Au cours de cette étude, nous allons aborder un certain nombre de concepts relativement sophistiqués qui vont nous permettre de conclure notre apprentissage sur le langage C++ notamment sur sa dernière version savoir le ***C++11***. Nous allons tout d'abord revoir les conteneurs associatifs avec les algorithmes qui sont souvent très utiles.

Les conteneurs associatifs

Le conteneur ***vector*** que nous connaissons bien est comparable a une version améliorée d'un tableau. Il possède en effet toutes les caractéristiques des tableaux, plus quelques fonctions supplémentaires comme par exemple l'affectation entre tableaux. Il souffre malheureusement également d'un inconvénient significatif des tableaux ; vous ne pouvez pas retrouver un élément autrement que par son indice dans le conteneur. Les conteneurs associatifs, en revanche, offrent un accès direct rapide, reposant sur une association entre des ***clés*** et des ***valeurs***, à l'image du dictionnaire où la ***clé*** représente le mot à rechercher et la ***valeur***, la définition du mot.

Les conteneurs se classent en deux catégories

- ***Les conteneurs séquentiels :*** *sont ordonnés suivant un ordre imposé explicitement par le programme lui-même ; nous accédons à un des éléments en tenant compte de cet ordre au moyen d'un indice.*
- ***Les conteneurs associatifs*** *: ont pour principale vocation de retrouver une information, non plus en fonction de sa place dans le conteneur, mais en fonction de sa valeur ou d'une partie de sa valeur nommée **clé**. L'exemple classique est le répertoire téléphonique, dans lequel nous retrouvons le numéro de téléphone à partit d'une **clé** formée du nom de la personne concernée. Malgré tout, pour de simples questions d'efficacités, un conteneur associatif se trouve **ordonné** intrinsèquement en permanence, en se fondant sur une relation (par défaut **<**) choisie à la construction.*

Les deux conteneurs associatifs les plus importants sont map et multimap. Ils correspondent pleinement au concept de conteneurs associatifs, associant une clé et une valeur. Mais, alors que map impose l'unicité des clés (comme pour les clés primaires d'une base de données), autrement dit l'absence de deux éléments ayant la même clé, multimap ne l'impose pas et nous pourrons y trouver plusieurs éléments de même clé qui apparaîtrons alors consécutivement. Si nous reprenons l'exemple du répertoire téléphonique, nous pouvons dire que multimap autorise la présence de plusieurs personnes de même nom (avec des numéros associés différents ou non), tandis que map ne l'autorise pas.

Le conteneur *map*

Maintenant que nous connaissons les prérogatives des conteneurs associatifs, c'est très souvent le conteneur ***map*** qui est très largement utilisé vu justement le caractère d'unicité des ***clés***. Par ailleurs, rien empêche d'avoir des ***valeurs*** avec une structure relativement sophistiquée, les ***clés*** doivent demeurer relativement simples afin de permettre une recherche facile par la suite.

Une fonctionnalité vraiment intéressante avec le conteneur map est la possibilité d'utiliser l'opérateur [ ] pour introduire ou récupérer une valeur avec pour indice la clé correspondante. Par exemple, avec un conteneur nommé annuaire, dans lequel les clés sont des chaînes, nous pourrons utiliser l'expression annuaire[« Dudand »] pour désigner l'élément correspondant à la clé « Durand ».

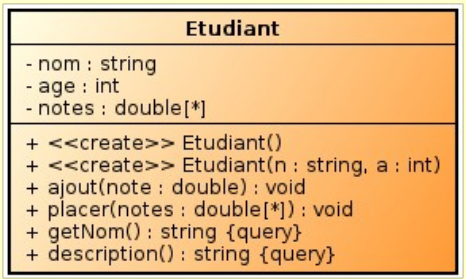
Le conteneur ***map*** est donc formé d'un ensemble d'éléments composés de deux parties : une ***clé*** et une ***valeur***. Pour représenter de tels éléments, il existe un modèle de classe approprié, nommé ***pair***, paramétré par le type de la ***clé*** et par celui de la ***valeur*** - ***pair<string, double>*** par exemple, où la ***clé*** est de type ***chaîne*** et la ***valeur*** de type ***réel***.

Afin de pouvoir récupérer séparément la clé et la valeur de l'élément, la classe paramétrée pair dispose de deux attributs publics nommés respectivement : first qui représente la clé, et second qui correspond à la valeur associée.

Après toutes ces considérations techniques, voici un exemple qui va nous permettre de bien comprendre toutes les fonctionnalités associées à ce type de conteneur.

principal.cpp

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

class Etudiant
{
  string nom;
  int age;
  vector<double> notes;
public:
  Etudiant() = default;

  Etudiant(const string& n, int a) : nom(n), age(a) {}

  void ajout(double note)              { notes.push_back(note); }

  void placer(const vector<double>& n) { notes = n; }

  string getNom() const                { return nom; }

  string description() const
  {
     ostringstream info;
     info << nom << ", " << age << " ans, [";
     for (double note : notes) info << note << ' ';
     info << "\b]";
     return info.str();
  }
};
```



```
int main()
{
  map<string, Etudiant> etudiants;

  Etudiant marcel("Marcel", 18);
  marcel.placer({12.5, 8.5, 15.0});
  etudiants[marcel.getNom()] = marcel;

  Etudiant alice("Alice", 19);
  alice.placer({18.0, 5.5, 12.0, 11.5});
  etudiants[alice.getNom()] = alice;

  Etudiant bruno("Bruno", 22);
  bruno.placer({12.5, 9.0, 13.0, 11.0, 7.5});
  etudiants[bruno.getNom()] = bruno;

  cout << etudiants["Marcel"].description() << endl;

  for (pair<string, Etudiant> etudiant : etudiants) cout << etudiant.second.description() << endl;

  return 0;
}
```

```
Marcel, 18 ans, [12.5 8.5 15]
Alice, 19 ans, [18 5.5 12 11.5]
Bruno, 22 ans, [12.5 9 13 11 7.5]
Marcel, 18 ans, [12.5 8.5 15]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Inférence de type : auto

Jusqu'à présent, pour déclarer une variable en C++, il était obligatoire de spécifier son type. Cependant, avec l'arrivée des templates, cela n'est pas toujours aisé, notamment pour les valeurs de retour des fonctions templates. De même il est parfois pénible de déterminer de type de certaines variables que nous souhaitons sauvegarder, oblige à fouiller loin dans les définitions des classes.

Grâce au mot clé auto, C++11 permet de déclarer une variable sans en connaître le type, à condition toutefois de l'initialiser explicitement. Le type est ainsi spécifié implicitement par la valeur d'initialisation. Ainsi, par exemple, dans le code précédent, dans la boucle for(:), c'est beaucoup plus agréable d'utiliser auto à la place de pair<string, Etudiant>.

```
  cout << etudiants["Marcel"].description() << endl;
  for (auto etudiant : etudiants) cout << etudiant.second.description() << endl;
```

Déclaration de variables multiples dans une même entité – les tuples - C++17

La déclaration de variables multiples dans une même entité rend le code beaucoup plus « **léger** ». Cette possibilité d'assignation multiple pour une même entité est aussi appelée **tuple**. Grâce à ce mécanisme, dans une fonction ou une méthode, nous avons enfin l'opportunité de renvoyer plusieurs valeurs en même temps puisque les **tuples** factorisent ces valeurs dans un même élément (avec éventuellement plusieurs variables de natures différentes).

Ces tuples permettent ainsi de séparer l'entité à scruter avec des variables uniques exprimées entre crochet. Tous les types composés peuvent utiliser cette technique de simplification. Par ailleurs, toujours avec la même syntaxe de déclaration nous pouvons séparer les différents éléments constituant le tuple renvoyé par une fonction. Par contre, dans la fonction, si vous désirez factoriser plusieurs variables dans une même entité, vous devez passer par la classe tuple.

```
#include <iostream>
#include <tuple> // utile uniquement pour utiliser la classe
using namespace std;

auto reel(double nombre)
{
  int entiere=nombre;
  double decimale=nombre-entiere;
  return tuple(entiere, decimale);
}

struct Personne
{
  string nom, prenom;
  int age;
};

int main()
{
  int tableau[] =  {-12, 15, 23};
  Personne lui = {"PÊCHEUR", "Martin", 23};
  pair<string, int> elle = {"Alice", 19};
  double nombre=5.8;

  auto [un, deux, trois] = tableau;
  auto [n, p, a] = lui;
  auto [prenom, age] = elle;
  auto [entiere, decimale] = reel(nombre);

  cout << "Tableau : " << un << ' ' << deux << ' ' << trois << endl;
  cout << "lui : " <<  n << ' ' << p << ' ' << a << endl;
  cout << "elle : " << prenom << ' ' << age << endl;
  cout << nombre << " = " << entiere << " + " << decimale << endl;

  return 0;
}
```

```
Tableau : -12 15 23
lui : PÊCHEUR Martin 23
elle : Alice 19
5.8 = 5 + 0.8
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

En reprenant le sujet précédent, nous pouvons discriminer pair<string, Etudiant> en séparant le nom de l'étudiant de sa structure globale (ce qui évite d'utiliser les termes first et second qui ne sont pas spécialement intuitifs).

```
  cout << etudiants["Marcel"].description() << endl;
  for (auto [nom, etudiant] : etudiants) cout << etudiant.description() << endl;
```

Recherche et suppression

Il est souvent très utile lorsque nous manipulons des collections, de pouvoir introduire de nouvelles valeurs, ce que nous venons de faire, mais également de retrouver une valeur déjà introduite et aussi de pouvoir retirer certains éléments.

1. *Si vous savez que l'élément a déjà été introduit, comme nous l'avons déjà expérimenté, l'opérateur **[ ]** convient tout-à-fait, il est d'ailleurs très simple à utiliser.*
2. *Dans le cas où vous désirez savoir si un élément existe déjà dans la collection, il est alors préférable de choisir la méthode **find()** qui renvoie un itérateur (pointeur) sur l'élément possédant la clé donnée en argument. Si aucun élément n'est trouvé, cette méthode renvoie l'itérateur **end()**. Malheureusement, il n'existe pas de méthode **exist()**.*
3. *Grâce à la méthode **erase()**, il est possible de supprimer un élément de la collection au moyen de la clé spécifiée par l'argument de la méthode.*

Modification du code précédent

```
int main()
{
  map<string, Etudiant> etudiants;

  Etudiant marcel("Marcel", 18);
  marcel.placer({12.5, 8.5, 15.0});
  etudiants[marcel.getNom()] = marcel;

  Etudiant alice("Alice", 19);
  alice.placer({18.0, 5.5, 12.0, 11.5});
  etudiants[alice.getNom()] = alice;

  Etudiant bruno("Bruno", 22);
  bruno.placer({12.5, 9.0, 13.0, 11.0, 7.5});
  etudiants[bruno.getNom()] = bruno;

  cout << etudiants["Marcel"].description() << endl;
  etudiants.erase("Marcel");

  auto etudiant = etudiants.find("Marcel");
  cout << "L'étudiant Marcel " << (etudiant!=etudiants.end() ? "existe!" : "n'existe plus!") << endl;

  etudiant = etudiants.find("Bruno");
  cout << "L'étudiant Bruno " << (etudiant!=etudiants.end() ? "existe!" : "n'existe pas!") << endl;
  cout << etudiant->second.description() << endl;

  for (auto [nom, etudiant] : etudiants) cout << etudiant.description() << endl;

  return 0;
}
```

```
Marcel, 18 ans, [12.5 8.5 15]
L'étudiant Marcel n'existe plus!
L'étudiant Bruno existe!
Bruno, 22 ans, [12.5 9 13 11 7.5]
Alice, 19 ans, [18 5.5 12 11.5]
Bruno, 22 ans, [12.5 9 13 11 7.5]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Les algorithmes standards

Lorsque nous choisissons une collection, cette dernière possède pas mal de méthodes intéressantes, très souvent en relation avec le type de collection que nous avons choisi. Toutefois, afin de ne pas surcharger ces classes spécifiques, toutes les fonctionnalités ne sont pas intégrées. Afin de combler certaines lacunes, il existe des fonctions génériques qui s'adaptent à toutes les collections quelque soit leurs natures.

Je rappelle que chaque type de conteneur fournit deux méthodes : begin() qui retourne un itérateur sur le premier élément du conteneur, end() qui retourne un itérateur sur un élément (le prochain) après le dernier élément du conteneur. Ces itérateurs sont fondamentaux pour les algorithmes puisque ces fonctions sont opérationnelles quelque soit le type de conteneur, et ceux sont ces itérateurs qui fixent les limites du traitement souhaité.

Voici une liste non exhaustive de quelques algorithmes qui s'avèrent bien utiles dans pas mal de situation. La plupart sont plus adaptés à des conteneurs séquentiels, comme **vector**, plutôt qu'avec des conteneurs associatifs, comme **map**. La suite nous donne un exemple concret de gestion de notes au travers d'un conteneur de type **vector**.

- ✔ ***accumulate()** : calcule la somme des éléments d'une séquence avec une valeur initiale (éventuellement donnée par une fonction).*
- ✔ ***find()** : cherche le premier élément égal à une valeur dans une séquence. Retourne un itérateur (pointeur).*
- ✔ ***find_if()** : cherche le premier élément vérifiant un test donné. Retourne un itérateur (pointeur).*
- ✔ ***for_each()** : applique une fonction/foncteur sur tous les éléments d'une séquence.*
- ✔ ***generate()** : initialise une séquence à l'aide d'un générateur de valeurs (fonction/foncteur).*
- ✔ ***max_element()** : récupère le plus grand élément d'une séquence. Retourne un itérateur (pointeur).*
- ✔ ***min_element()** : récupère le plus petit élément d'une séquence. Retourne un itérateur (pointeur).*
- ✔ ***sort()** : trie une séquence.*
- ✔ ***transform()** : transforme une séquence en une autre.*

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

int main()
{
    vector<double> notes = {12.5, 18.0, 8.5, 15.0, 5.0};
```

```
Notes = [5 8.5 12.5 18]
Moyenne = 11
Maxi = 18
Mini = 5
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
    auto endroit = find(notes.begin(), notes.end(), 15.0);
    bool existe =  endroit != notes.end();
    if (existe) notes.erase(endroit);

    double moyenne = accumulate(notes.begin(), notes.end(), 0.0) / notes.size();
    double maxi = *max_element(notes.begin(), notes.end());
    double mini = *min_element(notes.begin(), notes.end());

   sort(notes.begin(), notes.end());

    cout << "Notes = [";
    for (double note : notes) cout << note << ' ';
    cout << "\b]" << endl;
    cout << "Moyenne = " << moyenne << endl;
    cout << "Maxi = " << maxi << endl;
    cout << "Mini = " << mini << endl;

    return 0;
}
```

Quatre algorithmes n'ont pas été utilisés dans cet exemple – generate(), for_each(), transform() et find_if(). Ils sont un petit peu particulier dans la mesure où ils ont besoin d'une fonction annexe (ou un objet fonction – foncteur) pour permettre leurs exploitations. C'est avec ce type d'algorithme que les foncteurs et les expressions lambda vont avoir toutes leurs significations.

Utilisation de fonctions classiques associées avec les algorithmes

Il est fréquent que nous ayons besoin d'initialiser un conteneur par des valeurs résultant d'un calcul. La bibliothèque standard offre un outil assez général à cet effet, à savoir ce que nous nommons souvent un algorithme générateur, avec la fonction ***generate()***. Nous lui fournissons, en argument, un objet fonction (***foncteur*** – il peut s'agir aussi d'une fonction ordinaire) qu'il appellera automatiquement pour déterminer la valeur à attribuer à chaque élément de la collection. Pour ce type algorithme une telle fonction ne reçoit aucun argument.

Pour vérifier cet algorithme, nous allons réaliser un programme qui permet de générer la progression des puissances de deux sur une collection de nombres entiers.

Un autre algorithme très souvent utilisé est la fonction ***for_each()*** qui permet de parcourir une collection et d'effectuer un traitement spécifique à chacun des éléments successifs à l'aide d'un ***foncteur*** (ou fonction classique) passé en argument. Cette fois-ci, pour ce type d'algorithme, cette fonction devra possédée un argument du même type que les éléments de la collection. Attention, cet algorithme n'est pas prévu pour modifier le contenu de la collection, c'est l'algorithme ***transform()*** qui permet de le faire.

Afin de vérifier les fonctionnalité de cet algorithme, nous en profiterons pour afficher la suite des éléments de la collection des puissances de deux ainsi que la collection représentant l'ensemble des notes.

L'algorithme ***transform()*** permet de modifier une collection après coup à l'aide d'une fonction qui prend un seul argument et qui retourne une valeur du même type que la collection. Il est également possible de réaliser des traitements spécifiques à l'aide de deux collections de même capacité, par exemple additionner le contenu de deux tableaux, élément par élément. Dans ce cas là, il sera nécessaire d'avoir deux arguments pour la fonction de traitement.

Durant le test d'évaluation, nous prévoyons deux transformations, d'une part, l'ensemble des notes seront augmentées de une unité, d'autre part, nous ferons une fusion de deux collections d'entiers avec un traitement spécifique qui sera répercuté sur la deuxième suite. Voici les deux signatures possibles pour cet algorithme transform().

transform(début_première, fin_première, début_résultat, fonction_traitement)
transform(début_première, fin_première, début_seconde, début_résultat, fonction_traitement)

Enfin, il est possible de réaliser des recherches avec une condition particulière adaptée à la situation actuelle à l'aide d'une fonction qui retourne un booléen associée à l'algorithme ***find_if()***.

Toujours à titre d'exemple, nous rechercherons la première note au dessus de 10 dans la collection stockant l'ensemble des notes d'un étudiant.

algorithme.cpp

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

int suite()
{
    static double n = 0.5;
    return n*=2;
}

void affiche(const auto& valeur) // C++14 uniquement
{
  cout << valeur << ' ';
}

bool bonneNote(const int& note)
{
  return note > 10;
}

double plusUn(const double& premier)
{
  return premier + 1.0;
}
```

```
Progression binaire : 1 2 4 8 16 32 64 128 256 512 1024
Première note supérieure à dix : 12.5
Suite des notes : 5 8.5 12.5 15 18
Suite des notes (+1) : 6 9.5 13.5 16 19
Suite entière : 1 2 3 5 9 17 33 65 129 257 513
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
int somme(const int& premier, const int& deuxieme)
{
    return premier/2 + deuxieme;
}

int main()
{
    vector<int> binaire(11, 0); // vecteur de 11 entiers initialisés à 0
    vector<double> notes = {12.5, 18.0, 8.5, 15.0, 5.0};

    cout << "Progression binaire : ";
    generate(binaire.begin(), binaire.end(), suite);
    for_each(binaire.begin(), binaire.end(), affiche<int>);

    cout << endl << "Première note supérieure à dix : ";
    cout << *find_if(notes.begin(), notes.end(), bonneNote) << endl;

    sort(notes.begin(), notes.end());
    cout << "Suite des notes : ";
    for_each(notes.begin(), notes.end(), affiche<double>);

    transform(notes.begin(), notes.end(), notes.begin(), plusUn);
    cout << endl << "Suite des notes (+1) : ";
    for_each(notes.begin(), notes.end(), affiche<double>);

    vector<int> entiers(11, 1);  // vecteur de 11 entiers initialisés à 1
    transform(binaire.begin(), binaire.end(), entiers.begin(), entiers.begin(), somme);
    cout << endl << "Suite entière : ";
    for_each(entiers.begin(), entiers.end(), affiche<int>);
    cout << endl;
    return 0;
}
```

Tous les algorithmes standard de C++ sont définis dans le fichier référencé par la directive <algorithm>. Chacun de ces algorithmes sont appelés pour parcourir chaque élément de la collection souhaitée. À chacun de ces éléments est appliqué un traitement spécifique associé à la fonction de rappel. Cette dernière, mis à part l'algorithme generate() doit posséder au moins un argument correspondant au type de l'élément à traiter. Seul l'algorithme transform() peut utiliser une fonction de traitement avec deux arguments puisqu'il est possible de travailler conjointement avec deux collections de même nature.

Les foncteurs - objets fonction

Ce que nous venons de réaliser dans la section précédente est déjà très performant. Il existe toutefois quelques lacunes. Nous constatons qu'il est difficile d'imposer une valeur initiale (valeur de départ) pour une suite de nombres, autrement qu'en la fixant définitivement dans la fonction elle-même.

Si nous reprenons l'exemple de l'algorithme generate(), il n'est pas possible de choisir une valeur de départ en argument de la fonction de traitement. Il faudrait alors proposer deux (ou plusieurs) fonctions séparées pour imposer une valeur initiale à chaque cas de figure.

C'est précisément dans ce genre de situation que la notion d'objet fonction (***foncteur***) prend tout son sens. Le foncteur est un objet qui se comporte comme une fonction. En réalité, il s'agit d'une classe dans laquelle est (notamment) redéfini l'opérateur d'appel de fonction, ***operator()***. Le foncteur étant une classe, il peut conserver un ou plusieurs états grâce à ces attributs éventuels, ce qui le rend beaucoup plus puissant qu'une simple fonction. Les foncteurs sont ainsi très utiles en combinaison avec les algorithmes de la librairie STL.

Dans la perspective de leur utilisation avec les conteneurs de la librairie STL, les foncteurs se rangent dans deux catégories :

- ✔ ***Foncteurs unaires*** *: trie une séquence.l'opérateur* ***()*** *appelé avec un seul argument, comme* ***f(x)****. Lorsque le type de valeur envoyée est booléen, le foncteur s'appelle un* ***prédicat****.*
- ✔ ***Foncteurs binaires*** *: l'opérateur* ***()*** *appelé avec deux arguments, comme* ***f(x, y)****. Lorsque le type de valeur envoyée est booléen, le foncteur s'appelle un* ***prédicat binaire****.*

Les prédicats sont naturellement adaptés à être transmis en tant qu'arguments des algorithmes de conteneurs qui ont besoin de prendre des décisions. Un foncteur qui en combine deux autres est appelé foncteur adaptatif.

Dans l'exemple qui suit, nous allons générer deux suites numériques entières dont la progression est de une unité à chaque fois avec la possibilité de choisir la valeur du premier élément. Nous fabriquons deux foncteurs, ***Suite*** spécialisé pour la génération et ***Affiche*** spécialisé pour l'affichage des suites avec la possibilité de choisir un préfixe. Ce dernier, emmagasine également le nombre d'éléments déjà affichés. Dans un deuxième temps, nous rajoutons un autre foncteur ***Notes*** qui est capable d'afficher un ensemble de notes et de calculer automatiquement la moyenne.

foncteurs.cpp

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

class Suite
{
  int n;
public:
  Suite(int i) : n(i) {}
  int operator()() { return n++; }
};
```

```
Suite numérique : 0 1 2 3 4 5 6 7 8 9 10 [11 valeurs]
Autre suite : 5 6 7 8 9 10 11 [7 valeurs]
Notes : 12.5 18 8.5 15 5 [moyenne = 11.8]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
class Affiche
{
  int n = 0;
public:
  Affiche(const string& prefixe) { cout << prefixe << " : "; }
  void operator()(const int& element)
  {
    cout << element << ' ';
    n++;
  }
  int nombre() { return n; }
};
class Notes
{
  double somme = 0.0;
  int nombre = 0;
public:
  Notes() { cout << "Notes : "; }
  void operator()(const double& note)
  {
    cout << note << ' ';
    somme += note;
    nombre++;
  }
  double moyenne() const { cout << "[moyenne = " << somme/nombre << ']' << endl; }
};

int main()
{
    vector<int> premiers(11);
    vector<int> seconds(7);
    vector<double> notes = {12.5, 18.0, 8.5, 15.0, 5.0};

    generate(premiers.begin(), premiers.end(), Suite(0));
    generate(seconds.begin(), seconds.end(), Suite(5));

    Affiche resultat = for_each(premiers.begin(), premiers.end(), Affiche("Suite numérique"));
    cout << '[' << resultat.nombre() << " valeurs]" << endl;

    resultat = for_each(seconds.begin(), seconds.end(), Affiche("Autre suite"));
    cout << '[' << resultat.nombre() << " valeurs]" << endl;

    for_each(notes.begin(), notes.end(), Notes()).moyenne();

    return 0;
}
```

La particularité de l'algorithme for_each() est qu'il retourne systématiquement le foncteur placé dans le troisième argument de cette fonction, ce qui est très utile pour récupérer l'évolution et les changements d'états du foncteur.

Prédicat simple (unaire)

Comme nous l'avons évoqué dans la section précédente, un foncteur qui renvoie un résultat de type booléen s'appelle une prédicat, service très prisé dans les algorithmes de la STL. Dans l'exemple qui suit, le foncteur ***EstMultiple*** permet de déterminer si un nombre est multiple d'une valeur (diviseur) choisie au préalable.

prédicat.cpp

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

class EstMultiple
{
  int diviseur;
public:
  EstMultiple(int div) : diviseur(div) {}
  bool operator()(const int& n) { return n % diviseur == 0; }
};

class Suite
{
  int n;
public:
  Suite(int i) : n(i) {}
  int operator()() { return n++; }
};

int main()
{
    vector<int> entiers(20);

    generate(entiers.begin(), entiers.end(), Suite(9));
    cout << "Premier élément divisible par 7 => ";
    cout << *find_if(entiers.begin(), entiers.end(), EstMultiple(7)) << endl;

    return 0;
}
```

```
Premier élément divisible par 7 => 14
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Les lambdas (C++11)

Nous venons de voir que les foncteurs offrent une très grande richesse pour exploiter pleinement les compétences des algorithmes de la STL. Seulement, ils peuvent être un petit peu lourd à construire. Les expressions lambdas (fonctions anonymes) proposent justement une solution beaucoup plus compacte pour utiliser les foncteurs (anonymes). Elles constituent une nouveauté de C++11. C++14 permet maintenant de faire des fonctions polymorphes (anonymes ou pas) grâce à l'introduction de la déduction du type de retour (**auto**). Ces lambdas peuvent alors être nommés.

Vous comparez l'expression lambda à une version compacte d'une classe anonyme dotée d'un opérateur () à accès public. C'est donc un foncteur particulier. Grâce aux expressions lambdas, tout le bloc de définition du foncteur disparaît au profit du lambda inline.

Une expression lambda possède systématiquement trois parties symbolisées par des opérateurs spécifiques :

- `[]` : la liste de capture : vide ou non, stipule quelles variables environnantes (extérieures), représentant autant d'états, doivent être atteintes, par valeur ou par référence. Voici quelques exemples : `[]` ne capture rien, `[=]` capture tout par copie, `[=, &x]` capture tout par copie, sauf x par référence, `[&]` capture tout par référence, `[&, x]` capture tout par référence, sauf pour x.
- `()` : la liste des paramètres : se présente comme celle requise pour définir l'opérateur () des foncteurs.
- `{}` : délimiteur du corps : peut englober plusieurs instructions séparées par des points virgules. Cela correspond au traitement spécifique qui sera effectué pour chacun des éléments de la collection.

Équivalence entre l'expression lambda et le foncteur correspondant

```
[varExt1, varExt2] (Type1 var1, Type2 var2) { instruction1 ; instruction2 ; return (valeur ou expression) ; } // ou bien

[varExt1, varExt2] (Type1 var1, Type2 var2) -> TypeRetour (plus nécessaire en C++14)
{
  instruction1 ;
  instruction2 ;
  return (valeur ou expression) ;
}
Foncteur équivalent

class Foncteur
{
  TypeExt1 varExt1;
  TypeExt2 varExt2;
public:
  Foncteur(TypeExt1 in1, TypeExt2 in2) : varExt1(in1), varExt2(in2) {}
  TypeRetour operator()(Type1 var1, Type2 var2)
  {
    instruction1;
    instruction2;
    return (valeur ou expression);
  }
};
```

À titre d'exemple, je vous propose de reprendre les deux projets précédents que nous avons élaborés dans la section traitant des foncteurs et de voir l'équivalent avec une écriture exclusivement faite avec des expressions lambdas.

lambdas.cpp

```
Suite numérique : 0 1 2 3 4 5 6 7 8 9 10 [11 valeurs]
Autre suite : 5 6 7 8 9 10 11 [7 valeurs]
Notes : 12.5 18 8.5 15 5 [moyenne = 11.8]
Premier élément divisible par 7 => 14
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

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

int main()
{
    vector<int> premiers(11), seconds(7), entiers(20);
    vector<double> notes = {12.5, 18.0, 8.5, 15.0, 5.0};

    int initial = 0, nombre = 0;
    auto creation = [&initial] () { return initial++; };
    auto suite = [&nombre] (const int &n) { nombre++; cout << n << ' '; };

    generate(premiers.begin(), premiers.end(), creation);
    initial = 5;
    generate(seconds.begin(), seconds.end(), creation);
    cout << "Suite numérique : ";
    for_each(premiers.begin(), premiers.end(), suite);
    cout << '[' << nombre << " valeurs]" << endl;
    cout << "Autre suite : ";
    nombre = 0;
    for_each(seconds.begin(), seconds.end(), suite);
    cout << '[' << nombre << " valeurs]" << endl;
    cout << "Notes : ";
    nombre = 0;
    double somme = 0.0;
    for_each(notes.begin(), notes.end(), [&nombre, &somme](double note) { cout << note << ' '; nombre++; somme+=note; });
    cout << "[moyenne = " << somme/nombre << ']' << endl;

    initial = 9;
    generate(entiers.begin(), entiers.end(), creation);
    int diviseur = 7;
    cout << "Premier élément divisible par 7 => ";
    cout << *find_if(entiers.begin(), entiers.end(), [diviseur](const int &n) { return n % diviseur == 0; }) << endl;
    return 0;
}
```

Notre code est maintenant beaucoup plus compact, c'est vraiment l'intérêt des lambdas. Pour des traitements relativement simples, ils sont tout à fait adaptés. Les foncteurs toutefois peuvent intégrer beaucoup plus de fonctionnalités. Si vous devez réaliser des traitements courts et simples, avec une capture préalable, le lambda est alors intéressant puisque sa syntaxe est vraiment très concise. Dans le cas d'une initialisation systématique, comme nous le voyons dans le code précédent, le foncteur me paraît plus adapté. Enfin, si vous n'avez pas besoin de capture, une simple fonction suffit. Finalement, les trois types d'écritures peuvent parfaitement coexister.

Exemple qui permet d'utiliser les lambdas et les foncteurs

```
class Affiche
{
  int nombre = 0;
public:
  Affiche(char* prefixe) { cout << prefixe << " : "; }
  void operator()(const auto& element) { cout << element << ' '; nombre++; }
  void fin() const { cout << '[' << nombre << " valeurs]\n"; }
};

class Increment
{
  int initial;
public:
  Increment(int initial) : initial(initial) {}
  int operator ()() { return initial++; }
};

int main()
{
    vector<int> premiers(11), seconds(11), entiers(20);
    vector<double> notes = {12.5, 18.0, 8.5, 15.0, 5.0};
    int nombre = 0; double somme = 0.0;
    auto moyenne = [&nombre, &somme](double& note) { cout << note << ' '; nombre++; somme+=note; };
    generate(premiers.begin(), premiers.end(), Increment(0));
    generate(seconds.begin(), seconds.end(), Increment(5));
    for_each(premiers.begin(), premiers.end(), Affiche("Suite numérique")).fin();
    for_each(seconds.begin(), seconds.end(), Affiche("Autre suite")).fin();

    cout << "Notes : ";  for_each(notes.begin(), notes.end(), moyenne);
    cout << "[moyenne = " << somme/nombre << ']' << endl;

    generate(entiers.begin(), entiers.end(), Increment(9));
    cout << "Premier élément divisible par 7 => ";
    cout << *find_if(entiers.begin(), entiers.end(), [diviseur=7](const int &n) { return n % diviseur == 0; }) << endl;

    transform(premiers.begin(), premiers.end(), seconds.begin(), premiers.begin(), [](int& x, int& y) { return x+y; });
    for_each(premiers.begin(), premiers.end(), Affiche("Somme des deux")).fin();
}
```

```
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
Suite numérique : 0 1 2 3 4 5 6 7 8 9 10 [11 valeurs]
Autre suite : 5 6 7 8 9 10 11 12 13 14 15 [11 valeurs]
Notes : 12.5 18 8.5 15 5 [moyenne = 11.8]
Premier élément divisible par 7 => 14
Somme des deux : 5 7 9 11 13 15 17 19 21 23 25 [11 valeurs]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Deux classes sont prévues pour ce code, Affiche et Increment vues qu'elles sont utilisées plusieurs fois. Remarquez qu'il est possible de déclarer une variable en l'initialisant directement dans la capture du lambda « [diviseur=7] » ce qui permet d'être encore plus concis (C++14). Enfin, pour le dernier lambda « [](int& x, int& y) { return x+y; } », aucune capture n'est sollicité. Nous aurions pu déclarer une fonction classique en amont mais, dans notre cas, comme ce traitement n'est exécuté qu'une seule fois, il me paraît plus judicieux d'utiliser l'expression lambda. C'est justement l'intérêt de ce type d'expression, très avantageux pour une utilisation ponctuelle.

Pour conclure sur ce sujet, je vous propose de réaliser un dernier projet qui permet d'enregistrer des étudiants et de les classer par ordre alphabétique ou par leurs moyennes.

etudiants.cpp

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

class Etudiant
{
  string nom;
  int age;
  vector<double> notes;
public:
  Etudiant(const string& nom, int a, const vector<double>& n) : nom(nom), age(a), notes(n) {}
  string getNom() const { return nom; }
  double moyenne() const
  {
    int nombre = 0;
    double somme = 0.0;
    for_each(notes.begin(), notes.end(), [&nombre, &somme](double note) { nombre++; somme+=note; });
    return somme/nombre;
  }
  string description() const
  {
    ostringstream info;
    info << nom << ", " << age << " ans, [";
    for (double note : notes) info << note << ' ';
    info << "\b], moyenne = " << moyenne();
    return info.str();
  }
};
```

```
Liste alphabétique :
Alice, 19 ans, [18 5.5 12 11.5], moyenne = 11.75
Bruno, 22 ans, [12.5 9 13 11 7.5], moyenne = 10.6
Marcel, 18 ans, [12.5 8.5 15 17.5], moyenne = 13.375
Meilleure moyenne :
Marcel, 18 ans, [12.5 8.5 15 17.5], moyenne = 13.375
Alice, 19 ans, [18 5.5 12 11.5], moyenne = 11.75
Bruno, 22 ans, [12.5 9 13 11 7.5], moyenne = 10.6
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
void description(const Etudiant& etudiant) { cout << etudiant.description() << endl; }

int main()
{
  vector<Etudiant> etudiants;
  etudiants.push_back(Etudiant("Marcel", 18, {12.5, 8.5, 15.0, 17.5}));
  etudiants.push_back(Etudiant("Alice", 19, {18.0, 5.5, 12.0, 11.5}));
  etudiants.push_back(Etudiant("Bruno", 22, {12.5, 9.0, 13.0, 11.0, 7.5}));

  cout << "Liste alphabétique :" << endl;
  sort(etudiants.begin(), etudiants.end(), [](const Etudiant& x, const Etudiant& y)
  {
    return x.getNom() < y.getNom();
  });
  for_each(etudiants.begin(), etudiants.end(), description);

  cout << "Meilleure moyenne :" << endl;
  sort(etudiants.begin(), etudiants.end(), [](const Etudiant& x, const Etudiant& y)
  {
    return x.moyenne() > y.moyenne();
  });
  for_each(etudiants.begin(), etudiants.end(), description);

  return 0;
}
```

Fabriquer une fonction (algorithme) qui appelle d'autres fonctions

Malgré la programmation orienté objet et son paradigme vraiment passionnant, vous remarquez que quelque soit les techniques à implémenter, dans pas mal de situations, nous avons besoin de fonctions, de foncteurs ou d'expressions lambdas qui permettent de résoudre, de façon très concise, un certain nombre de traitements spécifiques.

Toutefois, une question qui peut se poser, c'est de savoir comment sont implémentés les objets (comme thread) ou les fonctions (comme l'algorithme for_each()) qui font appel à une autre fonction (foncteur ou lambda) passée en argument. Contrairement à ce que nous pourrions penser, la solution est relativement simple, il suffit de fabriquer un type variable (template), correspondant à la fonction de rappel, à l'aide d'un modèle de fonction (fonction générique).

Afin de mieux comprendre ces différents concepts, nous allons réaliser un exemple qui simule le même type de comportement que l'algorithme ***for_each()***. Pour cela, je vous propose de fabriquer un nouvel algorithme personnalisé, une fonction générique ***pour_chaque(),*** qui prend en arguments un vecteur d'entiers et une ***fonction*** (***foncteur*** ou ***lambda***) qui proposera un traitement spécifique sur chacun des éléments de la collection.

Cet algorithme personnalisé recevra un lambda qui permettra d'incrémenter la suite des éléments de la collection, une fonction doubler() qui reprendra chacun des éléments pour doubler leurs valeurs respectives et enfin un foncteur afficher qui s'occupera d'afficher la collection dans son ensemble.

Algorithme personnalisé

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

// Fonction générique qui fait appel à une autre fonction, foncteur ou lambda
template <class Fonction>
void pour_chaque(vector<int>& tableau, Fonction f)
{
  for (int& element : tableau) f(element);
}

// fonction classique
void doubler(int& element) {  element*=2; }

// foncteur
struct afficher
{
  afficher(char* prefixe)        { cout << '[' << prefixe << " = "; }
  void operator() (int& element) { cout << element << ' '; }
  ~afficher()                    { cout << "\b]\n"; }
};

int main()
{
  vector<int> entiers(15);
  int depart = 0;
  // utilisation d'un lambda
  pour_chaque(entiers, [&depart](int &element) { element = ++depart; });
  pour_chaque(entiers, doubler);
  pour_chaque(entiers, afficher("Liste"));
  return 0;
}
```

```
[Liste = 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Algorithme personnalisé avec collection et type variables

```
template <class Type, class Fonction> void pour(Type& tableau, Fonction f)
{
  for (auto& element : tableau) f(element);
}
```

```cpp
template <class Type> struct afficher
{
  afficher(char* prefixe)         { cout << '[' << prefixe << " = "; }
  void operator() (Type& element) { cout << element << ' '; }
  ~afficher()                     { cout << "\b]\n"; }
};

int main()
{
  vector<int> entiers(15);
  list<double> reels(9);
  vector<string> prenoms = {"Bruno", "Marcel", "Alice", "Fabien", "Michèle", "Jean"};

  int depart = 0;
  pour(entiers, [&depart](int& element) { element = ++depart; });
  pour(entiers, [](int& element) { element*=2; });
  pour(entiers, afficher<int>("Entiers"));

  double debut = 0.2;
  pour(reels, [&debut](double& element) { element = debut+=0.3; });
  pour(reels, [](double& element) { element*=1.2; });
  pour(reels, afficher<double>("Réels"));

  pour(prenoms, afficher<string>("Prénoms"));

  return 0;
}
```

```
[Entiers = 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30]
[Réels = 0.6 0.96 1.32 1.68 2.04 2.4 2.76 3.12 3.48]
[Prénoms = Bruno Marcel Alice Fabien Michèle Jean]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Inférence de type : auto en C++14

Grâce aux dernières petites améliorations données par la version **C++14** du langage, souvenez-vous que le fonctionnement du mot clé **auto** a été élargi pour remplacer, dans certaines situations, la syntaxe des templates mais qui propose malgré tout les mêmes fonctionnalités. Le code devient ainsi plus concis et plus simple à comprendre.

Algorithme personnalisé avec collection et type variables

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

void pour(auto& tableau, auto f)
{
  for (auto& element : tableau) f(element);
}
struct afficher
{
  afficher(char* prefixe)         { cout << '[' << prefixe << " = "; }
  void operator() (auto& element) { cout << element << ' '; }
  ~afficher()                     { cout << "\b]\n"; }
};

int main()
{
  vector<int> entiers(15);
  list<double> reels(9);
  vector<string> prenoms = {"Bruno", "Marcel", "Alice", "Fabien", "Michèle", "Jean"};

  int depart = 0;
  pour(entiers, [&depart](int& element) { element = ++depart; });
  pour(entiers, [](int& element) { element*=2; });
  pour(entiers, afficher("Entiers"));

  double debut = 0.2;
  pour(reels, [&debut](double& element) { element = debut+=0.3; });
  pour(reels, [](double& element) { element*=1.2; });
  pour(reels, afficher("Réels"));

  pour(prenoms, afficher("Prénoms"));

  return 0;
}
```

```
[Entiers = 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30]
[Réels = 0.6 0.96 1.32 1.68 2.04 2.4 2.76 3.12 3.48]
[Prénoms = Bruno Marcel Alice Fabien Michèle Jean]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Ce code est beaucoup plus concis que le précédent. Plus aucune utilisation de la syntaxe des templates même si le compilateur traduit le code de la même façon. La fonction pour() devient rudimentaire, ainsi que la classe afficher.

Patrons à nombre de paramètres variables (*variadic template*)

C++11 introduit la possibilité de définir des patrons avec un nombre variable de paramètres nommé ***variadic template***. Les ***variadic template*** sont le pendant, pour les templates, des fonctions ***variadiques*** (ellipse exprimée par un point de suspension), dont la syntaxe la plus connue est ***printf()***.

```cpp
template <class ... Types> ... ; // ... ellipse qui détermine un nombre de paramètres variables
```

Cette nouvelle fonctionnalité est toutefois très différente de la notion de fonction à nombre d'arguments variables, ne serait-ce que parce que cette dernière s'exprime au moment de l'exécution, tandis que la variabilité des patrons s'exprime directement au niveau de la compilation elle-même.

Pour les implémenter correctement, deux approches sont à privilégier, soit à partir d'une écriture récursive, soit par la création d'un tableau intermédiaire. Dans les deux cas, si vous souhaitez que les types de la collection soit libre, vous devez rendre variable le premier type de la première valeur fournie. Si vous fixez le type de la collection, la syntaxe avec une simple ellipse suffit.

Je rappelle que depuis le C++11, le langage a introduit la déduction de types (ce que nous appelons l'inférence de types) sous deux formes : **auto** et ***decltype()***. Cette fonctionnalité permet ainsi de laisser le compilateur choisir le type d'un élément en fonction de ce qui lui est assigné. Nous connaissons déjà le mot clé **auto** qui nous permet d'évitez d'utiliser la syntaxe des templates afin d'avoir une écriture plus concise, comme par exemple :

retour fonction(auto ... paramètres) { ... }

Le mot-clé ***decltype()*** permet de déduire le type d'une autre expression. L'intérêt de ***decltype()*** ci-dessous est très limité, mais devient vite indispensable lorsque certaines variables utilisent **auto** (car seul le compilateur possède le type explicite) ou lors de l'utilisation de templates (puisqu'on ne connaît pas le type à l'avance).

```
int i = 0;
double k = 2.0;
decltype(k) j = i + 2; // j sera du même type que k, donc double.
```

À l'issu de la compilations decltype(k) nous indique que le type de la variable « j » est un double qui est déduit de la variable « k ». C'est comme si nous avions directement déclaré : double j = i+2 ; Grâce à cette directive, nous pouvons choisir le type d'une variable par rapport à une autre, ou suivant le calcul d'une expression plus complexe.

Suite de fonctions avec un nombre d'arguments variable

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

// Écritures récursives
auto maximum(const auto& x, const auto& y) { return x>y ? x : y; }
auto maximum(const auto& premier, const auto& ... arguments) { return maximum(premier, maximum(arguments...)); }

// Afficher des valeurs de types variables
void affiche(const auto& val) { cout << val << endl; }
void affiche(const auto& premier, const auto& ... arguments)
{
  cout << premier << ", "; affiche(arguments...);
}

void afficher(char *prefixe, auto val, auto ... args)
{
  decltype(val) tableau[] = {args ...};
  cout << '[' << prefixe << ": " << val << ' ';
  for (auto& valeur : tableau) cout << valeur << ' ';
  cout << "\b]\n";
}
// Liste d'initialisation de tableau
int somme(const auto& ... n)
{
  int resultat = 0;
  cout << "[Somme de : ";
  for (int valeur : {n ... }) { cout << valeur << ' '; resultat+=valeur; }
  return resultat;
}

// Connaître le nombre d'arguments
double moyenne(const auto& ... notes)
{
  int nombre = sizeof...(notes);
  double somme = 0.0;
  cout << "[Moyenne de : ";
  for (double note : {notes ...}) { cout << note << ' '; somme += note; }
  return somme / nombre;
}

// Utilisation de foncteurs
void traitement(auto foncteur, auto premier, auto ... suivants)
{
  decltype(premier) tableau[] = {premier, suivants ...};
  cout << '[';
  for (auto& x : tableau) { cout << x << ' '; foncteur(x); }
  cout << "\b] ";
}

int main()
{
  cout << "Maximum = " << maximum(3, -2, 5, 18, -8, 4, 7) << endl;
  affiche(3, 'a', 5.8, "un texte");
  afficher("Entiers", 3, -2, 5, 18, -8, 4, 7);
  afficher("Caractères", 'a', 'l', 'h', 'q');
  afficher("Réels", 5.8, 6.25, -8, -4.5);

  cout << "=> " << somme(5, 6, 12) << ']' << endl;
  cout << "=> " << moyenne(11.5, 15.0, 8.5, 17.5, 7.5) << ']' << endl;

  int maxi = 0;
  traitement([&maxi](int x) {if (x>maxi) maxi = x; }, 3, -2, 5, 18, -8, 4, 7);
```

```
Maximum = 18
3, a, 5.8, un texte
[Entiers: 3 -2 5 18 -8 4 7]
[Caractères: a l h q]
[Réels: 5.8 6.25 -8 -4.5]
[Somme de : 5 6 12 => 23]
[Moyenne de : 11.5 15 8.5 17.5 7.5 => 12]
[3 -2 5 18 -8 4 7] maxi(18)
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
  cout << "maxi(" << maxi << ')' << endl;

  return 0;
}
```

Foncteur d'affichage et suite de fonctions avec un nombre d'arguments variable

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

// Foncteur d'affichage
struct Afficher
{
  Afficher(char* prefixe) { cout << prefixe << " de ["; }
  void operator() (const auto& element) { cout << element << ' '; }
  ~Afficher() { cout << "\b] = "; }
};

// Liste d'initialisation de tableau
auto somme(auto premier, auto ... n)
{
  decltype(premier) resultat = premier;
  Afficher affiche("Somme");  affiche(premier);
  for (auto& valeur : {premier, n ... }) { affiche(valeur); resultat+=valeur; }
  return resultat;
}

// Connaître le nombre d'arguments
auto moyenne(auto premier, auto ... notes)
{
  int nombre = sizeof...(notes)+1;
  decltype(premier) somme = 0.0;
  Afficher affiche("Moyenne");
  for (auto& note : {premier, notes ...}) { affiche(note); somme += note; }
  return somme / nombre;
}

// Utilisation de lambdas
void traitement(char* prefixe, auto foncteur, auto premier, auto ... suivants)
{
  decltype(premier) tableau[] = {premier, suivants ...};
  Afficher affiche(prefixe);
  for (auto& x : tableau) { affiche(x); foncteur(x); }
}
```

```
Somme de [5 6 12] = 23
Moyenne de [11.5 15 8.5 17.5 7.5] = 12
Maximum de [3 -2 5 18 4 -8 7] = 18
Minimum de [3 -2 5 18 4 -8 7] = -8
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

```
int main()
{
  cout << somme(5, 6, 12) << endl;
  cout << moyenne(11.5, 15.0, 8.5, 17.5, 7.5) << endl;

  int maxi = 0;
  traitement("Maximum", [&maxi](int x) {if (x>maxi) maxi = x; }, 3, -2, 5, 18, 4, -8, 7);
  cout << maxi << endl;

  int mini = 100;
  traitement("Minimum", [&mini](int x) {if (x<mini) mini = x; }, 3, -2, 5, 18, 4, -8, 7);
  cout << mini << endl;

  return 0;
}
```

Sémantique du mouvement – les « rvalue » référence

La sémantique de mouvement est un nouveau concept apporté par le standard C++11, qui permet dans certaines situations de remplacer une copie par une « vampirisation » de l'objet source, généralement lorsque celui-ci est amené à disparaître très prochainement.

lvalue *: Acronyme de « left hand side value », c'est-à dire une expression qui peut se trouver à gauche d'un opérateur d'affectation. C'est maintenant généralisé à tout ce qui se réfère à un espace en mémoire, donc tout ce dont nous pouvons obtenir l'adresse avec l'opérateur* ***'&'*** *(adresse de la variable).*

rvalue *: Acronyme de « right hand side value », c'est-à-dire une expression qui peut être à droite d'un opérateur d'affectation. C'est maintenant généralisé en tout ce qui n'est pas une* ***lvalue****. C'est souvent une variable temporaire (retour de fonction qui renvoie une valeur…).*

rvalue reference *: (nouvelle fonctionnalité !) Comme son nom l'indique, est une référence sur une* ***rvalue****. Une référence classique est symbolisée dans le code par* ***'&'****, et une* ***rvalue reference*** *par* ***'&&'****.*

Avant de se pencher sur cette nouvelle notion, revenons sur le concept classique de référence. Elle est souvent très appréciée lorsque nous devons passer des structures de données à des paramètres de fonctions ou de méthodes.

Pour récupérer la valeur Soit nous proposons un paramètre classique, auquel cas une copie est réalisée entre l'argument et le paramètre. Soit nous utilisons une référence. Dans ce cas là, le paramètre représente l'argument, c'est un deuxième nom pour une même zone mémoire. Cela évite d'effectuer une copie systématique entre l'argument et le paramètre, d'où un gain en espace mémoire et surtout un gain en rapidité de traitement. La signature classique est la suivante :

void uneFonction(const Type& paramètre) ;

Penchons nous maintenant sur l'intérêt d'utiliser des « **rvalue reference** ». Pour cela, analysons le code suivant :

Fonction qui génére des vecteurs d'entiers

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

vector<int> creation(unsigned nombre, auto lambda)
{
  vector<int> tableau(nombre);
  for (int &element : tableau) lambda(element);
  return tableau;
}

int main()
{
  int depart=0;
  vector<int> entiers = creation(15, [&depart](int& element){ element = ++depart; });
  for (int element : entiers) cout << element << ' ';
  cout << endl;
  return 0;
}
```

Ce code peut sembler inoffensif mais il est, en réalité, très inefficace. Dans la fonction, nous créons un vecteur qui peut contenir plein de données potentiellement très volumineuses. L'enchaînement des opérations est la suivante :

*- Lors du **return**, ce vecteur sera copié dans un objet temporaire.*
*- Le vecteur temporaire sera ensuite copié dans le vecteur **entiers**.*
- L'objet temporaire est ensuite détruit.

Au final, nous avons 2 copies et 1 destruction inutiles et potentiellement très coûteuses. Tout ce que nous voulions faire, était de déplacer le contenu du vecteur tableau créé dans creation() dans notre vecteur entiers.

Évidemment, il est possible d'éviter cela par plusieurs méthodes, comme créer le vecteur à l'avance et retourner une référence sur celui-ci. Mais cela complique un petit peu la fonction puisque vous devez rajouter un paramètre supplémentaire, et surtout elle est moins utile puisque vous devez initialiser le nombre d'élément lorsque vous créez le vecteur à l'extérieur. La fonction telle qu'elle est écrite me paraît plus générique. Voici une deuxième alternative qui utilise les « **rvalue reference** » :

Fonction qui génére des vecteurs d'entiers

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

vector<int> creation(unsigned nombre, auto lambda)
{
  vector<int> tableau(nombre);
  for (int &element : tableau) lambda(element);
  return tableau;
}

int main()
{
  int depart=0;
  vector<int>&& entiers = creation(15, [&depart](int& element){ element = ++depart; });
  for (int element : entiers) cout << element << ' ';
  cout << endl;
  return 0;
}
```

De cette manière, nous récupérons le contenu du vecteur tableau créé dans la fonction via l'objet temporaire. Par conséquent, plus de copies inutiles et l'objet temporaire ne sera pas détruit immédiatement. Nous faisons référence en quelque sorte à l'objet temporaire. Ainsi le code reste parfaitement lisible et simple.

Ainsi globalement, le contenu du vecteur ne sera plus recopié, mais transféré (vampirisé) depuis le temporaire vers sa nouvelle destination, et c'est précisément ce à quoi servent les « **rvalue reference** » : rendre transparent pour l'utilisateur l'élimination des temporaires.