Faculté d'Ingénieurs en Informatique, Multimédia, Systèmes, Télécommunication et Réseaux

Master en Génie Logiciel

C et C++ avancée

Préparé par Elie MATTA

Elie

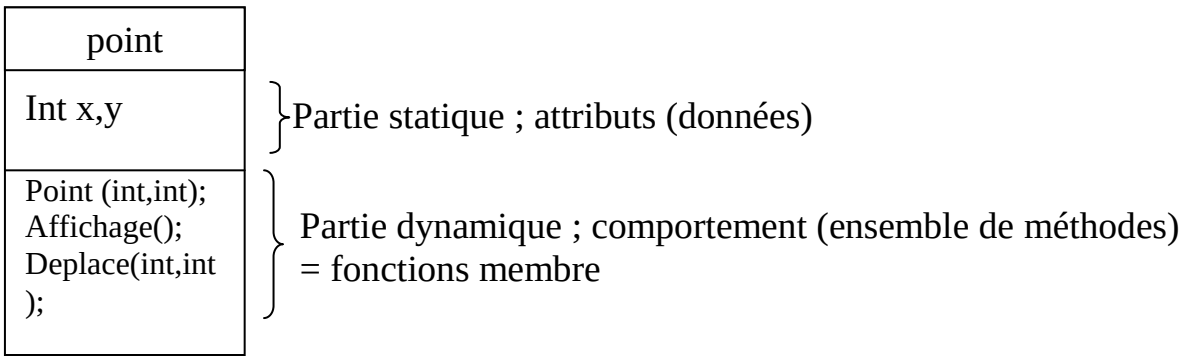
Cours et TD de la matière C et C++ avancée

Rappel:

Les concepts de la programmation orientée objet

1- Modélisation a l'aide des classes et objets
2- L'activation de méthodes a l'aide de l'envoi de messages
3- La construction par affinage par l'intermédiaire d'héritage

1-Classes et objets
Exemple 1 :



Les propriétés de fonctions membre :

1. **La surdefinition de fonctions**
2. **Arguments par defaut**
3. **Les fonctions membre inline :**

Les fonctions membre inline sont plus rapide car il n'y a pas retour a la mémoire car on invoque la fonction chaque fois qu'on l'a besoins mais l'inconvénient c'est qu'il y aura congestion dans le programme

4. **Mode de transmission des objets : arguments d'une fonction**
   4.1 Transmission des arguments par défaut se fait par valeur
   4.2 Transmission par adresse
   4.3 Transmission par référence
5. **Fonctions membres statiques**
6. **Fonctions membres constants**

Les fonctions membres constats peuvent appeler seulement aux méthodes qui sont constantes

7. **Pointeurs sur des fonctions membre**

Ils doivent avoir le même type d'argument
En général :

```
int f1(char,double) ;
int f2(char,double) ;
```

On peut définir un pointeur :

```
int(*adf)(char,double) ;
adf=$f1; //7eme propriete
(*adf)('c',12.5); //ca sera egale a f1('c',12.5);
```


Exemple 1 : Application

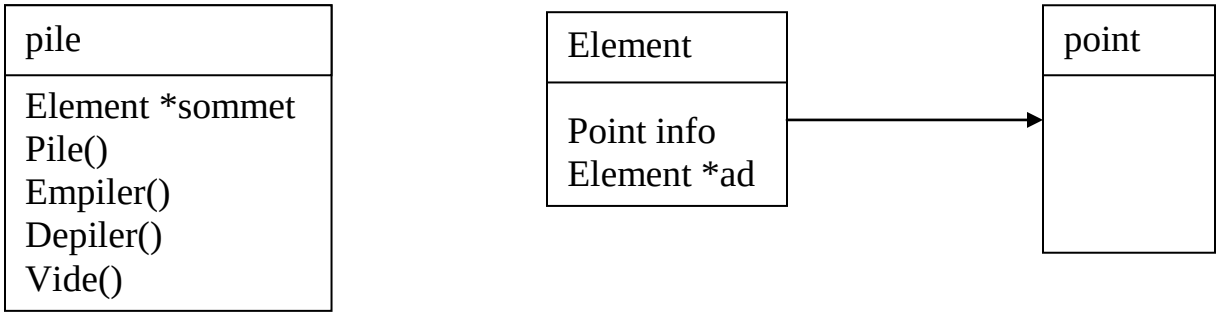
```
// tp1 - c et c ++.cpp :
//** Exemple 1 - CTRL+F5 TO RUN

#include "stdafx.h"
#include <iostream>
using namespace std; //le workspace std contient la librairie iostream et autres librairies

class Point{
	//toutes les attributs et methodes dans c++ sont private par defauts au contraire du
java, c-a-d encapsulation des donnees
	//ou tous les attributs sont definis explicitement private

	int x,y;
	static int np; //cette variable est statique on ne la met pas dans le constrcuteur et
on est la declare au dehors de la classe
	int f1(char,double);

public:
	Point (int,int);
	void affiche();
	void deplace(int,int);
	Point (int x=0,int y=0);
	~Point(); //destructeur
	Point(); //surdefinition (1ere propriete)
	Point (Point &); // passage de parametre par reference = ici c'est un constructeur par
recopie (1ere propriete)

	void affiche(char*=" ");

	void Point::affiche(char *message){ //(3eme propriete) on ne met pas inline ici car
nous somme dans la classe
		cout <<message<<"coord:"<<x<<";"<<y<<"\n";
	}

	bool coincide(Point); //4eme propriete - Transimission par valeur

	bool coincide(point*);//4eme propriete - Transimission par argument

	bool coincide(point &);//4eme propriete - Transimission par reference

	static int compte();

	void dep-hor(int dx){ //7eme propriete
		x+=dx;
	}
	void dep-ver(int dy){//7eme propriete
		y+=dy;
	}
};//fin classe
	Point::Point(int x,int y){ //Constructeur (1) - les :: definits que la methode Point
(int x,int y) est une methode membre de la classe Point
		this->x=x;                    //le constructeur est appele automatiquement lors
de la creation d'un objet
		this->y=y;
	}

	void Point::affiche() const { //5eme Propriete
		cout <<"coordonnees: " <<x<<";"<<y<<endl; //endl = \n
		cout <<"nombre d'objets:"<<Point::np<<"\n";//Point::np car np est statique
	}

	void Point::deplace(int dx, int dy){
```

```
		x+=dx;
		y+=dy;
	}

	Point::~Point(){ //appel du destructeur seulment lors de la fermeture du programme,
		cout <<"appel de destructeur\n"; //on peut l'utiliser aussi pour detruire les
elements d'un tableau
	}

	Point::Point(){ //Constructeur (2) - surdefinition (1ere propriete)
		x=y=0;
	}

	Point::Point(Point &b){ //(1ere propriete)
		x=b.x;
		y=b.y;
	}
	void Point::affiche(char *message){ //(Affiche (2))
		cout <<message<<"coord:"<<x<<";"<<y<<"\n";
	}
inline void Point::affiche(char *message){ //(3eme propriete)
		cout <<message<<"coord:"<<x<<";"<<y<<"\n";
	}
bool Point::coincide(Point p){ //(4eme propriete) Creation d'un objet temporaire qui met a=4
et y=5 (Les arguments de f) (Cette methode est nomme transmission par valeur)
	return x==p.x && y==p.y;
}

bool point::coincide(point *ad){
	return x==ad->x && y==ad->y;
}

bool point::coincide(point &p){ //Transmission par reference
	return x==p.x && y==p.y;

	int Point::compte(){ //(5eme propriete)
		return np;
	}
}
	int Point::np=0; //initialisation de np statique en dehors de la classe et methodes

	void main (){
		Point a(20,5),b; //on a creer le point a avec le premier constructeur, et le
point b par le constructeur par defaut dont x et y de b =0
		Point c(a); //constructeur par recopie
		a.affiche();
		a.deplace(3,2);
		a.affiche();
		a.affiche("Point a"); //elle appel a la methode Affiche (2)
		Point d(2,3),e,f(4,5); //(4eme propriete)
		d.affiche();
		e.affiche();
		f.affiche();
		d.deplace(1,2);
		if (d.coincide(f)){ //dans la premier methode Transmission par valeur on cree un
objet temporarire
			cout<<"les point sont identique\n";
		}
		else{
			cout<<"les point sont distinct\n";
		}

if (d.coincide(&f)){
```

```
                  cout<<"les point sont identique\n"; //dans cette methode on prend
l'addresse (par exemple l'addresse de f est 1EA2)- generalement si on veut changer la donne
on utilise la transmission par addresse ou par reference
            }                                                   //pour les tableaus on utlise la
transmission par address
            else{ //la difference entre la transmission par adresse et par reference est la
declaration et l'usage des methodes, pour la transmission par adresse on utlisie la methode
if (d.coincide(f)
                  cout<<"les point sont distinct\n";
            }
point g(2,3);
const point h;
g.affiche(); //g peut utiliser tous les methodes constantes et non constantes
g.deplace(2,3);
h.affiche();
h.deplace(); //impossible car h peut utiliser seulment les methodes constantes car h est
declare constantes

void (Point::*adf)(int); //7eme propriete
adf=&Point::dep-hor;

a.*adf(1); //a.*adf c'est a dire le contenue de adf c'est-a-dire dep-hor
adf=&Point::dep-ver;
a.*adf(2);
      }
```

Exemple 2 :

LIFO – Last in first out




```
//** Exemple 2

#include "stdafx.h"
#include <iostream>
using namespace std;

class Point{
      int x,y;

public:
      Point (int x=0;int y=0){
            this->x=x;
            this->y=y;
      }
      void affiche(){
            cout <<"coord"<<x<<";"<<y<<"\n";
      }
};

      struct Element{ //element qui est dans la pile
            Point info; //on doit mettre cette structure apres la classe point car elle
contient creation d'un objet point si on l'a met avant la classe Point on aura une erreur
            Element *suivant;
      };
```

```
class Pile{
            Element *sommet;
public:
      Pile();
      void empiler(point);
      Point depiler();
      bool vide();
      };

      Pile::Pile(){
            sommet=NULL;
      }
      void Pile::empiler(Point p){
            Element *nouveau;
            nouveau ->info=p;
            nouveau ->suivant=sommet;
            sommet = nouveau; //affectation des adresses
      }
      Point Pile::depiler(){
Point a =info;
sommet = sommet->suivant;
return a;
      }

      bool Pile::vide(){
            return sommet=NULL;
      };

      void main(){
            Point a(2,3),b(5,6),c;
            Pile p;
            p.empiler(a);
            p.empiler(b);
            p.empiler(c);
            while(!p.vide()){
                  p.depiler().afficher();

      }
```

Chapitre 2 : Les fonctions amies :

1- Introduction :

La P.O.O impose l'encapsulation de données c.-à-d. les membres privées ne sont pas accessibles qu'aux fonctions membre de cette classe. Ce principe interdit à une fonction menu d'une classe d'accéder a des données privées d'une autre classe. La solution en C++ c'est les fonctions amies. En effet il est possible lors de la définition d'une classe de déclarer qu'une ou plusieurs fonctions (extérieure de cette classe) comme fonctions amies (friend)
Une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.

2- Fonctions amies

2-1 Fonction indépendante, amie d'une classe.

Exemple :

```
#include "stdafx.h"
#include <iostream>
using namespace std;

class Point{
int x,y ;
public :
Point(int =0, int=0) ;
void affiche() ;
friend bool coincide(Point,Point); //independante de la classe et peut accede au argument
prives de la classe
}

Point :: Point(int x, int y){
this -> x=x ;
this -> y=y ;
}

void Point :: affiche(){
      cout <<"coord"<<x<<";"<<y<<endl;
}

bool coincide (Point a,Point b){ //coincide n'est pas une fonction membre
      return a.x==b.x && a.y==b.y; //statique c'est pourquoi on l'ecrit comme
}//ca

void main (){
      Point a(1,2),b(3,4);
      Point *adp=new Point(5,6);
      a.affiche();
      b.affiche();
      adp->affiche();
      if (coincide(a,*adp))
                                    //*adp est le contenu de adp
                                    // par exemple: int A=5;
                                                    // int *p=&A;
                                                //      *p = 5;
}
```

2-2 Fonction membre d'une classe, amie d'une autre classe

Exemple :

```
Class B ;
Class A{
Friend void B :: f(char,A);
}

Class B{
Void f(char,A); //ne peut être pas déclarer dans la méthode inline
}
B :: f(char c, A a){
…
}
```

2-3 Fonction amie de plusieurs classes

```
Class A ;
Class B{
…
Friend void f(A,B) ;
}
Class A{
..
Friend void f(A,B);
}
Void f(A a, B b){
…
}
```

2-4 Toutes les fonctions d'une classe sont amies d'une autre classe pour dire que toutes les fonctions membres de B sont amies de A

TD1 :

Vecteur
-float v[3] -const par def v[i]= -const -affiche()

Matrice
-float m[3][3]; -const par def m[i][j] -const -Affiche() -vecteur PMV (vecteur)

To create a new project we should create first an empty project (File -> Project -> Visual C++ -> Empty project) then we should click on Project in the taskbar above and add new item(Project -> Add new item -> C++ file (.cpp)) and when debugging we should choose -> Start Without Debugging

Solution TD1:

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

class Vecteur;
class Matrice{
	float m[3][3]; //allocation statique
public:
	Matrice(){ //inline
		for (int i=0;i<3;i++)
			for (int j=0;j<3;j++)
				m[i][j]=0;
	}
	Matrice (float t[3][3]){ //quand on a deux dimensions on ne peut pas declarer le
	for (int i=0;i<3;i++)   // tableau sans aucune dimension,
			for (int j=0;j<3;j++)// on doit mettre au minimum une dimension
				m[i][j]=t[i][j];
	}

	void affiche (){
		for (int i=0;i<3;i++){
			for (int j=0;j<3;j++)
				cout <<m[i][j]<<"\t";
		cout <<"\n";
	}

	Vecteur PMV(Vecteur); //on ne peut pas faire inline ici car on ne connait pas encore
les paramatres
}; //fin classe Matrice

class Vecteur{
	float v[3];

public:

	Vecteur (){
		for (int i=0;i<3;i++)
			v[i]=0;
	}

	Vecteur (float p[3]){
		for (int i=0;i<3;i++)
			v[i]=p[i];
		}
		void affiche(){
			for (int i=0;i<3;i++)
				cout <<v[i]<<"\t";
			}

			friend Vecteur Matrice::PMV(Vecteur);
		};

		Vecteur Matrice :: PMV (Vecteur a){
			Vecteur res;
			for (int i=0;i<3;i++)
				for(int j=0;j<3;j++)
					res.v[i]+= m[i][j] * a.v[j];
				return res;
			}

			void main(){
				float tv[3];
				float tm[3][3];
```

```
        cout <<"donner un vecteur de 3 composant \n";
        for (int i=0;i<3;i++)
              cin >>tv[i];
              cout <<"donner une matrice 3x3 \n";
              for (int i=0;i<3;i++)
                    for (int j=0;j<3;j++)
                          cin >>tm[i][j];

Vecteur a(tv);
Matrice p(tm);

a.affiche();
p.affiche();

Vecteur r;
r=p.PMV(a); //c'est comme ca qu'on utilise la methode friend dans main
r.affiche();

}
```

TD2:

Point	Cercle
-int x,y; définir une variable pour compter le nombre de points - const avec paramètre par défaut x=y=0 -affiche() -static compte() qui retourne le nombre de point crées -distance () retourne la distance entre 2 points -coincide() vrai si 2 points se coïncident	-point p centre du cercle -int r -constructeur par defaut -affiche() qui affiche les coordonnées du centre et du rayon -coincide() vrai si 2 cercles se coïncident

+Une fonction indépendante DansDisque() qui retourne vrai si un point appartient au disque du cercle

Solutions TD2 :

```
# include <iostream>
# include<math.h>
using namespace std;
class point{
      int x,y;
      static int nb;
public:
      point(int x,int y){
            this->x=0;
            this->y=0;
            nb++;
      }
      point(point &p){
            this->x=x;
            this->y=y;
            nb++;
      }
      void affiche(){
            cout<<"les coordonnes sont:"<<x<<";"<<y<<"\n";
```

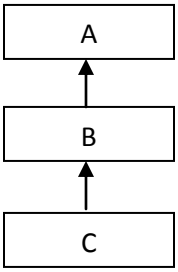
```
	}

	static int compte(){
		return nb;
	}
	int distance(point b){
		double dx,dy;
		dx = (this->x-b.x)*(this->x-b.x);
		dy = (this->y-b.y)*(this->y-b.y);
		return sqrt(dx+dy);
	}
	bool coincide(point b){
		return(this->x==b.x && this->y==b.y); //on peut mette x==b.x
	}
};
class cercle{
	point p;
	int r;
public:
	cercle(point p,int r=0){
		this->r=r;
		this->p=p;
	}
	void affiche(){
		this->p.affiche();
		cout<<"le rayon est:"<<r<<"\n";
	}
	bool coincide(cercle c){
		return (p.coincide(c.p) && r==c.r);
	}
	friend bool DansDisque(cercle,point);
};
bool DansDisque(cercle c,point a){
	return a.distance(c.p)<=c.r;
}

void main(){
	point p(2,3);
	point b(1,2);
	cercle c(p,10);
	c.affiche();
	if(DansDisque(c,b))
		cout<<"point b apparteint au cercle\n";
	else
		cout<<" le point n'appartient pas\n";
}
```

Chapitre 3 : L'héritage

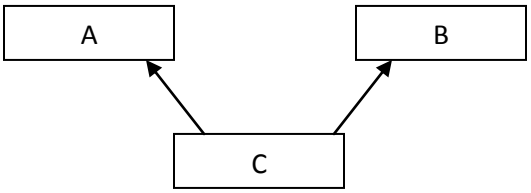
Constitue l'un des fondements de la P.O.O, il est à la base de réutilisation de composants logiciels. En effet, il nous autorise à définir une nouvelle classe dite « dérivée » a partir d'une classe existante dite de « base » la classe dérivée hérite toutes les potentialités de la classe de base, tout en lui ajoutant de nouvelles

Plusieurs classes peuvent être dérivées d'une même classe de base

1-L'héritage n'est pas limite a un seul niveau



C++ autorise l'héritage multiple dans lequel une classe peu être dérivée de plusieurs classes de base



2-Mise en œuvre de l'héritage en C++

```
# include <iostream>
# include<math.h>
using namespace std;
class Point{
	int x,y;

public:
	Point(int x=0,int y=0){
		this->x=x;
		this->y=y;

	}

	void affiche(){
		cout<<"les coordonnes sont:"<<x<<";"<<y<<"\n";
	}
	void deplace (int dx,int dy){
		x+=dx;
		y+=dy;
	}
	~Point(){
		cout<<"appel destructeur \n";
	}
};
```

```
class PointCol:public Point{ //pointcol derive de la classe point, public signifie que
short couleur; // pointcol peut acceder seulment au membres publics de la classe point

public:
	PointCol(int =0,int =0, short=1);
	void affichec();
};

PointCol :: PointCol (int x, int y, short c) : Point(x,y){ //point (x,y) est l'heritage de
la superclasse
	couleur =c;
}

void PointCol :: affichec(){
	affiche();//c'est comme ca qu'on utlise la methode affiche de la classe point
	cout <<"couleur"<<couleur<<endl;
}


void main(){
	PointCol ac(2,3,5),bc,d(3,4),e(7);
	ac.affiche(); //coord: 2;3
	ac.affichec(); //coord 2;3 couleur :5
	ac.deplace(1,1);
	ac.affiche();
}
```

3-Hiérarchie des appels de constructeurs:

1- Appel du constructeur de la classe de base(Point)
2 - Appel du constructeur de la classe derivee(PointCol)

Destructeur: (marche inverse)

1- Appel destructeur PointCol
2- Appel destructeur Point

4-Redéfinition des fonctions membre:

Une fonction membre de la classe dérivée qui a même signature qu'une fonction de la classe de base

```
void PointCol::affiche(){
	Point::affiche();
	cout<<"couleur:"<<couleur<<endl;
}
```

donc dans main:

ac.Point::affiche();//appel de la fonction affiche de la classe Point

5-Control des accès:

1- Toute fonction membre d'une classe dérivée a accès aux membres publics de la classe de base.
2- Tout objet de la classe dérivée à accès a ses membres publics, ainsi qu'aux membres publics de la classe base.

Lors de la conception de la classe dérivée, on peut choisir de rendre inaccessible les membres publics de sa classe de base.

class pointcol:private Point{
...
}
ac.deplace(1,1)// erreur acces non garanti a la classe point

6-Les membres protégés:

Autre que public et private, il existe un troisième statut : protected
Les membres protégés restent inaccessible aux utilisateurs de la classe (de l'extérieur de la classe)
Mais il serait accessible aux membres de la classe dérivée seulement

class point{
protected:
int x,y;
...
}

Exemple :

```
# include <iostream>
# include<math.h>
using namespace std;
class Point{
	int x,y;

public:
	Point(int x=0,int y=0){
		this->x=x;
		this->y=y;

	}

	void affiche(){
		cout<<"les coordonnes sont:"<<x<<";"<<y<<"\n";
	}
	void deplace (int dx,int dy){
		x+=dx;
		y+=dy;
	}
	~Point(){
		cout<<"appel destructeur \n";
	}
};

class PointCol:private Point{ //pointcol ne peut pas acceder meme aux fonction declarees
public dans point
short couleur;
public:
	PointCol(int =0,int =0, short=1);
	void affiche();
};

PointCol :: PointCol (int x, int y, short c) : Point(x,y){ //point (x,y) est l'heritage de
la superclasse
	couleur =c;
```

```
}

void PointCol :: affiche(){
      Point::affiche();
      cout <<"couleur"<<couleur<<endl;
}


void main(){
      PointCol ac(2,3,5),bc,d(3,4),e(7);
      ac.affiche(); //dans main on ne peut pas appeler une fonction public de la classe
point mais dans ce cas affiche de pointcol appel affiche de point donc l'acces est permis
      ac.deplace(1,1); //rejete par le compilateur car on peut pas l'acceder
      ac.Point::affiche();//rejete par le compilateur car on peut pas l'acceder
}
```

TD3

Compte
int Id ; double debit,credit ; -const par defaut debit=credit=0 -affiche() -ajout(double) -retire(double) -solde()//retourne le solde credit-debit

CompteEpargne

char *nom ;
double taux ;
-const par def
-affiche() //id,deb,nom,taux
-interet()//retourne intérêt solde*taux
-update()//credit +interet

Solutions TD3 :

compte.h:

```
#pragma once
using namespace std;
# include <iostream>
using namespace std;
class compte1{
      int ID;
      double debit,credit;//private par defaut
public:
      compte(int ID=0, double=0.0,double= 0.0);
      void affiche();
      void ajout(double);
      void retirer (double);
      double solde();
      double getc();
      void setc();
};
```

compte.cpp:

```
#include "compte1.h"
compte1::compte1(int ID, double debit, double credit)
{
	this ->ID=ID;
	this -> debit=debit;
	this-> credit=credit;
}

void compte1::affiche()
{
	cout<<" ID:"<<ID<<endl;
	cout<<"Debit:"<<debit<<endl;
	cout<<"Credit:"<<credit<<endl;
}
void compte1::ajout(double S){
	credit+=S;
}
void compte1::retirer(double S){
	debit+=S;
}
void compte1::solde(){
	return credit-debit;
}
	double getc(){
		return credit
	}
	void setc(){
		this-> credit=credit;
	}
```

CompteEpargne.h:

```
#pragma once
# include"compte1.h";
class CompteEpargne: public compte1{
	char* nom;
	double taux;
public:
	CompteEpargne(int=0,double=0.0, double=0.0,char*="",double=0.0);
	void affiche();
	double interet();
	void update();
};
```

CompteEpargne.cpp:

```
#include "CompteEpargne.h"

CompteEpargne::CompteEpargne(int ID, double debit, double credit, char* nom, double
taux):compte1(ID,debit,credit)
{
	this-> nom=nom;
	this->taux=taux;
}

void CompteEpargne::affiche()
{
	cout<<"nom:"<<nom<<endl;
	cout<<"taux:"<<taux<<endl;
	compte1::affiche();
}
double CompteEpargne::interet(){
	return solde()*taux;
}
void CompteEpargne::update(){
	setc(getc()+interet());
}
```

Main :

```
#include "CompteEpargne.h"
void main(){
	CompteEpargne client(100,0.0,1000.0,"nom",0.025);
	client.affiche();
	client.ajout(5000);
	client.retirer(2000);
	client.affiche();
	cout<<"interet:"<<client.interet()<<endl;
	cout<<"nouvo solde:"<<client.solde()<<endl;
}
```

TD4 :

$$m(\mathbb{R}^n)(\mathbb{R}^n) = (\mathbb{R}^m)$$

1ere methode :

Un vecteur de n composants

```
Double *v
V=new double ;
```

V est l'adresse de la case mémoire

Int n ;
V=new double [n] ;

Double *mat ;
Mat=new double [m*n] ; //un seul block

Adresse du premier élément

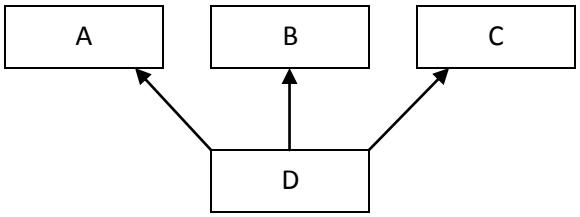
mat[i][j]= (mat +i*n+j)

Solutions TD4:

```
#include <iostream>
using namespace std;
class vecteur;
class matrice{
	int m,n;
	double *mat;

public:
	matrice(int m,int n){
		this->m=m;
		this->n=n;
		mat=new double [m*n];
	}
	matrice(int m,int n,double *t){
		this->m=m;
		this->n=n;
		mat=new double[m*n];
		for (int i=0; i<m*n;i++)
			*(mat+i)=0;
	}

	matrice(double *t,int m,int n){
		this ->m=m;
		this ->n=n;
		mat=new double[m*n];
		for(int i=0;i<m*n;i++){
		*(mat+i)=*(t+i); //mat[i]=t[i]
		}

		vecteur pmv(vecteur);
	};

	class vecteur{
		int n;
		double *v;
	public:
		vecteur(int n){
```

```
	this ->n=n;
	v=new double[n];
	for (int i=0;i<n;i++)
		*(v+i)=0.0;
}

vecteur (int *t,int n){
	this ->n=n;
	v=double[n];
	for (int i=0;i<n;i++)
		*(v+i)=*(t+i);
}
void affiche(){
	cout <<"affichage du vecteur \n";
	for(int i=0;i<n;i++){
		cout<<*(v+i)<<"\t";
		cout<<"\n";
	}

	friend vecteur matrice::pmv(vecteur);
}; //fin vecteur

vecteur matrice ::pmv(vecteur w){
	vecteur res(m);
	for (int i=0;i<m;i++)
		for (int j=0;j<m;j++)
			res.*(v+i)+=*(mat+i*n+j)*w.*(v+j);
	return res;
}

void main(){
	double tv[]={6,5,2.5,1};
	double tm[3][4]={{1,7,2,3},{4,10,12,5},{8,9,11,7}};
	vecteur w(tv,4);
	matrice M((double*)tm,3,4);
	vecteur res(3);
	res = M.pmv(w);
		res.affiche();
}
```

Chapitre 4 : L'heritage multiple



1-C++ a introduit l'héritage multiple :

La plupart de choses que nous avons dites à propos de l'héritage simple se généralisent au cours de l'héritage multiple

Un certain nombre d'information doivent être introduites

- Comment peut-on exprimer l'héritage multiple dans la classe dérivée
- Appel des constructeurs et de destructeur, ordre, transmission d'informations etc.

Exemple :

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

class point{
	int x,y;
public:
	point(int x=0,int y=0){
		this->x=x;
		this->y=y;
		cout <<"appel const point\n";
	}

	~point(){
		cout<<"appel dest point\n";
	}
	void affiche(){
		cout<<"coord:"<<x<<";"<<y<<endl;
	}
};

class couleur{
	short coul;
public:
	couleur(short coul=1){
		this->coul=coul;
		cout<<"appel const couleur\n";
	}
	~couleur(){
		cout<<"appel dest couleur\n";
	}
	void affiche(){
		cout<<"couleur:"<<coul<<endl;
	}
};

class pointcol:public point,public couleur{
public:
	pointcol(int x=0,int y=0,short coul =1): point(x,y),couleur(coul){
		cout<<"appel const pointcol \n";
```

```
	}
	~pointcol(){
		cout<<"appel dest pointcol\n";
	}
	void affiche(){
		point::affiche();
		couleur::affiche();
	}
};

void main(){
	pointcol pc(2,3,5);
	pc.affiche();
	pc.point::affiche();
	pc.couleur::affiche();
}
```

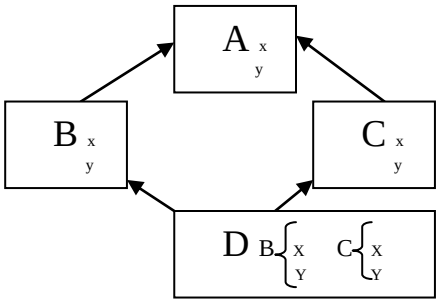
NB : L'ordre d'appel de constructeurs :

- Constructeur des classes de base dans l'ordre où les classes de base sont déclarées dans la classe dérivée : const point puis const couleur
- Const de la classe dérivée

L'ordre d'appel de destructeurs est dans l'ordre inverse : pointcol, couleur, point

2-Pour régler les conflits : Les classes virtuelles

Prenons le cas d'héritage multiple suivant :



On peut dire que D hérite deux fois de A. Alors les membres de A vont apparaitre deux fois dans D. Les fonctions de A n'ont pas réellement dupliquées. Les membres donnes(x et y) ils seront effectivement duplique. On distinguera dans D:
A ::B ::x de A ::C ::x
Ou éventuellement si B et C ne possèdent pas de membre x
B ::x de C ::x
Nous pouvons demander à C++ de n'incorporer qu'une seule fois les membres de A dans D, pour cela il faut préciser dans les déclarations de B et C que A est virtuelle
Class B : public virtual A{
…
}
Class C: public virtual A{
…
}

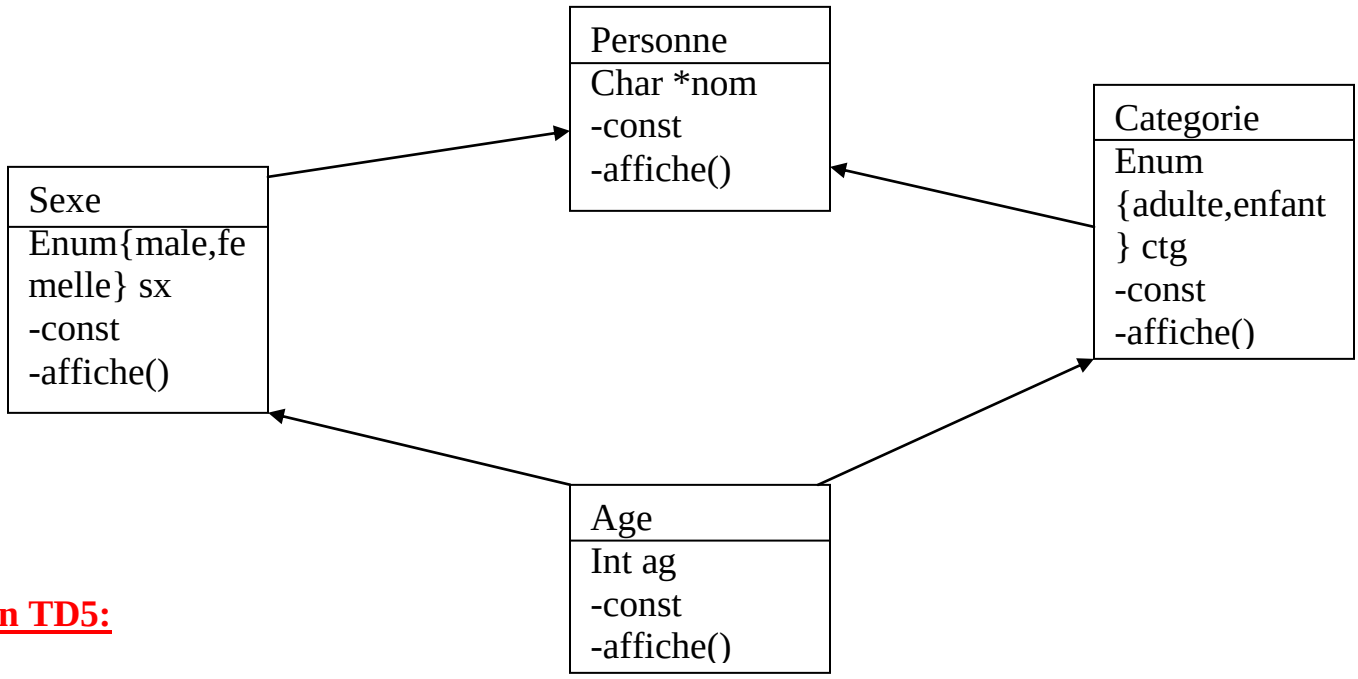
Il est nécessaire que le constructeur de la classe déviée puisse préciser les informations à transmettre au constructeur de la classe virtuelle
D(…,int x,int y) : B(…,x,y), C(…,x,y), A(x,y){
…
}
Appel de constructeurs
A,B,C,D

Enum jours{ lundi,mardi,…,dimanche} ;
Enum jours j ;

J= mardi ; ou j=1

Enum jours{lundi=1,mardi,…,dimanche} ;
Cout<< « j= »<<j ; j=1
Switch(j){
Case lundi:cout<<"lundi";
Break;

TD5:

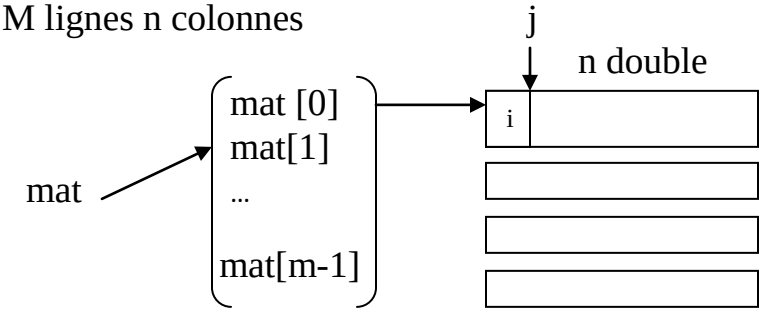


Solution TD5:

```
#include <iostream>
using namespace std;
class Personne{
      char* nom;
public:
      Personne(char *nom="")
      {
            this->nom=nom;
      }

      void affiche()
```

```
	{
		cout<<"nom:"<<nom<<endl;
	}
};

enum sex{male,female};

class Sexe:public virtual Personne{
	enum sex sx;
public:
	Sexe(char* nom="", enum sex sx=male):Personne(nom)
	{
		this->sx=sx;
	}

	void affiche()
	{
		Personne::affiche();
		switch(sx)
		{
		case male: cout <<"sexe: male"<<endl;break;
		case female: cout <<"sexe:female"<<endl;break;
		}
	}
};

enum Catg{enfant ,adulte};
class categorie:public virtual Personne{
	enum Catg ct;
public:
	categorie(char* nom="", enum Catg ct=enfant):Personne(nom){
		this-> ct=ct;
	}
	void affiche(){
		Personne::affiche();
		switch(ct){
		case enfant:cout<<"cat:enfant"<<endl;break;
		case adulte: cout<<"cat:adulte"<<endl;break;
		}
	}
};
class Age:public Sexe,public categorie{
	int ag;
public:
	Age(char* nom="",enum sex sx=male,enum Catg ct=enfant, int
ag=12):Sexe(nom,sx),categorie(nom,ct),Personne(nom){
		this->ag=ag;
	}
	void affiche()
	{
		Sexe::affiche();
		categorie::affiche();
		cout<<"Age:"<<ag<<endl;
	}
};
void main ()
{
	Age p("joseph",male,enfant,20);
	p.affiche();
}
```

Creation d'un tableau dynamique mat a 2 dimensions
M lignes n colonnes



double * * un tableau de pointeur contenant des pointeurs de type double
double * * mat ;

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

class Matrice{
	int m,n;
	double * *mat;
public:
	Matrice (int m,int n){
		this->m=m;
		this->n=n;
		mat=new double *[m];
		for (int i=0;i<m;i++)
			mat[i]=new double [n];
		for(int i=0;i<m;i++)
			for(int j=0;j<n;j++)
				*(mat[i]+j) = 0.0; // mat[i] est le premier membre de la ligne et j
est le nombre de la colonne
	}
	Matrice (int m,int n,double * *mat){
		this ->m=m;
		this ->n=n;
		this ->mat=mat;
	}
	//et le reste est le meme
```

TD6:

Sans utiliser les fonctions de <string.h>
Définir une classe chaine offrant des possibilités plus proches d'un véritable type string

Chaine
Int lg Char *adr Public: -Chaine() -Chaine (char*) -Chaine(Chaine &) par recopie - ~Chaine() //delete adr -void affiche() -bool equals(Chaine &) -Chaine concat(Chaine &)

Solution TD6 :

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

class chaine{
	int lg;
char *adr;

public:
	chaine (){
		lg=0;
		adr= NULL;
	}

	chaine (char *p){
		lg=0;
		for(char*q=p; *q; q++)// char*q est le contenu de la premiere case de p, on peut
mettre *q!0 a la place de *q ;
		lg++;// on aura la longeur de la chaine
		adr=new char[lg];
		for(int i=0; i<lg; i++)
			*(adr+i)=*(p+i);
	}
	chaine(chaine &ch){
		lg=ch.lg;
		adr= new char[lg];
		for(int i=0; i<lg; i++)
			*(adr+i)=*(ch.adr+i);
	}
	~chaine(){
		delete adr;
	}
	void affiche(){
		for(int i=0; i<lg; i++)
			cout<<*(adr+i);
		cout<<endl;
	}
	bool equals (chaine &ch){
		if(lg!=ch.lg)
			return false;
		for(int i=0;i<lg; i++){
			if (*(adr+i)!=*(ch.adr+i))
				return false;
			return true;
		}
	}

	chaine concat(chaine &ch){
		chaine res;
		res.adr=new char[res.lg=lg+ch.lg];
		for(int i=0;i<lg;i++)
			*(res.adr+i)=*(adr+i);
		for(int i=0;i<ch.lg;i++)
			*(res.adr+lg+i)=*(ch.adr+i);
		return res;
	}
};
void main(){
	chaine ch1;
	chaine ch2("bonjour");
	chaine ch3(" a tous");
	chaine ch4(ch3);
```

```
	chaine ch5= ch2.concat(ch3);
	ch5.affiche();
	if(ch3.equals(ch4))
		cout<<"deux chaines egales \n";
	else
		cout<<"deux chaines distinctes \n";
}
```

Chapitre 5 : La surdefinition d'operateurs

1- Introduction

C++ permet la surdefinition des operateurs, C réalise déjà la surdefinition de certains operateurs
Par exemple :
a*b : * signifie une multiplication
a=*adr : * signifie une indirection

En C++ vous pourrez sur définir n'importe quel operateur (unaire ou binaire) dans la mesure où il porte sur au moins un objet. Si vous définissez une classe complexe destinée à représenter des nombres complexes.

Il vous sera possible de donner une signification à des expressions a+b a-b a/b a*b où a et b sont de type complexe.

2- Le mécanisme de la surdefinition d'operateurs

Class point{
int x,y ;
…

Supposons que nous souhaitons définir l'operateur + afin de donner une signification a une expression a+b lorsque a et b sont de type point
Si nous conviendrons que la somme de 2 points dont les coordonnées sont la somme de leurs coordonnées
L'operateur + doit disposer de 2 arguments de type point et fournir une valeur de la même type
La convention adoptée par C++ pour surdefinir cet operateur consiste a définira une fonction de nom operator +
Cette fonction peut être une fonction membre de la classe concerne ou une fonction indépendante (fonction amie)

2-1 Fonction amie

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

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

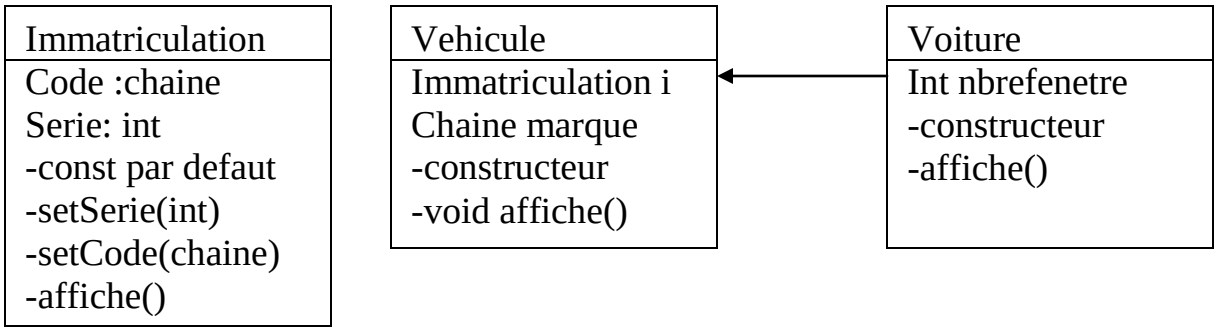
```
this ->x=x;
this ->y=y;
}

void affiche(){
      cout <<"coord:"<<x<<";"<<y<<endl;
}
friend point operator+(point, point);

};

point operator+(point a,point b){
      point c;
      c.x=a.x+b.x;
      c.y=a.y+b.y;
      return c;
}
void main (){

      point a(2,3),b(5,6);
            point c=a+b;
      c.affiche();
      point d=a+b+c;
      d.affiche();
}
```

2-2 Fonction membre sans inline

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

class point{
int x,y ;
public :
point(int x=0,int y=0){
this ->x=x;
this ->y=y;
}

void affiche(){
      cout <<"coord:"<<x<<";"<<y<<endl;
}
point operator+(point );

};

point point::operator +(point b){
point c;
      c.x=x+b.x;
      c.y=y+b.y;
      return c;
}

void main (){

      point a(2,3),b(5,6);
            point c=a+b;a
      c.affiche();
      point d=a+b+c;
      d.affiche();
}
```

2-3 Transmission par référence :

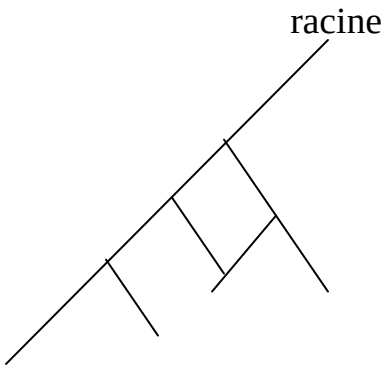
Dans les deux exemples précédents la transmission des arguments se fait par valeur. En particulier dans le cas des objets de grande taille on peut faire appel au transfert par référence.
Point & point ::operator(point &) ;

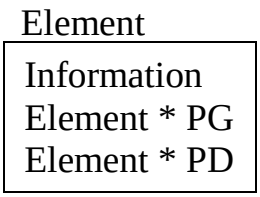



```
#include <iostream>
using namespace std;

class immatriculation{
      char* chaine;
      int serie;
public:
      immatriculation(char* chaine="", int serie=0){
            this ->chaine=chaine;
            this ->serie=serie;
      }
      void setSerie (int serie){
            this ->serie=serie;
      }
      void setCode(char* chaine){
            this->chaine=chaine;
      }
      void affiche(){
            cout <<"la chaine est "<<chaine<<" la serie est "<<serie<<endl;
      }
};

class vehicule{
immatriculation i;
char* marque;
public:
vehicule(char *marque,immatriculation i){
      this ->marque=marque;
      this->i=i;
}

void affiche(){
      i.affiche();
      cout<<"la marque est "<<marque<<endl;
}
};

class voiture:public vehicule{
      int nbfenetre;
public:
voiture(char *marque,immatriculation i,int nbfenetre):vehicule(marque,i){
            this->nbfenetre=nbfenetre;
      }
      void affiche(){
            vehicule::affiche();
```

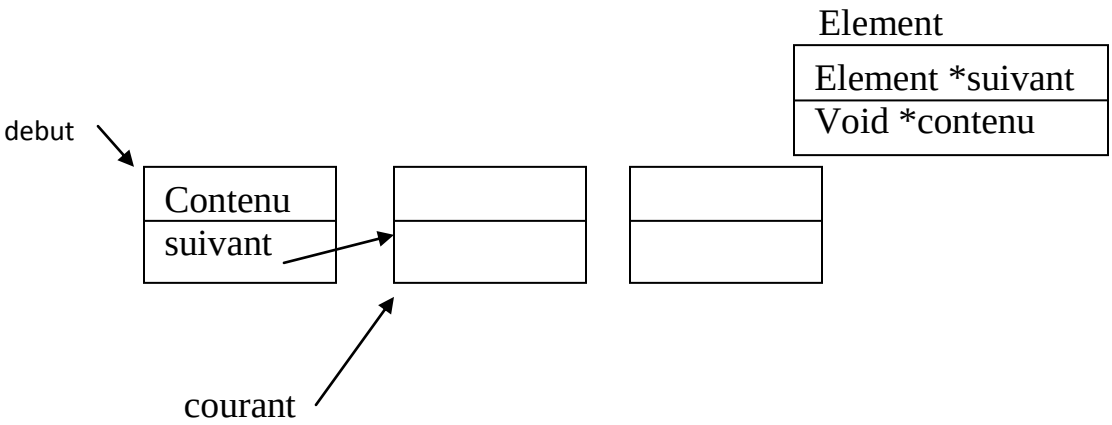
```
            cout<<"le nb de fenetre est "<<"\a"<<nbfenetre<<endl;
      }
};

void main(){
      immatriculation i("SPORT",3);
      voiture a("bmw",i,4);
      a.affiche();
}
```

Projet: Arbre binaire





TD 7 : Liste chainée



-constructeur
-destructeur pour détruire la liste
-void premier ()//positionne courant sur premier élément
-ajoute(void*)
-void *prochain()//retourne l'info de l'élément courant
-bool fini

Solution TD7 :

Liste.h:

```
#include <iostream>
```

```
using namespace std;

struct Element {
	Element *suivant;
	void *contenu;
};

class liste{
	Element *debut;
	Element *courant;
public:
	liste();
	void ajoute(void *);
	void premier();
	void prochain();
	bool fini();
	~liste();
};
```

List.cpp:

```
#include "liste.h"

liste::liste(){
	debut=NULL;
	courant=NULL;
}

void liste::ajoute(void *chose){
Element *nouveau = new Element;
nouveau ->contenu=chose;
nouveau ->suivant=debut;
debut=nouveau;
}

void liste::premier(){
	courant=debut;
}

void *liste::prochain(){
	void *chose=courant->contenu;
	courant =courant->suivant;
	return chose;
}

bool liste::fini(){
	return courant== NULL;
}

~liste(){
	Element *sv;
	premier();
	while (!fini()){
		sv=courant->suivant;
		delete courant;
		courant=sv;
	}
};

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

```
this ->x=x;
this ->y=y;
}

void affiche(){
      cout <<"coord:"<<x<<";"<<y<<endl;
}
};

class listepoint:public liste,public point{
public:
      listepoint(){}

      void affiche(){
            premier();
            while(!fini()){
                  point *ptr=(point *)prochain();
                  ptr->affiche();
      }
      }
};

void main(){
      listepoint l;
      point a(1,2),b(3,4),c(5,6);
      l.ajoute(&a); //on le donne l'addresse de a
      l.affiche();
      l.ajoute(&b);
      l.affiche();
      l.ajoute(&c);
      l.affiche();
}
```

3- **Limites et possibilités de la surdefinition d'operateurs**

3-1 Il faut se limiter aux operateurs existants. Le symbole suivant le mot operator doit être un operateur déjà définit pour les types de base

Certains operateurs ne peuvent être redéfinis(c'est le cas .) et que d'autres imposent quelques contraints supplémentaire

Il faut conserver la pluralité (unaire, binaire) de l'operateur initial. Ainsi vous pourrez surdefinir + unaire ou binaire mais vous ne pourrez pas définir = unaire ou de ++ binaire

3-2 Au contexte de classe

On ne peut pas sur définir un operateur que s'il comporte au moins un argument de type classe

De plus certains operateurs doivent obligatoirement être définis comme membre d'une classe par exemple [], (). -> Ainsi que de new et delete

3-3 Les operateurs = et & ont une signification prédéfinie

= un operateur prédéfinie a=b (a et b type classe)

Constructeur par recopie est prédéfinie

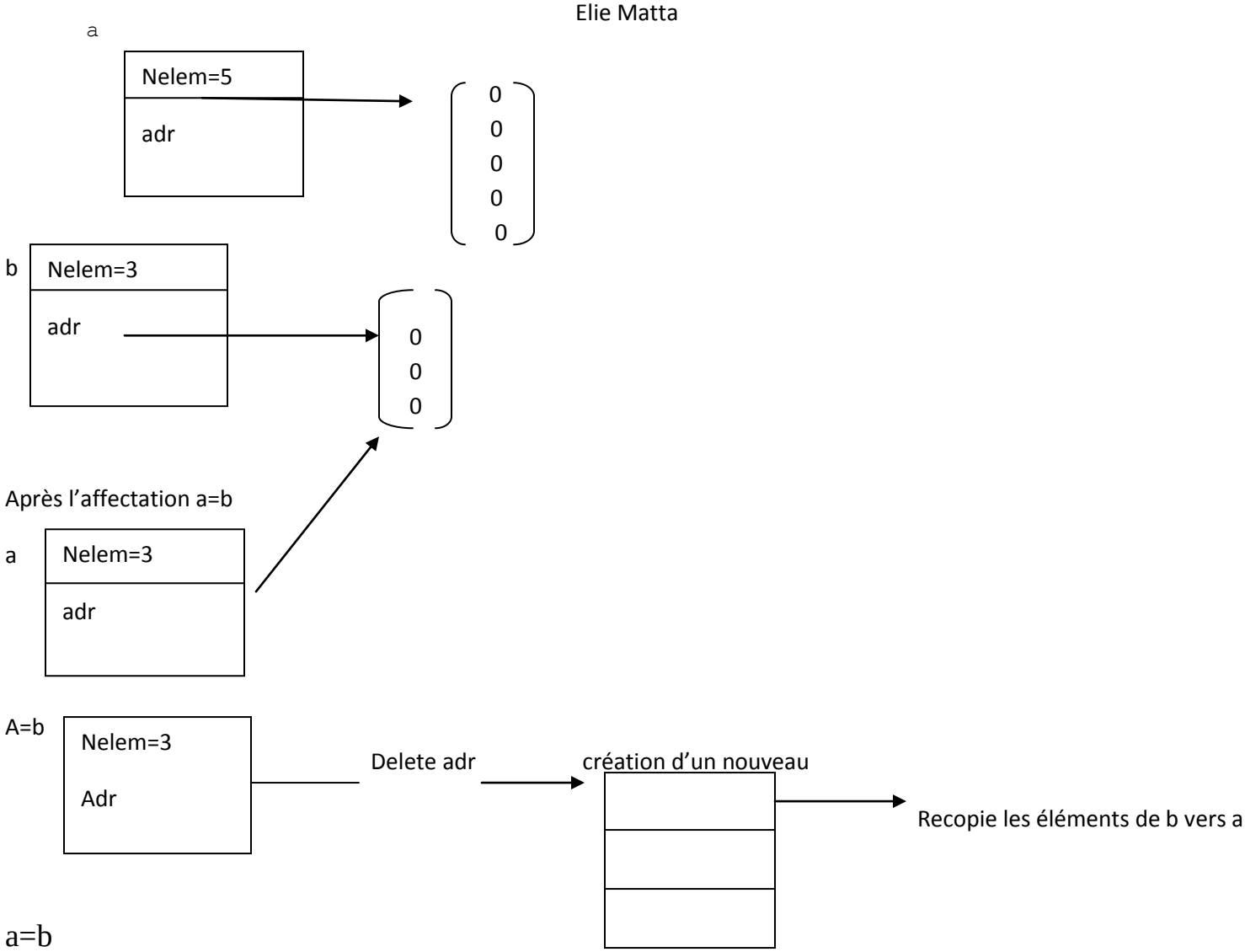
Point a(b) ; (a et b type point avec b déjà existe)

- Le constructeur par recopie : s'il n'en exister pas d'explicite, il y a appel d'un constructeur par défaut.

- L'operateur d'affectation = : s'il n'en existe pas d'explicite, il y a appel d'un operateur d'affectation par défaut
- Constructeur par recopie par défaut et affect par défaut effectivement le même travail.

3-4Exemple de sur définition de l'operateur

```
#include <iostream>
using namespace std;
class vecteur {
int nelem ;
int *adr ;
public :
vecteur (int n){
nelem=n ;
adr= new int[n] ;
for(int*p=adr ; p<adr+n; p++)
*p=0;
}
~vecteur(){
      cout<<"destruction de l'objet en "<<this<<"vecteur dyn en :"<<adr<<endl;
      delete adr;
}
void affiche (){
      cout<<"cont en adresse: "<<this<<"tableau dynamique: "<<adr<<endl;
      for(int *p=adr; p<adr+nelem;p++)
            cout<<*p<<"\t";
      cout<<"\n";
}
vecteur & vecteur::operator=(vecteur &v){
      cout<<"appel operateur= taille"<<nelem<<"adresse objet"<<this<<"adresse vecteur
dynamique"<<adr<<"\n";
      if(this!=&v){
            cout<<"effacement vecteur dynamique en "<<adr<<"\n";
            delete adr;
            adr= new int[nelem=v.nelem];
            cout<<"nouveau vect dyn en"<<adr<<"\n";
            for(int i=0; i<nelem;i++)
                  adr[i]=v.adr[i];
}
return *this;
}
};
void main(){
      vecteur a(5),b(3);
      a.affiche();
      b.affiche();
      cout<<"affect par defaut a=b \n";
      a=b;
      a.affiche();
      b.affiche();
}
```



a=b

une solution consiste a:

-delete le adr de a

-créer un nouveau (nelem=b.nelem)

-recopie les éléments de b.adr dans a.adr

Surdefinition de [] de maniere que v[i] designe l'element d'emplacement v.adr[i](v est un vecteur)

```
Class vecteur{
.
.
Public :
.
.
Int & operator[](int i ){
Return *(adr[i]);
}
};
```

Autre sur définition de =

L'inconvénient de la méthode précédente est les duplications des parties dynamiques=> une perte de temps et d'espace mémoire.

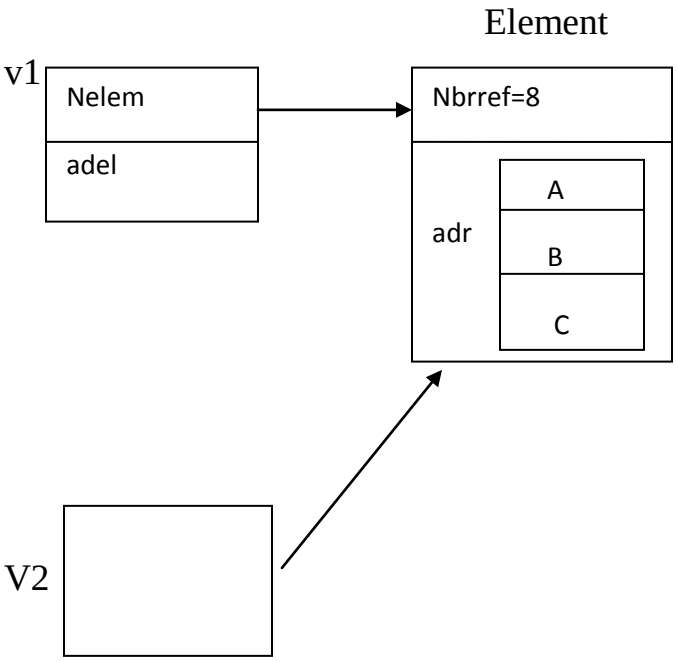
Exemple d'utilisation d'un compteur de référence :

Les méthodes devant agir sur le compteur de références :

-Le constructeur pas recopie : il doit initialiser un nouvel objet pointant sur un emplacement déjà référencé et donc incrémenter son compteur de référence.

a=b doit :

1. décrémenter le compteur de référence de l'emplacement référencé par a et procéder a sa libération lorsque le compteur est nul.
2. Incrémenter le compteur de référence de l'emplacement référencé par b.



Vecteur v2(v1)

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

struct element {
	int nref;
	int *adr;
};
class vecteur{

	int nelem;
	element *adel;
	void decremente(){
		adel->nref--;
		if(!adel->nref){
			delete adel->adr;
			delete adel;
		}
```

```
	}
public:
	vecteur(int n){
		adel= new element;
		adel->adr= new int[nelem=n];
		adel->nref=1;
	}
	vecteur(vecteur &v){// const par recopie
		nelem=v.nelem;
		adel=v.adel;
		adel->nref++;
	}
	~vecteur(){
		decremente();
	}
	vecteur operator=(vecteur &v){//surdefinition
		if(this!=&v){
			decremente();
			v.adel->nref++;
			adel=v.adel;
			nelem= v.nelem;
		}
		return *this;
	}
	int &operator[](int i){
		return adel->adr[i];
	}
	void affiche(){
		for(int i=0;i<nelem;i++)
			cout<<adel->adr[i]<<" ";
	}
};
void main(){
	vecteur v1(5),v2(10);
	for(int i=0; i<5;i++)
		v1[i]=i;
	cout<<"v1=\n";
	v1.affiche();
for(int i=0; i<5;i++)
		v2[i]=i;
	cout<<"v2=\n";
	v2.affiche();
	v1=v2;
	cout<<"v1=\n";
	v1.affiche();
	vecteur v3=v1;
	cout<<"v3=\n";
	v3.affiche();
}
```

5- Surdefinition des operateurs new et delete:

La surdefinition de new se fait obligatoirement par une fonction membre celle-ci doit :

-recopier un argument du type size_t (stddef.h ou stdlib.h)

Cet argument correspond à la taille(en octets) de l'objet à allouer.

Bien qu'il figure dans la définition de new, il n'a pas a être spécifié lors de son appel, car c'est le compilateur qui le génèrera automatiquement (en fonction de la taille d'objet concerné).

-fournir en retour une valeur du type void* correspond a l'adresse de l'emplacement alloue par l'objet.

La surdefinition de delete(obligatoire, fonction membre) et elle doit :

-recevoir un argument du type pointeur sur la classe correspondante. Il représente l'adresse de l'emplacement alloué a l'objet a détruire.

Remarque :

Il reste possible de faire appel a l'operateur prédéfini en utilisant

:: new

::delete

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class point{
	static int npt;//nbr total des pts
	static int nptdyn;//nbr de pts dynamiques
	int x,y;
public:
	point(int abs=0, int ord=0){
		x=abs;
		y=ord;
		npt++;
		cout<<"il y a maintenant "<<npt<<" points"<<endl;
	}
	~point(){
		npt--;
		cout<<"il y a maintenant "<<npt<<" points"<<endl;
	}
	void* operator new(size_t sz){
		nptdyn++;
		cout<<"dont"<<nptdyn<<"pts dynamiques";
		return ::new char[sz];
	}
	void operator delete(void* dp){
		nptdyn--;
		cout<<"dont"<<nptdyn<<"pts dynamiques";
		::delete(dp);
	}
};
int point ::npt=0;
int point::nptdyn=0;
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;
}
```

Chapitre 6 : La conversion de types :

1- Conversions de types de base

C++ offre les possibilités de conversion d'un type de base à un autre type de base. Ces conversions peuvent être implicites ou explicites.

- Conversions explicites :
  - Int n ;
    Double z ;
    …
    Z= (double)n ; // z=double(n) ;

- Conversion implicites sont mises en place par le compilateur en fonction du contexte.
  - Dans les affectations : il y a alors conversion forcés dans le type de la variable réceptrice.
    Int n ;
    Double z ;
    n=z ; // conversion implicite double -> int puis affect.

  - Dans les appels de fonctions : comme le prototype est obligatoire en C++ il y a également conversion forcée d'un argument dans le type déclaré dans le prototype.
    void fct(int, double) ;//prototype
    char c ;
    int n ;
    …
    Appel :
    Fct(c,n) ; // conversion char->int et int -> double

  - Dans les expressions :
    Char->short->long
    Int->float->double
    Int n ;
    Short s ;
    Float f,z ;
    Double d ;
    z=n*s + f*d ;
    Short->int n*s
    Float-> double f*d
    Int->double +
    Double->float z=

2- Les conversions définis par l'utilisateur

- Conversion d'un type de base en un type objet
  Ce type de conversion est réalise par le constructeur de la classe.
  On peut définir un constructeur avec un argument de type de base.
  Ex :

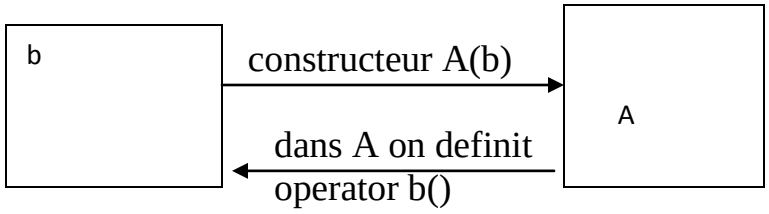
```
Dans la classe point
Class point{
Int x,y ;
…
Public :
        point(int a){x=a ;y=0 ;}
        point  a ;
        a=point(3) ;
```

mais un constructeur ne peut en aucun cas permettre de réaliser une conversion

```
point ->int
```

ce type de conversion pourra être traité. En définissant au sein de la classe un operateur de « cast »
operator int()

si A et B 2 classes, b un type de base



- Conversion d'un type objet en un type de base operator b()
  En c++ un operateur de cast doit toujours être défini et le type de la valeur de retour ne doit pas être mentionné
  Ex :

```
Class point{
….
Public :
…
Operator int(){
Return x;
}
….
};
```

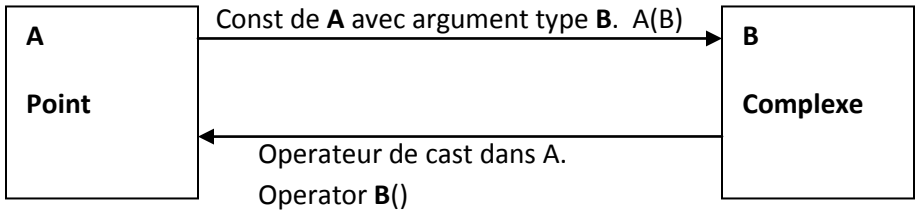
n2=b a été traduite par le compilateur :
-une conversion point->int
-affectation

-dans l'appel de fonction :

```
#include <iostream>
using namespace std;
class point{
	int x,y;
public:
	point(int x=0, int y=0){
		this->x=x;
		this->y=y;
	}
	point(point &b){
		cout<<"appel const par recopie \n";
		x=b.x;
		y=b.y;
	}
	point(int a){
		x=a ;
		y=0 ;
	}
	operator int(){
		return x;
	}
};
void fct(int n){
	cout<<"appel fonction avec argument : "<<n<<endl;
}

void main(){

point a(2,3), b(4,5);
int n3,n4 ;

fct(6);//appel normal
fct(a);
n3=a+3 ;
n4=a+b ;
double z1,z2 ;
z1= a+3;
z2= a+b;
int n1,n2;
n1=int(a);//conversion explicite
cout<<"n1="<<n1<<endl;
n2=b;//conversion implicite en utilisant l'operator
cout<<"n2="<<n2<<endl;
cout<<"n3="<<n3<<endl;
cout<<"n4="<<n4<<endl;
}
```

-En cas d'ambiguite
Operator(){-----}
Operator double(){----}
a+3.85;//Dans ce cas le compilateur refusera l'expression en fournissant une erreur d'ambiguité

-En cas d'un type classe en un autre type classe



Exemple:

point ⟷ complexe

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

class Complexe{
            double re,im;
      public:
            Complexe(double re=0,double im=0){
                  this->re=re;
                  this->im=im;
            }

            void affiche(){
                  cout << re << " +i "<< im << endl;
            }
            double getRe(){
                  return re;
            }
            double getIm(){
                  return im;
            }
      };


class Point{

int x,y;

public:
      Point (int x=0,int y=0){
            this ->x=x;
            this->y=y;
      }
      Point (Complexe c){ //point b(c) : Complexe -> Point
            x=c.getRe();
            y=c.getIm();
      }
      operator Complexe(){//d=complexe(a) : Point -> Complexe
            Complexe c(x,y);
            return c;
      }

void affiche(){
            cout <<"coord:"<< x <<";"<< y <<"\n";
      }
```

```
};

      void main(){
            Point a(1,2);
            Complexe c(4.5,3.0);
            a.affiche();
            c.affiche();
            Point b(c);
            //ou bien on peut faire Point b=c; Conversion implicite : Complexe ->Point
            b.affiche();
            Complexe d;
            d=Complexe(a); //Explicite
            //ou d = a : Implicite
            d.affiche();
              }
```

Exemple:

Polygone
n: int //nombre de sommets sommet : tableau dyn. du type point
polygone() affiche() :void perimetre() :double surdef == fonction amie //vrai si 2 polygone identique

```
#include <iostream>
#include <math.h>
using namespace std;

class Point{

int x,y;

public:
      Point (int x=0,int y=0){
            this ->x=x;
            this->y=y;
      }

void affiche(){
            cout <<"coord:"<< x <<";"<< y <<"\n";
      }
int distance(point b){
            double dx,dy;
            dx = (this->x-b.x)*(this->x-b.x);
            dy = (this->y-b.y)*(this->y-b.y);
            return sqrt(dx+dy);
      }
};


class Polygone{
      int n;
      point *sommet;
public:
            Polygone(int n, point *sommet){
            this->n=n;
            this->sommet=sommet;
      }
```

```
	void affiche (){
		cout <<"Les sommets sont: "<<endl;
		for (int i=0;i<n;i++){
			cout <<"sommet["<<i<<"]="<<*(sommet+i).affiche()<<endl;
		}
		double perimetre(){
			double p=0;
			for(int i=0; i<n; i++){
				p+=*(sommet+i).distance(*(sommet+i+1));
				p+=*(sommet+n-1).distance(*(sommet));
			return p;
			}
			bool operator ==(polygon b){
			if(n!=b.n)
				return false;
			for(int i=0;i<n;i++){
				if(!*(sommet+i)==b.*(sommet+i));
				return false;
			}
			return true;
			}
		};
```

Typage statique et fonctions virtuelles:

```
#include <iostream>
#include <math.h>
using namespace std;

class point{
protected:
int x,y;

public:
	point (int x=0,int y=0){
		this ->x=x;
		this->y=y;
	}

void affiche(){
	cout<<"je suis un point\n";
		cout <<"coord:"<< x <<";"<< y <<"\n";
	}
};
class pointcol:public point{
	short couleur;
public:
	pointcol(int x=0, int y=0, short couleur=1):point(x,y){
		this->couleur=couleur;
	}
	void affiche(){
	cout<<"je suis un point\n";
	cout <<"coord:"<< x <<";"<< y <<"\n";
		cout<<"et ma couleur"<<couleur<<endl;
	}

};
void main(){
point a(2,3);
point *adp;
pointcol *adpc;
pointcol b(4,5,2);
```

```
a.affiche();
b.affiche();
adp=&a;
adp->affiche();//affiche point
adpc=&b;
adpc->affiche();//affiche pointcol
a=b; //1- a)
a.affiche();//appel affiche() point
//b=a;
adp=adpc;
adp->affiche();//affiche de la classe point
//adpc=adp;
adpc=(pointcol*)adp; //1- b)
}
```

1- Compatibilité entre objet d'une classe de base et objet d'une classe dérivée :

En POO on considère qu'un objet d'une classe dérivée peut remplacer un objet d'une classe de base. Cette idée repose sur le fait que tout ce que l'on trouve dans une classe de base (fonctions et données) se trouve également dans la classe.
Cette comptabilité se résume à l'existence de conversions implicite :
a)d'un objet d'un type dérivé dans un objet d'un type de base
b)'un pointeur sur une classe dérivée en un pointeur sur une classe de base

2- Limitations liées au typage statique :

```
point p(3,5);
pointcol pc(8,6,2);
adp=&p;
adpc=&pc;
adp->affiche();//appel affiche point
adpc->affiche();//appel affiche pointcol
adp=adpc;
adp->affiche();//appel affiche point l'appel de la methode s'effectue lors de la
compilation selon le type de l'objet qui est ici point
```

De meme :

```
p.affiche();
pc.affiche();
p=pc;
p.affiche();
```

Les risques de violation des protections de la classe de base

```
class A{
      int x;
public:
      float z;
      void fa(x);

};
class B: private A{
      int x;
public:
      double v;
      void fb();
};
void main(){
      Aa;
      Bb;
      a.z=8.25;
```

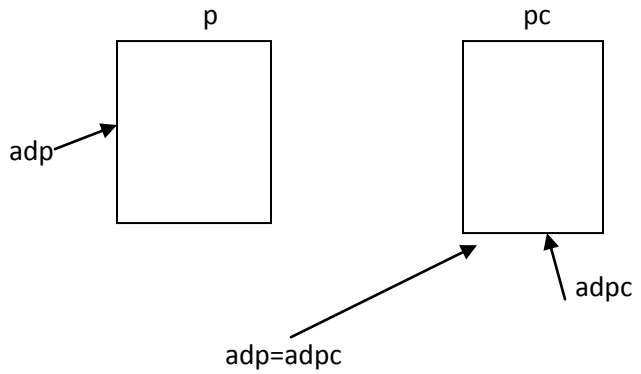
```
        a.fa();
        b.z=---//cette ecriture est rejetee par le compilateur
        b.fa();//cette ecriture est rejetee par le compilateur
        A *ada;
        B *adb;
        adb= &b;
        ada= adb;
        ada->z=2.5;
        ada->fa();
}
```

3- Fonctions virtuelles et typage dynamique :

L'appel d'une methode pour un objet pointé conduisait systematiquement a appeler la methode correspondant au type du pointeur et non pas au type effectif de l'objet pointé lui-meme (typage statique) pour pouvoir obtenir l'appel de la methode correspondant au type de l'objet pointé, il est necessair que le type de l'objet ne soit pris en compte qu'au moment de l'execution. On parle alors de typage dynamique.

3-1 Mecanisme des fonctions virtuelles

Ce mecanisme nous permet de faire en sorte que adp->affiche() (apres adp=adpc) apelle la methode qui correspond au type de l'objet designé par adp (ici pointcol)



Pour ce faire, il suffit tout simplement de déclarer virtuelle la méthode affiche de la classe point

```
virtual void affiche()// affiche de la classe point


point p(2,3);
pointcol pc(8,6,2);
point *adp;
pointcol *adpc;
adp->affiche();
adpc->affiche();
adp=adpc;
adp->affiche();//appel affiche pointcol
```

3-2 Une autre situation où le typage dynamique est indispensable

```
point p(2,3);
pointcol pc(8,6,2);
point *adp=&p;
pointcol *adpc=&pc;
adp->affiche();
adpc->affiche();
adp=adpc;
adp->affiche();//appel affiche pointcol
```

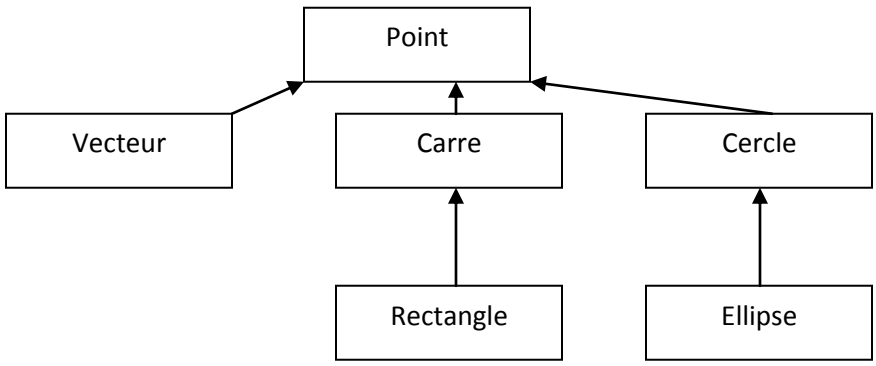
Pour éviter dans l'exemple précédent la redondance dans chaque fonction affiche () des instructions d'affichage

```
Cout<< « coord »-------
Nous pouvons définir notre fonction affiche ()
Class point{
…
Virtual void identifie(){
Cout<<"je suis un point \n";
}
Void affiche(){
Identifie() ;
Cout<< « coord : » <<x<< » ; »<<y<<endl ;
}

Class pointcol : public point{
…
Void identifie(){
Cout <<"je suis un point coloré \n";
}
..
} ;
Void main(){
…
Pc.affiche()
…
}
```

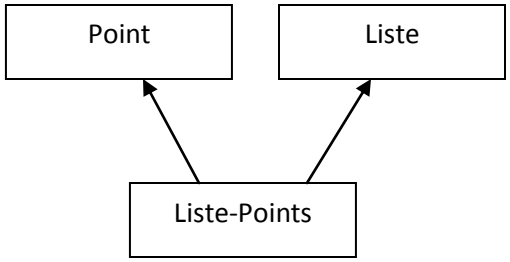
4. Les fonctions virtuelles en general

4.1 Leurs limitations sont celles de l'heritage

Une fonction a été déclaré virtuelle dans une classe A, elle sera soumise a la ligature dynamique dans A et dans toutes les classes descendant de A



De même si l'on dispose de l'héritage multiple



Si dans point la fonction affiche() a été déclarée virtuelle, il devient possible d'utiliser la classe Liste-Points pour gérer d'objets hétérogènes (vecteur, carre….)

4-2 Surdefinition

On peut surdefinire une fonction virtuelle chaque fonction surdefinit pouvant être ou ne pas être déclarée virtuelle

4-3 On peut déclarer une fonction virtuelle dans n'importe quelle classe

4-4 Quelques restrictions

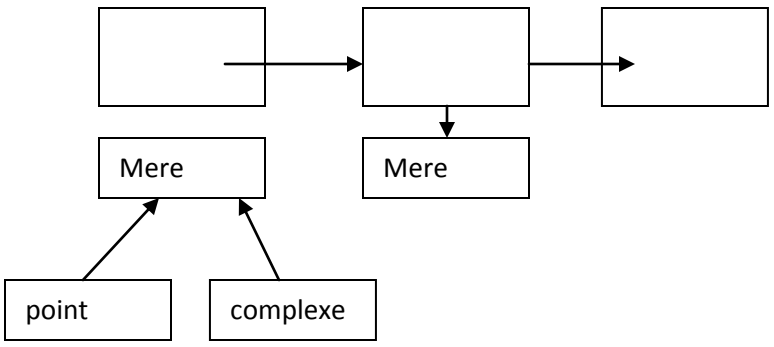
Seule une fonction membre peut être virtuelle
Un constructeur ne peut pas être virtuelle par contre un destructeur peut l'être

5-Classes abstraites

Nous pouvons définir des classes destinées non pas à instancier des objets, mais simplement a donner naissance a d'autres classes par héritage.
En P.O.O on dit qu'on a affaire a des classes abstraites.
C++ propose un outil facilitant la définition de classes abstraites, il s'agit des fonctions virtuelles pures.
Une fonction virtuelle pure est une fonction qui n'a pas un corps.

```
Virtual void affiche()=0 ;
```

Toute classe qui contient une fonction virtuelle est une clase abstract
On ne peut pas instancier une classe abstraite
Une fonction définit virtuelle pure dans une classe de base doit obligatoirement être redéfinit dans une classe dérivée ou déclaré implicitement ou explicitement a virtuelle pure dans ce dernier cas, la classe dérivée est elle aussi abstraite



```
#include <iostream>
#include <math.h>
using namespace std;
class Mere{
public:
	virtual void affiche()=0;
};

class point:public Mere{
protected:
int x,y;

public:
	point (int x=0,int y=0){
		this ->x=x;
		this->y=y;
	}

void affiche(){
	cout<<"je suis un point\n";
		cout <<"coord:"<< x <<";"<< y <<"\n";
	}
};
class pointcol:public point{
	short couleur;
public:
	pointcol(int x=0, int y=0, short couleur=1):point(x,y){
		this->couleur=couleur;
	}
	void affiche(){
	cout<<"je suis un point\n";
	cout <<"coord:"<< x <<";"<< y <<"\n";
		cout<<"et ma couleur"<<couleur<<endl;
	}

};

class Complexe:public Mere{
	double reel, image;

public:

	void affiche(){
		cout<<image;
		cout<<reel;
	}
};

class ListeHetero:public Liste, public Mere{
public:
	ListeHetero(){}
	void affiche(){
		Mere *ptr=(Mere*)premier(); //pointe sur le premier element et le retourne
		while(!fini()){
ptr->affiche();
ptr=(Mere*)prochain();
		}
	}
```

TD 8: Conversion point-> complexe

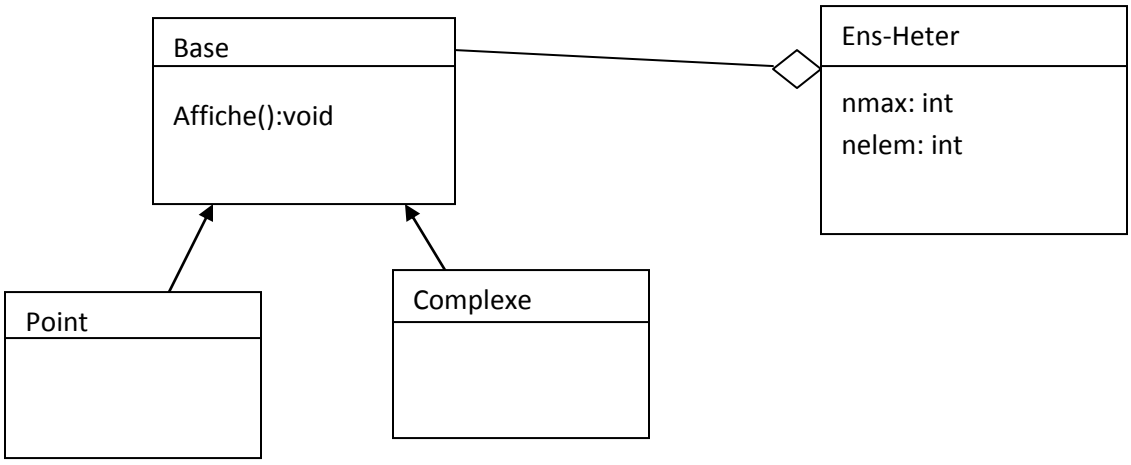
1) Dans complexe surdefinir l'operateur +
2) En utilisant un constructeur dans complexe
3) En definissant un operateur de cast des point

```
#include <iostream>
#include <math.h>
using namespace std;

class point;
class Complexe{
      double re, im;
public:
Complexe(double re=0,double im=0){
                  this->re=re;
                  this->im=im;
            }

Complexe(point); //prototype

void affiche(){
      cout<<"nb complexe:"<<re<<" +i "<<im<<endl;
}
      Complexe operator+(Complexe z){
            Complexe r;
            r.re=z.re+re;
            r.im=z.im+im;
            return r;
      }

};

class point{
      int x,y;
public:
      point(int x=0, int y=0){
            this->x=x;
            this->y=y;
      }

void affiche(){
      cout<<"je suis un point\n";
            cout <<"coord:"<< x <<";"<< y <<"\n";
      }
      friend Complexe::Complexe(point);
};

Complexe::Complexe (point p){ //si on a fait pas comme ca (friend et prototype) on ne
//pourrait pas accede au membre prive de la classe point (x, y)
      re=p.x;
      im=p.y;
}

void main(){
      point a(2,5);
      Complexe c;
      c=(Complexe)a; //conversion explicite
      c.affiche();
      point b(9,12);
      c=b; //conversion implicite
      c.affiche();
      Complexe d(2.5,3.0);
```

```
Complexe z=a+d; //on aura une erreur car a est de type point
z.affiche();
 z=d+a;
 cout<<endl;
z.affiche();
cout<<endl;
 z=a+b;
z.affiche();
cout<<endl;
}
```

TD9 : Classes abstraites :

Créer une classe ens-Heter permettant de manipuler des éléments de types différents. Pour cela tous les types concernés devront dériver d'un même type de base nommé Base
Fonctions membre :
-Ajoute pour ajouter un nouvel élément
-appartient pour tester l'appartenance d'un élément
-card qui fournira le nombre d'éléments
-init pour initialiser le mécanisme d'itération
-suivant qui fournira en retour le prochain élément
-existe pour préciser s'il existe encore un élément non examiné
-liste permettra d'afficher les caractéristiques de tous les éléments de l'ensemble
On réalisera ensuite un petit programme (main) d'essai de la classe ens-Heter, en créant un ensemble des objets de type point(coord entiers) et complexe(float, float)

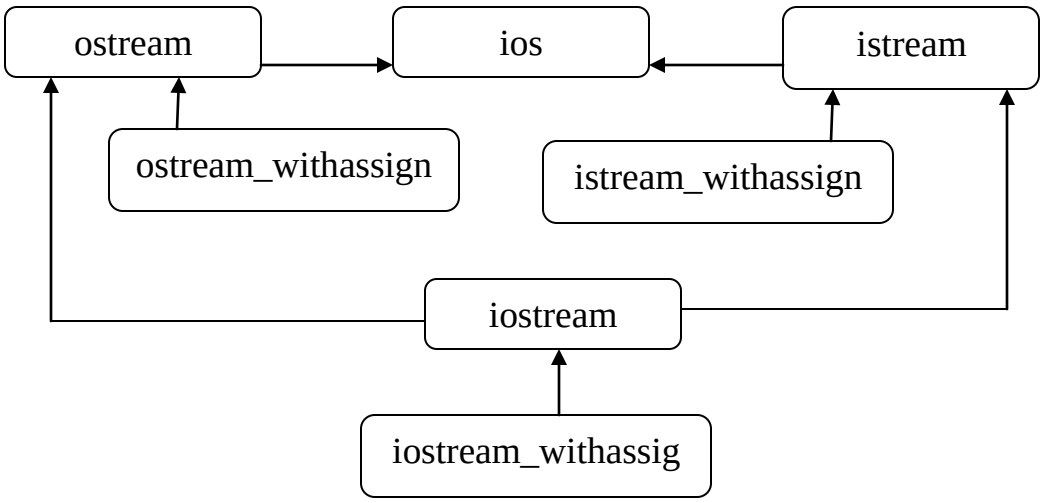


Chapitre 7 : Les flux

Un flux (stream) est une abstraction logicielle représentant un flot de données entre :
Une source produisant de l'information une cible consommant cette information
Il peut être représenté comme un buffer et des mécanismes associes a celui-ci et il prend en charge quand le flux est crée l'acheminement de ces données
Les flux sont dans une librairie standard qui implémente les flots a partir de classes
Par défaut chaque programme C++ peut utiliser 3 flots :
Cout qui correspond a la sortie standard
Cin qui correspond a l'entrée standard
Cerr qui correspond a la sortie standard d'erreur

Pour utiliser d'autres flots, vous devez donc créer et attacher ces flots a des fichiers ou a des tableaux de caractères

Flots de classes
Classes déclarées dans iostream.h et permettant la manipulation de périphériques standard



Ios : classe de base des E/S par flot
Elle contient un objet de la classe streambuf pour la gestion des tampons d'E/S

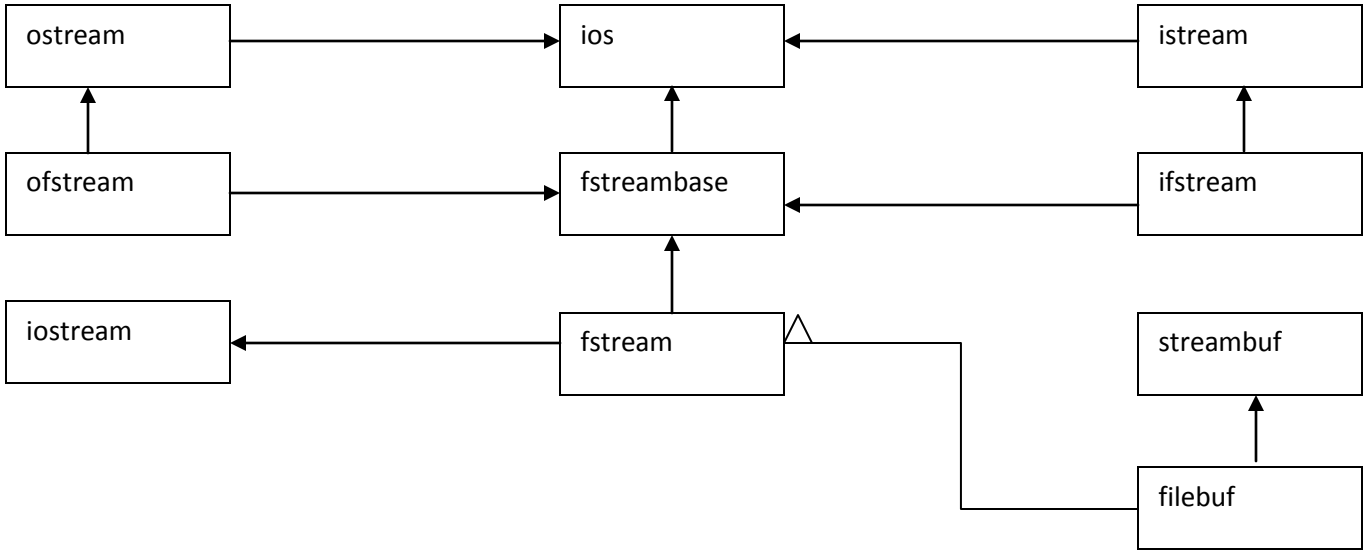
Istream : classe dérivée de ios pour les flots d'entrée

Ostream : classe dérivée de ios pour les flots de sortie

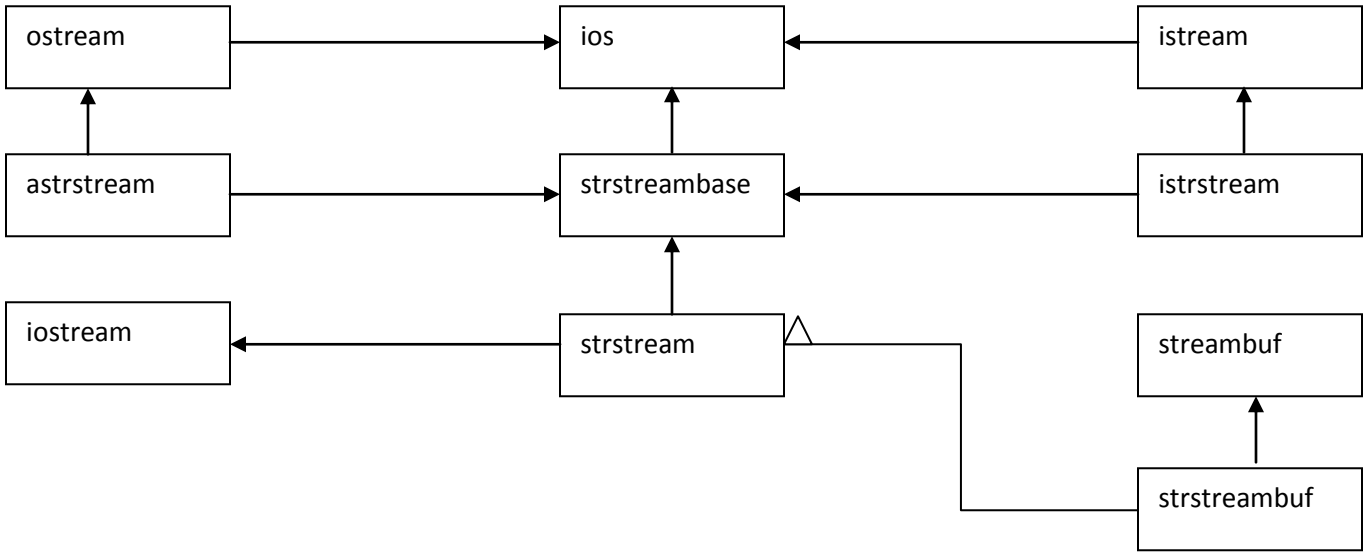
Iostream : pour l'entrée sortie

Istreamwithassign : classe dérivée et qui ajoute l'operateur l'affectation
Les flots cout, cin, cerr sont des instances de ces classes

Classes déclarées dans fstream.h permettant les manipulations des fichiers disques



Classes déclarées dans strstream.h permettant de simuler des opérations d'E/S avec des tampons en mémoire centrale



Le flot de sortie : ostream

- Il fournit des sorties formatées et non formatées (dans un streambuf)
- Surdefinition de l'operateur <<
- Il gère les types prédéfinis de C++
- On doit le surdefinir pour ses propres types

Example : Surdefinition de l'operateur <<

```
Class Exemple{
Char nom[20] ;
Int valeur ;

Public :
```

…
Friend ostream &operator<< (ostream & as, const Exemple &ex) ;
..
};

Ostream & operator << (ostream & os, const Example &ex){
Return os<<ex.nom<<" "<<ex.valeur<<endl;
}

Void main(){
Example e1("item",100);
Cout<<"Example ="<<e1<<endl;
}

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

class point{
	int x,y;

public:
	point(int x, int y){
		this->x=x;
		this->y=y;
	}

	friend ostream &operator <<(ostream & os, const point &p);
};

ostream &operator <<(ostream & os, const point &p){
	return os<<p.x<<" "<<p.y;
}

void main(){
	point p (1,3);
	cout <<p<<endl;
}
```

Principales méthodes de ostream

- ostream & put(char c) ; //insere un caractere dans le flot
  cout.put('\x) ;
- ostream & write(const char* , int n) ; //insere n caracteres dans le flot
  cout.write(« Bonjour», 7) ;
- streampos tellp() ; // retourne la position courante dans le flot
- ostream & seekP(streampos m) ;//se positionne a m octets par rapport au début du flux. Les positions commencent a 0 et le type streampos correspond a la position d'un caractère dans le fichier.

Ostream & seekP (streampoff dep, seek_dir dir) ;
Se positionne dep octets par rapport:

- au début du flot : dir=beeg
- a la position courante : dir=cur

- a la fin du flot : dir=end

un flot fout
streampos old_pos = fout.tellp()//memorise la position
fout.seekp (0, end) ; //se positionne a la fin du flux
cout<< « taille du fichier : » <<fout.tell()<< »octets \n » ;
font.seekp(old_pos,beg); //se repostionne comme au depart

ostream& flush() ; //vide les tamponjs du flux

Le flot d'entrée : istream

- il fournit des entrées formatées et non formatées (dans streambuf)
- surdefinition de l'operateur ??
- il gère les types prédéfinis
- on dot le surdefinir pour ses propres types
- par défaut >> ignore tous les espaces

Exemple : surdef de >>

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

class point{
	int x,y;

public:

	point(){
		this->x=x;
		this->y=y;
	}

	point(int x, int y){
		this->x=x;
		this->y=y;
	}

	friend istream & operator >>(istream & is, point &p);
};

istream & operator >>(istream & is, point &p){
	return is>>p.x>>p.y;
}

void main(){
	point a;
	cin >>a;
}
```

Les principles methods de istream

- Lecture d'un caractere
  - Int get(); //retourne la valeur du caractère lu (ou EOF si la fin du fichier est atteinte)
  - Istream & get (char &c) ; //extrait le premier caractere du flux (meme si c'est un espace) et le place dans c

- Lecture d'une chaine de caract
  - Istream & get (char *, int n, char delim='\n') extrait n-1 caractere du flux et les place a l'addresse ch. La lecture s'arrete au delimiteur qui est par defaut 'n ou la fin du fichier
  - Istream & getline (char * ch, int n, char delim='\n'); //extrait n-1 carachtere. Le delimiteur est extrait du flux mais n'est pas recopie dans les tampons.
  - Istream & read (char *ch, int n) extrait un bloc d'au plus n octets du flux et les place a l'addresse ch. Le nombre d'octets effectivement lus peut etre obtenu par la methode gcount()
  - Int gcount()//retourne le nombre de caractere non formatees extraits lors de la derniere lecture
  - Streampos tellg() : retourne la position courante dans le flux
  - Istream & seekg(streamoff dep, seek_dir dir) ; //se positionne a dep octets par rapports…
  - Istream & flush(); //vide les tampons du flux

Contrôle de l'état d'un flux

La classe ios décrit les aspects communs des flots d'E/S
C'est une classe de base virtuelle pour tous les objets flots. (On ne peut pas instancier cette classe)
Vous utilisez ses méthodes pour tester l'état d'un f lot ou pour contrôlés le formatage des informations

Methodes de la classe de base ios

- Int good() : retourne une valeur différent de 0 si la dernière opération d'E/S s'est effectuée avec succes et une valeur nulle en cas d'echec
- Int fail() : fait l'inverse
- Int eof : different de 0 si la fin du fichier est atteinte
- Int bad() : different de 0 si vous avez tente une operation interdite
- Int rdstate () : retourne la valeur de la variable d'état du flux
- Void clear() : remet a 0 l'indicateur d'erreur du flux. C'est une opération obligatoire à faire après qu'une erreur se soit produite sur un flux

Surdef de () et !
Exemple :
If (fl) … //equivalent a direif (fl.good())
If (!fl) //if (!fl.good())
If (!(cin>>x))

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

class voiture{
    char *marque;
    char *modele;
```

```
    char *prix;
public:
    voiture(char *ma=" ", char *mo=" ",char *pr=" "){
            marque=ma;
            modele=mo;
            prix=pr;
}
    void saisie();
    void affiche();
};
void voiture::saisie(){
    marque=new char[0];
    modele=new char[10];
    prix=new char[10];
    cout<<"Marque:";
    cin.getline(marque,10);
    cout<<"Modele:";
    //cin.getline(Modele,10);//Errreur n'extrait pas le \n
    cout<<"Prix: ";
    cin.getline(prix,10);
}
void voiture::affiche(){
    cout<<
```

Chapitre 8 – Flux

1) ostream

- ostream & put (char c);
  cout.put('\n');
- ostream & write (const char*, int n);
  cout.write("rayon",7);
- streampos tellp();
- ostream & seekp (streampos n)
- ostream & seekp(streamoff dep,seekdir dir)
- ostream & flush();

2) istream

- int get();
- istream & get(char &c)
- int peek();//lecture non destructrice du caractère suivant retourne le caractère ou EOF si la fin du fichier est atteinte.
- istream & get (char *ch, int n, char delim='\n');

- istream & getline (char *ch, int n, char delim='\n');
- istream & read (char *ch, int n)
- int gcount() // nbre de caractère affectés par ligne
- streampos tellg() //retourne la position courante du stream
- istream & seekg (streamoff dep, sikdir dir);

Methods de ios
int good() //lister l'etat du fichier
int fail() ;
int eof(); //lorsque la fin du fichier est atteinte
int bad() ;

Surdefinition de ()
Un flux fl
if (fl) <-> if(fl.good)
if(!fl)<-> if(!fl.good)
if (!(cin>>x))

Associer un flux d'E/S a un fichier
Il est possible de créer un objet flot associé à un fichier autre les fichiers standards.
3 flots permettant de manipuler des fichiers sur disque.

- ifstream permet l'accès du flux en lecture.
- ofstream permet l'accès du flux en écriture
- fstream permet l'accès du flux en lecture/écriture

La classe fstreambase offre des méthodes communes a ces classes

- void open(const char *name, int mode);

qui permet d'ouvrir le fichier et d'associer un flux avec ce dernier
name : nom du fichier a ouvrir
mode : mode d'ouverture du fichier
(enum open_mode de la classe ios)

enum open_mode{
app //ajout en fin du fichier
ate //positionnement a la fin du fichier
in //ouverture en lecture (par défaut pour ifstream)
out //ouverture en écriture (par défaut pour ofstream)
binary //ouverture en mode binaire (par défaut en mode texte)
trunc //détracte le fichier s'il existe
nocreate //si le fichier n'existe pas, l'ouverture échoue
noreplace //si le fichier existe. L'ouverture échoue

on peut continuer plusieurs modes par ios ::in\ios ::out

Exemple :

```
#include <fstream>
using namespace std;
ifstream fl;
fl.open("C:\\Documents and Settings\\assembleur\\My Documents\\a.txt");//lectture

expl , la meme :
fl.open("C:\\Documents and Settings\\assembleur\\My Documents\\a.txt",ios::in);


ofstream fl
fl.open("c:\\ssss\\\")


fl.open("c:\\ssss\\\",ios::out) //ouv/ec
```

Exemple :

```
#include <fstream>
#include <iostream>
using namespace std;
void main() {
ofstream fl;
ifstream lect;

//mode ecriture
fl.open("C:\\Documents and Settings\\assembleur\\My Documents\\a.txt",ios::out);
if(fl.good())
      cout << "File Opened " << endl;
else
      cout << "Error!" << endl;


//mode lecture
lect.open("C:\\Documents and Settings\\assembleur\\My Documents\\b.txt",ios::in);
if(lect.good())
      cout << "File Opened " << endl;
else
      cout << "Error!" << endl;
}
```

Exemple :

```
#include <fstream>
#include <iostream>
using namespace std;
void main() {
      //ecrire les informations sur un fichier
ofstream ecrire;
ifstream lecture;

ecrire.open("C:\\Documents and Settings\\assembleur\\My Documents\\a.txt",ios::out);
if(ecrire.fail()){
      cout << "l'ouverture est echoue " << endl;
      exit (-1) ;
}
      cout << "le fichier est ouvert" << endl;
      ecrire<<"rabih"<<18;
      ecrire<<"joelle"<<17;
      ecrire.close();
```

```
//lecture du fichier
lecture.open("C:\\Documents and Settings\\assembleur\\My Documents\\b.txt",ios::in);
if(lecture.good())
      cout << "le fichier existe" << endl;
char *nom=new char[20];
int note;
while(!lecture.eof()){
      lecture>>nom>>note;
      cout<<nom<<note<<endl;
}}
```

N'on pas besoin de imanip.h :

Manipulateur	Rôle
dec	Convertir en base décimale
hex	Convertir en base hexadécimale
oct	Convertir en base octale
ws	Supprimer les espaces
endl	Ajouter un saut de ligne en fin de flux (= à \n)
ends	Ajouter un caractère de fin de chaîne
flush	Vider un flux de sortie
setbase(int a)	Choisir la base (0,8,10 ou 16). 0 est la valeur par défaut
set fill(int c)	Choisir le caractère de remplissage (padding)
set precision(int c)	Indiquer le nombre de chiffres d'un nombre décimal
set w	Choisir la taille du champ (padding)

Exemple:

int e=31 ;

cout<<e ;

cout<<hex<<e ;

Le padding consiste à compléter généralement avec des espaces ou des zéros, un élément affiché à l'écran en précisant la taille d'un champ ou d'une colonne (set w).

Tout élément affiché dont la taille est insuffisante sera complété par défaut avec des espaces. De cette manière chaque élément affiché possédera la même taille, ce qui facilite l'alignement.

Méthodes	Rôle
fill()	Renvoie le caractère de remplissage
fill(char c)	modifie le caractère de remplissage
précision ()	renvoie la précision pour le nombre décimal
précision (int n)	modifie la précision pour le nombre décimal
setf(long flag)	modifie une propriété de formatage (format)
setf(long flag,longchamp)	Modifie une propriété de formatage d'un champ
width()	Renvoie la valeur d'affichage
width(int n)	Renvoie la valeur d'affichage

TD:

```
#include <iostream>
#include <ifstreeam>
#include <ostream>
#include <iomanip>

using namespace std;

void main(){
	int e=15;
	cout.setf(ios::dec,ios::basefield); //decimal
	cout<<"Decimal : e"<< e<<endl;
	cout.setf(ios::oct,ios::basefield);
	cout<<"Octal : e" << e<< endl;
	cout.setf(ios::hex,ios::basefield);
cout<<"Hexadecimal : e" << e<< endl;

float x=3.15;

cout.width(7); //champ largeur 7
	cout<<"Width : x"<< x+1000 <<endl;

cout.setf(ios::dec,ios::basefield); //decimal
	cout<<"Decimal : x"<< x+1000 <<endl;

cout <<setw(7)<<setfill('0')<<hex<<x<<endl;

double p1=3.1415
cout.flush();

cout<<p1<<endl;

cout<<setprecision(3)<<p1<<endl;
cout<<setptecision(5)<<p1<<endl;

char text=[50]="bonjour";
cout.setf(ios::right)<<text<<fill(' * ')<<endl;
}
```

Deuxieme solution (Dans la classe)

```
#include <iostream>
#include <iomanip>
#include <string.h>
using namespace std;

void main(){
	int e=15;
	cout <<"conversion: "<<endl;
	cout <<"entier: "<<e<<endl;
	cout<<"hexadecimal: "<<hex<<e<<endl;
	cout<<"octal: "<<oct<<e<<endl;
	cout<<"largeur et padding: "<<endl;

	float x=3.15;
	cout<<"Decimal: "<<setw(7)<<x<<endl;
	cout<<"Decimal: "<<setw(7)<<x+1000<<endl;
	cout<<"Decimal: "<<setw(7)<<setfill('0')<<x<<endl;
	cout<<"Nombre et chiffres: "<<endl;

	double pi=3.1514;
	cout<<"PI: "<<pi<<endl;
```

```
	cout<<"PI: "<<setprecision(3)<<pi<<endl;
	cout<<"PI: "<<setprecision(5)<<pi<<endl;
	cout<<"chaines, alignement et padding \n";

	char texte[50];
	strcpy(texte,"Bonjour tout le monde");
	cout<<texte<<endl;
	cout.setf(ios::left,ios::adjustfield); //alignement to left
	cout.width(50);
	cout.fill('_');
	cout<<texte<<endl;
}
```

Chapitre 9 : Les patrons des classe (template)

1- Introduction

Les patrons sont des mécanismes qui permettent de créer de fonctions et de classes génériques dont des codes réutilisables.

- Les patrons de fonctions
- Les patrons de classes

2- Les Patrons de fonctions :

Exemple :

```
void echange(int &x ,int &y){
int z=x;
x=y;
y=z;
	}
```

Cette fonction permet d'échanger les valeurs de deux entiers. Si nous désirons échanger les valeurs de deux variables dont le type de paramètres est réel ou char etc…, il faut définir une autre fonction échanger en remplaçant le type int par le type voulu.
Le mécanisme template permet de définir une fonction générique utilisable pour tous les types.

```
template<class T>void echange(T&x, T&y){
	T z=x;
	x=y;
	y=z;
}
```

On peut écrire :

```
template <typename T>void echange....

void main(){
	int x=20, y=40;
	echange(x,y);
		double z=12.5,w=20.6;
	echange(z,w);
}
```

Plusieurs paramètres de types différents:

```
template<class T, class U> void f(Tx, Uy){
```

3- Les patrons de classes:

C++ permet de définir des patrons de classes. Là encore, il suffira d'écrire une seule fois la définition de la classe pour que le compilateur puisse automatiquement l'adapter a différents types.

TD:

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



template<class T> class point{
	T x,y;
public:
	point(T x=0,  T y=0){
		this->x=x;
		this->y=y;
	}
	void affiche();
};
template<class T>void point<T>::affiche(){
	cout<<"coord:"<<x<<";"<<y<<endl;
}
void main(){
//utilisation du patron point
	point<int>a(2,3);
	point<double>b(2.6,4.5);
	a.affiche();
	b.affiche();
}
```

4- Les paramètres de type d'un patron de classe :

Les paramètres de classes peuvent comporter des paramètres de type et des paramètres expressions

4-1 Les paramètres de type peuvent être en nombre quelconque

Ex:

```
template<class T, class U, class V> class Essai{
	T x;//un membre x de type T
	U t[5];//un tableau de 5 elements de type U
public:
	V f(int, U);
	};
```

4-1-1 Instantiation:

```
	Essai<int, float, int> e1;
	Essai<int, int*, double> e2;
	Essai<char*, int, point> e3;
```

4-2 Les paramètres expressions

Un patron de classe peut comporter des paramètres expressions.

Les valeurs effectives d'un paramètre expression devront obligatoirement être constantes.

Ex :

```
template<class T, int n>class tab{
	T tab[n];
public:
};
void main(){
	tab<int,4> t;
	//on definit un tableau de type int de 4 elements
	}
```

Nous déclarons une classe t correspondant à la déclaration suivante :

```
class t{
	int tab[4];
public:
};
```

TD:

Définir une classe Tableau

- Sur définir l'operateur []
- La classe comporte des tableaux de type point

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

class point{
	 int x,y;
public:
	point(int x=0,  int y=0){
		this->x=x;
		this->y=y;
	}
	void affiche(){
	cout<<"coord:"<<x<<";"<<y<<endl;
	}
};



template<class T,int n>class tableau{
	T tab[n];
public:
	tableau(){
		cout<<"construction du tableau";
	}
	T & operator[](int i){
		return tab[i];
	}
};
void main(){
	//un tableau de type int
	tableau<int,4>ti;
	for(int i=0; i<4;i++){
		ti[i]=i;//ti[i] est tab[i]
	}
	for(int i=0; i<4; i++){
			cout<<ti[i]<<"\t";
		cout<<"\n";
	}
		tableau<point,3>tp;
		for(int i=0; i<3;i++)
			tp[i].affiche();
	}
```

5- Spécialisation d'un patron de classe

5-1 Exemple de spécialisation d'une fonction membre

Si l'on souhaite adapter une fonction membre a une situation particulière il est possible d'en définir une nouvelle.

```
template<class T> class point{
```

```
	T x,y;
public:
	point(T x=0,  T y=0){
		this->x=x;
		this->y=y;
	}
	void affiche();
};

//Definition des fonctions
template<class T> void point<T>::affiche(){
	cout<<"coord"<<x<<";"<<y<<endl;
}

void point<char>::affiche(){
	cout<<"coord:"<<(int)x<<";"<<(int)y<<endl;
}

void main(){
	point<int> ai(2,3);
	ai.affiche();
	point<char> ac('d','y');
	ac.affiche();
	point<double> ad(2.5,5,6);
	ad.affiche();
}
```

5-2 On peut spécialiser une fonction membres pour tous les paramètres

```
template <class T, int n> class Tableau{
	T tab[n];
public:
	tableau(){
		cout<<"construction tableau \n";
	}
	...
};

tableau <point,10> ::tableau(){
	...
}
```

On peut spécialiser une classe :

```
class point<char>
{
	...
}
```

6- Patron de functions members

Le mécanisme de définition de patrons de fonction peut s'appliquer a une fonction membre

```
class A{
	...
		template <class T> void fonction (T a) {
			...
	}
};
```

TD 2 :

```
template <class T> class A{
	...
		template <class U> void fonction (U x,T u){
```

```
                ...
        }
};
```

7- Identité de classes patrons (instance d'une classe Patron)

```
tableau <int,12> t1;
tableau <float, 12> t2;
t1=t2;
```

8- Classes patrons et déclaration d'amitié

8-1 Déclaration de classes ou fonctions ordinaires

Si A est une classe ordinaire et fonction une fonction ordinaire

```
template <class T>
class Essai{
        int x;
public:
        friend class A;
        friend int fonction (float);
        ...
};
```

8-2 Déclaration d'instances particulières de classes patrons ou de fonction patrons

```
template <class T> class point{
        ...
}
template <class T> int fonction (T x){
        ...
}
template <class T, class U> class Essai1{
        int x;
public :
        friend class point <int>;
        friend int fonction (double);
        ...
};
```

8-3 Déclaration d'un autre patron de fonctions ou classes

```
template <class T, class U> class Essai2{
        int x;

public:
        template <class X> friend class point<X>;
        template <class X> friend int fonction (point <X>);
```

9- Exemple de classe tableau a 2 indices

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

template <class T, int n> class tableau{
   T tab[n];
public:
   tableau(){
         cout <<"construction tableau a"<<n<<"elements \n";
   }
   T & operator [](int i){
         return tab[i];
   }
};

void main(){
   tableau <tableau <int,2>,3> t2d;
```

```
	t2d[1][2] =15;
	cout<<"t2d[1][2]="<<t2d[1][2]<<"\n";
	cout <<"t2d[0][1]="<<t2d[0][1]<<"\n";
   }
```

TD3 :

Ecrire un fichier moyenne .txt
Les donnees :
Julien 2,45
Maurice 3,7
Andre 4,0

Lire le même fichier en affichant les données :

```
	+-------------+-------+
	- Julient       - 2,45  -
	- Maurice     - 3,7    -
	- Andre         - 4,0    -
	+---------------------+
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;

void main(){
	string nom;
	double moyenne;
	ofstream ofichier("C:\\a.txt");
	ofichier<<"Julien \t"<<2.45<<"\n";
	ofichier<<"Maurice\t"<<4<<"\n";
	ofichier<<"Luc\t"<<3.43667<<"\n";
	ofichier<<"LAndre\t"<<5.215<<"\n";
	ofichier.close();
	ifstream fichier("C:\\a.txt");
	cout<<"+"<<setfill('_')<<setw(18)<<"+"<<endl;
	cout<<setfill('_')<<setw(8)<<"+"<<endl;
	cout<<setfill(' ');
	cout<<fixed<<setprecision(2); //fixed --> elle ecrit sous une forme decimal

	while(!fichier.eof())
	{
		fichier>>nom>>moyenne;
		cout<<"|"<<left<<setw(15)<<nom;
		cout<<"|"<<right<<setw(15)<<moyenne<<endl;
	}
	fichier.close();
	cout<<"+"<<setfill('_')<<setw(8)<<"+"<<endl;
	cout<<setfill('_')<<setw(7)<<"+"<<endl;
}
```