LANGAGE DE PROGRAMMATION ORIENTE OBJET C++

(E.T. : 12 heures, E.D. : 12 heures, E.P. : 12 heures)
Année Universitaire 2023/2024

Monsieur RAKOTOASIMBAHOAKA CYPRIEN ROBERT,
Maître de conférences

Enseignant-chercheur à l'Ecole Nationale d'Informatique
UNIVERSITE DE FIANARANTSOA

I. PRESENTATION GENERALE DE C++
II. INCOMPATIBILITES DE C++ AVEC LE C ANSI
III. SPECIFICITES DE C++
IV. CLASSE, CONSTRUCTEUR ET DESTRUCTEUR
V. PROPRIETES DES FONCTIONS MEMBRE
VI. CONSTRUCTION DESTRUCTION ET INITIALISATION DES OBJETS
VII. FONCTIONS AMIES
VIII. SURCHARGE D'OPERATEURS
IX. CONVERSIONS de TYPE DEFINI par l'UTILISATEUR (CDU)
X. TECHNIQUE DE L'HERITAGE
XI. HERITAGE MULTIPLE
XII. EXCEPTIONS
XIII. FONCTIONS VIRTUELLES
XIV. FLOTS

Chap I : PRESENTATION GENERALE DE C++

C++ a été conçu (1983) par Bjarne Stroustrup (Danemark) Université de Texas aux USA.

Objectif : ajouter au langage **C** des **classes** analogues à celles du langage Simula (ajouter au **C** des possibilités de **Programmation Orientée Objet**).

Qualité de C++ : exactitude, robustesse, extensibilité, reutilisabilité et efficience (capacités de rendement).

Par analogie avec l'équation de Wirth, on pourrait dire que l'équation de la **POO** est :

Méthodes + Données = Objet

Nouveaux concepts :

- **Classe** est un type qui décrit un ensemble d'objets ayant une structure des données communes et disposant des mêmes méthodes.
- **Objets** (instances) sont des variables de type **classe**. Créer une nouvelle instance (objet). L'adresse de l'instance actuelle est appelée **this**.
- **Encapsulation** réalise une « abstraction des données ». Elle facilite considérablement la maintenance et la réutilisabilité d'un objet.
- **Héritage** permet de définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute de nouvelles données et méthodes. Il facilite largement la réutilisabilité de produits existants.

C++ est un sur-ensemble du langage **C**, tel qu'il est défini par la norme **ANSI**. Mais il existe quelques incompatibilités.

Chap II : INCOMPATIBILITES DE C++ AVEC LE C ANSI

C++ est une extension de **C**, mais un certain nombre d'incompatibilités ont subsisté.

2.1. Qualificatif CONST

Il permet de spécifier un symbole dont la valeur ne doit pas changer en **C**.

En **C++**, un certain nombre de différences apparaissent (portée et utilisation dans une expression).

a) Portée : lorsque **CONST** s'applique à une variable globale, **C++** limite la portée du symbole au fichier source concerné. **C** ne faisait pas cette limitation.

Exemple :

En **C**, on procède de cette façon :

#define N 8	**#define** N 3
…	**….**
Fichier 1	fichier 2

En **C++**, on écrit :

int const N**(**8**)**	**int const** N**(**3**)**
...	**...**
fichier 1	fichier 2

b) Utilisation dans une expression : la valeur d'une expression constante est calculée lors de la compilation. Ainsi, avec : **int const** p=3**;**

2*p*5 n'est pas une expression constante en **C**. Mais c'est une expression constante en **C++**.

Exemple : **int const** nel=15**;**

```
...
double t1[nel+1], t2[nel+2]; //acceptées en C++, mais refusées en C (#define ).
```

2.2. Type VOID*

En **C**, le type générique **VOID*** est compatible avec les autres types pointeurs, et ceci dans les deux sens :

Exemple : **void*** gen**;**
int* adi**;** ...

Ces 2 affectations sont légales en **C** : gen=adi**;**
adi=gen**;**

En **C++**, seule la conversion d'un pointeur quelconque en **VOID*** peut être implicite.

Exemple : **void*** gen**;**
int* adi**;** **...**
gen=adi**;** (est acceptée)
adi=gen**;** (est refusée)

Appel explicit de la conversion **void*** => **int*** (utiliser un opérateur de **cast**)

adi=**(int*)**gen;

Chap III : SPECIFICITES DE C++

C++ dispose d'un certain nombre de spécificités qui ne sont pas axées sur la **POO**.

3.1. Nouvelles entrées-sorties en C++

La bibliothèque **iostream** : **cout** (afficher à l'écran) et **cin** (saisir au clavier). L'avantage de ces fonctions est qu'elles peuvent être facilement surchargées**.**

```
#include<iostream>
#include<string>
using namespace std;
int main()
{  int n=25; bool ok=true; unsigned int q=63000; char c='a';
   double y=12.3456789e16; string ch(″bonjour″); int *ad=&n;
   cout<<"valeur de n : "<<n<<endl;
   cout<< "caractere c : "<<c<<endl;
   cout<< "valeur de q : "<<q<<endl;
   cout<< "valeur de y : "<<y<<endl;
   cout<< "chaine ch : "<<ch<<endl;
   cout<<"adresse de n : "<<ad<<endl;
   cout<<"adresse de ch : "<<(void*)ch<<endl ;
  return 0;
}
```

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

int main()
{  int i=0; char tc[80];
   cout<<“ Saisir une suite de caracteres termines par un point\n”;
   do
      cin>>tc[i];
   while(tc[i++] != ‘.’)
   cout<<“Voici les caracteres effectivement lus : \n”;
   i=0;
   do
      cout<<tc[i];
   while(tc[i++] != ‘.’);
   return 0;
}
```

```
#include <iostream>
#include <string>
using namespace std;
int main()
{ cout << "Combien vaut pi ?" << endl;
   double piUtilisateur(-1.); //On crée une case mémoire pour stocker un nombre réel
   cin >> piUtilisateur; //Et on remplit cette case avec ce qu'écrit l'utilisateur
   cin.ignore();
   cout << "Quel est votre nom ?" << endl;
   string nomUtilisateur("Sans nom"); //On crée une case mémoire pour contenir une chaine de caractères
   getline(cin, nomUtilisateur); //On remplit cette case avec toute la ligne que l'utilisateur a écrit
   cout <<"Vous vous appelez "<<nomUtilisateur<<" et vous pensez que pi vaut " <<piUtilisateur<< "." <<endl;
   return 0;
}
```

Quand on mélange l'utilisation des chevrons et de getline(), il faut toujours placer l'instruction cin.ignore() après la ligne cin>>valeur. C'est une règle à apprendre.

3.2. Nouvelle forme de commentaire

Aux commentaires entre **/*…*/**, on y ajoute ceux débutant par // et se finissant à la fin de la ligne.

3.3. Emplacement libre des déclarations

En **C++**, il n'est plus nécessaire de regrouper au début les déclarations dans une fonction (bloc).

Exemple1 :
```
int main()
{ int  n;.....
  n=....; ....
  int q = 2*n-1 ;….
  return 1;
 }
```

Exemple2 :
```
for (int i = 0 ; ... ;...)
{.......}
```

La portée de **i** est limitée à la partie de la fonction main() suivant sa déclaration.

Cet exemple est équivalent à :
```
int i;
 for ( i = 0 ; .... ; ....)
{.....}
```

3.4. Résolution de portée

Si vous disposez de 2 ou plus entités (données ou méthodes) de même nom en **C** seule la plus locale est accessible. En **C++**, la notation **nom::variable** permet de préciser de quelle **variable** on parle (**nom** correspond à une classe), **::variable** peut accéder à une variable globale.

3.5. Transmission par référence en C++

C++ peut lui-même prendre en charge la transmission des arguments par **adresse** (transmission par **référence**), il suffit d'ajouter le signe **&** dans le prototype et l'en-tête de la fonction.

```
#include<iostream>
using namespace std;
int main()
{  void echange(int &, int &);
   int n=10, p=20;
   cout<<“avant appel : ”<<n<<” ”<<p<<”\n”;
   echange(n, p);
   cout<<“apres appel : ”<<n<<” ”<<p<<”\n”;
   return 1;
}

void echange(int &a, int &b)
{  int c;
   cout<<“debut echange :  ”<<a<<” ”<<b<<”\n”;
   c=a; a=b; b=c;
   cout<<“fin echange : ”<<a<<” ”<<b<<”\n”;
}
```

a) Transmission par référence de la valeur de retour d'une fonction

Son véritable intérêt apparaît sur la surcharge d'opérateurs.

```
#include<iostream>
using namespace std;
int main()
{ int & max(int &, int &);
  int m=44; int n=22;
  cout<<m<<" "<<n<<", "<<max(m, n)<<endl;
  max(m, n)=55;
  cout<<m<<"  "<<n<<",  "<<max(m, n)<<endl;
  retrun 1;
}

int & max(int &m, int &n)
{   return (m>n?m:n);
}
```

La fonction **max()** renvoie une référence à la plus grande de 2 variables que l'on lui passe. Puisque la valeur de retour est une référence, **max(m,n)** agit comme une référence à **m**.

b) Il est possible de **déclarer un identificateur** comme **référence** d'une autre variable :

int n**;**
int &p = n**;** //**p** est une référence à la variable **n**.

c) Initialisation de référence

Déclaration :**int &**p = n**;** est une déclaration de référence **p** accompagnée d'une initialisation.
Il n'est pas possible de déclarer une référence sans l'initialiser : **int &**p**;** //incorrect
Une déclaration : **int &**p=3**;** //est incorrect.
Ecrivez **const int &**p=3**;** //Création d'une variable temporaire.
Comme **int** temp=3**;**
int &p=temp**;**

d**) Transmission d'un argument ou d'une valeur de retour est une initialisation**

Soit : **void** fct**(int &);**
les appels suivants sont incorrects : fct**(**3**);**
const int c=15**;** fct**(c); .....**

En effet, de tels appels sont assimilés à des initialisations de références (à quelque chose de non constante) avec une référence à une constante.

- Soit : fct1**(const int &)**;
les appels suivants sont corrects : fct**(**3**);**
const int c=15**;** fct**(c); .....**

En effet, **fct1** a prévu de recevoir une référence à quelque chose de constante ; pas de risque de modification.

3.5. Arguments par défaut

En déclarant des valeurs par défaut des arguments d'une fonction, les arguments réels peuvent être omis.

Exemple : **float** fct**(char, int**=10**, float**=0.0**);**

Des appels fct**('a');** sera équivalent à fct('a', 10, 0.0);
fct**('b',** 12 **)**; -//- fct('b', 12, 0.0);
fct**('c' ,** 14**,** 2**);** appel normal.

Mais l'appel **fct();** est incorrect.

a) Arguments par défaut et transmission par référence

```
#include<iostream>
using namespace std;
int n;
int main()
{ void fct(int, int &=n ); int p;
   fct(10, p);
   fct(10);
   return 1;
}
void fct( int a, int &b)
{ b=a;
}
```

La notation **int & =n** signifie que :

- l'argument correspondant, de type **int**, est transmis par référence,

-cette référence reçoit une valeur par défaut correspond à l'adresse de **n**.

Si nous avons souhaité que cet argument **transmis par référence** reçoive par défaut la valeur 3, nous aurons déclaré ainsi fct : **void fct(int, int = 3);** //erreur. C'est possible si le second argument avait reçu l'attribut const : **void fct (int, const int = 3) ;** // correct. En effet, ceci a entraîné la création d'une variable temporaire, contenant la valeur 3, dont on aurait transmis la référence comme argument par défaut.

b) Arguments par défaut et transmission par pointeur

```
#include<iostream>
using namespace std;
int n;
int main()
{ void fct(int, int* =&n); int p;
   fct(10, &p);
   fct(10);
   return 1;
 }
void fct(int a, int *adb)
{ *adb=a;
 }
```

3.6. Surcharge de fonctions

Un même symbole possède plusieurs significations différentes, le choix de l'une des significations se faisant en fonction du **contexte**.

Exemple : **int puissance(int, int)** et **double puissance(double, double).**

Les 2 fonctions ayant une implémentation différente suivant le type de données, le compilateur choisit en fonction de types de données.

On peut même surcharger les opérateurs classiques du langage **C**.

Mise en œuvre de la surcharge de fonctions

Définir et utiliser 2 fonctions nommées toutes les deux **sosie**. Le premier a un argument de type **entier** et le second de type **réel**.

```
#include<iostream>
using namespace std;
void sosie(int);
void sosie(double);
int main()
{ int n=5, p; double x=2.5;
   sosie(n); sosie(x);
   return 1;
}
void sosie(int a)
{ cout<<"sosie numéro I : a =  "<<a<<"\n";
}
void sosie(double a)
{ cout<<"sosie numéro II : a =  "<<a<<"\n";
}
```

On peut appeler la fonction **sosie** avec un argument de type autre que celui déclaré. Par exemple: **char** c**; float** y**; long** q**;**

Avec les déclarations :
void affiche**(char*);** //affiche I
void affiche**(void*);** //affiche II
char* ad1**; double*** ad2**;**
- l'instruction affiche**(**ad1**);** // affiche I
- l'instruction affiche**(**ad2**);** // affiche II, après conversion de **ad2** en **void***.

Si l'on a affaire à des fonctions comportant plusieurs arguments.
Par exemple : **void** essai**(int, double);** // essai I
void essai**(double, int);** // essai II
int n**,** p**; double** z**; char** c**;**
- l'instruction **essai(n, z)** appellera la fonction essai I.
- l'instruction **essai(c, z)** appellera la fonction essai I.
- l'instruction **essai(n, p)** une erreur de compilation, ambiguïté : 2 possibilités existent :
  * convertir **p** en double sans modifier **n** et appeler essai I.
  * ou convertir **n** en double sans modifier **p** et appeler essai II.

Le compilateur recherche la "meilleure" correspondance possible pour choisir une fonction, en utilisant les conversions :
- **promotions numériques** : char et short ➔ int ; float ➔ double ;
- **conversions dites « standards »** : int➔char ; double ➔float ; double ➔int.

3.7. Gestion dynamique de la mémoire

En C++, les fonctions **malloc** et **free** sont remplacées par **new** et **delete**.

3.7.1. Syntaxe et rôle de NEW

Syntaxe : **new type**;

Le résultat est :

- un pointeur (de type **type***), lorsque l'allocation a réussi,
- un pointeur nul (**NULL,** NULL=0), dans le cas contraire.

Exemple : **int*** ad**;**

ad**=new int;**

permet d'allouer l'espace mémoire nécessaire pour un élément du type indiqué et d'affecter à **ad** l'adresse correspondante.

Autre syntaxe : **new type[**n**];** //Où **n** désigne une expression entière quelconque.

new alloue l'emplacement nécessaire pour **n** éléments du type indiqué et fournit en résultat un pointeur (de type **type***) sur le 1^{ier} élément de ce tableau.

Allouer un emplacement pour un tableau à plusieurs dimensions : **new type[**n**][**10**];**

Dans ce cas, **new** fournit un pointeur de type **type*[**10**]**.

La 1^{ière} dimension peut être une expression entière quelconque, les autres doivent obligatoirement être des expressions constantes.

3.7.2. syntaxe et rôle de l'opérateur DELETE

- **delete adresse;** //libère un emplacement préalablement alloué par **new** à l'adresse indiquée.
- **delete[] adresse;** //n'intervient que dans le cas de tableaux d'objets.

3.8. Fonctions en ligne

En **C**, une fonction courte avec un temps d'exécution rapide, s'utilise avec une **macro**.
En **C++**, on utilise une **fonction en ligne**.

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{ inline double norme(double [ ]);
   double v1[3], v2[3]; int i;
   for(i=0; i<3; i++)
  { v1[i]=i; v2[i]=2 *i–1;
   }
   cout << "norme de v1 : "<<norme(v1)<<"-norme v2 : "<<norme(v2);
   return 1;
}
inline double norme(double vec[ ])
{  int i; double s=0;
   for(i=0; i<3; i++)s += vec[i]+vec[i];
   return sqrt(s);
}
```

Le mot **INLINE** demande au compilateur de traiter la fonction **norme** différemment d'une fonction ordinaire : à chaque appel de **norme**, il devra incorporer, au sein du programme, les instructions correspondantes.

Une fonction en ligne doit être définie dans le même fichier source que celui où l'on l'utilise. Elle **ne peut pas être compilée séparément** (tableau 1).

Tableau 1 : Comparaison entre macro, fonction et fonction en ligne

	AVANTAGES	INCONVENIENTS
Macro	- économie de temps d'exécution	- risque d'effets de bord non désirés - pas de compilation séparée possible
Fonction	- compilation séparée possible	- perte d'espace mémoire - temps d'exécution lent
Fonction en ligne	- économie de temps d'exécution	- risque d'effets de bord - pas de compilation séparée

3.9. En-tête cmath

Le fichier d'en-tête **cmath** permet d'accéder à des fonctions mathématiques .

sqrt(valeur) délivre la racine carée de **valeur**.

Autres fonctions dans cmath

Sinus	**sin()**	resultat = sin(valeur);
Cosinus	**cos()**	resultat = cos(valeur);
Tangente	**tan()**	resultat = tan(valeur);
Exponentielle	**exp()**	resultat = exp(valeur);
Logarithme népérien	**log()**	resultat = log(valeur);
Logarithme en base 10	**log10()**	resultat = log10(valeur);
Valeur absolue	**fabs()**	resultat = fabs(valeur); …Il existe encore d'autres **fct**.

Fonction puissance : **pow()** prend 2 arguments, par exemple : **resultat = pow(4, 5);**

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{  double a(2), b(4);
   cout<<"Bienvenue dans le programme de calcul de a^b !"<<endl;
   cout<<"Donnez une valeur pour a : "; cin>>a;
   cout<<"Donnez une valeur pour b : "; cin>>b;
   double const resultat(pow(a, b)); //Ou double const resultat=pow(a, b);
   cout <<a<<"^"<<b<<" = "<<resultat<<endl;
   return 0;
}
```

3.10. Booléens et combinaisons de conditions

Le type booléen se déclare avec **bool.**
Par exemple : **bool** adulte**(true);**
if(adulte**==true)cout**<<"Vous etes un adulte !"<<**endl;**
Ou **if(**adulte**)cout**<<"Vous etes un adulte !"<<**endl;**

Combiner des conditions
On peut faire plusieurs tests au sein d'une même **if** en utilisant : **&&** (ET), || (OU) et **!** (NON).

Test ET
Tester si la personne est adulte **et** a au moins 1 enfant.
if(adulte **&& nbEnfants >= 1)**
Ou bien **if (adulte == true &&** nbEnfants>=1**)**.

Test OU (||). Tester si le nombre d'enfants est égal à 1 OU 2.
if(nbEnfants==1 || nbEnfants==2**)**

Test NON (!). Ce signe placé **avant** une condition peut dire **Si cela n'est pas vrai**.
if (!adulte)… se traduirait par **Si la personne n'est pas adulte**.

3.11. Tableaux

Des variables du même type qui jouent le même rôle : liste des utilisateurs d'un site web (**string)** ou 10 meilleurs scores du jeu (**int**)…, sont regroupées dans un paquet appelé **tableau**. Il y a 2 types des tableaux : la taille est connue à l'avance (**10 scores)** et ceux dont la taille peut varier en permanence (**liste des visiteurs d'un site web)** et qui ne cesse de grandir.

Tableaux statiques

Afficher la liste des 3 meilleurs scores des joueurs : liste des noms de joueurs et liste des scores.

```
int main()
{ string nomMeilleurJoueur1("Nanoc");
  string nomMeilleurJoueur2("lbert Einstein");
  string nomMeilleurJoueur3("Archimede");
  int meilleurScore1(118218);
  int meilleurScore2(100432);
  int meilleurScore3(64523);
  cout<<"1) "<<nomMeilleurJoueur1<<" "<<meilleurScore1<<endl;
  cout<<"2) "<<nomMeilleurJoueur2<<" "<<meilleurScore2<<endl;
  cout<<"3) "<<nomMeilleurJoueur3<< " "<<meilleurScore3<<endl; return 1;
}
```

Avec 100 joueurs, nous avons besoin de 100 variables (100 places dans le tableau) appelée **taille** du tableau.

Si la taille du tableau reste inchangée et est fixée dans le code, on parle d'un **tableau statique**.

Déclarer un tableau statique : indique le type, le nom choisi et entre crochets, la taille du tableau.

Exemple : calculez la moyenne des notes de l'année : mettre toutes les notes dans un tableau et utiliser une boucle **for** pour le calcul de la moyenne.

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

int main()
{ void creerNotes(double [], int);
   double moyenne(double [], int);
   int nombreNotes; double notes[100];
   cout<<"Saisir nombre de notes ? "; cin>>nombreNotes;
   creerNotes(notes, nombreNotes);
   cout<<"Votre moyenne est : "<<moyenne(notes, nombreNotes)<<endl;
}
void creerNotes(double v[], int nb){
  for(int i(0); i<nb; ++i){ cout<<"notes ? "; cin>>v[i];
                                  }
}
double moyenne(double v[], int nb){ double moy(0);
   for(int i(0); i<nb; ++i)moy += notes[i];
   moy /= nb;
   return moy;
}
```

Tableaux et fonctions

La 1ière restriction est **qu'on ne peut pas écrire une fonction qui renvoie un tableau statique**.
La 2ième restriction est qu'un tableau statique est **toujours passé par référence**. Mais on n'utilise pas l'esperluette (**&**). Exemple : fonction qui reçoit un tableau en argument.

```
void fonction(double tableau[])
{ //…
}
```

Pour parcourir le tableau avec une boucle **for**, il faut **ajouter** un **2ième argument** qui est **la taille**.

```
void fonction(double tableau[], int tailleTableau)
{ //…
}
```

Ecrire une fonction moyenne() qui calcule la moyenne des valeurs d'un tableau.

```
double moyenne(double tableau[], int tailleTableau)
{ double moyenne(0);
  for(int i(0); i<tailleTableau; ++i)moyenne += tableau[i];
  moyenne /= tailleTableau;
  return moyenne;
}
```

Tableaux dynamiques

Sont des tableaux dont la taille peut varier.

Déclarer un tableau dynamique

#include <vector> permet d'utiliser ces tableaux. On parle de **vector** non **tableau dynamique**.

Déclarer un tableau dynamique de 5 entiers.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{ vector<int> tableau(5);
   return 0;
}
```

On peut directement remplir toutes les cases du tableau en ajoutant un 2ième argument entre les parenthèses : **vector<int>** tableau**(**5**,** 3**);** *//Crée un tableau de 5 entiers valant tous 3*

vector<string> listeNoms**(**12**,** "Sans nom"**);** *//Crée un tableau de 12 strings valant toutes « Sans nom »*

On peut déclarer un tableau sans cases en ne mettant pas de parenthèses.

vector<double> tableau**;** *//Crée un tableau de 0 nombre réel*

Ce sont des tableaux dont la taille peut varier. On peut ajouter des cases par la suite.

Accéder aux éléments d'un tableau

On utilise les crochets. Réécrivez l'exemple précédent avec un **vector**.

```
int const nombreMeilleursScores(5); //La taille du tableau
vector<int> meilleursScores(nombreMeilleursScores); //Déclaration du tableau
meilleursScores[0] = 118218; //Remplissage de la première case
meilleursScores[1] = 100432; //Remplissage de la deuxième case
meilleursScores[2] = 87347; //Remplissage de la troisième case
meilleursScores[3] = 64523; //Remplissage de la quatrième case
meilleursScores[4] = 31415; //Remplissage de la cinquième case
```

Changer la taille

La fonction **push_back()** fait varier la taille d'un tableau en ajoutant des cases à la fin d'un tableau.

```
vector<int> tableau(3, 2); //Un tableau de 3 entiers valant tous 2
tableau.push_back(8); //On ajoute une 4ème case qui contient la valeur 8
tableau.push_back(7); //On ajoute une 5ème case qui contient la valeur 7
tableau.push_back(14);
```

La fonction **pop_back()** supprime la dernière case d'un tableau.

```
vector<int> tableau(3, 2); //Un tableau de 3 entiers valant tous 2
tableau.pop_back(); //Il reste 2 cases
tableau.pop_back(); //Il reste 1 case
```

La fonction **size()** récupère un entier correspondant au nombre d'éléments de tableau.

```
vector<int> tableau(5, 4); //Un tableau de 5 entiers valant tous 4
int const taille(tableau.size()); //Une variable constante taille contient la taille du tableau.
```

Retour sur l'exercice calcul des moyennes mais en le réécrivant en utilisant un tableau dynamique.

```
#include <iostream>
#include <vector>
using namespace std;
int main(){ vector<double> notes; //Un tableau vide
  notes.push_back(12.5); //On ajoute des cases avec les notes
  notes.push_back(19.5);  notes.push_back(6);
  notes.push_back(12); notes.push_back(14.5); notes.push_back(15);
  double moyenne(0);
  for(int i(0); i<notes.size(); ++i)moyenne += notes[i];
  moyenne /= notes.size();
  cout<<"Votre moyenne est : "<<moyenne<<endl;
  return 0;
}
```

Vector et fonctions

Pour passer 1 tableau dynamique en argument à une fonction on met **vector**<**type**> en argument., On n'a pas besoin d'ajouter un 2ième argument pour la taille du tableau en utilisant **size().**

void fonction**(vector<int>** tab**)**
{ *//…*
}

Utiliser le passage par référence constante optimise la copie. En effet, si le tableau contient beaucoup d'éléments, le copier prendra du temps. Il vaut mieux utiliser cette astuce.

void fonction**(vector<int> const &**tab**)**
{ *//…*
}

Le tableau dynamique ne peut pas être modifié. Pour changer le contenu du tableau, il faut utiliser un passage par référence sans **const**.

Tableaux multi-dimensionnels

On peut créer des tableaux de tableaux ! Un tableau de tableau est une grille de variables.

Déclaration d'un tableau multi-dimensionnel

type nomTableau**[**tailleX**][**tailleY**];**

Pour reproduire un tableau entier de 2 dimensions. on doit déclarer le tableau : **int** tableau**[**5**][**4**];**

Ou mieux encore, en déclarant des constantes : **int const** tailleX**(**5**); int const** tailleY**(**4**);**
int tableau**[**tailleX**][**tailleY**];**

Strings comme tableaux : les chaînes de caractères sont des tableaux de lettres. Il y a de points communs avec les **vector**.

Accéder aux lettres : l'intérêt de voir une chaîne de caractères comme un tableau de lettres est qu'on peut accéder à ces lettres et les modifier. On utilise aussi les crochets.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{ string nomUtilisateur("Julien");
  cout<<"Vous etes " <<nomUtilisateur<<"."<<endl;
  nomUtilisateur[0]='L'; //On modifie la première lettre
  nomUtilisateur[2]='c'; //On modifie la troisième lettre
  cout<<"Ah non, vous etes "<<nomUtilisateur<<"!"<< endl;
  return 0;
}
```

Fonctions : **size()** donne le nombre de lettres et **push_back()** ajoute des lettres à la fin.

```
string  texte("Portez ce whisky au vieux juge blond qui fume."); //46 caractères
cout << "Cette phrase contient " << texte.size() << " lettres." <<endl;
```

On peut ajouter **plusieurs lettres** d'un coup. Et on utilise le **+=.**

```
#include <iostream>
#include <string>
using namespace std;
int main(){ string prenom("Albert"); string nom("Einstein"); string total; //Une chaîne vide
  total += prenom; //On ajoute le prénom à la chaîne vide
  total += " "; //Puis un espace
  total += nom; //Et finalement le nom de famille
  cout<<"Vous vous appelez "<<total<<"."<<endl; return 0;
}
```

3.12. Lire et modifier des fichiers

Des logiciels **bloc-note**, un **IDE** ou un **tableur** sont des **programmes** qui lisent des **fichiers** et écrivent dedans. Et même dans le monde des jeux vidéo, on a besoin de cela. En somme, un **programme** qui ne sait pas interagir avec des **fichiers** risque d'être très limité.

Écrire dans un fichier

Avant de manipuler les fichiers on doit les **ouvrir**. Les **flux** sont les moyens de **communication** d'un programme avec l'**extérieur**, par exemple : **flux** vers les **fichiers**.

L'en-tête fstream

Pour pouvoir utiliser des fichiers, il faut spécifier : **#include<fstream>**.

iostream : «flux d'entrées/sorties».

fstream correspond à file stream, «flux vers les fichiers».

Il faut un **flux** par **fichier**. Créez un flux sortant c'est-à-dire un flux qui écrit dans un fichier.

Ouvrir un fichier en écriture

Les **flux** sont en réalité des **objets**. On les déclare comme une **variable** dont le type est **ofstream** et la valeur serait le **chemin** d'**accès** du fichier à lire. **Chemin absolu**, l'emplacement du fichier depuis la racine du disque. C**hemin relatif,** l'emplacement du fichier depuis l'endroit où se situe le programme sur le disque.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() //Déclaration d'un flux permettant d'écrire dans le fichier C:/Nanoc/scores.txt
{ ofstream monFlux("C:/Nanoc/scores.txt");
  return 0;
}
```

Si le **fichier** n'existait pas**,** le programme **le créerait automatiquement !**
Le nom du fichier est contenu dans une chaîne de caractères **string**. Utiliser la fonction **c_str()** lors de l'ouverture du fichier.

```
string const nomFichier("C:/Nanoc/scores.txt");
ofstream monFlux(nomFichier.c_str()); //Déclaration d'un flux écrivant dans un fichier.
```

Des problèmes peuvent survenir lors de l'ouverture d'un fichier : ce n'est pas votre fichier ou le disque dur est plein. Il faut **toujours** tester , en utilisant la syntaxe **if(**monFlux**)**.

```
 //Si tout est OK, on peut utiliser le fichier.
```

Écrire dans un flux : pour envoyer des informations dans un flux, on utilise les chevrons <<.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main(){ string const nomFichier("C:/Nanoc/scores.txt");
  ofstream monFlux(nomFichier.c_str());
   if(monFlux)
   { monFlux<<"Bonjour, je suis une phrase écrite dans un fichier."<<endl;
      monFlux<<42.1337<<endl; int age(23);
      monFlux<<"J'ai "<<age<<" ans."<<endl;
    }
   else cout<<"ERREUR: impossible d'ouvrir le fichier."<<endl;
   return 0;
}
```

En exécutant ce programme, un fichier **scores.txt** se trouve sur mon disque .

Exercice : écrire un programme qui demande à l'utilisateur son nom et son âge et qui écrit ces données dans un fichier.

Différents modes d'ouverture

Si le fichier existe déjà. Il sera supprimé et remplacé par ce que vous écrivez. Pour pouvoir écrire à la fin d'un fichier, lors de l'ouverture on ajoute un 2ième paramètre à la création du flux :

ofstream monFlux**("C:/**Nanoc/scores.txt**", ios::app);** //app (**append)** « ajouter à la fin ».

Lire un fichier

Ouvrir un fichier en lecture…Même principe, mais on utilise **ifstream**. Il faut tester l'ouverture.

ifstream monFlux**("C:/**Nanoc/C++/data.txt**");** *//Ouverture d'un fichier en lecture*
if(monFlux**){** *//Tout est prêt pour la lecture.*
}
else cout<<"ERREUR: Impossible d'ouvrir le fichier en lecture." **<<endl;**

Trois manières de lire un fichier : par ligne **getline()**, par mot (>>), par caractère **get()**.

- Lire ligne par ligne récupère une ligne entière et la stocke dans une chaîne de caractères.

string ligne**;** **getline(**monFlux**,** ligne**);** *//On lit une ligne complète*

- Lire mot par mot

double nombre**;** monFlux>>nombre**;** *//Lit un nombre à virgule depuis le fichier*
string mot**;** monFlux>>mot**;** *//Lit un mot depuis le fichier*

Cette méthode lit ce qui se trouve entre l'endroit où l'on se situe dans le fichier et l'espace suivant. Ce qui est lu est traduit en **double, int** ou **string** selon le type de variable dans lequel on écrit.

- Lire caractère par caractère

char a**;** monFlux**.get(**a**);** ce code lit **une seule lettre** et la stocke dans la variable **a**.

Cette méthode lit **tous** les caractères y compris espace, retour à la ligne et tabulation.

Lire un fichier en entier

getline() renvoie un booléen; **true** : la lecture continue ; **false** : on est à la fin du fichier ou il y a eu une erreur. Il faut s'arrêter de lire. On lit le fichier tant qu'on n'a pas atteint la fin **(while).**

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{  ifstream fichier("C:/Nanoc/fichier.txt");
    if(fichier){ //L'ouverture s'est bien passée, on peut donc lire
      string ligne; //Une variable pour stocker les lignes lues
      while(getline(fichier, ligne)) //Tant qu'on n'est pas à la fin, on lit
        cout<<ligne<<endl; //On l'affiche dans la console ou on fait autre chose avec cette ligne.
     }
    else cout<<"ERREUR: Impossible d'ouvrir le fichier en lecture."<< endl;
   return 0;
}
```

Une fois les lignes lues, on les manipule facilement (afficher les lignes ou les utiliser autrement).

les fichiers ouverts sont automatiquement refermés lorsque l'on sort du bloc où le flux est déclaré.

```
void f()
{ ofstream flux("C:/Nanoc/data.txt"); //Le fichier est ouvert
//Utilisation du fichier
} //Lorsque l'on sort du bloc, le fichier est automatiquement refermé
```

Fermer le fichier avant sa fermeture automatique, on utilise la fonction **close()** des flux**.**

```
void f(){ ofstream flux("C:/Nanoc/data.txt"); //Le fichier est ouvert et on l'utilise
  flux.close();  } //On referme le fichier
```
Il est possible de retarder l'ouverture d'un fichier après la déclaration du flux en utilisant la fonction **open()**.
```
void f(){ ofstream flux; //Un flux sans fichier associé
  flux.open("C:/Nanoc/data.txt"); //On ouvre le fichier C:/Nanoc/data.txt et on l'utilise
  flux.close(); //On referme le fichier
```
} Ouvrir directement le fichier et le laisser se fermer automatiquement suffit.

Curseur dans le fichier : **ifstream fichier("C:/Nanoc/scores.txt") ;** Le fichier **C:/Nanoc/scores.txt** est ouvert et le curseur est placé au début du fichier. Si on lit le premier mot du fichier, on obtient la chaîne de caractères « Bonjour, ». Le curseur se déplace jusqu'au début du mot suivant. Le mot suivant qui peut être lu c'est «je», puis «suis», ainsi de suite jusqu'à la fin. On a lu un fichier **dans l'ordre**. Il existe des moyens de se déplacer dans un fichier : connaître sa position, se déplacer.

Connaître sa position : il existe une fonction permettant de savoir à quel caractère du fichier on se situe : **tellg()**, **tellp()**.
```
ofstream fichier("C:/Nanoc/data.txt");
int position = fichier.tellp(); //On récupére la position
cout << "Nous nous situons a la " << position << "eme caractere du fichier." << endl;
```
Se déplacer : **seekg()**, **seekp()** à 2 arguments : une position dans le fichier et 1 nombre de caractères à ajouter à cette position.
```
flux.seekp(nombreCaracteres, position);
```
Trois positions possibles sont : début du fichier : **ios::beg** ; fin du fichier : **ios::end** ; position actuelle : **ios::cur**.

Exemple : se placer 10 caractères après le début du fichier, **flux.seekp(10, ios::beg)**; Aller 20 caractères plus loin que l'endroit où se situe le curseur, **flux.seekp(20, ios::cur)**;

Connaître la taille d'un fichier : on se déplace à la fin et on demande au flux de nous dire où il se trouve.
```
#include <iostream>
#include <fstream>
using namespace std;
int main(){ ifstream fichier("C:/Nanoc/meilleursScores.txt"); //On ouvre le fichier
  fichier.seekg(0, ios::end); //On se déplace à la fin du fichier
  int taille;
  taille=fichier.tellg();  //On récupère la position qui correspond donc a la taille du fichier !
  cout<<"Taille du fichier : " <<taille<<" octets." << endl;
  return 0;
}
```

IV. CLASSE, CONSTRUCTEUR ET DESTRUCTEUR

Les possibilités de **C++** reposent sur le concept de **classe**. **C++** autorise à n'encapsuler qu'une partie seulement des données d'une classe.

4.1 Notion de classe

Une classe est une structure dans laquelle seulement certains membres données ou fonctions sont

publics : accessibles de « l'extérieur », les autres membres étant dits

protégés : accessibles aux membres de la classe et les classes dérivées ou

privés : accessibles uniquement aux autres membres de la classe.

4.1.1. Déclaration de la classe POINT

```
class point
{ [ private : ]          // déclaration des membres privés
            int x; int y;
    public :             // déclaration des membres publics
            void initialise(int, int);
            void deplace(int, int);
            void affiche();
} ;
```

4.1.2. Définition des fonctions membres de la classe point

```
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 point::affiche()
{ cout<<"Je suis en "<<x<<"  "<<y<<endl;
 }
```
Dans la fonction **initialise()**, l'affectation **x=abs;** signifie que **abs** est un argument. Mais **x** n'est ni argument, ni variable locale. Cette association est réalisée par le **point::** de l'entête.

4.1.3. Utilisation de la classe point

```
int main()
{ point a, b;
  a.initialise(5, 2); a.affiche();
  a.deplace(-2, 4); a.affiche();
  b.initialise(1, -1); b.affiche(); return 1;
}
```
- **a** et **b** sont des instances de la classe **point** ou des objets de type **point**. Les données sont complètement encapsulées. Ainsi dans la fonction **main()** l'accès direct à un membre **a** est interdit : **a.x=5;** // erreur.
Cela se fait par l'intermédiaire des fonctions membres : **a.initialise().**
a.initialise(5, 2); appelle la fonction membre **initialise()** de la classe à laquelle appartient l'objet **a**.

- Les mots clés **public, private** et **protected** peuvent apparaître à plusieurs reprises dans la définition d'une classe.
- Si aucun de ces mots n'apparaît au début de la définition, tout se passe comme si **private** y avait été placé.
- Si aucun de ces 2 mots n'apparaît dans la définition d'une classe, tous ses membres seront donc privés, donc inaccessibles.

Exemple :
```
class point
{ int x;
  public :  int y; .....
};
```

4.2 Affectation entre objets

C++ autorise l'affectation d'un objet d'1 type donné à 1 autre objet de même type : il y a recopie des valeurs des champs de données (publics ou privés). Toutefois, si parmi ces champs, se trouvent des pointeurs, les emplacements pointés ne seront pas recopiés. Si un tel effet est nécessaire, on devra surcharger l'opérateur d'affectation pour la classe concernée.

Exemple :
```
class point
{       int x;
        public : int y;.....
};
point a, b;
```

L'action **b=a;** la recopie des valeurs des membres **x** et **y** de a dans les membres données correspondant de b. Mais l'écriture des actions : **b.x=a.x; b.y=a.y;** // b.x = a.x est interdite.

4.3 Constructeur et destructeur

Un constr. : une fct membre ayant le même nom que sa classe. Un constr ne renvoie pas de valeur (**void** ne doit pas figurer devant sa déclaration ou sa définition). Dès qu'une classe comporte un constr, on doit créer un objet en fournissant des valeurs pour les arguments requis par ce constr. Le constr est appelé après l'allocation de l'espace mémoire destiné à l'objet.

a) **Exemple de classe comportant un constr :** soit la classe **point** précédente, transformons la fonction membre **initialise** en un constructeur en la renommant **point** :

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

A partir du moment où un constructeur est défini, il doit être appelé lors de la création de l'objet :
point a(1, 3);
Un destructeur est une fct membre portant le même nom que sa classe, précédé du symbole tilda (~). Le destr est appelé avant la libération de l'espace mémoire alloué à l'objet. Un destr ne peut pas comporter d'arguments et il ne renvoie pas de valeur.
b) Construction et destruction d'objet soit le prog définissant une classe nommé **test** comportant 2 fct membres. En outre, le membre donnée **num** initialisé par le constr, permet d'identifier l'objet concerné. Créons des objets automatiques de type **test** à 2 endroits différents : dans la fct **main()**, puis dans une fonction **fct()** appelée par **main()**.

```
#include<iostream>
using namespace std;
class test
{  public : int num;
                test(int);    ~test();
};
test::test(int nb)
{ num=nb; cout<<"++appel constructeur – num="<<num<<"\n";
 }
test::~test()
{ cout<<"—Appel destructeur – num ="<<num<<"\n"; }
int main()
{ void fct(int);
   test a(1);
   for(int i=1; i<=2; i++)fct(i); return 1;
}
void  fct(int p)
{ test x(2*p);
}
```

c) Rôle du constructeur et du destructeur le travail réalisé par le constructeur peut être beaucoup plus élaboré que l'initialisation de l'objet à l'aide des valeurs qu'il avait reçues en argument : un programme exploitant une classe appelé **hasard**, dans laquelle le constructeur fabrique 10 valeurs entières aléatoires qu'il range dans le membre donné **val**.

```
#include<iostream>
#include<stdlib>
using namespace std;
class hasard
{  int val[10];
    public : hasard(int);
                void affiche();
};
hasard::hasard(int max)
{ int i;
   for(i=0; i<10; i++)val[i]=rand()%max;
 }
void hasard::affiche()
{ for(i=0; i<10; i++)cout<<val[i]<<" "; cout<<"\n";
}
int main()
{ hasard suite1(5); suite1.affiche();
   hasard suite2(12); suite2.affiche();
   return 1;
 }
```

En pratique, on préférera disposer d'une classe dans laquelle le nombre de valeurs peut-être fourni en argument du constructeur. Dans ce cas, il est préférable que l'espace soit alloué dynamiquement au lieu d'être surdimentionné. Cette allocation dynamique est effectuée par le constructeur lui-même.

```
#include<iostream>
#include<stdlib>
using namespace std;
class hasard
{  int nbval; int* val;
    public : hasard(int, int);
                ~hasard();
                void affiche();
};
hasard::hasard(int nb, int max)
{ int i;
   val=new int[nbval=nb];
   for(i=0; i<nb; i++)val[i]=rand()%max;
 }
hasard::~hasard()
{ delete val; }
void hasard::affiche()
{ for(i=0; i<nbval; i++)cout<<val[i]<<" "; cout<<"\n";
}
int main()
{ hasard suite1(10, 5); suite1.affiche();
   hasard suite2(6, 12); suite2.affiche();
   return 1;
 }
```

4.4. Membres donnée statiques

Dans un même programme, pour différents objets crées dans une même classe, chaque objet possède ses propres membres données.

class exple1
{ int n**; float** x**;**
…..
};
….
exple1 a**,** b**;** // **a** possède ses propres données **n** et **x, b** possède ses propres données **n** et **x**.

Si l'on veut que la valeur d'un membre donnée s'applique à tous les objets de la classe, on déclare le membre donnée avec l'attribut **static**.

class exple2
{ static int n**; float** x**;**
…
};
….
exple2 a**,** b**;** // le membre donnée **n** est partagé par **a** et **b**. Mais **a** possède son propre membre **x** et **b** a son propre membre **x**.

Les membres données statiques sont des variables globales dont la portée est limitée à la classe. Il n'est pas possible d'écrire dans la classe **exple2** : **... static int n=0;** // interdit !!

On peut introduire soit dans la fonction **main()**, soit à la suite de définition de la classe (mais en-dehors de celle-ci), l'instruction : .... **int exple2::n=5**;

Ecrire un programme contenant la classe **cpte_objet**, qui permet de connaître, à tout moment, le nombre d'objets existant.

Solution : nous déclarons avec l'attribut **static** le membre **ctr**, sa valeur est incrémenté de 1 à chaque appel du constructeur et décrémenté de 1 à chaque appel du destructeur.

```
#include<iostream>
using namespace std;
class cpte_obj
{ static int ctr;
   public : cpte_obj();
  ~cpte_obj();
};
int cpte_obj::ctr=0;
cpte_obj::cpte_obj()
{ cout<<"++construction : il y a maintenant "<<++ctr<<"objets\n";
}
cpte_obj::~cpte_obj()
{ cout<<"--destruction : il reste maintenant "<<--ctr<<"objets\n";
}
int main(){ void fct();
  cpte_obj a; fct();
  cpte_obj b; return 1;
}
void fct(){ cpte_obj u, v;
}
```

4.5 Exploitation d'une classe

Dans la pratique, dans le souci de réutilisabilité, la classe sera fournie comme un composant séparé destiné à être employé par plusieurs programmes. Cela signifie qu'un utilisateur potentiel (client) de cette classe disposera :

- d'un fichier en-tête contenant la déclaration de la classe,
- d'un fichier source contenant la définition de la classe.

Exemple : le concepteur de la classe **point** précédent pourra créer le fichier en-tête.

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

Le fichier d'entête se nomme **point.h**. le concepteur fabriquera alors un module objet, en compilant la définition de la classe **point** qui est enregistré dans le fichier **point1.cpp** :

```
#include< iostream >
#include"point.h"
using namespace std;
point::point(int abs, int ord)
{ x=abs; y=ord;  }
void point::deplace(int dx, int dy)
{ x=x+dx; y=y+dy; }
void point::affiche()
{ cout<<"je suis en "<<x<<"  "<<y<<"\n";}
```

Pour exécuter son programme, l'utilisateur inclura la définition de la classe point « point1.cpp » dans le fichier source contenant son programme.

```
#include"point1.cpp"
int main()
{ point a(1, 2), b(2, 3);
  b.affiche(); a.deplace(2, 2); a.affiche();
  return 1;
}
```

Protection contre les inclusions multiples

Quelques circonstances peuvent amener l'utilisateur d'une classe à inclure plusieurs fois un même fichier en-tête, lors de la compilation d'un même fichier source, par exemple le cas dans la situation d'objets membre et de classes dérivées.

On réglera ce problème en protégeant systématiquement tout fichier en-tête des inclusions multiples par une technique de compilation conditionnelle :

```
#ifndef POINT.H
#define POINT.H
  /* déclaration de la classe point */
#endif
```

V. PROPRIETES DES FONCTIONS MEMBRE

Le C++ offre ces possibilités aux fonctions membres : surcharge, arguments par défaut, fonction en ligne, transmission par référence.

5.1 Surcharge des fonctions membre

C++ offre la possibilité de surcharger les fonctions membres d'une classe et le constructeur lui-même. Nous surchargeons :

* le constructeur *point* ; le choix du bon constructeur se faisant suivant le nombre d'arguments :

- 0 argument : les deux coordonnées attribuées au point construit sont toutes deux nulles.
- 1 argument : il sert de valeur commune aux deux coordonnées.
- 2 arguments: c'est le cas usuel.

•la fonction *affiche ()* de manière qu'on puisse l'appeler :

- sans argument,

-avec un argument de type *chaîne* : dans ce cas, elle affiche le texte correspondant avec les coordonnées du point.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
  public : point();
           point(int);
           point(int, int);
           void affiche();
           void affiche(char*);
};
```

```
point::point()
{  x=0; y=0;
}
point::point(int abs)
{  x=y=abs;
}
point::point(int abs, int ord)
{   x=abs; y=ord;
}
void point::affiche()
{  cout<<"je suis en : "<<x<<" "<<y<<"\n";
}
void point::affiche(char *message)
{   cout<<message; affiche();
}
int main(){  point a; a.affiche();
    point b(5); b.affiche("Point b- ");
    point c(3, 12); c.affiche("Hello---");
    return 1;
}
```

5.2 Arguments par défaut

Nous allons modifier l'exemple précédent pour que la classe ***point*** ne possède plus qu'une seule fonction ***affiche()*** à un seul argument de type chaîne.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
  public : point();
           point(int);
           point(int, int);
           void affiche(char* = "");
};
point::point() {
  x=0; y=0; }
point::point(int abs){
  x=y=abs; }
point::point(int abs, int ord){
  x=abs; y=ord; }
void point::affiche(char *message) { cout<<message<<"je suis en : "<<x<<" "<<y<<"\n";
 }
int main(){ point a; a.affiche();
  point b(5); b.affiche("Point b- ");
  point c(3, 12); c.affiche("Hello---");
  return 0;
}
```

Cette simplification ne peut pas être appliquée au constructeur **point**. Néanmoins, en admettant que, dans le constructeur **point** à un seul argument, ce dernier représente seulement l'abscisse du point auquel on aurait alors attribué une ordonnée nulle, nous aurions pu alors définir un seul constructeur :

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

5.3. Fonctions membre en ligne

Rendre **en ligne** une fonction membre, on peut :

- soit fournir directement la définition de la fonction dans la déclaration même de la classe.

Exemple : rendre **en ligne** les trois constructeurs points.

```
#include< iostream >
using namespace std;
class point
{           int x, y;
public :   point(){ x=0; y=0; }
           point(int abs){x=y=abs; }
           point(int abs, int ord) {x=abs; y=ord; }
.....
```

- soit procéder comme pour une fonction ″ordinaire″ en fournissant une définition en dehors de la déclaration de la classe ; dans ce cas, le qualicatif **INLINE** doit apparaître, à la fois devant la déclaration et devant l'en-tête.

```
class point
{.....
   public : inline point();
  ......
}
inline point::point() { x=0; y=0;
}
......
```

5.4. Cas des objets transmis en argument d'une fonction membre

Une fonction membre reçoit **implicitement** l'adresse de l'objet l'ayant appelé. Mais il est toujours possible de lui transmettre **explicitement** un argument (ou plusieurs) du type de sa classe ou du type d'une autre classe. Dans le premier cas, la fonction membre aura **accès aux membres privés** de l'argument en question. Dans le second, la fonction membre **n'aura accès qu'aux membres publics** de l'argument.

Un tel argument peut être transmis par **valeur**, par **adresse** ou par **référence**. Avec *la* **transmission par valeur**, il y a recopie des valeurs membres donnée dans un emplacement local à la fonction appelée. Des problèmes peuvent surgir, dès lors que l'objet transmis en argument contient des **pointeurs** sur des parties dynamiques. Ils seront réglés par l'emploi d'**un constructeur par recopie**.

Exemple d'objets transmis en argument à une fonction membre : supposons que nous souhaitions, au sein de la classe **point**, introduire une fonction membre **coïncide()**, chargée de détecter la coïncidence éventuelle de 2 points.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
   public : point(int abs=0, int ord=0){
                x=abs; y=ord;
              }
              bool coïncide(point);
};
bool point::coïncide(point pt){ if((pt.x==x)&&(pt.y==y))return true;
  else return false; }
int main(){ point a, b(1), c(1, 0);
  cout<<"a et b"<<a.coïncide(b)<<"ou"<<b.coïncide(a)<<"\n";
  cout<<"b et c"<<b.coïncide(c)<<"ou"<<c.coïncide(b)<<"\n";
  return 0;
}
```

Remarques: Nous aurions pu écrire **coïncide** de la manière : **return((pt.x==x) && (pt.y==y)) ;**

- On pourrait penser qu'on viole le principe de l'**encapsulation** dans la mesure où, lorsque l'on appelle la fonction **coïncide** pour l'objet **a** (a.coïncide(b)), elle est autorisée à accéder aux données de **b**. En fait, en C++, n'importe quelle fonction membre d'une classe, peut accéder à n'importe quel membre (public ou privé) de n'importe quel objet de cette classe. En C++, l'unité de protection est la classe et non l'objet.

- L'objet *pt* était transmis à *coïncide* par valeur : ***a.coïncide(b) ;*** les valeurs de *b* sont recopiées dans un emplacement (de type point) local à *coïncide* (nommé *pt*).

a) Transmission de l'adresse d'un objet

Dans la classe **point**, modifions la fonction **coïncide()**.

```
class point {…
... bool coïncide(point*);
};
…
bool point::coïncide(point* adpt)
{ if((adpt->x==x)&&(adpt->y==y))return true;
   else return false;
}
int main(){…
    a.coïncide(&b); ou b.coïncide(&a);
 …
 return 0; }
```

b) Transmission par référence

l'emploi des **références** permet de mettre en place une transmission par adresse, sans avoir à en prendre en charge soi-même la gestion. Elle simplifie l'écriture de la fonction et ses appels.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
   public : point(int abs=0, int ord=0){x=abs; y=ord; }
                bool coïncide(point &);
};
```

```
bool point::coïncide(point &pt)
{  if((pt.x==x)&&(pt.y==y))return true;
   else return false;
}
int main(){  point a, b(1), c(1, 0);
  cout<<"a et b"<<a.coïncide(b)<<"ou"<<b.coïncide(a)<<"\n";
  cout<<"b et c"<<b.coïncide(c)<<"ou"<<c.coïncide(b)<<"\n";
  return 0;
}
```

Remarque : Pour éviter des risques des effets de bord au sein de la fonction membre concernée (par adresse ou par référence), on peut toujours employer le qualificatif **CONST**.

en-tête

bool point::coïncide (const point *adpt) ou // par référence

bool point::coïncide (const point &adpt)

prototype

bool coïncide (const point*) ou // par référence **bool coïncide (const point&)**

5.6. Cas de fonction membre fournissant un objet en retour

Une fonction membre peut fournir comme valeur de retour un objet de type de sa classe ou d'un autre type classe (dans ce dernier cas, elle n'accédera qu'aux membres publics de l'objet en question). La transmission peut se faire par valeur, par adresse ou par référence.

La **transmission par valeur** implique une recopie qui pose les mêmes problèmes que ceux évoqués ci-dessus pour des objets comportant des pointeurs sur des parties dynamiques.

La **transmission par adresse** ou **par référence** doivent être utilisées avec beaucoup de précautions, dans ce cas, on renvoie l'adresse d'un objet alloué automatiquement, c'est-à-dire dont la durée de vie coïncide avec celle de la fonction. Il vaut mieux éviter qu'il s'agisse d'un objet local à la fonction.
Exemple : une fonction membre nommée **symétrique** pourrait être introduite dans une classe **point** pour fournir en retour un point symétrique de celui l'ayant appelé :

```
point point::symetrique()
{  point res;
   res.x=-x; res.y=-y;
   return res;
}
```

Il a été nécessaire de créer un objet automatique (res). Il ne serait pas conseillé de prévoir ici une transmission par référence.

5.7 Autoréférence : THIS

Une fonction membre d'une classe peut recevoir une information lui permettant d'accéder à l'objet l'ayant appelé, sous-forme d'un paramètre implicite de type pointeur, nommé **THIS**. Cette adresse sera manipulée explicitement dans un cas de gestion d'une liste chaînée d'objets de même nature : pour écrire une fonction membre insérant un nouvel objet (supposé transmis en argument implicite), il faudra bien placer son adresse dans l'objet précédent de la liste. Pour résoudre de tels problèmes, C++ a prévu le mot : **THIS**.
Au sein d'une fonction membre, **THIS** représente **un pointeur sur l'objet ayant appelé** ladite fonction membre, ***THIS** est l'objet lui-même.

Exemple : dans la classe point, la fonction **affiche()** fournit l'adresse de l'objet l'ayant appelé.

```
#include<iostream>
using namespace std;
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<< "\n";
}
int main(){ point a(5), b(3, 15);
  a.affiche(); b.affiche();
  return 0;
}
```

Remarque : A titre indicatif, la fonction **coïncide()** transmit l'adresse d'un objet.

```
bool point::coïncide(point* adpt)
{ if((this->x==adpt->x)&&(this->y==adpt->y))return true;
  else return false;
}
```

La symétrie du problème y apparaît très clairement.

Ce serait moins le cas si l'on écrivait ainsi la fonction *coïncide()* dans la transmission d'un argument par valeur.

```
bool point::coïncide(point pt)
{ if((this->x==pt.x))&&(this->y==pt.y))return 1;
  else return 0;
 }
```

5.8. Fonctions membres statiques

Lorsqu'une fonction membre a une action indépendante d'un quelconque objet de sa classe, on peut la déclarer avec l'attribut **static**. Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant son nom du nom de la classe suivi de (::).

Exemple : reprenons le programme du ch4.4 dans lequel nous avons introduit une fonction membre statique nommée **compte()**, affichant simplement le nombre d'objets de sa classe.

```
#include<iostream>
using namespace std;
class cpte_obj
{  static int ctr;
    public : cpte_obj();
   ~cpte_obj();
   static void compte();
};
int cpte_obj::ctr=0;
cpte_obj()::cpte_obj()
{ cout<<"++construction : il y a maintenant "<<++ctr<<"objets\n";
}
```

```
cpte_obj::~cpte_obj()
{ cout<<"--destruction : il reste maintenant "<<--ctr<<"objets\n";
}
void cpte_obj::compte()
{ cout<<"Appel compte : il y a  "<<ctr<<"objets\n";
}
int main()
{ void fct();
  cpte_obj::compte();
  cpte_obj a;
  cpte_obj::compte(); fct(); cpte_obj::compte();
  cpte_obj b; cpte_obj::compte();
  return 0;
}
void fct()
{   cpte_obj u, v;
}
```

5.9. Fonctions membre constantes

On peut déclarer des objets constants à l'aide du qualificatif CONST. Dans ce cas, seules les fonctions membre déclarées et définies avec ce même qualificatif peuvent recevoir implicitement ou explicitement en argument un objet constant.

Les seules fonctions membres qui peuvent être appelées par les objets constants sont les constructeurs et les destructeurs. Pour les autres, on doit les déclarer comme des fonctions membres constantes.

Exemple : **class** point

```
{  int x, y;
   public : point( ... );
            void affiche() const;
            void deplace( ... );
};
....
point a;
const point c;
....
a.affiche();          // erreur, appel non autorisé
c.affiche();          // correcte
a.deplace(...);       // correcte
```

Remarques :

- Ce mécanisme s'applique aux fonctions membres volatiles et aux objets membres volatiles (mot clé VOLATILE). Il suffit de transposer tout ce qui vient d'être dit en remplaçant le mot clé CONST par le mot clé VOLATILE.

- Il est possible de surcharger une fonction membre en se basant sur la présence ou l'absence du qualificatif CONST. Ainsi dans la classe *point*, nous pouvons définir ces 2 fonctions :

```
void affiche() const; // affiche I
void affiche();       // affiche II
```

Avec ces déclarations : **point** a**; const point** c**;**

La fonction **a.affiche()** appellera la II, tandis que **c.affiche()** appellera la fonction I.

CHAP.VI. CONSTRUCTION DESTRUCTION ET INITIALISATION DES OBJETS

Il existe 3 genres d'objets : **statiques, automatiques et dynamiques** auxquels il faudra ajouter les objets **temporaires**.

6.1. Objets automatiques et statiques

Objets **automatiques** sont créés par une déclaration dans une fonction ou au sein d'un bloc. Ils sont crées au moment de l'exécution de la déclaration et détruits lorsque l'on sort de la fonction ou du bloc.

Objets **statiques** sont créés par une déclaration en dehors de toute fonction ou précédée du mot clé **static** dans une fonction ou un bloc. Ils sont crées avant l'entrée dans la fonction **main()** et détruits après la fin de son exécution.

6.1.1. Appel du constructeur et du destructeur

Cas d'objets statiques, dynamiques ou automatiques : s'il y a appel du **constructeur**, celui-ci a lieu après l'allocation de l'emplacement mémoire destiné à l'objet. S'il existe un **destructeur**, ce dernier est appelé avant la libération de l'espace mémoire associé à l'objet.

Exemple : un programme crée et détruit des objets **statiques** et **automatiques**. Une classe **point**, dans laquelle le constructeur et le destructeur affichent un message permettant de repérer :

- le moment de leur appel,
- l'objet concerné.

Construction, destruction et initialisation d'objets

```
#include<iostream>
using namespace std;
class point
{ int x, y;
  public : point(int abs, int ord)
           { x=abs; y=ord;
             cout<<"++Construction d'un point : " <<x<<" "<<y<<endl;
           }
           ~point()
           { cout<<"--Destruction du point : "<<x<<" "<<y<<endl;
           }
};
point a(1, 1);
int main()
{  cout<<"***Debut main***\n";
   point b(10, 10);
   for(i=1; i<=3; i++)
  { cout<<"**Boucle le tour numero "<<i<<endl;
     point b(i, 2*i);
   }
   cout<<"***Fin main***\n"; return 0;
}
```

6.1.2. Listes d'initialisations des constructeurs

La plupart des constructeurs se contentent d'initialiser les données membres de l'objet.

```
class point
{ int x, y;
  public : point(int abs, int ord)
           { x=abs; y=ord;
             cout<<"Construction d'un point : "<<x<<" "<<y<<endl;
           }
  …
};
```

A cet effet, le C++ simplifie le code d'initialisation en utilisant une **liste d'initialisation.**

```
point(int abs, int ord) : x(abs), y(ord)
{ cout <<" Construction d'un point : "….;
}
```

La liste commence par 2 points (:) et le corps de la fonction peut être vide.

```
point(int abs, int ord) : x(abs), y(ord){}
```

6.1.3. Fonctions d'accès

Sont des fonctions membres publiques donnant un **accès** aux données (privées ou publiques) en lecture seule.

```
#include<iostream>
using namespace std;
class point
{  int x, y;
   public : point(int abs=0, int ord=0) : x(abs), y(ord)
            {
             }
            int abscisse()const
            {  return x;
            }
            int ordonnee()const
            {  return y;
            }
};
int main(){   point a(22, 7);
    cout<<" abscisse = "<< a.abscisse()<<endl;
    cout<<"ordonnee = "<<a.ordonnee()<<endl;
    return 0;
}
```

6.2 Objets temporaires

Une classe disposant d'un constructeur peut être appelé explicitement. Dans ce cas, il y a création d'un **objet temporaire**, qui pourra être automatiquement détruit dès lors qu'il ne sera plus utile.

La classe **point** possède le constructeur : **point(float, float)** et on a déclaré **point a**;

on peut écrire **a=point(1.5, 2.25);**

- Création d'**objet temporaire** de type point (**at** a une adresse, mais inaccessible au programme), avec appel du constructeur **point** (transmission des arguments spécifiés 1.5 et 2.25).
- La recopie de cet objet temporaire dans **a** (affectation d'un objet dans un autre).

6.3. Objets dynamiques

Sont crées par **NEW**, auquel on fournit, le cas échéant, les valeurs des arguments destinés à un constructeur.

Soit la classe **point** possédant le constructeur **point(float, float)**;

point *adp; ....

adp=new point(2.5, 5.32); une allocation dynamique d'espace mémoire pour un élément de type **point** et affecte son adresse au pointeur **adp.**

new retourne l'adresse de l'objet créé et la valeur **NULL** si l'objet n'a pas été créé.

Si le constructeur **point** est sans arguments, on écrit : **adp=new point**;

Si l'on veut créer un tableau de 100 objets dynamiques : **adp=new point[100];**

Si **point** possède une méthode appelée **affiche(),** on l'appelle par **adp->affiche();**

Les objets dynamiques sont détruits à la demande en utilisant **delete** : **delete adp;**

Après l'allocation dynamique de l'emplacement mémoire, **NEW** appellera un **constructeur** de l'objet et avant la libération de l'emplacement mémoire correspondant, l'opérateur **DELET**E appellera le **destructeur.**

Un programme crée dynamiquement un objet de type **point** dans la fonction **main(),** le détruit dans une fonction **fct()**. Des messages permettent de mettre en évidence les moments d'appel des **constructeur** et **destructeur**.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
  public : point(int abs, int ord)
           { x=abs; y=ord;
             cout<<"++ Appel constructeur\n";
           }
           ~point()
           {cout<<"--Appel destructeur\n";
            }
};
int main(){  void fct(point*);
   point *adr; cout << "***Debut main***\n";
   adr=new point(3, 7); fct(adr);
   cout<<"***Fin main***\n"; return 0;
}
void fct(point *adp)
{ cout<<"***Debut fct***\n";
  delete adp; cout<<"***Fin fct***\n";
}
```

6.4. Initialisation d'objets

On distingue **initialisation** et **affectation.** Il y a i**nitialisation** d'un objet dans une :
- déclaration d'un objet avec **initialisateur.** Ce dernier est une **expression d'un type quelconque**.
Exemple : **point a=5;**
point b=a; équivalente à **point b(a);**
- transmission d'un objet par **valeur en un argument** d'appel d'une fonction,
- transmission d'un objet par valeur en **valeur de retour** d'une fonction.

Constructeurs particuliers
-Constructeur **implicite : point()** est automatiquement appelé chaque fois qu'un objet est déclaré **point a();**
-Constr **recopie: point(point &)** ou **point(const point &)** est implicitement appelé chaque fois qu'1 objet est copié (dupliqué ou initialisé) : **point b(a)** ; **point b=a**; ou lorsqu'une fonction retourne 1 valeur de type point.
* Si aucun constructeur de la forme **type(type &)** ou **type(const type &)** n'a pas été défini, il y aura appel du constructeur de **recopie par défaut**. Ce dernier recopie les valeurs des différents membres données de l'objet comme fait **l'affectation**. Des problèmes peuvent alors se poser dès lors que l'objet contient des pointeurs sur des parties dynamiques : on aura affaire à une « **copie superficielle »,** seules les valeurs des pointeurs sont copiées, les emplacements pointés ne le seront pas. Ils risquent alors d'être détruits 2 fois.
* Si un constructeur de la forme **type(type &)** ou mieux encore **type**(**const type &**) existe, il sera appelé. Le **constructeur de recopie** doit prévoir la recopie de tous les membres de l'objet.
•Il est préférable de définir un **constructeur de recopie** si vous devez surcharger l'opérateur d'affectation.

Remarque : lorsqu'on initialise un objet par un simple appel d'un constructeur par exemple :
point a=point(1.5, 4.3);
Il n'y a pas d'appel de **constructeur de recopie**, précédé de la création d'un objet **temporaire.**
La déclaration précédente est équivalente à **point a(1.5, 4.3);**

Exemple d'utilisation du constructeur de recopie : objet **transmis par valeur.**

Une classe **vect** permet de gérer de tableaux d'entiers de taille variable. L'utilisateur de cette classe déclare un tableau sous la forme : **vect t(dim); //dim** est une expression entière représentant sa taille.

La classe **vect** a :

- en membre donné, la taille du tableau et un pointeur sur ses éléments lesquels verront leurs emplacements alloués dynamiquement,
- un constructeur recevant un argument entier chargé de cette allocation dynamique,
- un destructeur libérant l'emplacement alloué par le constructeur.

a) Emploi d'un constructeur de recopie par défaut

Utiliser la classe **vect** avec des messages pour suivre à la trace les constructeurs et destructeurs d'objets.

Transmettons par valeur un objet de type **vect** à une fonction affichant un message indiquant son appel.

```
#include<iostream>
using namespace std;
class vect
{   int nelem; double *adr;
   public:vect(int n){  adr=new double[nelem=n];
            cout<<"++constructeur usuel–adresse objet : "<<this<<"-adresse vecteur : "<<adr<<endl;
            }
      ~vect(){cout<<"--destructeur usuel –adresse objet : "<<this<<"-adresse vecteur:"<<adr<<endl;
            delete adr;
         }
} ;
void fct(vect b){ cout<<"Appel de fct\n";
}
int main(){ vect a(5);
  fct(a); return 0;
}
```

b) Définition d'un constructeur de recopie

Pour que l'appel de **fct()** puisse créer intégralement un nouvel objet de type **vect**, avec ses membres donnée **nelem** et **adr**, mais aussi son propre emplacement de stockage de valeurs de tableau. //schéma

Nous définissons un **constructeur par recopie** : **vect(const vect &);** //ou **vect(vect &);** qui

- crée dynamiquement un nouvel emplacement dans lequel il recopie les valeurs correspondant à l'objet reçu en argument,
- renseigne convenablement les membres donnée du nouvel objet (**nelem**=valeur du membre **nelem** de l'objet reçu en argument, **adr** = adresse du nouvel emplacement).

```
#include<iostream>
using namespace std;
class vect
{  int nelem; double* adr;
   public : vect(int n){ … ;
                }
                ~vect(){ …;
                }
                vect(const vect &);
} ;
vect::vect(const vect & v)
{  adr=new double[nelem=v.nelem];
   int i; for(i=0; i<nelem; i++)adr[i]=v.adr[i];
   cout<<"++const. recopie –adresse objet : "<<this <<"-adresse vecteur : "<<adr<<endl;
}
void fct(vect b)
{…; }
int main(){ vect a(5);
  fct(a); return 0;
};
```

Remarques : le problème d'**initialisation** d'un objet de type **vect** par un autre objet du même type a été réglé, mais celui qui se poserait en cas d'**affectation** entre objets de type **vect** n'a pas été réglé. Ce dernier ne peut se résoudre que par la **surcharge** de l'opérateur (**=**).

6.5. Tableaux d'objets

Un tableau d'objets est un tableau possédant des éléments de type **classe**.

point courbe[20] ; crée un tableau **courbe** de 20 objets de type **point** en appelant, le cas échéant, le constructeur pour chacun d'entre eux.

Exemple de construction et d'initialisation d'un tableau d'objets.

```
#include<iostream>
using namespace std;
class point
{ int x, y;
  public :
    point(int abs=0, int ord=0)
    { x=abs; y=ord;
      cout<<"++construction point : "<<x<<" "<<y<<endl;
    }
    ~point()
    { cout<<"--destruction point : "<<x<<" "<<y<<endl;
    }
} ;
int main(){ int n=3; point courbe[5]={7, n, 2*n+5}; return 1;
}
```

Si la classe comporte un constructeur sans argument, celui-ci sera appelé successivement pour chacun des éléments (de type **point**) du tableau **courbe**.
Il est possible de faire une déclaration avec un initialisateur comportant une liste de valeurs; chaque valeur sera transmise à un constructeur approprié. Pour les tableaux de *classe* **automatique**, les valeurs de l'**initialisateur** peuvent être une expression quelconque. En outre, l'**initialisateur** peut comporter moins de valeurs que le tableau n'a d'éléments. Dans ce cas, il y a appel du **constructeur sans argument** pour les éléments auxquels ne correspond aucune valeur.

- Cas des tableaux dynamiques d'objets

Disposant d'une classe **point**, on peut créer dynamiquement un tableau de points en appelant new : **point *adcourbe = new point[20]** ; alloue l'emplacement mémoire nécessaire à 20 objets de type **point** et place l'adresse du premier de ces éléments dans **adcourbe**.
Pour détruire le tableau d'objets, on appellera **delete[] adcourbe** ou **delete[20] adcourbe** ;
Remarque : On peut compléter la déclaration d'un tableau d'objets par un **initialisateur** contenant une liste de valeurs. Chaque valeur est alors transmise à un constructeur approprié. Cette facilité ne peut toutefois pas s'appliquer aux **tableaux dynamiques**.

6.6. Objets membres

6.6.1. Construction d'objets contenant des objets membres

Une classe peut posséder un membre donné qui est lui-même de type **classe**.

Exemple : **class point**

```
{  double x, y;
   public : point( double,  double);.....
};
```

Nous définissons une classe **pointcol**, dont un membre est de type **point** :

```
class pointcol
{  point p;
   int couleur;
   public : pointcol(double, double, int);.....
};
```

Lors de la création d'un objet de type **pointcol**, il y aura appel du **constructeur de point**, puis appel d'un **constructeur de pointcol** ; ce dernier recevra les arguments qu'on aura mentionné dans l'en-tête du **constructeur point**. Dans la destruction d'un objet, ces appels sont inversés.

```
pointcol::pointcol (double abs, double ord, int coul ):p(abs, ord)
{ ....
 }
```

L'en-tête de **pointcol** spécifie, après les 2 points, la liste des arguments qui seront transmis à **point**.

```
#include<iostream>
using namespace std;
class point
{   int x, y;
   public : point(int abs=0, int ord=0){ x=abs; y=ord;
              cout<<"++construction point : "<<x<<" "<<y<<endl; }};
```

```
class pointcol
{  point p; int couleur;
   public : pointcol(int, int, int);
};
pointcol::pointcol(int abs, int ord, int coul):p(abs, ord)
{  couleur=coul;
   cout<<"++construction pointcol : "<<couleur<<endl;
 }
int main()
{  pointcol a(1, 3, 9); return 1;
}
```

Remarque : dans le cas d'objets comportant plusieurs objets membres, la sélection des arguments destinés aux différents constructeurs se fait en séparant chaque liste par une virgule :

```
class A                 class B
{ ……                  { .....
  A(int) ;                B(double, int) ;
  ....                    .....
} ;                     } ;
class C
{   A a1, B b, A a2 ;
     ...
    C(int n, int p, double x, int q, int r) : a1(p), b(x, q), a2(r)
    {.....}
} ;
```

6.6.2. Initialisation d’objets contenant des objets membres

Le rôle du **constructeur de recopie par défaut** est, dans le cas d’objets ne comportant pas d’objets membres, de recopier les valeurs des différents membres donnée.

Lorsque l’objet comporte des objets membres, le **constructeur par recopie par défaut** procède **membre** par **membre** ; cela signifie que si l’un des membres est lui-même un **objet**, il est recopié en appelant son **propre constructeur par recopie**.

CHAP VII. **FONCTIONS AMIES**

En C++, l'unité de protection est la **classe** et non pas l'objet. Une fonction **membre** d'une classe peut accéder à tous les membres **privés** de n'importe quel **objet** de sa classe.
Ces membres privés restent **inaccessibles** à n'importe quelle fonction membre d'une **autre** classe ou à n'importe quelle fonction **indépendante**.

La notion de **fonction amie**, ou ″la déclaration d'amitié″, permet de déclarer dans une classe, les fonctions que l'on **autorise** à accéder à ses membres **privés**. Les situations d'amitiés sont :
- fonction indépendante, amie d'une classe ;
- fonction membre d'une classe, amie d'une autre classe ;
- fonction amie de plusieurs classes ;
- toutes les fonctions membres d'une classe, amies d'une autre classe.

7.1. Fonction indépendante, amie d'une classe

Une fonction **coïncide** examine la ″coïncidence″ de 2 objets de type **point**.
Fonction **coïncide** : fonction indépendante amie de la classe **point**.

Syntaxe de la déclaration d'amitié d'une classe point :

```
class point
{   ...
    friend...coincide(...) ; ....
}
```

```
#include<iostream>
using namespace std;
class point
{  int x, y;
    public : point(int abs=0, int ord=0)
              { x=abs; y=ord;
                }
              friend int coincide(point, point);
} ;
int coincide(point p, point q)
{  if((p.x==q.x)&&(p.y==q.y))return 1;
    else return 0;
}
int main()
{ point a(1, 0), b(1), c;
   if(coincide(a, b))cout<<"a coincide avec b\n";
   else cout<<"a et b sont differents\n";
   if(coincide(a, c))cout<<"a coincide avec c\n";
   else cout<<"a et c sont differents\n"; return 0;
}
```

Remarques

- **Emplacement** de la déclaration d'amitié, au sein de la classe point, est absolument **indifférent**.
- Une fonction **amie** d'une classe possédera 1 ou plusieurs **arguments** ou 1 **valeur de retour** du type de cette **classe**. Ce n'est toutefois pas une obligation.
- Lorsqu'une fonction **amie** d'une classe fournit une **valeur de retour** du type de cette classe, il est fréquent que cette valeur soit celle d'un **objet local** à la fonction. Dans ce cas, il est **impératif** que sa **transmission** ait lieu **par valeur** ; dans le cas d'une transmission par **référence** (ou par adresse), la fonction **appelante** recevrait l'**adresse** d'un emplacement mémoire qui **aurait été libéré** à la sortie de la fonction.

7. 2. Autres situations d'amitié

7.2.1. Fonction membre d'une classe, amie d'une autre classe

Il suffit, dans la **déclaration d'amitié**, de **préciser la classe** à laquelle appartient la fonction concernée, à l'aide de l'opérateur de résolution de portée (**::**).

Exemple : soient 2 classes **A** et **B**, et une fonction membre **f** de la classe **B**, amie d'une autre classe **A**.

```
Class A                          Class B
{  ...                           {  ...
   friend int B::f( char, A) ;      int f( char , A);
   ...                              ...
} ;                              } ;

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

La fonction membre **f** de la classe **B** est autorisé à accéder aux membres privés de la classe **A**.
Pour compiler les déclarations d'une classe **A** contenant une déclaration d'amitié, le compilateur devra savoir que **B** est une classe : **class B ;**

7.2.2. Fonction amie de plusieurs classes

Rien n'empêche qu'une **fonction** (indépendante ou membre) fasse l'objet de déclaration d'**amitié** dans **différentes classes**.
Exemple : fonction indépendante **f** amie de 2 classes **A** et **B**.

```
class A                          class B
{...                             {...
friend void f( A, B) ;…            friend void f( A,B ) ;....
} ;                               };
                                  void f( A..., B...)
                                  { // on a accès aux membres privés de n'importe quel objet de type
                                    // A ou B
                                  }
```

La déclaration de **A** peut être compilée avant celle de **B**, en la faisant précéder de : **class B ;**

7.2.3 Fonctions membres d'une classe B amies d'une autre classe A

Plutôt que d'utiliser **autant** de déclaration d'**amitié** que de **fonctions membres**, on utilise dans **A** une déclaration globale : **friend class B ;** //Fonctions membre de classe **B** st amies de la classe **A**.
Ce type de la déclaration d'amitié évite d'avoir à fournir les entêtes des fonctions concernées.

7.3. Exemples

Réaliser une fonction permettant de déterminer le produit d'un vecteur (objet de classe **vect**) par une matrice (objet de classe **matrice**). Nous limitons aux fonctions membre : un constructeur pour **vect** et pour **matrice** et une fonction d'affichage **affiche()**.

Solutions adoptées :

1. emploi d'une fonction indépendante **prod** et amie des 2 classes **vect** et **matrice**,
2. emploi d'une fonction **prod** membre de **matrice** et amie de la classe **vect**.

```
#include<iostream>
  using namespace std;
  class matrice;
  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 prod(matrice, vect);
             void affiche(){  int i;
               for(i=0; i<3; i++)cout<<v[i]<<" ";
               cout<<endl;
             }
  } ;
```

```
class matrice
{  double mat[3][3];
   public : matrice(double t[3][3])
            {   int i, j;
                for(i=0; i<3; i++)
                   for(j=0; j<3; j++)
                      mat[i][j]=t[i][j];
             }
             friend vect prod(matrice, vect);
};
vect prod(matrice m, vect x)
{  int i, j; double som; vect res;
   for(i=0; i<3; i++)
   { for(j=0, som=0; j<3; j++)som += m.mat[i][j]*x.v[j];
     res.v[i]=som;
    }
   return res;
}
int 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=prod(a, w); res.affiche(); return 0;
}
```

```
#include<iostream>
using namespace std ;
class vect;
class matrice
{  double mat[3][3] ;
   public : matrice(double t[3][3])
            { int i, 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, 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;
}
int 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(); return 0;
} //14 32 50
```

VIII. **SURCHARGE D'OPERATEURS**

La surcharge de fonctions consiste à attribuer le même nom à des fonctions différentes.

C++ permet de surcharger des opérateurs existants, en leur donnant une nouvelle signification.

8.1. Mécanisme de surcharge d'opérateurs

Pour surcharger un opérateur existant **op**, on définit une fonction nommée **operator op.**

• Surcharge d'un opérateur avec une **fonction amie**

si **op** est un opérateur binaire, la notation **a op b** est équivalent à **operator op(a, b)**.

Exemple: **surcharge** de l'opérateur **+** pour des objets de type **point** avec une **fonction amie**.

```
#include<iostream>
using namespace std;
class point { int x, y;
    public : point(int abs=0, int ord=0){x=abs; y=ord; }
                friend point operator +(point, point);
                void affiche(){cout<<"Coordonnees "<<x<<" "<<y<< endl;
                }
};
point operator +(point a, point b){  point p;
     p.x=a.x+b.x; p.y=a.y+b.y; return p;
}
int main()
{ point a(1, 2); a.affiche();
   point b(2, 5); b.affiche();
   point c; c=a+b; c.affiche(); c=a+b+c; c.affiche(); return 0;
}
```

• Surcharge d'un opérateur avec une **fonction membre** : le 1ier opérande de l'opérateur correspondant au 1ier argument de la fonction **operator +**, sera transmis implicitement : c'est l'objet ayant appelé la fonction membre. Une expression **a+b** sera interprétée par le compilateur **a.operator(b)** ; le prototype est **point operator +(point);**

Surcharge de l'opérateur+ pour des objets de type **point** avec une **fonction membre**.

```
#include<iostream>
using namespace std;
class point {  int x, y;
  public :  point(int abs=0, int ord=0)
            { x=abs; y=ord;
             }
            point operator +(point);
            void affiche(){ cout<<"Coordonnees "<<x<<" "<<y<< endl;
            }
} ;
point operator +(point a)
{ point p;
   p.x=x+a.x; p.y=y+a.y; return p;
}
int main()
{  point a(1, 2); a.affiche();
    point b(2, 5); b.affiche();
    point c; c=a+b; c.affiche(); c=a+b+c; c.affiche(); return 0;
}
```

Dans le cas d'objets de grande taille, on peut envisager de faire appel au transfert par référence.
Le prototype de la fonction amie **operateur +** serait : **point operator +(point &a, point &b);**
La transmission par référence poserait un problème si l'on cherchait à l'appliquer à une valeur de retour. Si l'on cherche à protéger contre d'éventuelles modifications un argument transmis par référence, on pourra toujours faire appel au mot clé **CONST**.
point operator +(const point &a, const point &b);

8.2. Possibilités et limites de la surcharge d'opérateurs

On se limite aux opérateurs existants, en conservant leur pluralité (unaire ou binaire). Lorsque plusieurs opérateurs sont combinés au sein d'une même expression (qu'ils soient surchargés ou non), ils conservent leur priorité relative et leur associativité.

Pluralité	Opérateurs	Associativité
Unaire	()(1) [](1) ->(1)	→
	+ - ++ -- ! ~ * &	<-
	new delete (cast)	<-
Binaire	* / %	→
	+ -	→
	<< >>	→
	< <= > >=	→
	== !=	→
	& ^ |	→
	&& ||	→
	=(1), +=. -= *=. /= %=...&= ^= \= <<= >>=	←
	,	→

(1) Doit être surchargé comme **fonction membre**.
- On ne peut pas surcharger un opérateur que s'il comporte au moins un argument. Autrement dit, il doit s'agir :
* soit d'une fonction membre, auquel cas elle dispose obligatoirement d'un argument implicite du type de sa classe (this),
* soit d'une fonction indépendante (ou plutôt amie) possédant au moins un argument de type classe.
-Les opérateurs ., ::, ?:, #, ## ne peuvent pas être surchargés.

8.3. Cas particuliers

8.3.1. Cas des opérateurs ++ et --

Si **T** désigne un type classe quelconque :
- L'opérateur d'en-tête **T operator ++()** est utilisé en cas de notation préfixée,
- L'opérateur d'en-tête **T operator ++(int)** est utilisé en cas de notation postfixée.

Le second opérande de type **int** est fictif.
Exemple : Définir ++ pour qu'il incrémente d'une unité les 2 coordonnées du point et pour qu'il fournisse comme valeur :
Soit celle du point avant incrémentation (notation postfixée),
Soit celle du point après incrémentation (notation préfixée).

```
#include<iostream>
using namespace std;
class point{ int x, y;
  public : point(int abs=0, int ord=0){x=abs; y=ord;}
           point operator ++(){ x++; y++; return *this; }
           point operator ++(int){ point p = *this; x++; y++; return p;
           }
           void affiche(){cout<<x<<" "<<y<<endl;
           }
} ;
int main(){ point a1(2, 5); a2(2, 5), b;
  b=++a1; cout<<"a1:"; a1.affiche(); cout<<"b:"; b.afffiche();
  b=a2++; cout<<"a2:"; a2.affiche(); cout<<"b:"; b.afffiche(); return 0;
}
```

8.3.2. Opérateur = a une signification prédéfinie

A l'absence de surcharge explicite, l'opérateur = correspond à la recopie des valeurs de son second opérande dans le 1ier. Cette recopie est insatisfaisante si les objets concernés comportent des pointeurs sur des emplacements dynamiques. Dans ce cas, on doit surcharger l'opérateur =.

```
#include<iostream>
using namespace std;
class vect {  int nelem; double* adr;
  public : vect(int); void initialise();
           void operator =(const vect &);
           ~vect(){ delete adr; }; void print(char*);
} ;
```

```
vect::vect(int n)
{   adr=new double[nelem=n];
}
void vect::initialise()
{  double nombre;
   for(int i=0; i<nelem; i++) {
      cout<<"Saisir nombre ? " ; cin>>nombre; adr[i]=nombre; }
   cout<<endl;
}
void vect::operator =(const vect &v)
{  for(int i=0; i<nelem; i++)adr[i]=v.adr[i];
}
void vect::print(char* message)
{   int i;
   cout<<message<<endl;
   for(i=0; i<nelem; i++)cout<< "v["<<i+1<<"]="<<adr[i]<<endl;
}
int main()
{  vect a(5);
   a.initialise(); cout<<"Nom du vecteur\n"; a.print("vecteur a ");
   vect b(5); b=a; b.print("vecteur b"); return 0;
}
```

8.3.3. Opérateurs [], (), ->, new et delete

Doivent être définis comme fonctions membre. Soit une classe **vect.**

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

Cherchons à la munir d'outils permettant d'accéder à un élément de l'emplacement pointé par **adr**, à partir de la connaissance de sa position que l'on repèrera par un entier compris entre **0** et **nelem-1**.

```
#include<iostream>
using namespace std;
class vect
{  int nelem; int* adr;
   public : vect(int n){ adr=new int[nelem=n];
              }
            ~vect(){ delete adr;
              }
              int& operator [](int);
};
int& vect :: operator [](int  i){  return adr[i];
 }
int main(){   int i; vect a(5), b(5), c(3);
  for(i=0; i<5; i++){a[i]=i; b[i]=2*i;}
  for(i=0; i<3; i++)c[i]=a[i]+b[i];
  for(i=0; i<3; i++){cout<<c[i]<<" "; cout<<endl; return 0;
}
```

La surcharge de **new**, pour un type donné, se fait par une fonction de prototype :
void* operator new(size_t);
où **size_t** est un type entier spécial défini dans le fichier string.h, ou stddef.h ou stdlib.h.
void* operator new [](size_t); //pour les tableaux.

la surcharge de **delete**, pour un type donné, se fait par une fonction de prototype :
void operator delete(type *);
void operator delete [](type *); //pour les tableaux.
Exemple de surcharge de l'opérateur **new** et **delete** pour la classe point.

Remarques
Lorsqu'ils sont surchargés, les opérateurs **new** et **delete** ne peuvent pas s'appliquer à des tableaux d'objets.
Même lorsque l'on a surchargé, les opérateurs **new** et **delete** pour une classe, il reste possible de faire appel aux opérateurs **new** et **delete** usuels, en utilisant l'opérateur de résolution de portée (**::**).

```
#inlude<iostream>
#include<stddef.h>
using namespace std;
static int npt=0; static int npt_dyn=0;
class point {   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) { npt_dyn++;
                  cout << "il y a " <<npt_dyn<<"points dynamiques sur un\n";
                  return ::new char[sz];
               }
               void operator delete(void* dp){ npt_dyn--;
                  cout<<"il y a "<<npt_dyn<<"points dynamiques sur un\n";
                  ::delete dp;
               }
} ;
int 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;
  return 0;
 }
```

X. TECHNIQUE DE L'HERITAGE

L'**héritage** est à la base des possibilités de **réutilisation** de composants logiciels (**classes**). Il définit une nouvelle classe **B** (**dérivée)**, à partir d'une classe existante A (**de base)** ;

```
class B : public A  // ou private A ou protected A
{ // définition des membres supplémentaires (données ou fonctions)
  // ou redéfinition des membres existants dans A (données ou fonctions)
} ;
```

Avec **public** A, on parle de **dérivation publique** ;
Avec **private** A, on parle de **dérivation privée** ;
Avec **protected** A, on parle de **dérivation protégée** ;

Dérivation publique.

-Les membres publics de la classe de base sont accessibles à « tout le monde » : FM, FMA et user de la classe dérivée (FM=fonction membre, FMA=fonction amie, user=utilisateur).
-Les membres protégés de la classe de base sont accessibles aux FM, FMA, mais pas aux user de la classe dérivée.
- Les membres privés de la classe de base sont inaccessibles aux FM, FMA et user de la classe dérivée.

Dérivation privée.
-Un utilisateur de la classe dérivée n'a pas accès aux membres publics de sa classe de base.
-Les membres protégés de la classe de base restent accessibles aux FM et FMA.
-Les membres de la classe de base deviennent privés pour la classe dérivée : dans une classe, une méthode privée ne sera atteint de l'extérieur de la classe. Elle ne pourra être appelée que de l''intérieur dans d''autres méthodes d'une même classe.

Dérivation protégée.
Intermédiaire entre publique et privée
Les membres de la classe de base deviennent protégées dans la classe dérivée.

C++ autorise l'héritage **multiple** dans laquelle une classe est dérivée de plusieurs classes de base.

10.1. MISE EN OEUVRE DE L'HERITAGE

Soit **point** une classe de base. Nous définissons une classe **pointcol** destinée à manipuler des points colorés. Un tel point coloré peut être défini par ses coordonnées (objet de type **point** auquel on adjoint une information de couleur de type **short/ char**).
Dans ce cas, on définit **pointcol** comme une classe dérivée de **point**. En plus, nous prévoyons une fonction membre spécifique à **pointcol**, nommée **colore**, destinée à attribuer une couleur à un point. On enregistre la classe **point** dans un fichier **point.h**.

```
#include<iostream>
using namespace std;
class point
{  int x, 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+=dx; y+=dy;
}
void point::affiche()
{  cout<< "Je suis en "<<x<<" "<<y<<" \n ";
}
```

Dans un autre fichier, on enregistre la classe **pointcol.**

```
#include "point.h"
class pointcol : public point
{ short couleur;
  public : void colore(short cl)
             { couleur=cl;
               }
} ;
```

class pointcol : public point
pointcol est une classe dérivée de la classe de base **point**.
public : les membres publics de la classe **point** seront des membres publics de la classe **pointcol**.
pointcol p, q ; chaque objet **p ou q** peut faire appel :
- aux méthodes publiques de **pointcol** (**colore**),
-aux méthodes et aux constructeurs publics de la classe de base (**deplace, affiche, initialise**).

Utilisation de la classe **pointcol** dérivée de **point**.

```
#include<iostream>
#include "pointcol.h"
using namespace std;
int main()
{ pointcol p;
   p.initialise(10, 20); p.colore(5); p.affiche();
   p.deplace(2, 4); p.affiche(); return 0;
}
```

10.2. UTILISATION, DANS UNE CLASSE DERIVEE, DES MEMBRES DE LA CLASSE DE BASE

Grâce à l'emploi de **public**, les membres de **point** sont membres publics de **pointcol**. Mais avec l'appel de **affiche** pour un objet de type **pointcol,** nous n'obtenons aucune information sur sa couleur.
Pour améliorer cette situation, écrivons une **nouvelle** fonction membre publique de **pointcol**, censée à afficher à la fois les coordonnées et la couleur.

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

La fonction **affichec** membre de **pointcol**, aurait accès aux membres privés de **point**.

Règle : une classe dérivée n'a pas accès aux membres privés de sa classe de base.

Si la fonction **affichec** ne peut pas accéder directement aux données privées **x** et **y** de la classe de base **point**, elle peut faire appel à la fonction **affiche()** de cette classe.

```
void pointcol::affichec()
{ affiche() ;
   cout<<" et ma couleur est : "<<couleur<< "\n";
}
```

Nous pouvons aussi définir dans **pointcol**, une nouvelle fonction d'initialisation nommée **initialisec**, chargée d'attribuer des valeurs aux données **x, y** et **couleur**, à partir de 3 valeurs reçues en argument.

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

10.3. REDEFINITION DES FONCTIONS MEMBRES

Lorsqu'un membre est redéfini dans une classe dérivée, il est possible d'accéder aux membres de même nom de la classe de base en utilisant l'opérateur de résolution de portée (**::**).

Soit une classe **pointcol** dans laquelle les méthodes **initialise** et **affiche** sont redéfinies.

```
#include<iostream>
#include "point.h"
using namespace std;
class pointcol : public point
{   short couleur;
    public :  void initialise(int, int, short);
              void colore(short cl){ couleur=cl; }
              void affiche();
};
void pointcol::initialise(int abs, int ord, short cl)
{ point::initialise(abs, ord);
   couleur=cl;}
void pointcol::affiche()
{ point::affiche();
  cout<<" et ma couleur est : "<<couleur<<endl ; }
int main(){ pointcol p;
  p.initialise(10, 20, 5); p.affiche();
  p.point::affiche();
  p.deplace(2, 4); p.affiche();
  p.colore(2); p.affiche(); return 0;
}
```

Remarque : bien que cela soit d'un emploi moins courant, tout ce qui a été dit à propos de la surcharge de la fonction membre s'applique tout aussi bien aux membres données.

```
class A
{     …
      int a, char b ; …
} ;
class B : public A
{     float a ; …
} ;
```

B x ; x.a fera référence au membre **a** de type **float de B**.

x.A::a accédera au membre **a** de type **int** (hérité de A).

Le membre **a** défini dans **B** s'ajoute au membre **a** hérité de **A,** mais il ne le remplace pas.

10.4. APPEL DES CONSTRUCTEURS ET DES DESTRUCTEURS

Règles de l'**appel** d'un constructeur ou d'un destructeur d'une classe :

- s'il existe au moins un **constructeur**, toute **création** d'un objet entraînera l'appel d'un **constructeur**. Le choix du constructeur est fourni en fonction des informations fournies ; si aucun constructeur ne convient, il y a erreur de compilation : impossible de créer un objet.

-s'il n'existe aucun constructeur, il n'est pas possible de préciser des informations lors de la création d'un objet.

- s'il existe un destructeur, il sera appelé avant la destruction de l'objet.

10.4.1. Hiérarchisation des appels

B est une classe dérivée de **A** et chaque classe possède un constructeur et un destructeur.

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

Pour créer **objet** de **B** créer d'abord **objet** de **A** : faire appel au constructeur de **A**, puis celui de **B**. Lors de destruction d'un objet de **B**, il y aura appel du destructeur **B**, puis appel de celui de **A**.

10.4.2. Transmission d'informations entre constructeurs

Les informations fournies lors de la création d'un objet de **B** sont destinées à son constructeur. C++ a prévu la possibilité de 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.

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

Si l'on souhaite que **pointcol** retransmette à **point** les 2 1^{ières} informations reçues, on écrira :
pointcol (int abs, int ord, char cl) : point(abs, ord)
pointcol a(10,15, 3) ; entraînera : - l'appel de **point** qui recevra les arguments 10 et 15,
- l'appel de **pointcol** qui recevra les arguments 10, 15 et 3.
pointcol(int abs=0, int ord=0, char cl=1) : point(abs, ord) //Arguments par défaut de pointcol.
pointcol p(5) ; entraînera : - appel de **point** avec les arguments 5 et 0,
- appel de **pointcol** avec les arguments 5, 0 et 1.

```
#include<iostream>
using namespace std;
class point{  int x, y;
   public : point(int abs=0, int ord=0){ x=abs; y=ord;
              cout<<"++constructeur point : "<<x<<" "<<y<<"\n";
             }
            ~point(){  cout<<"--destruction point<<x<<" "<<y<<endl;
             }
} ;
class pointcol : public point {   short couleur;
    public : pointcol(int, int , short);
              ~pointcol(){ cout<<"—destr. Pointcol – couleur "<<couleur<<endl;
               }
    pointcol::pointcol(int abs=0, int ord=0, short cl=1):point(abs, ord){
        cout<<"++construction pointcol :"<<abs<<" "<<ord<<" " <<cl<< "\n" ; couleur=cl; };
int main(){ pointcol a(10, 15, 3); pointcol b(2, 3);
  pointcol c(12); pointcol d;
  pointcol *adr;
  adr=new pointcol(12, 25); delete adr; return 0;
}
```

Remarque : Un membre d'une classe dérivée n'a pas accès aux membres privés de la classe de base. C'est un aspect fondamental dans la conception de classes «réutilisables ».

10.4.3. Cas du constructeur par recopie

Le constructeur par recopie est appelée en cas :
- d'**initialisation** d'un **objet** par un objet de même type,
-de **transmission** de la **valeur** d'un objet en **argument** ou en **retour** d'une fonction.

* Si la classe dérivée **B** ne possède **pas** de constructeur par **recopie**, il y a appel du constructeur par recopie par défaut de **B**. Il y a :
- appel du constructeur par recopie de **A**,
-initialisation des membres données de **B** qui ne sont pas hérités de **A**.

* Si la classe dérivée **B** a défini un constructeur par **recopie**, ce dernier est naturellement appelé. Cette fois, la règle qui s'applique est celle de **transmission** entre **constructeur** :
- Si le constructeur **B** ne prévoit pas d'information pour un constructeur de **A** c-à-d l'en-tête est **B(B&x)**. Il y aura appel d'un constructeur sans argument, si aucun constructeur de la classe de base n'est mentionné dans l'en-tête ; dans ce cas, il est nécessaire que la **classe** de **base** dispose d'un tel **constructeur sans argument**, faute de quoi, on obtiendra une erreur de compilation.
- Si le constructeur **B** prévoit des informations pour un constructeur **A** : **B(B&x) : A(...).**
/*Il y aura appel du constructeur par recopie de **A**, auquel sera transmisse la partie de **B** héritée de **A** (possible uniquement grâce aux règles de compatibilité entre la classe de base et la classe dérivée). */

classe de base			dérivée publique		dérivée protégée		dérivée privée	
statut initial	accès FMA	accès user	nouveau statut	accès user	nouveau statut	accès user	nouveau statut	accès user
public	O	O	public	O	protégé	N	privé	N
protégé	O	N	protégé	N	protégé	N	privé	N
privé	O	N	privé	N	privé	N	privé	N

Accès FMA : accès aux fonctions membres ou amies de la classe.
Nouveau statut : statut qu'aura ce membre dans une éventuelle classe dérivée.
Accès user : accès à un utilisateur.
Transmission des informations entre constructeur par recopie de **pointcol** et celui de **point**.

```
#include<iostream>
using namespace std;
class point { int x, y;
   public : point(int abs=0, ord=0){
                x=abs; y=ord;
                cout<<"++point "<<x<<" "<<y<<"\n" ;
            }
            point(point &p){
                x=p.x; y=p.y;
                cout<<"CR point "<<x<<" "<<y<<"\n" ;
            }
} ;
```

```
class pointcol : public point {
   char coul;
   public : pointcol(int abs=0, int ord=0, int cl=1):point(abs, ord)
            {  coul=cl;
               cout<<"++pointcol "<<int(coul)<<endl;
             }
             pointcol(pointcol &p):point(p)
            {   coul=p.coul;
                cout<<"CR pointcol "<<int(coul)<<endl;
             }
} ;
int main(){ void fct(pointcol);
  pointcol a(2, 3, 4);  pointcol b=a; fct(a); return 1;
}
void fct(pointcol pc){ cout<< "***entree dans fct*** \n"; }
```

Appel explicite du constructeur par recopie de **point** au sein de celui de **pointcol**.

```
#include<iostream>
using namespace std;
 class point {  int x, y;
                  public : point(int abs=0, ord=0) {
                               x=abs; y=ord;
                               cout<<"++point "<<x<<" "<<y<<"\n"; }
                             point(point &p) {  x=p.x; y=p.y;
                               cout<<"constr recopie point "<<x<<" "<<y<<"\n" ; }
} ;
```

```
class pointcol private point {
  char coul;
  public : pointcol(int abs=0, int ord=0, int cl=1):point(abs, ord){ coul=cl;
                cout<<"pointcol "<<int(coul)<<endl;
             }
             pointcol(pointcol &p) { coul=p.coul;
                cout<<"constr recopie pointcol "<<int(coul)<<endl;
                point(p);
             }
} ;
int main(){
  pointcol a(2, 3, 4);
  pointcol b=a; return 0;
}
```

10.5. **CONTROLE DES ACCES**

La situation d'héritage la plus naturelle est celle dans laquelle - la classe dérivée a accès aux membres **publics** de la classe de base. Les « utilisateurs » de la classe dérivée ont accès à ses membres **publics**, ainsi qu'aux membres **publics** de sa classe de base.

C++ permet d'intervenir en partie sur ces 2 sortes d'autorisation d'accès, et ceci à 2 niveaux :

Lors de la conception de la classe de base : les membres **protégés** se comportent comme des membres **privés** pour l'utilisateur de la classe dérivée mais comme membres **publics** pour la classe dérivée elle-même.

Lors de la compilation de la classe dérivée : on peut restreindre les possibilités d'accès aux membres de la classe de base.

10.5.1. Membres protégés

La définition d'une classe peut prendre l'allure suivante :

```
class  x
{    private : ...
     protected : ...
     public : ...
 } ;
```

Statut protégé : les membres protégés restent inaccessibles à ″l'utilisateur″ de la classe, mais ils seront accessibles aux membres d'une éventuelle classe dérivée.

Exemple : Dans la définition de la classe **point .**

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

Il devient possible de définir, dans **pointcol,** une fonction membre **affiche**.

```
class pointcol : public point
{     short couleur ;
      public : void affiche()
             { cout<<‘’je suis en ‘’<<x<<‘’ ‘’<<y<<‘’\n” ;
               cout<<«   et ma couleur est”<<couleur<<‘’\n” ; }
} ;
```

10.5.2. Intérêt du statut protégé

Une dérivation **protégée** est l'intermédiaire entre la dérivation **privée** et la dérivation **publique**. Dans ce cas, les membres **publics** de la classe de base seront considérés comme protégés lors de dérivations ultérieures.

11. **HERITAGE MULTIPLE**

Une classe peut hériter de plusieurs autres classes.

11.1. **Mise en œuvre de l'heritage multiple**

Une classe **pointcoul** hérite simultanément des classes **point** et **coul** :

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

Définissons la classe **pointcoul** publique pour chacune de 2 classes :

```
class pointcoul : public point, public coul
{ //définition des membres supplémentaires (données ou fonctions)
// ou redéfinition de membres existants déjà dans point ou coul
} ;
```

Le constructeur doit pouvoir retransmettre des informations au constructeur de 2 classes de base. L'en-tête du constructeur se présentera : **pointcoul(...) : point(...), coul(...)**

- Ordre d'appel des constructeurs : ceux des classes de base et ceux de la classe dérivée.

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

Dans une fonction membre de la classe dérivée, on peut utiliser toute fonction membre publique (ou protégée) d'une classe de base.
En utilisant les fonctions **affiche** de **point** et de **coul,** la fonction **affiche** de **pointcoul** sera :

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

Si les fonctions d'affichage se nommaient **affp()** pour **point** et **affc()** pour **coul,** on écrit :

```
void affiche()
{  affp(); affc() ;
}
```

Un objet de type **pointcoul** peut faire appel aux fonctions membres de **pointcoul** ou, éventuellement, aux fonctions membre des classes de base **point** et **coul** :

```
pointcoul  p(3, 9, 2);
```

p.affiche() appellera la fonction **affiche()** de **pointcoul**,
p.point::affiche() appellera la fonction **affiche()** de **point**.

Exemple : **pointcol** hérite de **point** et de **coul** avec quelques affichages affirmatifs.

```
#include<iostream>
using namespace std;
class point {   int x, y;
                public :  point(int abs, int ord){ cout<<"++construction point\n"; x=abs; y=ord; }
                          ~point(){cout<<"--destruction point \n "; }
                           void affiche(){ cout<<"coordonnees :"<<x<<" "<<y<<"\n"; }
} ;
class coul { short couleur;
              public : coul(int cl){ cout<< "++construction coul\n"; couleur=coul; }
                       ~coul(){cout<<"--destruction coul\n"; }
                       void affiche(){ cout<<"couleur : "<<couleur<<"\n";
                        }
} ;
class pointcol : public point, public coul
{  public :  pointcol(int, int, int);
             ~pointcol(){ cout<<"--destruction pointcol\n"; }
             void affiche(){ point ::affiche(); coul ::affiche(); }
} ;
pointcol ::pointcol(int abs, int ord, int cl):point(abs, ord), coul(cl)
{ cout<<"++construction pointcol\n";
 }
int main(){ pointcol p(3, 9, 2);
  cout<<"----------\n"; p.affiche();
  cout<<"----------\n"; p.point::affiche();
  cout<<"----------\n"; p.coul::affiche();
  cout<< »----------\n"; return 0;
}
```

Exemple

```
class A                 class B
{ …                     {   ...
  public : int x;           public : int x;
  ...                       ...
};                      };
class C : public A, public B
{ ...};
```

C possédera 2 membres nommés **x**, l'un hérité de **A**, l'autre hérité de **B**. Au sein des fonctions membre de **C**, on fera la distinction à l'aide de l'opérateur (**::**), **A::x** ou de **B::x**.

11.2. CLASSES VIRTUELLES

Une classe peut hériter 2 fois d'une même classe. Exemple : une classe **D** hérite 2 fois de **A.**

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

Les membres donnée ou fonction de la classe **A** apparaissent **2 fois** dans la classe dérivée de **D**.
On fait appel à l'opérateur de résolution de portée pour lever l'ambiguïté.
Exemple : **A::B::x** et **A::C::x.**

Si l'on souhaite que de tels membres n'apparaissent qu'une seule fois dans la classe dérivée de 2ième niveau, il faut, dans les déclarations des dérivées de 1ier niveau **B** et **C,** déclarer avec l'attribut **virtual** la classe dont on veut éviter la duplication **A**. La classe A est **virtuelle**.

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

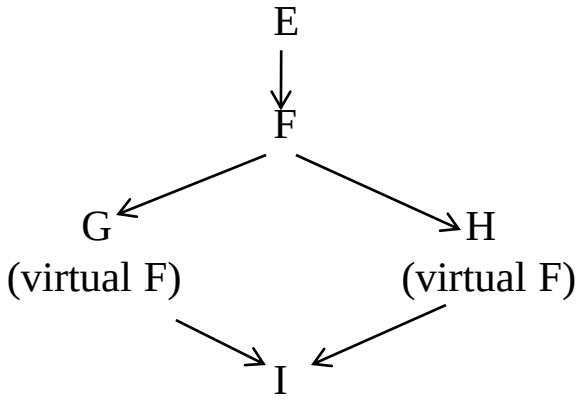
Mentionner **A** comme virtuelle dans la déclaration de **B** signifie que **A** ne devra être introduite qu'une seule fois dans les descendants éventuels de **C**.

Lorsque l'on a déclaré ainsi une classe virtuelle, il est nécessaire que les **constructeurs** d'éventuelles classes dérivées puissent préciser les informations à transmettre au constructeur de cette classe virtuelle.

D(Arguments) : B(Arguments), C(Arguments), A(Arguments)
de **D** pour **B** pour **C** pour **A.**

De plus, les constructeurs des classes **B** et **C** (qui ont déclaré que **A** était **virtuelle**) n'auront plus à spécifier d'informations pour un constructeur de **A**.

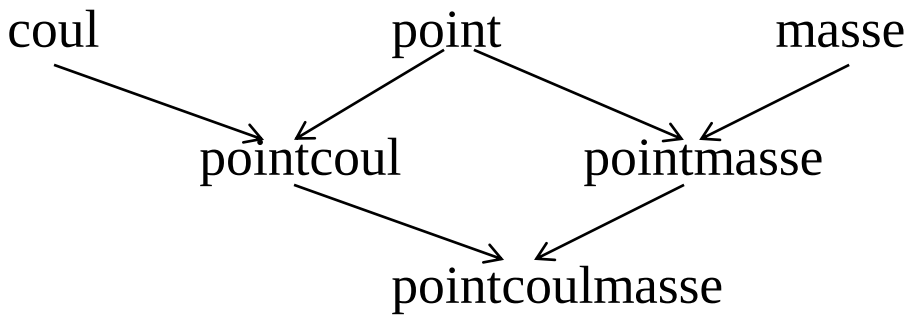
Le constructeur d'une classe virtuelle est toujours appelé avant les autres.



Cela conduit à l'ordre F, E, G, H, I

11.3. Exemple d'utilisation de l'héritage multiple et de la dérivation virtuelle

Réalisons une classe **coul** représentant une couleur et une classe **pointcol** dérivée de **point** pour représenter des points colorés. Nous définissons une classe **masse** pour représenter une masse et une classe **pointmasse** pour représenter des points dotés d'une masse. Créons une classe **pointcolmasse** pour représenter des points dotés d'une couleur et d'une masse. Nous la faisons dériver de **pointcol** et **pointmasse**.



Pour éviter la duplication des membres de **point** dans cette classe, les classes **pointcoul** et **pointmasse** doivent dériver virtuellement de la classe **point** qui doit disposer d'un **constructeur sans argument**.

```
#include<iostream>
uses namespace std;
class point
{ int x, y;
  public :
   point(int abs, int ord)
  { cout<<"++constr. point "<<abs<< "   "<< ord<< "\n";
    x=abs; y=ord;
   }
   point() //Nécessaire pour dérivation virtuelle
   {cout<<"++constr. defaut point  \n";  x=0; y=0;
    }
   void affiche()
   {cout<<"++coordonnees "<<x<< "   "<< y<< "\n";
    }
};
class masse
{ int mass;
  public : masse(int m)
            {cout<<"++constr. masse"<<m<< "\n";  mass=m;
            }
            void affiche()
            {cout<<"++Masse :  "<< mass<< "\n";  }
};
```

```
class coul
{ short couleur;
  public :
  coul(short cl)
  {cout<<"++constr. coul" <<cl<< "\n
";
    couleur=cl;
  }
  void affiche()
  {cout<<"couleur:"<< couleur<< "\n";
  }
};
```

```
class pointcoul : public virtual point, public coul
{  public :
   pointcoul(int abs, int ord, short cl) : coul(cl)   //pas d'info. pour point car dérivation virtuelle
   { cout<<"++++constr. pointcoul  "<<abs<< "   "<< ord<<"   "<<cl<< "\n";
    }

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

};

class pointmasse : public virtual point, public masse
{ public :
    pointmasse(int abs, int ord, int m) : masse(m) //pas d'info. pour masse car dérivation virt
    {cout<<"++++constr. pointmasse "<<abs<< "   "<< ord<<"   "<<m<< "\n";
    }

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

};
```

```
class pointcoulmasse : public pointcoul, public pointmasse
{  public :
   pointcoulmasse(int abs, int ord, short cl, int m) : point(abs, ord) ,
‘       pointcoul(abs, ord, cl), pointmasse(abs, ord, m)
        //Infos abs et ord inutiles pour pointcoul et pointmasse
   { cout<<"++++constr. pointcol masse "<<abs<< "   "<< ord<<"   "<<cl<< "   "<<m<< "\n";
    }

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

int main()
{  pointcoul p(3,9, 2); p.affiche();
   pointmasse pm(12, 25, 100); pm.affiche();
   pointcoulmasse pcm(2, 5, 10, 20); pcm.affiche();
   return 1;
 }
```

12. Fonction virtuelle

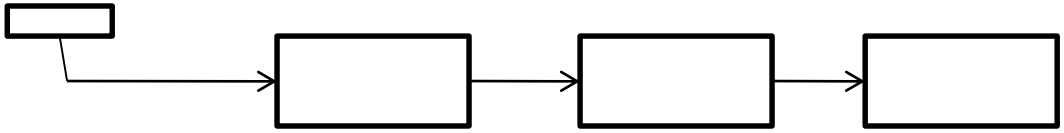
11.3. Exemple d'utilisation de l'héritage multiple

Réalisons une classe permettant de gérer une liste chaînée d'objets de type **point**. Une classe pourrait hériter de la classe **point**.

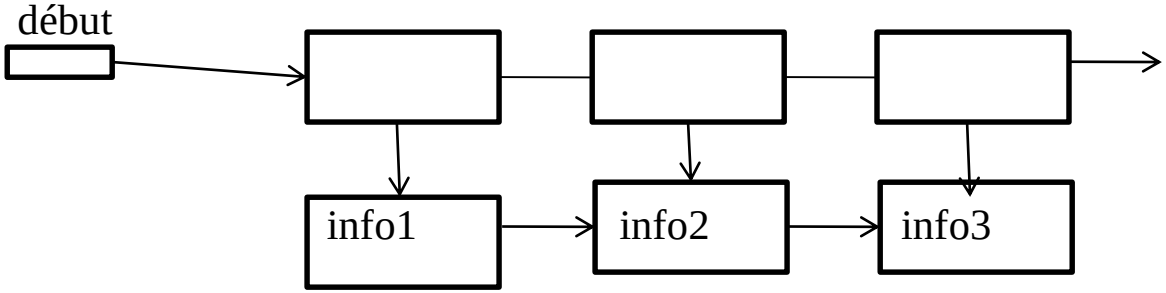
Nous allons concevoir une classe prenant en charge tout ce qui concerne la gestion de la liste chaînée, sans entrer dans les détails spécifiques aux types des objets concernés. Une telle classe n'aura aucun intérêt en soi elle sera uniquement conçue en vue d'être dérivée : une classe **abstraite**.

11.3.1. une classe abstraite : liste chaînée.

Soit une liste chaînée définie par un 1ier élément où chaque élément comporte un pointeur sur le suivant.
Schéma classique : l'information associée à chaque élément n'est pas connue.



Pour que la nature de l'information associée à chaque élément de la liste soit connue, nous allons adopter le schéma :



La classe **liste** gérera des éléments simples réduits chacun à : un pointeur sur l'élément suivant, un pointeur sur l'information associée (un objet). La classe possède au moins :
- un membre donné : pointeur sur le 1ier élément,
-une fonction membre destinée à insérer dans la liste un objet dont on lui fournira l'adresse. Cette adresse doit être de type **void***.

```
struct element
{ element * suivant ;
  void * contenu ;
} ;
```

```
class liste
{ element *debut ;
   public : liste() ;
            ~liste() ;
              void ajoute(void*) ;
...
} ;
```

D'autres possibilités : afficher les objets pointés par la liste, rechercher un élément, supprimer un élément...pourraient être réalisées par la classe **liste** elle-même, si elle connaissait la nature des objets. Mais nous allons adopter une autre démarche.

Les activités évoquées font toutes appel à un mécanisme de **parcours** de la liste. Certes ce parcours devra pouvoir être contrôlé (initialisé, poursuivi, interrompu...) depuis l'**extérieur** de la liste ; mais il est possible de prévoir des fonctions élémentaires telles que : initialiser le parcours, avancer d'un élément. Celles-ci nécessitent un pointeur sur un **élément courant**.

Par ailleurs, les 2 fonctions membre évoquées peuvent fournir en retour une information concernant l'objet courant : à ce niveau, on peut choisir entre :

- l'adresse de l'élément courant,
- l'adresse de l'objet courant,
- la valeur de l'élément courant.

La 1ière solution contient la seconde ; si **ptr** est l'adresse de l'élément courant, **ptr->contenu** fournit celle de l'objet courant. Malgré tout, on pourrait objecter que l'utilisateur de la classe n'a pas à accéder aux éléments de la liste.

La 3ième solution paraît sûre. Dans ce cas, l'utilisateur n'a alors plus de moyen de détecter la fin de la liste. Donc, on prévoira une fonction supplémentaire permettant de savoir si la fin de liste est atteinte.
A la fin, nous introduisons 3 nouvelles fonctions : void* premier() ; void* prochain() ; int fini()

```
#include<stddef>
using namespace std;
struct element{ element *suivant;
                void *contenu;
} ;
class liste { element *debut;
              element *courant;
    public : liste(){ debut=NULL ; courant=debut; }
             ~liste();
              void ajoute(void*);
              void* premier(){ courant=debut; return courant-> contenu; }
              void* prochain(){ if(courant != NULL)courant=courant->suivant;
                                return courant->contenu ; }
              int fini(){return (courant==NULL) ; }
} ;
liste ::~liste(){ element *suiv; courant=debut;
                  while(courant != NULL){ suiv = courant->suivant;
                      delete courant; courant=suiv; }
}
```

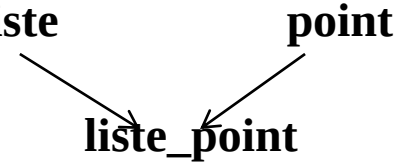
```
void liste ::ajoute(void * chose){ element *adel = new element ; adel->suivant = debut ;
   adel-> contenu = *chose ; debut = adel ;
}
```

11.3.2 Création par héritage multiple, d'une classe liste_points

Supposons une classe **point.**

```
#include<iostream.h>
class point
{ int x, y ;
  public : point(int abs=0, int ord=0)
           { x=abs ; y=ord ; }
           void affiche() { cout<<‘’coordonnées : ‘’
               << x << ‘’ ‘’ << y << ‘’\n” ; }
};
```

Créons une nouvelle classe nous permettant de gérer une liste chaînée d'objets de type **point** :une classe **liste_points** héritant simultanément de **liste** et de **point**.



Cet héritage conduit à introduire dans la classe **liste_points** *2* membres donnée **x**, **y** n'ayant pas vraiment d'intérêt : les objets concernés seront créés de manière indépendante de l'objet de type **liste_points**.

Supposez que nous souhaitons, dans la classe **liste_points**, pouvoir simplement :

- introduire un nouveau point en début de liste,
- afficher tous les objets de la liste.

La 1ière fonction est assurée par la fonction membre ajoute de la classe liste. Il n'est même pas nécessaire de la surcharger. Quant à la fonction d'affichage **affiche()**, elle va exploiter :
- les fonctions **premier**, **prochain** et **fini** pour explorer toute la liste,
- la fonction **affiche()** de la classe point pour afficher le contenu d'un point.

```
#include<iostream>
#include<stddef>
using namespace std;
struct element { element *suivant ;
                 void *contenu ;
} ;
class liste { element *debut ;
              element *courant ;
  public : liste(){ debut = NULL ; courant = debut ;}
           ~liste() ;
           void ajoute(void *);
           void * premier(){ courant = debut ; return courant-> contenu . ; }
           void *prochain(){ if(courant != NULL) courant = courant-> suivant ;
              return courant->contenu ; }
           int fini(){return (courant == NULL) ; }
} ;
liste ::~liste(){ element *suiv ; courant = debut ;
  while(courant != NULL){ suiv = courant->suivant ;
  delete courant ; courant = suiv ; }
}
```

```
void liste ::ajoute(void * chose) {element *adel = new element ; adel->suivant = debut ;
  adel-> contenu = *chose ; debut = adel ;
}
class point{.
  int x, y ;
  public :
      point(int abs=0, int ord =0){ x = abs ; y = ord ; }
      void affiche(){cout<<″coordonnées : ″<<x<<″ ″<<y<<″\n″ ; }
} ;
class liste_points : public liste, public point{
      public : liste_points() ;
      void affiche() ;
} ;
void liste_points ::affiche(){
      point *ptr = (point*)premier ;
      while( !fini()){ptr->affiche() ; ptr = (point*)prochain() ;}
}
main(){
      liste_points l ; point a(2, 3), b(5, 9), c(0, 8) ;
      l->ajoute(&a) ; l->affiche() ; cout<<″----------\n″ ;
      l->ajoute(&b) ; l->affiche() ; cout<<″----------\n″ ;
      l->ajoute(&c) ; l->affiche() ; cout<<″----------\n″ ;
}
```

12. Exceptions

12.1. Principe et syntaxe

Le modèle de gestion des **exceptions** de terminaison est un moyen de traiter les **erreurs** qui peuvent survenir dans un programme.

Exemple d'erreur d'implémentation

En créant une calculatrice, nous codons la division de 2 nombres entiers.

```
int division(int a,int b)
{ return a/b;
}
int main(){ int a, b;
cout << "Valeur pour a : "; cin >> a;
cout << "Valeur pour b : "; cin >> b;
cout << a << " / " << b << " = " << division(a,b) << endl;
return 0;
}
// Correct sauf si b=0.
```

Solutions

```
1) int division(int a,int b) .
{ if(b!=0)return a/b;
   else return ERREUR;
}
```

ERREUR n’est pas une instruction.

- Afficher un message d'erreur : **cout** << **"ERREUR : Division par 0 !"** << **endl;**
- Modifier la signature et le type de retour de la fonction.

```
bool division(int a,int b, int& resultat)
{ if(b!=0){ resultat = a/b; return true;
                }
   else return false; // On renvoie false pour montrer qu'une erreur s'est produite.
}
```

C'est la meilleure solution, mais il est impossible de réaliser le calcul a/(b/c) de manière simple et intuitive.

Solution correcte est l**a gestion des exceptions** dont le principe général est :

on crée des zones où l'ordinateur va **essayer** le code en sachant qu'une erreur peut survenir ;

Si une erreur survient, on la signale en **lançant un objet** contenant des informations sur l'erreur ;

A l'endroit où l'on souhaite gérer les erreurs survenues, on **attrape** l'objet et on gère l'erreur.

On lance un objet en espérant qu'un autre bout de code le rattrapera, sinon le programme plantera.

Les mot-clés du C++ correspondant à ces actions sont :

try{ ...} (**essaye**) signale une portion de code où une erreur peut survenir ;

throw (**lance**) signale l'erreur en lançant un objet ;

catch(...){...} (**attrape**) introduit la portion de code qui récupère l'objet et gère l'erreur.

Actions try/ throw/ catch

- **try** introduit un bloc sensible aux exceptions (indique au compilateur qu'une certaine portion du code source pourrait lancer un objet.

try { **instructions C++** (code qui pourrait créer une erreur) ou un autre bloc **try**
}

- **throw,** grâce à lui qu'on lance l'objet.

Syntaxe : **throw** expression.

```
throw 123; // On lance l'entier 123, par exemple si l'erreur 123 est survenue
throw string("Erreur fatale. Contactez un administrateur"); // On peut lancer un string.
throw personnage; // lancer une instance d'une classe.
throw 3.14 * 5.12; // Ou même le résultat d'un calcul
```

throw peut se trouver n'importe où mais, il doit être dans un bloc **try**.

- catch crée un bloc de gestion d'une exception survenue. Il faut créer un bloc **catch** par type d'objet lancé. Chaque bloc **try** doit obligatoirement être suivi d'un bloc **catch**.

Syntaxe : **catch** (type **const**& e){...}

On attrape les exceptions par référence constante (&) afin d'éviter une copie et de préserver le polymorphisme de l'objet reçu.

```
try{ // Le bloc sensible aux erreurs.
}
catch(int e) //On rattrape les entiers lancés (pour les entiers, une référence n'a pas de sens)
{ //On gère l'erreur
}
catch(string const& e) //On rattrape les strings lancés
{ // On gère l'erreur
}
catch(Personnage const& e) //On rattrape les personnages
{ //On gère l'erreur
}
```

On peut mettre autant de blocs **catch**. Il en faut **au moins un** par type d'objet pouvant être lancé.

Reprenons l'exemple de la calculatrice et de la division par 0, en donnant la bonne solution.

```
int division(int a, int b){ return a/b; }
```

Une erreur peut survenir si b vaut 0, lançons une **exception chaîne de caractères** dans ce cas.

```
throw string("ERREUR : Division par zéro !");
```

throw doit toujours se trouver dans un bloc **try** qui doit être suivi d'un bloc **catch**.

```
int division(int a,int b)
{ try { if(b == 0)throw string("Division par zéro !");
          else return a/b;
        }
  catch(string const& chaine)
  { //On gère l'exception.
  }
}
```

Gérer l'erreur

1) afficher un message d'erreur : **cerr** << chaine << endl; //Flux standard des erreurs.

Valeur pour a : 3
Valeur pour b : 0
ERREUR : Division par zéro !

Inconvénients : la fonction écrit dans la console mais ce n'est pas son rôle. Le programme continue alors qu'une erreur est survenue.

2) lancer l'exception dans la fonction et récupérer l'erreur, si elle se produit, dans le main.

```
int division(int a,int b) { if(b==0)throw string("ERREUR : Division par zéro !");
  else return a/b;
}
```

```
int main()
{ int a,b;
  cout << "Valeur pour a : "; cin >> a;
  cout << "Valeur pour b : "; cin >> b;
  try
  { cout << a << " / " << b << " = " << division(a, b) << endl;
    }
  catch(string const& chaine)
  { cerr << chaine << endl;
   }
  return 0;
}
```

12.2. Exceptions standard

On peut aussi lancer un **objet** qui contiendrait **plusieurs attributs** :

- une phrase décrivant l'erreur ;
- le numéro de l'erreur ;
- le niveau de l'erreur (erreur fatale, erreur mineure...) ;
- l'heure à laquelle l'erreur est survenue ; etc.

Pour réaliser cela on peut dériver la classe **exception** de la bibliothèque standard (SL) du C++.

class exception

```
{ public : exception() throw(){ }          //Constructeur.
           virtual exception() throw(); //Destructeur.
  virtual const char* what() const throw(); //Renvoie une chaîne contenant des infos sur l'erreur.
} ;
```

Les **méthodes** de la classe suivies de **throw,** signifie qu'**elles** ne vont pas lancer d'exceptions.
Créons notre propre classe d'exception en la dérivant grâce à un héritage :

```
#include <exception>
using namespace std;
class Erreur : public exception
{ public   : Erreur(int numero=0, string const& phrase="", int niveau=0)
              throw() : m_numero(numero), m_phrase(phrase), m_niveau(niveau)
              {}
              virtual const char* what() const throw()
             { return m_phrase.c_str() ;
             }
              int getNiveau() const throw()
             { return m_niveau;
              }
              virtual Erreur() throw()
              {}
  private: int m_numero; string m_phrase;  //Description de l'erreur
           int m_niveau;  //Niveau de l'erreur
} ;
```

On peut alors créer sa propre classe d'exception en la dérivant grâce à un héritage.

```
int division(int a,int b) // Calcule a divisé par b.
{ if(b==0)throw Erreur(1,"Division par zéro",2);
  else return a/b;
}
```

```
int main(){ int a,b;
cout << "Valeur pour a : "; cin >> a; cout << "Valeur pour b : "; cin >> b;
try{ cout << a << " / " << b << " = " << division(a,b) << endl;
}
catch(std::exception const& e){ cerr << "ERREUR : " << e.what() << endl;
}
return 0;
}
```

Valeur pour a : 3
Valeur pour b : 0
ERREUR : Division par zéro

Toutes les exceptions lancées par les fonctions standard dérivent de la classe **exception** ce qui permet, avec un code générique, de rattraper toutes les erreurs qui pourraient arriver. Code générique : **catch**(std::exception **const**& e){ cerr << "ERREUR : " << e.what() << endl; }
On attrape un objet de type **exception**, grâce aux fonctions virtuelles et à la référence, c'est la méthode **what()** de la classe fille qui sera appelée, c'est justement ce que l'on souhaite.

La bibliothèque standard peut lancer 5 types d'exceptions différents :
bad_alloc lancée s'il se produit une erreur en mémoire.
bad_cast lancée s'il se produit une erreur lors d'un **dynamic_cast**.
bad_exception lancée si aucun **catch** ne correspond à un objet lancé.
bad_typeid lancée s'il se produit une erreur lors d'un **typeid**.
ios_base::failure lancée s'il se produit une erreur avec un flux.

Exemple d'un code qui utilise **bad_alloc**.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{ try
{ vector<int> a(1000000000,1); //Un tableau bien trop grand
}
catch(exception const& e) //On rattrape les exceptions standard de tous types
{ cerr << "ERREUR : " << e.what() << endl; //On affiche la description de l'erreur
}
return 0;
}
```

Résultat affiché dans la console : **ERREUR : std::bad_alloc**

Si l'on avait attrapé l'exception par valeur et pas par référence, le message aurait été **std::exception** car le polymorphisme n'est pas conservé.

12.3. Exceptions de vector

Accéder à la 10ième case d'un vector de 8 éléments est une erreur. Ayant un index invalide, le programme va planter. Pour utiliser une exception en cas d'erreur d'index les **vector** proposent une méthode appelée **at()** qui fait la même chose que les crochets mais qui lance une exception en cas d'indice erroné.

```
#include <vector>
#include <iostream>
using namespace std;
int main(){ vector<double> tab(5, 3.14); //Un tableau de 5 nombres à virgule
  try{ tab.at(8) = 4.; //On essaye de modifier la 8ème case
  }
  catch(exception const& e){ cerr << "ERREUR : " << e.what() << endl;
  }
  return 0;
}
```

Résultat affiché dans la console : ERREUR : vector::_M_range_check

Relancer une exception

Il est possible de relancer une exception reçue par un bloc **catch** afin de la traiter une 2ième fois, plus loin dans le code. Pour ce faire, il faut utiliser le mot-clé **throw** sans expression derrière.

```
catch(exception const& e) // Rattrape toutes les exceptions
{ //On traite une première fois l'exception
   cerr << "ERREUR: " << e.what() << endl;
   throw; // Et on relance l'exception reçue pour la retraiter dans un autre bloc catch plus loin dans le code.
}
```

En résumé

Les exceptions servent à réparer des erreurs. Les exceptions sont lancées grâce au mot-clé **throw**, placé dans un bloc **try**, et rattrapées par un bloc **catch**. La bibliothèque standard propose la classe **exception** comme base pour créer ses exceptions personnalisées.