enseignant : Philippe REITZ
année universitaire : 2001-2002

Programmation

unité de Travaux Pratiques

TP n°1 : C++

[Corrigé]

Exercice 1 : les dates

Définir une classe permettant de représenter des dates ; une date s'initialise connaissant son jour, son mois et son année ; le mois doit pouvoir être donné soit sous forme numérique, soit sous forme symbolique.

Définir une fonction membre permettant de transformer une date en son lendemain (ajout de 1 au jour), et le comparateur `<` qui permet de tester qu'une date est plus ancienne qu'une autre.

Définir une fonction membre `affiche` permettant d'afficher une date sous le format `jj/mm/aaaa`.

Correction

De nombreuses solutions sont possibles à ce problème ; nous privilégions la simplicité, au détriment de la compacité : en effet, le codage d'un jour nécessite au pire 5 bits (de 1 à 31, donc $\log_2 31$ bits), un mois en nécessite 4 ($\log_2 12$) ; pour les années, en supposant que nous restreignons aux années 1900 à 2055, 8 bits sont nécessaires. Au total, 17 bits suffiraient pour coder une date, avec nos restrictions.

Nous définissons une date ainsi :

```
class Date {
  public:
    typedef unsigned short int Jour;
    enum Mois {Jan=1, Fev, Mar, Avr, Mai, Jun, Jul, Aou, Sep, Oct, Nov, Dec};
    typedef unsigned int Annee;
  private:
    Jour  le_jour;
    Mois  le_mois;
    Annee l_annee;
  public:
    Date(Jour, unsigned short int, Annee);
    Date(Jour, Mois, Annee);
  public:
    void lendemain();
    void affiche();
    bool operator<(Date);
  public:
    Jour dernier_jour_mois();
    bool bissextile();
};
```

Nous supposons que le type `bool` des booléens est défini.

Un type local public, `Mois`, permet une manipulation symbolique des mois. Notons que le codage des mois correspond à leur valeur numérique standard (par exemple, janvier est codé par l'entier 1). Deux autres types locaux, `Jour` et `Annee`, permettent de s'affranchir en partie du choix d'implantation opéré, et améliorent grandement la lisibilité du code.

Trois membres données caractérisent complètement une date, chacun ayant un type qui soit le proche des contraintes imposées. Ces membres sont tous privés, afin que notre classe soit, d'un point de vue exploitation, relativement indépendante de sa réalisation actuelle.

Deux constructeurs permettent d'initialiser une date, définis comme suit :

```
Date::Date(Date::Jour j, unsigned short int m, Date::Annee a)
  : le_jour(j), le_mois((Mois)m), l_annee(a) {};

Date::Date(Date::Jour j, Date::Mois m, Date::Annee a)
  : le_jour(j), le_mois(m), l_annee(a) {};
```

Les constructeurs mériteraient un traitement d'erreur lorsque les entiers représentant le jour ou le mois sont incorrects.

La fonction `lendemain` ne pose pas de difficulté majeure, excepté qu'il nous faut être en mesure de calculer le nombre de jours d'un mois, et ceci quelque soit l'année. Rappelons que les règles suivantes doivent être vérifiées, dans l'ordre où elles sont indiquées : une année :

1. est bissextile si elle est un multiple de 400.
2. n'est pas bissextile si elle est un multiple de 100.
3. est bissextile si elle est un multiple de 4.

Ainsi 2000 et 1992 sont bissextiles, alors que 1900 et 1999 ne le sont pas.

Nous définissons pour cela 2 fonctions membres utiles : `bissextile` retourne un booléen vrai ou faux selon que l'année de la date est bissextile ou pas, et `dernier_jour_mois` retourne le dernier jour du mois de la date.

```
bool Date::bissextile() {
  if (l_annee % 400 == 0)
      return true;
  else if (l_annee % 100 == 0)
      return false;
  else
      return (l_annee % 4 == 0); };

Date::Jour Date::dernier_jour_mois() {
  if (le_mois == Fev)
      return bissextile() ? 29 : 28;
  else if (le_mois == Avr || le_mois == Jun || le_mois == Sep || le_mois == Nov)
      return 30;
  else
      return 31; };

void Date::lendemain() {
  if (le_jour == dernier_jour_mois()) {
      le_jour = 1;
      if (le_mois == Dec) {
          le_mois = Jan;
          l_annee += 1; }
      else
          le_mois = (Mois)((int)le_mois + 1); }
  else
      le_jour += 1; };
```

L'affichage d'une date ne pose aucune difficulté :

```
void Date::affiche() {
  cout << le_jour << '/' << (int)le_mois << '/' << l_annee; };
```

La comparaison de dates est relativement simple avec notre choix de représentation :

```
bool Date::operator<(Date d) {
  if (l_annee < d.l_annee)
      return true;
  else if (l_annee > d.l_annee)
      return false;
  else if (le_mois < d.le_mois)
      return true;
  else if (le_mois > d.le_mois)
      return false;
  else
      return le_jour < d.le_jour; };
```

Le programme suivant permet de tester l'ensemble des fonctionnalités offertes par notre classe :

```
int main() {
   Date d1 = Date(28, Date::Fev, 1996),
        d2 = Date( 1,   3, 1996);

   cout << "d1 = "; d1.affiche(); cout << endl;
   cout << "d2 = "; d2.affiche(); cout << endl;

   cout << "d1 est ";
   if (d1 < d2)
      cout << "plus ancienne";
   else
      cout << "au moins plus recente";
   cout << " que d2." << endl;
```

```
d1.lendemain();
cout << "Le lendemain de d1 est le "; d1.affiche(); cout << endl; };
```

Exercice 2 : points et rectangles

Définir une classe permettant de représenter des points du plan, et offrant les services suivants :

- affichage sous forme de texte des coordonnées.
- déplacement selon l'axe des x de δ_x et selon les y de δ_y.

Définir, en s'appuyant sur la classe précédante, une classe permettant de représenter des rectangles du plan (cotés parallèles aux axes), et offrant les services suivants :

- calcul de la surface
- affichage sous forme de texte des coordonnées.
- déplacement selon l'axe des x de δ_x et selon les y de δ_y.

Correction

Un point du plan est caractérisé par ses coordonnées. Lorsqu'un point est créé, ses coordonnées doivent être connues ; sinon, elles sont considérées comme nulles, par défaut.

Si le choix de représentation des points semble immédiat, notons qu'un premier problème se pose : quel système de coordonnées allons-nous adopter : le système cartésien (x et y) ou le système polaire (ρ et θ) ?

Puisque notre classe `Point` consiste à privilégier un système plutôt qu'un autre, nous choisissons de cacher (privatiser) les membres données représentant les coordonnées. Des fonctions membres devront en permettre la lecture.

```
class Point {
    float _x;
    float _y;
  public: // constructeur
    Point(float=0.0, float=0.0);
  public: // observateurs
    float x();
    float y();
  public:
    void afficher();
    void deplacer(float=0.0, float=0.0);
};
```

Le corps du constructeur se définit simplement comme suit :

```
Point::Point(float X, float Y) {
   _x = X; _y = Y; };
```

Les membres données sont initialisés par affectation. Nous pourrions initialiser ces membres en usant de la forme spéciale d'initialisation des membres données des constructeurs :

```
Point::Point(float X, float Y) : _x(X), _y(Y) { };
```

Les définitions des autres fonctions membres ne posent aucune difficulté :

```
float Point::x() { return _x; };

float Point::y() { return _y; };

void Point::afficher() { cout << '(' << _x << ',' << _y << ')'; };

void Point::deplacer(float dx, float dy) { _x += dx; _y += dy; };
```

Nous aurions pu ajouter deux fonctions membres permettant de calculer les coordonnées polaires $\rho = \sqrt{x^2 + y^2}$ et $\theta = \arctan \frac{y}{x}$, définies par :

```
float Point::rho() { return sqrt(_x*_x+_y*_y); };

float Point::theta() {
  if (_x == 0.0)
      return _y > 0.0 ? pi/2.0 : -pi/2.0;
```

```
    else
        return atan2(_x, _y); };
```

Pour les rectangles, une première solution consiste à considérer qu'un rectangle est complètement défini par les coordonnées de son coin en bas à gauche et de son coin en haut à droite :

```
class Rectangle {
    Point bg, hd;
  public:
    Rectangle(float=0.0, float=0.0, float=0.0, float=0.0);
    ...
```

Le constructeur attend (au mieux) les quatre coordonnées de ces 2 points :

```
Rectangle::Rectangle(float X1, float Y1, float X2, float Y2)
  : bg(X1, Y1), hd(X2, Y2) {};
```

Notons l'initialisation des membres données : il s'agit de points construits (initialisés) en leur passant les 2 coordonnées correspondantes. Contrairement aux points, c'est ici la seule façon d'initialiser ces membres (pas question d'affectation).

Une autre solution consiste à considérer qu'un rectangle est complètement défini par les coordonnées de son coin inférieur gauche, sa hauteur et sa longueur. Dès lors, la définition de la classe est :

```
class Rectangle {
    Point bg;
    float h, l;
  public:
    Rectangle(float=0.0, float=0.0, float=0.0, float=0.0);
  public:
    float surface();
    void afficher();
    void deplacer(float, float); };
```

Le constructeur permet d'initialiser un rectangle en connaissant les coordonnées x et y de son coin inférieur gauche, sa hauteur et sa longueur :

```
Rectangle::Rectangle(float X, float Y, float H, float L)
  : bg(X, Y), h(H), l(L) {};
```

La fonction membre qui calcule la surface ne possède aucun paramètre ; en effet, n'oublions que cette fonction membre est nécessairement appliquée à un objet de type `Rectangle`, lequel dispose de toute l'information pour ce calcul.

```
float Rectangle::surface() { return h * l; };
```

Pour l'affichage, nous choisissons d'afficher les coordonnées du coin inférieur gauche et celles du coin supérieur droit.

```
void Rectangle::afficher() {
  cout << "[";
  bg.afficher();
  cout << ',';
  Point hd = Point(bg.x()+l, bg.y()+h);
  hd.afficher();
  cout << "]"; };
```

Notons la création d'un point temporaire `hd` représentant le coin en haut à droite, qui permet l'affichage de ses coordonnées. En effet, nous ne devons avoir aucun a priori sur la méthode d'affichage des points ; le fait d'appeler la fonction membre d'affichage des points nous permet de ne pas modifier le code de la fonction d'affichage associée aux rectangles, même si l'affichage des points devait changer par la suite.

Ainsi, si nous exécutons ce morceau de code :

```
Rectangle r(-1, 2, 5, 3);
r.afficher();
```

nous observons l'affichage du texte suivant :

```
[ (-1, 2) , (4, 5) ]
```

Déplacer un rectangle, c'est simplement déplacer son coin en bas à gauche :

```
void Rectangle::deplacer(float dx, float dy) {
   bg.deplacer(dx, dy); };
```

Exercice 3 : initialisation - destruction d'objets

Voici un programme C++ :

```
#include <iostream.h>

class Objet {
      char c;
  public:
      Objet(char);
      ~Objet();
      void Affiche(); };

Objet::Objet(char car) : c(car) {
       cout << "initialise objet "; Affiche(); cout << endl; };

Objet::~Objet() {
       cout << "détruit objet "; Affiche(); cout << endl; };

void Objet::Affiche() { cout << '[' << c << ']'; };

Objet o1('A');

void main() {
     cout << "C'est parti ...\n";
     Objet o2('B'), *o3;
     cout << "Début sous-bloc\n";
     { Objet o4('D');
       o3 = new Objet('C'); };
     cout << "Fin sous-bloc\n";
     delete o3;
     cout << "C'est fini ...\n"; };
```

Observer le comportement de ce programme ; expliquer l'ordre d'apparition des messages à l'écran.

Correction

L'ordre d'apparition des messages est le suivant :

```
initialise objet [A]
C'est parti ...
initialise objet [B]
Début sous-bloc
initialise objet [D]
initialise objet [C]
détruit objet [D]
Fin sous-bloc
détruit objet [C]
C'est fini ...
détruit objet [B]
détruit objet [A]
```

Tout d'abord, nous constatons qu'il existe une variable globale `o1`. Avant toute chose (en particulier avant exécution de `main`), cette variable doit être allouée automatiquement sur la pile puis initalisée (appel du constructeur), en tout début d'exécution. D'où un premier affichage :

```
initialise objet [A]
```

Toutes les variables globales étant allouées, l'exécution de la fonction `main` peut commencer. La première instruction est un simple affichage :

```
C'est parti ...
```

Ensuite, un premier objet `o2` est alloué automatiquement sur la pile et initialisé (appel du constructeur), d'où le message :

```
initialise objet [B]
```

Cette définition est suivie de celle de `o3`, un pointeur sur un objet. La machine alloue automatiquement sur la pile l'espace nécessaire au stockage d'un pointeur, mais n'**alloue pas** un objet.

A ce niveau, il n'y a donc aucun message affiché, puisqu'il n'y a pas eu d'allocation d'`Objet`.

L'instruction suivante affiche alors :

```
Début sous-bloc
```

Commence alors l'exécution d'un sous-bloc ; la première instruction est une définition de l'objet `o4`. Il y a donc allocation automatique sur la pile, puis initialisation par le constructeur, d'où le message :

```
initialise objet [D]
```

L'instruction suivante est une allocation dynamique (opérateur **new**), c'est à dire sur le tas. L'objet nouvellement alloué est initialisé par le constructeur, d'où le message :

```
initialise objet [C]
```

Le sous-bloc est terminé. Toute variable déclarée dans ce bloc, donc dont l'allocation fut automatiquement effectuée sur la pile, est désallouée. C'est le cas de `o4` ; il y a donc appel du destructeur pour cet objet, d'où le message :

```
détruit objet [D]
```

Notons que cette terminaison n'a aucune influence sur les objets alloués dynamiquement.

Suit une instruction d'affichage de :

```
Fin sous-bloc
```

L'instruction qui suit est une désallocation explicite de l'objet `o3`. Suite à l'appel du destructeur, le message suivant apparait :

```
détruit objet [C]
```

Suit un simple affichage de message :

```
C'est fini ...
```

Le bloc que constitue le corps de la fonction `main` est terminé. Toute variable ayant été déclarée localement est désallouée ; c'est le cas de l'objet `o2` (appel au destructeur) et du pointeur `o3`. Le message suivant apparait :

```
détruit objet [B]
```

Comme précédemment, c'est bien l'espace alloué au pointeur `o3` qui est libéré, et en aucune manière l'objet qu'il pointait (qui a par ailleurs été explicitement détruit auparavant).

Le programme est terminé ; reste alors à désallouer les variables globales ; c'est le cas de l'objet `o1`, qui provoque l'affichage de :

```
détruit objet [A]
```

Exercice 4 : les piles

Écrire deux classes permettant de représenter et manipuler des piles de taille maximum fixée à l'initialisation de chaque pile, et une pile dont la taille n'est pas limitée. Les éléments d'une pile sont supposés être des entiers.

Correction

Une pile dont la taille est figée à l'avance peut être réalisée en usant d'un tableau pour stocker ses éléments. Cette taille n'étant connue qu'au moment de la création de la pile, ce tableau n'a pas de taille prédéfinie (bien sûr, une fois fixée, cette taille ne variera plus pour une pile donnée, mais des piles distinctes peuvent ne pas avoir la même taille). En terme d'allocation, nous devrons allouer dynamiquement un tel tableau (appelons-le `tableau`). Au fur et à mesure de l'exploitation de la pile, le tableau qui la représente est plus ou moins rempli. Un membre (appelons-le `courant`) doit nous indiquer où en est l'exploitation du tableau. Nous devons connaître la taille du tableau, afin de tester si notre pile est pleine ou pas. Un membre donnée (appelons-le `taille`) est donc nécessaire pour mémoriser cette taille. D'où la déclaration suivante :

```
class PileBornee {
    unsigned int courant, taille;
    int *tableau;
  public:
    PileBornee(unsigned int);
    ~PileBornee();
  public:
    void empiler(int);
    void depiler();
    int sommet();
```

```
  bool estVide();
  bool estPleine(); };
```

Nous faisons l'hypothèse que le type `bool` des booléens existe.

Définissons le constructeur : le paramètre passé est la taille maximale autorisée pour la pile créée. Nous allouons donc dynamiquement un tableau de cette taille, et initialisons les autres membres comme il se doit :

```
PileBornee::PileBornee(unsigned int t)
  : taille(t), courant(0), tableau(new int[t]) {};
```

Nous faisons en sorte que `courant` pointe sur le prochain emplacement libre du tableau pour tout empilement ultérieur.

Le destructeur se contente de désallouer le tableau :

```
PileBornee::~PileBornee() { delete [] tableau; };
```

Notons la syntaxe particulière de l'opérateur **delete** : l'indication `[]` est impérative si l'on souhaite que tout le tableau soit désalloué. En effet, un simple **delete** `tableau` ne désallouerait que `tableau[0]` !

Pour empiler un élément, il faut s'assurer que la pile n'est pas pleine (n'oublions pas, la taille maximale est bornée) :

```
void PileBornee::empiler(int e) {
  if (estPleine())
    erreur("Pile pleine"); // nous supposons qu'une telle fonction existe
  else
    tableau[courant++] = e; };
```

Pour dépiler la pile, assurons-nous qu'elle n'est pas vide :

```
void PileBornee::depiler() {
  if (estVide())
    erreur("Pile vide");
  else
    --courant; };
```

La consultation du sommet suit presque le même schéma :

```
int PileBornee::sommet() {
  if (estVide()) {
    erreur("Pile vide");
    return 0; } // il faut impérativement retourner un résultat
  else
    return tableau[courant-1]; }; // attention, ne pas toucher à courant
```

Les deux fonctions qui testent la vacuité ou la saturation d'une pile :

```
bool PileBornee::estVide() { return courant==0; };

bool PileBornee::estPleine() { return courant==taille; };
```

Pour représenter une pile dont la taille n'est pas bornée, nous pourrions reprendre tout le code défini ci-dessus. Lorsqu'une telle pile serait pleine, il suffirait d'allouer un tableau de taille supérieure d'une unité, de recopier tous les éléments de l'ancien tableau dans le nouveau, puis de remplacer l'ancien par le nouveau (en n'oubliant pas de désallouer l'ancien).

Nous préférons procéder autrement : considérons que la pile est représentée comme une liste d'éléments tous chainés les uns aux autres :

```
class PileNonBornee {
  private: // un type local à la classe : les chainons
    class Chainon { public:
      int element;
      Chainon *suivant;
      Chainon(int e, Chainon *s) : element(e), suivant(s) {}; };
  private:
    Chainon* premier;
  public:
    PileNonBornee();
    ~PileNonBornee();
  public:
    void empiler(int);
    void depiler();
    int sommet();
    bool estVide(); };
```

Une classe `Chainon`, locale à la classe `PileNonBornee`, est définie. Sa définition étant privée, seules les fonctions membres de la classe `PileNonBornee` y ont accès.

Donnons-nous une constante souvent exploitée dans la suite :

```
#define aucunChainon 0
```

Définissons le constructeur : il se contente de positionner notre pointeur des chainons sur le pointeur nul :

```
PileNonBornee::PileNonBornee() : premier(aucunChainon) {};
```

Le destructeur doit balayer tous les chainons, et les désallouer un à un. Nous pouvons exploiter les services offerts par une pile : dépilons tant que la pile n'est pas vide (nous supposons donc que la fonction qui dépile un élément se charge de désallouer le chainon correspondant) :

```
PileNonBornee::~PileNonBornee() { while (!estVide()) depiler(); };
```

Pour empiler un élément, il faut allouer un nouveau chainon :

```
void PileNonBornee::empiler(int e) { premier = new Chainon(e, premier); };
```

Ce nouveau chainon porte le dernier élément empilé, soit le premier qui devra être considéré ; il pointe aussi sur l'éventuel chainon auquel il s'est substitué.

Pour dépiler la pile, assurons-nous qu'elle n'est pas vide. Si tel n'est pas le cas, le chainon perdu doit être désalloué :

```
void PileNonBornee::depiler() {
  if (estVide())
    erreur("Pile vide");
  else {
    Chainon *p = premier;
    premier = premier->suivant;
    delete p;} };
```

La consultation du sommet suit presque le même schéma :

```
int PileNonBornee::sommet() {
  if (estVide()) {
    erreur("Pile vide");
    return 0; } // il faut impérativement retourner un résultat
  else
    return premier->element; };
```

La fonction qui teste la vacuité d'une pile :

```
bool PileNonBornee::estVide() { return premier==aucunChainon; };
```

Exercice 5 : les polynômes

Écrire les classes permettant de représenter un polynôme. Rappelons qu'un polynôme $P(x)$ s'écrit $\sum_{i=0}^{n} m_i(x)$, où $m_i(x) = a_i.x^i$ est un monôme de degré i, de coefficient a_i. Le degré du polynôme est le plus fort degré de ses monômes non nuls. L'ensemble des monômes non nuls d'un polynôme pourra être représenté par une liste ordonnée par leurs degrés. Un polynôme a un nom et possède une variable caractérisée par son nom.

Définir alors les fonctions suivantes :

- ajouter un monôme à un polynôme existant
- lire et écrire un polynôme à partir d'un terminal
- ajouter deux polynômes
- évaluer un polynôme $P(x)$ pour une valeur de x donnée
- dériver un polynôme

Tester ces fonctions avec le polynôme suivant : $3 - 5x^2 + x^5$.

Correction

Définissons quelques types utiles par la suite :

```
typedef float          reel;  // type des réels
const reel             zero = 0;
typedef float          non_nul; // type des réels non nuls
typedef unsigned int   naturel; // type des entiers naturels
```

Nous faisons l'hypothèse qu'il existe un type `texte` permettant de manipuler des chaînes de caractères, avec en particulier la possibilité de concaténer deux textes via l'opérateur `+`.

Nous supposons également que le type `bool` des booléens existe.

L'entité principale manipulée dans cet exercice est le polynôme. Il est représenté par une liste de monômes non nuls, ordonnée par valeurs croissantes de leurs degrés. Nous choisissons de définir localement les monômes à la classe des polynômes ; ainsi, l'usager ne pourra manipuler que des polynômes. De plus, afin de mieux distinguer un monôme d'un élément d'une liste, nous définissons une autre classe locale représentant les différents chainons d'une liste :

```
class Polynome {

    class Monome {
      public:
        reel     Coefficient;
        naturel  Degre;
      public:
        Monome(non_nul coef, naturel deg=0) : Coefficient(coef), Degre(deg) {};
      public:
        reel     ajout(reel);
        reel     evalue(reel);
        Monome*  derive();
        void     affiche(char, bool=false); };

    class Chainon {
      public:
        Monome*  monome;
        Chainon* suivant;
      public:
        Chainon(reel coef, naturel deg, Chainon *suiv)
               : monome(new Monome(coef, deg)), suivant(suiv) {};
        Chainon(Monome *m, Chainon *suiv)
               : monome(m), suivant(suiv) {};
        ~Chainon() { delete monome; }; };

  private: // les membres données
    texte    nom;
    char     variable;
    Chainon *monomes;

  public: // le seul constructeur
    Polynome(texte="", char='x');

  public: // les fonctions d'intérêt général
    void     affiche();
    void     lire();
    void     ajout(reel, naturel);
    Polynome operator+(Polynome);
    reel     evalue(reel);
    Polynome derive();

  private: // quelques fonctions membres privées
    void ajout(Monome *);
    void insere(Monome *);
};
```

Donnons-nous une constante indiquant qu'aucun monome n'est pointé :

```
#define aucunMonome 0
```

C'est la seule définition possible, compte tenu de nos connaissances actuels de `C++` : le type `Chainon` étant privé, pas question de définir une constante de ce type hors de la classe. Il serait peu judicieux d'en faire un membre donnée classique, puisque dès lors tout objet `Chainon` porterait ce membre (4 octets perdus). La bonne manière de procéder consiste à définir `aucunMonome` comme un membre donnée statique du genre constante... ce que nous étudierons plus tard.

Commençons par le constructeur des polynômes : pour créer un polynôme, son nom et celui de sa variable doivent être précisés :

```
Polynome::Polynome(texte n, char v)
  : nom(n), variable(v), monomes(aucunMonome) {};
```

Pour afficher un polynôme, nous commençons par son nom et sa variable, suivi de l'affichage de chacun des monômes qui le composent :

```
void Polynome::affiche() {
     cout << nom << '(' << variable << ") = ";
     if (monomes == aucunMonome)
        cout << "0";
     else { Chainon* c = monomes;
            do { c->monome->affiche(variable, c == monomes);
                 c = c->suivant; }
            while (c != aucunMonome); } };
```

Pour un monôme, la méthode d'affichage est :

```
void Polynome::Monome::affiche(char var, bool premier) {
     if (Coefficient == 1) {
        if (!premier) cout << '+';
        if (Degre == 0) cout << Coefficient; }
     else if (Coefficient == -1)
        if (Degre == 0)
           cout << Coefficient;
        else
           cout << '-';
     else {
        if (!premier && Coefficient >= 0) cout << '+';
        cout << Coefficient; };
     if (Degre > 0)        {
        cout << var;
        if (Degre > 1) cout << '^' << Degre; } };
```

Ce code un peu compliqué résume tous les cas d'affichage suivants (à considérer dans cet ordre) :

Coefficient	**Degré**	1er **affiché**	**affichage**
1	0	oui	1
1	1	oui	x
1	1	non	+x
1	$i \geq 2$	oui	x^i
1	$i \geq 2$	non	+x^i
-1	0		-1
-1	1		-x
-1	$i \geq 2$		-x^i
$a_i > 0$	0	oui	a_i
$a_i > 0$	0	non	+a_i
$a_i > 0$	1	oui	a_ix
$a_i > 0$	1	non	+a_ix
$a_i > 0$	$i \geq 2$	oui	a_ix^i
$a_i > 0$	$i \geq 2$	non	+a_ix^i
$a_i < 0$	0		a_i
$a_i < 0$	1		a_ix
$a_i < 0$	$i \geq 2$		a_ix^i

La méthode de lecture d'un polynôme peut s'écrire ainsi :

```
void Polynome::lire() {
     reel coef;
     do {
        cout << "Valeur d'un coefficient de monome (0 = fin) = ";
        cin >> coef;
        if (coef!=0.0) {
           cout << "                                et son degre = ";
           naturel deg; cin >> deg;
```

```
            ajout(coef, deg); } }
      while (coef!=0.0); };
```

Cette méthode fait appel à une méthode capable d'ajouter un monôme (seuls son degré et son coefficient sont précisés) aux monômes déjà existants du polynôme. Voici le code de cette méthode :

```
void Polynome::ajout(reel coef, naturel deg) {
      Chainon *pr = aucunMonome, *cr = monomes;

      while (cr != aucunMonome && cr->monome->Degre < deg) {
            pr = cr;
            cr = cr->suivant; };

      if (cr != aucunMonome && cr->monome->Degre == deg) {
         if (cr->monome->ajout(coef) == 0) {
            (pr == aucunMonome ? monomes : pr->suivant) = cr->suivant;
            delete cr; } }
      else
         (pr == aucunMonome ? monomes : pr->suivant) = new Chainon(coef, deg, cr); };
```

Plusieurs situations peuvent être rencontrées :

- soit aucun monôme existant ne possède le même degré que celui qui est ajouté ; dans ce cas, le problème essentiel est d'insérer le monôme au bon endroit dans la liste des monômes, puisque celle-ci est ordonnée par degrés croissants.
  C'est l'objet de la première boucle que de trouver cet éventuel point d'insertion.
- soit il existe un monôme existant de même degré, auquel cas il nous faut vérifier qu'après somme des coefficients, ce monôme est non nul. S'il est nul, il doit être éliminé de la liste des monômes, sinon il reste en place. Notons que, pour procéder à la somme de ce nouveau monôme, nous faisons appel à une méthode `ajout` de la classe `Monome` qui retourne le nouveau coefficient du monôme. Le code de cette méthode est simplement :

  ```
  reel Polynome::Monome::ajout(reel v) { return Coefficient += v; };
  ```

  Pour finir sur cette méthode d'ajout d'un monôme, notons aussi qu'une facilité d'écriture de C++ a été employée : le schéma suivant

  ```
  if (condition)
     var_A = expression;
  else
     var_B = expression; // il s'agit de la même expression
  ```

  peut simplement s'écrire :

  ```
  (condition ? var_A : var_B) = expression;
  ```

La méthode permettant d'évaluer un polynôme pour une valeur de sa variable est triviale : il suffit d'évaluer chacun des monômes qui le composent, puis de sommer les résultats :

```
reel Polynome::evalue(reel x) {
      reel accu = zero;
      Chainon* c = monomes;
      while (c != aucunMonome) {
            accu += c->monome->evalue(x);
            c = c->suivant; };
      return accu; };
```

Cette méthode fait appel à une méthode d'évaluation sur la classe des monômes :

```
reel Polynome::Monome::evalue(reel x) {
      return Coefficient * pow(x, Degre); };
```

La fonction `pow` est, a priori, une fonction prédéfinie telle que `pow(a, b)` $= a^b$.

Le calcul de la dérivée d'un polynôme suit à peu près le même schéma : après création d'un nouveau polynôme, tous les monômes de la dérivée sont obtenus par dérivation de chacun des monômes du polynôme de départ. Sachant que la dérivée d'une constante (monôme de degré nul) est nulle, et qu'un objet monôme n'est créé que si son coefficient est non nul, nous obtenons le code suivant :

```
#define monomeNul 0

Polynome::Monome* Polynome::Monome::derive() {
       if (Degre == 0)
```

```
            return monomeNul;
         else
            return new Monome(Coefficient*Degre, Degre-1); };
```

D'où la méthode de dérivation d'un polynôme :

```
Polynome Polynome::derive() {
         Polynome d(nom+"'", variable);

         Chainon *c = monomes;
         while (c != aucunMonome) {
               d.insere(c->monome->derive());
               c = c->suivant; };

         return d; };
```

Cette méthode fait appel à une méthode privée `insere`, capable d'insérer un nouveau monôme non nul dans une liste de monômes dont on sait qu'elle ne contient pas ce monôme :

```
void Polynome::insere(Monome *m) {
  if (m != monomeNul) {
     // ATTENTION : présuppose que le monôme est non nul, et n'est
     //             pas déjà présent dans la liste des monômes
     Chainon *pr = aucunMonome, *cr = monomes;
     while (cr != aucunMonome && cr->monome->Degre < m->Degre) {
           pr = cr;
           cr = cr->suivant; };
     (pr == aucunMonome ? monomes : pr->suivant) = new Chainon(m, cr); } };
```

Pour finir, définissons la méthode ajoutant deux polynômes :

```
Polynome Polynome::operator+(Polynome p) {
         Polynome s(nom+"+"+p.nom, variable);

         Chainon *cm = monomes, *cp = p.monomes;
         while (cm != aucunMonome && cp != aucunMonome) {
               if (cm->monome->Degre <= cp->monome->Degre) {
                  s.ajout(cm->monome);
                  cm = cm->suivant; };
               if (cm->monome->Degre >= cp->monome->Degre) {
                  s.ajout(cp->monome);
                  cp = cp->suivant; } };

         if (cm == aucunMonome) cm = cp;
         while (cm != aucunMonome) {
               s.ajout(cm->monome);
               cm = cm->suivant; };

         return s; };
```

Cette méthode exploite une méthode privée des polynômes, capable d'ajouter un (vrai) monôme à un polynôme :

```
void Polynome::ajout(Monome *m) {
     ajout(m->Coefficient, m->Degre); };
```

Enfin, voici un code permettant de tester notre classe :

```
void main() {
  Polynome P("P"); // le polynome de l'enonce
  P.ajout(3.0, 0);
  P.ajout(-5.0, 2);
  P.ajout(1.0, 5);
  P.affiche(); cout << endl;

  Polynome D;
  D = P.derive();
  D.affiche(); cout << endl;
```

```
Polynome S;
S = P + D;
S.affiche(); cout << endl; };
```