TPs 1 et 2
Mise au point d'une classe Rationnel

Nous nous proposons d'écrire une application permettant de manipuler des nombres rationnels. Nous allons pour cela représenter le type Rationnel à l'aide d'une classe. Avant de commencer, voici en vrac quelques bonnes habitudes pour faciliter la lecture (et la relecture) de votre code (ces pratiques sont classiques) :

Conventions de nommage

- Traditionnellement, les noms des données membres d'une classe (ou *attributs* d'une classe) sont en minuscules et commencent par `my_` ou `mon_` ou `m_` ou juste `_`. Si cela n'a rien d'obligatoire, cela permet de distinguer très facilement les variables et les paramètres des données propres à la classe. (Voir l'exemple de la classe `Rationnel` ci-dessous.)
- Les noms de classes ou structures commencent par des majuscules (si ce sont des mots composés, les mots sont accolés et le début de chaque mot est mis en majuscule) : `class Point; class Vecteur; class TriangleIsocele; ...`
- les noms des fonctions membres (ou *méthodes*, ou *opérations*) commencent par une minuscule ; si leur nom est formé de plusieurs mots, mettre une majuscule au début des mots intérieurs : `void initialise(); void translateHorizontalement(float dx); float vitesseAngulaire(); ...`
- les noms des paramètres d'une fonction et des variables locales sont en minuscules ; s'ils sont composés de plusieurs mots, les séparer par `_` : `float delta_x, delta_y; float distance(const Point & autre_point); ...`
- les noms des constantes, même propres à la classe, sont en majuscules ; si ils sont formés de plusieurs mots, mettre des `_` entre : `const float PI = 3.14; const int MAX_NB_ELEMENTS = 100;`

Exercice 1 : La classe Rationnel

1. Proposez une interface (fichier `Rationnel.h`) ; l'accès aux attributs sera privé :

```
class Rationnel {
  private :
    int my_num ;
    int my_deno ;
  public :
  ...
} ;
```

   Dans cette première version, vous offrirez les fonctions membres suivantes :
   - Un constructeur prenant en paramètres deux entiers.
   - affiche : affiche sur la sortie standard le Rationnel sous la forme : `<numérateur> / <dénominateur>`
2. Ecrivez les fonctions membres (fichier `Rationnel.cc`).

3. Ecrivez un programme permettant de tester le fonctionnement de la classe (fichiers TesteRationnel.cc et Makefile).

Exercice 2 : Les constructeurs

Le constructeur que vous avez défini prend en paramètres deux entiers ; mais dans certaines situations, on peut avoir besoin de construire des objets Rationnel sans pouvoir fournir les paramètres ; une situation typique est celle d'un tableau de Rationnel. Il faut donc définir un constructeur par défaut. De plus, pour utiliser des Rationnel définis par défaut, on est généralement conduit à modifier leurs valeurs. Mais le numérateur et le dénominateur sont des attributs privés ! Une solution est que la classe fournisse des accesseurs en écriture.

1. Ecrivez les fonctions membres suivantes :
   - Un constructeur par défaut (quelles valeurs choisir ?).
   - setNum : affecte une valeur au numérateur.
   - setDeno : affecte une valeur au dénominateur.
2. Modifiez votre programme pour vérifier que tout cela fonctionne bien.

Exercice 3 : Les opérations sur les rationnels

1. Ajoutez deux fonctions membres qui permettent de :
   - transformer un rationnel en son inverse,
   - tester l'égalité d'un rationnel avec un autre rationnel.

   Testez à l'aide de votre programme d'essai ces nouvelles fonctionnalités.
2. Vous allez maintenant ajouter les 4 opérations arithmétiques de base : addition, soustraction, multiplication et division que vous prendrez soin de tester. Par exemple, le prototype de la fonction de soustraction sera :

```
void Rationnel::soustraction(const Rationnel & autre, Rationnel & difference) const
```

Exercice 4 : Pour aller un peu plus loin

Ecrivez et testez les fonctions membres suivantes :

- reduit : transforme le Rationnel en sa forme réduite (simplifiée). Vous pourrez écrire une fonction utilitaire pgcd ; une solution élégante serait de créer deux fichiers util.h et util.cc dans lequels vous déclarerez puis définirez pgcd.
- toString : retourne la chaîne de caractères correspondant au Rationnel. Cette méthode facilite l'affichage. On peut maintenant écrire :

```
q1.addition(q2,q3);
cout << q1.toString() << ‘‘+’’ << q2.toString() <<‘‘=’’ << q3.toString() << endl;
```

  Pour écrire cette méthode, vous définirez une fonction utilitaire intToString en vous inspirant du code suivant :

```
ostringstream oss;
oss << x;
string s = oss.str();
```

  Il est possible d'associer un flux à une chaîne de caractères grâce à un `ostringstream` de la biblothèque `sstream`. Nous utiliserons alors ces flux de la même manière que

nous utilisons les flux d'entrée/sortie `cin` et `cout`. A tout moment, nous pourrons récupérer la chaîne associée au flux grâce à la méthode `str()`.

Selon le temps qui vous reste, vous pouvez faire en sorte que la chaîne retournée par toString corresponde aux usages : forme réduite, éventuel signe "-" au numérateur, enfin 3 et non 3/1, et 0 au lieu de 0/12.

Exercice 5 : Une classe Complexe

On rappelle qu'un nombre complexe peut s'écrire $a+ib$ où a et b sont des nombres réels, tandis que i est un nombre dit "imaginaire" et tel que $i^2 = -1$.

a est appelé la partie réelle, b la partie imaginaire du nombre complexe $a + ib$.

Soient deux complexes $a + ib$ et $a' + ib'$, leur somme vaut $(a + a') + i(b + b')$

et leur produit $(aa' - bb') + i(ab' + ba')$

En vous inspirant de la classe Rationnel, écrivez une classe Complexe : `Complexe.h, Complexe.cc, TesteComplexe.cc, Makefile`.

Si vous avez terminé, passez au TP 2 bis.

TP 2 bis
Mise au point d'une classe Polygone

Nous nous proposons de définir une classe `Polygone` où le nombre de côtés n'est pas défini à l'avance (mais est cependant inférieur à un maximum donné). Pour représenter un polygone, nous allons utiliser un tableau de taille variable de `Point`. Vous trouverez dans `www.labri.fr/perso/fbaldacc/AP2/2/source/` les fichiers `Point.cc` et `Point.h`.

Exercice 6 : Fichier en-tête `Polygone.h`

Ecrivez le fichier `Polygone.h`. Pour cette première question, vous intégrerez les déclarations des méthodes suivantes : saisie d'un `Polygone`, affichage un `Polygone` (méthode `toString()`), translation d'un `Polygone` (méthode `deplace(...)`) et ajout d'un sommet (instance de la classe `Point`) à un polygone. Pensez à déterminer les méthodes constantes ou non ainsi que les passages par référence constante.
Le diagramme de classes ci-dessous donne la liste de toutes les méthodes que vous aurez mises en oeuvre à la fin de ce TP.

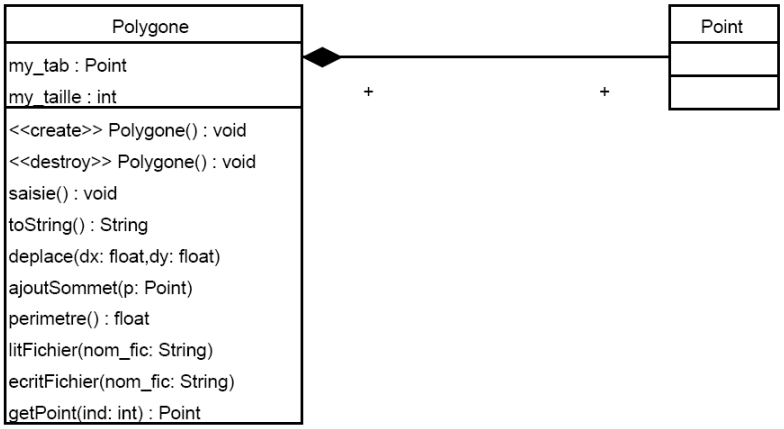

FIGURE 1 – Interface de la classe Polygone.

Exercice 7 : Les autres fichiers

Ecrivez un fichier `Polygone.cc` dans lequel vous donnerez pour chacune des fonctions proposées dans l'interface une définition provisoire minimale, par exemple

```
void Polygone::saisie() {
    cout << "saisie..." << endl;
}
```

Cette démarche vous permet d'écrire « dans la foulée » un fichier `main.cc` contenant la fonction `main`, ainsi qu'un fichier `Makefile`. Vous pouvez ainsi vérifier ce que vous faites à mesure.

Exercice 8 : Méthodes de la classe `Polygone`

Définissez complètement les fonctions membres de la classe `Polygone`. Vous ne devez pas modifier la classe `Point`. Vous ferez attention à l'accès aux données membres de cette classe `Point`.

Exercice 9 : Périmètre d'un `Polygone`

Ajoutez à la classe `Polygone` une méthode permettant au `Polygone` de calculer son périmètre.

Exercice 10 : Lecture et écriture d'un `Polygone` dans un fichier

On considère un fichier dans lequel sont stockés des `Point`. On souhaite pouvoir affecter un `Polygone` à partir d'un tel fichier de `Point`, et aussi sauvegarder un `Polygone` dans un fichier.

1. Ecrivez une méthode pour la classe `Point` qui permet de lire un `Point` dans un flux passé en paramètre.
2. Ecrivez une méthode pour la classe `Polygone` qui prend en paramètre le nom d'un fichier de `Point`, et affecte au `Polygone` les `Point` qui sont lus dans ce fichier.
3. Ecrivez une méthode pour la classe `Polygone` qui permet de sauvegarder un `Polygone` dans un fichier. Quelle fonctionnalité faut-il ajouter à la classe `Point` ? Modifiez la classe `Point` en conséquence.

Exercice 11 : Récupérer un point d'un `Polygone`

Ajoutez à la classe `Polygone` une méthode `getPoint` permettant de récupérer le `Point` d'indice `ind` dans le `Polygone`. Les deux prototypes suivants sont possibles. Testez les deux, expliquez pourquoi et comment ça fonctionne.

```
Point getPoint( int ind ) const;
// ou
const Point & getPoint( int ind ) const;
```

Exercice 12 : Pour compléter proprement la classe `Polygone`...

Il faudrait rajouter au minimum :

- le constructeur de copie

- l'opérateur d'affectation

Surcharger aussi l'opérateur de flux `operator<<` serait également bien utile !

Pourquoi pas aussi d'autres opérateurs comme `operator==`, etc... A vous alors d'en définir la signification sur un `Polygone` et de l'implémenter dans chacune des méthodes.

TP 3
Notion d'agrégation

1 Les principaux mécanismes à travers un exemple

Le but de la séance est de savoir manipuler des classes dans lesquelles les données membres peuvent comprendre non seulement des types de base, mais aussi d'autres classes.

Pour cela, nous devons revenir sur le mécanisme d'instanciation d'une classe afin de le comprendre parfaitement et savoir, *dans tous les cas*, répondre aux questions suivantes :
- Quand et comment est appelé le constructeur ? Quel constructeur est appelé ?
- Quand et comment est appelé le destructeur ?

Vous savez actuellement répondre à ces questions dans le cas d'une classe *simple*, mais pas dans le cas d'une classe contenant des objets appartenant à d'autres classes. C'est donc le principal objectif de la séance !

Remarque importante : vous aurez très peu de code `C++` à écrire lors de la première partie de cette séance, ce n'est pas le but ! Nous vous fournissons volontairement la majorité du code pour vous permettre de faire de nombreux tests interactifs, et ainsi répondre par vous-même aux questions posées dans le sujet.
Compiler et exécuter le programme sans chercher à comprendre TOUS les affichages **ne vous apportera rien**.

L'exemple de base : les classes Point et Cercle

Recopiez chez vous le répertoire (et la totalité de son contenu) :
`www.labri.fr/perso/fbaldacc/AP2/3/source/`. Il contient un `Makefile`, les fichiers `mainpoint.cc` et `main.cc` utilisant deux classes dont les fichiers sont aussi dans le répertoire :
- la classe `Point`, que vous ne modifierez pas ; ces fichiers sont les mêmes qu'au TP précédent, pour une version "papier", reportez vous au document distribué en TD,
- une ébauche de la classe `Cercle` que vous allez tester et compléter. Les principaux fichiers sont en annexes de ce document.

Remarque : toutes les méthodes constructeur et destructeur de ces classes affichent un message à l'écran, ceci permettra de vérifier quand une méthode est appelée.

Exercice 13 : Classe `Point`.
Lisez attentivement le code de la classe `Point`, ainsi que le fichier `mainpoint.cc`.
Exécutez ce programme `mainpoint`, et assurez-vous que vous savez interpréter TOUS les affichages produits.

Exercice 14 : Classe `Cercle`.

1. Lisez attentivement le code de la classe `Cercle`. Vous remarquerez qu'elle possède une donnée membre de type `Point`, et vous porterez une attention particulière à la façon dont cette donnée membre est traitée (cf. commentaires dans le code).

2. Lisez attentivement le fichier `main.cc`. Exécutez le programme, et grâce aux affichages, reconstituez l'enchaînement des appels des méthodes et la *vie* de chaque objet, pas à pas sur une feuille de brouillon. En particulier, comment et quand est construit le point `my_centre` de chacun des cercles instanciés dans la fonction `main` ? Même question pour sa destruction.
3. La classe `Cercle` ne définit pas explicitement de constructeur par copie, donc un constructeur par copie est fourni par défaut.
   (a) Que remarquez-vous sur ce constructeur ? En particulier, comment est construit le point `my_centre` du cercle `c4` à la fin de la fonction `main` ?
   (b) Définissez explicitement un constructeur par copie dans la classe `Cercle`. Que remarquez-vous par rapport à l'exécution précédente ?

Constructeurs et la notation " :"

Nous venons de voir que, pour chaque constructeur défini explicitement dans la classe `Cercle`, son exécution fait appel automatiquement au constructeur par défaut de la classe `Point` pour instancier la donnée membre `my_centre`. Ainsi, si le point `my_centre` doit contenir d'autres valeurs que celles par défaut, il faut ensuite (dans le constructeur `Cercle` courant) modifier les valeurs de ce point. (cf. "constructeur rayon + x + y" et "constructeur rayon + point"). C'est un peu dommage...

Il est possible de spécifier à un constructeur de la classe `Cercle` quel constructeur de la classe `Point` utiliser pour instancier le point `my_centre` en donnant les paramètres d'appel de ce constructeur. Voici par exemple le "constructeur rayon + x + y" :

```
Cercle::Cercle( float r, float x, float y )
: my_centre( x, y )
// ici, appel Point::constructeur
{
  cout << "Cercle::Constructeur rayon + x + y " << endl;
  my_rayon = r;
  // plus besoin de modifier my_centre
}
```

De manière générale, à la suite du prototype du constructeur, et AVANT son corps délimité par `{ }`, on rajoute `: my_donnee1( params ), my_donnee2( params )...` (attention, les " :" font partie de la syntaxe !). Le type et l'ordre des `params` déterminent bien sûr le constructeur appelé ; ce sont soit des constantes, soit des paramètres du constructeur courant (comme pour l'exemple de `Cercle` donné ci-dessus).

Exercice 15 : Modifiez les constructeurs de la classe `Cercle` pour lesquels vous jugerez utile un tel fonctionnement. Vérifiez alors le bon enchaînement des appels à l'exécution du programme.
Le constructeur par copie n'échappe bien sûr pas à cette règle : faites en sorte qu'il fonctionne comme celui par défaut (cf. exercice précédent), c'est-à-dire qu'il fasse appel au constructeur par copie de la classe `Point`.

2 Pour aller plus loin

Nous vous proposons d'enrichir les classes Point et Cercle de la manière suivante : nous allons ajouter dans chaque classe deux données membres

- `static int nb_instance`, qui va compter le nombre d'instances de la classe.
- `int id`, qui va identifier l'instance et va être automatiquement initialiser lors de la construction de l'instance.

```
//Point.h
class Point {
  private :
    float my_abs, my_ord;
    static const float EPSILON;
    static int nb_instance;
    int id;
  public :
    ...
};

//Point.cc
...
const float Point::EPSILON=0.00000001; // calcul
int Point::nb_instance = 0;

Point::Point( float x, float y ) {
  nb_instance ++;
  id = nb_instance;
  // ou alors.. mais là on complique tout en rendant plus lisible!
  //    Point::nb_instance++;            // variable de classe
  //    (*this).id = Point::nb_instance; // affectation de cette variable
                                        // à une variable d'instance

  cout << "Point::Constructeur : " << x << " , " << y
       << " pour le point "<< id <<endl;
  my_abs = x;
  my_ord = y;
}
```

À vous de deviner l'intérêt d'un tel mécanisme.

Un Point ayant une "vie propre", nous pourrions définir la classe Cercle de la manière suivante :

```
class Cercle {
  private :
    Point *my_centre; // un pointeur sur un Point
    float my_rayon;
```

...

Implémenter.

3 Une application complète : la boîte englobante

Dans ce problème, on s'intéresse à définir et représenter une boîte englobant un certain nombre de points du plan. On se sert de deux classes : une classe `Point` qui représente tout point du plan, une classe `Boite` qui représente un ensemble de points et un rectangle (le rectangle minimal du plan pour lequel tous les points associés à cette boîte sont à l'intérieur). On considère que les points sont placés dans un repère orthonormé classique, i.e. que l'axe des x est orienté de la gauche vers la droite et que l'axe des y est orienté du bas vers le haut. La Figure 2 ci-dessous illustre le concept de boîte englobante d'un ensemble de points.

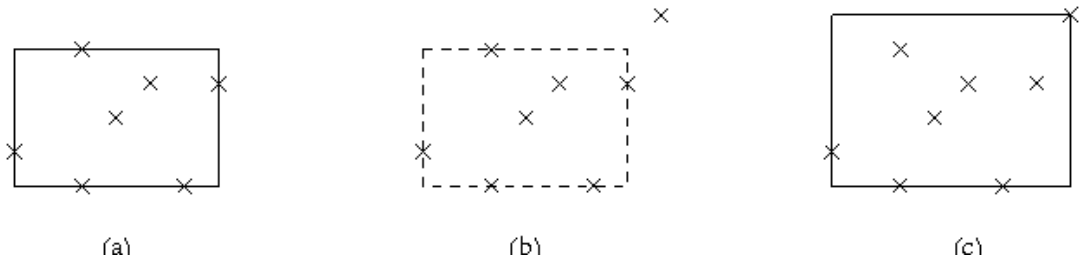

FIGURE 2 – Boîte englobante : (a) boîte englobante d'un ensemble de points, (b) ajout d'un point, (c) nouvelle boîte englobante de ces points.

Dans le répertoire
`www.labri.fr/perso/fbaldacc/AP2/3/source/Boite_englobante`, vous avez à votre disposition un `Makefile`, la classe `Point` vue précédemment, et l'interface de la classe `Boite`. Vous allez écrire dans la suite les fichiers `Boite.cc` et `main.cc`.

Vous écrirez (**et testerez pas à pas ! !**) les fonctions de l'interface de la classe `Boite` en suivant les étapes ci-dessous.

Exercice 16 : Les constructeurs et le destructeur.
A noter que le constructeur par défaut (construction sans paramètre) de la classe `Boite` est interdit d'usage (car placé dans la section `private`) : une boîte englobante n'a de sens qu'à partir de un point (au moins). L'attribut `my_bas_gauche` permet de mémoriser le coin bas-gauche du rectangle englobant, l'attribut `my_haut_droite` permet de mémoriser le coin haut-droit de ce rectangle. L'attribut `my_nb_alloues` désigne le nombre de cellules allouées dynamiquement dans le tableau de points `my_points`. L'attribut `my_nb_points` désigne le nombre de points effectivement stocké dans ce tableau.

Ecrivez le constructeur qui construit une boite englobant un point : on suppose qu'on alloue un tableau de 5 points et que le point passé en paramètre est stocké en premier.

Puis écrivez le destructeur.

Exercice 17 : Deux méthodes simples.

1. Ecrivez la méthode `affiche` de la classe `Boite` qui affiche à l'écran la valeur de l'ensemble des attributs d'une boite.
2. La méthode `interieur` de la classe `Boite` doit renvoyer `true` si le point `p` passé en paramètre est à l'intérieur de la boîte, `false` sinon. Ecrivez cette méthode.

Exercice 18 : Pour ajouter un point.

1. Ecrivez la méthode `agrandir` de la classe `Boite`. Cette méthode est appelée lorsque l'on ajoute un point à la boîte et qu'il n'y a plus de place pour le mémoriser dans le tableau `my_points`. On considère que l'on double la taille du tableau à chaque fois.
2. Ecrivez maintenant la méthode `ajouterPoint`. L'objet `Boite` doit à la fois mémoriser le nouveau point **et** mettre à jour si nécessaire la boîte englobante.

Exercice 19 : Ecrivez la méthode `supprimerPoint`. Vous devez vérifier que le point appartient bien à l'ensemble des points mémorisés dans la boîte englobante. Puis, vous devez enlever ce point de ce tableau et recalculer **si nécessaire** la boîte englobante.

Exercice 20 : Rajoutez deux méthodes : le constructeur par copie et l'opérateur d'affectation de la classe `Boite`.

Annexes : classe Cercle et main.cc

```
////////////////////////////////////////////////////////////////////////
// Fichier Cercle.h

#ifndef __CERCLE__
#define __CERCLE__

#include <sstream>
#include <string>
// Inclusion de l'entete "Point.h" car utilise ci-dessous.
#include "Point.h"

using namespace std;

class Cercle {
  private :
    Point my_centre;
    float my_rayon;
  public :
    Cercle();                                   // constructeur par defaut
    ~Cercle();                                  // destructeur
    Cercle( float r );                          // constructeur rayon
    Cercle( float r, float x, float y );        // constructeur rayon + x + y
    Cercle( float r, const Point & p );         // constructeur rayon + point
    string toString() const;
    // ... a completer par vos soins.
};

ostream& operator<<(ostream& out, const Cercle& c);
#endif

////////////////////////////////////////////////////////////////////////
// Fichier Cercle.cc

#include <iostream>
#include <sstream>
#include <string>
#include "Cercle.h"
#include "Point.h"

using namespace std;

ostream& operator<<(ostream& out, const Cercle& c) {
  out << c.toString();
  return out;
}
```

```
Cercle::Cercle()
// ici, appel Point::constructeur par defaut
{
  cout << "Cercle::Constructeur par defaut" << endl;
  my_rayon = 0;
}

Cercle::~Cercle()
{
  cout << "Cercle::Destructeur" << endl;
  // ici, appel Point::destructeur
}

Cercle::Cercle( float r )
// ici, appel Point::constructeur par defaut
{  cout << "Cercle::Constructeur rayon " << endl;
  my_rayon = r;
}

Cercle::Cercle( float r, float x, float y )
// ici, appel Point::constructeur par defaut
{
  cout << "Cercle::Constructeur rayon + x + y " << endl;
  my_rayon = r;
  // modification du point construit par defaut
  my_centre.setX( x );
  my_centre.setY( y );
}

Cercle::Cercle( float r, const Point & p )
// ici, appel Point::constructeur par defaut
{
  cout << "Cercle::Constructeur rayon + point " << endl;
  my_rayon = r;
  // modification du point construit par defaut
  my_centre = p;  // affectation bien definie
                  // ou sinon :
                  // my_centre.setX( p.getX() );
                  // my_centre.setY( p.getY() );
}

string
Cercle::toString() const
{
  ostringstream ostr;
```

```
  ostr << "centre : " << my_centre.toString() << endl;
  ostr << "rayon :  " << my_rayon;
  return ostr.str();
}

////////////////////////////////////////////////////////////////////////////
// Fichier main.cc

#include <iostream>
#include "Cercle.h"
#include "Point.h"

using namespace std;

int
main()
{
  // test de tous les constructeurs
  Cercle c0 ;
  cout << c0 << endl ;

  Cercle c1( 4 ) ;
  cout << c1 << endl ;

  Cercle c2( 2, 3.5, 8.2 ) ;
  cout << c2 << endl ;

  Point p( 6, 7.5 );
  Cercle c3( 2, p ) ;
  cout << c3 << endl ;

  // test du constructeur par copie fourni par defaut (que remarquez-vous ??)
  // puis du votre quand vous l'aurez ecrit...
  Cercle c4 = c3;
  cout << c4 << endl ;

  // tout est desalloue ! verifiez les affichages...
}
```

TP 4
Héritage simple - Hiérarchie Oeuvre

Le but de la séance est d'apprendre à créer et manipuler une hiérarchie d'héritage simple en C++. Nous nous appuierons sur un exemple de hiérarchie de classes entre `Oeuvre` et `Peinture` dont différentes versions sont présentées dans `www.labri.fr/perso/fbaldacc/AP2/4/source/`.

La dernière version nous intéressera tout particulièrement pour ce TP. Elle fait intervenir 3 classes, `Date`, `Oeuvre`, et `Peinture`, formant la hiérarchie d'héritage suivante :

```
Date ( int my_jour,          Oeuvre  ( string my_auteur,my_titre,my_style,
       int my_mois,            ^       Date   my_date_creation,
       int my_annee )          |       int    my_valest )
                               |
                           Peinture ( string my_possesseur,
                                      Date   my_date_acquis )
```

Définition de la hiérarchie d'héritage

Vous allez reprendre et modifier la dernière version en suivant les contraintes suivantes.

Vous avez à votre disposition (inchangées pour l'instant) les classes `Date` et `Oeuvre`.

On distinguera une œuvre `plastique` d'une œuvre `écrite`. Une œuvre `plastique` aura comme particularité d'appartenir à "quelqu'un", et l'on voudra alors connaître la date à laquelle l'œuvre a changé de main. Pour une œuvre `écrite`, les caractéristiques qui nous intéressent sont la langue dans laquelle est écrite l'œuvre, ainsi que le thème qu'elle aborde.

On voudra alors pouvoir définir et manipuler des peintures, des sculptures et des livres. Pour une `peinture`, on aimera connaître son support (toile, papier, etc) et sa matière (aquarelle, peinture à l'huile, gouache etc), pour une `sculpture`, ce sera son matériau (pierre, bois, marbre, etc), et pour un `livre`, son éditeur et sa date d'édition.

Exercice 21 : Dessiner sur votre feuille une nouvelle hiérarchie d'héritage respectant le texte précédent. Précisez pour chaque classe quels sont les membres "rajoutés", ainsi que leurs types.

Implémentation de la hiérarchie d'héritage

Exercice 22 : Modifications légères dans la classe `Oeuvre`.
Modifiez le constructeur de `Oeuvre` pour qu'il fasse appel au "bon" constructeur de `Date`

et non le constructeur par défaut (cf. commentaires dans le source).
Rajoutez également un constructeur par copie pour `Oeuvre`.

Exercice 23 : Implémentez ET testez une à une, en "descendant", les classes de votre hiérarchie d'héritage. Chaque classe devra contenir au minimum un constructeur, un constructeur par copie, un destructeur et une méthode pour afficher son contenu.

Exercice 24 : On veut pouvoir faire changer de main une peinture ou une sculpture, c'est-à-dire changer son possesseur et sa date d'acquisition. Que faut-il modifier/ajouter pour que cela fonctionne ? Il y a 2 ou 3 solutions (au moins), explorez et testez chacune d'elles.

TPS 5 et 6 (2 séances)
Hiérarchie Formes géométriques

Le but de ce TP est de compléter puis de manipuler par polymorphisme une hiérarchie d'héritage de taille conséquente en C++. Nous nous appuierons sur un exemple de hiérarchie de classes de formes géométriques introduite en TD.

1 La hiérarchie de classes de formes

Une partie de cette hiérarchie est déjà implémentée : récupérez chez vous le répertoire (et la totalité de son contenu)
`www.labri.fr/perso/fbaldacc/AP2/5_6/source/`.

Etude de la hiérarchie et du code C++ fourni

Exercice 25 : Le but ici est de se familiariser avec l'ensemble du code fourni. Pour cela :

1. Consultez le fichier `A_LIRE.txt` qui vous servira de guide.
2. Consultez le code de l'ensemble des fichiers des classes des formes, et dessinez sur votre feuille la hiérarchie d'héritage mise en oeuvre.
3. Compilez, testez, lisez le reste du code, etc... En particulier, soyez sûrs d'avoir bien compris le rôle de la classe `Screen` et de la manière de l'utiliser dans les autres classes et dans le `main`.

Rajout de classes dans la hiérarchie

On souhaite compléter la hiérarchie avec 2 classes de formes : `Triangle` qui hérite de la classe abstraite `Shape`, et `Square` qui hérite de `Rectangle`.

Exercice 26 : Implémentez ces deux nouvelles classes :

1. complétez les fichiers vides à votre disposition, `Triangle.*` et `Square.*`,
2. et testez les en décommentant leur utilisation dans le `main`.
3. Les fichiers `util.*` contiennent des fonctions de saisie des formes qui seront utiles pour la suite. Rajoutez les fonctions de saisie pour les deux nouvelles formes que vous venez d'implémenter.

2 Gestion d'un dessin = ensemble de formes

Avant de commencer cette partie, recopiez dans votre répertoire de travail sur les formes les fichiers contenus dans le sous-répertoire `Supplements` :

- `DessinShapes.*` qui sont vides,

- `main.cc` à décommenter et à compléter,

- `Makefile` complet.

Vous trouverez également dans ce sous-répertoire `Supplements` un exécutable de la version finale du programme que vous allez développer au cours de ce TP. Testez le pour vous faire une idée précise des fonctionnalités attendues.

On souhaite donc écrire ici un programme permettant de créer et de manipuler un ensemble de formes dessinées à l'écran, en interaction avec un utilisateur via un menu.

Pour cela, il faut une structure de données qui permet de stocker les formes créées pour pouvoir à tout moment les afficher, les modifier, etc... Nous proposons d'utiliser un **tableau de pointeurs sur** `Shape`. Ce tableau sera mis à jour à chaque ajout/suppression de formes, et sera parcouru "par polymorphisme" pour l'affichage ou toute modification des formes.

La classe `DessinShapes`

Pour rendre le code plus lisible et plus modulaire, on propose de gérer ce tableau de pointeurs sur `Shape` au sein d'une classe dont voici une entête (minimum) :

```
class DessinShapes
{
  private:
    static const int MAX = 100;
    Shape * my_tabShapes[MAX];
    int my_nbShapes;
  public:
    DessinShapes();
    ~DessinShapes();
    void addShape( Shape * pshape );
    void refresh( Screen & s );
};
```

Les attributs `my_tabShapes` et `my_nbShapes` servent évidemment à stocker les pointeurs sur les formes créées par ailleurs (dans le `main` par exemple). La méthode `addShape` sert à rajouter un pointeur au tableau, et la méthode `refresh` à afficher l'ensemble des formes à l'écran. Voici un exemple d'utilisation classique de ces méthodes si la variable `dessin` est de type `DessinShapes` :

```
dessin.addShape( new Line(black, Point(4,5), Point(15,22)) );
dessin.refresh( ecran );
```

Exercice 27 : Implémentez cette version minimum de la classe `DessinShapes`. Cela vous permettra ainsi d'obtenir les fonctionnalités 1 à 6 du menu proposées dans le `main`. Implémentez une à une ces fonctionnalités d'ajout des 6 formes et de leur affichage à l'écran.
Remarque : pensez que les fichiers `util.*` contiennent des fonctions de saisie des paramètres pour les formes, utilisez les !

3 Rajout de fonctionnalités

S'appliquant à toutes les formes (c, d et e dans menu)

Exercice 28 : Rajoutez à la classe `DessinShapes` les 3 méthodes suivantes :

```
void setColourAll( char col );
void moveAll( int dx, int dy );
void eraseAll();
```

et utilisez les pour implémenter les options c, d et e du menu.

S'appliquant seulement à une forme (x, y et z dans menu)

Ces fonctionnalités se réalisent en deux étapes :

1. sélection d'une forme,
2. application de la modification sur cette forme.

L'étape de sélection d'une forme peut être réalisée de plusieurs façons différentes, par exemple en fonction de l'ordre de création des formes (indice dans le tableau), ou selon l'appartenance ou non d'un point à cette forme.

Exercice 29 : Avec une sélection par indice.
Rajoutez à la classe `DessinShapes` la méthode `Shape * select( int ind ) const` qui retourne le pointeur sur la forme d'indice `ind`, ou `NULL` si l'indice est mauvais. Il suffit alors d'appliquer la bonne fonction sur le pointeur de forme obtenu pour implémenter les options x, y et z du menu.
Remarque : il y a une subtilité pour l'option z, il faut aussi modifier le tableau de formes... donc, une nouvelle fonction dans la classe `DessinShapes` serait bien utile pour effacer "proprement" la forme d'indice `ind` du dessin : `void erase( int ind )` .

Exercice 30 : Avec une sélection par point contenu dans la forme.
Même chose que précédemment, mais en utilisant comme fonction de sélection une méthode qui retourne le pointeur sur la première forme du dessin contenant un point p, ou `NULL` sinon : `Shape * select( const Point & p ) const`
Cela implique donc de rajouter dans la hiérachie des formes une méthode qui permet de tester l'appartenance d'un point à une forme. Pour certaines formes, le calcul géométrique exact peut s'avérer très compliqué... Vous vous restreindrez à tester si le point appartient à la plus petite boite englobant la forme.

4 Pour aller plus loin

Modifier le rayon d'un cercle (r dans menu)

On souhaite offrir la possibilité à l'utilisateur de sélectionner un cercle du dessin (avec une des méthodes vues avant), puis lui appliquer une méthode de modification de la valeur

du rayon (`void setRadius( int r )` par exemple). Cette méthode `setRadius` n'a bien sûr un sens QUE pour la classe `Circle`, toutes les autres classes de la hiérarchie n'ont pas à contenir cette méthode.
Voici donc un bon exemple pour vous essayer au transtypage !

Dessin d'un cercle

Dans la classe `Screen`, la méthode `put_circle` qui permet de dessiner un cercle discret est vide. Implémentez la grâce à l'algorithme fourni en Annexe ci-après.

Annexe. L'algorithme de dessin d'un cercle discret de Bresenham.

C'est un algorithme classique, largement commenté et étudié dans tous les bons livres sur le graphisme.

L'algorithme décrit ci-dessous dessine un cercle dont le centre a pour coordonnées $(0, 0)$. Vous l'adapterez à votre cas.

Principe. À chaque tour de boucle, on calcule les coordonnées d'un point du cercle dans un octant du plan, et par symétrie, on obtient les 7 autres. Pour choisir quels points à coordonnées entières "donnent la meilleure approximation" des points réels du cercle, on gère le calcul d'une erreur sur la distance (d) entre ces points.

```
// Affiche les points dans chacun des 8 octants
Action affiche_points_cercle (point pt)
Debut
  Affiche(pt.x, pt.y)   ; Affiche(pt.y, pt.x)
  Affiche(pt.y, -pt.x)  ; Affiche(pt.x, -pt.y)
  Affiche(-pt.x, -pt.y) ; Affiche(-pt.y, -pt.x)
  Affiche(-pt.y, pt.x)  ; Affiche(-pt.x, pt.y)
Fin
// Affiche un cercle grace a l'algorithme de Bresenham
Action dessine_cercle()
Debut
  point pt(0, radius);
  int d = 3 - 2 * radius;
  Tant Que (pt.x < pt.y)
  Faire Debut
           affiche_points_cercle(pt)
           Si (d < 0)
           Alors   d += 4 * pt.x + 6;
           Sinon Debut
                   d += 4 * (pt.x - pt.y) + 10;
                   pt.y--;
                 Fin
           pt.x++;
         Fin
  Si (pt.x == pt.y)
  Alors  affiche_points_cercle(pt)
Fin
```

TP 7
Evaluation d'expressions

Dans ce TP, nous montrons comment évaluer des expressions arithmétiques données sous forme postfixée d'abord, puis sous forme infixée.

Préparation

Copiez tous les fichiers du répertoire
`www.labri.fr/perso/fbaldacc/AP2/7/source/` chez vous. Tapez `make` pour construire automatiquement l'exécutable `main`. Tapez `make clean` pour effacer les fichiers objets.

Il y a trois modules : le module `main.cc` que vous allez éditer, le module `Pile.cc` qui vous fournit le type abstrait pile, le module `token.cc` qui permet de décomposer en opérandes et opérateurs les expressions tapées sur la ligne de commandes.

Dans le module `main.cc` sont décrits les différents exercices.

Les piles en C++

On vous fournit une *classe générique* `Pile` vous permettant de manipuler des piles d'objets de type quelconque. Le prototype de la classe `Pile` est dans `Pile.h`. Son implémentation est dans `Pile.cc`.

L'exemple suivant montre comment cloner une pile de flottants en C++ :

```
void clonerPile( Pile<float> entree, Pile<float> & clone )
{
  Pile<float> interm;
  while ( ! entree.pileVide() )
    {
      interm.empiler( entree.valeurSommet() );
      entree.depiler();
    }
  while ( ! interm.pileVide() )
    {
      clone.empiler( interm.valeurSommet() );
      interm.depiler();
    }
}
```

On remarque qu'il n'y a pas besoin d'initialiser la pile (c'est fait automatiquement dans la déclaration de la variable). On remarque aussi qu'on utilise la notation objet au lieu de la notation fonctionnelle (voir schéma ci-dessous).

ASD	C++
var p : pile d'entiers	Pile<int> p;
PileVide(p)	p.pileVide()
Empiler(p, val);	p.empiler(val);
Depiler(p);	p.depiler();
val ← ValeurSommet(p);	val = p.valeurSommet();

Avant de se lancer ...

Regardez la fonction `analyseExpr`. Elle permet de découper une expression arithmétique (passée sous forme de chaîne de caractères) en ses constituants (opérateurs, opérandes, divers). Elle vous inspirera pour les exercices suivants.

Vous pouvez constater aussi que vous disposez d'un ensemble de fonctions simples vous permettant de savoir si une partie d'expression est un nombre, un opérateur, etc.

Et maintenant ...

Exercice 31 : Ecrivez la fonction `evalExprPost` qui permet d'évaluer une expression postfixée donnée en paramètre.

Exercice 32 : Comme précédemment sauf qu'il faut vérifier la validité de l'expression postfixée donnée (fonction `evalSecuriseeExprPost`).

Exercice 33 : Ecrivez la fonction `exprInfVersExprPost` qui convertit une expression infixée vers une expression postfixée. Pour ce faire, il faudra utiliser une pile d'opérateurs (pile de caractères). La fonction `priorite` sera utile.

Exercice 34 : Comme l'exercice précédent mais en ajoutant la possibilité de mettre des parenthèses dans l'expression infixée (fonction `exprInfVersExprPost2`)..

TP 8
Manipulation de Listes

Le but de ce TP est de vous faire manipuler les listes. Vous ne serez que des utilisateurs de cette classe. Nous verrons plus tard comment elle peut être implémentée. Notez que cette pratique est rendue possible grâce à la compilation séparée.

Pour utiliser la classe liste, vous devez donc dans un premier temps récupérer son fichier entête et son fichier source. Créez chez vous un répertoire de travail pour ce TP, puis copiez tous les fichiers du répertoire
`www.labri.fr/perso/fbaldacc/AP2/8/source/` dans ce répertoire.

Les listes en C++

Nous souhaitons développer un module de gestion des notes des élèves. Ainsi, une classe `Etudiant` vous est proposée (fichiers `Etudiant.cc` et `Etudiant.h`). Chaque `Etudiant` est formé d'un nom (`string`) et d'une note (`float`). Pour la suite du TP, vous pourrez éventuellement rajouter des méthodes à cette classe.

On vous fournit une *classe générique* `Liste` vous permettant de manipuler des listes d'objets de type quelconque. Le prototype de la classe `Liste` est dans `Liste.h`. Son implémentation est dans `Liste.cxx`.

On remarque qu'on utilise la notation objet au lieu de la notation fonctionnelle (voir le schéma suivant).

ASD	C++
var l : liste d'etudiants	Liste<Etudiant> l;
adr ← AdressePremier(l)	adr = l.adressePremier()
adr2 ← AdresseSuivant(l, adr)	adr2 = l.adresseSuivant(adr)
val ← ValeurElément(l, adr)	val = l.valeurElement(adr)
ModifieValeur(l, adr, val)	l.modifieValeur(adr, val)
InsérerEnTete(l, elem)	l.insererEnTete(elem)
InsérerAprès(l, elem, adr)	l.insererApres(elem, adr)
SupprimerEnTete(l)	l.supprimerEnTete()
SupprimerAprès(l, adr)	l.supprimerApres(adr)
NULL	l.null()

L'exemple complet suivant montre comment inverser une liste d'étudiants en C++ :

```
// Inclusion de l'en-tete definissant le type abstrait.
#include "Liste.h"
```

```
// Raccourci pour TAdresse.
typedef Liste<Etudiant>::TIterator TAdresse;

// Inversion de liste.
void inverserListe( Liste<Etudiant> l_entree,      // E : passage par valeur
                    Liste<Etudiant> & l_inverse )  // ES : passage par reference
{
  // l_inverse est supposee vide.
  TAdresse adr = l_entree.adressePremier();
  while ( adr != l_entree.null() )
    {
      l_inverse.insererEnTete( l_entree.valeurElement( adr ) );
      adr = l_entree.adresseSuivant( adr );
    }
}

// Programme principal
int main()
{
  Liste<Etudiant> L1;
  Liste<Etudiant> L2;
  for ( int i = 0; i < 10; i++ )
    {
      Etudiant e(``sans nom'', i);
      L1.insererEnTete( e );
    }
  // en ne considerant que les notes:
  // L1 vaut 9 8 7 6 5 4 3 2 1 0
  inverserListe( L1, L2 );
  // en ne considerant que les notes:
  // L2 vaut 0 1 2 3 4 5 6 7 8 9
}
```

On remarque qu'il n'y a pas besoin d'initialiser la liste : le constructeur par défaut est invoqué automatiquement à la déclaration des variables. Comme le C++ est fortement typé, on est obligé de définir un type `TAdresse` pour chaque type de liste. Dans l'exemple ci-dessus, on a défini le type `TAdresse` pour les listes d'étudiants.

Dans ce TP, vous allez écrire dans un fichier `gestion.cc` toutes les fonctions demandées ci-dessous. Vous écrirez le programme principal permettant de les utiliser (et de les valider) dans le fichier `main.cc`. Les prototypes des fonctions de `gestion.cc` seront placés dans le fichier entête `gestion.h`.

Exercice 35 : Fichiers gestion.* et Makefile
Créez (vides pour l'instant) les deux fichiers `gestion.cc` et `gestion.h` et modifiez le `Makefile` en conséquence.
Dans quel fichier doit maintenant se trouver la définition suivante ?
`typedef Liste<Etudiant>::TIterator TAdresse;`

Exercice 36 : Liste non triée

1. Écrivez une action qui permette de saisir une liste d'étudiants (noms et notes).
2. Écrivez une action qui permette d'afficher une liste d'étudiants (noms et notes). Vérifiez que votre fonction de saisie fonctionne correctement.
3. Écrivez une fonction qui étant donné un nom `n` renvoie son rang dans une liste (1, 2, etc.) ou retourne `0` s'il est absent.
4. Écrivez une fonction qui retourne la moyenne des notes de la liste des étudiants.

Exercice 37 : Liste triée

Nous considérons à présent la liste d'étudiants triée en fonction des notes (ordre croissant).

1. Écrivez une action qui permette de rajouter un étudiant à la bonne place dans une liste triée.
2. Écrivez une fonction qui vérifie si une liste est triée (retourne un booléen).
3. Écrivez une action qui fusionne deux listes triées en une liste triée.

Exercice 38 : Tri

Supposons que le module serve à gérer la notation d'un qcm (questionnaire à choix multiples). Les étudiants ne peuvent donc avoir que répondu faux (note 0) ou juste (réponse 1). Écrivez une action qui place tous les élèves ayant répondu faux en début de liste et tous les élèves ayant répondu juste en fin de liste.

TPs 9 et 10
Manipulation de Listes (suite)

Comme pour le TP précédent, nous allons manipuler des listes d'Etudiants. Créez donc chez vous un répertoire de travail pour ce TP, puis copiez tous les fichiers du répertoire `www.labri.fr/perso/fbaldacc/AP2/9_10/source/` dans ce répertoire.

Ouvrir/Enregistrer

Exercice 39 : Création d'une liste à partir d'un fichier

1. Ajoutez dans la classe `Etudiant` une méthode `lireFlux` comportant un paramètre `fstream`.
2. Ajoutez maintenant dans les fichiers `gestion` (`.h` et `.cc`) une fonction `fic2Liste` qui construit une liste d'étudiants à partir d'un fichier. Vous disposez d'un fichier `liste1` pour tester votre fonction. Votre `main` peut ressembler à ceci :

```
int main(){
  Liste<Etudiant> l;
  fic2Liste(''liste1'', l);
  afficherListe(l);
}
```

Exercice 40 : Sauvegarde d'une liste

1. Ajoutez dans la classe `Etudiant` une méthode `ecrireFlux`.
2. Ecrivez une fonction `liste2Fic` qui sauvegarde une liste dans un fichier.

Tri de listes

Nous allons maintenant implémenter le *tri fusion*. Le tri se fera sur les noms des étudiants, par ordre croissant. Le principe du tri fusion est simple : on partage la liste en deux sous-listes consécutives, on trie ces sous-listes, et on les fusionne. Dans toute la suite, il sera question de sous-listes ; nous adopterons la convention suivante : si `l` est une liste, `a` et `b` deux TAdresse dans cette liste, la sous liste `(l, a, b)` est l'intervalle *ouvert* `]a, b[`.

L'algorithme s'écrit naturellement de façon récursive :

```
tri(ES l : Liste, E  a, b : TAdresse)
var m : TAdresse
début
  Si (l, a, b) possède au moins deux éléments
  Alors début
```

```
        m <-  milieu(l, a, b)
        tri(l,a,adresseSuivant(l,m))
        m <-  milieu(l, a, b) // le tri de la première moitié
                              // a pu changer l'adresse de l'objet médian
        tri(l, m,b)
        fusion(l,m,a,b)
        fin
fin
```

Exercice 41 : recherche du milieu

1. Créez deux nouveaux fichiers `tri.h` et `tri.cc`, et modifiez le `Makefile` en conséquence.
2. Ecrivez une fonction qui, étant donnée une sous liste `(l, a, b)`, retourne l'adresse de son élément médian ; on conviendra que l'élément médian d'une sous-liste ayant $2k$ éléments est le k^e, tandis que c'est le $(k+1)^e$ dans le cas d'une sous-liste de longueur $2k+1$. De plus, cette fonction ne sera appelée que sur des sous-listes ayant au moins deux éléments. Enfin, la première borne définissant la sous-liste ne sera jamais NULL ; si l'on souhaite connaître le milieu d'une liste entière, on insèrera d'abord en tête un élément fictif, puis on calculera le milieu sur
   `(l, adressePremier(l), NULL)`, et on n'oubliera pas ensuite de supprimer en tête.
   Exemples avec une liste d'entiers :

   `l= (1, 7, 5, 14, 2, 3, 8)`

   supposons que les adresses soient : `a, b, c, d, e, f, g, NULL` ; donc `adressePremier(l)` vaut `a`, `valeurElement(l, f)` vaut `3`, etc... dans ce cas, l'adresse de l'élément médian de `(l, b, g)` est `d` ; de même, l'adresse de l'élément médian de `(l, d, NULL)` est `f`. Insérons 0 en tête, supposons que `adressePremier(l)` vaille maintenant `x` ; alors `milieu(l, x, NULL)` vaut `d`.
   Pour programmer cette fonction, vous ne devez pas compter les éléments de la sous-liste, mais utiliser deux TAdresse dont l'un progressera deux fois plus que l'autre.
   Testez votre fonction, en faisant afficher l'élément médian de quelques sous-listes.

Exercice 42 : fusion Ecrivez une fonction qui, étant données deux sous-listes consécutives déjà triées `(l, a, adresseSuivant(m))` et `(l, m, b)` fusionne ces deux sous-listes. A la fin de la fonction,
`(l, a, b)` est triée. **Vous ne devez pas utiliser de structure intermédiaire (liste ou autre).**

Testez votre fonction à l'aide du fichier `liste2` contenant une liste dont les deux "moitiés" sont déjà triées.

Exercice 43 : tri

1. Vous pouvez maintenant écrire la fonction `triFusion` qui trie récursivement un intervalle ouvert. Testez votre fonction sur les listes disponibles.
2. Ecrivez une fonction `tri` qui trie une liste en utilisant la fonction `triFusion`. Testez.

Tri indirect

A la fin du module AP1, vous avez entendu parler du tri indirect. Il s'agissait de trier des indices au lieu des éléments eux-mêmes ; dans une liste, il n'y a pas d'indices, mais des `TAdresse`. Nous allons donc trier des `TAdresse`, bien sûr toujours en fonction des valeurs des éléments auxquels ils renvoient. L'intérêt de cette démarche est double :

- on n'effectue les échanges que sur les `TAdresse`, ce qui optimise le tri lorsque les éléments de la liste sont des objets de taille importante.
- on peut, sur une même liste, effectuer plusieurs indexations.

Exercice 44 : création d'une liste de TAdresse Ecrivez une fonction `creerIndex` qui crée la liste des `TAdresse` d'une liste donnée. Les adresses de cette liste seront du type `TAdresse2` (défini dans `tri.h`), et les valeurs seront donc du type `TAdresse`.

Exercice 45 : affichage indirect Ecrivez une fonction qui affiche les éléments d'une liste en utilisant une liste d'index (liste de `TAdresse`).

Exercice 46 : tri indirect selon les notes

1. Ecrivez une fonction `milieuIndex` analogue à `milieu` (cf. ci-dessus).
2. Ecrivez `fusionIndirectNotes`
3. De même, écrivez `triFusionIndirectNotes` et enfin `triIndirectNotes`

Exercice 47 : tri fusion - encore !

Une des bases du tri fusion est le découpage de la liste (ou tableau) en deux sous-listes de même taille (ou presque). Cette opération peut être réaliser de la manière suivante :

```
Action découpage (E L: TListe, S laurel : Tliste, S hardy : TListe)
Var flipflop : booléen
Début
  flipflop <- vrai
  adr <- AdressePremier(L)

  Tant Que adr <> NULL
  Faire Début
      Si flipflop = VRAI Alors
          InsérérEnTête(laurel,valeurElément(L,adr))
      Sinon
          InsérérEnTête(hardy,valeurElément(L,adr))
      flipflop <- non flipflop
      adr <- AdresseSuivant(L,adr)
  Fin
Fin
```

Implémenter le tri fusion en vous basant sur cette idée.