

Conteneurs dynamiques

Une fonction essentielle de la bibliothèque standard est de fournir des mécanismes pour supporter les algorithmes avec la meilleure efficacité possible. Cet énoncé comporte plusieurs objectifs contradictoires. Les algorithmes réclament de la généricité, autrement dit des méthodes de travail indépendantes du type de données à manipuler. Le langage C utilisait volontiers les pointeurs `void*` pour garantir la généricité mais cette approche entraîne une perte d'efficacité importante dans le contrôle des types, une complication du code et finalement de piètres performances. L'efficacité réclamée pour STL s'obtient au prix d'une conception rigoureuse et de contrôles subtils des types. Certes, le résultat est un compromis entre des attentes parfois opposées mais il est assez probant pour être utilisé à l'élaboration d'applications dont la sûreté de fonctionnement est impérative.

Les concepteurs de la STL ont utilisé les modèles de classes pour développer la généricité. Les modèles de classes et de fonctions sont étudiés en détail au chapitre Programmation orientée objet, et leur emploi est assez simple. Une classe est instanciée à partir de son modèle en fournissant les paramètres attendus, généralement le type de données effectivement pris en compte pour l'implémentation de la classe. Il faut absolument différencier cette approche de l'utilisation de macros (`#define`) qui provoque des résultats inattendus. Ces macros se révèlent très peu sûres d'emploi.

Une idée originale dans la construction de la bibliothèque standard est la corrélation entre les conteneurs de données et les algorithmes s'appliquant à ces conteneurs. Une lecture comparée de différents manuels d'algorithmie débouche sur la conclusion que les structures de données sont toujours un peu les mêmes, ainsi que les algorithmes s'appliquant à ces structures. Il n'était donc pas opportun de concevoir des structures isolées, comme une pile ou une liste sans penser à l'après, à l'application d'algorithmes moins spécifiques. Les concepteurs de la STL ont su éviter cet écueil.

1. Conteneurs

Les conteneurs sont des structures de données pour ranger des objets de type varié. Les conteneurs de la STL respectent les principales constructions élaborées en algorithmie.

Classe	Description	En-tête
<code>vector</code>	Tableau de <code>T</code> à une dimension. Structure de référence.	<code><vector></code>
<code>list</code>	Liste doublement chaînée de <code>T</code> .	<code><list></code>
<code>deque</code>	File d'attente (<i>queue</i> en anglais) de <code>T</code> à double accès.	<code><deque></code>
<code>priority_queue</code>	File d'attente à priorité.	<code><deque></code>
<code>stack</code>	Pile de <code>T</code> .	<code><stack></code>
<code>map</code>	Tableau associatif de <code>T</code> .	<code><map></code>
<code>multimap</code>	Tableau associatif de <code>T</code> .	<code><map></code>
<code>set</code>	Ensemble de <code>T</code> .	<code><set></code>
<code>multiset</code>	Ensemble de <code>T</code> .	<code><set></code>
<code>bitset</code>	Tableau de booléens, ensemble de bits.	<code><bitset></code>

La classe `vector` est un peu une structure de référence. Bien que les autres classes ne soient pas basées sur celle-ci pour des raisons de performances, `vector` ouvre la voie aux autres conteneurs. Cette première partie explicite son fonctionnement, tandis que la partie suivante est consacrée aux séquences, c'est-à-dire aux conteneurs de plus haut niveau, mais reprenant la logique du vecteur.

a. Insertion d'éléments et parcours

Les deux principales fonctions attendues des conteneurs sont de pouvoir agréger de nouveaux éléments et de parcourir les éléments référencés par le conteneur.

L'insertion est une opération dont le déroulement est généralement propre au type de conteneur considéré. Par exemple, l'insertion dans une file d'attente à priorité est sensiblement différente de l'insertion dans un tableau associatif.

Le parcours, lorsqu'il est complet, est fréquemment impliqué dans l'inspection de la collection d'objets que représente le conteneur. Ce parcours est également envisageable pour explorer les résultats fournis par un algorithme de sélection ou de recherche.

b. Itérateurs

Lorsque le programmeur effectue une recherche, un tri, une sélection ou d'autres opérations de ce type sur un conteneur, il n'a pas besoin de rendre son programme dépendant du conteneur utilisé. En effet, le parcours d'une sélection sur un tableau associatif ou sur un vecteur est identique d'un point de vue algorithmique. Les itérateurs ont justement pour mission de fournir un moyen général pour parcourir les données.

Voici maintenant un exemple de construction d'un vecteur de chaînes et de parcours à l'aide d'un itérateur :

```
#include <iostream>
#include <string>
#include <vector>
#include <iterator>

using namespace std;

int main(int argc, char* argv[])
{
    vector<string> tab;
    vector<string>::iterator iter;

    tab.insert(tab.begin(), "Mozart");
    tab.insert(tab.begin()+1, "Beethoven");
    tab.insert(tab.begin()+2, "Schubert");
```

```

for(iter=tab.begin(); iter!=tab.end(); iter++)
    cout << (*iter) << " ";

cout << endl;
return 0;
}

```

Un itérateur s'utilise donc à la manière d'un pointeur dont l'arithmétique est propre au conteneur considéré.

Les itérateurs servent au parcours des éléments d'un conteneur indépendamment du type réel de conteneur. Chaque conteneur possède des fonctions membres pour déterminer les bornes (début et fin) de la séquence d'éléments.

c. Opérations applicables à un vecteur

Le vecteur est un tableau dynamique, une collection d'éléments du même type. Le tableau peut également être considéré comme une pile, structure très importante pour l'écriture de programmes. Les méthodes `push_back()`, `pop_back()` et `back()` sont justement destinées à accomplir ce rôle.

```

vector<int> v;
v.push_back(1756); // empile 1756
v.push_back(1770); // empile 1770
v.push_back(1797); // empile 1797

int last=v.back(); // interroge le dernier élément, soit 1797
v.pop_back(); // dépile 1797

vector<int>::iterator i;
for(i=v.begin(); i!=v.end(); i++)
    cout << *i << " ";

```

Le vecteur possède également un certain nombre de méthodes pour l'insertion et l'effacement de données à des emplacements précis alors que les fonctions de piles agissent à l'extrémité de la séquence.

```
vector<string> opus;
opus.insert(opus.begin(), "K545");
opus.insert(opus.begin()+1, "K331");
opus.erase(opus.begin());
vector<string>::iterator o;

for(o = opus.begin(); o != opus.end(); o++)
    cout << *o << " "; //affiche seulement K331
```

Enfin le vecteur possède une fonction `swap()` qui échange les valeurs entre deux vecteurs, opération très utile pour le tri des éléments.

2. Séquences

a. Conteneurs standards

La plupart des conteneurs suivent le modèle de `vector`, sauf lorsque cet "héritage" dénature le rôle accompli par le conteneur. Nous pouvons d'ores et déjà dégager un certain nombre de caractéristiques et d'opérations communes.

Types de membres

<code>value_type</code>	Type d'élément.
<code>allocator_type</code>	Type du gestionnaire d'élément.
<code>size_type</code>	Type des indices, des comptes d'éléments.
<code>difference_type</code>	Type de différence entre deux itérateurs.
<code>iterator</code>	Espèce de <code>value_type*</code>
<code>const_iterator</code>	Type de const <code>value_type*</code>
<code>reverse_iterator</code>	Itérateur en ordre inverse, <code>value_type*</code>
<code>const_reverse_iterator</code>	<code>const value_type*</code>
<code>reference</code>	<code>value_type&</code>
<code>const_reference</code>	<code>const value_type&</code>

Itérateurs

<code>begin()</code>	Pointe sur le premier élément.
<code>end()</code>	Pointe sur l'élément suivant le dernier élément.
<code>rbegin()</code>	Pointe sur le dernier élément (premier en ordre inverse).
<code>rend()</code>	Pointe sur l'élément suivant le dernier en ordre inverse (avant le premier).

Accès aux éléments

<code>front()</code>	Premier élément.
<code>back()</code>	Dernier élément.
<code>[]</code>	Index en accès non contrôlé, non applicable aux listes.
<code>at()</code>	Index en accès contrôlé, non applicable aux listes.

Opérations sur pile et file

<code>push_back()</code>	Ajoute à la fin de la séquence.
<code>pop_back()</code>	Supprime le dernier élément.
<code>push_front()</code>	Insère un nouveau premier élément (applicable à <code>list</code> et <code>deque</code>).
<code>pop_front()</code>	Supprime le premier (uniquement <code>list</code> et <code>deque</code>).

Opération sur listes

<code>insert(p)</code>	Insère avant <code>p</code> .
<code>erase(p)</code>	Supprime <code>p</code> .
<code>clear()</code>	Efface tous les éléments.

Opérations associatives

<code>operator[] (k)</code>	Accède à l'élément de clé <code>k</code> .
<code>find(k)</code>	Cherche l'élément de clé <code>k</code> .
<code>lower_bound(k)</code>	Cherche le premier élément de clé <code>k</code> .
<code>upper_bound(k)</code>	Cherche le premier élément de clé supérieure à <code>k</code> .
<code>equal_range(k)</code>	Cherche tous les éléments <code>lower_bound</code> et <code>upper_bound</code> de clé <code>k</code> .

b. Séquences

Les séquences désignent les conteneurs qui suivent le modèle de `vector`. On trouve comme séquences les classes `vector`, `list` et `deque`.

`list`

Une liste est une séquence optimisée pour l'insertion et la suppression d'éléments. Elle dispose d'itérateurs bidirectionnels mais pas d'un accès par index, à la fois pour respecter la forme des algorithmes spécifiques aux listes et pour l'optimisation des performances.

La classe `list` est probablement implémentée à l'aide de listes doublement chaînées, mais le programmeur n'a pas besoin de connaître ce détail.

En supplément des opérations applicables aux séquences générales, la classe `list` propose les méthodes `splice()`, `merge()` et `sort()`.

<code>splice()</code>	Déplace les éléments d'une liste vers une autre, sans les copier.
<code>merge()</code>	Fusionne deux listes.
<code>sort()</code>	Trie une liste.

D'autres méthodes sont particulièrement utiles telles `reverse()`, `remove()`, `remove_if()` et `unique()`.

Nous donnons un exemple de mise en œuvre pour certaines de ces méthodes :

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

bool initiale_b(string s)
{
    return s[0]=='B';
}

int main(int argc, char* argv[])
{
    // une liste de chaînes
    list<string> liste;

    // quelques entrées
    liste.push_back("Beethoven");
    liste.push_back("Bach");
    liste.push_back("Brahms");
    liste.push_back("Haydn");
    liste.push_back("Rachmaninov");

    // retourner la liste
    liste.reverse();
}
```

```

// supprimer tous les compositeurs commençant par B
liste.remove_if(initial_b);

// afficher la liste
list<string>::iterator i;
for(i=liste.begin(); i!=liste.end() ; i++)
{
    string nom=*i;
    cout << " " << nom.c_str();
}
return 0;
}

```

deque

La classe `deque` ressemble beaucoup à la classe `list`, sauf qu'elle est optimisée pour l'insertion et l'extraction aux extrémités de la séquence.

Pour les opérations d'insertion portant sur le milieu de la séquence, l'efficacité est en deçà de ce qu'offre la liste.

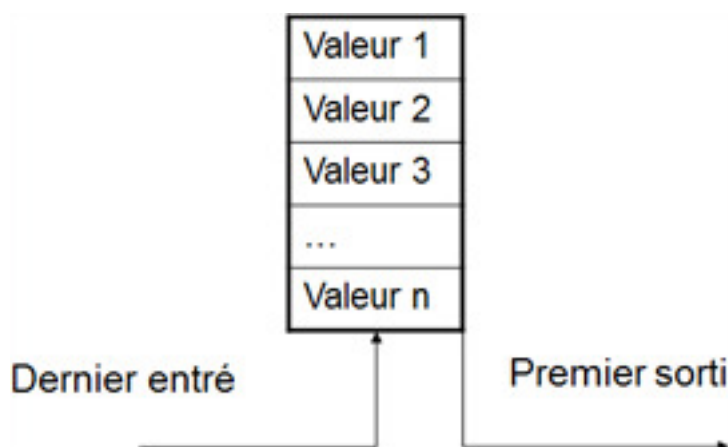
c. Adaptateurs de séquences

À partir des séquences `vector`, `list` et `deque`, on conçoit de nouvelles séquences `stack`, `queue` et `priority_queue`. Toutefois, pour des raisons de performances, les classes correspondantes refondent l'implémentation. Il ne s'agit donc pas de sous-classement mais plutôt d'une adaptation. Ceci explique leur nom d'adaptateurs de séquences.

Les adaptateurs de séquences ne fournissent pas d'itérateurs, car les algorithmes qui leur sont applicables n'en ont pas besoin.

stack

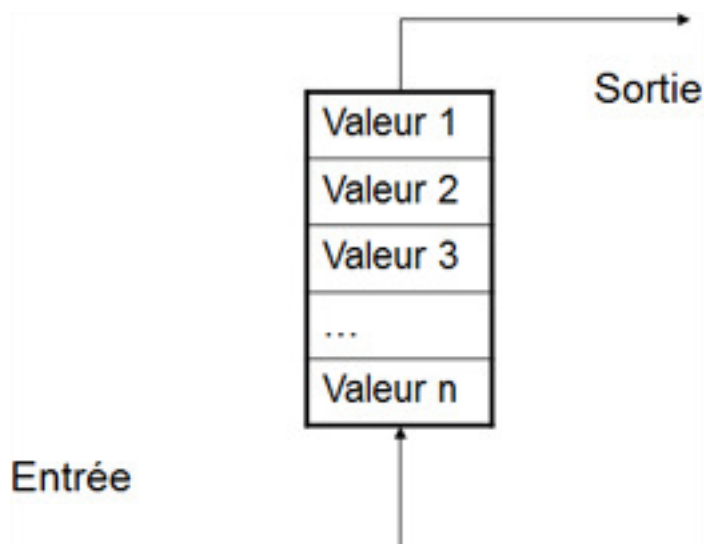
Une pile est une structure de données dynamique où il n'est pas possible d'accéder à un élément sans retirer ses successeurs. On appelle cette structure LIFO (*Last In First Out*).



L'interface `stack` reprend les méthodes qui ont toujours du sens pour une pile, à savoir `back()`, `push_back()` et `pop_back()`. Ces fonctions existent aussi sous la forme de noms plus conventionnellement utilisés en écriture d'algorithme : `top()`, `push()`, `pop()`.

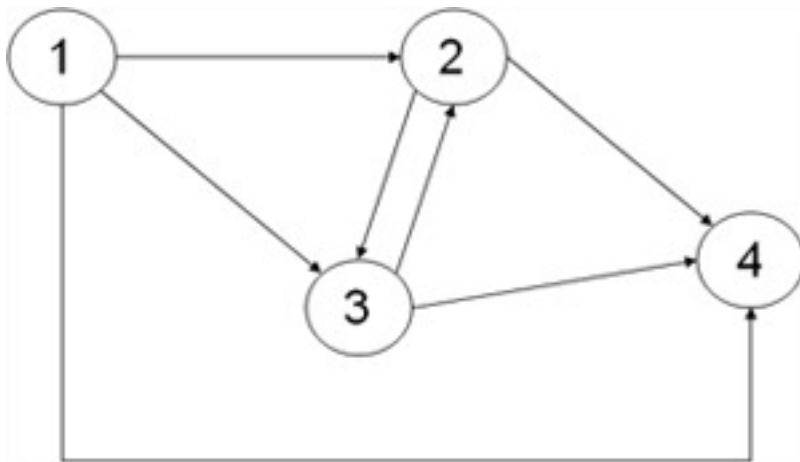
queue

Les files d'attente sont également appelées piles FIFO (*First In First Out*). Leur fonctionnement est un peu différent des piles classiques puisqu'on insère des valeurs à la fin (empilement) et que l'on retire les valeurs au début.



La classe `queue` propose les méthodes d'interface `front()`, `back()`, `push()` et `pop()`. Nous proposons un petit programme de parcours de graphe écrit à l'aide d'une classe `queue`.

Le graphe servant d'exemple est un graphe orienté composé de quatre sommets :



Nous avons choisi d'implémenter ce graphe à l'aide d'une matrice d'adjacence, structure qui permet de conserver un algorithme très lisible :

```

#define N 4

bool**init_graphe()
{
    bool**graphe;
    graphe=new bool*[N];

    graphe[0]=new bool[N];

    graphe[0][0]=false; // true lorsqu'il y a une adjacence
    graphe[0][1]=true;
    graphe[0][2]=true;
    graphe[0][3]=true;

    graphe[1]=new bool[N];
    graphe[1][0]=false;
    graphe[1][1]=false;
    graphe[1][2]=true;
    graphe[1][3]=true;
    graphe[2]=new bool[N];
    graphe[2][0]=false;
    graphe[2][1]=true;
    graphe[2][2]=false;
  
```

```

graphe[2][2]=true;

graphe[3]=new bool[N];
graphe[3][0]=false;
graphe[3][1]=false;
graphe[3][2]=false;
graphe[3][3]=false;

return graphe;
}

```

Voici maintenant la structure de données qui supporte le parcours dit en largeur d'abord : une file d'attente. Comme il s'agit de parcourir un graphe pour lequel plusieurs nœuds peuvent être adjacents à d'autres nœuds, un tableau de booléens conserve l'état de visite de chaque sommet.

```

queue<int> file;
bool*sommets_visites;

```

Nous poursuivons par le code des fonctions destinées au parcours du graphe :

```

void visite(bool**graphe,int n,int sommet)
{
    if(sommets_visites[sommet])
        return;

    sommets_visites[sommet]=true;
    cout << "Visite du sommet" << (sommet+1) << endl;

    // cherche toutes les adjacences non visitées
    for(int i=0; i<n; i++)
        if(graphe[sommet][i] && ! sommets_visites[i])
            file.push(i);
}

void parcours_largeur(bool**graphe,int n)
{

```

```

sommets_visites=new bool[n];
for(int i=0; i<n; i++)
    sommets_visites[i]=false; // sommet pas visité

file.push(0); // part du sommet 1

while(! file.empty())
{
    int s=file.front();
    file.pop();
    visite(graphe,n,s);
}
}

```

Notre exemple se termine par la fonction `main()` et par son exécution.

```

int main(int argc, char* argv[])
{
    bool**graphe;
    graphe=init_graphe();

    parcours_largeur(graphe,N);
    return 0;
}

```

L'exécution assure bien que chaque sommet accessible depuis le sommet 1 est visité une et une seule fois :

```

C:\WINDOWS\system32\cmd.exe
Visite du sommet 1
Visite du sommet 2
Visite du sommet 3
Visite du sommet 4
Appuyez sur une touche pour continuer...

```

File d'attente à priorité

Il s'agit d'une structure de données importante pour la détermination du meilleur chemin

dans un graphe. L'algorithme A* (A Star) utilise justement des files d'attente à priorité. Par rapport à une file d'attente ordinaire, une pondération accélère le dépilement d'éléments au détriment d'autres.

```
priority_queue<int> pq;
pq.push(30);
pq.push(10);
pq.push(20);

cout << pq.top() << endl; // affiche 30
```

d. Conteneurs associatifs

Les conteneurs associatifs sont très utiles à l'implémentation d'algorithmes de toutes sortes. Ces structures de données sont si pratiques que certains langages/environnements les proposent en standard. La classe principale s'appelle `map`, on la trouve parfois sous le nom de tableau associatif, ou de table de hachage.

map

La classe `map` est une séquence de clés et de valeurs. Les clés doivent être comparables pour garantir de bonnes performances. Lorsque l'ordre des éléments est difficile à déterminer, la classe `hash_map` fournira de meilleures performances.

Les itérateurs de la classe `map` fournissent des instances de la classe `pair` regroupant une clé et une valeur.

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

int main(int argc, char* argv[])
{
    map<int,string> tab;
    tab[1]="Paris";
    tab[5]="Londres";
    tab[10]="Berlin";
```



```

map<int,string>::iterator paires;
for(paires=tab.begin(); paires!=tab.end(); paires++)
{
    int cle=paires->first;
    string valeur=paires->second;
    cout << cle << " " << valeur.c_str() << endl;
}
return 0;
}

```

multimap

Le `multimap` est une table `map` dans laquelle la duplication de clé est autorisée.

set

Un jeu (set) est une table `map` ne gérant que les clés. Les valeurs n'ont aucune relation entre elles.

multiset

Il s'agit d'un set pour lequel la duplication de clé est autorisée.

3. Algorithmes

Un conteneur offre déjà un résultat intéressant mais la bibliothèque standard tire sa force d'un autre aspect : les conteneurs sont associés à des fonctions générales, des algorithmes. Ces fonctions utilisent presque toutes des itérateurs pour harmoniser l'accès aux données d'un type de conteneur à l'autre.

a. Opérations de séquence sans modification

Il s'agit principalement d'algorithmes de recherche et de parcours.

<code>for_each()</code>	Exécute l'action pour chaque élément d'une séquence.
<code>find()</code>	Recherche la première occurrence d'une valeur.
<code>find_if()</code>	Recherche la première correspondance d'un prédicat.
<code>find_first_of()</code>	Recherche dans une séquence une valeur provenant d'une autre.
<code>adjacent_find()</code>	Recherche une paire adjacente de valeurs.
<code>count()</code>	Compte les occurrences d'une valeur.
<code>count_if()</code>	Compte les correspondances d'un prédicat.
<code>mismatch()</code>	Recherche les premiers éléments pour lesquels deux séquences diffèrent.
<code>equal()</code>	Vaut <code>true</code> si les éléments de deux séquences sont égaux au niveau des paires.
<code>search()</code>	Recherche la première occurrence d'une séquence en tant que sous-séquence.
<code>find_end()</code>	Recherche la dernière occurrence d'une séquence en tant que sous-séquence.
<code>search_n()</code>	Recherche la nième occurrence d'une valeur.

Nous proposons un exemple de recherche de valeur dans un vecteur de chaînes `char*` :

```

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

int main(int argc, char* argv[])
{
    vector<char*> tab;
    tab.push_back("Sonate");
    tab.push_back("Partita");
    tab.push_back("Cantate");
    tab.push_back("Concerto");
    tab.push_back("Symphonie");
    tab.push_back("Aria");
    tab.push_back("Prélude");

    vector<char*>::iterator pos;
    char*mot="Concerto";
    pos=find(tab.begin(),tab.end(),mot);
    cout << *pos << endl; // affiche Concerto
    return 0;
}

```

b. Opérations de séquence avec modification

<code>transform()</code>	Application d'une opération sur tous les éléments d'une séquence.
<code>copy()</code>	Copie une séquence.
<code>copy_backward()</code>	Copie une séquence en ordre inverse, à partir de son dernier élément.
<code>swap()</code>	Échange deux éléments.

<code>iter_swap()</code>	Idem sur la base d'itérateurs.
<code>replace()</code>	Remplace les éléments par une valeur donnée.
<code>replace_if()</code>	Remplace les éléments par une valeur donnée lorsqu'un prédicat est vérifié.
<code>replace_copy()</code>	Copie une séquence en remplaçant les éléments par une valeur donnée.
<code>replace_copy_if()</code>	Idem sur la base d'un prédicat.
<code>fill()</code>	Remplace chaque élément par une valeur donnée.
<code>fill_n()</code>	Limite l'opération <code>fill</code> aux n premiers éléments.
<code>generate()</code>	Remplace tous les éléments par le résultat d'une opération.
<code>remove()</code>	Supprime les éléments ayant une valeur donnée.
<code>remove_if()</code>	Idem sur la base d'un prédicat.
<code>remove_copy()</code>	Copie une séquence en supprimant les éléments d'une valeur donnée.
<code>remove_copy_if()</code>	Idem sur la base d'un prédicat.
<code>unique()</code>	Supprime les éléments adjacents et égaux.

<code>unique_copy()</code>	Copie une séquence en supprimant les éléments adjacents et égaux.
<code>reverse()</code>	Inverse l'ordre des éléments d'une séquence.
<code>reverse_copy()</code>	Copie une séquence en ordre inverse.
<code>rotate()</code>	Opère un décalage circulaire des éléments.
<code>rotate_copy()</code>	Idem sur la base d'une copie.
<code>random_shuffle()</code>	Déplace les éléments de manière aléatoire.

c. Séquences triées

<code>sort()</code>	Trie les éléments.
<code>stable_sort()</code>	Trie en conservant l'ordre des éléments égaux.
<code>partial_sort()</code>	Trie la première partie d'une séquence.
<code>partial_sort_copy()</code>	Idem sur la base d'une copie.
<code>nth_element()</code>	Place le nième élément à l'emplacement approprié.
<code>lower_bound()</code>	Recherche la première occurrence d'une valeur.
<code>upper_bound()</code>	Recherche la dernière occurrence d'une valeur.
<code>equal_range()</code>	Recherche une sous-séquence d'une valeur donnée.
<code>binary_search()</code>	Recherche une valeur dans une séquence triée.
<code>merge()</code>	Fusionne deux séquences triées.
<code>inplace_merge()</code>	Fusionne deux sous-séquences consécutives triées.
<code>partition()</code>	Déplace les éléments autour d'une valeur pivot.
<code>stable_partition()</code>	Identique à l'algorithme précédent, mais conserve l'ordre des éléments égaux.

d. Algorithmes de définition

<code>includest()</code>	Vérifie si une séquence est une sous-séquence d'une autre.
<code>set_union()</code>	Réalise une union triée de plusieurs séquences.
<code>set_intersection()</code>	Intersection triée.
<code>set_difference()</code>	Construit une séquence d'éléments triés dans la première, mais pas dans la deuxième.

e. Minimum et maximum

<code>min_element()</code>	La plus petite valeur dans une séquence.
<code>min()</code>	La plus petite des deux valeurs.
<code>max_element()</code>	La plus grande valeur dans une séquence.
<code>max()</code>	La plus grande des deux valeurs.

4. Calcul numérique

Le langage C++ est particulièrement attrayant pour la clarté de son exécution, dont la durée peut être prévue. Les types habituels - du `char` au `double` - sont également connus d'autres langages, comme le C ou le Pascal, et ils assurent une grande partie des calculs nécessaires à l'exécution d'un programme.

Il existe des situations pour lesquelles ces types ne sont plus adaptés, soit la précision n'est pas suffisante - même avec un `double` - soit la rapidité des calculs est mise à mal par une volumétrie trop conséquente. Les logiciels nécessitant d'importants calculs en haute précision ont du mal à satisfaire une optimisation avancée des performances avec une représentation satisfaisante des valeurs numériques.

Dressons un rapide état des lieux des possibilités de calcul numérique en C++ avant d'introduire la partie correspondante dans la bibliothèque standard.

a. Limites des formats ordinaires

L'en-tête `<limits>` offre un certain nombre de classes paramétrées pour déterminer les valeurs caractéristiques d'un format tel `short` ou `double`. Quelle est la plus petite valeur, quel est le plus petit incrément (epsilon)...

Le programme suivant nous donne des informations sur le type `double` :

```
#include <iostream>
#include <limits>
using namespace std;
int main(int argc, char* argv[])
{
    cout << "Plus petite valeur (double)=" <<
    numeric_limits<double>::min() << endl;

    cout << "Plus grande valeur (double)=" <<
    numeric_limits<double>::max() << endl;
    cout << "Plus petite valeur absolue significative (double)=" <<
    numeric_limits<double>::epsilon() << endl;
    cout << "Représentation d'une quantité qui n'est pas un nombre
(double)=" << numeric_limits<double>::signaling_NaN() << endl;
    return 0;
}
```

b. Fonctions de la bibliothèque

Les en-têtes de la bibliothèque du C, `<cmath>` et `<math.h>`, fournissent le support pour

les fonctions mathématiques usuelles, applicables pour la plupart au type double. Ce type a été choisi d'une part parce qu'il correspond à un format standardisé (IEEE 754), et d'autre part car les microprocesseurs savent travailler directement avec ce format, sans prendre plus de temps que le `float` qui offre, lui, une moins bonne précision. Attention toutefois à certaines implémentations logicielles qui peuvent faire varier légèrement ces formats.

Il peut être intéressant de consulter l'excellent ouvrage de Laurent Padjasek, "Calcul numérique en assembleur" (Sybex), pour découvrir toutes les subtilités du codage IEEE 754.

Nous proposons maintenant une liste des principales fonctions de la bibliothèque mathématique du C.

<code>abs()</code> , <code>fabs()</code>	Valeur absolue.
<code>ceil()</code>	Plus petit entier non inférieur à l'argument.
<code>floor()</code>	Plus grand entier non supérieur à l'argument.
<code>sqrt()</code>	Racine carrée.
<code>pow()</code>	Puissance.
<code>cos()</code> , <code>sin()</code> , <code>tan()</code>	Fonctions trigonométriques.
<code>acos()</code> , <code>asin()</code> , <code>atan()</code>	Arc cosinus, arc sinus, arc tangente, fonctions trigonométriques inverses.
<code>atan2(x,y)</code>	<code>atan(x/y)</code>
<code>sinh()</code> , <code>cosh()</code> , <code>tanh()</code>	Fonctions trigonométriques hyperboliques.
<code>exp()</code> , <code>log()</code> , <code>log10()</code>	Exponentielle et logarithmes.
<code>mod()</code>	Partie fractionnaire.

c. Fonctions de la bibliothèque standard et classe `valarray`

La bibliothèque standard s'est surtout attachée à proposer un mécanisme qui assure la jointure entre l'optimisation des calculs et le programme proprement dit. Les microprocesseurs adoptent le travail à la chaîne et en parallèle pour optimiser les calculs, concept que l'on désigne sous le terme de calcul vectoriel.

Nous trouvons dans la STL la structure `valarray` dont le rôle est justement de préparer l'optimisation des calculs tout en conservant une assez bonne lisibilité du programme.

Cette classe possède parmi ses constructeurs une signature qui accepte un tableau ordinaire, initialisation qui assure une bonne intégration avec l'existant.

Le programme suivant crée un `valarray` à partir d'une série de valeurs double fournies dans un tableau initialisé en extension. Une seule opération `v+=10` correspond à une addition effectuée sur chaque valeur du vecteur. Suivant les implémentations et les capacités du microprocesseur, cette écriture raccourcie ira de pair avec une exécution très rapide. Il faut d'ailleurs remarquer que les processeurs RISC tirent pleinement parti de ce type d'opération.

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

int main(int argc, char* argv[])
{
    double d[]={3, 5.2, 8, 4 };
    valarray<double> v(d,4);
    v+=10; // ajoute 10 à chaque élément

    return 0;
}
```

La bibliothèque standard offre également le support des matrices (`slice_array`) formées à partir de `valarray` et après une étape de structuration appelée slice.

5. Des apports du C++ moderne

Le langage C++ fait l'objet de révisions et d'innovations. Depuis la version de référence dite C++ 98 publiée en 1998, d'autres versions l'ont propulsé vers la modernité. Les standards C++ 11, C++ 14, C++ 17 et maintenant C++ 20 comprennent des fonctionnalités très puissantes jusque-là disponibles uniquement dans d'autres langages initialement conçus bien après C++.

Nous présentons ici de nouvelles possibilités en rapport avec le langage et l'algorithmique.

a. Les lambda-expressions

En C++, une lambda-expression (ou **lambda**) est une fonction anonyme (désignée par le terme *fermeture* ou *closure* en anglais) - appelée à l'endroit même où elle est définie. C'est un moyen pratique pour décrire des petits algorithmes sans chercher à rendre ces fonctions réutilisables.

La lambda est une notation composée des éléments suivants :

```
[capture] (paramètres) mutable exception { corps }
```

La capture permet à la lambda de travailler avec les variables déclarées dans la portée qui la définit, en référence **&** ou en valeur **=**. La capture par défaut peut être par valeur **=**, ce qui signifie que la lambda reçoit une "copie" des variables apparaissant dans le corps d'instruction. Si la capture est par référence **&**, la lambda a la faculté de modifier les variables.

Les paramètres sont optionnels, bien que fréquemment une lambda reçoive des éléments à tester, à comparer, à évaluer, à traiter etc.

Le mot-clé **mutable** est aussi optionnel. En règle générale, l'opérateur d'appel à une lambda est **const** par valeur mais la présence de **mutable** annule cette règle, en permettant au corps d'instruction de modifier localement des variables capturées par valeur.

L'indication d'exception par **throw** ou **noexcept** est facultatif comme pour l'ensemble des fonctions C++.

Le type de retour est déduit par le compilateur à partir du corps. Par défaut, il s'agit de **void**, à moins qu'une instruction **return expression** permette au compilateur de déduire un type particulier.

Etudions sans tarder la fonction **ends_with()** qui teste la terminaison d'une chaîne :

```
// vérifie si chaine se termine par suffixe (exemple chaine =
test.txt se termine par suffixe = .txt)
inline bool ends_with(std::string const& chaine, std::string
const& suffixe)
{
    if (suffixe.size() > chaine.size())
        return false;

    return std::equal(suffixe.rbegin(), suffixe.rend(), chaine.rbegin());
}
```

Cette fonction s'emploie dans la fonction `main()` pour tester un nom de fichier :

```
int main()
{
    string s = "document 1.txt";
    cout << s << " fichier texte ? " << (ends_with(s, ".txt") ?
"oui" : "non") << endl;
}
```



D'une écriture très directe, la fonction de test `ends_with()` n'a rien de particulier, elle est anodyne et il est de peu d'intérêt de l'inscrire au panthéon des fonctions à garder sous la main. Bref on ne s'en resserra sans doute jamais, il est inutile de la mettre dans un super framework !

Elle est tellement simple qu'on va la débaptiser en lui retirant son nom ; dès lors il faut s'en servir au moment où on la définit :

```
string fichier = "document 2.txt";
cout << fichier << " fichier texte ? " <<
(
    [](string chaine, string suffixe) {
        if (suffixe.size() > chaine.size())
            return false;
    })
```

```

        return std::equal(suffixe.rbegin(), suffixe.rend(),
chaine.rbegin());
    }
    (fichier, ".txt")

    ? " oui " : " non") << endl;

```

La fonction est devenue une lambda-expression anonyme. Elle est appelée avec les paramètres `fichier` et `".txt"`.



Nous allons voir un peu plus loin l'intérêt des lambdas pour alléger l'écriture des programmes traitant des collections d'objets.

b. L'inférence de type

Le sens du mot-clé `auto` a été modifié depuis C++ 11. Initialement issu du langage C, ce mot-clé ordonnait au compilateur de stocker une variable dans un espace automatique, autrement dit la pile comme pour les variables locales.

Depuis, le terme a pris un autre sens ; le mot-clé `auto` laisse le soin au compilateur de déduire le type d'une variable. C'est ce qu'on appelle l'inférence de type.



Le paramètre `/Zc:auto(-)` de MSVC contrôle la signification du mot-clé `auto`.

L'inférence de type est utilisée pour simplifier les notations, notamment lorsque des modèles sont en jeu. Prenons l'exemple de parcours de collections d'objets pour appréhender le fonctionnement du mot-clé `auto` :

```

class Personne {
public:
    string prenom;
    int age;

    Personne() : age(0) {
        prenom = "";
    }

    Personne(string nom,int weight) : age(weight), prenom(nom) {

    }
};
int main()
{
    vector<Personne> personnes;
    personnes.push_back(Personne("Quentin",40));
    personnes.push_back(Personne("Claudia", 19));
    personnes.push_back(Personne("Mehdi",20));
    personnes.push_back(Personne("Sybille",16));
    personnes.push_back(Personne("Clementine",31));
    personnes.push_back(Personne("Remi", 15));

    cout << "liste des personnes (version classique) :." << endl;
    for (vector<Personne>::iterator p = personnes.begin(); p <
personnes.end(); p++)
    {
        cout << (*p).prenom << " " << (*p).age << " ans\n";
    }

    cout << "liste des personnes (version auto) :." << endl;
    for (auto p = personnes.begin(); p < personnes.end(); p++)
    {
        cout << (*p).prenom << " " << (*p).age << " ans\n";
    }
}

```

L'écriture de la deuxième boucle (version auto) est nettement plus simple. Pour le compilateur elles sont identiques puisque dans chaque version la variable `p` porte le même type. Dans le premier cas, le développeur indique explicitement le type, dans le

second cas, c'est le compilateur qui le déduit.

c. De nouveaux types de boucles

Le langage C++ connaît enfin l'instruction de parcours de plage, aussi appelée `foreach` dans certains langages. La syntaxe C++ s'appuie plutôt sur une forme d'instruction `for`. L'utilisation combinée avec le mot-clé `auto` n'est pas nécessaire, mais elle rend le programme bien plus concis :

```
cout << "liste des personnes :" << endl;
for (auto& e : personnes)
{
    cout << e.prenom << " " << e.age << " ans\n";
}
```

Voici maintenant un exemple qui illustre ces trois notions, les lambda-expressions, l'inférence de type et les boucles de parcours de plage :

```
// inférence de type vers une lambda
auto comp = [](const Personne& w1, const Personne& w2)
{
    return w1.age < w2.age;
};
sort(personnes.begin(), personnes.end(), comp);

// parcours du tableau trié
cout << "personnes par age :" << endl;
for (auto i = personnes.begin(); i < personnes.end(); i++)
{
    cout << (*i).prenom << " " << (*i).age << " ans" << endl;
}
cout << endl;

int max = 18;

auto mineurs = begin(personnes);
while (mineurs != end(personnes)) {
```



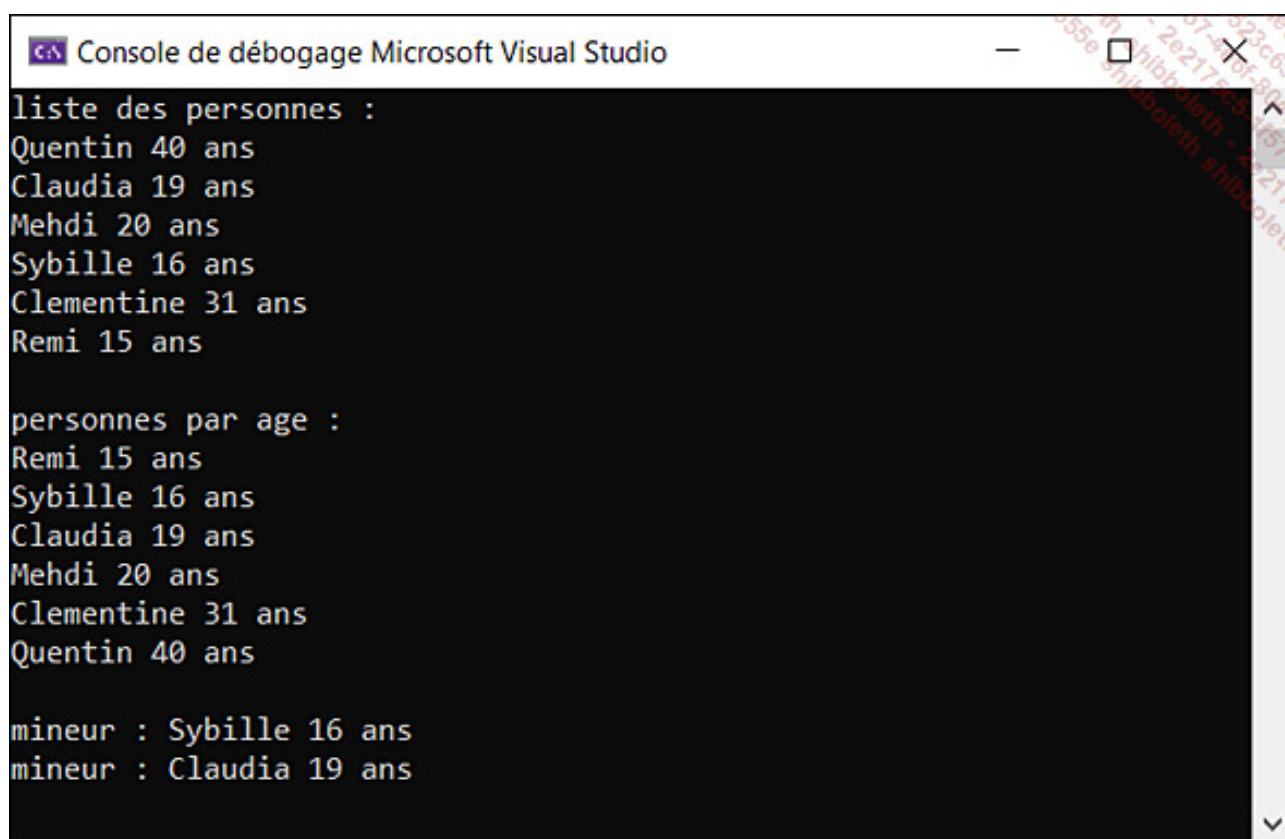
```

mineurs = find_if(mineurs,
    end(personnes),
    [](Personne p) {
        return p.age < max;
    });

if (mineurs == end(personnes))
    break;

mineurs++;
cout << "mineur : " << (*mineurs).prenom << " " <<
(*mineurs).age << " ans" << endl;
}

```



```

Microsoft Visual Studio
liste des personnes :
Quentin 40 ans
Claudia 19 ans
Mehdi 20 ans
Sybille 16 ans
Clementine 31 ans
Remi 15 ans

personnes par age :
Remi 15 ans
Sybille 16 ans
Claudia 19 ans
Mehdi 20 ans
Clementine 31 ans
Quentin 40 ans

mineur : Sybille 16 ans
mineur : Claudia 19 ans

```

d. Des pointeurs intelligents

Le terme d'origine est **smart pointer**, que l'on peut traduire par pointeur intelligent, élégant,

"bien conçu". Ces pointeurs sont des objets destinés à simplifier la gestion de la mémoire, à rendre la rédaction des programmes plus sûrs.

Nous avons étudié les mots-clés pour créer et détruire des objets, `new` et `delete`. Le programmeur C++ a la charge de veiller à supprimer les objets lorsque c'est le bon moment, autrement dit quand on n'en a plus besoin.

Les langages managés comme Java ou C# disposent d'une gestion automatique de la mémoire avec destruction des objets et compactage du tas. Ce processus est appelé ramasse-miettes (**garbage collector**) et la description de son fonctionnement est très simple. Le garbage collector suit le nombre de références sur un objet. À l'instanciation, le compte vaut 1. Chaque fois qu'une référence d'objet est dupliquée vers une autre variable comme argument de méthode, le compte de référence est incrémenté. Lorsque prend fin la portée d'une variable référence, le compte est décrémenté en même temps que la variable est détruite. Périodiquement, le garbage collector détruit les objets qui ne sont plus référencés (compte = 0), et peut même recycler la mémoire ou la compacter pour optimiser sa réallocation.



Les langages C# et Java n'autorisent pas l'accès direct à la mémoire par pointeur. Il n'y a pas de moyen de connaître l'adresse d'un objet. Seules les références sont autorisées, sans valeur numérique. Cette contrainte est nécessaire au fonctionnement du garbage collector qui ne peut pas rentrer en conflit avec une détermination algorithmique des adresses. Certains développeurs trouvent que c'est un avantage en termes de simplicité de rédaction des programmes et de sûreté d'exécution, d'autres considèrent au contraire que le garbage collector altère les performances et la vitesse d'exécution du programme.

Les smart pointers C++ adoptent la même logique. Pour suivre les références vers un objet, les opérations d'instanciation, de référencement et de destruction sont déléguées à ces pointeurs d'un nouveau genre.

Considérons l'exemple suivant élaboré avec les mots-clés standards `new` et `delete` :

```

void calculer_moyenne() {
    cout << "calcul de moyenne\n";

    // instantiation standard de la classe Personne
    Personne* p1 = new Personne("Clementine", 20);
    Personne* p2 = new Personne("Naima", 24);

    int m = (p1->age + p2->age) / 2;
    cout << p1->to_string() << endl;
    cout << p2->to_string() << endl;

    cout << "moyenne d'age : " << m << " ans." << endl;
    cout << endl;

    // destruction standard des objets
    // ne pas oublier sous peine de fuites mémoires
    delete p1;
    delete p2;
}

```

Dans la version smart, les objets sont détruits automatiquement à la fin de la méthode. Les destructeurs des objets smart pointers s'occuperont d'appeler `delete`. Pour suivre les étapes d'instanciation et de destruction, la classe `Personne` affiche des messages spécifiques :

```

class Personne {
public:
    string nom;
    int age;

    Personne() : nom(""), age(0) {

    }

    Personne(string nom, int age) : nom(nom), age(age) {
        cout << "instanciation de " << nom << endl;
    }
}

```

```

string to_string() {
    return nom.append(" ").append(std::to_string(age)).append(" ans");
}

~Personne() {
    cout << "destruction de " << nom << endl;
}
};

```

Le code de la version smart est similaire à l'exemple initial, si ce n'est l'emploi des nouveaux objets pointeurs `unique_ptr<T>` :

```

void calculer_moyenne_smart() {
    cout << "calcul de moyenne en utilisant des smart pointers\n";

    unique_ptr<Personne> p1 (new Personne("Clementine", 20));
    unique_ptr<Personne> p2 (new Personne("Naima", 24));

    int m = (p1->age + p2->age) / 2;
    cout << p1->to_string() << endl;
    cout << p2->to_string() << endl;

    cout << "moyenne d'age : " << m << " ans." << endl;
    cout << endl;
}

```

À l'exécution nous observons la destruction des objets sans appel explicite à `delete` :

```

Microsoft Visual Studio Debug Console
calcul de moyenne en utilisant des smart pointers
instanciation de Clementine
instanciation de Naima
Clementine 20 ans
Naima 24 ans
moyenne d'age : 22 ans.

destruction de Naima 24 ans
destruction de Clementine 20 ans

```

Découvrons maintenant d'autres possibilités : instanciation par l'intermédiaire de la fonction

`make_unique<T>()`, accès au pointeur brut (avec une adresse classique), et passage par référence :

```
// autre forme d'instanciation
unique_ptr<Personne> p3 = make_unique<Personne>("Henri", 13);

// récupérer un pointeur brut
Personne* p3_brut = p3.get();

// appeler une méthode en passant une référence
afficher_detail(*p3);
```

La méthode `afficher_detail` reçoit comme paramètre un `const Personne&`. Elle ne peut pas modifier l'objet passé en argument :

```
void afficher_detail(const Personne& p)
{
    cout << p.nom << " " << to_string(p.age) << " ans." << endl;
}
```

En complément des pointeurs uniques `unique_ptr`, il existe des pointeurs partagés `shared_ptr` à références comptabilisées. Ils suivent l'utilisation des pointeurs bruts par plusieurs propriétaires. Cela permet, par exemple, de renvoyer une copie d'un objet tout en conservant l'original. L'objet sous-jacent est détruit quand toutes les propriétés sont hors portée ou quand a cessé la propriété exercée sur l'objet. Les pointeurs faibles `weak_ptr` sont aussi partagés mais ne participent pas au décompte des références.

```
void partager() {
    shared_ptr<Personne> p1(new Personne("Alice", 35));
    shared_ptr<Personne> p2 = make_shared<Personne>("Alan", 52);

    shared_ptr<Personne> p3(p1); // copie, 2 références sur p1
    shared_ptr<Personne> p4 = p3; // copie, 3 références sur p1

    // une collection d'objets partagés
    vector<shared_ptr<Personne>> list;
```

```
list.push_back(shared_ptr<Personne>(new Personne("Ellis",60)));  
}
```

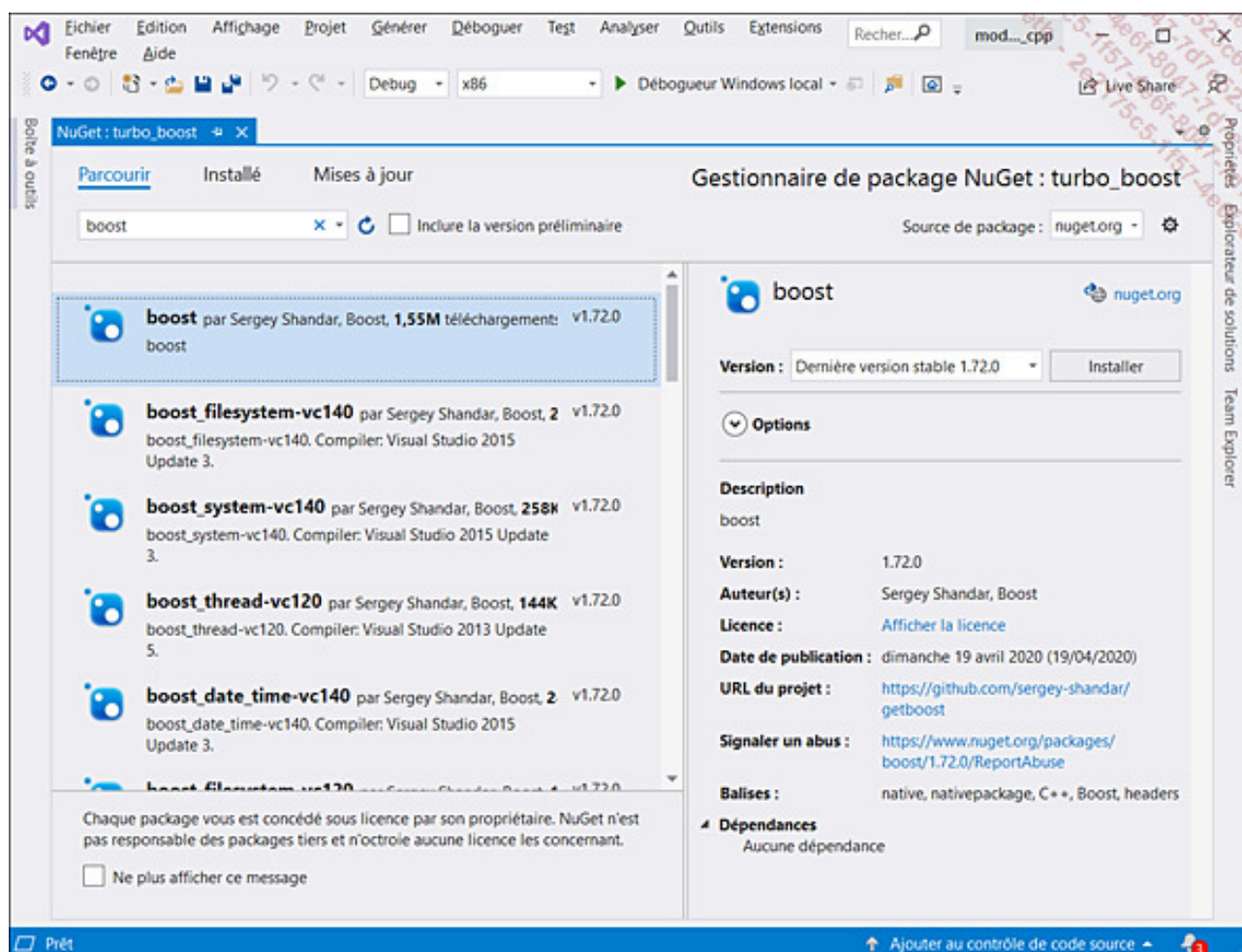
6. Introduction à la bibliothèque boost

La bibliothèque **boost** n'est pas un concurrent de la STL, mais plutôt un complément ou son prolongement. De fait, plusieurs modules du référentiel boost ont été intégrés dans la bibliothèque standard au fil des versions C++ 14 à C++ 17. Les deux bibliothèques STL et boost fonctionnent donc très bien ensemble.

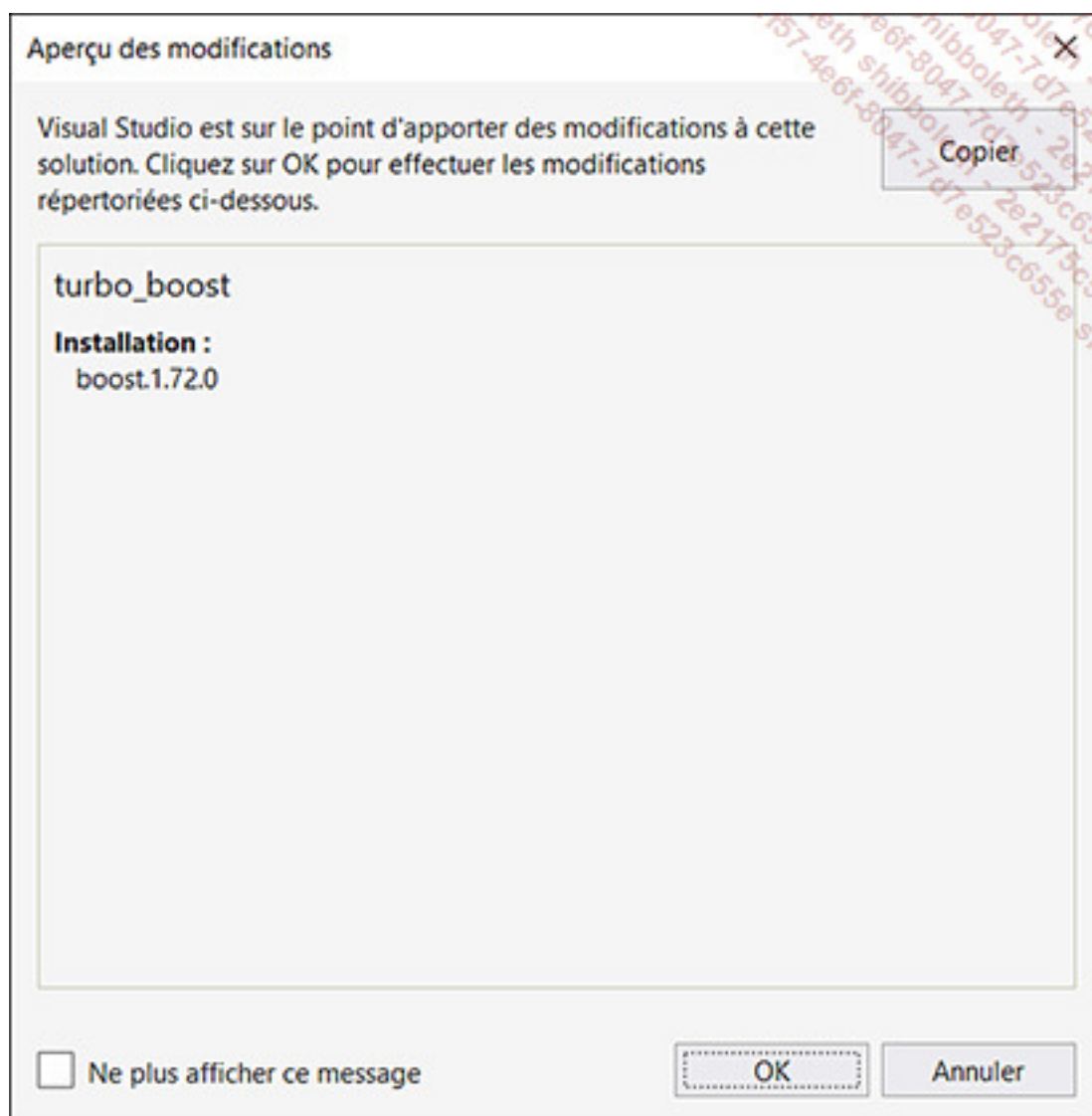
a. Installation de la bibliothèque

Il existe plusieurs moyens d'installer la bibliothèque au sein d'un projet. Sous Linux, on peut suivre les instructions du site <https://boost.org/> pour récupérer un package de sources ou de binaires.

Depuis Visual Studio, le moyen le plus direct consiste à télécharger un package NuGet. Depuis le menu **Projet**, utiliser la commande **Gérer les packages NuGet**. Indiquer **boost** dans la zone de recherche de l'onglet **Parcourir**. Le catalogue des packages présente alors la bibliothèque complète ainsi que des volumes de cette bibliothèque.



Après avoir cliqué sur le bouton **Installer** le gestionnaire de package demande de confirmer l'installation de la bibliothèque au sein du projet :



La confirmation déclenche le processus de téléchargement du package NuGet (au niveau de la solution Visual Studio) et d'installation de la bibliothèque au sein du projet.

b. Un premier exemple avec boost

Cet exemple reste dans le domaine des algorithmes et des collections. Il montre comment les lambdas de boost facilitent l'écriture de boucles de parcours d'une collection :

```
#include <boost/lambda/lambda.hpp>
#include <iostream>
#include <iterator>
```



```

#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<int> tab;
    tab.push_back(1);
    tab.push_back(2);
    tab.push_back(3);
    tab.push_back(4);

    using namespace boost::lambda;

    std::for_each(
        tab.begin(),
        tab.end(),
        std::cout << _1 << "**2 = " << (_1 * _1) << "\n");
}

```

La boucle `for_each` itère sur l'ensemble de la collection `tab`. Dans la lambda, l'argument anonyme `_1` représente le *xième* objet de la collection.



```

Microsoft Visual Studio Console
1**2 = 1
2**2 = 4
3**2 = 9
4**2 = 16

Sortie de C:\Users\gueri\source\repos\modern_cpp\Debug\turbo_boost.exe (processus 35724). Code : 0.
Appuyez sur une touche pour fermer cette fenêtre. . .

```

D'autres instructions de contrôle sont proposées sous la forme de lambda. L'exemple qui suit teste la parité de la valeur entière issue de la collection :

```

// utilise la lambda if_then_else dans la boucle for_each
std::for_each(
    tab.begin(),
    tab.end(),

```

```

    if_then_else(_1 % 2 == 0, cout << _1 << " pair\n", cout << _1
    << " impair\n")
);

```



Console de débogage Microsoft Visual Studio

```

1 impair
2 pair
3 impair
4 pair

```

c. Domaines d'application

La STL a été initialement conçue par Bjarne Soustrup pour répondre à un besoin d'ingénierie logicielle dans le domaine des télécoms. Depuis elle s'est progressivement recentrée sur les domaines liés aux algorithmes.

L'étendue de la librairie boost est comparativement beaucoup plus large. On recense des modules d'algorithmie, mais aussi de gestion du réseau, de prise en charge graphique, de gestion multithread, de jeux vidéo, de multimédia...

Rappelons que boost n'est pas incompatible avec la STL et que certains modules ont depuis été standardisés.

L'apport de boost se révèle donc particulièrement riche pour le développeur C++ qui peut sortir du cadre de la programmation procédurale et s'appuyer sur de puissants composants pour bâtir des applications en prise avec l'environnement technique, le monde réel en quelque sorte.

Voici quelques domaines supportés par boost :

RAII et gestion mémoire	Chaînes (string, expressions régulières...)	Conteneurs et collections	Structures de données composites (tuples, variant...)
Algorithmes et graphes	Communications	Flux et fichiers	Gestion des dates et du temps
Programmation fonctionnelle	Programmation parallèle	Programmation générique	Extensions du langage
Gestion des erreurs	Gestion des nombres	Design patterns	Graphismes