enseignant : Philippe REITZ
année universitaire : 2001-2002

Programmation

unité de Travaux Pratiques

TP n°2 : C++

[Corrigé]

Exercice 1 : les piles génériques

Écrire deux classes permettant de représenter et manipuler des piles de taille maximum fixée à l'initialisation de chaque pile, et des piles dont la taille n'est pas limitée. Les éléments d'une pile doivent pouvoir être d'un type T quelconque.

Correction

Cet exercice ayant déjà été traité pour des piles dont les éléments sont des entiers, il suffit donc de reprendre ce code et d'en paramétrer le type des éléments.

Pour les piles bornées, nous définissons la classe suivante :

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

```
template <class T>
PileBornee<T>::PileBornee(unsigned int t)
  : taille(t), courant(0), tableau(new T[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 :

```
template <class T>
PileBornee<T>::~PileBornee() { delete [] tableau; };
```

Pour empiler un élément :

```
template <class T>
void PileBornee<T>::empiler(T e) {
  if (estPleine())
    erreur("Pile pleine"); // nous supposons qu'une telle fonction existe
  else
    tableau[courant++] = e; };
```

Pour dépiler la pile :

```
template <class T>
void PileBornee<T>::depiler() {
  if (estVide())
    erreur("Pile vide");
  else
    --courant; };
```

La consultation du sommet suit presque le même schéma :

```
template <class T>
T PileBornee<T>::sommet() {
  if (estVide())
    erreur("Pile vide");
  else
    return tableau[courant-1]; }; // attention, ne pas toucher à courant
```

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

```
template <class T>
bool PileBornee<T>::estVide() { return courant==0; };

template <class T>
bool PileBornee<T>::estPleine() { return courant==taille; };
```

Le petit programme suivant teste une partie des fonctionnalités offertes par cette classe :

```
void testPileBornee() {
  PileBornee<char> p(10);
  char i = 'A';
  while (! p.estPleine()) p.empiler(i++);
  while (! p.estVide() ) {
    cout << p.sommet() << endl;
    p.depiler(); } };
```

Pour représenter une pile dont la taille n'est pas bornée, nous adoptons le choix d'une liste d'éléments tous chainés les uns aux autres :

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

```
template <class T>
PileNonBornee<T>::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) :

```
template <class T>
PileNonBornee<T>::~PileNonBornee() { while (!estVide()) depiler(); };
```

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

```
template <class T>
void PileNonBornee<T>::empiler(T e) { premier = new Chainon(e, premier); };
```

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 :

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

La consultation du sommet suit presque le même schéma :

```
template <class T>
T PileNonBornee<T>::sommet() {
  if (estVide()) {
    erreur("Pile vide"); }
  else
    return premier->element; };
```

La fonction qui teste la vacuité d'une pile :

```
template <class T>
bool PileNonBornee<T>::estVide() { return premier==aucunChainon; };
```

Pour finir, un petit programme de test :

```
void testPileNonBornee() {
  PileNonBornee<char> p;
  for (int i = 1 ; i <= 4 ; i++ )
    p.empiler((char)(64+i));
  while (! p.estVide() ) {
    cout << p.sommet() << endl;
    p.depiler(); } };
```

Exercice 2 : les vecteurs

Écrire une classe `Vecteur` permettant de représenter et manipuler des vecteurs à composantes réelles de taille maximum fixée à l'initialisation de chaque vecteur, et offrant les services suivants (nous supposons que v est un vecteur quelconque de taille n) :

- il doit être possible d'initialiser toutes les composantes d'un vecteur avec une même valeur x lors de sa définition.
- si a un réel, alors $v + a$ est un vecteur w de taille n tel que chacune de ses composantes $w_i = v_i + a$.
- si u est un vecteur de même taille n, alors $v + u$ est un vecteur w de taille n tel que $w_i = v_i + u_i$. De même, $v == u$ est vrai si leurs composantes sont égales 2 à 2 ($v_i == u_i$).
- si i est l'indice d'une composante possible pour v, alors `v[i]` désigne la composante v_i ; cette composante doit être accessible aussi bien en lecture (au sein d'une expression) qu'en écriture (partie gauche d'un opérateur d'affectation).
- si c est une valeur de type `ostream` (canal de sortie), alors il doit être possible de sortir v sur c en écrivant `c << v`.
- si v est une variable de type `Vecteur` et c une valeur de type `istream` (canal d'entrée), alors il doit être possible d'entrer toutes les composantes de v à partir de c en écrivant `c >> v`.

Tester alors le petit programme suivant :

```
void testAffectation () {
  Vecteur v1(10), v2(10);
  cout << "saisie des 10 composantes de v1 : "; cin >> v1;
  v2 = v1;
  cout << "v1 = " << v1 << ", v2 = " << v2 << endl;
  v1[2] = -1.0;
  cout << "Apres modification de v1[2], nous avons :" << endl;
  cout << "v1 = " << v1 << ", v2 = " << v2 << endl; };
```

Que s'est-il passé? Expliquer le problème, puis faire le nécessaire afin de remédier au problème ; tester alors :

```
void fonc(Vecteur v) { v[3] = -1.0; cout << "v = " << v << endl; };

void testPassageParValeur () {
  Vecteur v1(10);
  cout << "saisie des 10 composantes de v1 : "; cin >> v1;
  cout << "v1 = " << v1 << ", ";
  fonc(v1);
  cout << "Apres appel de fonc(v1), nous avons v1 = " << v1 << endl; };
```

Que s'est-il passé? Expliquer le problème, puis faire le nécessaire afin de remédier au problème.

Correction

Nous commençons par écrire une version naïve, laquelle posera de nombreux problèmes, comme le suggère l'énoncé. Un vecteur est implanté comme un tableau dont la taille n'est spécifiée qu'à son initialisation :

```
class Vecteur {
    int taille;
    float *composantes;
  public:
    Vecteur(int, float=0.0);
    ~Vecteur();
  public:
    bool operator==(Vecteur);
    Vecteur operator+(Vecteur);
    Vecteur operator+(float);
   float& operator[](int);
  public:
    ostream& afficher(ostream&);
    istream& saisir(istream&);
};
```

Définissons le constructeur : le premier paramètre passé est le nombre de composantes du vecteur, le second la valeur réelle écrite par défaut dans chacune des composantes. Nous allouons donc dynamiquement un tableau de cette taille, et initialisons les autres membres comme il se doit :

```
Vecteur::Vecteur(int t, float v)
  : taille(t), composantes(new float[t])
{ for (int i=0; i<t; i++) composantes[i]=v; };
```

Le destructeur se contente de désallouer le tableau :

```
Vecteur::~Vecteur() { delete [] composantes; };
```

Pour ajouter un réel à toutes les composantes d'un vecteur :

```
Vecteur Vecteur::operator+(float a) {
  Vecteur r(taille);
  for (int i=0; i<taille; i++) r.composantes[i] = composantes[i]+a;
  return r; };
```

Noter que le vecteur figurant en partie gauche de l'opérateur `+` ne doit pas être modifié : le résultat est un nouveau vecteur `r`.

Pour ajouter deux vecteurs, nous avons simplement (nous faisons l'hypothèse que les vecteurs ont la même taille) :

```
Vecteur Vecteur::operator+(Vecteur v) {
  Vecteur r(taille);
  for (int i=0; i<taille; i++) r.composantes[i] = composantes[i]+v.composantes[i];
  return r; };
```

Le test d'égalité n'est pas difficile à écrire :

```
bool Vecteur::operator==(Vecteur v) {
  if (taille != v.taille) return false;
  int i=0;
  while (i<taille && composantes[i] == v.composantes[i]) i++;
  return i==taille; };
```

L'affichage et la saisie des composantes passent par deux fonctions membres :

```
ostream& Vecteur::afficher(ostream& c) {
  for (int i=0; i<taille; i++) c << composantes[i] << ' ';
  return c; };

ostream& operator<<(ostream& c, Vecteur v) { return v.afficher(c); };
```

Noter que l'opérateur de sortie standard `<<` se contente de déléguer sa tâche à la fonction membre `affiche`.

```
istream& Vecteur::saisir(istream& c) {
  for (int i=0; i<taille; i++) c >> composantes[i];
  return c; };

istream& operator>>(istream& c, Vecteur& v) { return v.saisir(c); };
```

Idem pour l'opérateur >> (avec un passage par référence du vecteur à saisir, afin d'en modifier ses composantes).

Reste l'opérateur d'accès aux composantes :

```
float& Vecteur::operator[](int i) { return composantes[i]; };
```

Cet opérateur pourrait vérifier que l'indice est bien dans les bornes permises (de 0 à `taille`-1 inclus), et nécessiterait un traitement d'erreur propre, ce que nous ne savons pas encore réaliser.

Testons le premier programme porposé dans l'énoncé ; son exécution produit le texte suivant :

```
saisie des 10 composantes de v1 : 1 2 3 4 5 6 7 8 9 10
v1 = 1 2 3 4 5 6 7 8 9 10, v2 = 1 2 3 4 5 6 7 8 9 10
Apres modification de v1[2], nous avons :
v1 = 1 2 -1 4 5 6 7 8 9 10, v2 = 1 2 -1 4 5 6 7 8 9 10
```

L'affectation `v1[2]` a donc modifié `v2[2]` ; en effet, lors de l'affectation `v2 = v1`, le compilateur produit un code dont l'effet est de copier membre à membre tous les membres de `v1` dans ceux de `v2`. Autrement dit, écrire

```
v2 = v1;
```

c'est écrire (modulo les problèmes de protection de membres) :

```
v2.taille = v1.taille;
v2.composantes = v1.composantes;
```

Notons deux conséquences importantes :

- le pointeur qu'est `v1.composantes` est recopié dans `v2.composantes` → les 2 vecteurs partagent alors le même espace mémoire → si `v1` modifie cet espace, `v2` subit ces modifications.
- le pointeur `v2.composantes` étant modifié, il n'existe plus aucun moyen d'accéder à l'espace mémoire préalablement pointé → cet espace est perdu à jamais pour le programme : il ne pourra plus être réexploité par la suite → il n'y a pas eu d'appel explicite à l'opérateur **delete**.

Il nous faut donc remédier à ce problème ; seule solution : redéfinir l'opérateur d'affectation, afin d'en contrôler les effets.

```
Vecteur& Vecteur::operator=(const Vecteur& v) {
  if (&v != this && taille == v.taille)
     for (int i=0; i<taille; i++) composantes[i] = v.composantes[i];
  return *this; };
```

Nous avons pris le parti de n'autoriser l'affectation que de vecteurs de même taille. Noter les particularités de l'opérateur d'affectation :

- il doit retourner une référence sur l'objet affecté, d'où un résultat du type `T&`.
- il doit explicitement tester s'il ne s'agit pas d'une auto-affectation (cas de figure pouvant survenir, et dont le traitement peut être spécial ; ce n'est pas le cas de notre classe).
- l'objet retourné doit être celui qui a subit l'affectation, d'où le **return *this** final.

Une fois ce nouvel opérateur ajouté à notre définition initiale de la classe `Vecteur`, l'exécution du programme de test se passe correctement :

```
saisie des 10 composantes de v1 : 1 2 3 4 5 6 7 8 9 10
v1 = 1 2 3 4 5 6 7 8 9 10, v2 = 1 2 3 4 5 6 7 8 9 10
Apres modification de v1[2], nous avons :
v1 = 1 2 -1 4 5 6 7 8 9 10, v2 = 1 2 3 4 5 6 7 8 9 10
```

Passons au second programme : avec notre nouvelle classe vecteur munie de son opérateur d'affectation, nous constatons malgré tout un comportement erroné ; en effet, la trace d'exécution est la suivante :

```
saisie des 10 composantes de v1 : 1 2 3 4 5 6 7 8 9 10
v1 = 1 2 3 4 5 6 7 8 9 10
Apres appel de fonc(v1), nous avons v1 = 1 2 3 -1 5 6 7 8 9 10
```

Notons que le passage du paramètre `v` de la fonction `fonc` est effectué par valeur : le vecteur `v` doit donc être, en principe, une copie du vecteur passé lors de l'appel, soit `v1`. Si nous devions récrire ce code d'appel, nous devrions remplacer l'appel :

```
void testPassageParValeur () {
  ...
  fonc(v1);
  ... };
```

par le code équivalent (enfin, c'est ce que nous croyons à cet instant...) suivant :

```
void testPassageParValeur () {
  ...
  { Vecteur v = Vecteur(v1.taille); // en principe, membre privé...
     v = v1;
     v[3] = -1.0; };
  ... };
```

Si l'expansion du code de l'appel à `fonc` était véritablement celui indiqué ci-dessus, nous n'aurions pas de problème, puisque nous avons redéfini explicitement l'opérateur d'affectation pour corriger le problème précédent. C'est donc que le passage d'un paramètre par valeur n'exploite pas l'opérateur d'affectation lors d'un appel.

En fait, le véritable code équivalent est le suivant :

```
void testPassageParValeur () {
  ...
  { Vecteur v = Vecteur(v1);
     v[3] = -1.0; };
  ... };
```

où le constructeur invoqué a pour signature :

```
Vecteur::Vecteur(const Vecteur&);
```

Ce constructeur particulier porte un nom : le *constructeur de recopie* ; il est implicitement défini par défaut par le compilateur si le programmeur ne l'a pas défini lui-même. À chaque fois qu'un paramètre est passé par valeur, c'est ce constructeur de recopie qui est exploité pour initialiser le paramètre à chaque appel.

Il nous reste donc à définir correctement notre constructeur de recopie, et notre second problème sera résolu. Son principe est identique à l'opérateur d'affectation, excepté que le test d'auto-initialisation n'a pas de sens ici : puisque l'objet est initialisé, c'est donc qu'il n'existait pas avant, et donc que l'objet à recopier est nécessairement un objet autre que celui qui est initialisé. Le code du constructeur est le suivant :

```
Vecteur::Vecteur(const Vecteur& v)
  : taille(v.taille), composantes(new float[v.taille])
{ for( int i = 0; i<taille; i++) composantes[i] = v.composantes[i]; };
```

L'exécution du second programme de test se déroule alors normalement :

```
saisie des 10 composantes de v1 : 1 2 3 4 5 6 7 8 9 10
v1 = 1 2 3 4 5 6 7 8 9 10
Apres appel de fonc(v1), nous avons v1 = 1 2 3 4 5 6 7 8 9 10
```

Exercice 3 : trois en un

Écrire une classe `BIF` permettant de représenter au choix (exclusif) soit une valeur booléenne (type `bool`), soit une valeur entière (type **int**), soit une valeur réelle (type **float**). Deux fonctions seront définies : soient a et b deux valeurs de type `BIF` de même nature (par exemple toutes deux sont des **int**), alors :

- $a == b$ permet de tester leur égalité
- $a + b$ permet d'en faire la somme, c'est à dire retourne une valeur de la classe `BIF` de nature :
  - booléenne et la disjonction logique de a et b, où a et b sont toutes les deux de nature booléeenne.
  - entière et la somme entière de a et b, où a et b sont toutes les deux de nature entière.
  - flottante et la somme flottante de a et b, où a et b sont toutes les deux de nature flottante.

  Si a et b ne sont pas de même nature, alors les conversions suivantes seront privilégiées : `bool` → **int** → **float**.

Correction

La seule solution à notre problème, étant donnée notre connaissance actuelle de `C++`, consiste à user d'une construction de type du genre **union**, seule capable de traduire le concept de somme de type décrite dans l'énoncé. Une autre solution, beaucoup plus propre, et dans l'esprit de la programmation objet, consiste à exploiter l'héritage entre classes.

La construction d'**union** possède un gros défaut en `C++` : il n'y a aucun moyen direct pour connaître le membre donné effectivement exploité dans l'**union**. Puisqu'il nous est nécessaire de disposer de cette information, nous devrons la rajouter explicitement, et encapsuler le tout en une classe ; d'où le code suivant :

```
class BIF {
  private: // un type local
    enum Genre { unBool, unInt, unFloat };
```

```
private: // les attributs, ou membres donnees
  Genre g;
  union {
    bool  b;
    int   i;
    float f; };
public: // les constructeurs
  BIF(bool);
  BIF(int);
  BIF(float);
public: // les methodes, ou fonctions membres, publiques
  bool operator==(BIF);
  BIF operator+(BIF); };
```

Une valeur de classe `BIF` possède toujours deux membres données :

– le membre `g`

– au choix l'un des membres `b`, `i` ou `f`

Chacun des trois constructeurs permet d'initialiser une valeur de type `BIF` à partir d'une valeur de type `bool`, **int** ou **float**. Leur code n'est pas compliqué : il consiste simplement à écrire la valeur d'initialisation dans le bon champ, et mettre à jour le membre `g` afin de savoir lequel des 3 membres de l'**union** a été employé :

```
BIF::BIF(bool v) : g(unBool), b(v) {};

BIF::BIF(int v) : g(unInt), i(v) {};

BIF::BIF(float v) : g(unFloat), f(v) {};
```

L'opérateur de comparaison consiste simplement à tester les membres adéquats :

```
bool BIF::operator==(BIF v) {
  if (g != v.g)
     return false;
  else if (g == unBool)
     return b == v.b;
  else if (g == unInt)
     return i == v.i;
  else
     return f == v.f; };
```

L'addition ne pose guère plus de problème, excepté qu'il nous faut parfois effectuer des conversions de type :

```
BIF BIF::operator+(BIF v) {
  if (g == unBool)
     if (v.g == g)
        return BIF(b || v.b);
     else
        return v+*this;
  else if (g == unInt)
     if (v.g == g)
        return BIF(i + v.i);
     else if (g == unBool)
        return BIF(i + (int)(v.i));
     else
        return v+*this;
  else if (v.g == g)
     return BIF(f + v.f);
  else if (v.g == unBool)
     return BIF(f + (float)(v.b));
  else // v.g == unInt
     return BIF(f + (float)(v.i)); };
```

Notons que, pour simplifier, il arrive parfois qu'un appel du genre `a+b` soit transformé en `b+a`, uniquement pour réduire au mieux le nombre de cas possibles (remarque concernant les deux lignes contenant **return** `v+*this`).