

# Programmation C++: Synthèse du cours et du livre.

Par ALLEMAN Jean-Christophe

## Quelques prérequis :

J'ai à peu près suivis le plan du cours de programmation de Madame Buseyne en suivant les explications et les exemples présents dans le livre : Programmer en C++ de Claude Delannoy. Ne sachant pas le contenu complet de certains chapitres, je donne des références présentes dans ce livre.

Certains exemples nécessitent une petite correction. Si vous trouvez des erreurs, des mises à jours sont toujours possibles. Je ne certifie pas que tout ce qu'il faut savoir pour l'examen de C++ se trouve dans ce cours mais une bonne partie s'y trouve déjà.

Pour ce qui ne s'y trouverait pas, un bon petit livre et la réussite est possible !

## Plan du cours :

1. Les spécificités du C++
  - 1.1. Les fonctions
  - 1.2. L'allocation dynamique
2. Les classes et les objets
  - 2.1. Les constructeurs et destructeurs
  - 2.2. Le constructeur par copie
  - 2.3. Les données membres statiques
3. La surcharge des opérateurs
  - 3.1. Surcharge des opérateurs +, -, \*, /
  - 3.2. Surcharge de l'opérateur =
  - 3.3. Surcharge des opérateurs de flux
  - 3.4. Surcharge des opérateurs d'incrément et de décrémentation
  - 3.5. Surcharge des opérateurs arithmétiques relationnels
  - 3.6. Surcharge des opérateurs relationnels :
  - 3.7. Surcharge de l'opérateur d'indexation []
  - 3.8. Surcharge de l'opérateur fonctionnel () :
  - 3.9. Surcharge des opérateurs new et delete :
4. L'héritage
  - 4.1. L'héritage simple
  - 4.2. L'héritage multiple
5. Les fonctions amies
6. Les flux d'entrées/sorties
  - 6.1. La classe ostream
  - 6.2. La classe istream
  - 6.3. Connexion d'un flot à un fichier
  - 6.4. Possibilités de formatage en mémoire

### → La généricité

7. Les patrons de fonctions
  - 7.1. Les paramètres de type d'un patron
  - 7.2. La surcharge de patrons
8. Les patrons de classes
  - 8.1. Exemple de création et d'utilisation d'un patron de classe
  - 8.2. Les paramètres de type d'un patron de classes

- 9. Librairie standard du C++
  - 9.1. Notion d'itérateur
  - 9.2. Notion de conteneur
  - 9.3. Les conteneurs séquentiels**
  - 9.4. Le conteneur vector
  - 9.5. Le conteneur deque
  - 9.6. Le conteneur list
  - 9.7. Les adaptateurs de conteneur : queue, stack, priority\_que
  - 9.8. Les conteneurs associatifs**
    - 9.8.1. Le conteneur map
    - 9.8.2. Le conteneur multimap
    - 9.8.3. Le conteneur set
    - 9.8.4. Le conteneur multiset

## 1. Quelques spécificités du C++ :

### 1.1. Les fonctions :

- Les paramètres par défaut :

Les fonctions peuvent avoir 1nombre variable de paramètres. A ces paramètres peuvent être associés des valeurs par défauts.

Elles peuvent être appelées avec un nombre de paramètre inférieur à celui de la déclaration.

Exemple :

Déclaration du prototype de la fonction :

```
int fonction1(int param1=0,int param2=0,int param3=0 " ) ;
```

Définition de la fonction :

```
int fonction1(int param1, int param2, int param3) {  
    //code fonction1  
}
```

Exemples d'appel de la fonction :

```
int main () {  
    int res,a=5,b=4,c=3 ;  
    res=fonction1(a,b,c) ;  
    res=focntion1(a,b) ;  
    res=fonction1(a) ;  
    res=fonction1() ;  
}
```

Lors de la compilation, il n'y aura pas de message d'erreur dû à la fonction. Elle ne prendra pas en compte tous les paramètres que la fonction a dans son prototype.

- La surcharge :

C'est utiliser plusieurs fois le même nom pour une fonction dans un même programme. Pour se faire, il faut faire varier les types des arguments des fonctions.

## Exemples :

```
#include <iostream.h>
void sosie(int);
void sosie(double);

void main() {
    int n=5;
    double x=2.5;
    sosie(n);
    sosie(x);
}
void sosie(int a) {
    cout<<"Sosie numéro I : a = "<<a<<endl;
}
void sosie(double a) {
    cout<<"Sosie numéro II : a = "<<a<<endl;
}

void test (int i=0,double x=0);
void test (double y=0,int p=0);
int n ; double z ;

...
...
...
//exemple d'appel de fonction
test(n,z) ;
test(z,n) ;
test(n) ;
test(z) ;
test() ; //ambigu,erreur
```

Les types des paramètres sont différents. Le compilateur ne se trompe pas car dans une des fonctions « sosie » il y a un paramètre de type int et dans l'autre un double.

- La fonction en ligne :

Une fonction en ligne (inline) se définit et s'utilise comme une fonction ordinaire, à la seule différence qu'on fait précéder son en-tête de la spécification inline.

Exemple :

```
//définition d'une fonction en ligne
inline double norme(double vec[3]) {
    int i ; double s=0 ;
    for (i=0 ; i<3 ;i++) {
        s+=vec[i] * vec[i] ;
        return sqrt(s) ;
    }
}
```

La fonction calcule la norme d'un vecteur à trois composantes qu'on lui fournit en argument.

La présence du mot "inline" demande au compilateur de traiter la fonction norme différemment. Quand le compilateur incorporera des instructions correspondantes à chaque appel d'une fonction de ce type. Ce mécanisme fera gagner du temps à l'exécution mais il consommera une quantité de mémoire croissant avec le nombre d'appels.

## 1.2. L'allocation dynamique de mémoire :

Se fait grâce à deux nouveaux opérateurs : `new` et `delete`.

### a) Utilisation de `new` :

La déclaration : `int *ad ;`

Et l'instruction : `ad = new int ;`

Permettent d'allouer l'espace mémoire nécessaire pour un élément de type `int` et d'affecter à `ad` l'adresse correspondante. En C, cela se fait avec « `malloc` ».

La déclaration : `char *adc`

Et l'instruction : `adc = new char[100] ;`

Permettent d'allouer l'emplacement nécessaire pour un tableau de 100 caractères et place l'adresse (de début) dans `adc`.

- La syntaxe de l'opérateur `new` se fait comme suit : `new type`, où `type` représente un type absolument quelconque. Il fournit comme résultat un pointeur (de type `type *`) sur l'emplacement correspondant quand l'allocation est réussie.
- Cet opérateur accepte aussi une déclaration sous la forme : `new type [n]`, où `n` désigne une expression entière quelconque. Cette instruction alloue alors l'emplacement nécessaire pour `n` éléments du type indiqué.

### b) L'utilisation de `delete` :

Lorsque l'on souhaite libérer un emplacement alloué préalablement par `new`, on doit absolument utiliser l'opérateur `delete`. Ainsi, pour libérer les emplacements créés dans les exemples de `new`, on écrit :

`delete ad ;`

Pour l'emplacement alloué par : `ad = new int ;`

Et : `delete adc ;`

Pour l'emplacement alloué par : `adc = new char [100] ;`

La syntaxe de l'opérateur `delete` est la suivante : `delete adresse`, où `adresse` est une expression devant avoir comme valeur un pointeur sur un emplacement alloué par `new`.

## 2. Les classes et les objets :

En C, nous avons vu les types structurés qui regroupaient plusieurs variables dans une seule et même structure. Les structures en C++ ont été remplacées par des classes.

Dans une classe, variables et fonctions peuvent s'y retrouver. On parlera de membres de classes ou de fonctions membres. Ces membres et/ou fonctions membres seront "publics", c'est-à-dire accessible "de l'extérieur", les autres membres seront "privés".

Exemple d'une déclaration de classe :

```
class point {
    private :
        int x ;
        int y ;
    public :
        void initialise (int,int) ;
        void deplace (int,int) ;
        void affiche() ;
};
```

Les membres nommés x et y sont privés, tandis que les fonctions membres nommées initialise, déplace et affiche sont publiques.

La définition des fonctions membres se fait exactement de la même manière que celle des fonctions membres d'une structure.

Utilisation d'une classe :

```
#include <iostream.h>
class point {
private:
    int x;
    int y;
public:
    void initialise(int,int);
    void deplace(int,int);
    void affiche();
};
void point::initialise(int abs,int ord) {
    x=abs ; y = ord;
}
void point::deplace(int dx,int dy) {
    x=x+dx; y=y+dy;
}
void affiche() {
    cout<<"Je suis en : "<<x<<y<<endl;
}
void main() {
    point a,b;
    a.initialise(5,2);a.affiche();
    a.deplace(-2,4);a.affiche();
    b.initialise(1,-1);b.affiche();
}
```

L'opérateur :: est l'opérateur de déclaration de portée. Il s'utilise comme suit :

```
Type nomclasse ::nom_méthode() {  
    ...  
}
```

Il est utilisé pour :

- définir une fonction comme étant globale ;
- définir et déclarer les classes ;
- accéder à une variable globale qui a le même nom qu'une donnée membre de la classe ;

## 2.1. Les Constructeurs et destructeurs :

- Les constructeurs.

C'est une fonction membre qui sera appelée automatiquement à chaque création d'objet. Il attribue des valeurs aux données d'un objet (comme la fonction initialise de l'exemple précédent).

Si la classe est dynamique, la déclaration se fera avec new.

Le constructeur se reconnaît car il a le même nom que la classe.

Quand une crée une classe qui contient un constructeur, il n'est plus possible de créer un objet sans fournir les arguments requis pas son constructeur.

- Les destructeurs.

C'est une fonction membre qui sera appelée automatiquement au moment de la destruction de l'objet. Pour les objets automatiques, la destruction a lieu quand on quitte le bloc ou la fonction où il a été déclaré.

Le destructeur se reconnaît porte lui aussi le même nom que la classe à la différence prêt qu'on lui ajoute un tidle ~.

Exemple :

```
#include <iostream.h>  
#include <stdlib.h> //pour la fonction rand  
  
class hasard {  
  
    int nbval;        //nombre de valeur  
    int *val;        //pointeur sur un tableau de valeurs  
public:  
    hasard (int,int); //constructeur  
    ~hasard();        //destructeur  
    void affiche();   //affiche le nombre de valeur;  
};
```

```

hasard::hasard(int nb,int max) {
    int i;
    val=new int [nbval=nb];
    for (i=0;i<nb;i++) {
        val[i]=double (rand())/RNAD_MAX * max;
    }
}
hasard::~~hasard () {
    delete val;
}
void hasard::affiche() {
    int i;
    for(i=0;i<nbval;i++) {
        cout<<val[i]<<endl;
    }
}
void main() {
    hasard suite1 (10,5);    //10 valeurs entre 0 et 5
    suite1.affiche();
    hasard suite2(6,12);    //6 valeurs entre 0 et 12
    suite2.affiche();
}

```

Le nombre de valeur est en argument dans le constructeur de la classe. Il est alors préférable que l'espace (variable) soit alloué dynamiquement au lieu d'être surdimensionné. Cette allocation dynamique se fait a l'intérieur même du constructeur.

Si le constructeur reçoit en argument la valeur maximale que peut prendre un nombre, il doit aussi recevoir le nombre de valeurs souhaitées.

Si un emplacement a été alloué dynamiquement, il faut se soucier de sa libération lorsqu'il sera devenu inutile. Pour cela, on a utilisé l'opérateur delete dans le destructeur.

Les destructeurs et constructeurs peuvent être privés ou publics. Mais il vaut mieux les rendre publics : si un constructeur d'une classe est privé, il ne sera plus possible de créer des objets de cette classe. Si un destructeur d'une classe est privé, cela ne portera pas a conséquence : il ne pourra juste pas être appelé directement.

## 2.2. Le constructeur par copie :

Un constructeur donne la certitude qu'un objet créé par une déclaration ou par un new ne pourra pas être créé que s'il a été placé dans un état initial convenable. Cependant, il existe des circonstances dans lesquelles il est nécessaire de construire un objet... La situation la plus fréquente est celle où la valeur d'un objet doit être transmise en argument à une fonction. Il faut alors (dans un emplacement local à la fonction) un objet qui soit une copie de l'argument effectif. On appelle cela une initialisation par recopie. On crée un objet par

recopie d'un objet déjà existant de même type. Pour réaliser cette initialisation, on utilisera un constructeur par recopie.

Un tel constructeur n'existe pas vraiment. Pour en créer un, il faut fournir explicitement dans la classe un constructeur par copie. Ce constructeur possèdera un seul argument du type de la classe et transmis obligatoirement par référence.

Par exemple avec la classe point :

```
point(point &);
```

Ce constructeur est alors appelé de manière habituelle après création d'un objet.

Exemple et comparaison entre constructeur par copie et constructeur :

```
#include <iostream.h>

class vect {
    int nelem;                //nombre d'élément
    double *adr;             //pointeur sur ces éléments
public :
    vect(int n);              //constructeur "usuel" (par défaut)
    vect(const vect & v);     //constructeur par copie
    ~vect();
};

vect::vect(int n) {
    adr=new double [nelem = n];
    cout<<" + constructeur - adr obj : "<<this<<"- adr vec : "<<adr<<endl;
}

vect::vect(const vect & v) {
    adr=new double [nelem=v.nelem];    //création d'un nouvel objet par allocation dynamique
    for(int i=0;i<nelem;i++) {
        adr[i]=v.adr[i];              //recopie de l'ancienne adresse
    }
    cout<<" +constructeur par copie - adr obj : "<<this<<"-adr vec: "<<adr<<endl;
}

vect::~~vect() {
    cout<<" -Destructeur - adr obj: "<<this<<"-adr vect: "<<adr<<endl;
}

void fct (vect b) {
    cout<<"Appel de fct"<<endl;
}

void main() {
    vect a(5);
    fct(a);
}
```

A l'affichage :

```
+constructeur – adr obj : 0x0013FF6C – adr vec : 0x00441EA0
+constructeur – adr obj : 0x0013FF10 – adr vec : 0x00441C10
Appel de fonction
-Destructeur – adr obj : 0x0013FF10 – adr vect : 0x00441C10
-Destructeur – adr obj : 0x0013FF6C – adr vect : 0x00441EA0
```

Chaque objet possédant son propre emplacement mémoire, les destructions se font sans problèmes.

### 2.3. Les données membres statiques :

Les données membres statiques sont des sortes de variables globales dont la portée est limitée à la classe.

Ils sont utiles pour, par exemple, compter le nombre d'objets créés.

- Déclaration :

```
class exemple {
    static int n;
    float x;
    ...
};
```

- L'initialisation :

Les données membres statiques n'existent qu'en un seul exemplaire et sont indépendants des objets de la classe.

La syntaxe :

```
class exemple {
    static int n=2;
    ...
};
```

Est une erreur. Dans ce cas-ci le membre statique risquerait de se voir réserver différents emplacements dans différents modules objets. Ce membre doit être initialisé à l'extérieur de la déclaration de la classe :

```
int exemple ::n=5; // que ce soit pour les membres statiques privés que publics
```

Un membre statique n'est pas initialisé par défaut à 0.

- Exemple :

On souhaite connaître le nombre d'objets existants. On ajoute 1 quand un objet est créé (constructeur) et on enlève 1 quand un objet est détruit (destructeur).

```
#include <iostream.h>
```

```
class cpte_obj {
    static int ctr; //compteur du nombre d'objets créés
public:
    cpte_obj();
    ~cpte_obj();
};
```

```
int cpte_obj::ctr=0; //initialisation de la donnée membre statique
cpte_obj::cpte_obj() {
    cout<<"Constructeur - il y a maintenant "<<ctr<<"objets"<<endl;
}
```

```

cpte_obj::~cpte_obj() {
    cout<<"Destructeur - il y a maintenant "<<--ctr<<"objets"<<endl;
}
void main() {
    void fct();
    cpte_obj a;
    fct();
    cpte_obj b;
}
void fct() {
    cpte_obj u,v;
}

```

#### A l'exécution :

```

++ construction : il y a maintenant      1objets
++ construction : il y a maintenant      2objets
++ construction : il y a maintenant      3objets
-- destruction : il y a maintenant       2objets
-- destruction : il y a maintenant       1objets
++ construction : il y a maintenant      2objets
-- destruction : il y a maintenant       1objets
-- destruction : il y a maintenant       0objets

```

### 3. La surcharge des opérateurs :

Il est possible (en C++) de surdéfinir un opérateur si d'une part, l'opérateur existe et d'autre part qu'il porte au moins sur un objet. On peut donner aux opérateurs un comportement spécifique si ils sont appliqués à des objets. Par exemple, la classe complexe destinée a représenter des nombres complexes, il sera possible de donner une signification à des expressions telles que :

$a+b$              $a-b$              $a*b$              $a/b$

a et b étant des objets de type complexe. Par exemple, ces expressions peuvent servir à additionner, soustraire, multiplier ou diviser des nombres complexes ou à n'importe quelle autre rôle que l'on veut leur attribuer dans la classe.

Pour parler de surcharge, il faut impérativement surcharger des opérateurs déjà existants. Il y a aussi des opérateurs qu'il est impossible de surcharger (sizeof, ::, .., ?). Un opérateur surdéfinit doit avoir au minimum un paramètre de type de la classe.

Il y a différentes façons de surcharger des opérateurs :

- la surcharge d'opérateur avec une fonction amie ;
- la surcharge d'opérateur avec une fonction membre ;

- Surcharge d'opérateur avec une fonction amie :

Par exemple, on veut surcharger l'opérateur +. Le prototype de notre fonction `operator +` sera :

```
point operator + (point, point) ;
```

Les deux arguments correspondront aux opérands de l'opérateur + lorsqu'il sera appliqué à des valeurs de type `point`.

Exemple de surcharge de l'opérateur + dans la classe `point` avec une fonction amie :

```
#include <iostream.h>

class point {
    int x,y;
public:
    point (int abs=0,int ord=0);
    point friend operator+ (point p1,point p2);
    void affiche();
};
point::point(int abs,int ord) {
    x=abs;
    y=ord;
}
point operator+(point p1,point p2) {
    point tmp;
    tmp.x=p1.x+p2.x;
    tmp.y=p1.y+p2.y;
    return tmp;
}
void point::affiche() {
    cout<<"Coordonnees : "<<x<<","<<y<<endl;
}
void main() {
    point pt1(1,6),pt2(5,-5),res;
    pt1.affiche();
    res=pt1+pt2;
    res.affiche();
}
```

A l'exécution :

Coordonnées : 1,6

Coordonnées : 6,1

Le résultat de  $(1+5)$ ,  $(6+(-5))$  donne bien 6,1

- Surcharge d'opérateur avec une fonction membre :

On reprend l'exemple de la classe `point` et on resurcharge l'opérateur +. Le prototype sera le suivant :

```
point operator+ (point) ;
```

L'opérande de l'opérateur de notre opérateur (qui correspond à l'argument) va être transmis implicitement. Une expression telle que `a+b` sera interprétée par le compilateur comme : `a.operator + (b)`

Les opérateurs `new`, `delete`, `=`, `()`, `[]` et `->` ne peuvent être surchargés que par des fonctions membres.

Exemple de surcharge de l'opérateur `+` dans la classe `point` avec une fonction membre :

A l'exécution : comme pour l'exemple précédant.

### 3.1. Surcharge des opérateurs `+`, `-`, `*`, `/` :

Comme vu dans les 2 exemples précédents. La surcharge de ces types d'opérateurs peut se faire des 2 façons vues précédemment.

Exemple :

```
#include <iostream.h>
```

```
class point {
    float x,y;
public:
    point (float abs=0,float ord=0);
    point operator+ (point p1);
    point operator- (point p1);
    point operator* (point p1);
    point operator/ (point p1);
    void affiche();
};
```

```
point::point(float abs,float ord) {
    x=abs;
    y=ord;
}
```

```
point point::operator+(point p1) {
    point tmp;
    tmp.x=x+p1.x;
    tmp.y=y+p1.y;
    return tmp;
}
```

```
point point::operator -(point p1) {
    point tmp;
    tmp.x=x-p1.x;
    tmp.y=y-p1.y;
    return tmp;
}
```

```

point point::operator *(point p1) {
    point tmp;
    tmp.x=x*p1.x;
    tmp.y=y*p1.y;
    return tmp;
}
point point::operator /(point p1) {
    point tmp;
    tmp.x=x/p1.x;
    tmp.y=y/p1.y;
    return tmp;
}
void point::affiche() {
    cout<<"Coordonnees : "<<x<<","<<y<<endl;
}
void main() {
    point pt1(1,6),pt2(5,-5),res;
    pt1.affiche();
    pt2.affiche();
    res=pt1+pt2;
    cout<<"Addition:"<<endl;
    res.affiche();
    res=pt1-pt2;
    cout<<"Soustraction:"<<endl;
    res.affiche();
    res=pt1*pt2;
    cout<<"Multiplication:"<<endl;
    res.affiche();
    res=pt1/pt2;
    cout<<"Division:"<<endl;
    res.affiche();
}

```

### 3.2. Surcharge de l'opérateur = :

Avant de d'expliquer ce comment surdéfinir l'opérateur =, il faut expliquer ce que fait le mot clé this.

Ce mot clé est utilisé lorsque il faut manipuler explicitement l'adresse de l'objet en question. Il ne peut être utilisé que dans des fonctions membres et il désigne l'objet l'ayant appelé.

```

#include <iostream.h>

class point {
    int x,y;
public:
    point(int abs=0,int ord=0) {
        x=abs;y=ord;
    }
    void affiche();
};

void point::affiche() {
    cout<<"Adresse : "<<this<<" - coordonnees "<<x<<","<<y<<endl;
}

```

```

void main() {
    point a(5),b(3,15);
    a.affiche();
    b.affiche();
}

```

L'opérateur = est un opérateur qui doit être surchargé par une fonction membre.

La syntaxe : vect & operator=(const vect &)

Syntaxe d'utilisation :

```

vect vect::operator(const vect & v) {
    if(this != &v) {
        delete adr;
        for(int i=0;i<nelem;i++) {
            adr[i]=v.adr[i];
        }
    }
    return *this
}

```

Comme l'argument de la fonction membre operator= est transmis par référence, il est nécessaire de lui associer le qualificatif const si l'on souhaite affecter un vecteur constant à un vecteur quelconque.

### 3.3. Surcharge des opérateurs de flux :

- L'opérateur << :

Cet opérateur se retrouve dans la classe ostream. << est surdéfini pour les différents types de base, sous la forme :

```
ostream &operator <<(expression); // pour la surdéfinition
```

- Ce qu'il reçoit :
  - la classe l'ayant appelé (argument implicite this) ;
  - une expression d'un type de base quelconque.

- Rôle : transmettre la valeur de l'expression au flot concerné en la formatant de façon appropriée. Considérons, par exemple, l'instruction :

```
cout<<n;
```

n contient la valeur 1234, le travail de l'opérateur << consiste à convertir la valeur de n dans le système décimal et à envoyer au flot cout les caractères correspondant à chacun des chiffres ainsi obtenus. L'opérateur a un rôle d' "écriture".

- Résultat : fournit la référence au flot concerné, après qu'il a écrit l'information voulue. Cela permet d'appliquer facilement plusieurs fois de suite, comme dans :

```
cout<<"valeur : "<<n<<endl;
```

Tous les types de base sont acceptés par l'opérateur << (aussi char, char\* et les pointeurs sur les types quelconques).

- L'opérateur >> :

Cet opérateur se retrouve dans la classe `istream`. >> est surdéfini pour tous les types de base, y compris `char*`, sous la forme :

```
istream &operator >>(expression) ;
```

- Ce qu'il reçoit : - la classe l'ayant appelé (argument implicite `this`) ;  
- une "lvalue" d'un type quelconque.

- Rôle : extrait du flot concerné les caractères nécessaires pour former une valeur du type de base voulu en réalisant une opération inverse du formatage opéré par l'opérateur <<.

- Résultat : fournit la référence au flot concerné, après qu'il en a extrait l'information voulue. Cela permet de l'appliquer plusieurs fois de suite, comme dans :

```
cin>>n>>p>>x ;
```

### 3.4. Surdéfinition des opérateurs d'incrément et de décrémentation :

- Distinction entre pré et post in/décrément :

- Pour l'incrément : ++

-> préincrément : `j = ++ i ;` //on ajoute 1 à i puis on donne la valeur de i à j

-> postincrément : `j = i ++ ;` //on donne la valeur de i à j puis on ajoute 1 à i

- Pour la décrémentation : --

-> prédécrément : `j = -- i ;` //on retire 1 à i puis on donne la valeur de i à j

-> postdégrément : `j = i -- ;` //on donne la valeur de i à j puis on retire 1 à i

- Surdéfinition dans les 2cas :

Pour différencier les 2opérateurs dans la surdéfinition, on change la valeur de la signature en fonction des cas :

- Pour une notation préfixée :

```
T operator++() ;
```

- Pour une notation postfixée :

```
T operator++(int) ;
```

La présence de l'opérande de type `int` est totalement fictif, il permet au compilateur de choisir l'opérateur à utiliser. Aucune valeur ne sera réellement transmise lors de l'appel.

On peut faire de même pour l'opérateur --.

Exemple de programme avec surdéfinition des opérateurs de flux, de préincrément et de postincrément :

```
#include <iostream.h>
#include <string.h>

class exemple {
    char chaine[20];
public :
    exemple(char tab[20]);
    friend istream& operator>>(istream& o,exemple& p);
    friend ostream& operator<<(ostream& o,exemple p);
    exemple& operator--(); //prédécrément
    exemple& operator--(int); //postdécrément
};

exemple::exemple(char tab[20]) {
    strcpy(chaine,tab);
}

istream& operator>>(istream& o,exemple& p) {
    cout<<"La valeur de la chaine : "<<endl;
    o>>p.chaine;
    return o;
}

ostream& operator<<(ostream& o,exemple p) {
    o<<p.chaine<<endl;
    return o;
}

exemple& exemple::operator--() {
    int k;
    k=strlen(chaine);
    for (int i=0;i<k;i++) {
        if(chaine[i]=='a') chaine[i]='z';
        else if(chaine[i]=='A') chaine[i]='Z';
        else if(chaine[i]==' ') chaine[i]=' ';
        else chaine[i]--;
    }
    return *this;
}

exemple& exemple::operator--(int) {
    int k;
    k=strlen(chaine);
    for (int i=0;i<k;i++) {
        if(chaine[i]=='a') chaine[i]='z';
        else if(chaine[i]=='A') chaine[i]='Z';
        else if(chaine[i]==' ') chaine[i]=' ';
        else --chaine[i];
    }
    return *this;
}
```

```

void main() {
    exemple q("");
    cin>>q;
    q--;
    cout<<q;
    --q;
    cout<<q;
}

```

### 3.5. Surdéfinition des opérateurs arithmétiques relationnels :

La surcharge des opérateurs +=, -=, \*=, /=,... est possible grâce à une fonction membre.

### 3.6. Surdéfinition des opérateurs relationnels :

La surcharge des opérateurs <, >, <=, >=, ==, != est possible grâce à des fonctions amies.

Exemple avec les complexes de la surcharge des opérateurs relationnels et des opérateurs arithmétiques relationnels :

```

#include <iostream.h>

void menu();
class complexe {
    int re,im;
public:
    complexe(int r=0,int i=0);
    void affiche();
    complexe& operator +=(complexe& c);
    complexe& operator -=(complexe& c);
    friend bool operator==(complexe c1,complexe c2);
    friend bool operator>(complexe c1,complexe c2);
};

complexe::complexe(int r,int i){
    re=r;
    im=i;
}

void complexe::affiche(){
    if(im>=0)
        cout<<re<<"+"<<im<<"i";
    else
        cout<<re<<im<<"i";
    cout<<endl;
}

complexe& complexe::operator+=(complexe& c) {
    c.re=re+c.re;
    c.im=im+c.im;
    return *this;
}

```

```

complexe& complexe::operator-=(complexe& c){
    c.re=re-c.re;
    c.im=im-c.im;
    return *this;
}
bool operator==(complexe c1,complexe c2){
    if (c1.re==c2.re && c1.im==c2.im) return true;
    else return false;
}

bool operator>(complexe c1,complexe c2) {
    if (c1.re>c2.re) return true;
    else return false;
}

void main() {
    complexe c1(2,3),c2(2,3),res;
    int choix;
    c1.affiche();
    c2.affiche();
    do {
        res=c2;
        menu();
        cout<<"Votre choix : ";
        cin>>choix;
        switch(choix) {
            case 1:
                c1+=res;
                cout<<"c1+c2=";
                res.affiche();
                break;
            case 2:
                c1-=res;
                cout<<"c1-c2=";
                res.affiche();
                break;
            case 3:
                if((c1==res)==true) {
                    cout<<"Complexes egaux"<<endl;
                }
                else {
                    if((c1>res)==true) {
                        cout<<"c1 plus grand que c2"<<endl;
                    }
                    else cout<<"c2 plus grand que c1"<<endl;
                }
                break;
        }
    }
    while(choix!=4);
}

void menu() {
    cout<<"1.Additionner des complexe"<<endl;
    cout<<"2.Soustraire des complexe"<<endl;
    cout<<"3.Comparer des complexe"<<endl;
    cout<<"4.Quitter"<<endl;
}

```

### 3.7. Surcharge de l'opérateur d'indexation [] :

```
Soit : class vect {
        int nelem;
        int *adr;
        ...
    }
```

On voudrait accéder à un élément de l'emplacement pointé par `adr` à partir de sa position, que l'on repérera par un entier compris entre 0 et `nelem-1`.

Pour cela, il serait intéressant de pouvoir surdéfinir l'opérateur `[]` pour que `a[i]` désigne l'élément d'emplacement `i` de `a`. Il faudrait que `a[i]` soit une lvalue (= left value = expression placée en membre à gauche et associable à un objet unique).

Pour que cela soit possible, il faudra donc que la valeur de retour fournie par l'opérateur `[]` soit transmise par référence. Le C++ impose que cet opérateur soit surdéfini par fonction membre :

```
int & operator [] (int);
```

Pour le renvoi, si on se contente de renvoyer l'élément cherché, la syntaxe peut se réduire à : `return adr[i];`

Exemple :

```
#include <iostream.h>

class tableau {
    int taille;
    int *tab;
public:
    tableau(int ta=100,int *t=NULL);
    int & operator[](int indice);
    ~tableau();
};

tableau::tableau(int ta,int *t) {
    taille=ta;
    tab=new int[taille];
    for(int i=0;i<taille;i++) {
        tab[i]=t[i];
    }
}

tableau::~tableau() {
    delete tab;
}

int & tableau::operator[](int indice) {
    return tab[indice];
}

void main() {
    int tab[4];
    int *tab1;
    for(int i=0;i<4;i++) {
        tab[i]=i*2;
    }
    for(i=0;i<4;i++) {
        cout<<tab[i]<<" ";
    }
}
```

### 3.8. Surcharge de l'opérateur fonctionnel () :

Si on surcharge l'opérateur (), on dit que les objets auxquels elle donne naissance sont des objets fonctions car ils peuvent être utilisés de la même manière qu'une fonction.

```
class cl_fct {  
public:  
    cl_fct(float x) {...}; //constructeur  
    int operator()(int n,int p) {...}; //opérateur ()  
};
```

Une déclaration telle que : `cl_fct obj_fct1(2.5);` construit un objet nommé `obj_fct1` de type `cl_fct`, en transmettant le paramètre 2.5 à son constructeur.

Utilité : utilisé quand il est nécessaire d'effectuer certaines opérations d'initialisation d'une fonction ou de paramétrer son travail, cette surdéfinition sera encore plus intéressante dans le cas des fonctions de rappel, c'est-à-dire transmise en argument à une autre fonction.

### 3.9. Surcharge des opérateurs new et delete :

- L'opérateur new :

Sa surcharge se fait obligatoirement par une fonction membre qui doit :

- recevoir un argument de type `size_t` défini dans le fichier en-tête `stddef.h` ;
- retourner une valeur de type `void` qui correspond à l'emplacement alloué pour l'objet.

- L'opérateur delete :

La définition de la fonction membre correspondant à l'opérateur delete doit :

- recevoir un argument du type pointeur sur la classe correspondante qui représente l'emplacement alloué détruit.
- ne fournir aucune valeur de retour.

## Exemple :

```
#include <iostream.h>

class point {
    static int npt;           //nombre total de points
    static int npt_dyn;      //nombre de points "dynamiques"
    int x,y;
public:
    point (int abs=0,int ord=0) {
        x=abs;
        y=ord;
        npt++;
        cout<<"++nombre total de points : "<<npt<<endl;
    }
    ~point() {
        npt--;
        cout<<"-- nombre total de points : "<<npt<<endl;
    }
    void *operator new(size_t sz);
    //void operator delete(void *dp);
};

int point::npt=0;
int point::npt_dyn=0;

void * point::operator new(size_t sz) {
    npt_dyn++;
    cout<<"Il y a "<<npt_dyn<<" points dynamiques sur un"<<endl;
    return::new char[sz];
}

/*void operator delete(void *dp) {
    //npt_dyn --;
    //cout<<"Il y a "<<npt_dyn<<" points dynamiques sur un"<<endl;
    delete (dp);
}*/

void main() {
    point *ad1, *ad2;
    point a(3,5);
    ad1=new point (1,3);
    point b;
    ad2=new point (2,0);
    //delete ad1;
    point c(2);
    //delete ad2;
}
```

## 4. L'héritage :

Permet de définir une nouvelle classe "dérivée" à partir d'une classe déjà existante que l'on appelle classe "de base".

La classe dérivée "héritera" des "potentialités" de la classe existante dite de "base"

On peut distinguer 2 types d'héritage : - l'héritage simple ;  
- l'héritage multiple ;

### 4.1. L'héritage simple :

- La notion d'héritage :

Soit : la classe point :

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

Supposons que nous voulions ajouter à nos points une couleur : création d'une nouvelle classe "pointcol". Un point coloré peut-être défini par ses coordonnées (comme un objet de type point), auxquelles on adjoint une information de couleur. On peut alors tenté de définir "pointcol" comme étant dérivée de "point". Pour l'instant, on attribue une fonction membre couleur destinée à donner une couleur à un point :

```
class pointcol:public point { //1.  
    short couleur;  
public:  
    void colore (short c1) {  
        couleur=c1;  
    }  
};
```

1. Spécifie que pointcol est une classe dérivée de la classe point.

**Le mot public signifie que les membres publics de la classe de base (point) seront des membres publics de la classe dérivée (pointcol).**

La déclaration d'objet de type pointcol ne change pas :

```
pointcol p,q ;
```

Chacun de ces objets peut faire appel :

- aux méthodes publiques de pointcol (colore) ;
- aux méthodes publiques de la classe point.

- Utilisation des membres de la classe de base dans une classe dérivée :

Pour l'instant, on sait que l'utilisation du mot public nous permettait d'accéder aux données membres publiques de point à partir de la classe pointcol.

Problème : si l'on appelle affiche() dans un main, on n'a aucune information sur la couleur du point.

La première solution, c'est d'ajouter une fonction membre affichec à la classe pointcol.

Si l'on définit affichec comme suit :

```
void affichec() {
    cout<<"Je suis en "<<x<<","<<y<<endl;
    cout<<" et ma couleur est : <<couleur<<endl;
}
```

"affichec" aurait alors accès aux données membres privées de point, ce qui est impossible : **Une classe dérivée n'a pas accès aux membres privés de sa classe de base !**

Par contre, les classes dérivées ont tous les droits pour accéder à des données membres publiques d'une classe de base.

Dans notre cas, "pointcol" peut tout a fait accéder à la fonction membre affiche() de "point" :

```
void affichec() {
    affiche(); //1.
    cout<<" et ma couleur est : "<<couleur<<endl;
}
```

1. on ne spécifie pas que la fonction membre fait partie de point !

On peut aussi réaliser la définition d'une nouvelle fonction d'initialisation de la manière suivante :

```
void pointcol::initialisec(int abs,int ord,short c1) {
    initialise(abs,ord);
    couleur = c1;
}
```

Exemple de programme avec un héritage complet de pointcol :

```
#include <iostream.h>
```

```
class point {
    int x;
    int y;
public:
    void initialise (int abs=0, int ord=0);
    void deplace (int ab=0, int or=0);
    void affiche();
};
```

```
class pointcol:public point {
    short couleur;
public:
    void colore (short c1) {
        couleur=c1;
    }
}
```

```

    void initialisec(int abs=0,int ord=0,short c1=0);
    void affichec() {
        affiche();          //1.
        cout<<" et ma couleur est : "<<couleur<<endl;
    }
};

void point::initialise (int abs,int ord) {
    x=abs;
    y=ord;
}

void point::deplace(int ab, int or) {
    x=x+ab;
    y=y+or;
}

void point::affiche() {
    cout<<"Coordonnees du point : "<<x<<" , "<<y<<endl;
}

void pointcol::initialisec(int abs,int ord,short c1) {
    initialise(abs,ord);
    couleur = c1;
}

void main() {
    pointcol p;
    p.initialisec(10,20,5);
    p.affiche();
    p.affichec();
    p.deplace(2,4);
    p.affichec();
    p.colore(2);
    p.affichec();
}

```

- Redéfinition des membres d'une classe dérivée :

Dans l'exemple ci-dessus, on retrouvait deux fonctions pour l'affichage :

- l'une pour l'affichage des coordonnées (affiche());
- l'autre pour l'affichage des coordonnées et de la couleur (affichec()).

L'idée est de pouvoir leur donner le même nom... Tout a fait possible (avec une petite précaution) : avec la fonction affiche de pointcol, il nous sera impossible d'appeler la fonction affiche de point comme on le faisait auparavant -> appel récursif de la fonction affiche de pointcol. La solution est d'utiliser l'opérateur de résolution de portée (::) pour localiser ma méthode voulue.

Si pour un objet p de type pointcol, on appelle la fonction p.affiche(), il s'agira de la fonction redéfinie par pointcol. Si l'on veut celle de point : il nous faut alors appeler comme ceci : p.point::affiche() ;

## Transformation de l'exemple précédent :

```
#include <iostream.h>

class point {
    int x;
    int y;
public:
    void initialise (int abs=0, int ord=0);
    void deplace (int ab=0, int or=0);
    void affiche();
};

class pointcol:public point {
    short couleur;
public:
    void colore (short c1) {
        couleur=c1;
    }
    void initialise(int abs=0,int ord=0,short c1=0);
    void affiche();
};

void point::initialise (int abs,int ord) {
    x=abs;
    y=ord;
}

void point::deplace(int ab, int or) {
    x=x+ab;
    y=y+or;
}

void point::affiche() {
    cout<<"Coordonnees du point : "<<x<<" , "<<y<<endl;
}

void pointcol::initialise(int abs,int ord,short c1) {
    point::initialise(abs,ord);
    couleur = c1;
}

void pointcol::affiche() {
    point::affiche();
    cout<<" et ma couleur est : "<<couleur<<endl;
}

void main() {
    pointcol p;
    p.initialise(10,20,5);
    p.affiche();
    p.point::affiche();
    p.deplace(2,4);
    p.affiche();
    p.colore(2);
    p.affiche();
}
```

- Au point de vue des constructeurs et destructeurs :

-> Hiérarchie des appels :

```
class A {
    ...
public :
    A(...);
    ~A();
    ...
};

class B:public A {
    ...
public:
    B(...);
    ~B();
    ...;
};
```

Pour créer un objet B, il faut d'abord créer un objet A, c'est-à-dire d'abord appeler le constructeur de A et ensuite celui de B. L'idem se produit pour les destructeurs.

➔ Transmission d'informations entre constructeurs :

Un problème apparaît quand le constructeur de A a besoin des arguments : quand on crée un objet de type B, les informations fournies sont destinés à son constructeur !

Pour se faire, il faut spécifier (dans la définition d'un constructeur d'une classe dérivée) les informations que l'on souhaite transmettre à un constructeur de la classe de base. Ce mécanisme revient au même que dans le cas des fonctions membres des classes "point" et "pointcol".

```
class point {
    ...
public :
    point(int, int);
    ...
};

class pointcol:public point {
    ...
public:
    pointcol(int,int,short);
    ...;
};
```

On souhaite que pointcol transmette les deux premières informations reçues, son en-tête ressemblera à ceci : pointcol(int abs,int ord,short c1) :point(abs,ord) ;

Ainsi la déclaration : pointcol a(10,15,3) ;

Entraînera l'appel de point (qui recevra 13 et 15) et l'appel de pointcol (qui recevra 10, 15 et 3).

Si on prend un nombre inférieure d'arguments dans l'appel du constructeur, il y aura erreur du fait qu'il n'existe pas de constructeur pointcol à 2arguments. Et bien entendu il est toujours possible de mentionner des arguments par défauts dans pointcol.

## Exemple :

```
#include <iostream.h>

class point {
    int x,y;
public:
    point(int abs=0,int ord =0) {
        cout<<"++ constructeur point : "<<abs<<" , "<<ord<<endl;
        x=abs;
        y=ord;
    }
    ~point() {
        cout<<"-- destruction d'un point : "<<x<<" , "<<y<<endl;
    }
};

class pointcol:public point {
    short couleur;
public:
    pointcol(int,int,short);
    ~pointcol() {
        cout<<"Destruction pointcol - couleur : "<<couleur<<endl;
    }
};

pointcol::pointcol(int abs=0,int ord=0,short c1=1):point(abs,ord) {
    cout<<"++ constructeur pointcol : "<<abs<<" , "<<ord<<" -> "<<c1<<endl;
    couleur = c1;
}

void main() {
    pointcol a(10,15,3);
    pointcol b(2,3);
    pointcol c(12);
    pointcol *adr;
    adr=new pointcol(12,25);
    delete adr;
}
```

Il faut bien veillé que la classe dérivée possède bien un constructeur par défaut. Si elle n'en possède pas, il y aura problème lors de la transmission des informations attendues par le constructeur de la classe de base.

➔ Contrôle des accès : (p256)

Dans ce point nous verront l'utilisation des données membres "protected". Lors de la conception d'une classe de base, les membres protégés se comportent comme des membres privés pour l'utilisateur de la classe mais comme des membres publics pour la classe dérivée elle-même. Lors de la conception de la classe dérivée, on peut restreindre les possibilités d'accès aux membres de la classe de base.

```

class point {
protected:
    int x,y;
public:
    point(...);
    void affiche();
    ...
};

class pointcol:public point {
    short couleur;
public:
    pointcol(...)
    void affiche() {
        cout<<"Je suis en "<<x<<" , "<<y<<endl;
        cout<<" et ma couleur est : "<<couleur<<endl; // (1)
    }
    ...
};

```

- (1) On a bien accès aux données membres protégées de la classe "point" à partir de la classe "pointcol".

➔ Compatibilité entre classe de base et classe dérivée :

Tout ce que l'on trouve dans une classe de base se trouve également dans la classe dérivée (toute action réalisable sur une classe de base peut toujours être réalisée sur une classe dérivée) : un point coloré peut toujours être considéré comme un point car on possède les coordonnées d'un point coloré.

Par contre la réciproque de cette proposition est fautive : on ne peut pas colorer un point ou s'intéresser à sa couleur.

En définitive, tout objet de "pointcol" est un "point", mais tout objet de type "point" n'est pas un "pointcol".

La compatibilité entre les classes ne s'applique que dans le cas de dérivation publique. Cette compatibilité se résume à l'existence de conversions implicites :

- d'un objet d'un type dérivé dans un objet d'un type de base ;
- d'un pointeur (ou d'une référence) sur une classe dérivée en un pointeur (ou une référence) sur une classe de base.

1<sup>ère</sup> possibilité : on convertit un type dérivé en un type de base.

On reprend nos 2 classes habituelles : "point" et "pointcol"

Avec les déclarations :

```

point a ;
pointcol b ;

```

L'affection :

```

a = b ;

```

Est tout a fait légale : il y a conversion de b dans le type point et l'affectation du résultat à a.

Par contre `b = a ;` est rejetée.

2<sup>ème</sup> possibilité : conversion de pointeur.

Reprenons nos 2 classes : "point" et "pointcol" avec toutes les deux une fonction membre `affiche()` ;

Les déclarations :

```
point *adp ;
pointcol *adcp ;
```

Là aussi, l'affectation : `adp = adcp ;` est tout a fait possible. Elle correspond à une conversion du type `pointcol *` dans le type `point *`.

L'affectation inverse `adcp = adp ;` sera alors rejeté, mais sera tout de même réalisable en faisant appel à l'opérateur de "cast" : `adcp = (pointcol *) adp ;`

Imaginons maintenant l'appel des fonctions `affiche()` des 2 classes avec la méthode par conversion de pointeur...

Les déclarations :

```
point p(3,5) ;   pointcol pc(8,6,2) ;
adp = &p ;      adcp = &pc ;
```

A ce niveau, l'instruction : `adp -> affiche()` ; appellera la méthode `point::affiche()`.

Par contre, l'instruction : `adpc -> affiche()` ; appellera la méthode `pointcol::affiche()`.

Si nous exécutons : `adp = adcp ;`

`adp -> affiche()` ; appellera `point::affiche()` car `adp` est du type `point *` : l'objet `adp` est du type `point` mais l'objet pointé par `adp` est du type `pointcol`.

Exemple avec nos 2 classes "point" et "pointcol" :

```
#include <iostream.h>
```

```
class point {
protected:    //pour pouvoir avoir accès dans la classe pointcol
    int x, y;
public:
    point(int abs=0,int ord=0) {
        x=abs;
        y=ord;
    }
    void affiche() {
        cout<<"Je suis un point"<<endl;
        cout<<" et mes coordonnees sont : "<<x<<" , "<<y<<endl;
    }
};
```

```

class pointcol:public point {
    short couleur;
public:
    pointcol(int abs=0,int ord=0,short c1=1):point(abs,ord) {
        couleur = c1;
    }
    void affiche() {
        cout<<"Je suis un point colore"<<endl;
        cout<<" mes coordonnees sont : "<<x<<" , "<<y<<endl;
        cout<<" et ma couleur est : "<<couleur<<endl;
    }
};

void main() {
    point p(3,5);
    point *adp=&p;
    pointcol pc(8,6,2);
    pointcol *adpc=&pc;
    adp->affiche();
    adpc->affiche();
    cout<<"----- Apres conversion par pointeurs -----"<<endl;
    adp=adpc;
    adp->affiche();
    adpc->affiche();
}

```

## 4.2. L'héritage multiple :

Le principe c'est qu'une classe peut dériver de 2 autres classes. Ce principe n'est pas très utilisé du fait qu'il reste assez complexe à la réalisation, mais il garde les grandes idées de l'héritage simple.

Prenons un exemple : 2 classes "point" et "coul"

```

class point {
    int x,int y;
public:
    point(...) {...}
    ~point() {...}
    affiche() {...}
};

class coul {
    short couleur;
public:
    coul(...) {...}
    ~coul() {...}
    affiche() {...}
};

```

Nous pouvons maintenant définir une classe héritant de ces deux classes en la déclarant ainsi : class pointcoul :public point,public coul {...} ;

Le constructeur doit néanmoins pouvoir transmettre des informations aux constructeurs des classes de base : class pointcoul(.1.) :public point(.2.),public coul(.3.)

- .1. Arguments de pointcoul ;
- .2. Arguments à transmettre à point ;
- .3. Arguments à transmettre à coul.

Les constructeurs seront appelés dans l'ordre suivant :

D'abord les constructeurs des classe de base, dans l'ordre où les classes de base sont déclarées dans la classe dérivée (ici point puis coul). Ensuite vient le constructeur de la classe dérivée (ici pointcoul).

Les destructeurs seront appelés dans l'ordre inverse lors de la destruction d'un objet de type pointcoul.

Supposons que la fonction "pointcoul" possède elle aussi une fonction membre destinée à l'affichage faffiche(), comme dans l'héritage, il faut préciser de quelle classe la fonction affiche provient grâce à l'opérateur de résolution de portée :

```
void affiche() {
    point::affiche();
    coul::affiche();
};
```

Exemple complet de l'utilisation de la classe "pointcoul" :

```
#include <iostream.h>

class point {
    int x,y;
public:
    point(int abs,int ord) {
        cout<<"++ Constr. point"<<endl;
        x=abs;
        y=ord;
    }
    ~point() {
        cout<<"-- Destr. point"<<endl;
    }
    void affiche() {
        cout<<"Coordonnees : "<<x<<" , "<<y<<endl;
    }
};

class coul {
    int couleur;
public:
    coul(int c1) {
        cout<<"++ Constructeur coul"<<endl;
        couleur=c1;
    }
    ~coul() {
        cout<<"-- Destr coul"<<endl;
    }
    void affiche() {
        cout<<"Couleur :"<<couleur<<endl;
    }
};

class pointcoul:public point,public coul {
public:
    pointcoul(int,int,int);
    ~pointcoul() {
        cout<<"---- Destr. pointcoul"<<endl;
    }
};
```

```

        void affiche() {
            point::affiche();
            coul::affiche();
        }
};
pointcoul::pointcoul(int abs,int ord,int c1):point(abs,ord),coul(c1) {
    cout<<"++++ Constr. pointcoul"<<endl;
}

void main() {
    pointcoul p(3,9,2);
    cout<<"Appel de affiche pointcoul"<<endl;
    p.affiche();
    cout<<"Appel de affiche point"<<endl;
    p.point::affiche();
    cout<<"Appel de affiche coul"<<endl;
    p.coul::affiche();
}

```

- Résolution des éventuels conflits avec les classes virtuelles :

Les déclarations telles que :

```

class A {
    ...
    int x,y;
};
class B:public A {...};
class C:public A {...};
class D:public B, public C {...};

```

En quelque sorte, D hérite deux fois de A... Vu que B et C héritent déjà de A, et donc les membres de A apparaissent deux fois dans D. Pour les fonctions membres, cela ne prêle pas beaucoup à conséquence. Par contre, les membres données (x et y) seront effectivement dupliqués dans tous les objets de type D. Y aura-t-il redondance ? Cela dépend : Si on veut que D dispose de 2 jeux de données, on se contentera d'utiliser l'opérateur de résolution de portée :

```
A::B::x de A::C::x
```

En général, la duplication des données n'est pas très appréciée. Pour que cela n'arrive pas, on peut demander à C++ de n'incorporer qu'une seule fois les membres de A dans la classe D. Pour cela, il faut juste préciser que la classe A est de type virtuelle.

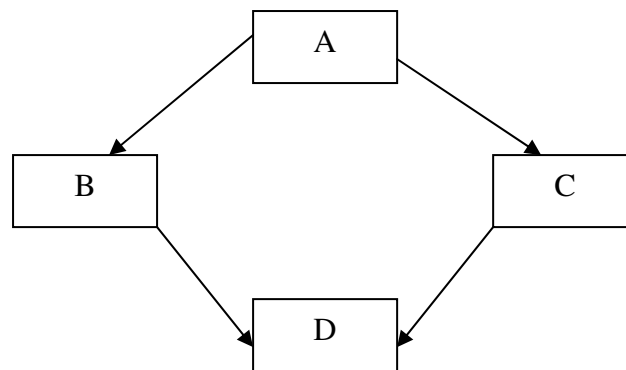
```

class B:public virtual A {...};
class C:public virtual A {...};
class D:public B, public C {...};

```

Introduire A comme "virtuelle" dans la déclaration de B signifie que A ne devra être introduite qu'une seule fois dans les descendants éventuels de C.

- Appels des constructeurs et des destructeurs : cas des classes virtuelles :



Si la classe de base n'est pas déclarée virtuelle, et que d'autres classes héritent de cette classe, il y aura copie de la classe de base dans les héritières. Donc il y aura appel 2 fois au constructeur de A. L'appel se fait ainsi : constructeur de la classe de base, constructeur de la première classe dérivée, constructeur de la classe de base, constructeur de la seconde classe dérivée, etc... et éventuellement un classe qui dérive encore des deux autres classes.

Ici on aura : A B A C D

Si la classe de base est déclarée virtuelle, tout change... On construit un seul objet de type A (donc pas de copies). Mais quels arguments faut-il transmettre ? Le choix des informations à fournir au constructeur de A n'a plus lieu dans B ou C mais bien dans D. Pour ce faire, C++ autorise à spécifier (dans le constructeur de D) des informations destinées à A : `D (int n, int p, double z) : B(n, p, z), A(n,p) ;`

Il faudra absolument que A dispose d'un constructeur sans arguments afin de pouvoir construire convenablement des objets de type B ou C.

Pour l'appel des constructeurs, on appelle toujours le constructeur de la classe virtuelle. Ici, on aura A B C D.

Exemple de la mise en œuvre de l'héritage multiple et de la dérivation virtuelle :

```

#include <iostream.h>

class point {
    int x,y;
public:
    point(int abs,int ord) {
        cout<<"++ Constr. point"<<endl;
        x=abs;
        y=ord;
    }
}
  
```

```

    point() { //constr par défaut nécessaire pour dérivations virtuelles
        cout<<"Constr. par default point"<<endl;
        x=0;
        y=0;
    }
    void affiche() {
        cout<<"Coordonnees : "<<x<<" , "<<y<<endl;
    }
};
class coul {
    int couleur;
public:
    coul(int c1) {
        cout<<"++ Constr. coul"<<endl;
        couleur=c1;
    }
    void affiche() {
        cout<<"Couleur : "<<couleur<<endl;
    }
};
class masse {
    int mas;
public:
    masse(int m) {
        cout<<"++ Constr. masse"<<endl;
        mas=m;
    }
    void affiche() {
        cout<<"Masse : "<<mas<<endl;
    }
};
class pointcoul:public virtual point,public coul {
public:
    pointcoul(int abs,int ord,int c1):coul (c1) {
        cout<<"++++ Constr. pointcoul "<<abs<<" , "<<ord<<" "<<c1<<endl;
    }
    void affiche() {
        point::affiche();
        coul::affiche();
    }
};
class pointmasse:public virtual point,public masse {
public:
    pointmasse(int abs,int ord,int m):masse (m) {
        cout<<"++++ Constr. pointmasse"<<abs<<" , "<<ord<<" "<<m<<endl;
    }
    void affiche() {
        point::affiche();
        masse::affiche();
    }
};
class pointcolmasse:public pointcoul,public pointmasse {
public:
    pointcolmasse(int abs,int ord, int c,int
m):point(abs,ord),pointcoul(abs,ord,c),pointmasse(abs,ord,m) {
        cout<<"++++ Constr. pointcolmasse"<<abs<<" , "<<ord<<" "<<c<<" "<<m<<endl;
    }
}

```

```

    void affiche() {
        point::affiche();
        coul::affiche();
        masse::affiche();
    }
};

void main() {
    pointcoul p(3,9,2);
    cout<<"Appel de affiche pointcoul"<<endl;
    p.affiche();
    pointmasse pm(12,25,100);
    cout<<"Appel de affiche pointmasse"<<endl;
    pm.affiche();
    pointcolmasse pcm(2,5,10,20);
    cout<<"Appel de affiche pointcolmasse"<<endl;
    pcm.affiche();
}

```

A l'exécution, on aura :

- D'abord le constructeur par défaut, le constructeur coul, celui de pointcoul.
- Le constructeur par défaut de point, constructeur de masse puis pointmasse.
- Le constructeur de point, de coul de pointcoul, de masse, pointmasse et enfin pointcolmasse.

## 5. Les fonctions amies.

Nous avons vu que le principe d'encapsulation interdisait à une fonction membre d'une classe d'accéder à des données privées d'une autre classe. Or, il est parfois intéressant que cela soit possible. Imaginons une classe "vecteur" et une classe "matrice". Il serait intéressant de calculer le produit d'une matrice par un vecteur. La solution, c'est d'utiliser une fonction amie. C'est une sorte de compromis entre encapsulation formelle des données privées et des données publiques. Une déclaration d'amitié rend possible l'accès aux données privées, au même titre que n'importe quelle fonction membre.

Il existe plusieurs situations d'amitiés :

- Fonction indépendante, amie d'une classe :

La fonction amie peut accéder aux membres privés de la classe.

La syntaxe est la suivante : friend type\_retour fonction(param...);

## Exemple :

```
#include <iostream.h>

class point {
    int x,y;
public:
    point(int abs=0,int ord=0) {
        x=abs;
        y=ord;
    }
    friend bool coincide (point,p2);
    void affiche() {
        cout<<"Coordonnees : "<<x<<" , "<<y<<endl;
    }
};

bool coincide(point p1,point p2) {
    if((p1.x==p2.x) && (p1.y==p2.y)) return true;
    else return false;
}

void main() {
    point a(1,0),b(1),c;
    a.affiche();
    b.affiche();
    c.affiche();
    if(coincide (a,b)==true) cout<<"le point a est le meme que b"<<endl;
    else cout<<"a et b sont differents"<<endl;
    if(coincide (a,c)==true) cout<<"le point a est identique au point c"<<endl;
    else cout<<"a est different de c"<<endl;
}
```

- Fonction membre d'une classe, amie d'une autre classe :

Dans ce cas-ci, il faut préciser (dans la déclaration d'amitié) la classe à laquelle appartient la fonction concernée (ceci à l'aide de l'opérateur de résolution de portée).

Supposons que nous ayons à définir deux classes nommées A et B et que nous ayons besoin dans B d'une fonction membre f, de prototype : int f(char, A) ;

Si f doit pouvoir accéder aux membres privés de A, elle sera déclarée amie au sein de la classe par : friend int B ::f(char, A) ;

## Exemple :

```
#include <iostream.h>

class vect;                                     //définie plus tard pour compiler correctement

class matrice {
    double mat[3][3];
public:
    matrice(double t[3][3]) {
        int i,int j;
        for(i=0;i<3;i++) {
            for(j=0;j<3;j++) {
                mat[i][j]=t[i][j];
            }
        }
    }
    vect prod(vect);
};

class vect {
    double v[3];
public:
    vect(double v1=0,double v2=0,double v3=0) {
        v[0]=v1;
        v[1]=v2;
        v[2]=v3;
    }
    friend vect matrice::prod (vect);
    void affiche() {
        int i;
        for(i=0;i<3;i++) {
            cout<<v[i]<<" ";
        }
        cout<<endl;
    }
};

vect matrice::prod(vect x) {
    int i,int j;
    double som;
    vect res;
    for (i=0;i<3;i++) {
        for(j=0,som=0;j<3;j++) {
            som+=mat[i][j]*x.v[j];
        }
        res.v[i]=som;
    }
    return res;
}

void main() {
    vect w(1,2,3);
    vect res;
    double tb[3][3]={1,2,3,4,5,6,7,8,9};
    matrice a=tb;
    res=a.prod(w);
    res.affiche();
}
```

- Fonction amie de plusieurs classes :

Rien n'empêche qu'une même fonction fasse l'objet de déclaration d'amitié dans différentes classes. Voici un exemple de fonction amie de deux classes A et B :

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

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

void f(A a1,B b1) {
    // on a accès ici aux membres privés
    // de n'importe quel objet de type A ou B
}
```

- Toutes les fonctions membres sont amies

Généralisation d'une fonction membre d'une classe, amie d'une autre. On pourrait effectuer autant de déclaration d'amitié qu'il y a de fonctions concernées. Mais il est plus simple d'effectuer une déclaration globale. Pour dire que toutes les fonctions membres de la classe B sont amies de la classe A, on utilisera la déclaration suivante :

```
friend class B ;
```

## 6. Les flux d'entrées/sorties :

En programmation, un flux est un canal qui peut recevoir soit recevoir une information (flux d'entrée) soit fournir une information (flux de sortie).

Un flot est un objet d'une classe prédéfinie, on connaît déjà :

- ostream pour un flot de sortie ;
- istream pour un flot d'entrée.

### 6.1. La classe ostream :

La classe ostream est une classe utilisée pour le flot de sortie.

On a déjà vu la surcharge de l'opérateur <<. La classe ostream ne contient pas que l'opérateur <<. On peut lui rajouter les fonctions membres put et write et aussi les possibilités de formatage.

- La fonction put :

Cette fonction transmet au flot correspondant le caractère reçu en argument.

On a donc : cout.put (c) ;

Qui transmet au flot `cout` le caractère contenu dans `c` (comme dans `cout<<c`).  
La valeur de retour de `put` est le flot concerné, après qu'on y a écrit le caractère correspondant. Cela permet d'écrire par exemple (`c1`, `c2`, `c3` étant de type `char`) :

```
cout.put(c1).put(c2).put(c3) ;
```

Ce qui est équivalent à :

```
cout.put(c1) ;  
cout.put(c2) ;  
cout.put(c3) ;
```

- La fonction `write` :

Cette fonction permet de transmettre au flot de sortie considéré une suite de caractères (octets) de longueur donnée.

Avec : `char t[t] = "bonjour" ;`, l'instruction : `cout.write(t, 4) ;`

Envoie sur le flot `cout` 4 caractères consécutifs à partir de l'adresse `t`, c'est-à-dire les caractères `b`, `o`, `n` et `j`.

La fonction `write` ne fait pas parvenir le caractère de fin de chaîne (caractère nul) ; si un tel caractère apparaît dans la longueur prévue, il sera transmis, comme les autres, au flot de sortie.

- Possibilités de formatage :
  - Action sur base de numération :

Quand on écrit une valeur entière sur le flot de sortie, on peut choisir d'afficher cette valeur dans les différentes bases qui existent : en décimal, hexadécimal, octal ou binaire.

Exemple :

```
#include <iostream.h>
```

```
void main() {  
    int n=12000;  
    cout<<"Par défaut : " <<n<<endl;  
    cout<<"En hexadécimal : " <<hex<<n<<endl;  
    cout<<"En décimal : " <<dec<<n<<endl;  
    cout<<"En octal : " <<oct<<n<<endl;  
    cout<<"Et ensuite : " <<n<<endl;  
  
    bool ok=1; //ou true  
    cout<<"Par défaut : " <<ok<<endl;  
    //cout<<"Avec nboolalpha : " <<nboolalpha<<ok<<endl;  
    //cout<<"Avec boolalpha : " <<boolalpha<<ok<<endl;  
    cout<<"Et ensuite : " <<ok<<endl;  
}
```

Les différents symboles `hex`, `oct`, `dec` se nomment des manipulateurs. Ce sont des opérateurs prédéfinis à un seul opérande de type flot. Ils fournissent en retour le même flot (après qu'ils ont opéré une certaine action).

- Action sur le gabarit de l'information écrite :

Ici on fait appel au manipulateur `setw` pour pouvoir agir sur la largeur sur laquelle une information sera écrite.

```
#include <iostream.h>
#include <iomanip.h>
void main() {
    int n=12345;
    int i;
    for(i=0;i<15;i++) {
        cout<<setw(2)<<i<<" : "<<setw(i)<<n<<endl;
    }
}
```

Ce manipulateur comprend un paramètre représentant le gabarit souhaité. Il nécessite l'incorporation du fichier en-tête `<iomanip.h>`.

## 6.2. La classe `istream` :

Cette classe fournit des entrées formatées ou non.

On a déjà vu le rôle de l'opérateur `>>`. Mais cette classe ne contient pas seulement cet opérateur. Il y a aussi des fonctions membres et aussi des possibilités de formatage des informations.

- La fonction `get` :

La fonction `istream & get (char &)` permet d'extraire un caractère d'un flot d'entrée et de le ranger dans la variable (de type `char`) qu'on lui fournit en argument. Elle fournit en retour la référence au flot concerné (après avoir extrait le caractère voulu).

Cette fonction n'est pas à confondre avec `int get()` qui permet aussi d'extraire un caractère d'un flot mais qui renvoi comme valeur un entier. Et si un EOF est trouvé, elle renvoi -1.

- Les fonctions `getline` et `gcount` :

Ces deux fonctions facilitent la lecture des chaînes de caractères (ou d'une suite de caractères quelconques).

`Getline` se présente sous la forme : `istream & getline (char *ch, int taille, char delim="\\n")`

Elle lit des caractères sur le flot l'ayant appelé et les place dans l'emplacement d'adresse `ch`. Elle s'interrompt quand le caractère de `delim` a été trouvé ou quand `taille - 1` caractères ont été lus.

La fonction `gcount` quant à elle fournit le nombre de caractères effectivement lus lors du dernier appel `getline`. `Gcount` fournit la longueur effective de la chaîne rangée en mémoire par `getline`.

Exemple :

```
#include <iostream.h>
void main() {
    const int max=120;           //longueur maximale d'une ligne de texte
    char ch[max+1];
    int lg;
    do {
        cout<<"Entrez une chaine : ";
        cin.getline(ch,max);
        lg=cin.gcount();
        cout<<"ligne de "<<lg-1<<"caractères : "<<ch<<endl;
    }
    while(lg>1);
}
```

- A fonction `read` :

La fonction `read` permet de lire sur le flot d'entrée considéré une suite de caractères (octets) de longueur donnée.

Soit avec : `char t[10]` ; l'instruction : `cin.read (t,5)` ; lira sur `cin` 5 caractères et les rangera à partir de l'adresse `t`.

Cette fonction s'avèrera indispensable pour accéder à des fichiers sous forme binaire, c'est-à-dire en recopiant en mémoire les informations telles qu'elles figurent dans le fichier.

### 6.3. Connexion d'un flot à un fichier :

- Connexion d'un flot de sortie à un fichier :

Pour associer un flot de sortie à un fichier, il suffit de créer un objet de type `ofstream` (classe dérivant de `ostream`). Pour pouvoir utiliser cette classe, il ne faut pas oublier d'inclure le fichier d'en-tête `fstream` (en plus d'`iostream`).

Le constructeur de cette classe nécessite deux arguments :

- le nom du fichier concerné (sous forme d'une chaîne de caractères) ;
- un mode d'ouverture défini par une constante entière.

Soit la déclaration d'un objet du type `ofstream` :

```
ofstream sortie("dudule.txt",ios ::out|ios ::binary) ;           //ou simplement ios ::out
```

L'objet `sortie` sera donc associé à un fichier nommé `dudule.txt`, après avoir été ouvert en écriture.

Ainsi la construction d'un objet faite, l'écriture dans le fichier associé peut se faire. Elle pourra être faite de différente façon :

- Pour réaliser des sorties formatées :  
sortie<<...<<...<<... ;
- En réalisant des écritures binaires :  
sortie.write(...);

De même, on peut connaître le statut d'erreur du flot correspondant en examinant la valeur de sortie : if(sortie) ...

### Exemple :

```
#include <fstream.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
const int lmax=20;
void main() {
    char nomfich[lmax+1];
    int n;
    cout<<"Nom du fichier a creer : ";
    cin>>setw(lmax)>>nomfich;
    ofstream sortie (nomfich,ios::out|ios::binary);
    if(!sortie) {
        cout<<"Creation impossible"<<endl;
        exit(1);
    }
    do {
        cout<<"Donnez un entier : ";
        cin>>n;
        if (n) sortie.write((char *)&n,sizeof(int));
    }
    while (n && (sortie));
    sortie.close;
}
```

- Connexion d'un flot d'entrée à un fichier :

Pour pouvoir associer un flot de sortie à un fichier, on emploie (similairement à un flot de sortie) un objet de type ifstream (qui dérive de la classe istream) en n'oubliant pas d'inclure le fichier en tête fstream.h.

Le constructeur comporte les mêmes arguments que le précédent : c'est-à-dire un nom de fichier et un mode d'ouverture.

Soit l'instruction : ifstream entree ("dudule.txt", ios ::in|ios ::binary); //ou seulement ios ::in

L'objet entree sera associé au fichier de nom dudule.txt, après avoir été ouvert en lecture.

Une fois l'objet de la classe ifstream construit, la lecture dans le fichier qui lui est associé pourra se faire comme n'importe quel flot d'entrée en faisant appel à toutes les facilités de la classe istream.

Après la déclaration d'"entree", différentes instructions sont possibles :

- pour réaliser des lectures formatées :  
entree>>...>>...>>... ;
- Pour réaliser des lectures binaires :  
entree.read(...);

Exemple :

```
#include <fstream.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
const int lmax=20;
void main() {
    char nomfich[lmax+1];
    int n;
    cout<<"Nom du fichier a lister : ";
    cin>>setw(lmax)>>nomfich;
    ifstream entree (nomfich,ios::in|ios::binary);
    if(!entree) {
        cout<<"Ouverture impossible"<<endl;
        exit(-1);
    }
    while (entree.read((char *)&n,sizeof(int))) {
        cout<<n<<endl;
    }
    entree.close();
}
```

- Connexion d'un flot d'accès direct :

Dès qu'un flot s'est connecté à un fichier, il est possible de réaliser un "accès direct" à ce fichier en agissant tout simplement sur un pointeurs dans ce fichier (un nombre précisant le rang du prochain octet à lire ou à écrire). Après chaque opération (écriture ou lecture), ce pointeur est incrémenté du nombre d'octets transférés.

Dans chacune des deux classes, une fonction membre nommé :

- seekg pour ifstream ;
- seekp pour ofstream ;

qui permet de donner une certaine valeur au pointeur. Ces fonctions comportent deux arguments :

- Un entier représentant un déplacement du pointeur par rapport à une origine précisée par le second argument ;
- Une constante entière choisie parmi trois valeurs prédéfinies dans ios :
  - ios::beg : le déplacement est exprimé par rapport au début du fichier ;
  - ios::cur : le déplacement est exprimé par rapport à la position actuelle.

- `ios::end` : le déplacement est exprimé par rapport à la fin du fichier.

Par défaut cet argument a comme valeur `ios::beg`.

Pour connaître la position courante du pointeur, on peut utiliser :

- `tellg` pour `ifstream` ;
- `tellp` pour `ostream` ;

Exemple :

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const int lmax_nom_fich = 20;
void main() {
    char nomfich[lmax_nom_fich+1];
    int n,num;
    cout<<"Nom du fichier a consulter : ";
    cin>>setw (lmax_nom_fich)>>nomfich;
    ifstream entree (nomfich,ios::in|ios::binary);
    if(!entree) {
        cout<<"Ouverture impossible"<<endl;
        // exit(-1);
    }
    do {
        cout<<"Numero de l'entier recherche : ";
        cin>>num;
        if (num) {
            entree.seekg (sizeof(int) *(num-1),ios::beg);
            entree.read ((char *) &n, sizeof(int));
            if(entree) {
                cout<<"-- Valeur : "<<n<<endl;
            }
            else {
                cout<<"-- Erreur"<<endl;
                entree.clear();
            }
        }
    }
    while (num);
    entree.close();
}
```

#### 6.4. Possibilités de formatage en mémoire :

- La classe `ostrstream` :

Un objet de la classe `ostream` peut recevoir des caractères, au même titre qu'un flot de sortie. La seule différence est que ces caractères ne sont pas transmis à un périphérique ou à un fichier, mais simplement conservés dans le fichier lui-même (plus précisément dans un tableau de la classe `ostream` qui est créé dynamiquement => aucun problème pour la taille).

Il existe une fonction membre permettant d'obtenir l'adresse du tableau en question. Elle existe sous le nom de "str". Il pourra alors être manipulé comme n'importe quel tableau de caractères (repéré par un pointeur de type char \*).

Soit la déclaration : `ostream tab ;`

Les instructions `tab << ... << ... << ... ;` ou `tab.put(...)` ; ou `tab.write(...)` ;

Pourront servir à insérer des caractères dans l'objet "tab".

L'adresse du tableau de caractères ainsi créé pourra être obtenue par :

```
char * adt = tab.str() ;
```

On pourra agir alors sur les caractères que contient cette adresse.

- La classe `istream` :

Pour pouvoir construire un objet de type `istream`, le constructeur doit recevoir 2 paramètres : l'adresse d'un tableau de caractères et le nombre de caractères à prendre en compte.

On peut alors extraire des caractères de cet objet.

Soit les déclarations : `char t[100]` ; et `istream tab(t,sizeof(t))` ;

Les caractères du tableau `t` pourront être extraits avec des instructions telles que : `tab >> ... >> ... >> ... ;` ou `tab.get(...)` ; ou `tab.read(...)` ;

Il est aussi possible de se déplacer dans ce tableau (comme dans un fichier) en faisant appel à la fonction `seekg`. Par exemple, pour replacer le pointeur en début de tableau : `tab.seekg(0, ios::beg)` ;

**Exemple :**

```
#include <iostream.h>
#include <sstream.h>
const int lmax = 122;           //longueur max d'une ligne clavier
void main() {
    int n, erreur;
    char c;
    char ligne [lmax];
    do {
        cout<<"Donnez un entier et un caractère : ";
        cin.getline(ligne, lmax);
        istream tampon (ligne,cin.gcount());
        if tampon >> n >>c) erreur=0;
        else erreur = 1;
    }
    while (erreur);
    cout<<"Merci pour "<< n <<" et "<< c <<endl;
}
```

Attention l'inclure `<sstream.h>` donne une erreur à la compilation... Je n'arrive pas à trouver la solution à ce problème alors que `<sstream>` existe bel et bien... Si quelqu'un a trouvé...

## → La Généricité :

### 7. Les patrons de fonctions :

Un patron de fonction permet au compilateur d'adapter une fonction à n'importe quel type.

C'est en fait un plan à partir duquel le compilateur est capable, en fonction des besoins, de générer plusieurs fonctions réelles, qui diffèrent par le type des objets qu'elles manipulent.

- Création d'un patron de fonction :

Supposons une fonction devant renvoyer le minimum de deux valeurs.

Définition de la fonction pour le type int :

```
int min (int a, int b) {  
    if(a<b) return a;           //ou bien return a<b?a:b;  
    else return b;  
}
```

Définition de la fonction pour le type float :

```
float min (float a, float b) {  
    if(a<b) return a;           //ou bien return a<b?a:b;  
    else return b;  
}
```

On peut alors écrire plusieurs définitions assez semblables les unes des autres.

Pour simplifier les choses, on pourrait créer un **patron de fonction** :

```
template <classe T> T min(T a, T b) {  
    if (a<b) return a;  
    else return b;  
}
```

template <class T> précise tout simplement que l'on a affaire à un patron de fonction dans lequel apparaît un paramètre de type T. Il est à noter que l'utilisation de class sert à préciser que T est un paramètre d'une certaine classe.

Le reste T min (T a, T b) précise que min est une fonction recevant 2 arguments de type T et qui fournit un résultat du même type.

Exemple complet avec la recherche d'un minimum entre deux nombres quelconques :

```
#include <iostream.h>
template <class T> T min (T a, T b) {
    if (a<b) return a;
    else return b;
}
void main() {
    int n=4,p=12;
    float x=2.5, y=3.25;
    cout<<"min(n,p)="<<min(n,p)<<endl;
    cout<<"min(x,y)="<<min(x,y)<<endl;
}
```

### 7.1. Les paramètres de type d'un patron :

Comme les fonctions, les patrons de fonctions peuvent comporter plusieurs arguments. Dans les patrons de fonctions, ils ne sont plus appelés paramètres mais paramètres d'expressions.

- Utilisation des paramètres de type dans la définition d'un patron :

Un patron de fonction peut comporter un ou plusieurs paramètres de type (chacun devant être précédé du mot clef `class`).

Soit : `template <class T, class U> fct (T a, T *b, U c) {.....}`

Ces paramètres peuvent intervenir à n'importe quel endroit de la définition d'un patron :

- dans l'en-tête ;
- dans les déclarations de variables locales ;
- dans les instructions exécutables (`new`, `sizeof`, etc...).

- Identification des paramètres de type d'une fonction patron :

Reprenons le patron `min` :

```
#include <iostream.h>
template <class T> T min (T a, T b) {
    if (a<b) return a;
    else return b;
}
```

Prenons maintenant les déclarations : `int n ; char c ;`

La question que l'on se pose, c'est que va faire le compilateur en présence d'un appel tel que `min(n,c)` ; ? Et bien, il génèrera une erreur... En effet, C++ a prévu une règle : il doit y avoir correspondance absolue des types. C'est-à-dire que nous pouvons utiliser le patron `min` que pour des appels dans lesquels les deux arguments ont le même type.

Il est quand même possible d'intervenir sur le mécanisme d'identification de type. Il est autorisé de spécifier un ou plusieurs paramètres de type au moment de l'appel du patron. Voici un exemple avec les déclarations précédentes :

```
min<int>(c,n) //force l'utilisation de min<int>, et donc la conversion de c en int;
              //le résultat sera de type int
min<char>(c,n) //force l'utilisation de min<char>, et donc la conversion de n en char;
               //le résultat sera de type char
```

**Autre exemple :**

```
#include <iostream.h>
template <class T,class U> T fct(T x,U y,T z) {
    return x+y+z;
}
void main() {
    int n=1,p=2,q=3;
    float x=2.5,y=5.0;
    cout<<fct(n,x,p)<<endl;
    cout<<fct(x,n,y)<<endl;
    cout<<fct(n,p,q)<<endl;
    //cout<<fct(n,p,x)<<endl;           //erreur pas de correspondance !
}
```

## 7.2. La surdéfinition de patrons :

Comme pour une fonction normale, il est possible de surdéfinir un patron de fonction. C'est-à-dire de définir plusieurs patrons possédant des arguments différents. Il est à noter que cette situation revient à définir plusieurs familles de fonctions.

- Exemples ne comportant que des paramètres de type :

Ici, on surdéfini deux patrons de fonctions min, dans le but de disposer :

- d'une première famille de fonctions à deux arguments de même type quelconque ;
- d'une seconde famille de fonction à trois arguments de même type quelconque.

```
#include <iostream.h>
//Patron numéro 1 :
template <class T>T min(T a,T b) {
    if (a<b) return a;
    else return b;
}
//Patron numéro 2 :
template <class T>T min(T a,T b,T c) {
    return min(min(a,b),c);
}
```

```

void main() {
    int n=12,p=15,q=2;
    float x=3.5,y=4.25,z=0.25;
    cout<<min(n,p)<<endl;           //patron 1
    cout<<min(n,p,q)<<endl;         //patron 2 avec des types int
    cout<<min(x,y,z)<<endl;         //patron 2 avec des types float
}

```

En règle générale, on peut surdéfinir des patrons possédant un nombre différent de paramètres de type. On peut alors avoir des en-têtes de fonctions aussi variés que l'on veut.

Voici encore un autre exemple avec la fonction min :

```

#include <iostream.h>
//Patron numéro 1 :
template <class T>T min(T a,T b) {
    if (a<b) return a;
    else return b;
}
//Patron numéro 2 :
template <class T>T min(T *a,T b) {
    if(*a < b) return *a;
    else return b;
}
//Patron numéro 3 :
template <class T>T min(T a,T *b) {
    if(a<*b) return a;
    else return *b;
}
void main() {
    int n=12,p=15;
    float x=2.5,y=5.2;
    cout<<min(n,p)<<endl;
    cout<<min(&n,p)<<endl;
    cout<<min(x,&y)<<endl;
    cout<<min(&n,&p)<<endl;
}

```

Avec l'appel `min(&n,&p)` on fera appel au patron numéro 1. En effet, on devra faire appel au patron `min(T *a,T *b)` ; ce qui revient au même de comparer deux nombres de type `int`.

- Exemples comportant des paramètres expressions :

Quand on a des paramètres d'expressions, on donne à la surdéfinition de patron un caractère plus général. Voici un exemple où il y a surdéfinition de deux familles de fonction min :

- L'une pour déterminer le minimum de deux valeurs de même type quelconque.
- L'autre pour déterminer le minimum des valeurs d'un tableau de type quelconque et de taille quelconque.

```

#include <iostream.h>
//Patron numéro 1 :
template <class T>T min(T a,T b) {
    if (a<b) return a;
    else return b;
}
//Patron numéro 2 :
template <class T>T min(T *t,int n) {
    int i;
    T min=t[0];
    for(i=0;i<n;i++) {
        if (t[i]<min) min=t[i];
    }
    return min;
}
void main() {
    long n=2,p=12;
    float t[6]= {2.5,3.2,1.5,3.8,1.1,2.8};
    cout<<min(n,p)<<endl;
    cout<<min(t,6)<<endl;
}

```

## 8. Les patrons de classes :

On revient sur le principe de patron de fonction pour des patrons de classes.

### 8.1. Exemple de création et d'utilisation d'un patron de classe :

- Création d'un patron de classes :

Reprenons (et oui encore un fois !) la classe point que nous avons tellement peu utilisée :

```

class point {
    int x;
    int y;
public:
    point(int abs=0,int ord=0);
    void affiche();
    ...
};

```

Avec une telle procédure, on impose que les coordonnées des points soient de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type, nous devons surdéfinir une autre classe en remplaçant le mot clé int par le nom du type voulu.

Il est plus simple de définir un seul patron de classe :

```
template <class T> class point {
    T x; T y;
public :
    point(T abs=0,T ord=0);
    void affiche();
};
```

Il ne faut pas oublier de définir les fonctions membres de la classe. La démarche va pour les définir varie si la fonction est en ligne ou non.

→ Pour les fonctions en ligne, les choses restent naturelles : il faut utiliser le paramètre T a bon escient. Par exemple pour notre constructeur :

```
point (T abs=0,T ord=0) {
    x=abs;
    y=ord;
}
```

→ En revanche si la fonction est définie en dehors de la définition de la classe, il faut rappeler au compilateur différents éléments :

- dans la surdéfinition de cette fonction, on doit faire apparaître des paramètres de type : `template<class T>`
- le nom du patron concerné : `point<T> ::affiche()`

En définitive, voici comment se présente l'en-tête de la fonction affiche si nous la définissons ainsi en dehors de la classe :

```
template <class T> void point<T> ::affiche()
```

Voici la déclaration complète et la définition complète du patron de classe point :

```
#include <iostream.h>
template <class T> class point {
    T x; T y;
public :
    point(T abs=0,T ord=0) {
        x=abs;
        y=ord;
    }
    void affiche();
};
template <class T> void point<T>::affiche() {
    cout<<"Coordonnées : "<<x<<" "<<y<<endl;
}
```

- Utilisation d'un patron de classes :

Voilà, c'est bien d'en avoir créé un, mais comment l'utiliser maintenant ?

Soit la déclaration : `point <int> ai ;`

Elle conduit le compilateur à instancier la définition d'une classe point dans laquelle le paramètre T prend la valeur int. Autrement dit, tout se passe comme si nous avions fourni une définition complète de cette classe.

Si nous avons déclaré : `point<double> ad ;`

On aurait eu un paramètre T de valeur double.

Si nous devons fournir des arguments au constructeur :

```
point <int> ai (3,5) ;  
point <double> ad (3.5,2.3) ;
```

Exemple complet de création et d'utilisation d'un patron de classe :

```
#include <iostream.h>  
template <class T> class point {  
    T x; T y;  
    public :  
        point(T abs=0,T ord=0) {  
            x=abs;  
            y=ord;  
        }  
        void affiche();  
};  
template <class T> void point<T>::affiche() {  
    cout<<"Coordonnees : "<<x<<" , "<<y<<endl;  
}  
void main() {  
    point<int> ai(3,5);  
    ai.affiche();  
    point<char> ac('d','y');  
    ac.affiche();  
    point<double> ad(3.5,2.3);  
    ad.affiche();  
}
```

## 8.2. Les paramètres de type d'un patron de classes :

Tout comme les patrons de fonctions, les patrons de classes peuvent comporter des paramètres de type et des paramètres expressions.

- Les paramètres de type d'un patron de classes :

Ils peuvent être en nombre et utilisés comme bon nous semble dans la définition du patron de classes. Exemple :

```
template <class T,class U,class V> class essai  
    //on a une liste de 3paramètres de nom T,U,V  
{  
    T x;                //membre x de type T  
    U t[5];            //tableau t de 5 éléments de type U  
    ...  
    V fm1(int,U);     //déclaration d'une fonction membre recevant 2 arguments  
                    // de type int et U renvoyant un résultat de type V  
    ...  
};
```

- Les paramètres expression d'un patron de classes :

Concept assez semblable que pour les patrons de fonctions mais les paramètres expression d'un patron de classes doivent être constantes.

Supposons la classe tableau :

```
template <class T,int n> class tableau {
    T tab[n];
    public:
        ...
};
```

La liste de paramètre du patron comporte 2 paramètres :

- un paramètre de type, introduit par le mot clé class ;
- un "paramètre expression" de type int ; on précisera sa valeur lors de la déclaration d'une instance particulière de la classe tableau.

Exemple :

```
#include <iostream.h>
template <class T,int n>class tableau {
    T tab[n];
    public :
        tableau() {cout<<"Construction tableau"<<endl;}
        T & operator[](int i) {
            return tab[i];
        }
};
class point {
    int x;
    int y;
public:
    point(int abs=1,int ord=1) {
        x=abs;
        y=ord;
        cout<<"Point construit"<<endl;
    }
    void affiche() {
        cout<<"Coordonnees : "<<x<<" , "<<y<<endl;
    }
};
void main() {
    tableau<int,4> ti;
    for (int i=0;i<4;i++) {
        ti[i]=i;
    }
    cout<<"ti : ";
    for(i=0;i<4;i++) {
        cout<<ti[i]<<" , ";
    }
    cout<<endl;
    tableau <point,3> tp;
    for(i=0;i<3;i++) {
        tp[i].affiche();
    }
}
```

## 9. Librairie standard du C++ :

La STL (Standard Template Library) ou Librairie standard est la librairie fournissant des implémentations génériques de la plupart des conteneurs. Cette librairie utilise l'espace de nom `std`.

Elle comprend les itérateurs et les conteneurs.

### 9.1. Notion d'itérateur :

Pour l'instant, on n'a pas encore vu les conteneurs... J'en parle dans ce point mais les conteneurs font partie du point suivant.

Cette notion a été introduite dans un souci d'homogénéisation des actions sur un conteneur. C'est en fait un objet défini par la classe conteneur concernée qui généralise la notion de pointeur. Un itérateur peut :

- Pointer (à un instant donné) sur un élément d'un conteneur ;

- Etre incrémenté par l'opérateur `++` de manière à pointer sur l'élément suivant du même conteneur ;

- Etre déréférencé en utilisant l'opérateur `*` ;

- Si ils sont deux sur un même conteneur, ils peuvent être comparés par égalité ou inégalité.

D'autres conteneurs pourront avoir des propriétés en plus de celles précédentes.

On pourra alors :

- Décrémenter avec l'opérateur `-` (pour les raison inverse que `++`) ;

- Accéder directement à un emplacement du conteneur ;

- Parcourir ce conteneur grâce aux fonctions membres `begin()` et `end()`. Si on a un conteneur `lp`, on aura `lp.begin()` pour le début du conteneur et `lp.end()` pour la fin du conteneur ;

- Parcourir un conteneur en sens inverse grâce aux fonctions membres `rbegin()` et `rend()`.

### 9.2. Notion de conteneur :

La bibliothèque standard fournit un ensemble de classes dites conteneurs. Ceux-ci permettent de représenter des structures de données les plus répandues telles que des vecteurs, des listes, des ensembles ou des tableaux associatifs.

On retrouve alors :

```
list <int> li ; //liste vide d'éléments de type int
vector <double> ld ; //vecteur vide d'éléments de type double
list <point> lp ; //liste vide d'éléments de type point
```

Les conteneurs peuvent être des conteneurs en séquence ou associatifs (on ne tarde pas sur les explications longues). Il faut juste savoir que les conteneurs séquentiels correspondent à des éléments ordonnés (comme dans un vecteur ou une liste) et qu'aux conteneurs associatifs est associée une clé pour accéder à la valeur associée.

### 9.3. Les conteneurs séquentiels :

On parle alors des conteneurs `vector`, `list` et `deque`. Ces conteneurs peuvent être construits de différentes manières selon que l'on veuille un conteneur vide, avec un nombre d'éléments prédéfinis, etc...

- Construction d'un conteneur vide :

On fait appel à un constructeur sans argument. Il ne comporte aucun élément :

```
list<float> lf ; // la liste lf est construite vide ; lf.size() vaudra 0 et
               // lf.begin() ==lf.end() ;
```

- Construction d'un conteneur avec un nombre donné d'éléments :

Il suffit d'ajouter un argument entier `n` au constructeur. Ce qui sera construit, ça sera un conteneur de `n` éléments.

```
list<float> lf(5) ; // lf construit avec 5 éléments de type float
vector<point> lp(5) ; // lp construit avec 5 éléments de type point
```

Maintenant, on veut en même temps initialiser les 5 éléments de `lf` et `lp` :

```
list<int> li(5, 999) ; // 5 éléments initialisés à 999
point a(3,8) ; // point est une classe
list<point> lp(5,a) ; // lp aura 5 éléments de type point avec les coordonnées 3,8
```

- Construction à partir d'une séquence :

On construit ici un conteneur à partir des éléments déjà présents dans un autre conteneur :

```
list<point> lp(6); //liste de 6points
...
vector<point> vp(lp.begin(),lp.end()); //on recopie les points de la liste lp
//dans le vecteur de points
list<point> lpi(lp.rbegin(),lp.rend()); //on inverse l'orde des points dans la liste lp
```

- Construction à partir d'un conteneur de même type :

On utilise un constructeur (classique) par copie :

```
vector<int> vi1 ; //vecteurs d'entiers
...
vector<int> vi2(vi1) ;
```

- Opérations sur les conteneurs :

Tous les opérateurs qui suivent sont valables pour les conteneurs list, deque et vector.

- ➔ La fonction membre assign(début,fin) : permet d'affecter à un conteneur existant les éléments d'une séquence définie par un intervalle.
- ➔ La fonction clear() : vide le conteneur de son contenu.
- ➔ La fonction swap(conteneur) : permet d'échanger le contenu de deux conteneurs de même type.

#### 9.4. Le conteneur vector :

Il reprend la notion de tableau en autorisant un accès direct à un élément quelconque. On peut accéder à un élément soit par accès direct soit par une façon plus classique à l'aide des opérateurs [].

- Accès par itérateur :

Par exemple : iv est un variable de type vector<int>::iterator. L'expression iv+i est alors tout à fait possible du fait que les itérateurs iterator et reverse\_iterator sont à accès direct.

Petit exemple :

```
vector <int> v(10);           //vecteur de 10éléments
vector <int>::iterator iv=v.begin(); //iv pointe sur le premier élém de v
...
iv=vi.begin(); *iv=0;       //place 0 dans 1er élém de vi
iv+=3; *iv=30;             //place 30 dans 4ème élém de vi
iv=vi.end()-2; *iv=70;     //place 70 dans 8ème élém de vi
```

- Accès par indice :

Si v est de type vector, l'expression v[i] est une référence de rang i, de sorte que les deux instructions suivantes sont équivalentes :

```
v[i] = ... ;                *(v.begin()+1) = ... ;
```

Il existe aussi une fonction membre at telle que v.at(i) soit équivalent à v[i]. Il génère une exception "out\_of\_range" en cas d'indice incorrect (ce que ne fait pas []). On peut réécrire l'exemple précédent :

```
v[0]=0;           //ou vi.at(0)=0 ;
v[3]=30;         //ou vi.at(3)=30 ;
v[7]=70;         //ou vi.at(7)=70 ;
```

Il est plus préférable d'utiliser les itérateurs que les indices pour des problèmes d'homogénéisation.

- Cas d'accès au dernier élément :

A l'aide de la fonction `back`, il est possible d'accéder au dernier élément d'un vecteur.

```
vector<int>v[10];
...
v.back()=25;
```

- Insertion et suppression :

Dans le conteneur `vector`, il existe des possibilités d'insertion et de suppression. On a alors :

- la fonction `push_back(valeur)` pour insérer un nouvel élément en fin ;
- la fonction `pop_back()` pour supprimer le dernier élément.

Exemple :

```
vector<int>v(5,99);
v.push_back(10); //ajoute 1élé de valeur 10 en fin de conteneur
v.push_back(20); //ajoute 1élé de valeur 20 en fin de conteneur
v.pop_back(); //supprime le dernier élément : v.size()=6;
```

Exemple reprenant tout ce qui a été vu avec les vecteurs :

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void affiche(vector<int>);
void main() {
    int i;
    int t[]={1,2,3,4,5,6,7,8,9,10};
    vector<int>v1(4,99);
    vector<int>v2(7,0);
    vector<int>v3(t,t+6);
    cout<<"V1 init = ";affiche(v1);
    for(i=0;i<v2.size();i++) v2[i]=i*i;
    v3=v2;
    cout<<"V2 init = ";affiche(v2);
    cout<<"V3 init = ";affiche(v3);
    v1.assign(t+1,t+6);cout<<"v1 apres assign : ";affiche(v1);
    cout<<"Dernier element de v1 : "<<v1.back()<<endl;
    v1.push_back(99);cout<<"v1 apres push_back : ";affiche(v1);
    v2.pop_back();cout<<"v2 apres pop_back : ";affiche(v2);
    cout<<"v1.size() : "<<v1.size()<<" v1.capacity() : "<<v1.capacity()<<" v1.max_size(): «
    "<<v1.max_size()<<endl;
    vector<int>::iterator iv;
    iv=find(v1.begin(),v1.end(),16);
    if(iv != v1.end()) v1.insert(iv,v2.begin(),v2.end());
    cout<<"v1 apres insert : ";affiche(v1);
}
```

```

void affiche(vector<int> v) {
    unsigned int i;
    for(i=0;i<v.size();i++) cout<<v[i]<<" ";
    cout<<endl;
}

```

### A l'affichage :

```

V1 init = 99 99 99 99
V2 = 0 1 4 9 16 25 36
V3 = 0 1 4 9 16 25 36
V1 après assign : 2 3 4 5 6 99
Dernier element de v1 : 6
V1 après push_back : 2 3 4 5 6 99
V2 après pop_back : 0 1 2 4 9 16 25
V1.size() : 6  v1.capacity() : 10  v1.max_size() : 1093741823
V1 après insert : 2 3 4 5 6 99

```

## 9.5. Le conteneur deque :

Il offre des fonctionnalités assez voisines de celles d'un vecteur. En plus d'offrir l'insertion, il offre également une insertion ou une suppression en début de conteneur.

On aura donc :

- front() pour accéder au premier élément ;
- push\_front(valeur) pour insérer un nouvel élément au début ;
- pop\_front() pour supprimer le premier élément.

### Exemple :

```

#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
void affiche(deque<char>);
void main() {
    char mot[]={"xyz"};
    deque<char> pile(mot, mot+3); affiche(pile);
    pile.push_front('a'); affiche(pile);
    pile[2]='+';
    pile.push_front('b');
    pile.pop_back(); affiche(pile);
    deque<char>::iterator ip;
    ip=find(pile.begin(),pile.end(),'x');
    pile.erase(pile.begin(),ip);affiche(pile);
}

void affiche(deque<char> p) {
    int i;
    for(i=0;i<p.size();i++) cout<<p[i]<<" ";
    cout<<endl;
}

```

Normalement à l'affichage, (si le compilateur accepte parce que moi il a pas voulu ☹ et je n'ai pas d'autre exemple) on devrait avoir :

```
x y z
a x y z
b a x +
x +
```

## 9.6. Le conteneur list :

Le conteneur list correspond au concept de liste doublement chaînée, ce qui signifie qu'on y disposera d'un itérateur bidirectionnel permettant de parcourir la liste à l'endroit ou à l'envers. Cette fois, les insertions ou suppressions vont pouvoir se faire quelle que soit la position de l'éléments. Petit désavantage : il ne dispose pas d'un itérateur à accès direct.

- Accès aux éléments existants :

Dans ce conteneur, il existe toujours les itérateurs iterator et reverse\_iterator, mais ils sont bidirectionnels. Si "it" désigne un tel itérateur, il est toujours possible de consulter l'élément pointé par "\*it" ou par : \*it = ... ;

On ne pourra pas l'incrémenter ou décrémenter d'une valeur quelconque mais bien par ++ ou --. Ainsi pour accéder une première fois à un élément, il faudra parcourir toute la liste. Pour se faire, on peut utiliser les fonctions front() et back() :

```
list<int> l();
...
if (l.front()==99) l.front()=0 ;
```

- Insertion et suppression :

On a toujours les possibilités des fonctions insert et reverse. On peut utiliser également des fonctions spécialisées d'insertion en début push\_front(valeur), à la fin push\_back(valeur) et de suppression en début pop\_front() et à la fin pop\_back().

Les deux types de suppressions :

- suppression des éléments de valeur donnée : remove (valeur) ;
- suppression d'éléments répondants à une condition : remove\_if(prédicat) ;

- Opérations globales :

Avec un conteneur list, il est possible d'intervenir sur le contenu d'une liste.

a) Tri d'une liste :

- ➔ `sort()` : trie la liste concernée en s'appuyant sur l'opérateur <
- ➔ `sort(prédicat)` : trie la liste concernée en s'appuyant sur le prédicat binaire "prédicat". Ex : `li.sort(greater<int>)` ;

b) Suppression des éléments en double :

- ➔ `unique()` : ne conserve que le premier élément d'une suite de valeurs consécutives égales.
- ➔ `unique(prédicat)` : conserve le premier élément d'une suite de valeur en satisfaisant le prédicat.

c) Fusion de 2listes :

- ➔ `merge(liste)` : fusionne "liste" avec la liste concernée en s'appuyant sur l'opérateur > en vidant "liste".
- ➔ `merge(liste,prédicat)` : fusionne "liste" avec la liste concernée, en s'appuyant sur le prédicat.

d) Transfert d'une partie d'une liste dans une autre :

- ➔ `splice(position,liste_or)` : déplace les éléments de "liste\_or" à l'emplacement "position".
- ➔ `splice(position,liste_or,position_or)` : déplace l'élément de "liste\_or" pointé par "position\_or" à l'emplacement "position". Position et position\_or peuvent être le début ou la fin de la liste.
- ➔ `splice(position,liste_or,debut_or,fin_or)` : déplace l'intervalle [début\_or,fin\_or] de liste\_or à l'emplacement position.

Exemple de quelques possibilités qu'offre le conteneur liste :

```
#include <iostream>
#include <list>
using namespace std;
void main() {
    void affiche(list<char>);
    char mot[]={"anticonstitutionnellement"};
    list<char>lc1(mot,mot+sizeof(mot)-1);
    list<char>lc2;
    cout<<"lc1 init : ";affiche(lc1);
    cout<<"lc2 init : ";affiche(lc2);
    list<char>::iterator il1,il2;
    il2=lc2.begin();
    for(il1=lc1.begin();il1!=lc1.end();il1++) {
        if(*il1!='t') lc2.push_back(*il1);
    }
    cout<<"lc2 apres : ";affiche(lc2);
    lc1.remove('t');
```

```

cout<<"l1 remove : ";affiche(l1);
if(l1==l2) cout<<"Les deux listes sont egales"<<endl;
l1.sort();
cout<<"l1 sort : ";affiche(l1);
l1.unique();
cout<<"l1 unique : ";affiche(l1);
}

```

```

void affiche(list<char> l) {
    list<char>::iterator il;
    for(il=l.begin();il!=l.end();il++) cout<<(*il)<<" ";
    cout<<endl;
}

```

### A l'affichage :

```

L1 init : a n t i c o n s t i t u t i o n n e l l e m e n t
L2 init :
L2 apres : a n i c o n s i u i o n n e l l e m e n
L1 remove : a n i c o n s i u i o n n e l l e m e n
Les deux listes sont egales
L1 sort : a c e e e i i i l l m n n n n n o o s u
L1 unique : a c e i l m n o s u

```

## 9.7. Les adaptateurs de conteneur : queue, stack, priority\_queue :

- L'adaptateur Stack :

Le patron stack est destiné à la gestion de pile. On y retrouve les fonctions membres suivantes :

```

empty() : fournit true si la pile est vide
size() : fournit le nombre d'éléments de la pile
top() : accès à l'information situé au sommet de la pile
push(valeur) : place valeur sur la pile
pop() : fournit la valeur de l'élément situé au sommet en le supprimant de la pile

```

Exemple de création :

```

stack<int>,vector<int>>q;
q.size();
q.top();
...

```

- L'adaptateur Queue :

Il est destiné à la gestion de files d'attentes ou de piles. On retrouve les mêmes fonctions membres que pour stack sauf top() mais il y a en plus :

```

front() : accès à l'information située en tête de la queue
back() : accès à l'information située en fin de la queue

```

- L'adaptateur `Priority_queue` :

Il ressemble à une file d'attente dans laquelle on introduit toujours des éléments à la fin. On retrouve les quelques fonctions membres suivantes :

`empty()` : fournit `true` si la queue est vide

`size()` : fournit le nombre d'éléments de la queue

`push(valeur)` : place valeur dans la queue

`top()` : accès à l'information placée en tête de la queue qu'on connaît ou modifier

`pop()` : fournit l'élément situé en tête de la queue en le supprimant

## 9.8. Les conteneurs associatifs :

Ils ont pour principale vocation de retrouver une information en fonction de sa valeur ou d'une partie de sa valeur nommée clé.

### 9.8.1. Le conteneur `map` :

Composé de deux parties : une clé et une valeur.

Soit la déclaration : `map<char, int> m ;`

Crée un conteneur de type `map` dans lequel les clés sont de type `char` et les valeurs de type `int`.

Et donc, l'instruction : `m['S'] = 5 ;` insère dans `m` un élément formé de l'association de la clé 'S' et de la valeur 5.

On peut toujours utiliser les fonctions `begin()` et `end()` pour le début et la fin.

Pour le reste sur le conteneur `map`, je sais pas trop ce qu'on a bien pu voir au cours, alors je vous renvoie au Claude Delannoy page 428.

### 9.8.2. Le conteneur `multimap` :

Ici une même clé peut apparaître plusieurs fois.

On peut créer un conteneur de ce type de la façon suivante :

```
multimap<char,int>m ;
```

Pour les fonctions membres qui pourront être utilisées, je vous renvoie dans le Claude Delannoy page 438.

### 9.8.3. Le conteneur set :

C'est un cas particulier du conteneur map dans lequel aucune valeur n'est associée à une clé. Les éléments d'un conteneur set ne sont donc plus des paires, ce qui en facilite la manipulation.

```
set<int> e(...) ; //ensemble d'entiers  
set<int> ::iterator ie ; //itérateur sur un ensemble d'entiers
```

Pour le reste (comme je ne sais pas trop de quoi nous avons parlé dans ce chapitre au cours), je vous renvoie encore une fois dans le Claude Delannoy page 441.

### 9.8.4. Le conteneur multiset :

C'est un conteneur set dans lequel on autorise plusieurs clés équivalentes.

```
multiset<int>e(...);
```

D'autres informations dans le Claude Delannoy page 442.