,i+j);
    M1.affiche();
    Matrice M2;
    M2.cree(3,5);
    for(int i=0; i<3; i++)
        for(int j=0; j<5; j++)
            M1.set(i,j,i*j);
    M2.affiche();
    Matrice M3 = M1 * M2;
    M3.affiche();
    M1.detrui();
    M2.detrui();
    M3.detrui();
}
```

Si on regarde attentivement, le développeur n'a plus qu'à utiliser les méthodes :

```
struct Matrice{
    void cree(int m1,int n1);
    void detruit();
    double get(int i,int j);
    void set(int i,int j,double
        ↪ x);
    void affiche();
};
```

C'est l'interface de l'objet Matrice.

## Facilité d'utilisation

L'utilisateur n'a besoin de connaître que l'interface pour utiliser l'objet Matrice. Les interfaces permettent de séparer l'**utilisation** de l'objet de sa **conception**.

## Abstraction

Une fois l'interface créée, le concepteur peut modifier l'organisation interne de l'objet (par exemple, changer les champs sans modification apparente pour l'utilisateur).

Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

TP

Rien n'empêche le développeur de manipuler directement les champs des objets que nous avons créé, y compris pour faire des opérations incohérentes.

```
Matrice A;  
A.cree(5,7);  
A.t[10] = 1000;  
A.m = 50; // il va y avoir des problèmes
```

En outre, si le concepteur de l'objet **Matrice** change le champ **t** en un champ **tab**, le programme de l'utilisateur ne fonctionne plus.

## Protection

→ Il faut empêcher l'utilisateur d'accéder à l'organisation interne de l'objet en utilisant un mécanisme de **protection**.

## Propriétés privées

Nous allons rendre **privées** certaines propriétés (méthodes ou champs) de l'objet. Elles ne seront alors plus accessibles de l'extérieur, seulement de l'intérieur de l'objet.

## Mécanisme

- ▶ On remplace **struct** par **class**,
- ▶ On utilise les mots clés **private:** et **public:** pour définir les zones privées et publiques.
- ▶ Par défaut, toute propriété d'une classe est privée.

```
// matrice.h
class Matrice{
    // prive par défaut
    int m,n;

public: //public a partir d'ici

    //methodes
    void cree(int m1,int n1);
    void detruit();
    double get(int i,int j);
    void set(int i,int j,double
        ↪ x);
    void affiche();

private: //prive a partir d'ici
    double* t;

};
```

Cela ne change rien aux définition des méthodes.

L'utilisateur n'a plus accès aux champs **m**, **n** et **t**.

Il est possible de mettre des méthodes dans les zones privées.

En C++, une structure est une classe dont toutes les propriétés sont publiques.

## Définition

Les **accesseurs** sont des méthodes publiques permettant de lire ou d'écrire dans les champs privés d'un objet.

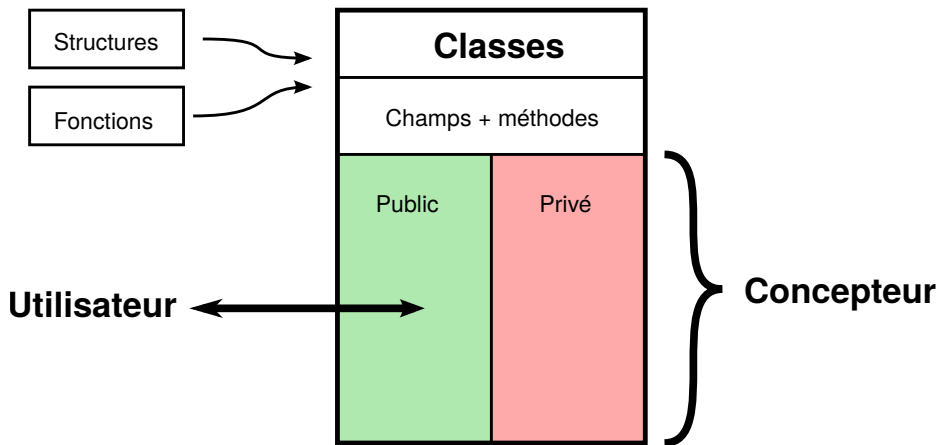
```
double get(int i, int j);  
void set(int i, int j, double x);
```

Maintenant que `m` et `n` sont aussi privés il faut aussi définir des accesseurs en lecture pour ces champs (pas en écriture).

```
int get_m();  
int get_n();
```

et les placer dans la partie publique de la classe **Matrice**.





Rappels

Programmation orientée objet

Espaces de noms et visibilité

Exemple d'objet : implémentation de matrices

Classes et protection des champs

TP

## Fractales

Dessiner des motifs fractales célèbres.

- ▶ Objets
- ▶ Fonctions récursives

