Les templates de la STL

La bibliothèque standard C++

Comme on l'a déjà plus qu'aperçu, on sait que le C++ dispose d'une bibliothèque standard (**SL** pour **Standard Library**) qui est composée

- de flux,
- de la bibliothèque standard du C,
- de la gestion des exceptions, et…
- de la **STL** (**Standard Template Library** : bibliothèque de « patrons » standard)

On l'a vu précédemment : concervoir des conteneurs de données génériques est une tâche de grande envergure de part le nombre de conteneurs différents (pile, file, liste, …), la complexité et surtout on se pose la question suivante : « Etant des classes génériques, indépendante de toute logique métier, quelqu'un n'aurait-il pas déjà fait tout ce boulot ? »

La réponse est évidemment positive : la STL du C++. Celle-ci implémente de nombreux types de données mais surtout de manière efficace et optimisée. Dans un programme C++, on privilégiera toujours l'utilisation de la STL par rapport à une implémentation manuelle : **gain en efficacité**, **robustesse**, **facilité**, **lisibilité** car standard.

Le tableau dynamique : la classe vector<T>

Un **vector** est un conteneur séquentiel qui encapsule un tableau de taille dynamique (sa taille peut donc varier). Les éléments sont stockés de façon contigüe.

La classe **vector<T>** dispose notamment

- de différents constructeurs : par défaut, initialisation, copie, …
- d'un opérateur **=** permettant d'affecter un vector a un autre vector
- d'un opérateur **[ ]** permettant d'accéder directement aux cases du vecteur
- des méthodes :
  - size_type **size()** → retourne la taille du vecteur (unsigned int), càd le nombre de cases actuellement allouées
  - void **push_back(const T& val)** → alloue et insére une nouvelle case à la fin
  - void **pop_back()** → supprime la dernière case du vecteur
  - void **clear()** → supprime toutes les cases du vecteur
  - …

Consérons l'exemple suivant (TestVector1.cpp) :

```
#include <iostream>
#include <vector>
using namespace std;   // sinon std::vector<int>

void affiche(const vector<int>& v);

int main() {
  vector<int> vec(5);  // vecteur de 5 cases non initialisé

  vec[0] = 10;
  vec[1] = 20;
  vec[3] = 40;
  affiche(vec);

  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);
  vec.pop_back();
  affiche(vec);

  vec.clear();
  affiche(vec);

  vector<int> vec2(3,100); // 3 cases initialisées à 100
```

```
  affiche(vec2);

  vector<int> vec3(vec2);  // copie
  affiche(vec3);

  vector<int> vec4;
  vec4 = vec2;             // opérateur =
  affiche(vec4);

  return 0;
}

void affiche(const vector<int>& v) {
  cout << "Taille actuelle de vec = " << v.size() << endl;
  cout << "Contenu : " << endl;
  for (int i=0 ; i<v.size() ; i++) {
    cout << "v[" << i << "] = " << v[i] << endl;
  }
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ TestVector1.cpp
[student@moon TemplateSTL]$ a.out
Taille actuelle de vec = 5
Contenu :
v[0] = 10
v[1] = 20
v[2] = 0
v[3] = 40
v[4] = 0
Taille actuelle de vec = 7
Contenu :
v[0] = 10
v[1] = 20
v[2] = 0
v[3] = 40
v[4] = 0
v[5] = 1
v[6] = 2
Taille actuelle de vec = 0
Contenu :
Taille actuelle de vec = 3
Contenu :
v[0] = 100
v[1] = 100
v[2] = 100
Taille actuelle de vec = 3
Contenu :
v[0] = 100
v[1] = 100
```

```
v[2] = 100
Taille actuelle de vec = 3
Contenu :
v[0] = 100
v[1] = 100
v[2] = 100
[student@moon TemplateSTL]$
```

Test de classe vector<T> avec des objets

Considérons à présent la classe **Person** dont voici le fichier **Person.h** :

```
#ifndef PERSON_H
#define PERSON_H

#include <iostream>
using namespace std;

class Person {
  private:
    string name;
    int age;

  public:
    Person();
    Person(string name, int age);
    Person(const Person& p);
    ~Person();

    void setName(const string& name);
    string getName() const;

    string toString() const;

    Person& operator=(const Person& p);
};

#endif
```

Et le fichier **Person.cpp** :

```
#include "Person.h"
#include <sstream>
using namespace std;

Person::Person() : name("---"), age(0) {
  cout << "constructeur par defaut" << endl;
}
```

```
Person::Person(string name, int age) : name(name), age(age) {
  cout << "constructeur d'initialisation" << endl;
}

Person::Person(const Person& p) : name(p.name), age(p.age) {
  cout << "constructeur de copie" << endl;
}

Person::~Person() {
  cout << "destructeur" << endl;
}

void Person::setName(const string& name) {
  this->name = name;
}

string Person::getName() const {
  return this->name;
}

string Person::toString() const {
  ostringstream ss;
  ss << name << " (" << age << ")";
  return ss.str();
}

Person& Person::operator=(const Person& p) {
  this->name = p.name;
  this->age = p.age;
  cout << "operateur =" << endl;
  return *this;
}
```

où on a délibérément tracer les différentes méthodes.

Et voici un programme exemple (**TestVector2.cpp**) :

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

void affiche(const vector<Person>& v);

int main() {
  vector<Person> vec(4);  // vecteur de 4 cases

  vec[0] = Person("wagner",50);
  vec[1] = Person("leonard",38);
  vec[3] = Person("quettier",51);
  affiche(vec);
```

```
  Person p(vec[0]);
  vec.push_back(p);
  affiche(vec);

  vec[2].setName("Paul");
  affiche(vec);

  return 0;
}

void affiche(const vector<Person>& v) {
  cout << "Taille actuelle de vec = " << v.size() << endl;
  cout << "Contenu : " << endl;
  for (int i=0 ; i<v.size() ; i++) {
    cout << "v[" << i << "] = " << v[i].toString() << endl;
  }
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestVector2.cpp Person.o -o TestVector2
[student@moon TemplateSTL]$ TestVector2
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
Taille actuelle de vec = 4
Contenu :
v[0] = wagner (50)
v[1] = leonard (38)
v[2] = --- (0)
v[3] = quettier (51)
constructeur de copie
constructeur de copie    // Appel de push_back()
constructeur de copie
constructeur de copie
constructeur de copie
constructeur de copie
destructeur
destructeur
```

```
destructeur
destructeur
Taille actuelle de vec = 5
Contenu :
v[0] = wagner (50)
v[1] = leonard (38)
v[2] = --- (0)
v[3] = quettier (51)
v[4] = wagner (50)
Taille actuelle de vec = 5
Contenu :
v[0] = wagner (50)
v[1] = leonard (38)
v[2] = Paul (0)
v[3] = quettier (51)
v[4] = wagner (50)
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

où on observe que

- l'appel de la méthode **push_back()** de la classe **vector<T>** provoque la réallocation totale d'un nouveau vecteur, la copie des objets présents dans l'ancien vecteur et la destruction de l'ancien vecteur → pour des raisons de performance, il est préférable de fixer la taille du vecteur à sa création (si on connait la taille à priori)
- les objets contenus dans le vecteur sont **modifiables**

Les itérateurs

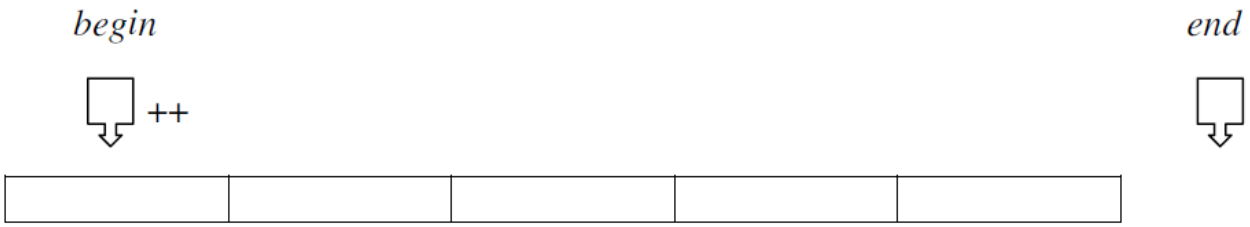
Comme déjà vu, les itérateurs sont une généralisation des pointeurs et permettent le parcours standardisé des conteneurs. Chaque conteneur dispose de ses propres ittérateurs mais ils s'utilisent tous de la même manière.

La classe **vector<T>** dispose des méthodes suivantes :

- **iterator begin()** → retourne un itérateur sur le début du vecteur
- **iterator end()** → retourne un itérateur su la fin du vecteur

- **const_iterator cbegin()** → retourne un itérateur « constant » sur le début du vecteur → cet itérateur n'a qu'un accès en lecture seule sur le conteneur
- **const_iterator cend()** → retourne un itérateur « constant » sur la fin du vecteur

Schématiquement nous avons :



La classe **vector<T>** dispose en outre des méthodes :

- **insert(iterator it, const T& val)** → qui permet d'insérer une nouvelle case dans le vecteur à l'endroit de l'itérateur
- **erase(iterator it)** → supprime la case à l'endroit de l'itérateur

L'objet **iterator** dispose

- de l'opérateur ***** permettant d'accéder à l'élément pointé
- d'un opérateur **++** pour avancer dans le parcours du conteneur
- d'un opérateur **+** (int) pour avancer de plusieurs cases à la fois

On peut également utiliser la méthode libre **std::advance(iterator it, int nbPos)** pour se déplacer de plusieurs cases à la fois.

Voici un exemple d'utilisation de l'itérateur sur un **vector<int>** (**TestVector3.cpp**) :

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

void affiche(const vector<int>& v);
void modifie(vector<int>& v);

int main() {
  vector<int> vec(5);  // vecteur de 5 cases non initialisé
  vec[0] = 10;
  vec[1] = 20;
  vec[3] = 40;
  affiche(vec);

  vector<int>::iterator it = vec.begin();
```

```
  it++;
  *it = 99;
  affiche(vec);

  modifie(vec);
  affiche(vec);

  auto it2 = vec.begin();
  it2++;
  vec.insert(it2,200);
  affiche(vec);

  it2 = vec.begin();
  advance(it2,3);
  vec.erase(it2);
  affiche(vec);

  return 0;
}

void affiche(const vector<int>& v) {
  cout << "Contenu : " << endl;
  for (vector<int>::const_iterator it=v.cbegin() ; it!=v.cend() ; it++) {
    cout << "--> " << *it << endl;
  }
}

void modifie(vector<int>& v) {
  cout << "Modification..." << endl;
  int val = 1;
  for (vector<int>::iterator it=v.begin() ; it!=v.end() ; it++, val++) {
    *it = val;
  }
}
```

Dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ TestVector3.cpp
[student@moon TemplateSTL]$ a.out
Contenu :
--> 10
--> 20
--> 0
--> 40
--> 0
Contenu :
--> 10
--> 99
--> 0
--> 40
--> 0
```

```
Modification...
Contenu :
--> 1
--> 2
--> 3
--> 4
--> 5
Contenu :
--> 1
--> 200
--> 2
--> 3
--> 4
--> 5
Contenu :
--> 1
--> 200
--> 2
--> 4
--> 5
[student@moon TemplateSTL]$
```

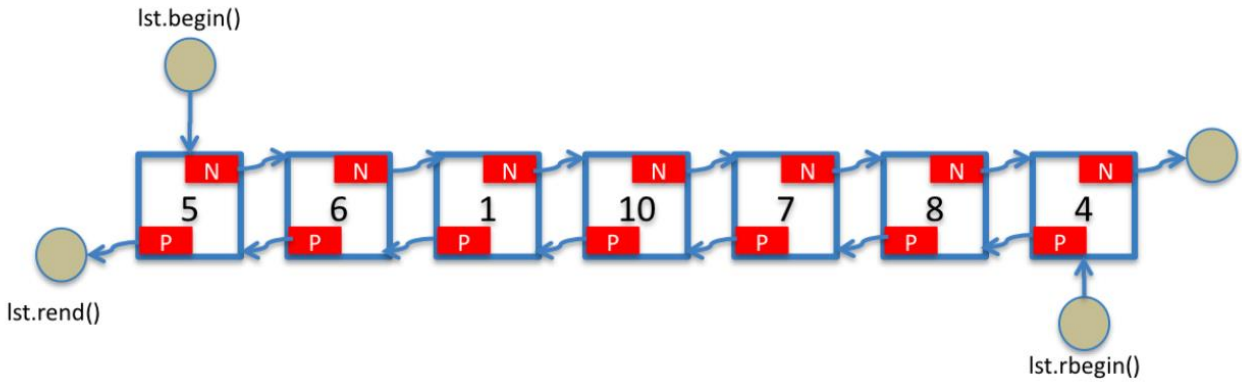
où on observe

- la différence d'utilisation entre un itérateur et un itérateur constant
- l'utilisation du mot clé « **auto** » permettant d'attribuer un type automatiquement à une variable

la classe list<T>

La classe **list<T>** fournit une structure de **listes doublement chaînées** (càd que l'on peut la parcourir dans les deux sens) **pouvant contenir des doublons**.

Schématiquement :



La classe **list<T>** dispose notamment

- de différents constructeurs : par défaut, copie, …
- d'un opérateur **=** permettant d'affecter une list a une autre list
- des méthodes :
  - size_type **size()** → retourne la taille de la liste (unsigned int)
  - void **push_back(const T& val)** → alloue et insére une nouvelle case à la fin
  - void **push_front(const T& val)** → alloue et insére une nouvelle case au début
  - void **pop_back()** → supprime la dernière case de la liste
  - void **pop_front()** → supprime la première case de la liste
  - T& **front()** → retourne une référence sur le premier élément de la liste
  - T& **back()** → retourne une référence sur le dernier élément de la liste
  - void **clear()** → supprime toutes les cases de la liste
  - **insert(iterator it, const T& val)** → qui permet d'insérer une nouvelle case dans la liste à l'endroit de l'itérateur
  - **erase(iterator it)** → supprime la case à l'endroit de l'itérateur
  - …
- des méthodes retournant des itérateurs :
  - **iterator begin()** → retourne un itérateur sur le début de la liste
  - **iterator end()** → retourne un itérateur su la fin de la liste
  - **reverse_iterator rbegin()** → retourne un itérateur sur la fin de la liste et permettant un parcours en sens inverse
  - **reverse_iterator rend()** → retourne un itérateur sur le début de la liste et permettant un parcours en sens inverse
  - les versions constantes de ces itérateurs (…)

Voici un exemple d'utilisation de la classe **list<int>** (**TestList1.cpp**) :

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

int main() {
  list<int> lst; // une liste vide

  lst.push_back( 5 );
  lst.push_back( 6 );
  lst.push_back( 1 );
  lst.push_back( 10 );
```

```
  lst.push_back( 7 );
  lst.push_back( 8 );
  lst.push_back( 4 );
  lst.push_back( 5 );

  lst.pop_back(); // enleve le dernier élément et supprime l'entier 5
  cout << "La liste lst contient " << lst.size() << " entiers : " <<
endl;

  // utilisation d'un itérateur pour parcourir la liste lst
  for (list<int>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << ' ' << *it;
  }
  cout << endl;

  // afficher le premier élément
  cout << "Premier element : " << lst.front() << endl;

  // afficher le dernier élément
  cout << "Dernier element : " << lst.back() << endl;

  // parcours avec un itérateur en inverse
  for (list<int>::reverse_iterator rit = lst.rbegin(); rit != lst.rend();
rit++) {
    cout << ' ' << *rit;
  }
  cout << endl;

  return 0;
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ TestList1.cpp
[student@moon TemplateSTL]$ a.out
La liste lst contient 7 entiers :
 5 6 1 10 7 8 4
Premier element : 5
Dernier element : 4
 4 8 7 10 1 6 5
[student@moon TemplateSTL]$
```

La classe list<> avec des objets

Reprenons la classe Person utilisée précédemment. Voici un exemple d'utilisation de la classe list<Person> (**TestList2.cpp**) :

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

```
using namespace std;
#include "Person.h"

int main() {
  list<Person> lst; // une liste vide

  lst.push_back(Person("wagner",50));
  lst.push_back(Person("leonard",38));
  lst.push_back(Person("quettier",51));

  cout << "La liste lst contient " << lst.size() << " personnes : " << endl;

  // utilisation d'un itérateur pour parcourir la liste lst
  for (list<Person>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << "--> " << (*it).toString() << endl;
  }
  cout << endl;

  auto it = lst.begin();
  it++;
  (*it).setName("Paul");

  // utilisation d'un itérateur pour parcourir la liste lst
  for (list<Person>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << "--> " << it->toString() << endl;
  }
  cout << endl;

  return 0;
}
```

où on observe que

- **l'itérateur** se comporte bel et bien comme un pointeur. On peut l'utiliser soit avec **l'opérateur *** ((*it).toString() dans l'exemple) soit avec **l'opérateur ->** (it->toString() dans l'exemple)

Un exemple d'exécution fournit :

```
[student@moon TemplateSTL]$ g++ TestList2.cpp Person.o -o TestList2
[student@moon TemplateSTL]$ TestList2
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
```

```
destructeur
La liste lst contient 3 personnes :
--> wagner (50)
--> leonard (38)
--> quettier (51)

--> wagner (50)
--> Paul (38)
--> quettier (51)

destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

Quelques algorithmes intéressants

La recherche dans un conteneur

La fonction **std::find()**

- permet de réaliser une recherche dans un conteneur entre une « borne initiale » et une « borne finale »
- utilise **l'opérateur ==** de la classe template pour comparer les objets

Le comportement de cette fonction est à comparer à

```
template<class InputIterator, class T>
InputIterator find (InputIterator first,InputIterator last,const T& val)
{
  while (first!=last) {
    if (*first == val) return first;
    ++first;
  }
  return last;
}
```

Cette fonction

- reçoit donc en paramètre deux itérateurs : **first** et **last**. La recherche se réalise entre les positions de ces deux itérateurs
- retourne un itérateur pointant sur l'objet trouvé si l'objet recherché est trouvé
- retourne **last** si l'objet recherché n'est pas trouvé,

Voici un exemple d'utilisation de cette fonction dans le cas d'un **vector<int>** (**TestFind1.cpp**) :

```
#include <iostream>
#include <vector>
#include <algorithm>    // pour std::sort()
using namespace std;

void affiche(const vector<int>& v);

int main() {
  vector<int> vec(10);  // vecteur de 10 cases non initialisé

  for (int i=0 ; i<10 ; i++) {
     vec[i] = 5 + 11*i;
  }

  affiche(vec);

  // Recherche de 71
  auto it = find(vec.begin(), vec.end(), 71);
  if (it != vec.end()) {
     cout << "71 trouvé ! --> " << *it << endl;
  } else {
     cout << "71 non trouvé..." << endl;
  }

  // Recherche de 42
  auto it = find(vec.begin(), vec.end(), 42);
  if (it != vec.end()) {
     cout << "42 trouvé ! --> " << *it << endl;
  } else {
     cout << "42 non trouvé..." << endl;
  }

  return 0;
}

void affiche(const vector<int>& v) {
  cout << "Taille actuelle de vec = " << v.size() << endl;
  cout << "Contenu : " << endl;
  for (int i=0 ; i<v.size() ; i++) {
    cout << "v[" << i << "] = " << v[i] << endl;
  }
}
```

dont un exemple d'exécution fournit :

```
[student@moon TemplateSTL]$ g++ TestFind1.cpp
[student@moon TemplateSTL]$ a.out
Taille actuelle de vec = 10
Contenu :
v[0] = 5
v[1] = 16
v[2] = 27
v[3] = 38
v[4] = 49
v[5] = 60
v[6] = 71
v[7] = 82
v[8] = 93
v[9] = 104
71 trouvé ! --> 71
42 non trouvé...
[student@moon TemplateSTL]$
```

Prenons à présent l'exemple d'une recherche dans une **list<Person>**. Cependant, il est tout d'abord nécessaire d'implémenter l'**opérateur ==** de la classe **Person**, sinon, le programme compilera avec une erreur.

Donc, nous choisissons ici de faire la comparaison sur le nom uniquement :

```
bool Person::operator==(const Person& p) const {
  return name == p.name;
}
```

qui est donc ici le critère de recherche.

Le programme de test est alors (**TestFind2.cpp**) :

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
#include "Person.h"

int main() {
  list<Person> lst; // une liste vide

  lst.push_back(Person("wagner",50));
```

```
  lst.push_back(Person("leonard",38));
  lst.push_back(Person("quettier",51));
  lst.push_back(Person("caprasse",43));
  lst.push_back(Person("charlet",54));

  for (list<Person>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << "--> " << (*it).toString() << endl;
  }
  cout << endl;

  // Recherche de quettier
  Person p;
  p.setName("quettier");      // critère de recherche
  auto it = find(lst.begin(), lst.end(), p);
  if (it != lst.end()) {
    cout << "quettier trouvé ! --> " << it->toString() << endl;
  } else {
    cout << "quettier non trouvé..." << endl;
  }

  // Recherche de madani
  p.setName("madani");        // critère de recherche
  it = find(lst.begin(), lst.end(), p);
  if (it != lst.end()) {
    cout << "madani trouvé ! --> " << it->toString() << endl;
  } else {
    cout << "madani non trouvé..." << endl;
  }

  return 0;
}
```

dont un exemple d'exécution fournit

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestFind2.cpp Person.o -o TestFind2
[student@moon TemplateSTL]$ TestFind2
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
constructeur de copie
```

```
destructeur
--> wagner (50)
--> leonard (38)
--> quettier (51)
--> caprasse (43)
--> charlet (54)

constructeur par defaut
quettier trouvé ! --> quettier (51)
madani non trouvé...
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

Le tri des conteneurs

La fonction **std::sort()**

- permet de réaliser le tri d'un conteneur entre une « borne initiale » et une « borne finale », selon l' « ordre ascendant »
- utilise **l'opérateur <** de la classe template pour comparer les objets et assurer l' « ordre ascensdant »

Cette fonction reçoit donc en paramètre deux itérateurs : **first** et **last**. Le tri se réalise entre les positions de ces deux itérateurs.

Attention que dans certains cas, la méthode **std::sort()** n'est pas utilisable. Par exemple, pour trier une instance de la classe **list<T>**, il faut utiliser la méthode d'instance **sort()** de la classe **list<T>** elle-même.

Voici un exemple dans le cas d'une **list<int>** (**TestSort1.cpp**) :

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

int main() {
  list<int> lst; // une liste vide

  lst.push_back( 5 );
  lst.push_back( 6 );
  lst.push_back( 1 );
```

```
  lst.push_back( 10 );
  lst.push_back( 7 );
  lst.push_back( 8 );
  lst.push_back( 4 );
  lst.push_back( 5 );

  // Avant tri
  cout << "Avant tri :" << endl;
  for (list<int>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << ' ' << *it;
  }
  cout << endl;

  // Tri
  lst.sort();    // Méthode membre de la classe

  // Apres tri
  cout << "Après tri :" << endl;
  for (list<int>::iterator it = lst.begin(); it != lst.end(); it++) {
    cout << ' ' << *it;
  }
  cout << endl;

  return 0;
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ TestSort1.cpp
[student@moon TemplateSTL]$ a.out
Avant tri :
 5 6 1 10 7 8 4 5
Après tri :
 1 4 5 5 6 7 8 10
[student@moon TemplateSTL]$
```

Prenons à présent l'exemple du tri d'un **vector<Person>**. Cependant, il est tout d'abord nécessaire d'implémenter l'**opérateur <** de la classe **Person**, sinon, le programme compilera avec une erreur.

Donc, nous choisissons ici de faire la comparaison sur le nom uniquement :

```
bool Person::operator<(const Person& p) const {
  return name < p.name;
}
```

qui est donc ici le critère de comparaison.

Le programme de test est alors (**TestSort2.cpp**) :

```
#include <iostream>
#include <vector>
#include <algorithm>    // Pour std::sort()
using namespace std;
#include "Person.h"

void affiche(const vector<Person>& v);

int main() {
  vector<Person> vec(6);  // vecteur de 6 cases

  vec[0] = Person("wagner",50);
  vec[1] = Person("leonard",38);
  vec[2] = Person("quettier",51);
  vec[3] = Person("wagner",37);
  vec[4] = Person("caprasse",43);
  vec[5] = Person("hiard",44);

  // Avant tri
  cout << "Avant tri :" << endl;
  affiche(vec);

  // Tri
  sort(vec.begin(), vec.end());

  // Après tri
  cout << "Après tri :" << endl;
  affiche(vec);

  return 0;
}

void affiche(const vector<Person>& v) {
  cout << "Contenu : " << endl;
  for  (vector<Person>::const_iterator  it=v.cbegin()  ;  it!=v.cend()  ;
it++) {
    cout << "--> " << it->toString() << endl;
  }
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestSort2.cpp Person.o -o TestSort2
```

```
[student@moon TemplateSTL]$ TestSort2
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur par defaut
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
constructeur d'initialisation
operateur =
destructeur
Avant tri :
Contenu :
--> wagner (50)
--> leonard (38)
--> quettier (51)
--> wagner (37)
--> caprasse (43)
--> hiard (44)
constructeur de copie            // Début tri
operateur =
operateur =
destructeur
constructeur de copie
operateur =
operateur =
destructeur
constructeur de copie
operateur =
destructeur
constructeur de copie
operateur =
operateur =
operateur =
operateur =
operateur =
```

```
destructeur
constructeur de copie
operateur =
operateur =
operateur =
operateur =
operateur =
destructeur                    // Fin tri
Après tri :
Contenu :
--> caprasse (43)
--> hiard (44)
--> leonard (38)
--> quettier (51)
--> wagner (50)
--> wagner (37)
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

où on observe que

- les doublons sont autorisés dans ce type de conteneur (tout comme la **list<T>** ; ici « wagner » appraît deux fois). Mais l'ordre des doublons après tri n'est pas conservé
- on voit clairement le tri lors de l'exécution (différents appels à l'opérateur = et constructeur de copie de la classe **Person**)

Il faut noter que la fonction **sort()** n'utilise que **l'opérateur <** de la classe template et non l'opérateur == qui est donc inutile ici.

Un conteneur ordonné : la classe set<T>

La classe **set<T>** fournit un conteneur générique pour lequel

- les éléments présents sont en permanence triés selon l'ordre imposé par **l'opérateur <** des objets présents dans le conteneur
- une fois insérés dans le conteneur, les objets ne peuvent plus être modifiés (ils sont **const**). En effet, une modification d'un objet en place pourrait corrompre l'ordre. Les objets peuvent parcontre être retirés et insérés.
- les « doublons » ou « objets équivalents » ne sont pas autorisés

Attention que l'équivalence entre objet n'est pas mesurée avec l'**opérateur ==** de ces objets mais bien avec leur **opérateur <**. Deux objets **o1** et **o2** sont « équivalents » si

```
!(o1 < o2) && !(o2 < o1)
```

c'est-à-dire s'ils ne sont ni plus grand, ni plus petit que l'autre. Deux objets peuvent donc être égaux selon leur opérateur == mais pas équivalent ! Cela dépend de comment on a défini ces opérateurs.

La classe **set<T>** dispose notamment

- de différents constructeurs : par défaut, copie, …
- d'un opérateur **=** permettant d'affecter un set à une autre set
- des méthodes :
  - size_type **size()** → retourne la taille de la liste (unsigned int)
  - **insert(const T& val)** → alloue et insére une nouvelle case au bon endroit
  - void **clear()** → supprime toutes les cases de la liste
  - **erase(iterator it)** → supprime la case à l'endroit de l'itérateur
  - …
- des méthodes retournant des itérateurs :
  - **iterator begin()** → retourne un itérateur sur le début de la liste
  - **iterator end()** → retourne un itérateur su la fin de la liste

- **reverse_iterator rbegin()** → retourne un itérateur sur la fin de la liste et permettant un parcours en sens inverse
- **reverse_iterator rend()** → retourne un itérateur sur le début de la liste et permettant un parcours en sens inverse
- les versions constantes de ces itérateurs (…)

Nous présentons ici un exemple de la classe **set<Person>**. Afin de mieux voir ce qui se passe, nous avons tracé les **opérateurs ==** et **<** de la classe Person :

```
bool Person::operator==(const Person& p) const {
  cout << "operateur ==" << endl;
  return name == p.name;
}

bool Person::operator<(const Person& p) const {
  cout << "operateur <" << endl;
  return name < p.name;
}
```

Voici le programme de test (**TestSort1.cpp**) :

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

void affiche(const set<Person>& v);

int main() {
  set<Person> lst;  // liste triée vide

  lst.insert(Person("wagner",50));
  lst.insert(Person("leonard",38));
  lst.insert(Person("quettier",51));
  lst.insert(Person("wagner",37));
  lst.insert(Person("caprasse",43));
  lst.insert(Person("hiard",44));

  affiche(lst);

  return 0;
}

void affiche(const set<Person>& l) {
  cout << "Contenu : " << endl;
  for (set<Person>::const_iterator it=l.cbegin() ; it!=l.cend() ; it++) {
    cout << "--> " << it->toString() << endl;
```

```
    }
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestSet1.cpp Person.o -o TestSet1
[student@moon TemplateSTL]$ TestSet1
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
operateur <
constructeur de copie
destructeur
Contenu :
--> caprasse (43)
--> hiard (44)
--> leonard (38)
--> quettier (51)
--> wagner (50)
destructeur
destructeur
destructeur
```

```
destructeur
destructeur
[student@moon TemplateSTL]$
```

où on observe que

- uniquement l'**opérateur <** de la classe **Person** est appelé lors de l'insertion dans le conteneur
- le doublon « wagner » (age == 37) n'a pas été inséré dans le conteneur

Si on refait le même test mais en changeant **l'opérateur <** de **Person** :

```
bool Person::operator<(const Person& p) const {
  cout << "operateur <" << endl;
  return age < p.age;
}
```

c'est-à-dire en choisissant l'age comme critère de comparaison, cela donne :

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestSet1.cpp Person.o -o TestSet1
[student@moon TemplateSTL]$ TestSet1
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
constructeur de copie
```

```
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
operateur <
constructeur de copie
destructeur
Contenu :
--> wagner (37)
--> leonard (38)
--> caprasse (43)
--> hiard (44)
--> wagner (50)
--> quettier (51)
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

où on observe à présent que

- le conteneur est trié selon l'age
- « wagner » est maintenant présent deux fois dans le conteneur étant donné que les deux objets « wagner » ne sont pas équivalents vu qu'ils n'ont pas le même age.

La recherche dans un conteneur set<T>

Pour rechercher un objet situé dans un conteneur **set<T>**, on doit utiliser la méthode membre **sort()** de la classe set<T>. Cette méthode ne prend aucun paramètre et

- retourne un itérateur pointant sur l'objet trouvé si l'objet recherché est trouvé
- retourne **set<T>::end()** si l'objet recherché n'est pas trouvé.

Comme pour l'insertion (et donc le tri), la recherche se fait en utilisant uniquement l'**opérateur <** de la classe template et non son opérateur ==.

Nous reprenons ici l'exemple de la classe **set<Person>** où l'**opérateur <** de la classe Person compare uniquement le nom (variable **name**). Voici le programme de test (**TestSet2.cpp**) :

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

void affiche(const set<Person>& v);

int main() {
  set<Person> lst;  // liste triée vide

  lst.insert(Person("wagner",50));
  lst.insert(Person("leonard",38));
  lst.insert(Person("quettier",51));
  lst.insert(Person("wagner",37));
  lst.insert(Person("caprasse",43));
  lst.insert(Person("hiard",44));

  affiche(lst);

  Person search1;
  search1.setName("quettier");
  auto it = lst.find(search1);
  if (it != lst.end()) {
    cout << "quettier trouvé ! --> " << it->toString() << endl;
  } else {
    cout << "quettier non trouvé..." << endl;
  }

  Person search2;
  search2.setName("madani");
  auto it = lst.find(search2);
  if (it != lst.end()) {
    cout << "madani trouvé ! --> " << it->toString() << endl;
  } else {
    cout << "madani non trouvé..." << endl;
  }

  return 0;
}

void affiche(const set<Person>& l) {
  cout << "Contenu : " << endl;
  for (set<Person>::const_iterator it=l.cbegin() ; it!=l.cend() ; it++) {
```

```
    cout << "--> " << it->toString() << endl;
  }
}
```

dont l'exécution fournit

```
[student@moon TemplateSTL]$ g++ Person.cpp -c
[student@moon TemplateSTL]$ g++ TestSet2.cpp Person.o -o TestSet2
[student@moon TemplateSTL]$ TestSet2
constructeur d'initialisation
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
constructeur de copie
destructeur
constructeur d'initialisation
operateur <
operateur <
operateur <
operateur <
operateur <
constructeur de copie
destructeur
Contenu :
--> caprasse (43)
--> hiard (44)
--> leonard (38)
--> quettier (51)
--> wagner (50)
constructeur par defaut
operateur <     // Début recherche
```

```
operateur <
operateur <
operateur <     // Fin recherche
quettier trouvé ! --> quettier (51)
constructeur par defaut
operateur <     // Début recherche
operateur <
operateur <
operateur <     // Fin recherche
madani non trouvé...
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
destructeur
[student@moon TemplateSTL]$
```

Bien d'autres conteneurs et outils…

Nous n'avons abordé ici que quelques-uns des conteneurs templates et des outils associés de la STL. Il y en a d'autres comme

- **stack<T>** → le pile
- **queue<T>** → la file
- **map<T,U>** → conteneur associatif (nous aurons l'occasion de revenir dessus plus tard)
- …

Mais le lecteur a à présent toutes les bases nécessaires pour les aborder de lui-même 😉