

# POO sous C++

## Construction & Destruction d'Objets

Med. AMNAI

Filière SMI - S5

Département d'Informatique

# Plan

## ① Construction, Destruction

# Plan

## ① Construction, Destruction

### ① Objets automatiques et statiques

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets
  - ⑤ Objets d'objets

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets
  - ⑤ Objets d'objets
- ② Initialisation d'un objet lors de sa déclaration



# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets
  - ⑤ Objets d'objets
- ② Initialisation d'un objet lors de sa déclaration
- ③ Propriétés des fonctions membres

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets
  - ⑤ Objets d'objets
- ② Initialisation d'un objet lors de sa déclaration
- ③ Propriétés des fonctions membres
- ④ Les fonctions amies

# Plan

- ① Construction, Destruction
  - ① Objets automatiques et statiques
  - ② Objets temporaires
  - ③ Objets dynamiques
  - ④ Tableaux d'objets
  - ⑤ Objets d'objets
- ② Initialisation d'un objet lors de sa déclaration
- ③ Propriétés des fonctions membres
- ④ Les fonctions amies
- ⑤ Surdéfinition des opérateurs

## Durée de vie d'allocation mémoire

- ① Les objets **automatiques** sont **créés** au moment de l'exécution d'une déclaration :
  - Dans une fonction.
  - Dans un bloc.
  - Ils sont **détruits** lorsqu'on **sort** de la fonction ou du bloc.
- ② Les objets **statiques** créés par une déclaration :
  - En dehors de toute fonction.
  - Dans une fonction ou dans un bloc, mais précédée par '**static**'.
  - Ils sont **créés avant** l'entrée dans la fonction **main** et **détruits** après la fin de son exécution.

# Appel des constructeurs et des destructeurs

```
class point
{
    int x,y;
    public :
    point(int a,int b){
        x=a;
        y=b;
    }
    ...
};
main(){
    point p1(2, 7) ; // est une déclaration correcte.
    point p2 ; point p3(17) ; // sont des déclarations incorrectes.
}
```

## RQ

- Le constructeur est appelé **après la création** d'un objet.
- Le destructeur est appelé **avant la destruction** d'un objet.

# Objets temporaires

Lorsqu'une classe dispose d'un constructeur, ce dernier peut être appelé explicitement ; dans ce cas, il y a alors création d'un objet temporaire.

```
class point
{
    int x,y;
    public :
        point(int a,int b){
            x=a;
            y=b;
        }
    ...
};
```

## Objets temporaires (suite)

- Supposons que 'P' est un objet de la classe '**point**' avec les paramètres (1,1) :  
**point p(1,1);**
- On peut écrire une affectation telle que :  
**p=point(2,7);**
- L'évaluation de l'expression **point(2,7);** entraîne :
  - La **création d'un objet temporaire** de type '**point**' (qui n'a pas de nom).
  - L'**appel du constructeur 'point'**, pour cet objet temporaire, avec transmission des arguments spécifiés.
  - La **recopie** de cet objet temporaire dans l'objet **p**.

# Exercice

Ecrire un programme permettant de créer *un objet automatique* dans la fonction '**main**', *deux autres objets temporaires* affectés à cet objet tout en **affichant** le moment de leur **création** et leurs **adresses**.

```
.....  
.....  
main()  
{  
    point p(1,1);  
    p=point(5,2);  
    p.affiche();  
    p=point(7,3);  
    p.affiche();  
}
```



## Objets dynamiques : Exemple

Créés par l'opérateur **new**, auquel on doit fournir, le cas échéant, les valeurs des arguments destinés à un constructeur.

```
main()
{
    point *p;
    p=new point(7,2);

    p->affiche();

    //ou

    (*p).affiche();

    p=new point(8,4);

    p->affiche();

    delete p;
}
```

Les objets dynamiques n'ont pas de durée de vie définie a priori. Ils sont détruits en utilisant l'opérateur **delete**.

# Tableaux d'objets

Un tableau d'objet est déclaré sous la forme :

```
point courbe[3];
```

Crée un tableau *courbe* de 3 objets de type *point* en appelant le constructeur pour chacun d'eux.

```
point * adcourbe = new point [3];
```

Alloue l'emplacement mémoire nécessaire à 3 objets (consécutifs) de type *point*, en appelant le constructeur pour chacun d'eux, puis place l'adresse du premier dans *adcourbe*.

Pour détruire le tableau précédent, on écrira :

```
delete [] adcourbe;
```

# Objets d'objets

Une classe peut contenir des membres de données de type quelconque y compris le type `'class'`.

## Exemple

Ecrire un programme permettant de définir une classe appelée `'point_coul'` à partir de la classe `'point'` définie précédemment et ayant comme données :  $x$ ,  $y$  et *couleur*.

# Objets d'objets (Sol Exemple)

```
class point{
    int x,y;
public :
    point (int a=0,int b=0){
        x=a; y=b;
        cout << " Construction du point : " << x << " - " << y << endl;
    }
    ~point(){
        cout<< " Destruction du point : " << x << " - " << y << endl;
    }
};

class point_coul{
    point p;
    int couleur;
public :
    point_coul(int,int,int);
    ~point_coul(){
        cout << "Destruction du point colore En couleur : " << couleur << endl;
        getch();
    }
};

point_coul::point_coul(int a,int b,int c):p(a,b){
    couleur=c;
    cout<< " Construction de point_coul en couleur : "<< couleur << endl;
}

main(){
    point_coul pc(3,7,8);
}
```

## Objets d'objets (suite)

- L'entête de *point\_coul* spécifie, après les deux points ( : ), la liste des arguments qui seront transmis au constructeur 'point'.
- Les constructeurs seront appelés dans l'**ordre suivant** 'point', 'point\_coul'.
- S'il existe des destructeurs, ils seront appelés dans l'**ordre inverse**.

# Initialisation d'un objet lors de sa déclaration

En plus d'une éventuelle initialisation par défaut (réservée aux variables statiques), une variable peut être initialisée explicitement lors de sa déclaration.

## Exemple

```
int n=2;  
int m=3*n-7;  
int t[3]={5,12,43};
```

## RQ

La partie suivant le signe '=' porte le nom d'**initialiseur**, il ne s'agit pas d'*un opérateur d'affectation*.

## Initialisation d'un objet (suite)

- C++ garantie l'appel d'un constructeur pour un objet créé par une déclaration sans initialisation (ou par '**new**').
- Mais, il est également possible d'associer un initialiseur à la déclaration d'un objet.

## Exemple

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

- **point p1(3);** est une déclaration ordinaire d'un objet '**p1**' aux coordonnées **3** et **0**.
- **point p2=7;** entraîne :
  - La création d'un objet appelé '**p2**'.
  - L'appel du constructeur auquel on transmet en argument la valeur de l'initialiseur '**7**'.
- En fin, les deux déclarations : '**point p1(3);**' et '**point p2=7;**' sont équivalentes.



## Constructeur par recopie

L'initialiseur d'un objet peut être d'un type quelconque, en particulier, il peut s'agir du type de l'objet lui-même.

### Exemple

```
point p1(33);
```

Il est possible de déclarer un nouvel objet 'p3' tel que :

```
point p3=p1 ; équivalente à : point p3(p1);
```

Cette situation est traitée par C++, selon qu'il existe un constructeur ou il n'existe pas de constructeur correspondant à ce cas.

## Cas 1 : Il n'existe pas de constructeur approprié

- Cela signifie que, dans la classe '**point**', il **n'existe pas de constructeur** à un seul argument de type '**point**'.
- Dans ce cas, C++ initialise '**p3**' avec les valeurs de l'objet '**p1**'. (Analogue à **recopie** des valeurs **ou affectation** entre objets de même type).
- Ce cas est le seul où C++ accepte qu'il n'existe pas de constructeur.
- Une déclaration telle que : '**point p(x);**' sera **rejetée** si '**x**' n'est pas de type '**point**'.

## Cas 2 : Il existe un constructeur approprié

- Cela signifie qu'il doit **exister un constructeur** de la forme : **'point (point &);'**.
- Dans ce cas, ce constructeur est appelé de manière habituelle, après la création de l'objet, sans aucune recopie.

RQ :

- C++ impose au constructeur en question que son unique **argument** soit **transmis par référence**.
- La forme **'point (point);'** serait rejetée par le compilateur.

## Exemple 1 : Traitement par défaut (excopie1.cpp)

```
class tableau{
    int ne;
    double *p;
public :
    tableau(int n){
        p=new double[ne=n];
        cout<< " Constructeur ordinaire : " << this << " avec son tableau : " << p << endl;
    }
    ~tableau(){
        delete p;
        cout<< " Destruction objet : " << this << " avec son tableau : " << p << endl;
    }
};

main(){
    tableau t1(3);
    tableau t2=t1; // equivalente a tableau t2(t1);
}
```

Constructeur ordinaire : 0x6ffe00 avec son tableau : 0x841a50  
Destruction objet : 0x6ffdf0 avec son tableau : 0x841a50

## Exemple 2 : Définition d'un constructeur par recopie

- On peut éviter le problème posé ci-dessus de telle façon à ce que la déclaration '**tableau t2=t1 ;**' conduise à créer un nouvel objet de type '**tableau**'
- avec non seulement ses membres données '**ne**' et '**p**' mais également son propre tableau dynamique.
- Pour ce faire, on définit un constructeur par recopie de la forme : **tableau (tableau &)** ;

## Exemple 2 : Définition d'un constructeur (excopie2.cpp)

```
class tableau{
    int ne;
    double *p;
public :
    tableau(int n){
        p=new double[ne=n];
        cout<<"Constructeur ordinaire : "<<this<<" avec son tableau dynamique : "<<p<<endl;
        for(int i=0;i<ne;i++)
            p[i]=(i+1)*10;
    }
    tableau(tableau & t){
        ne=t.ne; p=new double[ne];
        cout<<"Constructeur par recopie : "<<this<<" avec son tableau dynamique : "<<p<<endl;
        for(int i=0;i<ne;i++)
            p[i]=t.p[i];
    }
    ~tableau(){
        delete []p;
        cout<<"Destruction objet : "<<this<<" avec son tableau : dynamique "<<p<<endl;
    }
};

//-----
main(){
    tableau t1(3);
    tableau t2=t1; // équivalente à : tableau t2(t1);
}
```

# Modificateurs d'accès aux membres

```
class X {  
    private : //Mettre tous les membres privés ici  
        int a, b, c ;  
    public :  
        int d, e ; //Mettre tous les membres publiques ici  
  
    protected :  
        int f ; //Mettre tous les membres publiques ici  
};
```

## Autoréférence : 'this'

Le mot clé '**this**' utilisé uniquement au sein d'une fonction membre désigne un pointeur sur l'objet l'ayant appelé (auto.cpp).

```
class point
{
    int x, y;
public :
    point(int a=0,int b=0){
        x=a; y=b;
    }
    void affiche(){
        cout<< " Point : " << x << " - " << y << " de l'objet dont l'adresse est : "<< this << endl;
    }
};

main()
{
    point p1, p2(5,3), p3(6,70);
    p1.affiche();
    p2.affiche();
    p3.affiche();
}
```



# Encapsulation

Protéger les données par des accesseurs et des modificateurs.

```
class Box {  
  
    private:  
        double length;    // Length of a box  
        double breadth;   // Breadth of a box  
        double height;    // Height of a box  
  
    public :  
        double getLength() {return length;};  
        double getBreadth() {return breadth;};  
        double getHeight() {return height;};  
        void setLength(double length) {this->length = length;};  
        void setBreadth(double breadth) {this->breadth=breadth;};  
        void setHeight(double height) {this->height = height;};  
};
```

## Surdéfinition des fonctions membres

La surdéfinition des fonctions s'applique également aux fonctions membres d'une classe, y compris au constructeur (surdFctCls.cpp).

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

point::point() { //Surdéfinition des fonctions membres 'point'
    x=0; y=0;
}

point::point(int a){
    x=y=a;
}

point::point(int a,int b){
    x=a; y=b;
}

void point::affiche() { //Surdéfinition des fonctions membres 'affiche'
    cout<< " on est a : " <<x<< " - "<<y<<endl;
}

void point::affiche(char *t){
    cout<<t;
    affiche();
}

main(){
    point p1; p1.affiche();
    point p2(7); p2.affiche();
    point p3(44,52); p3.affiche();
}
```

```
on est a : 0 - 0
on est a : 7 - 7
on est a : 44 - 52
```

# Fonctions membres en ligne

Pour rendre 'en ligne' une fonction membre, on l'a défini dans la déclaration de la classe même (au lieu de la déclarer dans la classe et de la définir ailleurs). Le mot clé 'inline' n'est plus utilisé.

```
class point
{
    int x, y;
    public :
        point(){ x=0; y=0; }
        point(int a){ x=y=a; }
        point(int a,int b){ x=a; y=b; }
        void affiche(char *t);
};

void point::affiche(char *t)
{
    cout<<t<< " on est a : "<<x<< " - " <<y<<endl;
}

main()
{
```

```
class point
{
    int x, y;
    public :
        point();
        point(int);
        point(int,int);
        void point::affiche(char *t){ // Fonction membre en ligne
            cout<< t <<" on est a : " <<x<< " - "<<y<<endl;
        }
};

main(){
    .....;
    .....;
}
```

## Objets transmis en argument

Une fonction membre peut recevoir **un** ou **plusieurs** arguments du type de sa **classe**.

```
class point // ici mode par valeur (trsObjet.cpp)
{
    int x, y;
    public :
        point(int a,int b){ x=a; y=b; }
        int coincide(point);
};
//-----
int point::coincide(point o)
{
    if(x==o.x && y==o.y) return 1;
    return 0;
}
//-----
main()
{
    point p1(5,7), p2(5,7), p3(6,6);
    cout<< " p1 et p2 : " <<p1.coincide(p2)<<endl;
    cout<< " p2 et p3 : " <<p3.coincide(p2)<<endl;
    cout<< " p1 et p3 : " <<p1.coincide(p3)<<endl;
}
```

```
p1 et p2 : 1
p2 et p3 : 0
p1 et p3 : 0
```

## Transmission d'objets par adresse

Reprendre le programme précédent en faisant un passage d'objets par adresse.

*//Dans l'exemple ci-dessous, Le mode de transmission utilisé, était par adresse.*

```
class point //trsObjetAdr.cpp
{
    int x, y;
public :
    point (int a=0,int b=0){ x=a; y=b; }
    int coincide(point *);
};
int point::coincide(point *o)
{
    if((x==o->x) && (y==o->y))
        return 1;
    return 0;
}
//-----
main()
{
    point p1, p2(57), p3(57,0);
    cout<<" p1 et p2 : " <<p1.coincide(&p2)<<endl;
    cout<<" p3 et p2 : " <<p3.coincide(&p2)<<endl;
    cout<<" p1 et p3 : " <<p1.coincide(&p3)<<endl;
}
```

p1 et p2 : 0  
p3 et p2 : 1  
p1 et p3 : 0

## Transmission d'objets par référence

L'emploi des références permet de mettre en place une transmission par adresse, sans avoir à prendre en charge soi même la gestion.

```
class point{ //parRef.cpp
    int x,y ;
public :
    point(int a=0,int b=0){ x=a; y=b; }
    int coincide(point &);
};

int point::coincide(point & o)
{
    return (x==o.x && y==o.y) ? 1 : 0;
}

main()
{
    point p1, p2(57), p3(57,0);
    cout<< " p1 et p2 : " <<p1.coincide(p2)<<endl;
    cout<< " p2 et p3 : " <<p3.coincide(p2)<<endl;
    cout<< " p1 et p3 : " <<p1.coincide(p3)<<endl;
}
```

```
p1 et p2 : 0
p2 et p3 : 1
p1 et p3 : 0
```

# Principe

- L'**encapsulation interdit** à une fonction membre d'une classe ou toute fonction d'**accéder à des données privées** d'une autre classe.
- Grâce à la notion d'**amitié entre fonction et classe**, il est possible, lors de la définition d'une classe d'y déclarer une ou plusieurs fonctions (*extérieurs de la classe*) **amies** de cette classe.
- Une déclaration d'**amitié autorise l'accès aux données privées**, au même titre que que les fonctions membres.

## Cas d'amitiés

Il existe plusieurs situations d'amitiés :

- 1 Fonction **indépendante**, amie d'une **classe**.
- 2 Fonction **membre** d'une classe, amie d'une autre **classe**.
- 3 **Fonction** amie de **plusieurs** classes.
- 4 Toutes les fonctions **membres** d'une classe, amies d'une autre **classe**.



## Fonction indépendante amie d'une classe

**Déclarer** une fonction amie d'une classe, il suffit de la *déclarer dans cette dernière* en la *précédent* par le mot clé '**friend**'.

- L'**emplacement** de la déclaration d'amitié dans la classe est **quelconque**.
- Généralement, une fonction amie d'une classe possédera **1** ou **plusieurs** arguments.
- Peut avoir une **valeur de retour du type** de cette **classe**.

## Exemple (frFcCl.cpp)

```
class point{
    int x,y;
    public :
        point(int a=0,int b=0) {x=a; y=b;}
        friend int coincide(point,p2); //Fonction ami de la classe point
};

int coincide(point p1,point p2){
    if(p1.x==p2.x && p1.y==p2.y)
        return 1;
    else return 0;
}

main(){
    point o1(15,2), o2(15,2), o3(13,25);
    if(coincide(o1,o2))
        cout<< " les objets o1, o2 coïncident\n";
    else cout<< " les objets sont différents\n";
    if(coincide(o1,o3))
        cout<< " les objets o1, o3 coïncident\n";
    else cout<< " les objets sont différents\n";
}
```

les objets o1, o2 coïncident  
les objets sont différents

## Fonction membre d'une classe, amie d'une autre

```
class B; // Définir 'B' avant 'A'.

class A{ // Déclarer 'A' avant 'B'.
    .....
public :
    friend int B::f(int,A); // La fonction membre 'f' de la classe 'B'
                           // peut accéder aux membres privées de
                           // n'importe quel objet de la classe 'A'.
};

class B{
    .....
public:
    int f(int,A);
};

int B::f(int,A){
    .....
}
```

## Fonction amie de plusieurs classes

Toute fonction *membre* ou *indépendante*, peut être *amie* de *plusieurs classes*.

```
class A{
    .....
public :
    friend void f(A,B);
};

class B{
    .....
public :
    friend void f(A,B);
};

void f(A,B){
    .....
}
```

# Toutes les fonctions d'une classe amies d'une autre

Au lieu de faire autant de déclarations de fonctions amies qu'il y a de fonctions membres, on peut résumer toutes ces déclarations en une seule.

## Exemple

'**friend class B ;**' déclarée dans la classe '**A**' signifie que toutes les fonctions membres de la classe '**B**' sont amies de la classe '**A**'.

## RQ

Pour compiler la déclaration de la classe '**A**', il suffit de la faire précéder de : '**class B ;**'

## Exercice

Ecrire un programme permettant de réaliser le produit d'une matrice par un vecteur à l'aide d'une fonction **indépendante** appelée '**produit**' amie des deux classes '**matrice**' et '**vecteur**'.

La classe '**vecteur**' possède :

- comme **données** : un *vecteur de 3 éléments entiers*.
- comme **fonctions membres** :
  - Un *constructeur à 3 valeurs entiers*.
  - Une fonction '**affiche**' pour afficher le contenu du tableau (*vecteur*).

La classe '**matrice**' possède :

- comme **donnée** : une *matrice de 9 éléments (3x3)*.
- comme **fonction membre** : un *constructeur* ayant une matrice (3x3) comme paramètre.

La fonction '*produit*' **retourne** normalement un **objet** de type '**vecteur**' résultat du produit d'une matrice par un vecteur.

## Solution (exofrd.cpp)

```
class vecteur;  
class matrice{  
    int m[3][3];  
public :  
    matrice(int ma[3][3]){  
        for(int i=0;i<3;i++)  
            for(int j=0;j<3;j++)  
                m[i][j]=ma[i][j];  
    }  
    friend vecteur produit(matrice ma,vecteur ve);  
};  
class vecteur{  
    int v[3];  
public :  
    vecteur(int a=0,int b=0,int c=0){  
        v[0]=a;    v[1]=b;    v[2]=c;  
    }  
    void affiche() {  
        for(int i=0;i<3;i++)    cout<<v[i]<<"\t";  
    }  
    friend vecteur produit(matrice ma,vecteur ve);  
};  
vecteur produit(matrice ma,vecteur ve){  
    int i,j;  
    vecteur vect;  
    for(int i=0;i<3;i++)  
        for(int j=0;j<3;j++)  
            vect.v[i]+=ma.m[i][j]*ve.v[j];  
    return vect;  
}  
main(){  
    vecteur v1(1,2,3);  
    int mat[3][3]={1,2,3,4,5,6,7,8,9};  
    matrice m1(mat);  
    vecteur resultat;  
    resultat=produit(m1,v1);  
    resultat.affiche();  
}
```

14	32	50
----	----	----

# Introduction

C++ autorise la surdéfinition :

- Des fonctions **membres** ou **indépendantes** en fonction du nombre et du type d'arguments.
- Des **opérateurs** portant au moins sur un objet, tel que ;  
comme fonctions membres :
  - l'addition (+) ;
  - la soustraction (-) ;
  - l'affectation (=) entre objets.

Pour surdéfinir un opérateur '**op**', il faut définir une fonction de nom : '**operator op**'.

Exp : **Point operator + (point,point) ;**



## Surdéfinition d'opérateur avec une fonction amie

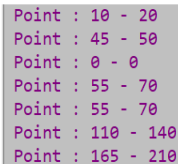
```
class point{ // surDifOpFcA.cpp
    int x,y;
public:
    point(int a=0,int b=0) { x=a; y=b; }
    void affiche(void){
        cout << " Point : " << x << " - " << y << endl;
    }
    friend point operator + (point,p2);
};
//-----
point operator + (point p1,point p2){
    point p;
    p.x=p1.x+p2.x;
    p.y=p1.y+p2.y;
    return p;
}
//-----
main(){
    point o1(10,20); o1.affiche();
    point o2(45,50); o2.affiche();
    point o3; o3.affiche();

    o3=o1+o2; o3.affiche();

    o3=operator+(o1,o2); o3.affiche();

    o3=o1+o2+o3; o3.affiche();

    o3=operator+(operator+(o1,o2),o3); o3.affiche();
}
```



Point : 10 - 20  
Point : 45 - 50  
Point : 0 - 0  
Point : 55 - 70  
Point : 55 - 70  
Point : 110 - 140  
Point : 165 - 210

## Surdéfinition d'opérateur avec une fonction membre

- L'expression '**`o1+o2`**' sera interprétée par le compilateur comme l'expression '**`o1.operator+(o2)`**'.
- Le prototype de la fonction membre '**`operator+`**' sera donc : '**`point operator+(point)`**' .

## Surdéfinition d'opérateur avec une fonction membre

```
class point{ // surDifOpFcM.cpp
    int x,y;
public :
    point(int a=0,int b=0) {x=a; y=b;}
    void affiche(){
        cout<< " Point : " << x << " - " << y << endl;
    }
    point operator + (point);
};

//-----
point point::operator+(point p1){
    point p;
    p.x=x+p1.x;
    p.y=y+p1.y;
    return p;
}

//-----
main(){
    point o1(10,20); o1.affiche();
    point o2(40,50); o2.affiche();
    point o3; o3.affiche();

    o3=o1+o2; o3.affiche();

    o3=o3+o1+o2; o3.affiche();
}
```

```
Point : 10 - 20
Point : 40 - 50
Point : 0 - 0
Point : 50 - 70
Point : 100 - 140
```

## Tableau d'opérateurs

Tableau d'opérateurs surdéfinissables, classés par priorité décroissante :

Pluralité	Opérateur	Associativité
Binaire	() <sup>◇</sup> [] <sup>◇</sup> → <sup>◇</sup>	→
Unaire	+ - ++ -- ! & new <sup>◇</sup> delete <sup>◇</sup>	←
Binaire	* / %	→
Binaire	+ -	→
Binaire	<< >>	→
Binaire	< <= > >=	→
Binaire	== !=	→
Binaire	& (niveau bit)	→
Binaire	^ (ou exclusif)	→
Binaire		→
Binaire	&&	→
Binaire	(niveau bit)	→
Binaire	= <sup>◇</sup> += -= *= /= %= &= ^=  = <<= >>=	←
Binaire	,	→

◇ : opérateur devant être surdéfini en tant que fonction membre.

## Choix entre fonction membre et fonction amie

Si un opérateur doit absolument recevoir un type de base en premier argument, il ne peut pas être défini comme fonction membre (laquelle reçoit implicitement un premier argument du type de sa classe).

## Surdéfinition de l'opérateur '[' ]'

Surdéfinir l'opérateur '[' ]' de manière à ce que '**o[i]**' désigne l'élément du tableau dynamique d'emplacement 'i' de l'objet '**o**' de la class '**tableau**'. Le premier opérande de '**o[i]**' étant '**o**'.

## Surdéfinition de l'opérateur '[' ]'

```
class tableau{ //surDifEx1.cpp
    int ne;
    int *p;
public :
    tableau(int n){
        p=new int[ne=n];
        for(int i=0;i<ne;i++)
            p[i]=(i+1)*10;
    }
    void affiche(){
        cout<<endl;
        for(int i=0;i<ne;i++)
            cout<<p[i]<<"\t";
        cout<<endl;
    }
    int operator[](int n){
        return p[n];
    }
    ~tableau(){
        delete []p;
    }
};

//-----
main(){
    tableau t1(3);
    t1.affiche();
    cout<<t1[2];
    t1.affiche();
}
```

10	20	30
30		
10	20	30

## Surdéfinition de l'opérateur '[' ]' (Discussion)

- La seule précaution à prendre consiste à faire en sorte que cette notation puisse être utilisée non seulement dans **une expression**, mais également à **gauche d'une affectation**.
- Il est donc nécessaire que la **valeur de retour** fournie par l'opérateur '[' ]' soit **transmise par référence** ;
- L'opérateur '[' ]' n'est pas imposé ici par C++, on aurait pu le remplacer par l'opérateur '()' : '**o(i)**' au lieu de : '**o[i]**', ou un autre opérateur.



## Surdéfinition de l'opérateur '[ ]'

```
class tableau{ //surDifExBis.cpp
    int ne;
    int *p;
public :
    tableau(int n){
        p=new int[ne=n];
        for(int i=0;i<ne;i++)
            p[i]=(i+1)*10;
    }
    void affiche(){
        for(int i=0;i<ne;i++)
            cout<<p[i]<<"\t";
        cout<<endl;
    }
    int & operator[](int n){
        return p[n];
    }
    ~tableau(){
        delete []p;
    }
};

main(){
    tableau t1(3);
    t1.affiche();
    cout<<t1[2];
    cout<<endl;
    t1[0]=55; // !!!
    cout<<t1[0]; cout<<endl;
    t1.affiche();
}
```

10	20	30
30		
55		
55	20	30

## Surdéfinition de l'opérateur '='

Reprenons le cas de la classe 'tableau' :

```
class tableau{  
    int ne;  
    int *p;  
    public :  
        .....  
};
```

## Surdéfinition de l'opérateur '='

Le problème de l'affectation est donc voisin de celui de la construction par recopie, mais non identique :

- Dans le cas de la construction par recopie (**tableau t1(3); tableau t2=t1;**), on a un seul tableau dynamique de 3 entiers pour les deux objets **t1** et **t2**.
- Dans le cas d'affectation d'objets, il existe deux objets complets (avec leurs tableaux dynamiques). Mais après affectation (**t2=t1**), **t2** et **t1** référencent le même tableau dynamique (celui de **t1**), le tableau dynamique de **t2** n'est plus référencé.

## Surdéfinition de l'opérateur '=' (suite)

- Ce problème peut être résolu en surdéfinissant l'opérateur d'affectation ; de manière à ce que chaque objet de type '**tableau**' possède son propre emplacement dynamique.
- Dans ce cas, on est sûr qu'il n'est référencé qu'une seule fois, et son éventuel libération peut se faire sans problèmes.

## Surdéfinition de l'opérateur '=' (Discussion)

Une affectation **t2=t1** ; pourrait être traitée de la façon suivante :

- **Libération** de l'**emplacement** pointé par le pointeur **p** de **t2**.
- **Création dynamique** d'un nouvel **emplacement** dans lequel on recopie les valeurs de l'emplacement pointé par **t1**.
- **Mise en place des valeurs** des membres données de **t2**.
- Il faut décider de la **valeur de retour** fournie par l'**opérateur** d'affectation en fonction de l'utilisation que l'on souhaite faire (**void** ou autre).
- Dans l'affectation **t2=t1** ; **t2** est le premier opérande (ici **this** car l'opérateur '=' est une fonction membre) et **t1** devient le second opérande (ici **t**).

## Surdéfinition de l'opérateur '='

```
class tableau{ //surDifEx2.cpp
    int ne;
    int *p;
public :
    tableau(int n){
        p=new int[ne=n];
        for(int i=0;i<ne;i++)
            p[i]=(i+1)*10;
    }
    void affiche(){
        for(int i=0;i<ne;i++)
            cout<<p[i]<<"\t";
        cout<<endl;
    }
    tableau & operator=(tableau & t){
        delete []p;
        p=new int[ne=t.ne];
        for(int i=0;i<ne;i++)
            p[i]=t.p[i];
        return *this;
    }
    ~tableau(){
        delete []p;
    }
};
```

```
//-----
main(){
    tableau t1(3); t1.affiche();
    tableau t2(4); t2.affiche();
    tableau t3(5); t3.affiche();
    t1=t3; t1.affiche();
    t3.affiche();
    t1=t2=t3;
    t1.affiche();
}
```

10	20	30		
10	20	30	40	
10	20	30	40	50
10	20	30	40	50
10	20	30	40	50
10	20	30	40	50