C++

La STL

Fonctions génériques

Enumérer tous les types de bases ?

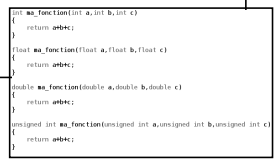
```
int ma_fonction(int a,int b,int c)
{
    return a+b+c;
}

float ma_fonction(float a,float b,float c)
{
    return a+b+c;
}

double ma_fonction(double a,double b,double c)
{
    return a+b+c;
}

unsigned int ma_fonction(unsigned int a,unsigned int b,unsigned int c)
{
    return a+b+c;
}

...
```


Fonctions génériques

Pire: Classes, multiplicité potentiellement infinie

```
struct vec2
{
    vec2(float x_param,float y_param):x(x_param),y(y_param){}
    float x,y;
};
vec2 operator+(const vec2& a,const vec2& b)
{
    return vec2(a.x+b.x,a.y+b.y);
}
```

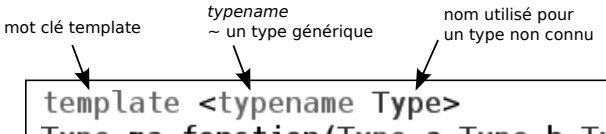
```
vec2 ma_fonction(const vec2& a,const vec2& b,const vec2& c)
{
    return a+b+c;
}
```



...

Fonctions génériques

Les templates




```
template <typename Type>
Type ma_fonction(Type a,Type b,Type c)
{
    return a+b+c;
}
```

Rem. **Type** peut prendre n'importe quel nom.
On retrouve généralement **T**

Utilisation fonctions template

```
template <typename Type>
Type ma_fonction(Type a,Type b,Type c)
{
    return a+b+c;
}
```

```
int main()
{
    {
        int a=5,b=4,c=8;
        int d=ma_fonction(a,b,c);
        std::cout<<d<<std::endl;
    }
    {
        float a=5.4,b=4.2,c=8.9;
        float d=ma_fonction(a,b,c);
        std::cout<<d<<std::endl;
    }
    {
        double a=5.4,b=4.2,c=8.9;
        double d=ma_fonction(a,b,c);
        std::cout<<d<<std::endl;
    }
}
```

Type est remplacé par int

Type est remplacé par float

Type est remplacé par double

Le type de "Type" est inféré à partir des arguments

Utilisation fonctions template

```
template <typename Type>
Type ma_fonction(Type a,Type b,Type c)
{
    return a+b+c;
}
```

Il est possible d'expliciter le type template

```
int main()
{
    {
        int a=5,b=4,c=8;
        int d=ma_fonction<int>(a,b,c);
        std::cout<<d<<std::endl;
    }
    {
        float a=5.4,b=4.2,c=8.9;
        float d=ma_fonction<float>(a,b,c);
        std::cout<<d<<std::endl;
    }
    {
        double a=5.4,b=4.2,c=8.9;
        double d=ma_fonction<double>(a,b,c);
        std::cout<<d<<std::endl;
    }
}
```

On "force" le type plutôt que de l'inférer

Utilisation fonctions template

```
template <typename Type>
Type ma_fonction(Type a,Type b,Type c)
{
    return a+b+c;
}
```

Fonctionne pour tout les types possibles
(qui implémentent l'opérateur +)

```
int main()
{
    {
        vec2 a(1.2,3.4),b(4.5,7.8),c(-1.1,2.1);
        vec2 d=ma_fonction(a,b,c);
        std::cout<<d.x<<" "<<d.y<<std::endl;
    }
}
```


Utilisation fonctions template

On peut éviter les copies inutiles
(fonctionne pour les types de bases également)

```
template <typename Type>
Type ma_fonction(const Type& a,const Type& b,const Type& c)
{
    return a+b+c;
}
```

```
int main()
{
    {
        vec2 a(1.2,3.4),b(4.5,7.8),c(-1.1,2.1);
        vec2 d=ma_fonction(a,b,c);
        std::cout<<d.x<<" "<<d.y<<std::endl;
    }
}
```

Fonction template

On retrouvera classiquement la syntaxe suivante

```
template <typename T>
T ma_fonction(const T& a,const T& b,const T& c)
{
    return a+b+c;
}
```


Classes génériques

Obligation d'un nom différent

```
struct vec2f
{
    float x,y;

    vec2f(float x_param,float y_param):x(x_param),y(y_param){}
};
struct vec2d
{
    double x,y;

    vec2d(double x_param,double y_param):x(x_param),y(y_param){}
};
struct vec2i
{
    int x,y;

    vec2i(int x_param,int y_param):x(x_param),y(y_param){}
};
```

Utilisation

```
int main()
{
    vec2d a0(1.2,4.5);
    vec2f a1(1.4,4.5);
    vec2i a2(1,4);
}
```


Classes templates

```
template <typename T>
struct vec2
{                       Peut utiliser T partout dans la classe
    T x,y;
    vec2(T x_param,T y_param):x(x_param),y(y_param){}
};
```

Utilisation

```
int main()
{
    vec2<float> a0(1.2,4.5);
    vec2<double> a1(1.4,4.5);
    vec2<int> a2(1,4);
}
```

Obligation de préciser explicitement le type

Classes templates

```
template <typename T>
struct vec2
{                       Peut utiliser T partout dans la classe
    T x,y;
    vec2(T x_param,T y_param):x(x_param),y(y_param){}
};
```

Très générique: peut créer un vecteur de vecteur, ...

Ex.

```
int main()
{
    vec2<vec2<float> > a(vec2<float>(1.2,4.5),
                         vec2<float>(7.4,-1.4));
    std::cout<<a.x.y<<std::endl;
}
```

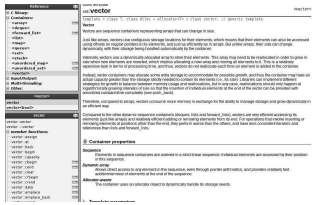
La STL

STL = **ST**andard **L**ibrary

- Des conteneurs génériques
- Des algorithmes génériques

Basées templates

http://www.cplusplus.com/reference/



La STL, les +/-

La STL est:

- La bibliothèque standard. Elle fait partie du C++
- Très générique
- Robuste
- Optimisée pour chaque architecture

Il faut utiliser la STL dès que possible

Il ne faut pas re-coder des équivalents

Moins robustes
Moins génériques
Moins rapides

à bannir
~~Listes chainées "maison"~~
~~Files à priorités "maison"~~

Mais

- La STL est technique/complexe à utiliser
- La syntaxe est longue et technique

Il faut prendre l'habitude

C++11 simplifie des choses

Le std::string

- Stockage d'une chaine de caractères

  Sans les inconvénients des pointeurs

- Remplace avantageusement le char[], et char*

Ne plus utiliser char* en C++

sauf compatibilité C

```
int main()
{
    std::string mot="ma maison";

    mot+=" a moi.";

    std::cout<<mot<<std::endl;

    int N=mot.size();
    mot[N-1]='!';

    for(int k=N-1;k>=0;--k)
        std::cout<<mot[k]<<std::endl;

    return 0;
}
```

initialisation dimensionnement automatique

concatenation (redimensionnement)

nombre de caractères

accès individuel aux caractères

Le std::string

Conversion aisées (en C++11)

convertion nombre vers string

```
std::string a=std::to_string(1.512+0.25);
float val=std::stof("12"+a);

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

convertion string vers nombre

```
stoi
stol
stoul
stoll
stoull
stof
stod
stold
```

Le std::vector

- Eléments contigus en mémoire
- Accès/utilisation identique à un tableau classique
- Redimensionne automatiquement lors de l'ajout en fin

équivalent du new[N] en mieux

```
#include <vector>


int main()
{
    std::vector<int> a;

    a.push_back(8);
    a.push_back(6);
    a.push_back(7);

    std::cout<<a[1]<<std::endl;

    return 0;
}
```

déclaration conteneur sur entier

paramètre template

ajout en fin

accès sur 2nd élément

Le std::vector

Exemple:

```
int main()
{
    std::vector<int> a;

    a.push_back(8);
    a.push_back(6);
    a.push_back(7);

    int k=0;
    int N=a.size();
    for(int k=0;k<N;++k)
        a[k] = 2*a[k]-3;

    for(int k=N-1;k>=0;--k)
        std::cout<<k<<":"<<a[k]<<std::endl;

    return 0;
}
```

nombre d'éléments

Le std::vector

Pas que des entiers:

```
int main()
{
    std::vector<std::string> a;

    a.push_back("je suis");
    a.push_back("etudiant");
    a.push_back("a CPE");

    int k=0;
    int N=a.size();
    for(int k=0;k<N;++k)
        std::cout<<a[k]<<std::endl;

    return 0;
}
```


Le std::vector

Classes personnelles:

```
struct vec2
{
    float x,y;
    vec2(float x_arg,float y_arg):x(x_arg),y(y_arg){}
};

int main()
{
    std::vector<vec2> a;

    a.push_back(vec2(1.2,3.3));
    a.push_back(vec2(4.7,5.6));
    a.push_back(vec2(8.1,-1.2));

    int k=0;
    int N=a.size();
    for(int k=0;k<N;++k)
        std::cout<<a[k].x<<";"<<a[k].y<<std::endl;

    return 0;
}
```

Le std::vector

Compatibilité avec les tableaux et pointeurs C.

```
int main()
{
    std::vector<float> v;
    v.push_back(1.4f);
    v.push_back(5.6f);
    v.push_back(-1.2f);

    float* pv=&v[0];

    std::cout<<*pv<<std::endl;
    ++pv;
    std::cout<<*pv<<","<<pv[1]<<std::endl;

    return 0;
}
```

adresse du 1er élément

pointeur classique

Le std::vector

Opérations principales

- `std::vector<Type> v` initialisation
- `std::vector<Type> v(N)` initialisation avec N cases vides
- `v.push_back(x)` ajout en fin
- `v.size()` nombre d'éléments
- `v.resize(N)` redimensionnement manuel
- `v[k]` accès lecture/ecriture élément k

Le std::vector

Opérations rapides (en O(1))

- Ajout en fin (sauf redimensionnement, O(N)) *v.push_back(x);*
- Accès aléatoire *v[k]*

Autres opérations
insertions, suppression, ... en O(N)

copie du tableau

Le std::list

- Eléments chaînés (doublement)
- Eléments non contigus en mémoire

Ne pas recoder de listes chaînées!

```
#include <list>

int main()
{
    std::list<int> a;

    a.push_back(4);
    a.push_front(10);
    a.push_front(-7);

    return 0;
}
```

initialisation

ajout en tête

ajout en fin

Accès aux éléments

Comment accéder au 1er élément?
Comment passer au suivant?

En C, pas de standard

```
struct node
{
    int value;
    node* next;
};
```

n.value
n.x
n.next
n.suivant
n.?

Généraliser pour des arbres?

n.gauche
n.droite
n.next_left
n.next_right
n.previous
n.?

Itérateur

La STL introduit la notion d'itérateur

Permet de se déplacer dans les conteneurs
et d'accéder à la valeur des éléments

Avantage: Parcours générique (ne dépend pas du conteneur vecteur, list, arbre, ...)

Inconvénient: Syntaxe lourde

Itérateur

- Classe interne à un conteneur *(std::list<int>::iterator it;)*
- Définit l'opérateur * et ++

* : donne la valeur *(*it)*
++: passe au suivant *(++it)*

Syntaxe similaire aux pointeurs

Attention! itérateur n'est pas un pointeur

valeur=*it
&valeur != it

Iterator Java connaissent leurs domaines de validité (has next), pas en C++ !

Itérateur

Parcours d'une liste

```
int main()
{

    std::list<int> a;

    a.push_back(4);
    a.push_front(10);
    a.push_front(-7);

    std::list<int>::iterator it;

    it=a.begin();

    std::cout<<*it<<std::endl;
    ++it;
    std::cout<<*it<<std::endl;
    ++it;
    std::cout<<*it<<std::endl;

    return 0;
}
```

```
-7
10
 4
```

Itérateur

Parcours générique

```
int main()
{

    std::list<int> a;

    a.push_back(4);
    a.push_front(10);
    a.push_front(-7);

    std::list<int>::iterator it,it_end;

    it=a.begin();
    it_end=a.end();

    for(;it!=it_end;++it)
    {
        int value=*it;
        std::cout<<value<<std::endl;
    }

    return 0;
}
```

sentinelle de fin
indique la fin de la liste

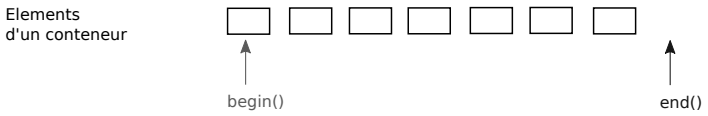
Itérateur

Parcours générique: version condensée

```
int main()
{

    std::list<int> a;

    a.push_back(4);
    a.push_front(10);
    a.push_front(-7);


    for(std::list<int>::iterator it=a.begin(),it_end=a.end();
        it!=it_end;++it)
    {
        int value=*it;
        std::cout<<value<<std::endl;
    }

    return 0;
}
```


Itérateurs particuliers

Chaque conteneur définit des itérateurs particuliers

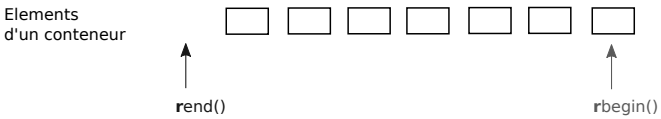


- `begin()` itérateur sur le 1er élément
- `end()` sentinelle de fin de conteneur
  (*it ne désigne pas un élément valide)

```
conteneur<type> c;
conteneur<type>::iterator it ,it_end;
for(it=c.begin(),it_end=c.end(); it!=it_end ; ++it)
{
   ...
}
```


Itérateurs particuliers

Parcours dans le sens inverse



- `rbegin()` (reverse) itérateur sur le dernier élément
- `rend()` sentinelle de fin de conteneur (avant le 1er élément)
  (*it ne désigne pas un élément valide)

```
conteneur<type> c;
conteneur<type>::iterator it ,it_end;
for(it=c.rbegin(),it_end=c.rend(); it!=it_end ; ++it)
{
   ...
}
```

parcours du dernier au premier

Itérateurs constants

On souhaite souvent parcourir les éléments sans les modifiers

En C, on a
- `int*` pointeur vers une valeur modifiable
- `const int*` pointeur vers une valeur non modifiable

En C++, on a
- `iterator` itérateur vers une valeur modifiable
- `const_iterator` itérateur vers une valeur non modifiable

- `cbegin()` const_itérateur sur le dernier élément
- `cend()` sentinelle de fin de conteneur (avant le 1er élément)
  (*it ne désigne pas un élément valide)
- `crbegin()` (reverse) const_itérateur sur le dernier élément
- `crend()` sentinelle de fin de conteneur (avant le 1er élément)
  (*it ne désigne pas un élément valide)

Itérateurs

Parcours sans modification

```
conteneur<type> c;
conteneur<type>::const_iterator it ,it_end;
for(it=c.cbegin(),it_end=c.cend(); it!=it_end ; ++it)
{
   ...
}
```

Parcours avec modification

```
conteneur<type> c;
conteneur<type>::iterator it ,it_end;
for(it=c.begin(),it_end=c.end(); it!=it_end ; ++it)
{
   ...
}
```

Bonne programmation:
Toujours préférer le const_iterator si possible

Différents types d'itérateurs

Les conteneurs ont des itérateurs avec +/- de possibilités

- Forward iterator `*it` `++it` (std::unordered_map)
- Backward iterator `--it` (std::list)
- Random access `it+n` `it-n` `it+=n` `it-=n` `it-it2` `it[n]` (std::vector)

Exercice listes

- Déclarer une liste contenant 4 nombres flottants
- Multiplier par 2 l'ensemble des valeurs de la liste

Fonctions sur listes

`size()`	nombre d'éléments
`push_front()`	ajoute en début
`push_back()`	ajoute en fin
`pop_front()`	suppression en début
`pop_back()`	suppression en fin
`insert(iterator,value)`	insère valeur avant l'itérateur
`erase(iterator)`	Supprime l'élément
`clear()`	Supprime l'ensemble des éléments
`remove(value)`	Supprime tous les éléments ayant une valeur donnée

std::list, efficacité

Opérations rapides (en O(1))

- Ajout quelconque — *push_back(); push_front(); insert();*
- Suppression — *erase()*

Autres opérations
parcours, accès en O(N)

Savoir lire la documentation

class template
std::**list** <list>

```
template < class T, class Alloc = allocator<T> > class list;
```

List

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the element preceding it and a link to the element following it.

They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient.

Compared to other base standard sequence containers (array, vector and deque), lists perform generally better in inserting, extracting and moving elements in any position within the container for which an iterator has already been obtained, and therefore also in algorithms that make intensive use of these, like sorting algorithms.

The main drawback of `list`s and forward_lists compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a `list` one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

Container properties

Sequence
Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Doubly-linked list
Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.

Allocator-aware
The container uses an allocator object to dynamically handle its storage needs.

Template parameters

`T`
Type of the elements.
Aliased as member type `list::value_type`.

`Alloc`
Type of the allocator object used to define the storage allocation model. By default, the allocator class template is used, which defines the simplest memory allocation model and is value-independent.
Aliased as member type `list::allocator_type`.

Savoir lire la documentation

Member functions

(constructor)	Construct list (public member function)
(destructor)	List destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin (C++11)	Return const_iterator to beginning (public member function)
cend (C++11)	Return const_iterator to end (public member function)
crbegin (C++11)	Return const_reverse_iterator to reverse beginning (public member function)
crend (C++11)	Return const_reverse_iterator to reverse end (public member function)

Capacity:

empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

Modifiers:

assign	Assign new content to container (public member function)
emplace_front (C++11)	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back (C++11)	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace (C++11)	Construct and insert element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Liste des fonctions de la classe

Savoir lire la documentation

public member function

std::**list::insert** <list>

C++98 | C++11

```
single element (1)  iterator insert (const_iterator position, const value_type& val);
fill (2)            iterator insert (const_iterator position, size_type n, const value_type& val);
range (3)           template <class InputIterator>
                    iterator insert (const_iterator position, InputIterator first, InputIterator last);
move (4)            iterator insert (const_iterator position, value_type&& val);
initializer list (5) iterator insert (const_iterator position, initializer_list<value_type> il);
```

Insert elements

The container is extended by inserting new elements before the element at the specified *position*.

This effectively increases the list size by the amount of elements inserted.

Unlike other standard sequence containers, list and forward_list objects are specifically designed to be efficient inserting and removing elements in any position, even in the middle of the sequence.

The arguments determine how many elements are inserted and to which values they are initialized:

Parameters

`position`
Position in the container where the new elements are inserted.
`iterator` is a member type, defined as a bidirectional iterator type that points to elements.

`val`
Value to be copied (or moved) to the inserted elements.
Member type `value_type` is the type of the elements in the container, defined in list as an alias of its first template parameter (T).

`n`
Number of elements to insert. Each element is initialized to a copy of *val*.
Member type `size_type` is an unsigned integral type.

`first, last`
Iterators specifying a range of elements. Copies of the elements in the range (`first`, `last`) are inserted at *position* (in the same order).
Notice that the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.
The function template argument `InputIterator` shall be an input iterator type that points to elements of a type from which `value_type` objects can be constructed.

`il`
An initializer_list object. Copies of these elements are inserted at *position* (in the same order).
These objects are automatically constructed from *initializer list* declarators.
Member type `value_type` is the type of the elements in the container, defined in list as an alias of its first template parameter (T).

Signature de la fonction

Savoir lire la documentation

Example

```
// inserting into a list
#include <iostream>
#include <list>
#include <vector>

int main ()
{
  std::list<int> mylist;
  std::list<int>::iterator it;

  // set some initial values:
  for (int i=1; i<=5; ++i) mylist.push_back(i); // 1 2 3 4 5

  it = mylist.begin();
  ++it;       // it points now to number 2           ^

  mylist.insert (it,10);                        // 1 10 2 3 4 5

  // "it" still points to number 2                     ^
  mylist.insert (it,2,20);                      // 1 10 20 20 2 3 4 5

  --it;       // it points now to the second 20            ^

  std::vector<int> myvector (2,30);
  mylist.insert (it,myvector.begin(),myvector.end());
                                                // 1 10 20 30 30 20 2 3 4 5
                                                //               ^
  std::cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Example d'utilisation

Liste de classes

```
struct vec2
{
    float x,y;
    vec2(float x_arg,float y_arg):x(x_arg),y(y_arg){}
};

int main()
{
    std::list<vec2> C;

    C.push_back(vec2(4.1,2.3));
    C.push_front(vec2(4.0,5.3));
    C.push_front(vec2(7.2,-1.3));

    for(std::list<vec2>::const_iterator it=C.cbegin(),it_end=C.cend();
        it!=it_end;++it)
    {
        const vec2& value=*it;
        std::cout<<value.x<<","<<value.y<<std::endl;
    }

    return 0;
}
```


Exemple de listes

- Créez une liste de 6 mots (std::string)
- Afficher cette liste

- Créez une liste de std::vector d'entiers
  - Le 1er vecteur contiendra [4,8,6]
  - Le 2nd vecteur contiendra [-1,7,7,4]
  - Le 3ème vecteur contiendra [5,7,9]
- Supprimez le second vecteur de la liste
- Afficher tous les éléments les un à la suite des autres

Les set

set = Conteneur d'éléments ordonnés (uniques)

Intérêt: Recherche rapide (dichotomie)
(binary search)

Insertion et suppression en log(N)

Implémentation possible : arbre binaire de recherche

Les std::set, exemple d'initialisation

```
#include <set>

int main()
{
    std::set<int> S;

    S.insert(4);S.insert(-6);S.insert(4);
    S.insert(8);S.insert(-3);S.insert(12);
    S.insert(-7);S.insert(-8);S.insert(0);

    for(std::set<int>::const_iterator it=S.cbegin(),it_end=S.cend();
        it!=it_end;++it)
    {
        int value=*it;
        std::cout<<value<<std::endl;
    }

    return 0;
}
```

Affiche: -8 -7 -6 -3 0 4 8 12

Les nombres sont triés, il n'y a pas de doublons

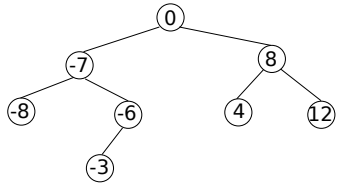
Les std::set, exemple d'initialisation

```
#include <set>

int main()
{
    std::set<int> S;

    S.insert(4);S.insert(-6);S.insert(4);
    S.insert(8);S.insert(-3);S.insert(12);
    S.insert(-7);S.insert(-8);S.insert(0);

    for(std::set<int>::const_iterator it=S.cbegin(),it_end=S.cend();
        it!=it_end;++it)
    {
        int value=*it;
        std::cout<<value<<std::endl;
    }

    return 0;
}
```

Affiche: -8 -7 -6 -3 0 4 8 12

Les nombres sont triés, il n'y a pas de doublons

En interne



Les std::set, recherche d'élement

```
#include <set>

int main()
{
    std::set<int> S;

    S.insert(4);S.insert(-6);S.insert(4);
    S.insert(8);S.insert(-3);S.insert(12);
    S.insert(-7);S.insert(-8);S.insert(0);


    std::set<int>::const_iterator it;
    it=S.find(8);
    if(it==S.end())
        std::cout<<"Pas de 8 dans S"<<std::endl;
    else
        std::cout<<"J'ai trouve un "<<*it
                 <<" en position "<<std::distance(S.begin(),it)<<std::endl;

    return 0;
}
```

std::set de mots

```
std::set<std::string> S;

S.insert("Hugo Victor");
S.insert("De Medicis Catherine");
S.insert("Presley Elvis");
S.insert("Colomb Christophe");
S.insert("Lincoln Abraham");
S.insert("Cesar Jules");
S.insert("Vian Boris");

if(S.find("L'eventreur Jack")==S.end())
    S.insert("L'eventreur Jack");
S.erase("Cesar Jules");
S.insert("Hallyday Johnny");

for(std::set<std::string>::const_iterator it=S.cbegin(),it_end=S.cend();
    it!=it_end;++it)
{
    std::cout<<*it<<std::endl;
}
```

Classement dans l'ordre alphabétique

Classement spécifique

```
struct comparator_inferior
{
    bool operator()(const std::string& word_1,
                    const std::string& word_2) const
    {
        return word_1.size()<word_2.size();
    }
};
```

répond à la question
word_1<word_2 ?

```
std::set<std::string,comparator_inferior> S;

S.insert("Hugo Victor");
S.insert("De Medicis Catherine");
S.insert("Presley Elvis");
S.insert("Colomb Christophe");
S.insert("Lincoln Abraham");
S.insert("Cesar Jules");
S.insert("Vian Boris");

if(S.find("L'eventreur Jack")==S.end())
    S.insert("L'eventreur Jack");
S.erase("Cesar Jules");
S.insert("Hallyday Johnny");

for(std::set<std::string>::const_iterator it=S.cbegin(),it_end=S.cend();
    it!=it_end;++it)
{
    std::cout<<*it<<std::endl;
}
```

Utilise la fonction de comparaison donnée

Classement par nombre de lettres
deux mots ayant le même nombre de lettres sont considérés comme égaux

```
Vian Boris
Presley Elvis
Lincoln Abraham
L'eventreur Jack
Colomb Christophe
De Medicis Catherine
```


Exercice sur le classement

- Soit une structure stockant des vecteurs flottants (x,y)
- Créez une std::set stockant 8 vecteurs ordonnés suivant leurs valeurs y

Attention sur des nombres flottants
égalité difficile à vérifier
find() ne donne pas toujours le résultat attendu

std::set : efficacité

Les opérations sur les std::set sont en O(log(N))

- Moins optimale que O(1)
- Bien plus rapide que O(N) pour N grand

Recherche d'élément
Ajout d'éléments
Suppression d'éléments

recherche base de données
grand nombre d'items classés
...

std::map

On veut souvent des éléments classés par clés et possèdant une valeur

- On peut faire une struct (clé,valeur)
  ordonné suivant la clé
- std::map propose un conteneur spécifique

Structure de donnée très utilisée!!

std::map

```
#include <map>
                    type clé    type valeur
int main()
{
    std::map<int,std::string> M;

    M[4]="valeur 0";
    M[9]="valeur 1";
    M[-12]="valeur 3";
    M[7]="valeur 4";

    std::map<int,std::string>::const_iterator it=M.cbegin(),it_end=M.cend();
    for(;it!=it_end;++it)
    {
        std::cout<<it->first<<" , "<<it->second<<std::endl;
    }
                            clé              valeur
    return 0;
}
```

M[clé]=valeur;

std::map

La clé peut être un mot

```
std::map<std::string,int> age;

age["Romain"]=45;
age["Mathilde"]=13;
age["Fabrice"]=43;
age["Romualde"]=25;

std::map<std::string,int>::const_iterator it=age.cbegin(),it_end=age.cend();
for(;it!=it_end;++it)
{
    const std::string& nom=it->first;
    int age_courant=it->second;

    std::cout<<nom<<" a "<<age_courant<<" ans."<<std::endl;
}
```

- std::map peut aider à la lisibilité du code
  peut jouer le rôle d'une LUT

Utilisation des std::map

Méthodes identiques à std::set

```
std::map<std::string,int> age;

age["Romain"]=45;
age["Mathilde"]=13;
age["Fabrice"]=43;
age["Romualde"]=25;

if(age.find("Martin")==age.end())
    age["Martin"]=16;

age.erase("Fabrice");

std::map<std::string,int>::const_iterator it=age.cbegin(),it_end=age.cend();
for(;it!=it_end;++it)
{
    const std::string& nom=it->first;
    int age_courant=it->second;

    std::cout<<nom<<" a "<<age_courant<<" ans."<<std::endl;
}
```

Exemple d'organisation avec std::map

```
typedef std::map<std::string,float> moyenne_matiere;
typedef std::map<std::string,moyenne_matiere > classeur_note;
classeur_note note;

note["Dupont Marc"]["Informatique"]=12.6;
note["Ulpere Francis"]["Math"]=9.5;
note["Lafon Benjamin"]["Anglais"]=7.75;
note["Dupont Fabrice"]["Informatique"]=11.2;
note["Dupont Marc"]["Math"]=9.5;
note["Ulpere Francis"]["Anglais"]=15;
note["Dupont Marc"]["Anglais"]=7.9;
note["Lafon Benjamin"]["Informatique"]=15.75;


for(classeur_note::const_iterator it=note.cbegin(),it_end=note.cend();
    it!=it_end ; ++it)
{
    const std::string& etudiant=it->first;
    const moyenne_matiere& moyenne=it->second;

    std::cout<<etudiant<<" a obtenu "<<std::endl;

    for(moyenne_matiere::const_iterator it_moyenne=moyenne.cbegin(),
        it_moyenne_end=moyenne.cend() ;
        it_moyenne!=it_moyenne_end ; ++it_moyenne)
    {
        const std::string& matiere=it_moyenne->first;
        float note_examen=it_moyenne->second;

        std::cout<<"  "<<note_examen<<" en "<<matiere<<std::endl;
    }
}
```

les types templates deviennent long à écrire
typedef permet de les renommer

Classé par nom et par matière

```
Dupont Fabrice a obtenu
  11.2 en Informatique
Dupont Marc a obtenu
  7.9 en Anglais
  12.6 en Informatique
  9.5 en Math
Lafon Benjamin a obtenu
  7.75 en Anglais
  15.75 en Informatique
Ulpere Francis a obtenu
  15 en Anglais
  9.5 en Math
```


Exemple d'organisation avec std::map

```
typedef std::map<std::string,float> moyenne_matiere;
typedef std::map<std::string,moyenne_matiere > classeur_note;
classeur_note note;

note["Dupont Marc"]["Informatique"]=12.6;
note["Ulpere Francis"]["Math"]=9.5;
note["Lafon Benjamin"]["Anglais"]=7.75;
note["Dupont Fabrice"]["Informatique"]=11.2;
note["Dupont Marc"]["Math"]=9.5;
note["Ulpere Francis"]["Anglais"]=15;
note["Dupont Marc"]["Anglais"]=7.9;
note["Lafon Benjamin"]["Informatique"]=15.75;
```

```
for(const auto& notes_courante : note)
{
    const auto& nom=notes_courante.first;
    std::cout<<nom<<" a obtenu "<<std::endl;

    for(const auto& moyenne : notes_courante.second)
        std::cout<<"  "<<moyenne.first<<" en "<<moyenne.second<<std::endl;
}
```

Plus simple en C++11

std::map avec comparateurs spécifiques

Structure pixel/couleur

```
struct pixel
{
    int x,y;
    pixel(int x_arg,int y_arg)
        :x(x_arg),y(y_arg) {}
};

struct color
{
    float r,g,b;
    color():r(0.0f),g(0.0f),b(0.0f){}
    color(float r_arg,float g_arg,float b_arg)
        :r(r_arg),g(g_arg),b(b_arg) {}
};

struct pixel_is_less
{
    bool operator()(const pixel& p0,const pixel& p1)
    {
        return p0.x+pic_dim*p0.y<p1.x+pic_dim*p1.y;
    }
    const static int pic_dim=256;
};
```

Compare l'offset dans l'image

std::map avec comparateurs spécifiques

Structure pixel/couleur

```
struct pixel
{
    int x,y;
    pixel(int x_arg,int y_arg)
        :x(x_arg),y(y_arg) {}
};

struct color
{
    float r,g,b;
    color():r(0.0f),g(0.0f),b(0.0f){}
    color(float r_arg,float g_arg,flo
        :r(r_arg),g(g_arg),b(b_arg) {
};

struct pixel_is_less
{
    bool operator()(const pixel& p0,c
    {
        return p0.x+pic_dim*p0.y<p1.x
    }
    const static int pic_dim=256;
};
```

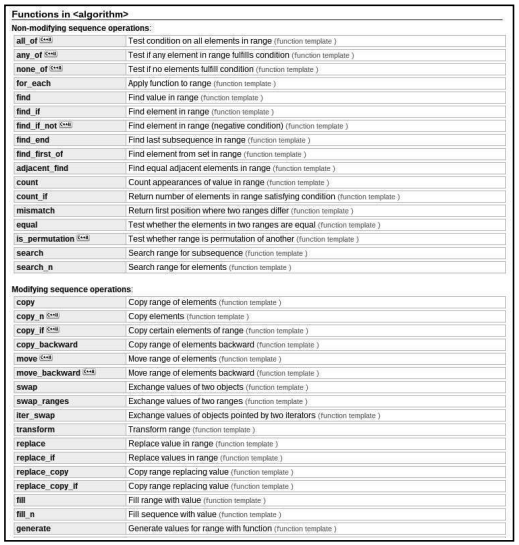
```
int main()
{
    typedef std::map<pixel,color,pixel_is_less> image_type;

    image_type image;
    image[pixel(25,36)]=color(0.4,0.5,0.6);
    image[pixel(32,18)]=color(0.8,0.7,0.1);
    image[pixel(47,11)]=color(0.1,0.7,0.7);

    if(image.find(pixel(12,48))==image.end())
        std::cout<<"pixel [12,48] non colore"<<std::endl;

    image_type::const_iterator it=image.find(pixel(32,18));
    if(it!=image.end())
    {
        const pixel& p=it->first;
        const color& c=it->second;

        std::cout<<p.x<<","<<p.y<<" a pour couleur "
                 <<c.r<<","<<c.g<<","<<c.b<<std::endl;
    }
    return 0;
}
```

STL définit d'autres conteneurs

- std::queue<type> — File (FIFO)
- std::priority_queue<type> — File ordonnée
- std::stack<type> — Pile (LIFO)
- std::unordered_set<type> — Tables de hachages
  c++11
- std::unordered_map<type>
  c++11
- std::array<type,taille> — éléments contigues en mémoire, taille connue à la compilation (remplacant de T[N])
  c++11

...

Les algorithmes



Functions in <algorithm>

Non-modifying sequence operations:

all_of (c++11)	Test condition on all elements in range (function template)
any_of (c++11)	Test if any element in range fulfills condition (function template)
none_of (c++11)	Test if no elements fulfill condition (function template)
for_each	Apply function to range (function template)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_if_not (c++11)	Find element in range (negative condition) (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)
mismatch	Return first position where two ranges differ (function template)
equal	Test whether the elements in two ranges are equal (function template)
is_permutation (c++11)	Test whether range is permutation of another (function template)
search	Search range for subsequence (function template)
search_n	Search range for elements (function template)

Modifying sequence operations:

copy	Copy range of elements (function template)
copy_n (c++11)	Copy elements (function template)
copy_if (c++11)	Copy certain elements of range (function template)
copy_backward	Copy range of elements backward (function template)
move (c++11)	Move range of elements (function template)
move_backward (c++11)	Move range of elements backward (function template)
swap	Exchange values of two objects (function template)
swap_ranges	Exchange values of two ranges (function template)
iter_swap	Exchange values of objects pointed by two iterators (function template)
transform	Transform range (function template)
replace	Replace value in range (function template)
replace_if	Replace values in range (function template)
replace_copy	Copy range replacing value (function template)
replace_copy_if	Copy range replacing value (function template)
fill	Fill range with value (function template)
fill_n	Fill sequence with value (function template)
generate	Generate values for range with function (function template)

Les algorithmes

Functions in <algorithm>

Non-modifying sequence operations:

all_of (c++11)	Test condition on all elements in range (function template)
any_of (c++11)	Test if any element in range fulfills condition (function template)
none_of (c++11)	Test if no elements fulfill condition (function template)
for_each	Apply function to range (function template)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_if_not (c++11)	Find element in range (negative condition) (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)
mismatch	Return first position where two ranges differ (function template)
equal	Test whether the elements in two ranges are equal (function template)
is_permutation (c++11)	Test whether range is permutation of another (function template)
search	Search range for subsequence (function template)
search_n	Search range for elements (function template)

Modifying sequence operations:

copy	Copy range of elements (function template)
copy_n (c++11)	Copy elements (function template)
copy_if (c++11)	Copy certain elements of range (function template)
copy_backward	Copy range of elements backward (function template)
move (c++11)	Move range of elements (function template)
move_backward (c++11)	Move range of elements backward (function template)
swap	Exchange values of two objects (function template)
swap_ranges	Exchange values of two ranges (function template)
iter_swap	Exchange values of objects pointed by two iterators (function template)
transform	Transform range (function template)
replace	Replace value in range (function template)
replace_if	Replace values in range (function template)
replace_copy	Copy range replacing value (function template)
replace_copy_if	Copy range replacing value (function template)
fill	Fill range with value (function template)
fill_n	Fill sequence with value (function template)
generate	Generate values for range with function (function template)

S'appliquent sur les conteneurs de la stl s'appuient sur les iterators

Exemple d'application d'algorithme

```
#include <vector>
#include <algorithm>

int main()
{
    std::vector<float> v;
    v.push_back(14.2);v.push_back(4.8);v.push_back(7.1);
    v.push_back(-0.2);v.push_back(1.1);v.push_back(3.4);

    std::sort(v.begin(),v.end());

    return 0;
}
```

itérateur de début *itérateur de fin*

Entrées/Sorties

Les entrées/sorties sont gérées comme des flux
(clavier,ecran,fichiers, etc)

Un flux est un buffer FIFO

Ex.		
	`cout<<valeur;`	cout: flux de sortie standard
	`cerr<<valeur;`	cerr: flux de sortie d'erreur
	`cin>>valeur;`	cin: flux d'entrée standard

cout est un objet de type std::ostream
cin est un objet de type std::istream

Par convention opérateur **<<** indique une entrée dans le flux
opérateur **>>** indique une sortie du flux

Exemple de flux de std::string

```
#include <sstream>
#include <vector>
#include <algorithm>

int main()
{
    std::stringstream s;
    s<<"J'envoie ";
    s<<"dans le ";
    s<<"flux le nombre 43";

    std::string str[6];
    for(int k=0;k<6;++k)
        s>>str[k];

    float nombre;
    s>>nombre;

    for(int k=0;k<6;++k)
        std::cout<<str[k]<<" ";
    std::cout<<nombre<<std::endl;

    return 0;
}
```

création d'un flux

envoie dans le flux s

extrait du flux
(s'arrête à chaque espace)

extrait du flux

Attention: Un flux ne se copie pas (il n'y a pas de duplication)

Exemple de flux de std::string

```
#include <sstream>
#include <vector>
#include <algorithm>

int main()
{
    std::stringstream s;
    s<<"J'envoie ";
    s<<"dans le ";
    s<<"flux le nombre 43";

    std::string str[6];
    for(int k=0;k<6;++k)
        s>>str[k];

    float nombre;
    s>>nombre;

    for(int k=0;k<6;++k)
        std::cout<<str[k]<<" ";
    std::cout<<nombre<<std::endl;

    return 0;
}
```

création d'un flux

envoie dans le flux s

extrait du flux
(s'arrête à chaque espace)

extrait du flux

Attention: Un flux ne se copie pas (il n'y a pas de duplication)

Lecture de fichiers

```
std::ifstream ifs;
ifs.open("fichier.txt",std::ifstream::in);

if(ifs.good()!=true)
    std::cout<<"fichier.txt n'a pas pu etre ouvert"<<std::endl;
else
{
    while(ifs.good())
    {
        std::string buffer;

        ifs >> buffer;

        if(ifs.good())
            std::cout<<buffer<<std::endl;
    }
}

ifs.close();
```

flux pointant vers un fichier

ouverture du fichier

vérification

lecture jusqu'à EOF

lecture du fichier resultat recu dans le buffer

évite d'afficher eof

fermeture fichier

Lecture de fichiers

```
std::ifstream ifs;
ifs.open("fichier.txt",std::ifstream::in);

if(ifs.good()!=true)
    std::cout<<"fichier.txt n'a pas pu etre ouvert"<<std::endl;
else
{
    while(ifs.good())
    {
        std::string buffer;

        ifs >> buffer;

        if(ifs.good())
            std::cout<<buffer<<std::endl;
    }
}

ifs.close();
```

Je
mange
du
pain.

*fichier.txt ×

```
Je mange du pain.
```


Lecture de fichiers par lignes

```
std::ifstream ifs;
ifs.open("fichier.txt",std::ifstream::in);

if(ifs.good()!=true)
    std::cout<<"fichier.txt n'a pas pu etre ouvert"<<std::endl;
else
{
    while(ifs.good())
    {
        std::string buffer;

        std::getline(ifs,buffer);

        std::cout<<buffer<<std::endl;
    }
}

ifs.close();
```

lit une ligne complète

Ecriture de fichiers

```
std::ofstream ofs;
ofs.open("fichier.txt",std::ofstream::out);

if(ofs.good())
{
    ofs<<"Bonjour a ";
    ofs<<"tous."<<std::endl;
    ofs<<"Je sais que "<<"2+2="<<4;
}
else
    std::cout<<"Cannot open fichier.txt"<<std::endl;

ofs.close();
```

```
Bonjour a tous.
Je sais que 2+2=4
```