Programmation C++
Les templates - la bibliothèque standard

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

3 février 2013

C++ : Fonctions template

- On peut écrire une fonction (ou une méthode) paramétrée par un ou plusieurs types.
- La liste des paramètres est introduite par `template` et placée entre chevrons, chaque paramètre est introduit par `typename` ou par `class`.

```
template <typename T>
T max(T x, T y){
  return (x>y) ? x : y;
}
```

C++ : Fonctions template

- Les valeurs passées en argument doivent avoir exactement le bon type. Si `max` a deux arguments de type template `T`,

```
max(3,4.5);
```

  provoque l'erreur de compilation :

```
error: no matching function for call to
'max(int, double)'
```

- On peut préciser quelle est la valeur du template lors de l'appel de la fonction en l'indiquant entre chevrons :

```
max<float>(3,4.5);
```

  appelle la fonction `max`, instanciée pour des `float` sur les arguments `3` et `4.5`.

C++ : Fonctions template

On peut déclarer plusieurs fonctions template ou non avec le même nom et le même nombre d'arguments, à condition que certaines aient plus de template que d'autres.

C++ : Fonctions template

```
template <typename T, typename U>
void comp_type(const T &x, const U &y){
  cout << "Types differents" << endl;
}

template <typename T>
void comp_type(const T &x, const T &y){
  cout << "Types identiques" << endl;
}

template <typename T>
void comp_type(const int &x, const T &y){
  cout << "Premier entier et l'autre non" << endl;
}

void comp_type(const int &x, const int &y){
  cout << "Deux entiers" << endl;
}
```

C++ : Fonctions template

- Si un type template ne correspond pas au type d'un argument (type de retour ou type utilisé à l'intérieur de la fonction), on doit le préciser lors de l'appel.

```
template <typename T, typename U>
const T& convert(const U& x){
  return reinterpret_cast<const T&>(x);
}
```

- Il faut préciser la valeur du premier paramètre lors de l'appel : `convert<int>(3.5);`
- On peut préciser la valeur de tous ou seulement des premiers paramètres.

C++ : Fonctions template

- En plus des types, les paramètres peuvent être des valeurs *intégrales* constantes : valeur numérique entière (`int`, `long`, `short`, `char` ou `wchar_t`, `signed`, `unsigned` ou non) ou des pointeurs.

```
template <typename T, unsigned int N>
void tri(T (&t)[N]);  //trie le tableau t
```

- On peut aussi spécifier qu'un type est une classe template (définition à suivre) :

```
template <template <typename, int> class Array>
void tri(Array<double,8> &t);//trie le tableau t
```

C++ : Types template

On peut aussi créer des type template. Par exemple, le type `pair<>` de la STL :

```
template <typename T, typename U>
struct pair {
  public:
  typedef T first_type;
  typedef U second_type;

  template <typename T2, typename U2>
  pair(const pair<T2,U2> &);

  pair(const T&, const U&);
  pair(){}

  T first;
  U second;
};
```

C++ : Types template

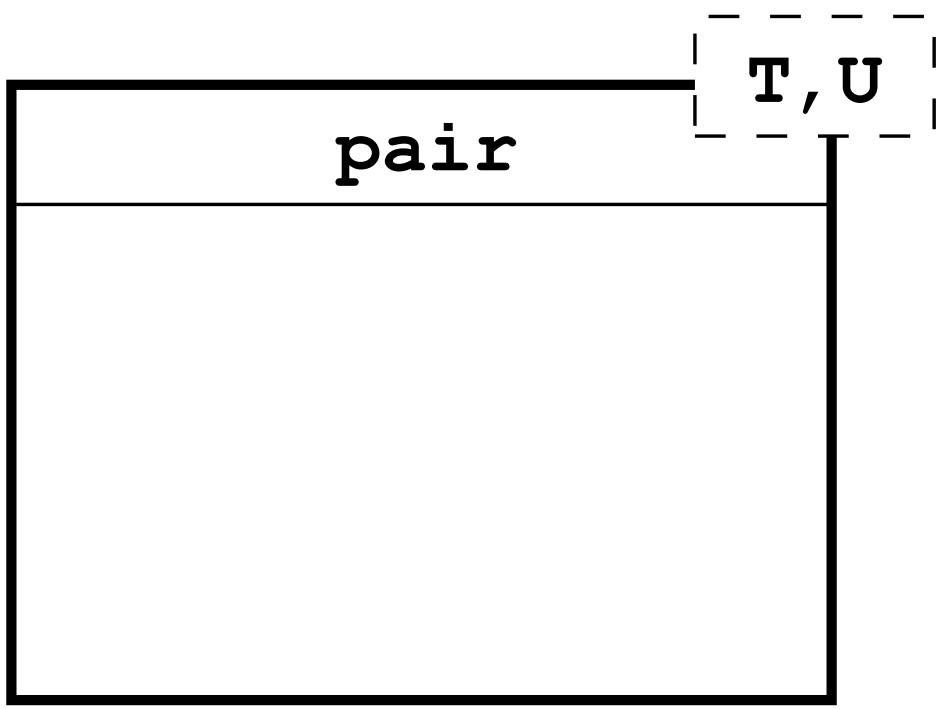
```
template <typename T, typename U>
pair<T,U>::pair(const T& a, const U& b)
  : first(a), second(b) {
  // empty
}

template <typename T, typename U>
template <typename T2, typename U2>
pair<T, U>::pair(const pair<T2, U2>& p)
  : first((T) p.first), second((T) p.second) {
  // empty
}

/*** Pas template <typename T, typename U,
typename T2, typename U2>   */
```

C++ : Types template

En UML, la représentation de classe template n'est pas standardisée, mais usuellement, on les représente par :



C++ : Types template

- Contrairement aux fonctions qui peuvent être surchargées, on ne peut pas déclarer deux types avec les mêmes noms, un template et un qui ne le serait pas.
- Un type template a un nombre de paramètres fixe déclaré lors de sa création

  (certains des paramètres peuvent avoir des valeurs par défaut).

C++ : Types template

- À l'intérieur d'une classe template ou du corps de ses méthodes, on peut utiliser le type template sans préciser ses paramètres.
- Dans un type template, on peut déclarer des méthodes ou constructeurs templatés par d'autres paramètres que ceux du type courant.

C++ : Types template

- Lorsque l'on déclare une variable d'un type template, on doit préciser la valeur des paramètres.

```
std::pair<int,double> x(3,5.2);
```

- Seules les méthodes qui sont effectivement appelées sont instanciées dans une classe template.
- Une méthode peut être définie dans une classe template sans être compilable pour tous les types avec lesquels on utilise cette classe ; si elle n'est pas appelée, il n'y a pas de problème.

C++ : Types template

On peut spécialiser un type template, par exemple, le type `pair<>` :

```
template <typename T>
struct pair<T,T>{
  public:
  typedef T first_type;
  typedef T second_type;

  template <typename T2, typename U2>
  pair(const pair<T2,U2> &);

  pair(const T&, const T&);
  pair();

  T first;
  T second;
};
```

C++ : Types template

Attention, même si on précise la valeur de tous ses paramètres, un type template reste un type template et sa spécialisation doit être précédée de `template <>` :

```
template <>
struct pair<int,int>{
  public:
  typedef int first_type;
  typedef int second_type;

  template <typename T2, typename U2>
  pair(const pair<T2,U2> &);
  pair(const int&, const int&);
  pair();

  int first;
  int second;
};
```

C++ : Types template

Un type template peut être paramétré par des valeurs *intégrales* ou des pointeurs constants.

```
template <typename T, int N>
struct Array{
  private:
  T t[N];

  public:
  Array();
  Array(const Array&);
  Array(const T s[N]);
  static const int length=N;

  T& operator[](int);
};
```

C++ : Types template

- On peut préciser des valeurs par défaut pour certains paramètres.

```
template <int N, typename T = int>
struct Array{
  private:
  T t[N];

  public:
  Array();
  Array(const Array&);
  Array(const T s[N]);
  static const int length=N;

  T& operator[](int);
};
```

- Ceci n'est pas possible pour les fonctions template.
- On peut aussi spécifier des paramètres de type "classe template".

C++ : Accès à un type induit par un template

Pour le compilateur, un identifiant représente à priori une variable ou un champ, sauf s'il est capable de vérifier qu'il s'agit d'un type.

- Si l'identifiant est le nom d'une classe, pas de problème ;
- Si l'identifiant est défini à l'intérieur d'une classe connue, pas de problème : `pair<double,int>::first_type` ;
- Si l'identifiant est défini à l'intérieur d'une classe dont les templates ne sont pas instanciés,

  ```
  pair<T,U>::first_type
  ```

  représente a priori un champ. Il faut alors utiliser `typename`

C++ : Accès à un type induit par un template

- Dans le cas d'un type paramètre, cette vérification ne peut pas avoir lieu lors de l'analyse du code ; il faut préciser que l'on fait référence à un type qui sera défini pour chaque instance du code que l'on écrit.

```
template <typename Pair>
void display_first(const Pair &p){
  typename Pair::first_type &x = p.first;
  std::cout << x << std::endl;
}
```

- Pour tout type pour lequel cette fonction est instanciée, il doit exister un type `first_type` induit et un attribut `first` qui retourne ce type, tous deux accessibles.

C++ : Borner les template

- En C++, contrairement à Java, on ne peut pas borner les template. On peut toutefois faire en sorte de bien les choisir !

```
template <typename Pair>
void display_first(const Pair &p){
  typename Pair::first_type &x = p.first;
  std::cout << x << std::endl;
}
```

- Si on veut réserver cette fonction à des paires, autant écrire

```
template <typename T, template U>
void display_first(const pair<T,U> &p){
  T &x=p.first;
  std::cout << x << std::endl;
}
```

C++ : STL - Programmation générique : notions

- Concept
- Trait
- Foncteur
- Allocateur
- Itérateur

C++ : STL - Programmation générique : Les concepts

- Un concept est un ensemble de services que doit offrir un type.
- En programmation objet classique, un concept se traduit généralement par une interface.
- En programmation générique, pour ne pas perdre de temps à l'exécution, on refuse le typage dynamique et on n'utilise pas le mécanisme de l'interface.
- Un concept est donc uniquement défini par l'usage qui en est fait !

C++ : STL - Programmation générique : Les concepts

- Par exemple, on verra que la bibliothèque standard offre différents concepts : `Container`, `Iterator`.
- Le seul point commun de deux classes qui implémentent le même concept est qu'elles offrent des méthodes similaires.
- On prendra soin lorsque l'on définit des fonctions ou types template à nommer les paramètres de façon à montrer à quel concept ils doivent correspondre.
- On verra comment faire vérifier un concept par le compilateur, ce que la STL ne fait pas.

C++ : STL - Programmation générique : Les traits

- Un trait est une classe qui contient des définitions de type.
- Il permet de fournir des types qui s'abstraient de l'implémentation.
- Il permet aussi de faire des calculs sur les types.

```
template <class Pair>
struct swap_pair
{
  typedef std::pair<typename Pair::first_type,
  typename Pair::second_type> type;
};
```

C++ : STL - Programmation générique : Les traits

Un exemple plus concret consiste à trouver l'élément maximum parmi un ensemble d'élements de même type.

```
#include <limits.h>

int find_max(vector<int> &values) {
  int maxi = INT_MIN;
  vector<int>::iter = values.begin();
  while (iter != values.end()) {
    if (*iter > maxi) maxi = *iter;
    ++iter;
  }
  return maxi;
}
```

C++ : STL - Programmation générique : Les traits

- Comment modifier ce code afin de pouvoir l'appliquer à d'autres types comme `float` et `double` ?
- La manière dont a été écrit l'algorithme est spécifique au type entier (`int maxi = INT_MIN;`).

C++ : STL - Programmation générique : Les traits

Prendre le premier élément du vecteur comme valeur maximale initiale :

```
template<class>
T find_max(vector<T>& values) {
  vector<T>::iter = values.begin();
  T maxi = *iter;
  ++iter;
  while (iter != values.end()) {
    if (*iter > maxi) maxi = *iter;
    ++iter;
  }
  return maxi;
}
```

Ne fonctionne pas si la liste est vide.

C++ : STL - Programmation générique : Les traits

On utilise une classe de **Traits** (ou classe de caractéristiques ou de propriétés) `MinValue` qui va nous permettre de définir une valeur constante `MINI` qui sera redéfinie (spécialisée) pour chaque type voulu.

```
template<typename T>
class MinValue {
  public:
  static const T    MINI = 0;
};
```

C++ : STL - Programmation générique : Les traits

On spécialise la valeur `MINI` pour les `int` et les `float` :

```
// for integers
template<>
class MinValue<int> {
  public:
  // -100 is just an example
  static const int   MINI = -100;
};
// for floats
template<>
class MinValue<float> {
  public:
  // -333.33 is just an example
  static const float   MINI = -333.33;
};
```

C++ : STL - Programmation générique : Les traits

L'algorithme générique `find_max` est alors le suivant :

```
// find_max with Traits
template<class T>
T find_max(vector<T>& v) {
  T maxi = MinValue<T>::MINI; // use of Traits
  typename vector<T>::iterator iter = v.begin();
  while (iter!=v.end()) {
    if (*iter > maxi) maxi = *iter;
    ++iter;
  }
  return maxi;
}
```

C++ : STL - Programmation générique : Les traits

Le programme de test :

```
vector<int> v;
v.push_back(2);
v.push_back(5);
v.push_back(25);
v.push_back(3);
cout << "find_max " << find_max(v) << endl;
v.insert(vInt.begin(), 333);
cout << "find_max " << find_max(v) << endl;
v.clear();
cout << "find_max " << find_max(v) << endl;
```

affiche

```
find_max 25
find_max 333
find_max -100
```

C++ : STL - Programmation générique : Les foncteurs

- Un foncteur est une classe qui contient la redéfinition de l'opérateur de prise d'argument `()`.
- Il permet de paramétrer le comportement de certaines classes.

C++ : STL - Programmation générique : Les foncteurs

```
#include <iostream>

class EcrireDansFlux
{
  std::ostream & flux_;

  public:
  ecrireDansFlux(std::ostream & flux)
  : flux_(flux)
  { }

  template <class T>
  void operator() (const T &val)
  {
    flux_ << val;
  }
};
```

C++ : STL - Programmation générique : Les allocateurs

- Un allocateur est une classe qui fournit un certain nombre de méthodes permettant d'allouer de la mémoire, de la libérer et de manipuler des pointeurs sur la mémoire allouée.
- C'est une Fabrique.
- Il est utilisé par les conteneurs de la STL et permet, si on le désire, en le redéfinissant d'avoir un accès total à la gestion de la mémoire pour les objets stockés.
- Il s'agit d'un exemple d'application du pattern Stratégie.

C++ : STL - Programmation générique : Les allocateurs

Structure d'un allocateur standard :

```
template <class T>
class allocator
{
  public:
  typedef size_t    size_type;
  typedef ptrdiff_t difference_type;
  typedef T         *pointer;
  typedef const T   *const_pointer;
  typedef T         &reference;
  typedef const T   &const_reference;
  typedef T         value_type;
  template <class U> struct rebind {
    typedef allocator<U> other;
  };
}
```

C++ : STL - Programmation générique : Les allocateurs

```
{
  ...
  allocator() throw();
  allocator(const allocator &) throw();
  template <class U> allocator(const allocator<U> &) th
  ~allocator() throw();

  pointer address(reference objet);
  const_pointer address(const_reference objet) const;
  pointer allocate(size_type nombre,
  typename allocator<void>::const_pointer indice);
  void deallocate(pointer adresse, size_type nombre);
  size_type max_size() const throw();
  void construct(pointer adresse, const T &valeur);
  void destroy(pointer adresse);
};
```

Il existe une spécialisation pour `void` qui ne parle pas de référence.

C++ : STL - Programmation générique : Les itérateurs

- Un itérateur est une classe qui est une abstraction de la notion de pointeur ; il offre des services du même type : déréférencement, incrémentation,...

```
const value_type& operator*() const;
const value_type* operator->() const;
It& operator++();
It operator++(int);
bool operator==(const It&) const;
bool operator!=(const It&) const;
```

- La STL contient différentes catégories d'itérateurs qui supportent plus ou moins d'opérations : décrémentation, arithmétique,...
- Généralement, chaque conteneur STL fournit un type `iterator` et un type `const_iterator` (déréférencement uniquement en lecture).

C++ : STL - Les itérateurs

Les différentes catégories d'itérateurs :

- Output : Accès uniquement en écriture,
  une seule fois par valeur ;
- Input : Accès uniquement en lecture,
  pas de garantie sur l'ordre de parcours ;
- Forward : Output+Input, ordre de parcours toujours identique ;
- Bidirectionnal : Forward, décrémentation possible ;
- Random Access : Bidirectionnal,
  possibilité d'accès par index (`[]`).

C++ : STL - Les itérateurs

Cinq types sont définis comme tags pour ces catégories :

```
struct output_iterator_tag{};
struct input_iterator_tag{};
struct forward_iterator_tag{};
struct bidirectionnal_iterator_tag{};
struct random_access_iterator_tag{};
```

C++ : STL - Les itérateurs

- Deux itérateurs i_1 et i_2 (Forward ou plus) sur la même structure définissent un *intervalle* d'itérateurs $[i_1; i_2[$.
- Les conteneurs de la STL fournissent des méthodes `begin()` et `end()` qui désignent l'intervalle des valeurs stockées dans le conteneur.
- Une boucle typique sur un conteneur :

```
C container;
...
for(C::iterator& it= container.begin();
    it != container.end();
    ++it )
{
  ...
}
```

C++ : STL - Les itérateurs

Transformer tous les caractères d'une `std::string` en majuscules.

```
#include <string>
#include <locale>

std::string en_majuscules(const std::string &s)
{
  std::string majuscules = s;
  for (std::string::iterator it = majuscules.begin();
       it != majuscules.end();
       ++it)
  {
    *it = std::toupper(*it,std::locale(""));
  }
  return majuscules;
}
```

C++ : STL - Les itérateurs

Transformer tous les caractères d'une `std::string` en majuscules.

```
#include <string>
#include <locale>
#include <algorithm>

void _majuscules(char & c)
{
  c = std::toupper(c, std::locale(""));
}

std::string en_majuscules(const std::string &s)
{
  std::string majuscules = s;
  std::for_each(majuscules.begin(),
                majuscules.end(),
                _majuscules);
  return majuscules;
}
```

C++ : STL - Classes utiles

- `string` et `basic_string`
- flots d'entrées/sorties
- conteneurs
- pointeurs automatiques
- paires, `valarray`, tableaux de bits, complexe, ...

C++ : STL - basic_string

- Le type `string` de la STL est en réalité une instance du type template `basic_string` :

```
typedef basic_string<char>    string;
typedef basic_string<wchar_t> wstring;
```

- En réalité un `basic_string` a 2 autres paramètres template : un trait pour gérer plus finement les types induits et un allocateur.

C++ : STL - `basic_string`

Contenu public de la classe (+ de 200 méthodes) :

- définitions des types obtenus du trait et de l'allocateur ;
- constructeurs : copie (partielle), à partir d'un `const char*`, en donnant un intervalle d'(input) itérateurs ;
  on peut passer un allocateur (facultatif) ;
- itérateurs :
  types : `iterator`, `const_iterator`,
  `reverse_iterator`, `const_reverse_iterator` ;
  méthodes : `begin()`, `end()`, `rbegin()`, `rend()` ;

C++ : STL - `basic_string`

- accesseurs : `size()`≡`length()`, `empty()` (est vide ?), `capacity()` (réservé), `max_size()` (réservable), `get_allocator()` ;
- manipulateurs : `resize(size, c = ..)`, redimensionne, `reserve(size)` réalloue de la mémoire ;
- facteur : `substr(deb = 0, lng = npos)`
- conversion en chaîne : `data()` crée une chaîne de caractère, contrairement à `c_str()` ;
  copie dans une chaîne : `copy(dest, taille, deb=0)` ;

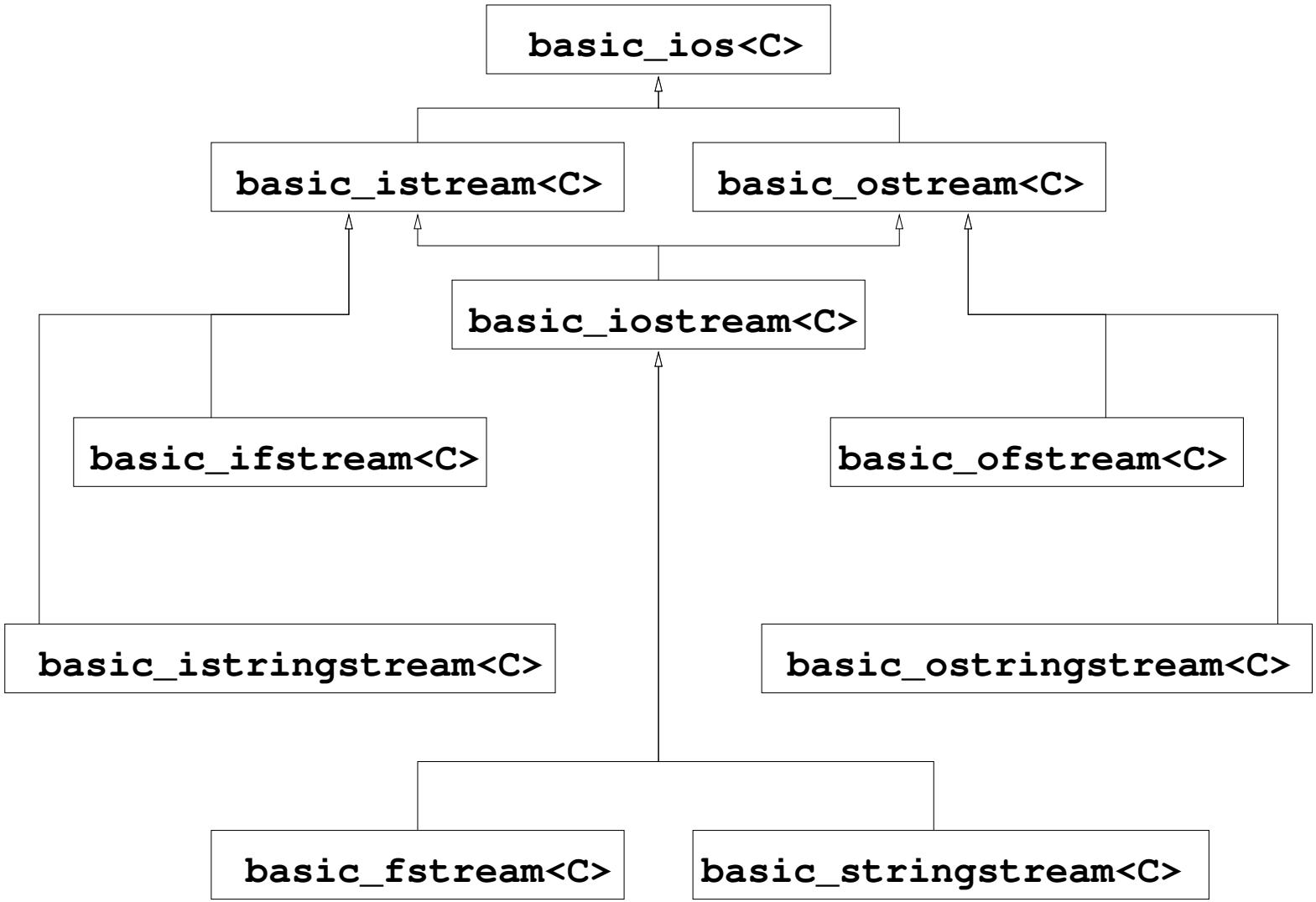
C++ : STL - `basic_string`

- accès : `[]`, ou `at(size_type)` ;
- affectation : `=` ou `assign(..)` ;
- concaténation : `+=` ou `append(..)` ;
- insertion : `insert(pos,chaine)` ou
  `insert(it, input_it_deb, input_it_fin);`
- suppression : `clear()`, `erase(deb, lng = npos)`,
  `erase(it);` ou `erase(it_deb, it_fin);`

C++ : STL - `basic_string`

- remplacement : `replace(..)`, en spécifiant
  soit le début et la longueur soit deux itérateurs,
  puis soit une chaîne soit deux (input) itérateurs ;
  - `swap(string)`
- comparaison : `compare(..)`
- recherche : `find(mot,p=0)`, `rfind(mot,p=npos)`,
  `find_{first,last}_[not_]of(ch,p=0)`

C++ : STL - Les flots



C++ : STL - Les flots

Chacun de ces types est instancié pour les `char` et pour les `wchar_t` : `istream`, `wistream`, `ofstream`, `wofstream`, etc.

C++ : STL - Les flots

- On peut créer des flots entrant (`ifstream`), sortant (`ofstream`) ou bidirectionnels (`fstream`) sur des fichiers.
- Les modes d'ouverture des flots sont des champs statiques de la classe `ios_base` de type `ios_base::openmode`.

`in`	ouverture en lecture
`out`	ouverture en écriture
`app`	écriture en ajout de donnée
`ate`	écriture avec position initiale en fin
`trunc`	écriture tronquante
`binary`	ouverture fichier binaire

C++ : STL - Les flots

Les constructeurs de flots prennent de zéro à deux arguments : une référence sur un fichier et un mode d'ouverture obtenu par composition booléenne des modes de base.

```
ofstream f("newFile", ios_base::out|ios_base::trunc);
```

C++ : STL - Les flots

Méthodes propres aux flots sur fichiers :

- `is_open()` teste si le flot est ouvert ;
- `open(ref,mod=out|trunc)` ouvre le flot ;
- `close()` ferme le flot ;

C++ : STL - Les flots

- On peut créer des flots à partir de `string` ; ceci facilite notamment l'analyse des chaînes de caractères : `istringstream`, `ostringstream`, `stringstream`
- Les constructeurs de ces classes prennent de zéro à deux arguments : soit un mode d'ouverture, soit un `string` et éventuellement un mode d'ouverture.

C++ : STL - Les flots

Méthodes propres aux flots sur `string` :

- `rdbuf()` retourne un pointeur sur le `string` buffer du flot ;
- `str()` retourne un `string` avec le contenu du flot ;
- `str(strg)` fixe le contenu du flot.

C++ : STL - Les flots

Les flots de sorties héritent de la classe `ostream` qui contient les définitions suivantes :

- types issus de `char_traits<>` ;
- surcharge de l'opérateur `<<` pour tous les types de base, les `char*` et les autres pointeurs ;
- `put(car)` et `write(ch,lng)` : écritures non formatées
- `flush()` vide le flot

C++ : STL - Les flots

- `tellp()` donne la position dans le flot ;
  `seekp(pos,modd)` modifie la position dans le flot selon la valeur du second argument de type `ios_base::seekdir`
  – soit de manière absolue : `ios_base::beg`
  – soit par rapport à la fin : `ios_base::end`
  – soit par rapport à la position courante : `ios_base::cur`
- une classe `sentry` dont on peut construire un objet à partir du flot et qui vérifie si celui-ci est apte à recevoir des données :

```
ostream o=...;
//...
ostream::sentry s(o);
if (s){ //on sait qu'on peut ecrire
  //...
}
```

C++ : STL - Les flots

L'opérateur `<<` est aussi surchargé pour accepter en second paramètre des fonctions de type `ostream& (*)(ostream&)` qui lorsqu'elles sont appliquées de la sorte modifient le flot courant. On les appelle des manipulateurs.

```
ostream& operator<< ( ostream& o,
                      ostream& (*manip)(ostream&)) {
  return manip(o);
}
```

C++ : STL - Les flots

Les manipulateurs prédéfinis sont :

`endl`	Retour à la ligne + vide le flot
`ends`	Fin de ligne
`flush`	Vide le flot
`left` / `right`	alignement
`unitbuf` / `nounitbuf`	vide ou non le flot à chaque écriture

C++ : STL - Les flots

`boolalpha` / `noboolalpha`	forme des booléens (true ou 1)
`hex` / `oct` / `dec`	base des entiers
`showbase` / `noshowbase`	indique ou non la base courante
`fixed` / `scientific`	forme des décimaux
`showpoint` /`noshowpoint`	si les décimaux sont entiers
`showpos` / `noshowpos`	+ pour les nombres positifs
`uppercase` / `nouppercase`	pour les chiffres hexa

On peut définir facilement soi-même des manipulateurs sans argument.

C++ : STL - Les flots

- Il existe aussi des manipulateurs avec argument, comme par exemple `setbase(int base)` ou `setprecision(int)`.
- Les manipulateurs agissent en fait sur des flags définis dans la classe `ios_base`.
- Exo : Sauriez-vous définir des manipulateurs avec arguments ?

C++ : STL - Les flots

La classe `ostream_iterator<T, C>` permet d'obtenir un output itérateur qui encapsule un flot `basic_ostream<C>` pour écrire des valeurs de type `T`.

C++ : STL - Les flots

Les flots d'entrées héritent de la classe `istream` qui contient les définitions suivantes :

- types issus de `char_traits<>` ;
- surcharge de l'opérateur `>>` pour tous les types de base et `char*` ;
- lectures non formatées : `get()` et `get(c&)` récupèrent un octet dans le flot, `peek()` regarde le premier octet, `putback(c)` et `unget()` remettent un octet dans le flot,

C++ : STL - Les flots

- `read(ch,n)` lit jusqu'a *n* octets, ainsi que `readsome(ch,n)` qui s'arrête si le buffer est vide ; `get(ch,n,delim)` lit dans le flot, éventuellement jusqu'à un délimiteur, `getline(ch,n,delim)` lit la ligne suivante ; `ignore(n, delim)` enlève *n* octets du flot.
- `gcount()` retourne le nombre de caractères lus lors de `getline`.

C++ : STL - Les flots

- `sync()` synchronise le buffer avec le média ;
- `tellg()` donne la position dans le flot ;
  `seekg(pos,modd)` modifie la position dans le flot
  selon la valeur du second argument de type
  `ios_base::seekdir`
  – soit de manière absolue : `ios_base::beg`
  – soit par rapport à la fin : `ios_base::end`
  – soit par rapport à la position courante : `ios_base::cur`
- une classe `sentry` dont on peut construire un objet à partir du flot et qui vérifie si celui-ci est apte à envoyer des données.

C++ : STL - Les flots

- L'opérateur `>>` est aussi surchargé pour accepter en second paramètre des manipulateurs de type `istream& (*)(istream&)`.
- Les manipulateurs prédéfinis sont :

`ws`	Efface les espaces à venir
`skipws` / `noskipws`	ignore les espaces
`boolalpha` / `noboolalpha`	forme des booléens (true ou 1)
`hex` / `oct` / `dec`	base des entiers

- On peut définir facilement soi-même des manipulateurs sans argument.

C++ : STL - Les flots

La classe `istream_iterator<T, C>` permet d'obtenir un input itérateur qui encapsule un flot `basic_istream<C>` pour lire des valeurs de type `T`.

C++ : STL - Les flots

La classe `ios` contient les définitions des fonctionalités communes à tous les flots, notamment la gestion des erreurs.

`good()`	vrai si Etat normal
`eof()`	vrai si Fin de lecture ou d'écriture
`fail()`	vrai si Erreur "logique"
`bad()`	vrai si Erreur fatale (matérielle,...)

C++ : STL - Les flots

Par ailleurs, le transtypage en `void*` et la redéfinition de `!` permettent d'écrire

```
// f flot
if (f){   //<=> if(f.good())
    //...
}
```

C++ : STL - Les foncteurs

- La bibliothèques standard définit dans le fichier `functional` un certain nombre de foncteurs : arithmétiques (`plus<T>`), logiques(`less<T>`), projections, encapsulation de pointeurs de fonction, ...
- Les foncteurs dont l'opérateur `()` (unaire ou binaire) retourne un booléen sont appelés prédicats.

C++ : STL - Les conteneurs

Il existe deux catégories principales de conteneurs STL :

- Les conteneurs séquentiels
- Les conteneurs associatifs

C++ : STL - Les conteneurs

Définitions communes à tous les conteneurs :

- Itérateurs : types `iterator`, `const_iterator`, `difference_type` ;
  – les itérateurs des conteneurs sont au moins forward ;
  – méthodes : `begin()`, `end()`
  – les conteneurs *réversibles* fournissent des itérateurs bi-directionnels de type `reverse_iterator`, `const_reverse_iterator`
  ainsi que les méthodes `rbegin()` et `rend()`

C++ : STL - Les conteneurs

- Éléments : types `size_type`, `value_type`, `reference`, `const_reference`
- Comparaison : opérateurs == et != ; les opérateurs < et > sont définis s'ils le sont sur les éléments (ordre lexicographique)
- Méthodes : `size()`, `empty()`, `max_size()`, `swap(..)`
- Le dernier paramètre template des conteneurs STL est un allocateur dont le type par défaut est `std::allocator<value_type>`.

C++ : STL - Les conteneurs

Définitions communes à tous les itérateurs séquentiels (ou séquences) :

- Constructeurs : sans argument, avec la taille, taille + valeur initiale, ou intervalle d'itérateurs ;
- `void assign(..)` : taille + valeur initiale, ou intervalle d'itérateurs, réinitialise la séquence ;

C++ : STL - Les conteneurs

- `void insert(it,..)` : insère juste avant `it`, soit une valeur, soit *n* fois la même valeur, soit un ensemble donné par intervalle d'itérateurs ;
- `iterator erase(..)` : efface le ou les éléments indiqué(s) par un itérateur ou un intervalle, retourne un itérateur sur la position suivante ;
- `void clear()` : vide la séquence.

C++ : STL - Les conteneurs

Les trois types de séquences de la STL sont :

- `list<T>` : listes doublement chaînées
- `vector<T>` : ”tableaux” redimensionnables
- `deque<T>` : tableau circulaire...

Il existe trois adaptateurs qui peuvent être implémentés grâce à ces types :

- `stack<>` : piles
- `queue<>` : files
- `priority_queue<>` : files de priorités

C++ : STL - Les conteneurs

Propriétés des listes :

- Itérateur bi-directionnel, pas d'accès aléatoire
- Insertion et suppression en temps constant, conservent la validité des itérateurs et références
- Insertion, suppression et lecture en début/fin de list :

```
void push_{front,back}(const T& o),
void pop_{front,back}(),
reference front(),reference back();
```

C++ : STL - Les conteneurs

- `void remove(const T& o)` Supprime toutes les occurences de `o` dans la liste (linéaire)
- `void remove_if(UPredicate<T>& p)` Supprime toutes les occurences vérifiant `p` dans la liste (linéaire)
- `void unique(BPredicate<T>& p)` Supprime toutes les occurences à une position `it` tq `p(*(it-1),*it)` est vrai (linéaire)

C++ : STL - Les conteneurs

- `void splice(it,l2,[deb,[fin]])` Enlève les éléments de `l2` et les place dans la liste courante ($O(|fin - deb|)$)
- `void sort([BPredicate<T>& order])` Trie la liste ($O(n \log n)$), respecte l'emplacement des éléments équivalents
- `void merge(l2, [BPredicate<T>& order])` Fusionne deux listes triées (linéaire)
- `void reverse()` retourne la liste (linéaire)

C++ : STL - Les conteneurs

Propriétés des vecteurs :

- Itérateur à accès direct
- Insertion et suppression en temps linéaire, sauf à la fin, temps constant amorti
- L'insertion invalide itérateurs et références (réallocation possible)
- La suppression invalide itérateurs et références qui suivent la position effacée

C++ : STL - Les conteneurs

- Insertion, suppression et lecture en fin de vecteur :
  `void push_back(const T& o)`,
  `void pop_back()`,
  `reference back()`, lecture en début : `reference front()` ;
- Accès direct : `at(i)` ou `[]` peut lever `out_of_range`
- Spécialisée pour les booléens : 1 bit / valeur + méthode `flip(i)`

C++ : STL - Les conteneurs

Propriétés des deques (double-ended queues) :

- Itérateur à accès direct
- Insertion et suppression en temps linéaire, sauf au début ou à la fin, temps constant
- L'insertion et la suppression aux extrémités conservent les références ;
- La suppression aux extrémités conserve les itérateurs, mais pas l'insertion ;
- L'insertion et la suppression ailleurs invalident tout ;

C++ : STL - Les conteneurs

- Insertion, suppression et lecture en début/fin de deque :
  `void push_{front,back}(const T& o),`
  `void pop_{front,back}(),`
  `reference front(),reference back();`
- Accès direct : `at(i)` ou `[]` peut lever `out_of_range`

C++ : STL - Les conteneurs

Les piles `stack<T,Container = deque<T> >` offrent l'interface :

- `void push(x)` à partir de `push_back(x)`
- `void pop()` à partir de `pop_back()`
- `T& top()` à partir de `back()`
- `size()`, `empty()`, `==`, `!=`, `<`, `>=`,...

On peut utiliser des listes ou des vecteurs à la place des deques

C++ : STL - Les conteneurs

Les files `queue<T,Container = deque<T> >` offrent l'interface :

- `void push(x)` à partir de `push_back(x)`
- `void pop()` à partir de `pop_front()`
- `T& back()` à partir de `back()`
- `T& front()` à partir de `front()`
- `size()`, `empty()`, `==`, `!=`, `<`, `>=`,...

On peut utiliser des listes à la place des deques

C++ : STL - Les conteneurs

Les files de priorité `priority_queue<T,Container = vector<T>, Order = less<T> >` (définies dans le fichier `queue`) offrent l'interface :

- `void push(x)` ($O(n \log n)$)
- `void pop()` ($O(n \log n)$)
- `const T& top()`
- `size(),empty(),==,!=,<,>=`,...

On peut utiliser des deques à la place des vecteurs

C++ : STL - Les conteneurs

- Un conteneur associatif contient des éléments accessibles par une *clé*.
- Il y a quatre types de conteneurs associatifs

	clé unique	clés multiples
ensembles	`set<K>`	`multiset<K>`
association	`map<K,T>`	`multimap<K,T>`

- Si le type des clés ne surcharge pas l'opérateur `<` , on doit spécifier un type prédicat binaire de comparaison admettant ce type.
- Pour les associations, `value_type` vaut `pair<K,T>`.

C++ : STL - Les conteneurs

Définitions communes à tous les conteneurs associatifs :

- Itérateurs : Le parcours d'un conteneur associatif est tel que l'élément suivant n'est pas stritement plus petit que l'élément courant ;
- Types : `key_type`, `value_type`, foncteurs `key_compare` et `value_compare`.
- Constructeurs : défaut (2 argument optionnels : comparateur et allocateur), ou intervalle d'itérateurs

C++ : STL - Les conteneurs

- Insertion : `void insert(it, vt)` insère `vt` de type `value_type`, `it` indiquant l'emplacement probable de l'insertion ;
  - `void insert(deb, fin)`
  -
  clé multiple, `iterator insert(vt)`
  clé unique, `pair<iterator, bool> insert(vt)`
- Suppression : `void erase(it)`, `void erase(deb, end)`, `size_t erase(vt)`

C++ : STL - Les conteneurs

- `iterator find(key)` renvoie un itérateur sur un élément correspondant à la clé,
  – `iterator lower_bound(key)` renvoie un itérateur sur le premier élément correspondant à la clé,
  – `iterator upper_bound(key)` renvoie un itérateur après le dernier élément correspondant à la clé,
  – `pair<iterator, iterator> equal_range(key)` retourne l'intervalle correspondant à la clé
- `count(key)` nombre d'éléments correspondant à la clé

C++ : STL - Les conteneurs

- Les conteneurs associatifs sont généralement implémentés par des arbres rouge et noir.
- La STL garantit que les opérations d'insertion, de suppression et de recherche se font en $O(\log n)$.
- Il n'y a pas dans la STL d'implantation de conteneurs associatifs par table de hachage.
- D'autres bibliothèques pallient cette carence (Boost, par exemple).

C++ : STL - Les algorithmes

- La bibliothèque standard présente de très nombreux algorithmes notamment sur les conteneurs. Il s'agit de fonctions définies pour la plupart dans l'en-tête `algorithm`.
- Ces algorithmes permettent de remplir, copier, trier ou faire des reherches dans des conteneurs. Ils fournissent aussi des outils pour manipuler des conteneurs structurés : tas ou conteneurs triés : insertion, suppression, intersection, etc.

C++ : STL - Les algorithmes
Opérations d'initialisation et de remplissage

```
template <class ForwarIterator, class T>
void fill(ForwardIterator premier,
          ForwardIterator dernier,
          const T &valeur);
template <class OutputIterator, class T>
void fill_n(OutputIterator premier, Size nombre,
            const T &valeur);
template <class ForwardIterator, class T,
          class Generator>
void generate(ForwardIterator premier,
              ForwardIterator dernier,
              Generator g);
template <class OutputIterator, class T,
          class Generator>
void generate_n(OutputIterator premier, Size nombre,
                Generator g);
```

C++ : STL - Les algorithmes
Opérations de copie

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator premier,
                    InputIterator dernier,
                    OutputIterator destination);

template <class BidirectionalIterator1,
                  class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(
              BidirectionalIterator1 premier,
              BidirectionalIterator1 dernier,
              BidirectionalIterator2 fin_destination);
```

C++ : STL - Les algorithmes
Opérations d'échanges d'éléments

```
template <class ForwardIterator,
          class ForwardIterator2>
ForwardIterator2 swap_ranges(
                    ForwardIterator premier,
                    ForwardIterator dernier,
                    ForwardIterator2 destination);
```

C++ : STL - Les algorithmes
Opérations de suppression d'éléments

```
template <class ForwardIterator,
          class T>
ForwardIterator remove(ForwardIterator premier,
                       ForwardIterator second,
                       const T &valeur);

template <class InputIterator,
          class OutputIterator,
          class T>
OutputIterator remove_copy(InputIterator premier,
                           InputIterator second,
                           OutputIterator destination,
                           const T &valeur);
```

C++ : STL - Les algorithmes
Opérations de suppression d'éléments

```
template <class ForwardIterator,
          class Predicate>
ForwardIterator remove_if(ForwardIterator premier,
                          ForwardIterator second,
                          Predicate p);

template <class InputIterator,
          class OutputIterator,
          class Predicate>
OutputIterator remove_copy_if(InputIterator premier,
                              InputIterator second,
                              OutputIterator destinatio
                              Predicate p);
```

C++ : STL - Les algorithmes
Opérations de remplacement

```
template <class ForwardIterator,
          class T>
void replace(ForwardIterator premier,
             ForwardIterator dernier,
             const T &ancienne_valeur,
             const T &nouvelle_valeur);

template <class InputIterator,
          class OutputIterator,
          class T>
void replace_copy(InputIterator premier,
                  InputIterator dernier,
                  OutputIterator destination,
                  const T &ancienne_valeur,
                  const T &nouvelle_valeur);
```

C++ : STL - Les algorithmes
Opérations de remplacement

```
template <class ForwardIterator,
          class Predicate,
          class T>
void replace_if(ForwardIterator premier,
                ForwardIterator dernier,
                Predicate p,
                const T &nouvelle_valeur);

template <class InputIterator,
          class OutputIterator,
          class Predicate,
          class T>
void replace_copy_if(InputIterator premier,
                     InputIterator dernier,
                     OutputIterator destination,
                     Predicate p,
                     const T &nouvelle_valeur);
```

C++ : STL - Les algorithmes
Opérations de rotation et de permutation

```
template <class ForwardIterator>
void rotate(ForwardIterator premier,
            ForwardIterator pivot,
            ForwardIterator dernier);

template <class ForwardIterator,
          class OutputIterator>
void rotate_copy(ForwardIterator premier,
                 ForwardIterator pivot,
                 ForwardIterator dernier,
                 OutputIterator destination);
```

C++ : STL - Les algorithmes
Opérations de rotation et de permutation

```
template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator premier,
                      BidirectionalIterator dernier);

template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator premier,
                      BidirectionalIterator dernier);
```

C++ : STL - Les algorithmes
Opérations d'inversion

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator premier,
             BidirectionalIterator dernier);

template <class BidirectionalIterator,
          class OutputIterator>
OutputIterator reverse_copy(
                        BidirectionalIterator premier,
                        BidirectionalIterator dernier,
                        OutputIterator destination);
```

C++ : STL - Les algorithmes
Opérations de mélange

```
template <class RandomAccessIterator>
void random_shuffle(RandomAccessIterator premier,
                    RandomAccessIterator dernier);

template <class RandomAccessIterator,
          class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator premier,
                    RandomAccessIterator dernier,
                    RandomNumberGenerator g);
```

C++ : STL - Les algorithmes
Opérations d'itération et de transformation

```
template <class InputIterator,
          class Function>
Function for_each(InputIterator premier,
                  InputIterator dernier,
                  Function f);

template <class InputIterator,
          class OutputIterator,
          class UnaryOperation>
OutputIterator transform(InputIterator premier,
                         InputIterator dernier,
                         OutputIterator destination,
                         UnaryOperation op);
```

C++ : STL - Les algorithmes
Opérations d'itération et de transformation

```
template <class InputIterator1,
          class InputIterator2,
          class OutputIterator,
          class BinaryOperation>
OutputIterator transform(InputIterator1 premier1,
                         InputIterator1 dernier1,
                         InputIterator2 premier2,
                         OutputIterator destination,
                         BinaryOperation op);
```

C++ : STL - Les algorithmes
Calculs

```
template <class InputIterator,
          class T>
T accumulate(InputIterator premier,
             InputIterator dernier,
             T init);

template <class InputIterator,
          class T,
          class BinaryOperation>
T accumulate(InputIterator premier,
             InputIterator dernier,
             T init,
             BinaryOperation op);
```

C++ : STL - Les algorithmes
Calculs

```
template <class InputIterator1,
          class InputIterator2,
          class T>
T inner_product(InputIterator1 premier1,
                InputIterator1 dernier1,
                InputIterator2 premier2,
                T init);

template <class InputIterator1,
          class InputIterator2, class T,
          class BinaryOperation1,
          class BinaryOperation2>
T inner_product(InputIterator1 premier1,
                InputIterator1 dernier1,
                InputIterator2 premier2,
                T init,
                BinaryOperation1 op1,
                BinaryOperation2 op2);
```

C++ : STL - Les algorithmes
Calculs

```
template <class InputIterator,
          class OutputIterator>
OutputIterator partial_sum(InputIterator premier,
                           InputIterator dernier,
                           OutputIterator destination);

template <class InputIterator,
          class OutputIterator,
          class BinaryOperation>
OutputIterator partial_sum(InputIterator premier,
                           InputIterator dernier,
                           OutputIterator destination,
                           BinaryOperation op);
```

C++ : STL - Les algorithmes
Calculs

```
template <class InputIterator,
          class OutputIterator>
OutputIterator adjacent_difference(
                          InputIterator premier,
                          InputIterator dernier,
                          OutputIterator destination);

template <class InputIterator,
          class OutputIterator,
          class BinaryOperation>
OutputIterator adjacent_difference(
                          InputIterator premier,
                          InputIterator dernier,
                          OutputIterator destination,
                          BinaryOperation op);
```

C++ : STL - Pointeurs automatiques

- Les pointeurs automatiques encapsulent des objets alloués par `new` ; l'objet encapsulé sera détruit en même temps que le pointeur automatique.
- On encapsule parfois temporairement de tels objets si une exception risque de survenir, ce qui permet de les libérer automatiquement.

C++ : STL - Pointeurs automatiques

```
template <class T> class auto_ptr{
  public:
  typedef T element_type;
  explicit auto_ptr(T *pointeur = 0) throw();
  auto_ptr(const auto_ptr &source) throw();
  template <class U>
  auto_ptr(const auto_ptr<U> &source) throw();
  ~auto_ptr() throw();

  auto_ptr &operator=(const auto_ptr &source) throw()
  template <class U>
  auto_ptr &operator=(const auto_ptr<U> &source) thro

  T &operator*() const throw();
  T *operator->() const throw();
  T *get() const throw();
  T *release() const throw();
};
```

C++ : STL - `valarray`

- La classe `valarray` est une classe template capable de stocker un tableau de valeurs de son type template.
- Il est possible de l'instancier pour tous les types de données pour lesquels les opérations définies sur la classe `valarray` sont elles-mêmes définies.
- La bibliothèque standard C++ garantit que la classe `valarray` est écrite de telle sorte que tous les mécanismes d'optimisation des compilateurs pourront être appliqués sur elle, afin d'obtenir des performances optimales.
- Chaque implémentation est libre d'utiliser les possibilités de calcul parallèle disponible sur chaque plate-forme, du moins pour les types pour lesquels ces fonctionnalités sont présentes.
- Par exemple, la classe `valarray` instanciée pour le type `float` peut utiliser les instructions spécifiques de calcul sur les nombres flottants du processeur si elles sont disponibles.

C++ : STL - valarray

```
// Construit un valarray de doubles
valarray<double> v1;

// Initialise un valarray de doubles explicitement
double valeurs[] = {1.2, 3.14, 2.78, 1.414, 1.732};
valarray<double> v2(valeurs,
                    sizeof(valeurs) / sizeof(double));

// Construit un valarray de 10 entiers initialises a 3
valarray<int> v3(3, 10);
```

C++ : STL - valarray

```
#include <iostream>
#include <valarray>

using namespace std;

int main(void)
{
  // Creation d'un valarray :
  valarray<double> v;
  cout << v.size() << endl;
  // Redimensionnement du valarray :
  v.resize(5, 3.14);
  cout << v.size() << endl;
  return 0;
}
```