ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

# TP 5
# Modèles de classes et de fonctions : ensembles de presque n'importe quoi.

**Objectifs :** Définition d'un type générique complet à l'aide des modèles de classes et de fonctions.
**Durée :** 2 séances

## Introduction

Le but de ce TP est de programmer le type générique *Set* à l'aide d'un modèle de classe et de quelques modèles de fonctions pour certaines opérations.

Le type ensemble sera implémenté à l'aide d'une représentation par liste chaînée simple. De plus, aucune relation d'ordre sur les éléments d'un ensemble n'est supposée. La liste chaînée n'est donc pas une liste triée. Il sera vu en cours bientôt que la bibliothèque standard C++ contient un type qui exploite l'ordre pour optimiser les opérations (pensez notamment aux arbres équilibrés). L'objectif premier de ce TP n'est pas l'efficacité algorithmique (dans la limite du raisonnable bien sûr). La concision, la simplicité et la clarté du code seront privilégiées à la complexité algorithmique des méthodes et fonctions.

Construction et affectation par déplacement ne sont pas demandés. Vous êtes bien entendu libres de les prévoir si vous pensez que vous devez les retravailler.

## 1 Ensemble de doubles

Vous trouverez en page 3 la déclaration d'une classe `SetOfDouble` qui définit les ensembles de nombres à virgule flottante grâce à une liste chaînée. L'en-tête complète et documentée de la classe `SetOfDouble` vous est fournie dans le fichier `SetOfDouble.h` téléchargeable sur Moodle. L'ordre d'apparition des méthodes et fonctions dans ce fichier d'en-tête est celui dans lequel il est naturel (et facile) de les définir. Il vous est donc conseillé de suivre ce même ordre lorsque vous répondrez à la question de la section suivante. Vous n'avez rien à programmer concernant les ensembles de doubles. Votre travail à proprement parler est décrit dans la suite.

Notez, chose importante pour l'utilisation du modèle que vous allez écrire, que l'absence de doublons dans un ensemble est gérée uniquement en faisant appel à l'opérateur == défini sur le type des éléments.

## 2 Modèle de classe : Ensemble générique

En vous inspirant de l'en-tête `SetOfDouble.h`, définissez un modèle de classe (ou *classe paramétrée*) pour un ensemble d'éléments de type quelconque `T`. L'instanciation du modèle devra être possible pour tous les types de base (`double`, `float`, `int`, `bool`, `void*`, etc.) mais aussi pour toute classe autorisant la recopie, l'affectation et pour laquelle l'opérateur == est défini.

Attention, le passage d'arguments par références peut être justifié si le type des éléments s'avère être « large » (pensez à un ensemble de matrices !). En prévision, utilisez donc les références là où cela est pertinent.

Notez que le code fourni dans `SetOfDouble.h` suggère d'implémenter la suppression d'un élément à l'aide d'une méthode (statique) récursive. C'est une suggestion et non pas une obligation.

**Image d'un ensemble par une fonction** Remarquez aussi que le programme de test se termine par la fonction `image` qui, comme son nom l'indique, permet de calculer l'image d'un ensemble par une fonction (i.e., un pointeur de fonction). Définir un modèle de fonction amie d'un modèle de classe demande un peu de « travail », comme vous pouvez le voir dans le polycopié à la page 99 pour l'opérateur +.

**Consigne particulière propre aux modèles** Dans le cas de l'écriture d'un modèle de classe ou de fonction, et pour permettre sa réutilisabilité complète, il faut le définir *entièrement* dans le fichier d'en-tête. (Ce point a été detaillé en cours.)

En effet, si au contraire la bibliothèque est définie à l'aide d'un fichier `.h` et d'un fichier `.cpp` destiné à être compilé, le programmeur est obligé de prévoir tous les types pour lesquels son modèle doit être instancié. Ceci afin que chaque instanciation soit effectivement créée, compilée et placée dans le fichier objet. Par exemple : `Set<double>`, `Set<int>`, etc. Un utilisateur qui disposerait uniquement du fichier `.h` et du fichier objet (`.o`) associé ne pourrait alors utiliser la bibliothèque que pour les types prévus lors de sa compilation.

Le fait de définir complètement un modèle dans le fichier d'en-tête permet de s'affranchir de cette limitation. Il n'y a plus dans ce cas de fichier source « `.cpp` ». Pour ce TP, le modèle de classe `Set<>` sera donc défini entièrement dans le seul fichier `Set.h`.

**Méthode de travail et validation** La validation de votre modèle consiste à tester le programme principal `set_test.cpp` qui vous est fourni sur Moodle (et reproduit en page 4).

Une bonne façon de procéder est de suivre une démarche incrémentale. Pour cela, commencez par commenter l'intégralité du corps de la fonction `main()` de `set_test.cpp` et définissez un modèle de classe quasi vide. Puis, enrichissez votre modèle et dé-commentez au fur et à mesure les opérations correspondantes effectuées par le programme.

La trace d'exécution attendue du programme `set_test.cpp` est donnée en page 5. Notez que l'ordre d'apparition des éléments d'un ensemble résulte ici, entre autre, de l'insertion des éléments en tête de liste.

**Question finale** Pourquoi, dans l'avant dernière instruction de `set_test.cpp`, utilise-t-on l'expression

```
singleton(string("void"))
```

et non pas

```
singleton("void")
```

?

```
class SetOfDouble {              //
                                 //  File SetOfDouble.h
public:                          //
  SetOfDouble();
  SetOfDouble(double x);
  void clear();
  ~SetOfDouble();
  bool isEmpty() const;
  std::ostream & flush(std::ostream & out) const;
  bool contains(double x) const;
  bool isSubsetOf(const SetOfDouble & other) const;
  void insert(double x);
  void remove(double x);
  void insertInto(SetOfDouble & other) const;
  void removeFrom(SetOfDouble & other) const;
  SetOfDouble & operator=(const SetOfDouble & other);
  SetOfDouble(const SetOfDouble & other);
  friend SetOfDouble image(const SetOfDouble & set, double (*function)(double));

private:
  class Node {
  public:
    Node(double value, Node * next = nullptr);
    ~Node();
    double getValue() const;
    Node * getNext() const;
    void setNext(Node * next);
  private:
    double value;
    Node * next;
  };

  Node * list;
  static Node * remove(Node * list, double x);
};

std::ostream & operator<<(std::ostream & out, const SetOfDouble & s);
SetOfDouble singleton(double x);
SetOfDouble emptySet();
bool operator==(const SetOfDouble & a, const SetOfDouble & b);
bool operator<(const SetOfDouble & a, const SetOfDouble & b);
bool operator>(const SetOfDouble & a, const SetOfDouble & b);
SetOfDouble operator|(const SetOfDouble & a, const SetOfDouble & b);
SetOfDouble operator-(const SetOfDouble & a, const SetOfDouble & b);
SetOfDouble operator^(const SetOfDouble & a, const SetOfDouble & b);
SetOfDouble operator&(const SetOfDouble & a, const SetOfDouble & b);
```

```
#include <iomanip>        //
#include <ios>            // File set_test.cpp
#include <iostream>       //
#include <string>
#include "Set.h"
using namespace std;
typedef Set<double> SetOfDouble;

int main(int, char *[])
{
  SetOfDouble e;
  Set<double> f, g, h;
  for (int i = 1; i <= 10; i++) { e.insert(i); }
  for (int i = 5; i <= 20; i++) { f.insert(i); }
  for (int i = 14; i < 20; i++) { g.insert(i); }

  cout << std::boolalpha; // <ios>
  cout << e.contains(5) << endl;
  cout << f.contains(5) << endl;
  cout << g.contains(5) << endl;

  g.remove(18);
  g.remove(19);

  cout << "e = " << e << endl;
  cout << "f = " << f << endl;
  cout << "g = " << g << endl;
  {
    SetOfDouble dummy = g; // Copy constructor
    cout << " dummy = " << dummy << endl;
    dummy.clear();
    cout << " dummy = " << dummy << endl;
    dummy = h; // Assignment
  }            // Destructor

  cout << " e U f = " << (e | f) << endl;
  cout << " e inter f = " << (e & f) << endl;
  cout << " e - f = " << (e - f) << endl;
  cout << " e inter g = " << (e & g) << endl;
  cout << " e diffSym f = " << (e ^ f) << endl;
  cout << " e diffSym g = " << (e ^ g) << endl;
  cout << " e - e = " << (e - e) << endl;
  cout << " e - {40.0} = " << (e - singleton<double>(40.0)) << endl;
  cout << " e U {40} U {50} = " << (e | singleton(40.0) | singleton(50.0)) << endl;
  cout << endl;
  cout << " e = " << e << endl;
  cout << " f = " << f << endl;
  cout << " (e U f) - f " << (e | f) - f << endl;
  cout << " (e U f) - f == e ?   " << (((e | f) - f) == e) << endl;
  cout << " En effet, car e inter f = " << (e & f) << endl;
  cout << endl;
  cout << " e == e U {} ? " << ((e | Set<double>()) == e) << endl;
  cout << " e - {5} < e ? " << ((e - Set<double>(5.0)) < e) << endl;
```

```
  Set<bool> a, b;
  a.insert(true);
  b.insert(false);
  cout << endl;
  cout << " a = " << a << endl;
  cout << " b = " << b << endl;
  cout << " a U b = " << (a | b) << endl;
  cout << " a U b - {true} = " << ((a | b) - singleton(true)) << endl;

  Set<string> dictionary;
  dictionary.insert("void");
  dictionary.insert("Hello");
  dictionary.insert("world");
  dictionary.insert("!");
  cout << " " << (dictionary - singleton(string("void"))) << endl;
  cout << " image(e&f,std::sin) = " << image(e & f, std::sin) << endl;
  return 0;
}
```

# Trace d'execution du programme `set_test.cpp`

```
true
true
false
e = {10,9,8,7,6,5,4,3,2,1}
f = {20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5}
g = {17,16,15,14}
 dummy = {14,15,16,17}
 dummy = {}
 e U f = {11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10}
 e inter f = {10,9,8,7,6,5}
 e - f = {1,2,3,4}
 e inter g = {}
 e diffSym f = {20,19,18,17,16,15,14,13,12,11,4,3,2,1}
 e diffSym g = {17,16,15,14,10,9,8,7,6,5,4,3,2,1}
 e - e = {}
 e - {40.0} = {1,2,3,4,5,6,7,8,9,10}
 e U {40} U {50} = {50,10,9,8,7,6,5,4,3,2,1,40}

 e = {10,9,8,7,6,5,4,3,2,1}
 f = {20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5}
 (e U f) - f {4,3,2,1}
 (e U f) - f == e ?   false
 En effet, car e inter f = {10,9,8,7,6,5}

 e == e U {} ? true
 e - {5} < e ? true

 a = {true}
 b = {false}
 a U b = {false,true}
 a U b - {true} = {false}
 {Hello,world,!}
 image(e&f,std::sin) = {-0.958924,-0.279415,0.656987,0.989358,0.412118,-0.544021}
```