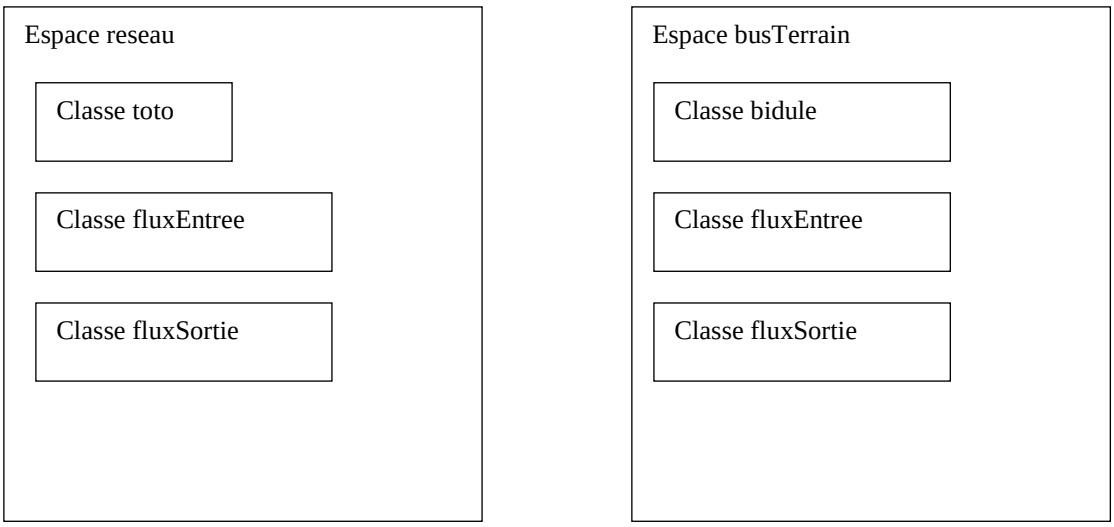
La STL : standard template library

La STL, est une bibliothèque C++ implémentée à l'aide de classe template contenant :

- Un ensemble de classes containers :
  - **Vecteurs**, **tableaux**, listes chaînées , map …
- **Itérateurs** permettant de se déplacer dans des classes telles que Piles …
- **Algorithmes** de tri, recherche …
- Une classe **string** de haut niveau.

1] L'espace de nommage :

L'espace de nommage permet de regrouper des classes par famille.



Plusieurs classes peuvent avoir le même nom si elles sont dans des espaces de nommages différents

Exemple :

```
namespace identificateur;

namespace projetIRIS2
{
      class RFID
      {
      // définition de la classe A
      };
      class RS232
      {
      // définition de la classe B
      };
}
```

Utilisation des classes :

```
      A )
            projetIRIS2::RFID com ;
      B )
            using namespace projetIRIS2 ;
            RFID com ;
```

Deux exemples pour utiliser cout et cin :

```
      A )
            std ::cout << « coucou ;
      B )
            using namespace std ;        //plus besoin de préciser le namespace

            cout << « coucou »  ;
```

2] Les classes conteneurs

std::pair<T1,T2>

Une paire est une structure contenant deux éléments éventuellement de types différents. Certains algorithmes de la STL (find par exemple) retournent des paires (position de l'élément trouvé et un booléen indiquant s'il a été trouvé).

```
#include <pair>
#include <iostream>
#include <string>

int main(){
  std::pair<int,std::string> p = std::make_pair(5,"truc");
  std::cout << p.first << ' ' << p.second << std::endl;
  return 0;
}
```

std::list<T,...>

La classe **list** fournit une structure générique de listes chaînées pouvant éventuellement contenir des doublons.
Cet exemple montre comment insérer les valeurs 4,5,4,1 dans une liste et comment afficher son contenu :

```
#include <list>
#include <iostream>

int main(){
  std::list<int> ma_liste;
  ma_liste.push_back(4);
  ma_liste.push_back(5);
  ma_liste.push_back(4);
  ma_liste.push_back(1);
  std::list<int>::const_iterator t;

  //std::list<int>::const_iterator    lit (ma_liste.begin()),   lend(ma_liste.end());
  for(t = ma_liste.begin()  ;t!=ma_liste.end(); t++)
        std::cout << *t << ' ';
  std::cout << std::endl;
}
```

std::vector<T,...>

La classe **vector** est proche du tableau du C. Tous les éléments contenus dans le **vector** sont adjacents en mémoire, ce qui permet d'accéder immédiatement à n'importe quel élément. L'avantage du **vector** est la ré-allocation automatique en cas de besoin lors d'un push_back par exemple.

Il est conseillé de créer un **vector** en précisant sa taille ;

Exemple :

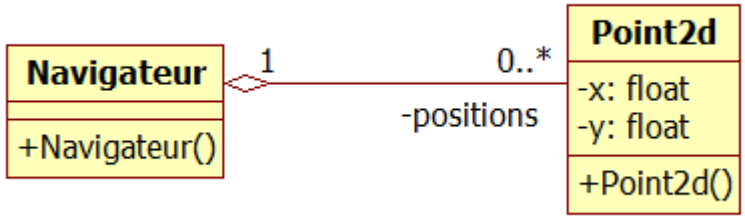
```
#include <vector>
#include <iostream>

int main(){
  std::vector<int> mon_vecteur;
  mon_vecteur.push_back(4);           //stockage du premier élément : int = 4
  mon_vecteur.push_back(2);           //stockage du deuxième élément : int=2
  mon_vecteur.push_back(5);
  // pour parcourir un vector (même const) on peut utiliser les iterators
  for(std::size_t i=0;i<mon_vecteur.size();++i)
        std::cout << mon_vecteur[i] << ' '
```

```
  std::cout << std::endl;

  std::vector<int> mon_vecteur2(5,45); // crée le vecteur 45,45,45,45,45
  for(std::size_t i=0;i<mon_vecteur2.size();++i)
   {std::cout << mon_vecteur2[i] << ' ' ;
   std::cout << std::endl;
   }}
```

Exemple :
Implémentation issu d'un diagramme des classes.



Ce modèle se traduira par une donnée membre '**positions**' dans la classe **Navigateur**. Elle sera de type **vector<Point2d>**

std::set<T,...>

La classe **set** permet de décrire un ensemble ordonné et sans doublons d'éléments. Il faut a priori passer cet ordre en paramètre template . Par défaut, le foncteur **std::less** (basé sur l'opérateur <) est utilisé, ce qui revient à avoir un ensemble d'éléments classés du plus petit au plus grand. Concrètement il suffit donc d'implémenter l'opérateur < d'une classe ou d'une structure de type T pour pouvoir définir un **std::set<T>.** De plus, le type T doit disposer d'un constructeur vide T().
Les autres foncteur sont :

equal_to
not_equal_to
greater
greater_equal
less_equal
binary_function

```
#include <set>
#include <iostream>

  std::set<int> s; // équivaut à std::set<int,std::less<int> >
//std::set<int , std::greater_equal<int>  > s;
  s.insert(2); // s contient 2
  s.insert(5); // s contient 2 5
  s.insert(2); // le doublon n'est pas inséré
  s.insert(1); // s contient 1 2 5
  std::set<int>::const_iterator    sit (s.begin()),    send(s.end());
  for(;sit!=send;++sit)
            std::cout << *sit << ' ';
  std::cout << std::endl;
```

Attention : le fait de supprimer ou ajouter un élément dans un std::set remet invalide ses iterators. Il ne faut pas modifier un std::set dans une boucle for basée sur ses iterators

std::map<K,T,...>

Une **map** permet d'associer une clé (identifiant) à une donnée (table associative).

La map prend au moins deux paramètres templates :

- le type de la clé K
- le type de la donnée T

Le type K doit être ordonné (cet ordre peut être passé en 3e paramètre template, **std::less<K>** par défaut) et .
Le type T impose juste d'avoir un constructeur vide.

Attention, le fait de supprimer ou ajouter un élément dans le map, modifie les itérateurs. Ne pas modifier un map dans un boucle basée sur les itérateurs.

Attention : le fait d'accéder à une clé via l'opérateur [ ] insère cette clé (avec la donnée T()) dans la map. Ainsi l'opérateur [ ] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find. De plus, il ne garantit pas la constance de la map (à cause des insertions potentielles) et ne peut donc pas être utilisé sur des const std::map.

Exemple :
```
#include <map>
#include <string>
#include <iostream>

int main(){
  std::map<std::string,unsigned> map_mois_idx;
  map_mois_idx["janvier"] = 1;
  map_mois_idx["février"] = 2;
  //...
  std::map<std::string,unsigned>::const_iterator    mit (map_mois_idx.begin()),
    mend(map_mois_idx.end());
  for(;mit!=mend;++mit) std::cout << mit->first << '\t' << mit->second << std::endl;
  return 0;
}
```
Ou
```
 std::map<std::string,unsigned , /*ordre porte sur la clé*/std::greater<std::string> > mmois;
 mmois["février"] = 2;
 mmois["juillet"]  = 7;
 mmois["janvier"] = 1;
 //...
 std::map<std::string,unsigned>::const_iterator   mit (mmois.begin());
 std::map<std::string,unsigned>::const_iterator   mend(mmois.end());
 for(;mit!=mend;++mit) std::cout << mit->first << '\t' << mit->second << std::endl;
```

3] Les itérateurs :

Nous avons vu dans la section précédente que les **iterators** permettaient de parcourir aisément une structure de la STL d'un bout à l'autre.
Ils sont définis pour toutes les classes de la STL évoquées ci-dessus (hormis bien sûr std::pair)

3.1] iterator et const_iterator

Un **iterator** (et un **const_iterato**r) permet de parcourir un **container** du début à la fin. Un **const_iterator**, contrairement à un **iterator**, donne un accès uniquement en lecture à l'élément "pointé".

De manière générale, quand on a le choix entre des iterators ou des const_iterator, il faut toujours privilégier les const_iterator car ils rendent la section de code à laquelle ils servent plus générique (applicable aux containers const ou non const).

- begin() : retourne un iterator qui pointe sur le premier élément
- end() : retourne un iterator qui pointe juste "après" le dernier élément
- ++ : permet d'incrémenter l'iterator en le faisant passer à l'élément suivant.

Exemple

```
 map<char,int> mymap;
  map<char,int>::iterator it;

  mymap['a']=50;
  mymap['b']=100;
```

```
  mymap['c']=150;
  mymap['d']=200;

  it=mymap.find('b');
  mymap.erase (it);
  mymap.erase (mymap.find('d'));

  // print content:
  cout << "elements in mymap:" << endl;
  cout << "a => " << mymap.find('a')->first << " "<< mymap.find('a')->second << endl;
  cout << "c => " << mymap.find('a')->first << " "<< mymap.find('c')->second << endl;
```

Exemple :

```
#include <list>
#include <iostream>

const std::list<int> creer_liste(){
  std::list<int> l;
  l.push_back(3);
  l.push_back(4);
  return l;
}

int main(){
  const std::list<int> ma_liste(creer_liste());
  // ne compile pas car l est const
  // std::list<int>::iterator
  //  lit1 (l.begin()),
  //  lend(l.end());
  //for(;lit1!=lend1;++lit1) std::cout << *lit1 << ' ';
  //std::cout << std::endl;
  std::list<int>::const_iterator
    lit2 (l.begin()),
    lend2(l.end());
  for(;lit2!=lend2;++lit2) std::cout << *lit2 << ' ';
  std::cout << std::endl;
  return 0;
}
```

3.2] reverse_iterator et const_reverse_iterator

Les reverse_iterators et const_reverse_iterator permettent de parcourir les conteneurs en sens inverse.
On utilise alors :

- rbegin() : retourne un iterator qui pointe sur le dernier élément
- rend() : retourne un iterator qui pointe juste "avant" le premier élément
- ++ : permet de incrémenter le reverse_iterator en le faisant passer à l'élément précédent. Exemple :

```
#include <set>
#include <iostream>

int main(){
  std::set<unsigned> s;
  s.insert(1); // s = {1}
  s.insert(4); // s = {1, 4}
  s.insert(3); // s = {1, 3, 4}
  std::set<unsigned>::const_reverse_iterator
    sit (s.rbegin()),
    send(s.rend());
  for(;sit!=send;++sit) std::cout << *sit << std::endl;
  return 0;
}
```

... affiche :

```
4
3
1
```

4] Quelques exemples :

Exemple d'utilisation de la STL : vector, iterator, map, set et list (voir cours)

```
std::vector<int> mon_vecteur;
  mon_vecteur.push_back(4);
  mon_vecteur.push_back(2);
  mon_vecteur.push_back(5);
  // pour parcourir un vector (même const) on peut utiliser les iterators ou les index
  for(std::size_t i=0;i<mon_vecteur.size();++i)
   {std::cout << mon_vecteur[i] << ' ' ;
   std::cout << std::endl;
   }
```

```
  std::vector<int> mon_vecteur2(5,69); // crée le vecteur 69,69,69,69,69
  for(std::size_t i=0;i<mon_vecteur2.size();++i)
   {std::cout << mon_vecteur2[i] << ' ' ;
   std::cout << std::endl;
   }
```

```
  std::set<int , std::greater_equal<int>  > s;
  s.insert(2); // s contient 2
  s.insert(5); // s contient 2 5
  s.insert(2); // le doublon n'est pas inséré
  s.insert(1); // s contient 1 2 5
  std::set<int>::const_iterator sit (s.begin()), send(s.end());
  for(;sit!=send;++sit) std::cout << *sit << ' ';
        std::cout << std::endl;
```

```
  std::map<std::string,unsigned , /*ordre porte sur la clé*/std::greater<std::string> > mmois;
  mmois["février"] = 2;
  mmois["juillet"]  = 7;
  mmois["janvier"] = 1;
```

```
  map<char,int> mymap;
  map<char,int>::iterator it;
  mymap['a']=50;
  mymap['b']=100;
  mymap['c']=150;
  mymap['d']=200;
  it=mymap.find('b');
  mymap.erase (it);
```

```
list <int> L;
list <int>::iterator it;

it = L.begin();
L.push_back(82); //- Ajouter à la fin
it = L.insert(it, 19);
it = L.insert(it, 1);
it = L.insert(it, 8);
it = L.insert(it, 0);
L.push_front(189); //- Ajouter au début
```

TP :

E.1] Dans le cadre de certains projets industriels, il est nécessaire d'analyser des images et pour cela, une bibliothèque nommée CImg permet de réaliser un grand nombre de manipulation sur des images de différents formats (bmp, tiff, png, jpg ..).
La classe Cimg est construite avec un type de données template permettent de coder les pixels de l'image (unsigned char, float …)
Parmi ces opérations, il en existe deux principales qui sont :

- créer une image et la stocker en mémoire (classe template CImg).
- Afficher une image (classe CImgDisplay).

Cette bibliothèque se trouve dans un fichier Cimg.h accessible sur le serveur de fichier.

La documentation concernant les classes CImg et CImgDisplay se trouvent dans le fichier suivant :
<rep_cimg>\documentation\reference\annotated.html.
Afin d'utiliser cette bibliothèque, il faut ajouter dans les sources :

```
#include "cheminReel\cimg.h"
using namespace cimg_library;
```

- En vous aidant de la documentation relative aux classes template de cette bibliothèque, écrire un programme qui charge une image et l'affiche.

E.4] Utilisation de la STL :

Ecrire et tester un programme de gestion de pile d'entiers sous Xp utilisant Cbuilder.
La classe se nomme 'stack' et son paramètre template correspond au type de données à empiler.
Le programme de test doit :

Créer un objet de type stack < int >,
empiler 10 entiers
répéter tant que la pile n'est pas vide
afficher le sommet de la pile
dépiler

corrige de la stl stack

```
//exemple d'utilisation de la stl stack
#include <stack>
void stackStl()
{
stack<int > * pile = new stack<int>();
pile->push(5);
pile->push(12);
cout << "Sommet = " << pile->top() << endl;
pile->pop();
cout << "Sommet = " << pile->top() << endl;
}
```