Département Informatique

Programmation avancée en C++

Miklós Molnár

9 septembre 2016

Table des matières

Chapitre 1

Utilisation des E/S et des fichiers

Objectifs

— Marquer la différence entre C et C++
— Montrer la structure des fichiers sources
— Comprendre la compilation séparée simple
— Illustrer l'utilisation des E/S en C++
— Présenter les fichiers stream
— Préparer une classe simple

1.1 Résumé

1.1.1 La différence entre C et C++

C++ est aussi proche des langages de bas niveau que C mais il implémente la plupart des concepts orientés objet :

— encapsulation des données et des méthodes
— accessibilité et visibilité "réglables"
— héritage (même multiple) et polymorphisme.

De plus :

— il garde les possibilités des fonctions externes
— il assure la programmation paramétrée, générique (*template*)
— il définit des accès "amis" particuliers
— il implémente les exceptions.

1.1.2 La structure des fichiers sources

On distingue deux types de fichiers :

— fichiers en-têtes (déclaration)
— fichiers avec des codes (définition).

Exemple :

```
\\point.h

class point{
   int x, y;
   public:
   int get_x();
   int get_y();
   void set_x(int);
   void set_y(int);
};

=====================================
\\point.cpp

   int point::get_x() {return x;}
   int point::get_y() {return y;}
   void point::set_x(int xx) {x = xx;}
   void point::set_y(int yy) {y = yy;}
```

1.1.3 Compilation séparée

Le compilateur est `g++` (par exemple). Les options les plus importantes de la compilations sont :

- -o nom_resultat
- -c : compiler les fichiers sources, mais sans édition de liens (les résultats sont des modules d'objet .o)
- -g : activer le débogage (pour gdb par exemple)

Pour regarder les autres options de g++ : `man g++`.

On compile les fichiers de source `.cpp`. Les fichiers en-tête `.h` ou `.hpp` ne contiennent pas des codes et des objets (ils contiennent uniquement des déclarations).

Exemple simple :

On a un programme principal `main.cpp` qui utilise la classe point. La classe point est déclarée dans `point.h` et est définie dans `point.cpp`. On peut compiler le tout par :

```
g++ -o exemple *.cpp
```

Une classe (un type, un objet) doit être connue au moment de son utilisation. Cette règle détermine l'ordre des fichiers à compiler et les dépendances entre fichiers de sources et fichiers en-tête.

1.1.4 Les flots en C

En C, les flots correspondant aux E/S peuvent être manipulés en utilisant des fonctions prévues dans la bibliothèque `stdio` pour cela.

Exemple :

```
                                /************************/
                                /*Table of Sine Function*/
                                /************************/
                                /* Michel Vallieres     */
#include < stdio.h>
#include < math.h>

void main()
{
    int    angle_degree;
    double angle_radian, pi, value;
                                        /* Print a header               */
    printf ("\nCompute a table of the sine function\n\n");
                                        /* obtain pi once for all       */
                                        /* or just use pi = M_PI, where
                                            M_PI is defined in math.h   */
    pi = 4.0*atan(1.0);
    printf ( " Value of PI = %f \n\n", pi );
    printf ( " angle     Sine \n" );
    angle_degree=0;                     /* initial angle value           */
                                        /* scan over angle               */
    while (  angle_degree <= 360 )      /* loop until angle_degree > 360 */
    {
       angle_radian = pi * angle_degree/180.0 ;
       value = sin(angle_radian);
       printf ( " %3d      %f \n ", angle_degree, value );
       angle_degree = angle_degree + 10; /* increment the loop index      */
    }
}
```

Un programme qui lit les caractères de son entrée (`stdin`) à l'aide de `getchar()` et les copie sur sa sortie (`stdout`) avec `putchar()` en comptant le nombre de caractères :

```
#include < stdio.h>
void main()
{
    int i, nc;
    nc = 0;
    i = getchar();
    while (i != EOF) {
nc = nc + 1;
        putchar(i);
i = getchar();
    }
    printf("\n Number of characters in file = %d\n", nc);
}
```

inconvénients :
- les fonctions `read, print, printf, scanf` ... sont hors classe
- des nombreuses classes créées par le programmeur ne sont pas connues par ces fonctions...

Une solution plus souple (celle qu'on aimerait faire) :

Un objet "responsable de la sortie (ou de l'entrée) standard" qui reçoit les objets à afficher et qui réalise l'affichage, si le programmeur de la classe lui a indiqué, comment le faire.

1.1.5 Utilisation des E/S standard en C++

En C++, la bibliothèque standardisée `iostream` contient les définitions et les déclarations nécessaires pour utiliser les E/S standard (`stdin, stdout, stderr`). Fondamentalement, il y a trois classes :

ios

La classe de base est `ios`. Toutes les classes manipulant les flots héritent de cette classe et des méthodes qu'elle définit. Quelques exemples :

- `int ios::good()` : retourne une valeur non nulle si l'E/S est OK
- `int ios::fail()` : retourne une valeur non nulle s'il y a eu un échec lors de l'E/S
- `int ios::eof()` : retourne une valeur non nulle si la fin de fichier est atteinte
- `int ios::width(int n)` : positionne la largeur du champ de sortie
- `char ios::fill(char c)` : positionne le caractère de remplissage
- `char ios::fill()` : retourne le caractère de remplissage
- `int ios::precision(int p)` : définit la précision (le nombre de caractères y compris le point) d'un réel
- `endl` : écrit un `\n` et vide le tampon du flot
- `ends` : écrit un `\0` et vide le tampon du flot
- `flush` : vide le tampon du flot
- `dec` : la prochaine opération d'entrée-sortie se fera en décimal
- `hex` : en hexadécimal
- `oct` : en octal

ostream

En géneral, on utilise deux objets (instances) de cette classe pour écrire
- à la sortie `stdout` (c'est l'objet `cout`)
- à la sortie `stderr` (c'est l'objet `cerr`)

Quelques méthodes de la classe :

- `ostream &ostream::put(char c)` : insère un caractère dans le flot.
- `ostream &ostream::write(const char* s, int n)` : insère n caractères dans le flot.
- `ostream &ostream::flush()` : vide le tampon du flot.

La classe `ostream` surcharge l'opérateur `<<` pour les types prédéfinis du C++ (`char`, `const char *`, `short`, `int`, `float`, `double`, `void *`, `const void *`, ...). Par contre, il faut le surcharger pour les nouvelles classes proposées par le programmeur.

Finalement, les méthodes de la classe `ostream` offrent le même confort que les fonctions de C.

Exemples :

```
cout.put('a');                      ==>a
cout.width(4); cout <<"a";          ==>~~~a
cout.fill('#'); cout <<"a";         ==>###a
cout << 123 << hex << 123;          ==>123#713 //dec, oct, hex, left, right, ...
cout << forme(" surface : %f \n", A); //formatage
```

istream

Entre autres, cette classe est dotée des méthodes suivantes :

- `int istream::get()` : retourne la valeur du caractère lu
- `istream &istream::get(char &c)` : extrait le premier caractère du flot et le place dans c
- `int &istream::peek()` : lit le prochain caractère du flot sans l'enlever (prélecture)
- `istream &istream::read(char* ch, int n)` : extrait au plus n caractères du flot et les place à l'adresse ch. Le nombre de caractères effectivement lus peut être obtenu grâce à la méthode gcount()
- `int istream::gcount()` : retourne le nombre de caractères extraits lors de la dernière lecture
- `istream &istream::flush()` : vide le tampon du flot

La méthode la plus utilisée de la classe `istream` est l'opérateur `>>` pour permettre la lecture dans les variables de type prédéfini du C++. Ici aussi, il faut surcharger cet opérateur pour les nouvelles classes proposées par le programmeur pour une éventuelle utilisation.

```
char initial;
char legume[25];
float f;
cin.get(initial);          <==  a
cin.get(legume,25,'\n');   <==  aubergine\n  // le separateur reste
cin.peek();                <==  \n           // prelecture
cin.putback();             <==               //remise

cin >> f >> initial;                         //lectures simples

if (cin.eof()) ...
```

1.1.6 Utilisation en cascade

```
cout << num << nom << endl;
```

Une écriture équivalente :

```
( (cout.operator<<(num)).operator<<(nom) ).operator<<(endl);
```

1.1.7 Fichiers stream

Les E/S sur les fichiers peuvent être réalisées avec des flots en C++. Dans ce cas, on utilise la bibliothèque `fstream`. Les deux grandes classes permettant de réaliser les opérations sur les flots à partir des fichiers sont :

ofstream

pour les écritures réalisées dans des fichiers. Elle est dérivée de la classe `ostream` et bénéficie de toutes les méthodes définies dans cette classe. Voici un exemple d'utilisation :

ifstream

pour les lectures réalisées dans des fichiers. Elle est dérivée de la classe `istream` et bénéficie de toutes les méthodes définies dans cette classe.

Ces classes possèdent aussi des méthodes liées à l'ouverture (identification) et à la fermeture des fichiers. Voici un exemple d'utilisation :

```
#include <fstream>

int main()
{
   char ch;
   ifstream FE("donnees.txt", ios::in);
   ofstream FS("result.txt", ios::out);
          // Deux modes d'ouverture sont possibles :
          //     ios::out   creation, fichier ecrase si existant
          //     ios::app   ajout en fin de fichier

   if (!FE || !FS) {
        cerr << "Probl\'eme d'ouverture de fichier" << endl;
        exit(1);
   }  // Test d'ouverture de fichiers

   while (!FE.eof()) {
     FE >> ch;
     FS << ch;
   }
  // Fermeture des fichiers
  FE.close();
  FS.close();
}
```

1.2 Questions

Question 1.1 *Quelle est la différence entre déclaration et définition d'une classe ?*

Question 1.2 *A partir de quelles informations peut-on instancier une classe (créer un objet) ?*

Question 1.3 *Un utilisateur d'une classe que doit-il connaître : la définition, la déclaration de la classe ou les deux ?*

Question 1.4 *Quel est le problème avec le fichier suivant ?*

```
                                    //personne.h
class personne{
 string nom;
 int age;
 personne(){string,int};
};
personne::personne(string name, int old): nom(name), age(old)
  {}
```

Question 1.5 *Comment fonctionne la compilation séparée pour des classes et des programmes en C++ ? Quels sont les fichiers manipulés au cours de cette compilation ?*

Question 1.6 *Le programme principal suivant peut-il être compilé ?*

```
void main(){
 point p1, p2;
 p1.set_x(3);
 p1.set_y(5);
 p2.set_x(0);
 p2.set_y(0);
 }
```

Question 1.7 *Supposons que le programme principal utilise des objets de la classe A. La classe A a des attributs de la classe B. Comment organiser leur compilation ?*

Question 1.8 *Comment afficher les attributs d'un type (d'une classe) en C ?*

Question 1.9 *Qelles sont les classes en C++ qui s'occupent des E/S standard ? Quelles sont les méthodes les plus typiques de ces classes ?*

Question 1.10 *Quels sont les objets associés à* `stdin`, `stdout` *et* `stderr` *?*

Question 1.11 *Quels sont les objets qui peuvent être lus et/ou écrits à partir de* `cin` *et de* `cout` *avec les méthodes respectives* `>>` *et* `<<` *?*

Question 1.12 *Quelle est la condition que l'objet* `cout` *puisse afficher un objet de type* `Personne` *?*

Question 1.13 *Pourquoi peut-on réaliser les affichages suivants en cascade ?*
`cout << "la valeur de l'indice : " << i << endl;`

Question 1.14 *Comment associer un* `stream` *à un fichier ?*

1.3 Exercice (TP)

Une société réalisant des sondages a besoin de faire une analyse rapide sur la consommation de fromages des personnes interogées (pas de SGBD sous la main).

Les personnes possèdent un identifiant (entier positif), un nom (chaîne de 15 caractères), on connaît leur département (entier entre 0 et 100), leur âge (entier positif) et leur consommation

estimée annuelle de fromages (réel positif, deux chiffres après la virgule). On possède également un tableau qui contient les données suivantes sur les fromages : nom (chaîne de 10 caractères) et département d'origine.

L'entrée du programme est illustrée selon ce qui suit :

```
brie        77
camembert   61
etorki      64
...
***         \\ fin de la liste des fromages
Dupont          35 56 15.88
Barbier         34 21 9.20
Lavoisier       75 20 21.11
. . .
            \\ EOF pour la fin des personnes
```

Remarque : l'identifiant des personnes doit être généré automatiquement.

Réalisation

Créez un programme en C++ qui lit les données en entrée et remplit deux tableaux : un pour les fromages et un pour les personnes.
Pour cela, créez les classes `fromage` et `personne` les plus simples possibles puis un programme principal.

Le programme doit afficher les informations formatées suivantes :

— liste des fromages triée par ordre croissant de départements
— liste des personnes triée par ordre croissant de leur âge
— liste des personnes qui n'ont pas de fromages dans leur département.

Pour simplifier l'affichage, les classes doivent offrir une méthode `afficher()` qui affiche les données de l'objet en question dans la ligne courante.

Réalisez le programme à partir des E/S standard mais aussi à partir des fichiers contenant les flots (ce qui donne deux programmes).

Chapitre 2

Conception des classes simples

Objectifs

- Enumérer les membres d'une classe : données et méthodes
- Discuter de la visibilité et de l'accessibilité des membres
- Introduire les membres statiques
- Discuter de l'instanciation
- Présenter la différence entre méthodes compilées et méthodes in-line
- Indiquer une relation de dépendance importante : utilisation d'une classe par une autre classe

2.1 Résumé

2.1.1 Membres d'une classe, données et méthodes

Les classes (les objets) peuvent avoir des membres de deux types :

- des attributs (des données)
- des méthodes.

Chaque objet, chaque instance de la classe peut avoir des valeurs différentes (uniques) concernant ses attributs mais toutes les instances de la classe partagent les mêmes méthodes. On peut considérer l'ensemble des valeurs comme l'état de l'objet. L'exécution des méthodes (les résultats) dépend(ent) alors de l'état de l'objet.

Exemple :

```
                    \\point.h
class point{
   int x, y;        \\ attributs
   public:
   int get_x();     \\ methodes
   int get_y();
   void set_x(int);
   void set_y(int);
   void dessiner();
};
```

La méthode `dessiner()` va dessiner un point (un petit cercle ou une autre signe) sur le plan en fonction de l'état (de la position) de l'objet concerné.

2.1.2 Visibilité et accessibilité des membres

Il y a trois catégories des attributs et des méthodes (ces catégories déterminent la visibilité des membres par des classes / utilisateurs et de cette manière l'accessibilité de ces membres)

- `private` : seules les instances de la classe donnée peuvent y accéder
- `protected` : seules les instances de la classe donnée et des classes dérivées peuvent y accéder
- `public` : tous les utilisateurs peuvent y accéder.

Masquage des données

En général, les attributs sont cachés devant les utilisateurs de la classe (ils sont privés ou protégés). Cela assure une indépendance de l'implémentation de la classe. L'utilisateur d'une classe ne doit connaître que les méthodes (l'interface) publiques de la classe.

Accéder et modifier les données

Les méthodes membres permettant d'accéder aux données membres (cachées) sont souvent appelées accesseurs (*getter*). Les méthodes membres permettant de modifier les données membres sont appelées mutateurs (*setter*).

Exemple :

```
                                //carte.h
class carte {
        int _coul;              // couleur
        int _haut;              // hauteur

public :
        carte(int c, int h);    // constructeur
 int    get_coul();             // recuperer la couleur
 int    get_haut();             // recuperer la hauteur
 void   set_coul( int);         // changer la couleur
 void   set_haut( int);         // changer la valeur
};
```

2.1.3 Membres statiques

Les attributs caractérisent l'objet qui les possède et les méthodes peuvent être évoquées à partir d'un objet. Cependant, il peut exister des attributs qui décrivent la classe entière et qui ne sont pas liés à un seul objet : ce sont les attributs indiqués comme `static`. Similairement, on peut créer des méthodes `static` qui portent sur la classe et non pas sur un objet de la classe. Ces attributs et ces méthodes appartiennent à la classe et doivent être évoqués à partir de la classe et non pas à partir d'un objet.

Exemple :

```
                                //carte.h
class carte {
        int _coul;              // couleur
        int _haut;              // hauteur
static int _long, _larg; // dimensions de toutes les cartes

public :
        carte(int c, int h);    // constructeur
 int    get_coul();             // recuperer la couleur
 int    get_haut();             // recuperer la hauteur
 static int get_long(); // recuperer la longueur (methode de la classe)
 static int get_larg(); // recuperer la largeur (methode de la classe)
 void   set_coul( int);         // changer la couleur
 void   set_haut( int);         // changer la valeur
 static void set_long( int);    // changer la taille de toutes les cartes
 static void set_larg( int);    //       ''
 bool   superieur(const carte&); // pour comparer deux cartes
 void   afficher();   // pour afficher

};
```

Attention : les objets `_long, _larg` n'appartiennent à aucune carte existante. Il faut les créer et gérer leurs valeurs...

Une possibilité :

```
                                        // carte.cpp
#include <iostream>
#include "carte.h"

int carte::_long = 40;                  // definitions obligatoires
int carte:: _larg = 25;                 // des membres statiques
                                        //(ici, avec initialisation)
carte::carte(int c, int h) : _coul(c), _haut(h)
{}
 int carte::get_coul() // recuperer la couleur
{ return _coul; }
 int carte::get_haut() // recuperer la hauteur
{ return _haut; }
 int  carte::get_long() // recuperer la longueur
{ return carte::_long; }
 int  carte::get_larg() // recuperer la largeur
{ return carte::_larg; }
 void carte::set_coul(int c)          // changer la couleur
{ _coul = c; }
 void carte::set_haut(int h)          // changer la hauteur
{ _haut = h; }
 void  carte::set_long(int l) // changer la taille des cartes
```

```
{ carte::_long = l; }
 void  carte::set_larg(int l)
{ carte::_larg = l; }
...
```

De plus :

```
int main()
{
        carte::set_long(25);
        carte::set_larg(18);    // valeurs pour toutes les cartes

        carte C (2, 7);         // une carte

        int col = C.get_coul();
        int lo = carte::get_long();// recuperer la longueur
}
```

2.1.4 Ecriture inline de certaines méthodes

En C++, il existe un mécanisme pour appliquer le code d'une méthode sans passer par un véritable appel de méthode (sans passer par la pile). Pour cela, on introduit les méthodes `inline` qui se comportent comme les macros sans cette fois passer par le préprocesseur. Pour une méthode `inline`, le compilateur se charge de faire le remplacement de code au moment de la compilation. (Chaque fois quand une méthode `inline` est appellée, son code est copié à l'endroit de l'appel ; ce qui augmente la taille du programme...) En général, on utilise les méthodes inline pour des méthodes simples et courtes. Il y a deux possibilités pour définir des méthodes inline :

- on donne le code da la méthode dans le fichier .h
- on indique la méthode par le mot-clé `inline` dans le fichier .cpp

```
class carte {                    // dans carte.h
        int _coul;               // couleur
        int _haut;               // hauteur
. . .
 bool   superieur(const carte& c)
 { return _haut > c._haut;}
};
```

ou encore

```
#include <iostream>             // dans carte.cpp
#include "carte.h"
...
 inline bool   carte::superieur(const carte& c)
 { return _haut > c._haut;}
```

2.1.5 Utilisation d'une classe par une autre classe

Une classe peut utiliser une autre : par exemple, les attributs d'une classe peuvent être des instances d'autres classes.

Exemple :

```
                                //paquet.h
#include "carte.h"

class paquet {
        int __nb;               // nombre de cartes
        carte * _C;             // cartes / tableau
. . .
```

ou encore

```
#include "carte.h"
#include "paquet.h"
int main()
{
        carte::set_long = 25;
        carte::set_larg = 18;   // valeurs pour toutes les cartes

        carte C (2, 7);         // une carte
        paquet P;               // un paquet (vide)
 ...
}
```

Combien de fois la déclaration de la classe `carte` est-elle inclue dans le programme ? Comment éviter les multiples déclarations (non voulues) ? Pour cela, on utilise un simple mécanisme du préprocesseur. Chaque fichier en-tête doit contenir les instructions de contrôle de définition suivantes :

```
#ifndef CARTE_H          //ou encore  #pragma once
#define CARTE_H
                                //carte.h
#include <iostream>
using namespace std;

class carte {
. . .
};
#endif
```

2.1.6 Dépendances entre les classes et entre les fichiers pour la compilation

Ici, on remarque que des dépendances existent entre les classes (utilisation,...) mais aussi entre les fichiers `.h` et `.cpp`. Pour gérer correctement les `#include`, il est intérressant de découvrir le graphe de dépendances.

2.2 Questions

Question 2.1 *quel est l'intêret des membres* `private`, `protected` *et* `public` *?*

Question 2.2 *Quelle est la visibilité (accessibilité) des membres de la classe suivante ?*

```
class produit {
 string nom;
 float quantite;
 string stock;
 int categorie;
};
```

Question 2.3 *Comment accéder aux attributs de cette classe ?*

Question 2.4 *On aimerait avoir une méthode* `get_categorie()` *accessible aux membres de la classe mais interdite pour un usage extern. Comment faire ?*

Question 2.5 *Quel est l'intêret des membres statiques ?*

Question 2.6 *On possède une carte* `carte A(2,7);` *dans un programme. Supposons que la classe* `carte` *correspond à la classe vue ci-dessus. Pour obtenir la longeur de A, on propose* `int i = A.get_long();` *Est-ce marche ? L'instruction* `int i = get_long();` *fonctionne-t-elle mieux ?*

Question 2.7 *Quand et comment écrire des méthodes* `inline` *? Peut-on mettre le code (la définition) d'une méthode* `inline` *dans la fichier* `.cpp` *?*

Question 2.8 *Quelle est la difficulté des méthodes* `inline` *? (Dans quels cas faut-il les éviter ?)*

Question 2.9 *Des classes peuvent utliser d'autres classes. Comment éviter les multiples définitions des classes ?*

Question 2.10 *Un programme utilise les classes* `paquet`, `carte` *et* `joueur`. *Chaque classe possède un fichier de définition et un fichier de déclaration. La classe* `joueur` *utilise la classe* `paquet` *et la classe* `paquet` *utilise la classe* `carte`. *Donnez le graphe de dépendances des fichiers.*

2.3 Exercice (TP)

On reprend l'informatisation du sondage utilisé dans le TP précédant. Les personnes et les fromages sont à manipuler dans cette exercice. Cependant, au lieu d'utiliser des tableaux, on veut réaliser des listes de personnes et de fromages sans utiliser des classes `list` standardisées. L'idée de base de l'enchaînement d'éléments est illustrée par la figure 2.1. Chaque élément de la liste connaît son successeur dans la liste et le premier élément est donné par un attribut de la classe (et pas par un attribut d'un objet). Pour donner le nom des fromages et des personnes, on propose l'utilisation de la classe `string` (voir les méthodes les plus importantes dans la figure 2.2.

Réalisation

Créez un programme en C++ qui lit les données en entrée et remplit deux listes : une à l'intérieur de la classe `fromage` et une autre dans la classe `personne`.
Les éléments qui nécessitent une attention particulère sont les suivants :

1. L'implémentation des classes doit être protégée

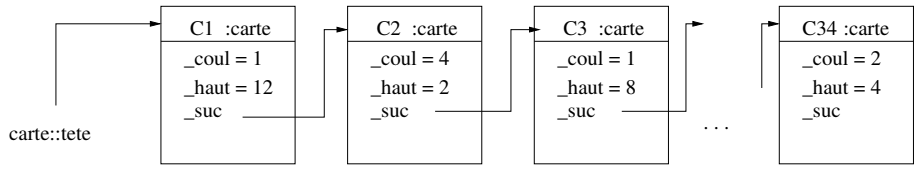

FIGURE 2.1 – La liste réalisée dans la classe

FIGURE 2.2 – Quelques méthodes de la classe string

2. L'accès et la modification des données doivent être réalisés par des accesseurs et des mutateurs
3. L'identification automatique des personnes doit être réalisée par un compteur statique de la classe
4. Les accesseurs et les mutateurs non statiques peuvent être implémentés inline
5. Réalisez les listes à l'aide des membres statiques
6. La gestion dynamique des objets (la création dynamique des éléments des listes) doit être réalisée soigneusement (y compris la destruction des objets alloués dynamiquement)
7. Utilisez les `string`s au lieu des `char *`.

Ici aussi, le programme doit afficher les informations formatées suivantes :
— liste des fromages triée par ordre croissant de départements
— liste des personnes triée par ordre croissant de leur âge
— liste des personnes qui n'ont pas de fromages dans leur département.

Chapitre 3

Classes "opératoires", opérateurs

Objectifs

- Parcourir les opérateurs en C++
- Différencier opérateurs hors classe et opérateurs membres d'une classe
- Voir le mécanisme de surcharge d'opérateurs
- Protection des paramètres et de l'objet implicite

3.1 Résumé

3.1.1 Opérateurs en C++

En C++, les éléments du langage correspondent souvent à des opérateurs. L'exemple classique des opérateurs est l'ensemble des opérateurs mathématiques (l'addition, la soustraction, la multiplication, la division et le modulo). C++ permet de les utiliser (surcharger) simplement dans les classes développées par des utilisateurs.

Les opérateurs sont des symboles qui permettent de manipuler simplement des objets. On peut les classer dans plusieurs catégories :

- les opérateurs mathématiques (+, -, *, /, %, ...)
- les opérateurs d'affectation et d'assignation (=, +=, -=, ...)
- les opérateurs d'incrémentation (++, - -)
- les opérateurs de comparaison (==, >, <, >=, <=, !=)
- les opérateurs de parenthèsage ((),[], ...)
- les opérateurs logiques (||, &&, !)
- les opérateurs sur les bits comme et, ou, décalages, etc (|, &, ˆ, <<, >>, ...)
- d'autres opérateurs correspondant à des éléments syntaxiques du langages...

Priorité des opérateurs												
++++++++++++	()	[]										
++++++++++	--	++	!	~	-							
++++++++++	*	/	%									
+++++++++	+	-										
++++++++	<	<=	>=	>								
+++++++	==	!=										
++++++	^											
+++++	|											
++++	&&	||										
+++	?	:										
++	=	+=	-=	*=	/=	%=	<<=	>>=	>>>=	&=	^=	|=
+	,											

FIGURE 3.1 – Priorité des opérateurs

3.1.2 Surcharge d'opérateurs

Les objets d'une classe peuvent recevoir des messages basés sur des symboles-opérateurs, si la classe contient la définition de l'opérateur. Exemples dans la classe carte :

```
                                //carte.h
class carte {
        int _coul;              // couleur
        int _haut;              // hauteur
public :
...
 bool   operator>(const carte& c) // pour comparer deux cartes
 { return _haut > c._haut; }
};
```

Remarquons que le paramètre de cette méthode est passé par référence (on verra les références en détail plus tard) et que cet objet passé est protégé contre les modifications (il est constant).

On peut alors appliquer cet opérateur partout où les cartes sont utilsées :

```
int main() {
        carte C1 (2, 7);        // une carte
        carte C2 (4, 1);        // une autre

        if (C1 > C2)
            cout << " C1 a gagne la bataille \n";
}
```

La condition peut aussi être écrite de la façon suivante :

```
        if (C1.operator>(C2))
            cout << " C1 a gagne la bataille \n";
```

3.1.3 Opérateurs hors classe, opérateurs membres d'une classe

Par rapport à des opérateurs définis dans les classes, on peut noter les difficultées suivantes : l'objet qui reçoit le message doit être un objet de la classe (c'est l'objet C1 dans l'exemple). Cepandant, on ne peut pas toujours assurer que l'objet devant l'opérateur soir une instance de la classe concernée. Exemples :

```
int main()
{
        complexe V1 (5.1, -3.3);          // une valeur complexe
        complexe V2 (0.008, 6.33);        // une autre

        V1 = V1 + V2;             // marche si operator+ existe...
        V1 = 4 + V2;           // ne marche pas...
        cout << V1;               // ???
}
```

Solution : surcharge des opérateurs hors classe avec deux paramètres.
Exemples :

```
                                //complexe.h
class complexe {
        float _x;              // x
        float _y;              // y
public :
   complexe(int i): _x(i), _y(0) // constructeur int -> complexe
    {}
...
 friend complexe  operator+(const complexe& c1, const complexe& c2) {
 { complexe res;     // resultat de l'addition
   res._x = c1._x + c2._x;
   res._y = c1._y + c2._y;
   return res;
 }
};
```

Pour que cette fonction puisse accéder aux membres de la classe `complexe`, cette méthode doit être déclarée comme "amie" de la classe.

De plus, il faut permettre que l'objet qui a reçu le message (dans notre exemple il est l'entier 4) puisse être transformé en un objet de type `complexe`. Ce qui nécessite la présence d'un constructeur particulier...

3.1.4 Classes et fonctions amies

Une méthode de la classe peut accéder aux données (même privées) de l'objet évoqué, d'un autre objet de la classe ou aux données statiques de la classe. Elle est appelée à partir d'un objet (l'objet implicite).

Une fonction externe peut être déclarée amie (mot clé `friend`) ; elle va posséder le droit d'accès aux données comme les membres de la classe. Des opérateurs externes opérant sur deux objets peuvent exécuter des opérations quand il n'y a pas d'opérateur correspondant dans la classe.

Remarque : une classe entière peut être `friend` : dans ce cas, tous les membres de cette dernière classe possèdent les accès aux membres de la première.

3.1.5 Avantages et inconvénients des "amies"

Avantages : flexibilité, possibilité de l'utilisation des outils du compilateur (par exemple les conversions) pour résoudre simplement des problèmes compliqués

Inconconvéniants : on perd le contrôle de cohérences des classes concernées (accès à des membres privés)

3.2 Questions

Question 3.1 *Quel est l'intêret de la surcharge des opérateurs de C++ ?*

Question 3.2 *Supposons que la classe* `logic` *traite les valeurs booléennes et surcharche tous les opérateurs utilisés pour des opérandes booléennes. Supposons que la classe* `array` *manipule des tableaux d'entiers et elle aussi possède tous les opérateurs nécessaires pour cela. Comment réécrire l'expression suivante à l'aide des opérateurs (à l'aide des appels explicites* `.operatorX`*, imbriqués) ?*

```
int main()
{
       array M(15);                  // un tableau de 15 entiers
       logic A;

         A = M[3]++ == M[5]+2 ;
}
```

Question 3.3 *Dans le langage, il existe un* `operator=` *par défaut. Que fait cet opérateur ?*

Question 3.4 *Supposons qu'un programme définit les opérateurs et les constructeurs suivants :*

```
class Fraction {
...
public :
       Fraction( int n);
       Fraction( int n,  int d );
...
 bool  operator==(const Fraction& f) const
 { return ...;}
```

```
 friend bool  operator==(const Fraction& f1, const Fraction& f2);
};
...
 bool  operator==(const Fraction& f1, const Fraction& f2)
 { return ...;}
```

Quels sont les opérateurs exécutés dans les cas A,B et C suivants ?

```
int main()
{
        Fraction F1 (2, 3);         // la fraction 2/3
        Fraction F2 (4);            //  la fraction 4/1

        if (F1 == F2)  ...          // A
        if (2  == F2)  ...          // B
        if (F1 ==  3)  ...          // C
}
```

Comment résoudre les problèmes d'ambiguïté ?

Question 3.5 *Le programme suivant utilise les fractions.*

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

int main() {
        Fraction F1 (2, 3);         // la fraction 2/3

        cout << F1 << endl;
}
```

Quel est le problème de l'opérateur << pour l'affichage ? Quel objet reçoit le message ? Peut-on surcharger l'opérateur dans sa classe pour la classe `Fraction` *? Comment procéder ?*

Question 3.6 *Quel est l'intêret des fonctions et des classes amies (*`friend`*) ?*

Question 3.7 *Où faut-il déclarer les amies (*`friend`*) ?*

Question 3.8 *Supposons les classes* `personne` *et* `fromage`*. On veut rendre la classe* `personne` *amie de la classe* `fromage`*. Détaillez ce qu'il faut mettre dans les fichiers* `.h`

Question 3.9 *Donnez les dépendances sous forme graphique entre* `personne.h`*,* `fromage.h`*,* `personne.cpp` *et* `fromag.cpp` *de la question précédente*

Question 3.10 *Soit la première version des classes* `A` *et* `B` *comme suit :*

```
                              // fichier  A.h
class A {
   int a1;
   float a2;
 public:
   A(int i, float f):a1(i), a2(f)
    {}
 ...
};
==========================================
                              // fichier  B.h
#include "A.h"
class B {
   int b1;
   A b2;
 public:
   B(int i, A a):b1(i), b2(a)
    {}
 ...
};
```

Pour faciliter certaines méthodes (non détaillées ici), on veut déclarer la classe `B` *comme amie de la classe* `A`*. Comment procéder ?*

3.3 Exercice (TP)

Dans l'espace euclidien, les vecteurs sont donnés par des triplets (par exemple : par leurs coordonnées x,y et z). L'algèbre de vecteurs définit l'addition, la soustraction et deux multiplications (les multiplications scalaire et vectorielle). On peut aussi multiplier un vecteur par une valeur scalaire. La valeur absolue d'un vecteur (sa longueur) est souvent utilisée.

On aimerait manipuler les vecteurs à l'aide d'une classe `Vecteur`. L'affichage direct des vecteurs sur la sortie standard et la lecture des vecteurs de l'entrée standard sont nécessaires. Voici un programme, qui utilise des vecteurs.

```
#include <iostream>
using namespace std;
#include "Vecteur.h"
main()
{
     Vecteur v1;
     cin >> v1;
     Vecteur v2(1.0,2.0,3.5);
     cout << "Les deux vecteurs sont : " << v1 << " " << v2 << endl;
     cout << "la projection de v1 sur x : " << v1.get_x() << endl;
     cout << "la projection de v1 sur y : " << v1.get_y() << endl;
     cout << "la projection de v1 sur z : " << v1.get_z() << endl;
```

```
    cout << "La valeur absolue de v1 : " << v1.abs() << endl;
    cout << "la somme des deux vecteur : "  << v1+v2 << endl;
    cout << "leur difference : " << v1-v2 << endl;
    cout << "leur produit scalaire : " << v1*v2 << endl;
    cout << "leur produit vectoriel : " << (v1^v2) << endl;
    cout << "le double de v1 : "  << 2*v1 << endl;
    cout << "pi fois v2 : " << v2*3.1417 << endl;
    cout << "le resultat du test d'egalite : "  << (v1==v2) << endl;
}
```

Ecrivez et testez la classe `Vecteur`.

Chapitre 4

Objets et objets référencés

Objectifs

— Distinguer objets, pointeurs et références
— Connaître portée et durée de vie des objets
— Allocation dynamique en C++
— protection des objets

4.1 Résumé

4.1.1 Objets, pointeurs, références

On sait que les objets d'un programme sont des objets abstraits en informatique : des variables, des constants. Ils sont placés quelque part en mémoire. De cette façon, ils possèdent chacun une adresse (connue ou pas).

Un **pointeur** est aussi une variable, il est destiné à contenir une adresse mémoire, c'est à dire une valeur identifiant un emplacement en mémoire.

```
int  i;     // entier
int * p;    // pointeur
p = &i;
p++;
```

Une **référence** est un nom alternatif (un alias) d'une variable. Elle peut être utilisée partout ou le nom de la variable peut l'être.

```
int  i;        // entier
int & r(i);    // reference de l'objet i
r++;           // on incremente la valeur de i
```

On peut identifier un objet par une référence, modifier le contenu de la variable en utilisant une référence, etc. Cependant, la référence est aussi un objet à part... L'exemple suivant illustre la différence :

```
Image I(1024, 512);        // une image de 1024 * 512 pixels
Image& R(I);               // reference sur l'image
```

```
bool B1= filtre1(I);    // appel de filtre1 en passant l'objet I
bool B2= filtre2(R);    // appel de filtre2 en passant la reference
```

Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée. Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

4.1.2 Utilisation des pointeurs et des références

En C, on utilise des pointeurs pour accéder à des objets passés en paramètres permettant leur modification, pour éviter la charge de la pile, etc... En C++, de plus, on peut utiliser les pointeurs pour assurer le polymorphisme (et faire des traitements homogènes sur un ensemble d'objets hétérogènes, ...)

On peut atteindre les mêmes objectifs avec les références.

La grande différence est qu'un pointeur peut designer des emplacements différents au cours du temps, tandis qu'une référence ne peut référencer que l'objet utilisé pour son initialisation.

4.1.3 Variables globales, locales et allocations dynamiques

Les **variables globales** sont déclarées au niveau supérieur d'un programme (dans le `main()`). Elles sont à l'extérieur de toute fonction ou de tout bloc et sont accessibles de partout dans le code. Elles sont stockées dans le **zone de données**.

Les **variables locales** sont des variables qui sont déclarées à l'intérieur d'un bloc d'instructions (entre des accolades, dans une fonction ou une boucle par exemple). Elles sont limitées à ce seul bloc d'instruction, c'est-à-dire qu'elle sont inutilisables ailleurs. Physiquement, les variables locales sont allouées au fur et à mesure dans la **pile**.

Les **variables allouées dynamiquement** nécessitent une création (typiquement avec `new` en C++) et une destruction explicite (avec `delete`). Elles sont allouées dans le **tas** jusqu'à leur destruction explicite. Pour y accéder, en général, il faut connaître leur adresse (stockée par un pointeur) ou leur référence.

4.1.4 Portée et durée de vie

Selon l'endroit où l'on déclare la variable, elle pourra être accessible (visible) de partout dans le code ou bien que dans une portion limitée de celui-ci (à l'intérieur d'un bloc, d'une fonction par exemple), on parle de portée (ou visibilité) de la variable. La portée

- d'une variable globale : partout dans le programme
- d'une variable locale : uniquement le bloc qui la définit
- d'une variable allouée dynamiquement : partout ou l'on possède un accès

La durée de vie

- d'une variable globale : pendant toute l'exécution du programme
- d'une variable locale : à partir de l'entrée dans le bloc jusqu'à ce qu'on sort
- d'une variable allouée dynamiquement : depuis sa création explicite jusqu'à sa destruction explicite

4.1.5 Constructeur par défaut

Souvent, pour créer des variables, on utilise la formule :

```
type nom; //exemple :  carte c;
```

Cela suppose que la construction sans aucun paramètre est possible : la classe possède le constructeur nécessaire. Un constructeur est une méthode particulière qui est appelée pour créer un objet.

Si dans une classe, aucun constructeur n'est donné, C++ génère deux constructeurs :

— le constructeur par défaut

— le constructeur par copie

Attention, si le programmeur de la classe ajoute un constructeur explicite à la classe, le constructeur par défaut généré par défaut est perdu.

4.1.6 Protection des objets, des paramètres et de l'objet implicite

Les objets peuvent être protégés contre tous modifications. En général, c'st le mot clé `const` qui est utiliser pour la protection.

Le C++ veille sévèrement à la cohérence de l'utilisation des différentes protections. Analysez l'exemple siuvant :

```
        int i;
        const int j(5); // int protege
        i = 0;
//      j++;     // on ne peut pas incrementer j
        int * p1;
        p1 = &i;
//      p1 = &j; // attention, ca ne marche pas...
        const int * p2;  //pointeur sur const int
        p2 = &i;  // ca va...
        p2 = &j;  // OK
        int * const  p3(&i); // pointeur constant sur int
//      int * const  p4(&j); // ne marche pas
        const int * const  p4(&j); // OK
```

La protection est particulièrement important pour être sûr que certaines méthodes et fonctions ne modifient pas les objets manipulés.

4.2 Questions

Question 4.1 *Quelle est la différence entre variables, références et pointeurs ?*

Question 4.2 *Corrigez si nécessaire le code suivant*

```
// pointeur sur un entier
       int* p;
       p = new int;
       delete p;
```

```
// pointeur sur un entier constant
       const int* q;
       q = new int;
       delete q;
// pointeur constant sur un entier
       int* const r;
       r = new int;
       delete r;
// pointeur sur un tableau
       int** tÂ ;
       t = new int*Â ;
       *t = new int[10];
       delete [] t;
```

Question 4.3 *Quel est le résultat de l'affichage dans les deux cas suivants ?*

```
int val = 5;
int * pt = &val.
*pt = 1;
pt++;
cout << val<< *pt;
cout << &val << pt;
//==================
int & ref = val;
ref = 1;
ref++;
cout << val << ref;
cout <<&val << &ref;
```

Question 4.4 *Comment utiliser la variable* `char & ref1;` *?*

Question 4.5 *Vérifiez la faisabilité des opérations suivantes :*

```
int* p = new int;
const int* q = new int;
int* const rÂ = new int;
q = p;
r = p;
//==================
int& s = *q;
const int& t = *q;
p = r;
q = r;
int& s = *r;
const int& t = *r;
//==================
int& s = *p;
```

```
const int& t = *p;
p = q;
r = q;
//==================
int a;
int& s = a;
p = &s;
q = &s;
r = &s;
const int& t = s;
```

Question 4.6 *Les grandes lignes de la classe* `Image` *sont proposées selon ce qui suit*

```
class Image {
const int size_x = 640;
const int size_y = 480;
int _pixel[size_x] [size_y]; // matrice des pixels
unsigned _ligne_cour;        // pixel courant
unsigned _colonne_cour;
...
public :
...
void get_pixelCour(unsigned & px, unsigned & py)
{ px = _colonne_cour; py = _ligne_cour; }
image& interference ( image & Im)
{ image Res; // objet local
  . . .                          // calcul de l'interference
return Res;
}
unsigned* getNbPixels() {
unsigned result = size_x * size_yÂ ;
return &result;
}
```

Faut-il protéger l'argument implicite et les paramètres (px,py et Im) des méthodes mentionnées ?
La méthode `interference` *fonctionne-t-elle correctement ?*
Même question pour la méthode `getNbPixels` .

Question 4.7 *Révisez votre classe* `Vecteur` *et appliquez la protection des objets explicites et implicites partout où c'est nécessaire.*

4.3 Exercice (TP)

On veut manipuler les graphes valués, non orientés. Les arêtes et les sommets sont énumérés (possèdent un numéro identifiant chacun). Un sommet peut être connecté à plusieurs arêtes et une arête relie toujours 2 sommets distincts entre-eux.

1. Proposez le diagramme de classes. Indiquez aussi les attributs des classes. Supposons que les associations du modèle doivent être utilisées (parcourues) dans les deux sens.
2. Ecrivez les classes en C++ avec les méthodes de base : ajout de sommet et d'arête
3. Ecrivez une fonction permettant de découvrir si deux sommets sont connectés. En d'autres termes existe-t-il un chemin reliant ces 2 sommets.

+ Exercices supplémentaires :

Ajoutez des lignes d'affichage dans les constructeurs et dans les destructeurs des classes pour tracer le moment quand ces méthodes sont appelées (par exemple : `cout << "sommet: contructeur par copie" << endl;`). Observez et testez l'ordre de la construction et de la destruction des objets composés. Quels sont les constructeurs et les destructeurs appelés au moment du passage des objets comme paramètres d'une méthode et quand on passe les paramètres par des références ? Suivez également la durée de vie des objets créés.

Chapitre 5

Objets composés, gestion de la représentation

Objectifs

— Connaître l'agrégation
— Distinguer agrégations fortes et faibles
— Analyser la constuction et la destruction des objets
— Maîtriser l'allocation dynamique et la gestion des membres alloués dynamiquement
— Maîtriser la construction et la destruction en profondeur

5.1 Résumé

5.1.1 Agrégations, agrégations fortes et faibles

L'agrégation est une relation asymétrique, qui exprime une prépondérence ou une composition.

Quand un objet est composé d'autres objets, on parle souvent de l'**agrégation forte**. Dans ce cas, les cycles de vies des "composants" et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi. Une instance de composant ne peut être liée qu'à un seul agrégat.

Dans le cas d'une **agrégation faible**, la prépondérence existe entre les objets qui possèdent une existence propre chacun qui est "indépendante" de l'existence des autres.

En C++, les associations (et de cette façon les agrégations) sont implémentées à l'aide des attributs. Dans certains cas, cette implémentation peut impliquer qu'un objet est imbriqué par un autre objet composé. Dans d'autres cas, la relation est plus faible et il ne s'agit pas d'une composition. L'analyse des relations entre les objets facilite la conception des classes et l'écriture des constructeurs et des destructeurs adéquats.

5.1.2 Construction des objets

Les membres d'un objet C++ (les attributs) sont construits dans l'ordre de leur énumération et en utilisant un constructeur de leur classe. S'il n'y a pas d'autre constructeur appelé explicitement pour un membre, alors c'est son constructeur par défaut qui est utilisé (si celui n'existe pas, on obtient un message d'erreur...). L'exemple suivant indique comment écrire des constructeurs.

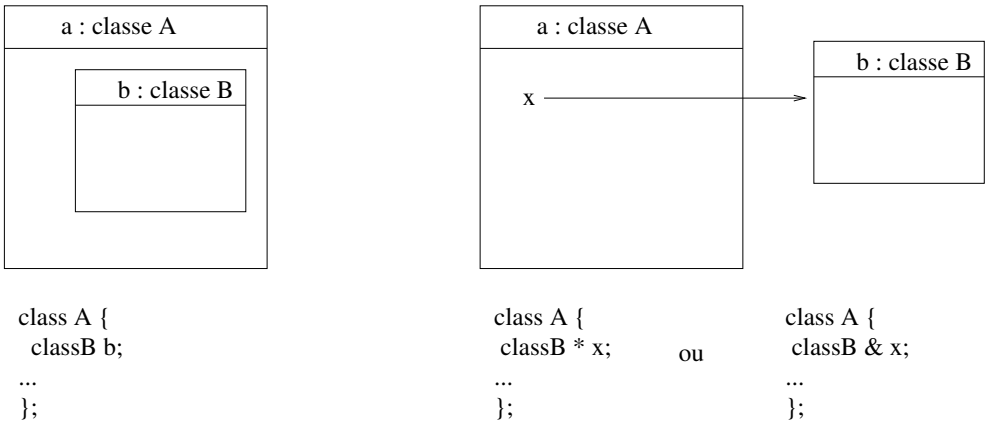

FIGURE 5.1 – Premier cas : un objet est incorporé par un autre ; deuxième cas : la relation entre deux objets est plus faible

```
class tableau {
       int _nb;       // nombre d'elements
       int * _T;      // tableau C
 public :
      ...
     tableau(): _nb(0), _T(NULL)     // constructeur par defaut
      { }
     tableau(int n): _nb(n), _T(new int[n]) // constructeur
      { }
     tableau(int n, int * T): _nb(n), _T(new int[n]) // constructeur
      { for (int i = 0; i< n; i++)
         _T[i] = T[i];
      }
...
```

Question 5.1 *Cependant, une question se pose : Un utilisateur de la classe peut tenter la création d'un objet "tordu" :*
`tableau A(-5);`
Comment éviter ce problème ?

Les trois constructeurs peuvent être regroupés en utilisant les valeurs par défaut des paramètres :

```
class tableau {
       int _nb;       // nombre d'elements
       int * _T;      // tableau C
 public :
      ...
     tableau(int n = 0, int * T = NULL): _nb(n), _T(NULL) // 3 constructeurs
      {
      if (n > 0) {
        _T = new int[n];
```

```
        for (int i = 0; i< n; i++)
          if (T != NULL)
              _T[i] = T[i];
          else
               _T[i] = 0;
        }
      }
...
```

Les appels possibles de ce constructeur sont :

```
tableau A1;
tableau A2(3);
tableau A3(3, tab);
```

`int tab[] = {33, 5, 7};` est un cas particulier : on va créer un tableau pour 3 éléments puis chaque élément sera initialisé.

Important : les objets membres sont **créés et initialisés** selon les constructeurs appelés explicitement ou implicitement selon l'en-tête du constructeur. Le code entre { } est exécuté **après** la construction.

5.1.3 Destruction des objets

Quand il faut détruire un objet (parce qu'on quitte le bloc qui le définit ou bien parce que l'objet est alloué dans le tas et il reçoit le message `delete`), les membres de l'objet sont détruits **automatiquement** dans l'ordre inverse de leur création. Pour compléter la destruction, on peut écrire une méthode qui est appelée destructeur. Cette méthode est exécutée **avant** la destruction des membres.

Exemple :

```
tableau * pM = new tableau(5); // creation d'un tableau dans le tas
...
delete pM;
```

L'instruction `delete pM` détruit les membres de l'objet pointé : le membre`_nb` et le pointeur `_T` de l'objet sont détruits. Les entiers alloués dans le tas ne sont pas détruits. Pour compléter la destruction, il faut écrire un destructeur :

```
class tableau {
      ...
public :
      ...
     ~tableau()
     { if (_nb > 0)
         delete[] _T;
     }
     ...
```

5.1.4 Gestion des membres alloués dynamiquement

On ne peut pas automatiser la destruction des objets alloués dynamiqement. Cependant, si les constructeurs contiennent des allocations dynamiques, il est fort probable que le desctructeur est aussi nécessaire.

En général, il faut analyser le cycle de vie et l'existence des objets pour utiliser la stratégie correcte pour la construction et la destruction des objets.

L'exemple suivant indique des objets indépendants. La relation entre les objets est réalisée à l'aide des pointeurs.

```
class matiere {
     string  _nom_m;
     etudiant * _premier;
 public :
      matiere(string n, etudiant * p): _nom_m(n), _premier(p)
      {}
      ...
 };
 class etudiant {
 ...  };
```

La destruction d'une matière ne doit pas provoquer la destruction de l'étudiant référencé par cette matière.

Par contre, dans le cas de la réalisation d'une agrégation forte à l'aide des pointeurs (ou des références), la gestion cohérante des membres est nécessaire(voir ci-dessous).

5.1.5 Construction et destruction en profondeur

Un premier exemple est la classe `tableau` vue ici. Cette classe correspond à une agrégation forte d'entiers.

Dans un deuxième exemple, on considère un document qui possède une page de titre (qui peut être elle aussi composée) et plusieurs chapitres (qui sont composés de sections,...). Le titre et les chapitres existent uniquement dans le document en question. L'agrégation forte du titre et celle des chapitres sont réalisées par un pointeur et un tableau de pointeurs respectivement (cf. la figure).

```
class document {
     titre * _t;
     chapitre ** _c;
     int _nb_c;
 public :
   document(titre t, int n = 0, chapitre * c): _t(new titre(t)),
                                                _nb_c(n), _c(NULL)
   { if (n > 0) {
      _c = new (chapitre *) [n];      // tableau de pointeurs
       for (int i = 0; i < n; i++)
         _c[i] = new chapitre(c[i]); // allocation des chapitres
      }
```

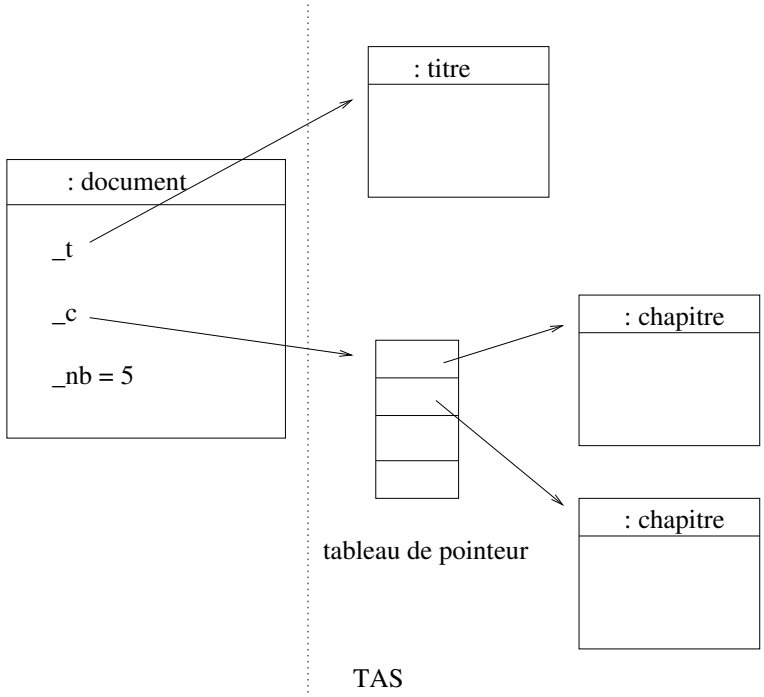

FIGURE 5.2 – Schéma d'allocations dynamiques pour la classe document

```
  }
  ~document()
  {
  delete _t;                          // destruction du titre
  for (int i = 0; i < _nb; i++)
        delete _c[i];                 // destruction des chapitres
  delete[] _c;                        // destruction du tableau de pointeurs
  }
  ...
};
```

5.1.6 Constructeur par copie

Le constructeur qui fait une copie est toujours nécessaire (il existe toujours). La version fournie par le compilateur ne fait qu'une copie simple des membres de données de l'objet. Dans certain cas, c'est suffisant (par exemple pour copier une matière), dans d'autres cas non. Pour copier un tableau d'entiers, on a l'appel :

```
tableau T1(5);
   ...
tableau T2(T1);     // initialisation par copie
```

Pour pouvoir faire la copie entière de l'objet, on doit réécrire ce constructeur dans la classe :

```
class tableau {
```

```
       int _nb;      // nombre d'elements
       int * _T;     // tableau
 public :
      ...
     tableau(const tableau & R): _nb(R._nb), _T(NULL) // constructeur
      { if (_nb > 0) {
          _T = new int[_nb];
           for (int i = 0; i< _nb; i++)
              _T[i] = R._T[i];
         }
      }
...
```

5.1.7 Problème de l'opérateur=

Il existe une affectation par défaut (operator=) en C++ qui effectue une simple copie champ par champ. Si `T1` et `T2` sont des tableaux, `T1 = T2` va copier la taille `_nb` et le pointeur `_T` de l'objet `T2` dans `T1` (deux pointeurs vont pointer sur le même tableau. Pour éviter cela, il faut réécrire l'opérateur= :

- libérer la mémoire de l'argument gauche
- copier l'argument droit

```
class tableau {
      ...
 public :
      ...
     tableau& operator=( const tableau& T)
     { if (this == &T)    // pour le cas A = A;
         return this*;
     if (_nb > 0)         // liberer la memoire
         delete[] _T;
     _nb = T._nb;         // copier
     if (_nb > 0) {
        _T = new int[_nb];
        for (int i = 0; i< _nb; i++)
         _T[i] = T._T[i];
        }
        }
     ...
```

5.2 Questions

Question 5.2 *Quelle est la différence entre "composition" et "agrégation faible" ?*

Question 5.3 *Quelle est la relation entre les classes A et B dans l'exemple suivant ?*

```
class A {
      B b;
     ...
```

Question 5.4 *Pourquoi le constructeur par copie doit-il exister dans chaque classe ?*

Question 5.5 *Le constructeur par défaut de la classe suivante existe-il ?*

```
class A {
      int k;
 public :
     A(int i):k(i)
     {}
     ...
```

Que se passe-t-il si l'on veut creer un objet de type `B` *avec le constructeur par défaut de* `B` *?*

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

Question 5.6 *Est-ce la version suivante fonctionne mieux ?*

```
class B {
      A a;
 public :
     B()
     { A w(0);
      a = w;}
     ...
```

(Quand est-ce que le corps d'un constructeur est executé ?)

Question 5.7 *Enumérez les constructeurs possibles de la classe* `tableau` *(classe qui gère des tableaux d'entiers).*

Question 5.8 *Proposez un constructeur par copie pour la classe* `document`.

Question 5.9 *Que fait la méthode appelée "destructeur" ?*

Question 5.10 *Pourquoi* `operator=` *retourne une référence ?*

Question 5.11 *Les constructeurs, le destructeur et* `operator=` *contiennent des opérations similaires. Comment assurer une implémentation homogène ?*

5.3 Exercice (TP)

5.3.1 Avant-propos

Un des types de données le plus couramment utilisé est celui des chaînes de caractères. Il existe en C++ ANSI une classe string permettant de gérer un certain nombre d'opérations standards sur les chaînes de caractères. Le but de ce TP n'est pas la création d'une classe concurrentielle mais la discussion des problèmes de la gestion de mémoire qu'on peut rencontrer quand on construit une telle classe. En général, l'utilisation des classes préfabriquées de C++ est vivement conseillée.

5.3.2 Définition de la classe chaîne

Données membres

Plusieurs possibilités d'implémentation existent. Vous pouvez utilisez les tableaux de caractères se terminant obligatoirement par `'\0'` comme en C, ou gérer et mémoriser la longueur de la chaîne à part. Quelque soit votre choix, soyez cohérent avec lui dans la suite.

Constructeurs et destructeur

Pensez à la mise en oeuvre de différents constructeurs et un destructeur qui doivent être définis pour notre classe chaîne.

```
ex :    chaine ();
        chaine (const char *);
        chaine (const chaine &);
```

Pour pouvoir suivre la construction et la destruction des objets, complétez les constructeurs et le desÂtructeur par des affichages qui indiquent leurs appels sur la sortie standard.

Opérations

Il est intéressant de fournir aux utilisateurs de la classe un certain nombre de fonctionnalités permettant de manipuler, tester et modifier les objets de type chaîne. Implémentez les différentes opérations présentées ci-dessous :

- `ch1.compareWith (ch2)` : comparaison de deux objets chaînes (ou compatibles), =0 si égalité, > 0 si ch1 $>$ ch2, < 0 si ch1 $<$ ch2
- `ch2.concat (ch1)` concaténation de deux objets
- `ch1.carac(index)` retourne le caractère situé à l'index indiqué

Définitions de méthodes

- `sous_chaine(car deb, car fin)` : extrait la sous-chaîne commençant par le caractère deb et se terminant par fin
- `sous_chaine(int ind1, int ind2)` : extrait la sous-chaîne commençant par le position ind1 et se terminant par ind2

Vous pouvez là aussi définir d'autres méthodes.... 3.Compte rendu Expliquez vos choix d'implémentation et la conception des méthodes. Ce TP sera décomposé en trois modules :

- `chaine.h`

— `chaine.cpp`
— `main.cpp` qui correspond à un ensemble de tests permettant de valider l'ensemble des fonctionnalités de votre classe chaîne.

Remarque : La classe string existe.. . Réalisez maintenant votre application (le programme principal) en utilisant cette classe C++ ANSI.

Chapitre 6

Héritage simple

Objectifs

- Présenter le concept et l'intérêt de l'héritage
- Mécanisme de la sur-définition des méthodes
- Membres dans une classe dérivée
- Analyser la constuction et la destruction des objets dérivés
- Allocation dynamique et gestion des membres alloués dynamiquement
- Construction et destruction en profondeur

6.1 Résumé

6.1.1 Héritage simple

L'héritage est une relation sémantique qui exprime que les instances d'une classe font aussi parti d'une autre classe (plus générique). On peut parler de

- **Spécialisation** : en descendant, elle consiste à étendre les propriétés de la classe (permet l'extension du modèle par réutilisation) ou introduire des nouvelles contraintes.
- **Généralisation** : en ascendant, elle regroupe les particularités communes d'un ensemble d'objets, issus de classes différentes

L'héritage permet la classification des objets. Une bonne classification est stable et extensible : ne classifiez pas les objets selon des critères instables. Seules les classifications (regroupements) vraiment pertinantes et existantes peuvent donner des classes stables. Le **principe de substitution** permet de déterminer si une relation d'héritage est bien employée : *"Il doit être possible de substituer n'importe quelle instance d'une super-classe, par n'importe quelle instance d'une de ses sous-classes, sans que la sémantique d'un programme écrit dans les termes de la super-classe n'en soit affectée."*

6.1.2 Contrôle d'accès

En C++, l'héritage peut être public, protégé ou privé.

```
class B {
      int priv;
```

```
protected :
    int prot;
public :
     int pub;
};

class D1 : public B {
...
};

class D2 : protected B {
...
};

class D3 : private B {
...
};
```

Rappelons ici, qu'un utilisateur de la classe B ne peut acceder qu'aux membres `public`.

Une classe qui hérite de B par `public` peut accéder aux membres `public` et `protected` mais pas aux membres `private` (ces derniers sont hérités mais ne sont pas accessibles).

De plus le type d'héritage contrôle les possibilités des classes dérivées des classes dérivées.

Dans `D1`, les parties publiques et protégées de `B` restent publiques et protégées.

Dans `D2`, les parties publiques et protégées de `B` sont protégées.

Dans `D3`, les parties publiques et protégées de B sont privées (une classe dérivée de D3 ne peut pas les accéder).

6.1.3 Utilisation des classes dérivées

On peut utiliser un objet d'une classe dérivée partout où un objet de sa classe mère peut l'être (substitution).

On peut affecter un objet d'une classe dérivée à un objet d'une classe mère (conversion statique d'objets). La partie de l'objet qui n'est pas présente dans la classe mère est perdue. En revanche, l'inverse est strictement interdit. En effet, les données de la classe dérivée qui n'existent pas dans la classe mère ne peuvent recevoir de valeur.

Les pointeurs des classes dérivées sont compatibles avec les pointeurs des classes mères. Il est donc possible d'affecter un pointeur de classe dérivée à un pointeur de sa classe mère.

6.1.4 Sur-définition des méthodes

Quand on sur-définit une méthode de la classe mère, on garde la signature (nom et paramètres) de la méthode.

Une classe dérivée peut avoir trois actions possibles concernant une méthode `m()` de la classe mère `B`, comme l'exemple le montre :

```
class B {
```

```
 public :
      void m()
      { cout << "m de B\n"; }
      void m(int i)
      { cout << "m de B avec int \n"; }
 };

 class C1 : public B {
 ...
                                    // pas de surdefinition de m()
 };

 class C2 : public B {
 ...
     void m()                       // surdefinition complete
      { cout << "m de C2\n"; }
};

 class C3 : public B {
 ...
     void m()                       // surdefinition avec extension
      { cout << "m de C2\n";
        B::m(); }
};
```

Attention : la sur-définition d'une méthode `B::m()` de la classe mère dans la classe dérivée par `C2::m()` peut masquer une méthode surchargée `B::m(int)`. Soit un objet `C2 obj;` L'appel `obj.m(1)` provoque une erreur, il faut appeller la méthode héritée de la classe de base par `obj.B::m(1)`.

6.1.5 Polymorphisme

Dans les modèles orientés objet, un arbre d'héritage donne la possibilité du polymorphisme : les objets des classes dérivées différentes peuvent répondre à des messages identiques (à des messages qui peuvent être posés dans la classe mère).

Par défaut, en C++, une méthode n'est pas polymorphe. Afin de la rendre polymorphe, il faut :

— déclarer la méthode `virtual` à partir de la classe mère

— la redéfinir en utilisant la même signature dans les classes dérivées.

Exemple pour illustrer :

```
class B {
...
virtual void afficher()
{cout << "B"; } // methode virtuelle (a redefinir dans les classes)
};
```

```
class D1 : public B {
...
virtual void afficher()
{cout << "D1"; } // methode virtuelle de la classe D1
};

class D2 : public B {
...
virtual void afficher()
{cout << "D2";
B::afficher(); } // methode virtuelle de la classe D2
};
```

Pour profiter du polymorphisme, on appelle les méthodes communes à partir des pointeurs (ou des références) :

```
B *  pt[15]:            // tableau de pointeurs (de la classe mere)
pt[0] = new D1();       // allocation des objets heterogenes
pt[1] = new D2();
pt[2] = new B();
...
for (int i = 0; i < 15; i++)
   pt[i]->afficher();   // utilisation homogene
```

6.1.6 Appel statique et dynamique des méthodes

On utilise un attachement statique d'une méthode quand on l'évoque à partir d'un objet :

```
B   b:                  // objet
...
b.afficher();           // attachement statique
```

On utilise un attachement dynamique grâce au mécanisme des méthodes virtuelles et à partir d'un pointeur :

```
B *  pt = new D1();     // pointeur
...
pt->afficher();         // attachement dynamique
```

Cet attachement dynamique peut être atteint à partir de l'objet implicite (`*this`) :

```
class B {
...
virtual void afficher()
 {cout << "B"; } // methode virtuelle (a redefinir dans les classes)
void calculer() // methode stable heritee partout
 { truc = truc2 + truc3;
```

```
   afficher();} // elle utilise la methode virtuelle
};

class D1 : public B {
...
virtual void afficher()
{cout << "D1"; } // methode virtuelle de la classe D1
};

int main() {
  B * pt = new D1();  // pointeur
  D1  d ();           // ou objet
  pt->calculer();
  d.calculer();       // avec affichage correcte
}
```

6.1.7 Construction et destruction

Supposons que D hérite de B (un objet de type D est aussi un objet de B). Pour construire un tel objet (D) il faut savoir comment créer B. Une partie de l'objet D doit suivre les règles de la construction de B. Techniqement parlant : pour construire D, il faut aussi appliquer un constructeur de B.

Remarque : une particularité du C++ impose que toute classe susceptible d'être polymorphe ait également un destructeur virtuel.

Soit :

```
class A {
A(){...}
~A(){...} // destruction de la partie A
... };

class B : public A {
B() { ...} // un constructeur de A est aussi appele
~B(){...} // destruction de la partie B
... };

int main() {
A * b = new B();  // polymorphisme !
...
delete b;  // ne detruit pas la partie B (le destructeur n'est pas virtuel)
}
```

6.2 Questions

Question 6.1 *La classe* `personne` *et la classe* `fromage` *contiennent l'attribut* `departement`. *Peut-on les regrouper par une classe mère commune ?*

Question 6.2 *La classe* `A` *contient les parties suivantes :*

```
class A {
 int I;
protected:
 float X;
public:
 char C;
...};
```

Quelles sont les parties heritées dans `class B : public A {...}` *?*

Question 6.3 *Quelles sont les parties accessibles dans* `class C : protected A {...}` *et dans* `class D : private C {...}` *?*

Question 6.4 *La classe mere* `class A` *possède une méthode* `calcul()` *. La classe dérivée* `class B : public` *surcharge cette méthode. Comment appeler la méthode* `calcul()` *de la classe mère dans la même méthode de la classe dérivée ?*

Question 6.5 *Comment assurer le polymorphisme des méthodes en C++ ?*

Question 6.6 *Que veulent dire attachement statique et dynamique ?*

Question 6.7 *Pourquoi un destructeur virtuel est utile ?*

6.3 Exercice (TP)

On veut manipuler les matrices d'entiers mais aussi des vecteurs colonnes et des vecteurs lignes. Le programme principal peut être considéré comme le cahier des charges de ces classes.

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

#include "matrice.h"
#include "vect_ligne.h"
#include "vect_colonne.h"

int main()
{
const unsigned dim=4;
matrice  M(dim,dim);
matrice  M2(dim,dim);

cin >> M;  // Donner la matrice de l'entree std
        cout << M;
        M2[0][0] = 1;
        M2[1][1] = 1;
        M2[2][2] = 1;
```

```
        M2[3][3] = 1;
        cout << M2;
        M = M * M2;
        cout << M;
M2 = 2*M2;
        cout << M2;

vect_ligne V(dim);
V[0] = 1;
V[1] = 2;
V[2] = 3;
V[3] = 4;
        cout << V;
V = V + V;
        cout << V;
V = 2 * V;
        cout << V;

vect_colonne V2(dim);
V2[1] = 3;
V2[2] = 1;
cout << V2;

cout << V;
cout << M2*V2;
V =   M2*V2;
cout << V;
}
```

Ecrivez et testez les classes.

Chapitre 7

Approfondir l'héritage

Objectifs

— Introduire les classes abstraites
— Présenter le concept et l'intérêt de l'héritage multiple
— Analyser les conflits possibles
— Donner des éléments pour résoudre les éventuels conflits d'un héritage multiple

7.1 Résumé

7.1.1 Classes abstraites

Souvent un concept générique (une classe mère) regroupe des concepts (des classes) différents tels que cette classe de base n'est pas directement instanciable (seules les classes dérivées peuvent avoir des instances). Par exemple, le concept `vehicule` regroupe des `voitures`, des `velos` mais aussi des `bateaux`, etc. Dans ce genre de concept générique, i l y a des "choses" et des méthodes qui ne sont pas connues : elles peuvent et doivent être précisées dans les classes dérivées. Par exemple, les vehicules possèdent une certaine "locomotion". Une voiture a un moteur, un vélo avance grâce à l'homme, etc. mais comment une véhicule avance-t-elle ? Un autre point : il existe des vélos, des bâteaux mais qui peut envisager un "véhicule" ? La classe mère `vehicule` est abstraite, non instanciable qui prévoit des concepts et des méthodes inconnus à son niveau mais existants et connus dans les classes dérivées. La figure illustre l'arbre d'héritage des véhicules.

En C++, une classe est abstraite, si elle possède au moins une méthode abstraite. Une méthode abstraite correspond à une méthode non définie, virtuelle pure. Elle est indiquée par le mot `virtual` et par `= 0 ;` au lieu de son code.

```
   class Vehicule {               // abstraite
   ...
   virtual void avancer() = 0;    // virtuelle pure
};
   class Bateau: public Vehicule {
   ...
   virtual void avancer()
   { methode qui fait avancer...}
```

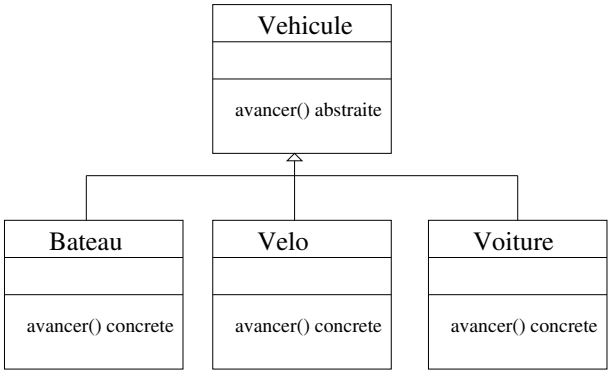

FIGURE 7.1 – Arbre d'héritage des véhicules : la classe mère est abstraite

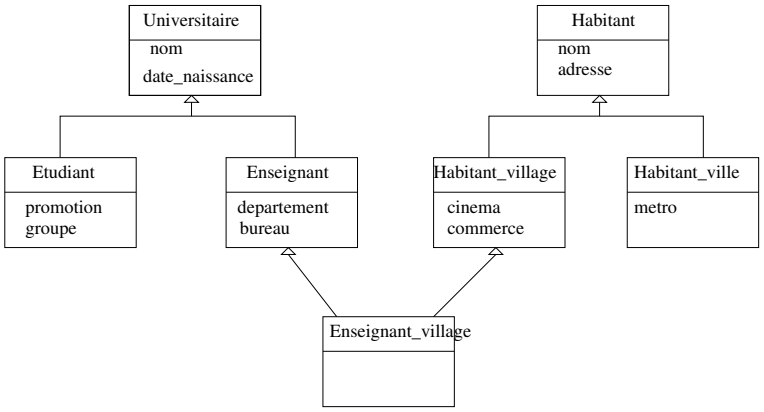

FIGURE 7.2 – Exemple de l'héritage multiple

```
};
```

7.1.2 Héritage multiple

Une classe peut hériter les propriétes de plusieurs classes mères. L'exemple classique de la présence de l'héritage multiplie est la cas où i l ya plusieurs classifications dans l'application et certaines classes héritent des sous-classes différentes (la figure illustre un tel cas).

7.1.3 Conflits d'un héritage multiple à partir d'une classe mère commune

L'héritage multiple peut générer des problèmes et des conflits (en C++). Notamment, une classe dérivée peut avoir des membres (attributs et méthodes) homonymes hérités de plusieurs classes mères. C'est le cas des attributs `nom` dans la classe `Enseignant_village` de l'exemple. Implicitement ce problème se pose aussi quand il s'agit de l'héritage multiple d'une classe mère commune via plusieurs classes intermédiaires (on parle souvent de l'héritage à répétition ou de l'héritage à losange). Ce genre d'héritage peut être envisagé quand les classes `Universitaire` et `Habitant` ont une classe mère commune avec l'attribut `nom`.

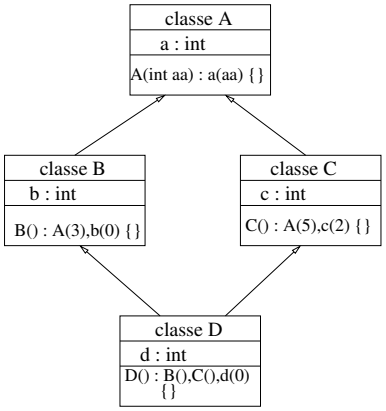

FIGURE 7.3 – Exemple de l'héritage à répétition

7.1.4 Eléments de la solution

Attributs

Supposons que les classes mères `A` et `B` d'une classe `C` possèdent un attribut avec le même nom : `int num;` Pour distinguer les deux attributs hérités, on spécifie l'origine de l'attribut utilisé en metant en préfixe le nom de la classe mère : comme `A::num` ou `B::num`.

Méthodes

Cas des homonymes simples

En cas de conflits, la précision de la méthode souhaitée en utilisant les préfixes est aussi une solution pour les appels de méthodes.

Cas de l'héritage à répétition

Dans ce cas, une même classe mère est héritée sur deux voix différentes (héritage en "losange"). La figure ci-après illustre l'héritage à répétition. Le problème peut être présenté en analysant le fonctionnement des constructeurs.

L'attribut `a` de la classe `A` est présent dans les classes `B` et `A` et deux fois dans la classe `D`. Une occurrence est héritée de la classe `B`, une autre est héritée de la classe `C`.

Les constructeurs des classes `B` et `C` construisent des objets contenant (`a` et `b`) ou bien (`a` et `c`) respectivement. Quant au constructeur de la classe `D`, il doit appeler un constructeur de `B` et un constructeur de `C` qui peuvent être en contradiction concernant la construction de la partie héritée de `A`.

L'héritage virtuel est proposé pour régler le problème de l'héritage à répétition. Cet héritage est construit grâce au mot clef `virtual` dans la description de l'héritage. Sur notre exemple, l'héritage virtuel peut être utilisé de la façon suivante pour résoudre le problème :

```
class A {
  public:
    A (int aa) : a(aa)
    {}
  virtual void m()= 0;  // une methode abstraite...
```

```
  private:
    int a;
};

class B : virtual public  A {
  public:
    // Le constructeur de B appelle celui de A
    // afin d'initialiser l'attribut a
    // present dans la classe B
    B() : A(3), b(8)
    {}
   void m() { ... activite de m() dans B ...
              cout << "cou-cou" << \endl;}
  private:
    int b;
};

class C : virtual public {
  public:
    // Le constructeur de C appele celui de A
    // afin d'initialiser aussi l'attribut a
    // present dans la classe C
    C() : A(5), c(0)
    {}
   void m() { ... activite de m() dans C ...
              cout << "cou-cou" << \endl;}  //
  private:
    int c;
};

// Afin de gerer l'heritage a repetition, on introduit egalement A dans la liste
// des super classes de la classe D
class D : virtual public A, public B, public C {
  public:
    D() : A(2), B(3), C(4)
    {}
   void m() { B::m();
               C::m();
              ... activite de m() dans D ...
              cout << "cou-cou" << \endl;}
  private:
    int d;
};
```

On peut noter un autre problème avec l'appel des méthodes des différentes classes mères. Dans l'exemple précédant, les méthodes `m()` des classes dérivées de la classe abstraite doivent

effectuer des taĉhes différentes mais elles doivent toujours se terminer avec le message "cou-cou" sur la sortie standard. On peut facilement voir que dans la solution indiquée, la méthode `m()` de la classe `D` donne trois messages sur la sortie standard...

Pour généraliser et structurer les méthodes virtuelles (ou sur-définies), on peut toujours les écrire de la façon suivante (proposition de Coplien) :

1. Une séquence d'instructions propre à la classe courante peut être écrite dans une méthode séparée nommée " PRE "
2. Le code partagé par la classe et par les classes mères
3. Une séquence d'instructions peut être écrite dans une méthode séparée nommée " POST"

On pourra alors écrire ainsi :

```
class A {
...
 void m(...)
  {
  PRE();
  base();
  POST();
  }
  virtual void PRE()= 0;  // une methode abstraite...
  virtual void base()= 0; // une methode abstraite...
  virtual void POST()= 0; // une methode abstraite...
};

class B : virtual public  A {
...
 void PRE(){actions sp\'ecifiques}
 void base(...)
  {
  // fonction de base pour la classe B
  }
 void POST(){actions sp\'ecifiques}
};

class D : virtual public A, public B, public C {
...
  void PRE(){actions sp\'ecifiques}
 void base(...)
  {
  B::base();
  C::base();
  // fonction de base pour la classe D
  }
 void POST(){actions sp\'ecifiques}
};
```

7.1.5 Héritage ou délégation ?

Notons ici, que - pour éviter l'héritage multiple - certaines fonctionnalités et propriétés peuvent être délégées à un composant de la classe (au lieu de les hériter d'une classe mère). Naturellement, on ne peut pas parler du polymorphisme dans ces cas.

7.2 Questions

Question 7.1 *Quand devient une classe abstraite ?*

Question 7.2 *Comment instancier une classe abstraite ?*

Question 7.3 *Quand devient une classe abstraite ?*

Question 7.4 *En quoi consiste le conflit des attributs dans le cas de l'héritage multiple ?*

Question 7.5 *En quoi consiste le conflit des méthodes dans le cas de l'héritage multiple ?*

Question 7.6 *Quel est l'intérêt de l'héritage virtuel ?*

Question 7.7 *Comment éviter l'héritage multiple par une délégation ?*

7.3 Exercice (TP)

Classes "figures"

Le but de ce TP est de modéliser et de réaliser un ensemble de classes simples permettant de manipuler dans un plan (dans une fenêtre graphique ou en mode console) les figures géométriques suivantes : point, segment, rectangle, cercle, carré. Ces figures devront notamment pouvoir être dessinées, effacées et déplacées dans une fenêtre d'affichage (ou les dessins doivent être suivis par affichage sur le console). Chaque figure possède une position (denÂnée par les coordonnées x et y) sur le plan (la position n'est pas une occurence de la classe Point, elle ne se dessine pas, ...). Pour faciliter le déplacement des figures, les autres attributs des différentes fiÂgures réfèrent à cette position.

Une attention particulière sera portée à la création de cette famille de figures et aux méthodes associées en utilisant notamment les notions d'héritage et de polymorphisme pour optimiser le code (trouver la bonne place des attributs et des méthodes). Modélisez les différentes figures et déduisez en l'arbre d'héritage souhaitable. Préciser les membres (attributs, méthodes) et leurs propriétés (abstrait, virtuel, public, privé, statique, etc.).

Chapitre 8

Conception d'applications

Objectifs

— Distinguer les aspects statiques et dynamiques de la conception
— Conception statique : trouver les classes pertinentes
— Conception dynamique : trouver la coopération des classes
— Donner la décomposition algorithmique

8.1 Résumé

8.1.1 Modèles de base de la conception

Dans des cas réels, on ne commence pas la réalisation d'un projet par la programmation des classes. Le codage doit être précédé par une conception soigneuse qui permet de découvrir les besoins de l'application future mais aussi de prévoir le comportement du logiciel (réactions aux stimulii, résultats attendus, temps d'exécution, classes qui réalisent les fonctions, conception de la coopération des classes, etc.) Dans le cycle de vie habituel, on prévoit les phases de la rédaction du cahier des charges, de l'analyse du domain si nécessaire, de la spécification fonctionnelle, de l'analyse organique avant de faire la conception des classes et le codage. Pour analyser et modéliser, le plus souvent, on utilise un langage de modélisation orienté objet comme UML.

Pour voir ce qu'un système existant fait ou ce qu'un système futur va faire, on a besoin d'analyser et de modéliser plusieurs vues du système :

— les concepts, les classes qui interviennent et les relations pertinentes entre eux
— le comportement des objets, les coopérations entre eux
— le déroulement des solutions algorithmiques
— les décisions sur l'architecture, sur le déploiement,...

8.1.2 Modèle statique : les classes et les relations entre elles

Le modèle de base utilisé et le diagramme de classes (englobé éventuellement par des diagrammes de paquetage. Exemple :

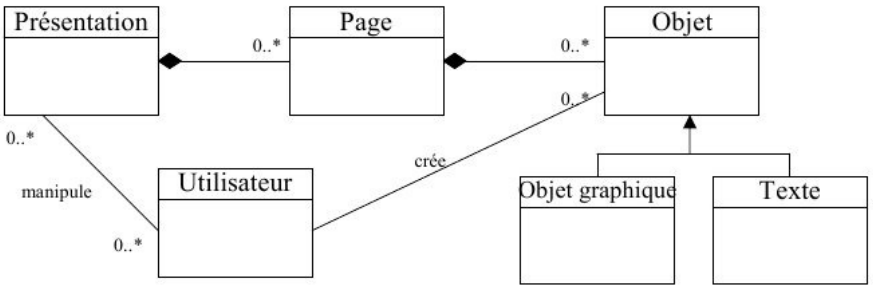

FIGURE 8.1 – Diagramme de classes des présentations sur ordinateur

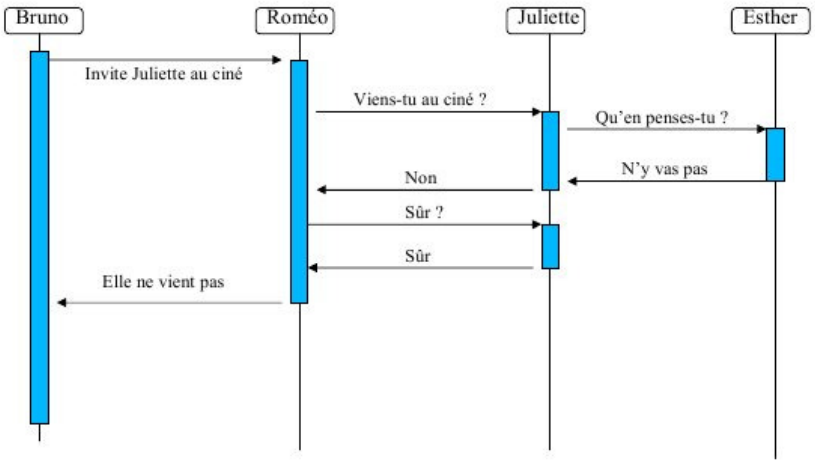

FIGURE 8.2 – Diagramme de séquences pour organiser la soirée

8.1.3 Modèle dynamique : coopération des classes grâce aux appels de méthodes

Le diagramme de classes ne donne pas d'information sur les interactions entre les classes : les relations sont des relations statiques ou encore, elles modélisent la possibilité des accès à des membres des classes. Pour modéliser l'aspect dynamique, UML propose deux types de diagrammes :

— diagramme d'état : ce diagramme explique les états possibles d'un objet avec les activités dans les états et les transitions autorisées grâce à des stimulii y compris les actions de l'objet
— diagramme de coopérations (comme le diagramme de séquences et le diagramme de collaborations

Exemple :

8.1.4 Utilisation de l'héritage

Comme nous avons vu, l'héritage peut grandement améliorer la structuration des classes et des codes. mais l'héritage ne peut être appliquer que dans les cas qui correspondent au règle de substitution.

8.2 Questions

Question 8.1 *Quel est l'intérêt du modèle statique d'un système ?*

Question 8.2 *Peut-on faire la conception complète d'un système par des diagrammes soigneux de classes ?*

Question 8.3 *Quel est l'intérêt du modèle dynamique d'un système ?*

Question 8.4 *Pourquoi créer des modèles fonctionnels d'un système ?*

Question 8.5 *Quel est le problème de l'analyse suivante ?*
On aimerai réaliser une classe `Ensemble`*. Pour cela, on peut utiliser la classe existante* `Liste`*. Les éléments de l'ensemble seront stockés dans une liste. Alors, l'ensemble est une liste. Techniquement parlant : la classe* `Ensemble` *hérite de la classe* `Liste`*.*

8.3 Exercice (TP)

On veut concevoir l'application donnée ci-dessous.

8.3.1 Cahier des charges

Soit un programme qui simule les déplacements de mobiles de deux types différents (ex. : une fusée et des projectiles). Ces deux types de mobiles vont pouvoir se déplacer dans un cadre en rebonÂdissant sur des parois ainsi que sur le cadre. Pour ce faire, ils possèdent une direction donnée et sont animés d'une vitesse constante ; leurs rebonds "parfaits" n'engendrent pas de perte de vitesse. Le cadre ainsi que la fusée ne s'abîment pas (la fusée rebondit sur les parois sans problème, mais elle est détruite quand elle rencontre un projectile). Les parois et les projectiles sont plus fragiles aux rebondissements. Chacun d'eux possède une durée de vie (admettons que les projectiles peuvent subir 50 rebondissements et que les parois peuvent en subir 600). Au milieu de l'espace du jeu, un canon est posé. Pour alimenter le jeu, le canon envoie des projectiles régulièrement (par exemple, toutes les 100 millisecondes). Pour avoir un jeu plus varié, il est intéressant d'avoir un canon qui change légèrement la direction selon laquelle il lance le projectile suivant. Le canon est aussi une paroi dans le sens où les mobiles rebondissent dessus en le touchant. Le canon a une vie illimitée. La simulation prend fin quand un projectile touche la fusée.

8.3.2 Conception des classes et des coopérations

Créer le diagramme de classes pertinentes. Pour faire la conception du jeu, envisager les algorithmes et les coopérations nécessaires sous forme de diagrammes d'activités et de séquences.

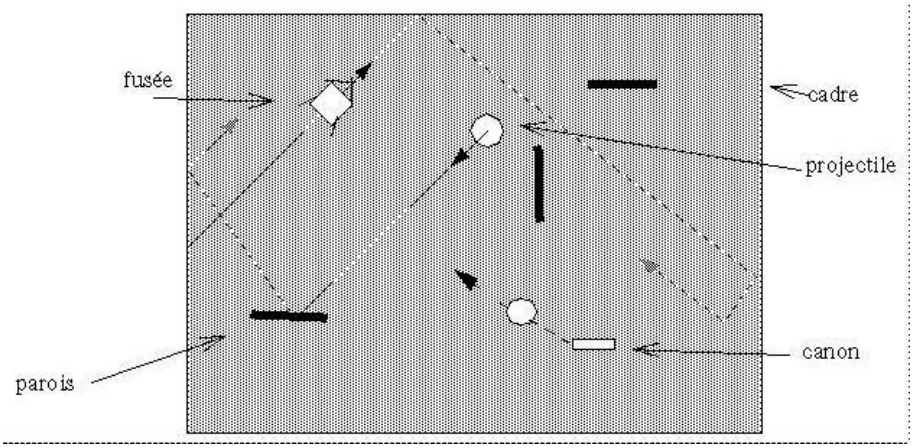

FIGURE 8.3 – Vue du jeu "fusée et projectiles"

Complétez les différentes classes avec les méthodes nécessaires. Pour visualiser les objets (mobiles et immobiles), envisagez l'utilisation des classes figures vues précédemment. (Question importante : un projectile est-il un cercle ?)

Chapitre 9

Exceptions

Objectifs

— Présenter l'intérêt des exceptions
— Quand et comment lancer une exception en C++ ?
— Comment capter des exceptions en C++ ?
— Donner la décomposition algorithmique

9.1 Résumé

Certains événements non prévus pour le déroulement normal d'un programme peuvent se produire :

— limitations du système dépassées (mémoire, capacité du disque, ...)
— valeurs non valides (saisies par l'utilisateurs, transmises par une autre méthodes, ...)
— mauvais fonctionnement d'une périphériques (pas de papier dans l'imprimante, ...)
— mauvaise utilisation des classes (indices dépassées, ...)

Le programme peut agir des façons différentes. Il peut :

— de ne rien faire
— donner un message d'erreur sur `stderr`, puis continuer l'exécution ou aborter
— mettre à jours des variables globales prévues pour les erreurs
— retourner un code `exit(1)`, ...
— utiliser une **exception**

Les problèmes découverts à l'exécution (les erreurs) sont souvent indiqués à l'aide de codes de retour des fonctions. Souvent, une fonction renvoie un code spécifique (par exemple : 0) à l'issue de son exécution normale et une valeur différente (par exemple : 1) si un problème a été détecté, permettant d'indiquer à son utilisateur si elle s'est correctement déroulée ou non. La valeur renvoyée peut être utilisée pour déterminer la nature de l'erreur (coder l'erreur). Cette méthode permet à chaque fonction de gérer les erreurs, et d'effectuer ainsi des traitements d'erreur. Cependant, cette technique nécessite de tester les différents codes de retour de chaque fonction appelée, et le traitement d'erreurs devient très lourde : on peut avoir un grand nombre de tests et beaucoup de cas particuliers à gérer dans les fonctions appelantes. De plus, le code du traitement des erreurs se trouve mélangé avec le code du fonctionnement normal de l'algorithme.

Le mécanisme des exceptions est standardisé et unifie le traitement des erreurs. Le but est de réaliser des traitements spécifiques aux événements "anormaux". Le traitement peut rétablir l'exécution du programme dans son mode de fonctionnement normal. Il se peut aussi que le programme se termine, si aucun traitement n'est prévu par exemple.

9.1.1 Séparation de la création et du traitement des exceptions

L'idée de base du mécanisme des exceptions est la séparation de la détection d'un problème et son traitement. En général, quand un erreur survient au cours de l'exécution d'une méthode, c'est la méthode qui est capable de détecter la nature exacte de ce qui s'est produit.

Par exemple : une classe Fraction est capable de détecter qu'il y a eu une tentative de division par 0, mais elle ne sait pas, pourquoi cette opération a été demandée. Le traitement de l'erreur doit se faire à l'endroit où l'on connait la réponse adéquate.

La gestion d'exceptions sépare deux parties dans le code :

1. le code où l'erreur se produit, est détecté et l'exception est lancée (mot clé `throw`)
2. le code où l'on capte l'exception (mot clé `try`) et où l'on décide son traitement (mot clé `catch`)

9.1.2 Création d'exceptions

Dans la conception d'une classe, on peut prévoir les erreurs qui peuvent se produire ainsi que les informations qu'on peut fournir sur l'erreur.

Exemple :

```
class fract_erreur   // exception  associee aux fractions
{
public:
    int num;        // numerateur
    int den;        // denominateur
    fract_erreur(int n, int d) : num(n), den(d) \\ constructeur
     {}
    ...    // Faut-il un constructeur par copie et un destructeur ?
};
```

Les instances de cette classe peuvent être lancées par les méthodes de la classe `Fraction` par exemple.

```
class Fraction {
    int _num;    // numerateur
    int _den;    // denominateur
 void  reduire(); // methode privee de service
public :
    Fraction( const int n=0, const int d=1 ) : _num(n), _den(d)
     { if (_den == 0)
         throw fract_erreur(_num, _den);   // point 2
     }
     ...
};
```

En C++, il est possible de spécifier les exceptions qu'une fonction peut lancer. Pour cela, il suffit de les indiquer dans son en-tête par le mot clé `throw` suivi les classes des exceptions qu'elle est autorisée à lancer. Exemple :

```
int fonction_ex1(int, Fraction, double ) throw (fract_erreur, int)
{
    ...
}
```

Cette fonction a le droit de lancer des exceptions du type `int` ou `fract_erreur`. Si une autre exception se produit, par exemple une exception du type `double`, une erreur à l'exécution interviendra. La liste des exceptions autorisées dans une fonction ne fait pas partie de sa signature. Elle n'influance pas le surcharge des fonctions.

Les exceptions sont des objets et elles peuvent être classées, avoir des attributs porteux d'informations qui décrivent la nature de l'erreur. Le traitement peut être alors très riche et efficace.

9.1.3 Traitement des exceptions

Dans le code susceptible de poduire des erreurs, des blocs d'instructions peuvent être mis sous surveillance pour attraper les exceptions. La gestion est activée uniquement dans les blocs délimités par

```
try {
     ...
    }
```

Les exceptions produites dans le bloc doivent être captées par les gestionnaires indiqués lavec le mot clé `catch`.

```
try {
     ...
    }
catch(exception1) {
     // traitement1
    }
catch(exception2) {
     // traitement2
    }
    ...
    // reprise
```

Un gestionnaire peut effectuer le traitement d'erreurs. Ce traitement comprend en générale le rétablissement de l'état des objets manipulés par la fonction et/ou la libération des ressources non encapsulées par les objets (par exemple, les fichiers ouverts, les connexions réseau, ...).

Une fois les actions du gestionnaire sont effectuées, celui peut relancer l'exception, afin de permettre un traitement complémentaire par les fonctions appelantes aux niveaux supérieurs (transmission de l'exception). Le parcours de l'exception s'arrêtera donc dès que l'erreur aura été complètement traitée. C++ permet de lancer une exception différente de celle reçue par un gestionnaire. De plus, le traitement de l'erreur peut lui aussi provoquer une erreur.

Pour relancer la même exception par un gestionnaire d'exception, il faut utiliser le mot clé `throw` sans argument :

```
   try {
        ...
       }
       catch(fract_erreur e) {
          ...
          throw ;   // retransmission de l'exception
       }
```

L'exception est alors relancée au niveau supérieur. Les gestionnaires peuvent naturellement modifier les paramètres des exceptions (pour retransmettre l'objet modifié, il faut le attraper avec une référence).

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

int main()
{
   try {
        Fraction F1 (2, 3);         //  la fraction 2/3
        Fraction F2 (4);            //  la fraction 4/1

        Fraction F3(2,0);           // point 1

        F3 = F1 * F2;
        cout << F3 << endl;
        ...
       }
       catch( fract_erreur f) {    // point 3
          F3 = 0;
          cerr << "F3 a ete initialise par un denominateur 0 << endl;
          }
       catch( int e) {
          ...;
          cerr << "une classe a lance le code : " << e << endl;
          }
       cout << apres le traitement de l'erreur, l'execution continue ici";
                                  // point 4
       ...
}
```

L'exécution normale du programme est interrompue dès que l'exception est lancée et le contrôle passe à un gestionnaire d'exception. Quand un gestionnaire d'exception s'exécute, il peut attraper (ou pas) l'exception. Il est possible que la fonction qui a appelée la fonction défaillante considère

elle aussi qu'une erreur a eu lieu et termine son exécution sans corriger l'exécution. L'erreur remonte ainsi la liste des appelants de la fonction qui a généré l'erreur. Ce processus continue, de fonction en fonction, jusqu'à ce que l'erreur soit complètement traitée par une fonction qui la capte ou jusqu'à ce que le programme se termine. Ce cas survient lorsque jusqu'à la fonction principale aucune fonction ne traite l'erreur.

Problème des objets mal construits

Ici, on trouve une remarque sur le traitement des exceptions lancées par un constructeur. Quand une exception est lancée par un constructeur, la construction de l'objet est interrompue. Le destructeur de cet objet (qui fait par exemple la destruction des objets à plusieurs niveaux) ne sera pas appelé et l'objet sera partiellement initialisé et pas détruit. Pour remettre les choses en claire, il est nécessaire de faire un peu de ménage après le lancement de l'exception. Le C++ dispose d'une syntaxe particulière pour les constructeurs des objets susceptibles de lancer des exceptions. Cette syntaxe permet simplement d'utiliser un bloc `try` pour le corps des constructeurs. Les blocs `catch` suivent alors la définition du constructeur, et effectuent la libération des ressources que le constructeur aurait pu allouer avant que l'exception ne se produise.

Le comportement du bloc `catch` des constructeurs est différent de celui des blocs `catch` classiques. Le plus souvent, les exceptions ne sont normalement pas relancées une fois qu'elles ont été traitées. Si l'on veut transmettre l'exception aux niveaux supérieurs, il faut utiliser explicitement la relance par `throw` après son traitement. Dans le cas des constructeurs avec un bloc `try`, l'exception est *systématiquement* relancée. Le bloc `catch` du constructeur ne doit prendre en charge que la destruction des données membres partiellement construites, et il faut toujours capter l'exception au niveau supérieur du programme qui a tenté à créer l'objet. L'exemple suivant (pris de []) illustre le cas :

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

using namespace std;

class A
{
    char * pBuffer;
    int  * pData;

public:
    A() throw (int);

    ~A()
    {
        cout << "A::~A()" << endl;
    }

    static A* operator new(size_t taille)
    {
        cout << "new()" << endl;
```

```
        return malloc(taille);
    }

    static void operator delete(void *p)
    {
        cout << "delete" << endl;
        free(p);
    }
};

// Constructeur susceptible de lancer une exception :
A::A() throw (int)
try
{
    pBuffer = NULL;
    pData = NULL;
    cout << "D\'ebut du constructeur" << endl;
    pBuffer = new char[256];
    cout << "Lancement de l'exception" << endl;
    throw 2;
    // Code inaccessible :
    pData = new int;
}
catch (int)
{
    cout << "Je fais le menage..." << endl;
    delete[] pBuffer;
    delete pData;
}
```

Si un objet de type A est alloué dynamiquement et une exception de type `int` intervient, le destructeur de l'objet n'est pas appellé. Il faut l'exécuter explicitement.

```
int main(void)
{
    try
    {
        A *a = new A(5);
    }
    catch (int e)
    {
        delete a;
    }
    return 0;
}
```

Polymorphisme des exceptions

Comme les exceptions sont des objets, elles peuvent avoir des relations d'héritage entre elles. Ainsi, il est possible de classifier les différents cas d'erreurs en définissant une hiérarchie de classe d'exceptions. Nous savons que les objets des classes dérivées peuvent être considérés comme des instances de leurs classes mère. Un gestionnaire peut être basé sur cette polymorphisme et peut récupérer les exceptions de différentes classes dérivées en récupérant un objet du type d'une de leurs classes mère. Il est alors possible d'écrire des gestionnaires génériques en utilisant des objets d'un certain niveau dans la hiérarchie des exceptions.

La bibliothèque standard C++ définit elle-même un certain nombre d'exceptions standard, qui peuvent être utilisées telles quelles ou servir de classes de base à des classes d'exceptions personnalisées (cf. une liste des exceptions standardisées dans []). Elles sont déclarées dans le fichier en-tête `stdexcept`, classées en deux grandes catégories et dérivent de la classe de base `exception`. Exemples :

La première catégorie regroupe les exceptions dont l'apparition vient d'une erreur de programmation et elles dérivent de la classe d'exception `logic_error` :

```
class logic_error : public exception
{
public:
    logic_error(const string &what_arg); // constructeur
};
```

L'argument du constructeur permet de définir la chaîne de caractères qui sera renvoyée par la méthode virtuelle `const string & what()`des instances de `exception`.

Les classes d'exception qui dérivent de la classe `logic_error` sont : `domain_error, invalid_argument, ler`

La deuxième catégorie d'exceptions caractérise les erreurs d'exécution. Elles dérivent de la classe d'exception `runtime_error` :

```
class runtime_error : public exception
{
public:
    runtime_error(const string &what_arg);
};
```

Les classes dérivées prédéfinies sont les suivantes : `range_error, overflow_error, underflow_error`.

9.1.4 Problème de la couverture des exceptions

Quand un gestionnaire d'exception s'exécute, il peut attraper (ou pas) l'exception.

En C++, la gestion des exceptions garantit que tous les objets de classe de stockage automatique sont détruits lorsque l'exception qui remonte sort de leur portée. Ainsi, si toutes les ressources sont encapsulées dans des classes disposant d'un destructeur capable de les détruire ou de les ramener dans un état cohérent, la remontée des exceptions effectue automatiquement le ménage (cf. []).

Mais que se passe-t-il quand une erreur non prévue se produit ?

Si une exception se produit dans un bloc `try` et il n'y a pas de bloc `catch` correspondant à cette exception, une erreur d'exécution sera produite. La fonction `std::terminate` prévue pour terminer l'exécution est appelée.

Si une erreur et/ou une exception se produit dans une zone non protégée, alors l'erreur est remontée (si pas de gestionnaire, jusqu'au programme principal) et le programme aborte. Le C++ ne vérifie ni la complétude des erreurs qui peuvent se produire ni la complétude des traitements. (En Java, tous les deux sont nécessaires : il faut déclarer toutes les exceptions lancées et les exceptions traitées. Si une exception n'est pas traitée : il y a une erreur de compilation.)

9.2 Questions

Question 9.1 *Quel est le problème avec la technique de détection d'erreurs qui retourne des codes entiers ? Comment distingueur les erreurs des sources (des objets) différentes ?*

Question 9.2 *Comment incorporer les codes dans les exceptions ?*

Question 9.3 *Comment lancer une exception ?*

Question 9.4 *Que se passe-t-il si l'exception lancée n'est pas captée ?*

Question 9.5 *Comment capter une exception ?*

Question 9.6 *Comment capter une exception à plusieurs niveaux ?*

Question 9.7 *Comment gérer les familles d'exceptions polymorphes ?*

Question 9.8 *Comment utiliser les exceptions prédéfinies ?*

Question 9.9 *Peut-on être sur en C++ que les exceptions lancées seront aussi traitées ?*

9.3 Exercices (TP)

9.3.1 Classe Fraction

- Complétez votre classe `Fraction` avec les exceptions qu'on peut prévoir
  - construction avec un dénominateur 0
  - division par 0 au cours des opérateurs
  - débordements inférieurs (underflow) et supérieurs (overflow) (pour cela, choisissez une implémentation des fraction avec un couple de `short int`...)
- Ecrivez un programme (et des fonctions) qui manipulent les fractions. Testez les cas où les exceptions lancées sont captées mais aussi les cas contraires.

9.3.2 Classe Chaine

- Complétez votre classe `Chaine` qui manipule les chaînes de caractères avec les exceptions qu'on peut prévoir
  - dépassement des limites de stockage (utilisation d'une indice trop grande ou trop petite)
- Ecrivez un programme (et des fonctions) qui manipulent les chaînes. Testez les cas où les exceptions lancées sont captées mais aussi les cas contraires.

Chapitre 10

Programmation générique : patrons

Objectifs

— Présenter l'intérêt des patrons
— Montrer le mécanisme et le syntaxe pour écrire des solutions génériques
— Discuter des classes patrons et des fonctions patrons
— Montrer les limites de l'utilisation des patrons

10.1 Résumé

10.1.1 Intérêt des patrons

Souvent, dans des classes différentes, on manipule des objets différents mais avec des opérations similaires. Par exemple : on peut manipuler des listes d'entiers mais aussi des personnes. On peut alors définir des classes mères assez génériques qui ne spécifient pas le type des objets (on peut stocker par exemple des pointeurs génériques, créer une classe liste de pointeurs génériques ...). Le C++ permet de résoudre ces problèmes d'une façon plus élégante grâce aux patrons génériques, paramétrés que l'on appelle encore classes et fonctions `template`.

Dans la définition des patrons, certains types et valeurs ne sont pas fixés mais symbolisés par des paramètres. Un paramètre template est alors soit un type générique, soit une constante d'un type connu. Les paramètres template permettent de formuler le fonctionnement des instances grâce à des paramètres génériques. Les fonctions et les classes ainsi paramétrées sont appelées respectivement fonctions template et classes template. Elles ne peuvent pas être instanciées directement (la signification des paramètres est inconnue). Elles peuvent être considérées comme des patrons qui peuvent être utiliser pour définir des classes et des fonctions concrètes. Elles correspondent alors à un ensemble de classes et de fonctions...

10.1.2 Création d'une classe patron paramétrée

Les classes template sont des classes qui contiennent des membres dont certains types sont génériques. Un exemple simple peut être donné par la classe `Tableau` générique :

```
#ifndef TABLEAU_T_H
#define  TABLEAU_T_H
```

```
template<class T> class Tableau {
 protected:
    T * _pt;          // pointeur sur le tableau
    int _nb;          // nombre d'elements
 public:
     Tableau():_pt(NULL), _nb(0)
      {}              //initialisation a vide
     Tableau(const T * pt, const int nb):_pt(NULL), _nb(nb)
      {               //a partir d'un tableau classique
       _pt = new T[_nb];
       for (int i = 0; i < _nb; i++)
          _pt[i] = pt[i];
      }
     Tableau(const int nb, const int v):_pt(NULL), _nb(nb)
      {               //un tableau initialise de nb elements
       _pt = new T[_nb];
       for (int i = 0; i < _nb; i++)
          _pt[i] = NULL;
      }
     Tableau(const Tableau<T>& TB):_pt(NULL), _nb(TB._nb)
      {               //a partir d'un autre tableau
       _pt = new T[_nb];
       for (int i = 0; i < _nb; i++)
          _pt[i] = TB._pt[i];
      }
      ...
      Tableau<T> operator+(const Tableau<T>  & OP) const;
  const T& min() const;  // retourne le plus petit element
      ...
  };  // fin des methodes in-line...

template<class T>
Tableau<T> Tableau<T>::operator+ (const Tableau<T>  & OP) //concatenation
     { ...
     }
```

Cette classe patron permet la création de multiples classes : tableau d'entiers, tableau de floats, tableau de personnes, etc.

Pour créer des objets de ces classes, il suffit de

— présenter le patron au compilateur pour pouvoir créer les classes concrètes

— définir et utiliser les objets et les fonctions concrets

Exemple :

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

```
#include "Personne.h"

int main() {
    // definition d'un tableau d'entiers
    Tableau<int> TE(100);
    for (int i = 0; i < 99; i++)
          TE[i] = i;
// definition d'un tableau de personnes
Tableau<Personne> TP(50);
    for (int i = 0; i < 49; i++)
          cin >> TP[i];
   ...
}
```

Question 10.1 *Dans cet exemple, combien de classes sont générées selon le patron donné ?*

10.1.3 Types des paramètres, classes avec multiples paramètres

Un paramètre template peut être

- soit un type générique(une classe) : `template<class T> Tableau`
- soit une constante d'un type défini : `template<int n> Tableau`
- soit une classe template : `template<template<class T> class Liste> Tableau`. Dans ce cas, les éléments du tableau seront des objets de l'instanciation de `Liste<T>`. Si l'on définit le patron `Liste<T>`, on peut créer des objets de type `Tableau<Liste<int>>`, ce qui donne des tableaux dans lesquels les éléments sont des listes d'entiers...
- soit un objet template (une fonction, un objet-fonction template ce qu'on va voir plus loins).

Un patron peut être basé sur plusieurs paramètres template, toutes les combinaisons sont possibles.

Par exemple, pour définir le patron d'une table de hachage, on va créer :

```
#ifndef T_HASHTABLE
#define T_HASHTABLE

template <class C, class T, class Comparateur, class Hash> class HashTable{
```

Les paramètres génériques sont les suivants :

- C : clé de la table (le hachage s'applique sur cette clé)
- T : une valeur de type T est associée à chaque clé
- Comparateur : un objet-fonction qui permet de comparer deux clés
- Hash : une fonction de hachage, sous forme d'un objet-fonction.

Dans la définition d'un patron, les types génériques peuvent être utilisés exactement comme s'il s'agissait de types habituels. Les constantes template peuvent être utilisées dans le patron comme des constantes locales.

10.1.4 Fonctions patrons

Les fonctions template sont des fonctions qui peuvent utiliser des objets dont le type est un type générique et/ou qui peuvent êtres paramétrées par une constante.

La définition d'une fonction template se fait comme une déclaration de la fonction en utilisant les paramètres template comme s'ils étaient des types normaux. Les variables peuvent être déclarées avec un type générique et les constantes template peuvent être utilisées comme des constantes locales. Les fonctions template s'écrivent donc exactement comme des fonctions classiques. Voici l'exemple d'une fonction template qui retourne l'objet qui est le supérieur entre deux objets de type T générique :

```
 ...
template <class T> T Max(T x, T y) {  // le patron
    if (x>y) return x ;
        else return y;
}

int main() {
    int i = 5;
    int j = 7;
    int k = Max(i, j); // l'instanciation
   ...
}
```

Comme les fonctions classiques, les fonctions template peuvent aussi être surchargées par des fonctions classiques ou par d'autres fonctions template. S'il y a une ambiguïté entre une fonction template et une fonction classique qui la surcharge (elles portent une signature commune), la fonction classique sera utilisée.

Une fonction template (ou une classe template) peut être déclarée amie d'une classe, template ou non. Naturellement, la fonction amie template (son instance générée) va avoir accès sur toutes les données de la classe qui l'a declaré amie.

Exemple :

```
#ifndef TABLEAU_T_H
#define  TABLEAU_T_H


template<class T> class Tableau;
template<class T>  Tableau<T> operator+(const Tableau<T>& T1, const Tableau<T>& T2);  templat

template<class T> class Tableau {
 protected:
    T * _pt;          // pointeur sur le tableau
    int _nb;          // nombre d'elements
 public:
     ...
 friend Tableau<T> operator+ <>( const Tableau<T>& , const Tableau<T>&);
     ...
```

```
 friend bool operator== <>( const Tableau<T>& , const Tableau<T>&);
    ...
};
#endif

                     // le patron de l'operator==
template<template<class T> class Tableau >
  bool  operator==( const Tableau<T>& T1, const Tableau<T>& T2) {
      if (T1._nb != T2._nb)
return false;
      for (int i = 0; i < T1._nb; i++)
         if (T1._pt[i] != T2._pt[i])
  return false;
      return true;
    }
...
```

Pour comprendre les prédéfinitions des fonctions amies template et les signes <> dans les lignes de définition, il faut lire la section sur la spécialisation des patrons.

10.1.5 Ecriture des patrons

Nous avons vu, comment les patrons (fonctions et classes) doivent être distingués par un préfix `template` qui donne la liste des paramètres génériques. Puis, dans le code, les paramètres génériques peuvent être utilisés comme des variables et des constantes.

Les méthodes qui sont écrites dans la classe (entre { et } ;) seront les modèles des méthodes inline. Si les méthodes de la classe template ne sont pas définies dans la déclaration de la classe, elles devront être déclarées template avec un préfix template particulier. Il est absolument nécessaire de spécifier tous les paramètres template pour donner à quelle classe template la méthode template appartient.

Exemple :

```
#ifndef TABLEAU_T_H
#define  TABLEAU_T_H

template<class T> class Tableau {
 protected:
    T * _pt;          // pointeur sur le tableau
    int _nb;          // nombre d'elements
 public:
     ...
 bool contient( const T&); //methode definie apres la classe
     ...
};

template <class T> bool Tableau<T>::contient( const T& e) {
     for (int i = 0; i < _nb; i++)
```

```
        if (_pt[i] == e)
  return true;
      return false;
 }
#endif
```

Quand le compilateur doit instancier un patron, il doit connaître tous les détails du patron pour le transmettree en concret. D'où la règle la plus importante : *il faut mettre le patron partout et entièrement où il est utilisé*. La technique la plus appropriée est de mettre tous les composents d'un patron dans un même fichier .h (Même les méthodes template seront alors dans le fichier .h et on ne fait pas de fichier .cpp pour elles !).

10.1.6 Patrons et compilation

En général, la génération d'une classe concrète se passe en remplaçant les types et les paramètres génériques par des vrais types et valeurs. Cette opération s'appelle l'instanciation des template. Souvent, la définition des objets d'un type qui est à l'issue d'un patron nécessite cette instanciation. Une fois un objet (et aussi sa classe) est créé, on peut l'utiliser selon les membres (données et méthodes) concrétisés. Les types réels à utiliser à la place des types génériques sont déterminés lors de cette première définition ou utilisation par le compilateur, soit implicitement à partir du contexte d'utilisation du template, soit par les paramètres donnés explicitement par le programmeur [].

10.1.7 Spécialisation des patrons

Dans certains cas, la solution générique offerte par les classes et les fonctions template ne convient pas pour certains types des paramètres template. Par exemple,dans la classe `template<class T> Tableau`, nous avons la méthode `T min()` qui retourne le plus petit élément du tableau. La méthode fonctionne pour les types `T` qui définissent la relation d'ordre `<`. Cependant, cette méthode n'est pas intéressante quand on utilise des pointeurs ; par exemple dans l'instance `Tableau<int *>` (supposons que ce ne sont pas les pointeurs qu'on va manipuler mais les objets pointés).

Supposons que l'instanciation concerne un tableau `Tableau<Personne*>`. On aimerais que la méthode `min()` retourne la personne qui a la plus petite capacité pneumoniale (attribut `cap_pn` de la classe). Dans ce cas, il est intéressant de définir une version particulière de la classe template pour ce type et de surcharger l'opérateur `<` dans la classe `Personne`.

Pour palier le problème, ils existent deux types de spécialisation :

- les spécialisations partielles, quand on précise une partie des paramètres template (valeurs et/ou types), mais ils restent des paramètres génériques à instancier
- les spécialisations totales, dans lesquelles il n'y a plus aucun paramètre template (ils ont tous fixés).

Pour revenir à notre problème, la classe patron `template<class T> Tableau` ne convient pas aux pointeurs. Il faut alors créer un patron similaire, dont l'utilisation est plus restrainte et limitée aux pointeurs : `template<class T> Tableau<T*>`.

```
template<class T> class Tableau<T*> {
 protected:
```

```
    T ** _pt;          //  tableau de pointeurs
    int _nb;           // nombre d'elements
 public:
     Tableau():_pt(NULL), _nb(0)
      {}              //initialisation a vide
     Tableau(const T * pt, const int nb):_pt(NULL), _nb(nb)
      {               //a partir d'un tableau classique
       _pt = new T* [_nb];
       for (int i = 0; i < _nb; i++)
          _pt[i] = new T(pt[i]);
      }
    ...
};
```

le problème suivant survient quand on veut utiliser le patron `Tableau` pour manipuler les chaînes classiques C `char *`. Dans ce cas, ce ne sont pas des objets de type `char` qui sont pointés par les éléments du tableau mais des chaînes se terminant par `'\0'`. Le traitement est particulier et on doit encore spécialiser le patron (cela sera une spécialisation totale).

```
template<> class Tableau<char*> {
 protected:
    char ** _pt;       //  tableau de chaines
    int _nb;           // nombre d'elements
 public:
     Tableau():_pt(NULL), _nb(0)
      {}               //initialisation a vide
     Tableau(const char ** pt, const int nb):_pt(NULL), _nb(nb)
      {                //a partir d'un tableau de chaines C
       if (pt == NULL)
 throw invalid_argument("pointeur null ch** ");
       _pt = new char* [_nb];
       for (int i = 0; i < _nb; i++)
         {
         if (pt[i] == NULL)
   throw invalid_argument("pointeur null sur chaine ");
         int m = strlen(pt[i]);
         _pt[i] = new char[m];
         strcpy(_pt[i],pt[i]);
         }
      }
    ...
};
```

10.2 Questions

Question 10.2 *A quoi sert la programmation générique ?*

Question 10.3 *On veut écrire la classe* `template<class T> class Matrice` *pour manipuler les matrices de type* `T`*. Que faut-il mettre dans le fichier* `.h` *et dans le fichier* `.cpp` *?*

Question 10.4 *Quels types de paramètres sont possibles pour les classes template ?*

Question 10.5 *Combien de paramètres peut avoir une classe template ?*

Question 10.6 *A quoi sert le paramètre* `n` *dans la définition suivante :*
`template<class T, int n> class Tampon` *? Supposons que le patron possède un* `Tampon<T,n> & operator=(const Tableau<T,n> & OP)`*. Peut on affecter le contenu d'un* `Tampon<int,200>` *à un* `Tampon<int,300>` *?*

Question 10.7 *Comment écrire les méthodes des classes patron ?*

Question 10.8 *Ecrivez la fonction générique* `void trier(T* T1, int nb)` *en supposant que* `T1` *est un tableau classique et* `nb` *indique le nombre d'éléments dans ce tableau.*

Question 10.9 *Sous quelle condition votre fonction fonctionne-t-elle ?*

Question 10.10 *Combien de fonctions sont amies pour les deux classes* `Tableau_int` *et* `Tableau<int>` *? Les définitons sont :*

```
template<class T> class Tableau;
template<class T>  bool operator==(const Tableau<T>& T1, const Tableau<T>& T2);

 class Tableau_int {
 protected:
    int * _pt;        // tableau int
    int _nb;          // nombre d'elements
 public:
     ...
 template<class T> friend  bool operator== (const Tableau<T>& T1, const Tableau<T>& T2);
    ...
};

template<class T> class Tableau {
 protected:
    T * _pt;          // pointeur sur le tableau
    int _nb;          // nombre d'elements
 public:
     ...
 friend  bool operator== <>(const Tableau<T>& T1, const Tableau<T>& T2);
    ...
};
```

Question 10.11 *Comment construire une classe patron* `Liste` *? (L'exercice est purement pédagogique, la classe existe déjà dans la STL...)*

Question 10.12 *Comment organiser les itérations au sein de cette classe ?*

Question 10.13 *Comment externaliser les itérations ?*

10.3 Exercice (TP)

10.3.1 Tableaux

Créez les patrons nécessaires pour traiter des tableaux, des tableaux qui gèrent les éléments à partir des pointeurs et des tableaux de chaînes de caractères.

10.3.2 Listes

Ecrivez deux versions de la classe template `template<class T> Liste` : la première gère les itérations par un élément courant dans la classe et la deuxième emploie une classe `template<class T> ListeIterateur` pour organiser les parcours.

Question 10.14 *Comment réaliser la classe* `template<class T> Ensemble` *en supposant que l'ensemble peut être implémenté par une liste ?*

Chapitre 11

Patrons et solutions standardisés : STL

Objectifs

— Présenter l'intérêt des patrons et des solutions réutilisables, paramétrés
— Description de la structure de la STL
— Familiariser les concepts "objet-fonction", "containeur", "iterateur", "adaptateur"
— Connaître les conteneurs usuels
— Approfondir les connaissances sur les iterateurs
— Présenter des solutions et des fonctions génériques

11.1 Résumé

11.1.1 Intérêt des patrons réutilisables, paramétrés

L'intérêt de la généralisation des solutions qui sont souvent utilisées sous forme d'une bibliothèque (la STL) est évident.

Les composants de la STL sont réutilisables. Cette bibliothèque est standardisée aujord'hui, son utilisation peut garantir la portabilité, la compréhensibilité et la maintenabilité du logiciel. L'implémentation des solutions dans la STL est souvent optimisée. Les composant offrent une bonne efficacité et garantissent des algorithmes avec des complexités documentées.

La STL contient des classes patrons (des conteneurs), des structures, des itérateurs, des algorithmes, des fonctions d'intérêt général, des exceptions, des adaptateurs, des allocateurs, des utilitaires ... A découvrir... Voici quelques éléments des solutions.

11.1.2 Conteneurs

Structures de données prédéfinies permettant de stocker des objets. Des accès aux éléments et des méthodes manipulant des données sont fournis ; ils sont indépendants du type des données. Exemples :

```
vector<T>// tableau dynamique, gere lui meme l'espace
        // memoire dont il a besoin
set<T>        // ensemble d'objets
list<T>        // liste chainee
map<key,T,less(key)>    // table d'association
```

Le conteneur `map` est une table, c'est-à-dire un tableau, qui, au lieu d'être indéxé par un entier est indéxé par un type `key` arbitraire. Ce conteneur permet de retrouver rapidement une information de type `T` à partir d'une clé de type `key` (une clé ne peut être stockée qu'une seule fois). Les vecteurs (les tableaux) sont des sortes de map pour lesquels le type des clés est le type entier. Cependant ils sont beaucoup plus limités que les maps :
- ils n'autorisent que des entiers comme clés ;
- ils utilisent tout l'espace mémoire de 0 a n-1 même si seulement quelques indices sont associés à des éléments.
Les maps :
- autorisent des clés de n'importe quel type
- réservent un espace mémoire proportionnel au nombre de clés stockées.

La méthode `find (key)` retourne un itérateur sur l'élément `T` associé à la clé fournie si elle existe dans la map ou sinon un itérateur sur la marque de fin. Un itérateur sur une map possède deux composantes : une clé et un 'élément. On y accède par les méthodes `first()` et `second()`.

11.1.3 Itérateurs sur les conténeurs

Eléments permettant de spécifier une position dans un conténeur. Un itérateur peut être incrémenté ou décrémenté et utilisé pour faciliter l'accès aux objets stockés. Pour parcourir des conteneurs, les procédures (algorithmes) génériques de la STL sont basées sur les itérateurs. Les conténeurs sont capables de retourner des itérateurs qui indiquent le début et la fin des parcours (des séquences de données).

`Exemples:`

```
vector<int> TAB; // tableau dynamique d'entiers
vector::iterator deb = TAB.begin();
vector::iterator fin = TAB.end();
vector::iterator it;
for ( it = deb; it < fin; it++) {. . . }
```

11.1.4 Objets-fonctions

Encapsulations des fonctions par des objets qui porte le même nom que la fonction en question et qui possède obligatoirement la surcharge de l'opérateur (). Grâce à cette surcharge, on peut appeler la fonction en faisant appel à l'opérator() de la classe. Cette solution permet de généraliser le passage de fonctions à des procédures/ méthodes. Exemple :

Soit la classe personne comme ci-dessous :

```
class personne {
string _nom;;
string _adresse;
int _haut;
public:
. . .
int hauteur() const {return _haut;} // retourne la hauteur
};
```

Pour comparer la taille de deux personnes, soit un objet fonction :

```
class sup{
public:
bool operator() (const personne& P1, const personne& P2)
{ return P1.hauteur() > P2.hauteur(); }
};
```

On veut utiliser un tableau de personnes. Pour cela, on instancie la solution générique vector de STL :

```
vector<personne> P;
```

Pour trier le tableau dans l'ordre croissant des hauteurs des personnes, on peut utiliser un algorithme générique, mais cette fois-ci avec l'objet fonction sup qui permet lq comparaison de deux personnes :

```
sort(P.begin(), P.end(), sup());
```

11.1.5 Solutions génériques dans la STL

Algorithmes génériques

Plusieurs sorts d'algorithmes génériques existent dans les bibliothèques de STL. Il existe des algoÂrithmes qui donnent des procédures / méthodes indépendantes des types de données, paramétrables par les types (tri générique, insertion générique, ...). Les conteneurs, les itérateurs permettent la généralisation des méthodes. Exemples :

```
sort(TAB.begin(),TAB.end());
```

Utilitaires

STL contient aussi des méthodes et des utilitaires d'intérêt général. Ces méthodes peuvent soulager le travail (souvent factidieux) du programmeur. Prenons comme exemple une possibilité codée dans STL et accessible par <utility.h>.

Exemple1 : Economie de codage en cas de l'uitlisation des relations d'ordre

Soit la classe vecteur (vue en TP) :

```
class vect_eu {
int _x, _y, _z;
public:
vect_eu(int , int, int) {. . .}
bool  operator==(const vect_eu& V) const
{ return _x == V._x && _y == V._y && _z == V._z ; }
bool  operator<(const vect_eu& V) const
{ return (_x*-x+_y*_y+_z*_z) <(V._x*V._x+V._y*V._y+ V._z *V._z); }
};
```

Ici, on suppose qu'on veut comparer les vecteurs selon les longueurs. En utilisant l'espace de nommage `rel_ops` de la partie utilitaire de la bibliothèque standarde, on a le droit d'utiliser toutes les relaÂtions d'ordre entre les `vect_eu` sans les coder :

```
#include <utility.h>
using namespace std::rel_ops;
main() {
vect_eu V1(5, 32, 131);
vect_eu V2(6, 44, 98);
. . .
if (V1 >= V2) . . .
. . .
```

Exemple2 : Utilisation d'une méthode générique de transformation d'un séquence d'objets à l'aide d'un objet fonction

Parmi les algorithmes génériques du STL, il existe plusieurs méthodes de transformations de séquenÂces d'objets. Voici la déclaration d'une d'elles :

```
template<class IterIn1, class IterIn2, class IterOut, class OperBin>
IterOut transform(IterIn1 debut1, IterIn1 fin1, IterIn2 debut2, IterOut result,
OperBin op) ;
```

Cet algorithme réalise la transformation générique suivante :

```
depuis i = 0 jusqu'\'a (fin1-debut1-1) faire :
*(result+i) = op( *(debut1+i), *(debut2+i) );
```

La bibliothèque standarde (cf. <functional.h>) contient également des objets fonctions qui sont applicable pour les fonctions souvent utilisées. Voici un exemple :

```
template <class T> struct plus : binary_function<T,T,T> {
T operator()(const T& x, const T& y) const
{ return x+y; }
};
```

L'addition de deux vecteurs (de même taille) peut être résolue de la façon suivante :

```
#include <functional.h>
#include <vector.h>
. . .
main() {
vector<double> A;
vector<double> B;
. . .
transform(A.begin(), A.end(), B.begin(), B.end(), A.begin(), plus<double>() );
```

11.2 Questions

Question 11.1 *Quel est l'intérêt de la STL ?*

Question 11.2 *Décrivez les conténeurs.*

Question 11.3 *Comment les itérateurs fonctionnent-ils ?*

Question 11.4 *Comment définir les objets-fonctions ?*

Question 11.5 *Etudiez les algorithmes génériques de la STL.*

11.3 Exercice (TP)

Réaliser les programmes suivants en utilisant les possibilités de la STL
Ecrire un programme permettant de lire une suite d'entiers, de les trier et de les afficher.

Soit deux vecteurs `x` et `y`. On suppose qu'ils sont stockés par des conténeurs vector. Ecrire et tester la fonction produit scalaire de deux vecteurs.

Supposons que l'on traite des vecteurs creux, c'est-à-dire des vecteurs dans lesquels la plupart des éléments sont égaux à 0. Supposons par exemple que la taille `n` du vecteur soit de 1 million mais que seulement quelques milliers d'éléments de `x` et de `y` soient non nuls. Ni l'algorithme écrit ci-desÂsus, ni l'algorithme générique ne tirent avantage du fait que l'on manipule des vecteurs creux. Ils vont calculer 1 million de multiplications et d'additions. En stockant les vecteurs `x` et `y` dans des tables d'asÂsociation et en ne stockant que les éléments non nuls, on réduit :
- la taille nécessaire pour stocker des vecteurs
- le temps nécessaire au calcul du produit scalaire.

Ecrire la méthode produit scalaire de deux vecteurs stockés dans des maps.

Soit `A` un vecteur ligne et `B` un vecteur colonne. Ecrire la procédure `mult_scalaire(...)` qui calcule - à l'aide des algorithmes génériques `transform(...)` - le produit scalaire des deux vecteurs.