C++ : Fonctions template

On peut écrire une fonction paramétrées 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){
  std::cout << "Types différents" << std::endl;
}

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

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

void comp_type(const int & x, const int& y){
  std::cout << "Deux entiers" << std::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>
T convert(const U& x){
  return (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<class<typename T, int N> Array>
void tri(Array<T,N>& t);  //trie le tableau t
```

C++ : Types template

On peut créer un type template de manière similaire à une fonction. 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;
}

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


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

C++ : Types template

À l'intérieur d'une classe template ou du corps de ses méthodes, on peut utiliser le type 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 définit 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.

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 doit être déclaré comme tel:

```
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 T&, const T&);
    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);
};
```

```
On peut aussi spécifier des paramètres de type "classe template".
```

Ceci n’est pas possible pour les fonctions template.

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

Si on a un type connu `typeA` à l'intérieur duquel est défini un type `typeInduit`, on a accès à ce second type en écrivant `typeA::typeInduit`. Le compilateur vérifie alors que `typeA` contient bien la déclaration de `typeInduit`.

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;
}
```

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

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 : 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*, dont le seul point commun est d'avoir 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.

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 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.

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.

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> &) throw();
    ~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

Tous les itérateurs de la STL dérivent de la classe `iterator` qui définit un certain nombre de types:

```
template <class Category,
          class T,              class Distance = ptrdiff_t,
          class Pointer = T*, class Reference = T &       >
struct iterator
{
    typedef T         value_type;
    typedef Distance  difference_type;
    typedef Pointer   pointer;
    typedef Reference reference;
    typedef Category  iterator_category;
};
```

C++ : STL - Les itérateurs

Afin d'assurer la compatibilité avec les pointeurs, le trait

```
template <class Iterator> struct iterator_traits {/*...*/};
```

redéfinit ces types et est spécialisée pour les pointeurs:

```
template <class T> struct iterator_traits<T *>
{
    typedef T                         value_type;
    typedef ptrdiff_t                 difference_type;
    typedef T                         *pointer;
    typedef T                         &reference;
    typedef random_access_iterator_tag iterator_category;
};
```

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

A partir d'un itérateur bidirectionnel, on peut définir un opérateur renversé dont le sens de parcours naturel est inverse:

```
template <class Iterator>
class reverse_iterator : public iterator<
        iterator_traits<Iterator>::iterator_category, /* etc */>{
public:
    typedef Iterator iterator_type;
    reverse_iterator();
    explicit reverse_iterator(Iterator iterateur);
    Iterator base() const;
    Reference operator*() const;     Pointer operator->() const;
    reverse_iterator &operator++();  reverse_iterator operator++(int);
    reverse_iterator &operator--();  reverse_iterator operator--(int);
    reverse_iterator operator+(Distance delta) const;
    reverse_iterator &operator+=(Distance delta);
    reverse_iterator operator-(Distance delta) const;
    reverse_iterator &operator-=(Distance delta);
    Reference operator[](Distance delta) const;
};
```

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;
```

C++ : STL - basic_string

La définition d'un `basic_string` requiert un trait, par défaut:

```
template<typename _CharT> struct char_traits
    {
      typedef _CharT          char_type; // type caractere
      typedef unsigned long   int_type;  // type valeur speciale (EOF)
      typedef long long       off_type;  // pos et deplnt
      typedef pos<mbstate_t>  pos_type;  // pos dans un flux
      typedef mbstate_t       state_type;// etat lors du transcodage
   };
```

La déclaration de la classe est alors:

```
template <class charT,
          class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
class basic_string
{
\\..
};
```

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 ds 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(..)`

- recharche : `find(mot,p=0)`, `rfind(mot,p=npos)`,

  `find_{first,last}_[not_]of(ch,p=0)`

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) throw();

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

C++ : STL - Paires

```
template <class T1, class T2>
struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;

    pair();
    pair(const T1 &, const T2 &);
    template <class U1, class U2> pair(const pair<U1, U2> &);
};
```

C++ : STL - Complexes

`complex<T>` est un type permettant de stocker des complexes.

- Fonctions mathématiques sur les complexes : `exp`, `cosh`, `sqrt`, etc.
- Les fonctions `abs`, `arg`, `norm`, `conj` retournent resp. le module, l'argument, le module^2 et le conjugué;
- `polar` construit un complexe à partir de son module et de son argument;
- Constructeur : 0 à 2 arguments;
- Surcharge des opérateurs arithmétiques.
- Méthodes `real()`, `imag()`

C++ : STL - Tableaux statiques de bits

La classe `bitset<N>` désigne un tableau de N bits.

- Constructeurs : sans argument, à partir d'un `unsigned long`, ou d'un `basic_string<>`.
- Conversion : `to_ulong()`, `to_string`
- Manipulation : opérateurs `&=`, `|=`, `^=`, `<<=`, `>>=`, `<<`, `>>`, `~`
- Comparaison : `==` et `!=`

C++ : STL - Tableaux statiques de bits

- `test(pos)`, `set(pos,v=true)` et `flip(pos)`

  resp. retourne, fixe, ou change la valeur du bit;

  `set()` et `reset()` fixe tous les bits resp. à 1 ou à 0.

  `[]` permet d'accéder à un bit géré par une classe interne `reference`; on peut écrire `t[45]=true;`

- Tests : `size()`(=N), `count()`, `any()`, `none()`

C++ : STL - Tableaux de valeurs

La classe `valarray<T>` implémentent des tableaux d'objets de type T.

Elle est écrite de sorte à utiliser toutes les optimisations possibles des compilateurs et de la plateforme : co-processeur arithmétique, parallélisme, etc...

C++ : STL - Tableaux de valeurs

- Constructeur : par défaut, taille, ou valeur initiale + taille, ou tableau + taille, ou valarray
- Opérations classiques sur les tableau;
- Opérations arithmétiques unaires (+, -, !, code~),

  affectation distribuée (`operator=(const T&)`, *=, -=, +=, /=, %=, ^=, &=, |=, <<= >>=)

  ou val par val (`operator=(const valarray<T>&)`).
- Accesseurs : `size()`, `sum()`, `min()`, `max()`, `shift(d)`, `cshift(d)`, `apply(fct)`, `resize(taille, i=T())`

C++ : STL - Tableaux de valeurs

L'opérateur `[]` des valarray est surchargé de sorte à pouvoir accepté au lieu d'un index habituel, des objets qui permettent une sélection plus puissante.

L'objet retourné alors par l'opérateur `[]` est d'un type intermédiaire qui ne doit être utilisé que pour construire un nouveau valarray ou modifier le valarray sur lequel la sélection est opérée.

C++ : STL - Tableaux de valeurs

Expression booléenne On peut utiliser une expression booléenne, dans laquelle l'identifiant correspondant au valarray sera remplacé par chaque indice; la sélection correspondra aux indices pour lesquels l'expression est vraie.

```
int main(){
  valarray<int> t(10);
  for(size_t i=0; i<t.size(); ++i) t[i]=i*i;

  valarray<int> h(t[t%2==0]);
  for(size_t i=0; i<h.size(); ++i) cout << h[i] << ' ';
  cout << endl;

  t[t%2==0] += valarray<int> (5,10);
  for(size_t i=0; i<t.size(); ++i) cout << t[i] << ' ';
  cout << endl;
  return 0;
}
```

C++ : STL - Tableaux de valeurs

Indexation explicite On peut utiliser au lieu d'un entier, un

`valarray<size_t>` qui indique les positions sélectionnées:

```
int main(){
  valarray<int> t(10);
  for(size_t i=0; i<t.size(); ++i) t[i]=i*i;

  size_t s[]={2,3,5,8}; valarray<size_t> sel(s,4);

  valarray<int> h(t[sel]);
  for(size_t i=0; i<h.size(); ++i) cout << h[i] << ' ';
  cout << endl;

  t[sel] += valarray<int> (5,10);
  for(size_t i=0; i<t.size(); ++i) cout << t[i] << ' ';
  cout << endl;
  return 0;
}
```

C++ : STL - Tableaux de valeurs

Indexation explicite par slice Au lieu de définir explicitement le tableau d'indice, on peut utiliser un objet de type `slice` dans le constructeur duquel on précise un premier indice i, un nombre d'indice n et un pas k.

Ceci définit un ensembles d'indices $\{i, i+k, ...i+k(n-1)\}$.

C++ : STL - Tableaux de valeurs

```
int main(){
  valarray<int> t(10);
  for(size_t i=0; i<t.size(); ++i) t[i]=i*i;

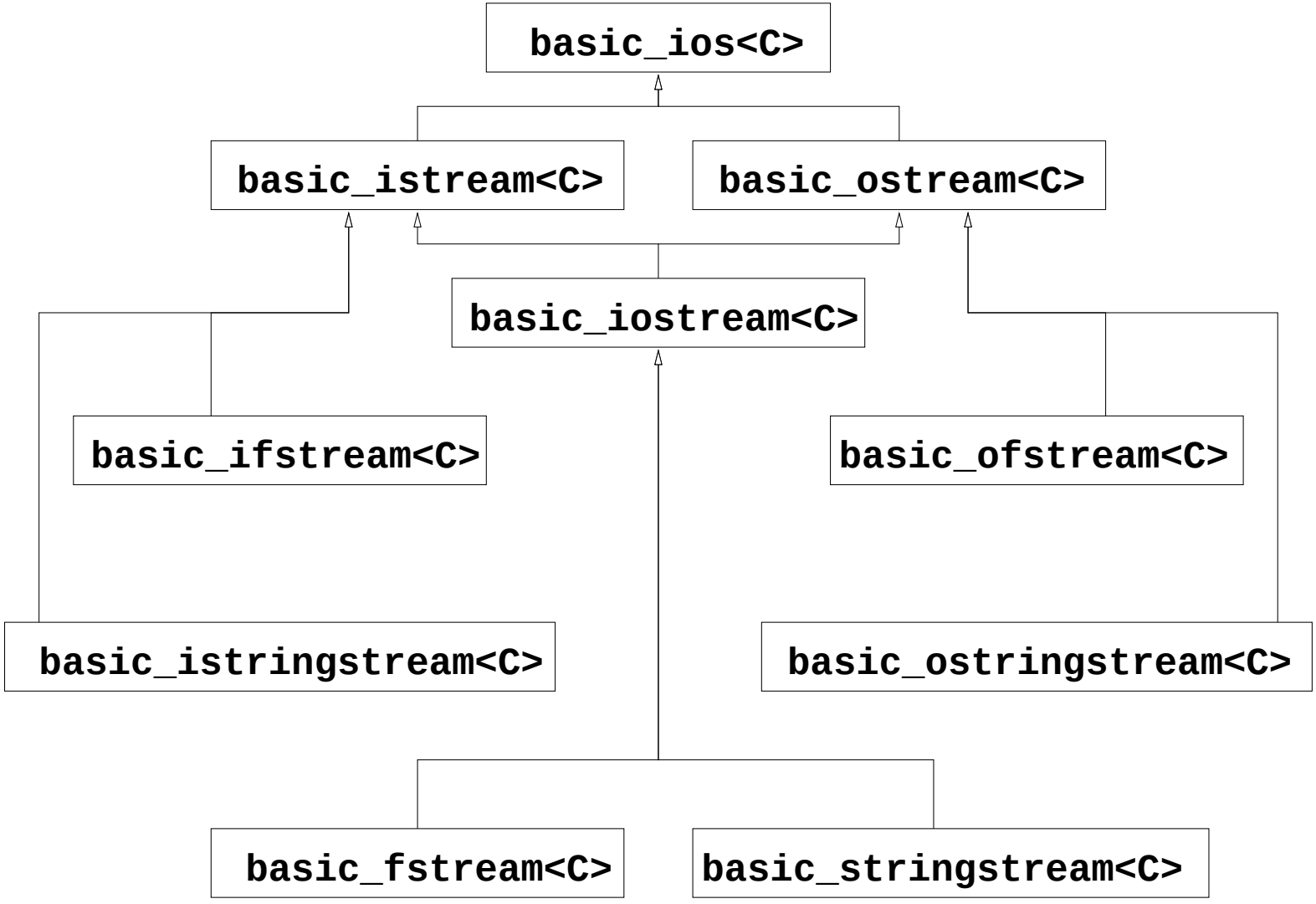
  slice sel(1,3,2);

  valarray<int> h(t[sel]);
  for(size_t i=0; i<h.size(); ++i) cout << h[i] << ' ';
  cout << endl;

  t[sel] += valarray<int> (5,10);
  for(size_t i=0; i<t.size(); ++i) cout << t[i] << ' ';
  cout << endl;
  return 0;
}
```

Il existe un type `gslice` qui permet de générer des ensembles en utilisant plusieurs suite arithmétiques...

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 notemment 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 écrire
  //...
}
```

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.

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:

`setbase(int base)`	fixe la base
`setprecision(int)`	nombre de caractères significatifs
`setw(int)`	largeur min de la donnée suivante
`setfill(char)`	caractère de remplissage
`resetiosflags(flags)`	efface les flags d'option pour le flot
`setiosflags(flags)`	positionne les flags d'option

Ces flags, de type `ios_base::fmtflags` ont pour valeurs possibles `boolalpha`, `hex`, `oct`, `dec`, `fixed`, `scientific`, `left`, `right`, `internal`, `showbase`, `showpoint`, `showpos`, `uppercase`, `unitbuf`, `skipws`, `adjustfield`, `basefield`, `floatfield`

C++ : STL - Les flots

Pour définir un manipulateur avec argument(s), il faut définir deux choses:

– une classe stockant les arguments du manipulateur et pour laquelle on redéfinit l'opérateur `<<` sur `ostream` pour effectuer la manipulation;

– le manipulateur qui est une fonction qui prend les arguments voulus et renvoie un objet de cette classe.

En réalité, dans le fichier `iomanip`, il existe un handler générique qui prend l'adresse d'une fonction qui manipule le flot...

C++ : STL - Les flots

```
struct manip_h{
 int i;
 manip_h(int i) : i(i) {}
 friend ostream& operator<<(ostream& o, const manip_h& h){
   for(int k=0; k<i; ++k) o << '-';
   return o;
 }
};

manip_h mon_manip(int i){
  return manip_h(i);
}
```

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, notemment la gestion des erreurs.

L'état d'un flot est donné par des valeurs définies dans `ios_base`:

`goodbit`	Etat normal
`eofbit`	Fin de lecture ou d'écriture
`failbit`	Erreur "logique"
`badbit`	Erreur fatale (matérielle,...)

C++ : STL - Les flots

On accède à ces valeurs à travers les méthodes de `ios`

`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,...)
`rdstate()`	retourne le statut d'erreur
`clear(i)`	fixe le statut d'erreur à `i`

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:

- `plus<T>`, `minus<T>`, `multiplies<T>`, `divides<T>`, `modulus<T>`, `negate<T>`
- Prédicats : `equal_to<T>`, `not_equal_to<T>`, `greater<T>`, `less<T>`, `greater_equal<T>`, `less_equal<T>`, `logical_and<T>`, `logical_or<T>`

C++ : STL - Les foncteurs

- `unary_negate<UPredicate>`, `unary_negate<BPredicate>` (dont on peut créer une instance avec `not1(..)` ou `not2(..)`);
- Réducteurs : `binder1st<Op>`, `binder2st<Op>` dont on peut créer une instance avec `bind1st(Op& f, const T& val)` ou `bind2st(Op& f, const T& val)`

C++ : STL - Les foncteurs

- Adaptateurs pour fonctions :

  `pointer_to_unary_function<Arg,Result>`,

  `pointer_to_binary_function<Arg1,Arg2,Result>`,

  dont on peut créer une instance avec `ptr_fun(..)`;

- Adaptateurs pour méthodes :

  `mem_fun_[ref_]t<Class, Result>`,

  `class mem_fun1_[ref_]t<Class, Arg, Result>`

  dont on peut créer une instance avec `mem_fun[_ref](..)`,

  donne un foncteur avec une méthode

  `Result operator()(Class[&] o[, Arg x])`;

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 notemment sur les conteneurs. Il s'agit de fonctions définies pour la plupart dans l'en-tête `algorithm`

Dans la suite on note

OI = output iterator, II = input iterator,

FI = forward iterator, BI = bidirectional iterator,

RI = random iterator

C++ : STL - Les algorithmes

- `void fill(FI deb, FI fin, val)` et
  `void fill_n(OI premier, nbe, val)`
  remplissent un conteneur avec `val` soit dans un intervalle, soit avec un certain nombre de valeurs
- `void generate(FI deb, FI fin, gen)` et
  `void generate_n(OI deb, nbe, gen)`
  idem, sauf que `gen` est un foncteur (sans argument) appelé pour remplir chaque case.

C++ : STL - Les algorithmes

- `OI copy(II deb, II fin, OI dest)` copie l'intervalle de valeurs;

  `BI2 copy_backward(BI1 deb, BI1 fin, BI2 dest)`

- `FI2 swap_ranges(FI1 deb, FI1 fin, FI2 dest)` échanges les valeurs de l'intervalle [deb, fin[avec celles de [dest, ret[

C++ : STL - Les algorithmes

- `FI remove(FI deb, FI fin, val)` supprime `val`

  `OI remove_copy(II deb, II fin, OI dest, val)` copie les éléments différents de `val`

  `FI remove_if(FI deb, FI fin, UPred p)`

  supprime les élts lorsque `p` est vrai

  `OI remove_copy_if(II deb, II fin, OI dest, UPred p)`

C++ : STL - Les algorithmes

- ```
  void replace(FI de, FI fin, old, new)
  void replace_copy(II deb, II fin, OI dest,
  old, new)
  void replace_if(FI deb, FI fin, UPred p,
  new)
  void replace_copy_if(II deb, II fin, OI
  dest, UPred p, new)
  ```
  ```

C++ : STL - Les algorithmes

- `void rotate(FI deb, FI pivot, FI fin)` rotation des valeurs de l'intervalle [deb, fin[de sorte que pivot se retrouve au début

  `void rotate_copy(FI deb, FI pivot, FI fin, OI dest)`

- `bool next_permutation(BI deb, BI fin)` calcule la permutation suivante des éléments, qui doivent supporter `<`

  `bool prev_permutation(BI deb, BI fin)`

- `void reverse(BI deb, BI fin)`,

  `OI reverse_copy(BI deb, BI fin, OI dest)`

C++ : STL - Les algorithmes

- `void random_shuffle(RI deb, RI fin)`, `void random_shuffle(RI deb, RI fin, randgen)` où `randgen` est un foncteur d'argument `d` de type `difference_type` qui retourne un entier "aléatoire" de [0,d-1[

C++ : STL - Les algorithmes

- `Functor for_each(II deb, II fin, f)` applique le foncteur `f` à chaque élément de l'intervalle

  `OI transform(II deb, II fin, OI dest, f)`

  `OI transform(II1 deb1, II1 fin1, II2 deb2, OI dest, f)` applique `f(*it1,*it2)` dans les intervalles

- `difference_type count(II deb, II fin, val)`

  `difference_type count_if(II deb, II fin, UPred p)`

C++ : STL - Les algorithmes

Les algorithmes suivant sont définis dans l'en-tête `numeric`

- `T accumulate(II deb, II fin, T init,[ op+])`
  somme cumulée
- `T inner_product(II1 deb1, II1 fin1, II2 deb2, T init,[ op+, op*])` produit scalaire
- `OI partial_sum(II deb, II fin, OI dest,[ op+])` somme 2 à 2
- `OI adjacent_difference(II deb, II fin, OI dest,[ op+])` différence 2 à 2

C++ : STL - Les algorithmes

- `II find(II deb, II fin, val)`

  `II find_if(II deb, II fin, UPred p)`

- `II find_first_of(II deb, II fin, FI vd, FI vf, [is_eq])` cherche le premier emplacement de [deb, fin[qui contient une valeur de [vd, vf[

C++ : STL - Les algorithmes

- `FI1 search(FI1 deb, FI1 fin, FI2 vd, FI2 vf, [is_eq])` cherche la séquence [vd,vf[dans [deb,fin[

  `find_end` idem mais pour la dernière occurence

- `FI search(FI deb, FI fin, nbe, val, [is_eq])` cherche `nbe val` consécutifs dans [deb,fin[

- `FI adjacent_find(FI deb, FI fin, [is_eq])`

  trouve le premier doublon.

C++ : STL - Les algorithmes

- `void make_heap(RI deb, RI fin, [ordre])`
  construit un tas ($O(n)$)
- `void pop_heap(RI deb, RI fin, [ordre])` sort la première valeur du tas et réorganise ($O(\log n)$)
- `void push_heap(RI deb, RI fin+1, [ordre])`
  insère la valeur `*fin` dans le tas [deb, fin[($O(\log n)$)
- `void sort_heap(RI deb, RI fin, [ordre])` trie le tas ($O(n \log n)$)

C++ : STL - Les algorithmes

- `void sort(RI deb, RI fin, [ordre])` trie le conteneur ($O(n \log n)$)
- `void sort_stable(RI deb, RI fin, [ordre])` idem, et conserve l'ordre des éléments équivalents($O(n \log n)$)
- `void partial_sort(RI deb, RI pivot, RI fin, [ordre])` trie les éléments inférieurs à `*pivot`
  `RI partial_sort_copy(II deb, II fin, RI dr, RI fr, [ordre])` trie les `fr-dr` plus petits élts

C++ : STL - Les algorithmes

- `void nth_element(RI deb, RI n, RI fin, [ordre])` place le (`n-deb`)-ème élément en position `n` (linéaire)
- `FI partition(FI deb, FI fin, UPred p)` place dans [deb, ret[les éléments tq `p` est vrai

  `FI partition_stable(FI deb, FI fin, UPred p)`

C++ : STL - Les algorithmes

- `FI min_element(FI deb, FI fin, [ordre])`
- `FI max_element(FI deb, FI fin, [ordre])`
- `bool binary_search(FI deb, FI fin, val, [ordre])` teste s'il existe `it` tq `*it`$\leqslant$`val`$\leqslant$`*it`

C++ : STL - Les algorithmes

- `FI lower_bound(FI deb, FI fin, val, [ordre])` retourne `it` tq `*(it-1)` $<$ `val` $\leqslant$ `*it`
- `FI upper_bound(FI deb, FI fin, val, [ordre])` retourne `it` tq `*(it-1)` $\leqslant$ `val` $<$ `*it`
- `pair<FI, FI> equal_range(FI deb, FI fin, val, [ordre])` retourne (lower,upper)

C++ : STL - Les algorithmes

- `bool equal(II1 deb1, II1 fin1, II2 deb2, [is_eq])` teste l'égalité des conteneurs
- `pair<II1, II2> mismatch(II1 deb1, II1 fin1, II2 deb2, [is_eq])` premier endroit différent
- `bool lexicographical_compare(II1 deb1, II1 fin1, II2 deb2, II2 fin2, [order])`

C++ : STL - Les algorithmes

- `bool includes(II1 deb1, II1 fin1, II2 deb2, II2 fin2, [order])` teste si $C_2 \subseteq C_1$
- `OI set_intersection(II1 deb1, II1 fin1, II2 deb2, II2 fin2, OI dest, [order])` calcule $C_1 \cap C_2$

  De même,

  `set_union` ($C_1 \cup C_2$),

  `set_difference` ($C_1 \setminus C_2$),

  `set_symmetric_difference` ($C_1 \Delta C_2$)

C++ : STL - Les algorithmes

- `OI merge(II1 deb1, II1 fin1, II2 deb2, II2 fin2, OI dest, [order])` fusionne C_1 et C_2
- `void inplace_merge(BI deb, BI mil, BI fin, [order])` fusionne [deb, mil[et [mil, fin[