# Programmation Orientée Objet (C++) : Librairies standards

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C



EPFL





## Objectifs du cours d'aujourd'hui

L'objectif du cours d'aujourd'hui est de vous présenter (sommairement) un certains nombre d'**outils** standards existant en C++

Le but ici n'est pas d'être exhaustif, mais simplement de vous :

- informer de l'existence des **principaux** outils
- faire prendre conscience d'aller **lire/chercher dans la documentation** les éléments qui peuvent vous être utiles



EPFL





## Bibliothèque standard

La bibliothèque standard (d'outils) C++ **facilite la programmation** et permet de la rendre **plus efficace**, si tant est que l'on connaisse bien les outils qu'elle fournit.

Cette bibliothèque est cependant vaste et complexe, mais elle peut dans la plupart des cas s'utiliser de façon très simple, facilitant ainsi la réutilisation des structures de données abstraites et des algorithmes sophistiqués qu'elle contient.

La bibliothèque standard C++11 est formée de 79 « paquets » :

- 33 « classiques » (C++98)
- 20 nouveaux (C++11)
- les 26 bibliothèques C (C99)



EPFL





## Contenu de la bibliothèque standard

La bibliothèque standard C++ contient 33 « paquets » de C++-98 :

| | |
|---|---|
| **<algorithm>** | plusieurs algorithmes utiles |
| <bitset> | gestions d'ensembles de bits |
| **<complex>** | les nombres complexes |
| <deque> | tableaux dynamiques avec `push_front` |
| <exception> | diverses fonctions aidant à la gestion des exceptions |
| **<fstream>** | manipulation de fichiers |
| <functional> | objets fonctions |
| **<iomanip>** | manipulation de l'état des flots |
| <ios> | définitions de base des flots |
| <iosfwd> | anticipation de certaines déclarations de flots |
| **<iostream>** | flots standards |
| <istream> | flots d'entrée |
| <iterator> | itérateurs |
| <limits> | diverses bornes concernant les types numériques |
| <list> | listes doublement chaînées |
| <locale> | contrôles liés au choix de la langue |



EPFL

# Contenu de la bibliothèque standard (2)

| | |
|---|---|
| **<map>** | tables associatives clé–valeur ordonnées |
| <memory> | gestion mémoire pour les containers |
| <new> | gestion mémoire |
| <numeric> | fonctions numériques |
| <ostream> | flots de sortie |
| <queue> | files d'attente |
| **<set>** | ensembles ordonnés |
| **<sstream>** | flots dans des chaînes de caractères |
| **<stack>** | piles |
| **<stdexcept>** | gestion des exceptions |
| <streambuf> | flots avec tampon (buffer) |
| **<string>** | chaînes de caractères |
| <strstream> | flots dans des chaînes de caractère [en mémoire] |
| <typeinfo> | information sur les types |
| <utility> | divers utilitaires |
| <valarray> | tableaux orientés vers les valeurs |
| **<vector>** | tableaux dynamiques |







# Contenu de la bibliothèque standard (3)

La bibliothèque standard C++ contient 20 nouveaux « paquets » de C++11 :

| | |
|---|---|
| **<array>** | tableaux de taille fixe |
| <atomic> | expression atomique |
| <chrono> | heures et chronomètres |
| <codecvt> | conversions d'encodage de caractères |
| <condition_variable> | concurence (multi-thread) |
| **<forward_list>** | listes simplement chaînées |
| <future> | concurence (multi-thread) |
| <initializer_list> | listes d'initialisation |
| <mutex> | concurence (multi-thread) |
| **<random>** | nombres aléatoires |
| <ratio> | constantes rationnelles ($\mathbb{Q}$) |
| <regex> | expressions régulières |
| <scoped_allocator> | allocation mémoire |
| <system_error> | erreurs système |
| <thread> | concurence (multi-thread) |
| <tuple> | *n*-uples |
| <type_traits> | caractéristiques de types |







# Contenu de la bibliothèque standard (4)

| | |
|---|---|
| <typeindex> | utiliser les types comme index de containers |
| **<unordered_map>** | tables associatives non ordonnées |
| **<unordered_set>** | ensembles non ordonnés |

Il existe aussi dans les outils standards les 26 « paquets» venant du langage C (C99) :

| | |
|---|---|
| <cassert> | test d'invariants lors de l'exécution |
| <ccomplex> | (inutile en C++) = <complex> |
| <cctype> | diverses informations sur les caractères |
| <cerrno> | code d'erreurs retournés dans la bibliothèque standard |
| <cfenv> | manipulation des règles de gestion des nombres en virgule flotante |
| <cfloat> | diverses informations sur la représentation des réels |
| <cinttypes> | `int` de taille fixée (C99) |
| <ciso646> | (inutile en C++) |
| <climits> | diverses informations sur la représentation entiers |
| <clocale> | adaptation à diverses langues |
| **<cmath>** | diverses définitions mathématiques |
| <csetjmp> | branchement non locaux |







# Contenu de la bibliothèque standard (5)

| | |
|---|---|
| <csignal> | contrôle des signaux (processus) |
| <cstdalign> | (inutile en C++) |
| <cstdarg> | nombre variables d'arguments |
| <cstdbool> | (inutile en C++) |
| <cstddef> | diverses définitions utiles (types et macros) |
| <cstdio> | entrées sorties de base |
| <cstdint> | sous-partie de `cinttypes` |
| <cstdlib> | diverses opérations de base utiles |
| <cstring> | manipulation des chaînes de caractères à la C |
| <ctgmath> | <cmath> + <complex> |
| <ctime> | diverses conversions de date et heures |
| <cuchar> | char de 16 ou 32 bits |
| <cwchar> | utilisation des caractères étendus |
| <cwctype> | classification des codes de caractères étendus |

# Outils standards

On distingue plusieurs types d'outils. Parmi les principaux :

- les containers de base
- les containers avancés (appelés aussi « adaptateurs »)
- les itérateurs
- les algorithmes
- les outils numériques
- les traitements d'erreurs
- les chaînes de caractères
- les flots







# Outils standards (2)

Les outils les plus utilisés par les débutants sont :

- les chaînes de caractères (`string`) ✔
- les flots (`stream`) ✔
- les tableaux dynamiques (`vector`) [container] ✔
- les listes chaînées (`list`) [container avancé]
- les piles (`stack`) [container avancé]
- les algorithmes de tris (`sort`)
- les algorithmes de recherche (`find`)
- les itérateurs (`iterators`)







# Plan

Présentons maintenant certains des outils standards de façon plus détaillée.

- `list` [container]
- `set`/`unordered_set` [container]
- `iterator`
- `map`/`unordered_map` [container]
- `stack` [container avancé]
- `queue` [container avancé]
- `sort`
- `find`
- `complex`
- `cmath`
- Nombres aléatoires







# Containers

Comme le nom l'indique, les containers sont des structures de données abstraites (SDA) servant à **contenir** (« collectionner ») **d'autres objets**.

Vous en connaissez déjà plusieurs : les tableaux, les piles et les listes chaînées.

Il en existe plusieurs autres, parmi lesquels, les files d'attentes (`queue`), les ensembles (`set`, `unordered_set`) et les tables associatives (`map`, `unordered_map`).

# Containers (2)

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier... comme dans une file d'attente à un guichet !

(alors que dans une pile « normale », c'est toujours le dernier arrivé qui est dépilé en premier)

Les `set` permettent de gérer des **ensembles** (finis !) au sens mathématique du terme : collection d'éléments où chaque élément n'est présent qu'une seule fois.

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple

```
tab["Informatique"]
```







# Containers (3)

Tous les containers contiennent les méthodes suivantes :

`bool empty()` : le containers est-il vide ?

`unsigned int size()` : nombre d'éléments contenus dans le container

`void clear()` : vide le container

`iterator erase(it)` : supprime du container l'élément pointé par *it*. *it* est un itérateur (généralisation de la notion de pointeur, voir quelques transparents plus loin)

Ils possèdent également tous les méthodes `begin()` et `end()` que nous verrons avec les itérateurs.

Passons maintenant à quelques containers particuliers







# Liste (doublement) chaînées

Les listes (doublement) chaînées sont, comme les tableaux dynamiques, des SDA **séquentielles**, c'est-à-dire stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes chaînées sont définies dans la bibliothèque `list` et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
list<int> maliste;
```

(quelques) méthodes des listes chaînées :

| | |
|---|---|
| `Type& front()` | retourne le premier élément de la liste |
| `Type& back()` | retourne le dernier élément de la liste |
| `void push_front(Type)` | ajoute un élément en tête de liste |
| `void push_back(Type)` | ajoute un élément en queue de liste |
| `void pop_front()` | supprime le premier élément |
| `void pop_back()` | supprime le dernier élément |
| `void insert(iterator, Type)` | insertion avant un élément de la liste désigné par un itérateur |







# C++11 Liste chaînées

Les listes chaînées sont, comme les tableaux dynamiques, des SDA **séquentielles**, c'est-à-dire stockant des séquences (ordonnées) d'éléments.

Par contre dans une liste chaînée, l'accès direct à un élément n'est pas possible, contrairement aux tableaux dynamiques.

Les listes simplement chaînées sont définies dans la bibliothèque `forward_list` et se déclarent de façon similaire à des tableaux dynamiques, par exemple

```
forward_list<int> maliste;
```

(quelques) méthodes des listes chaînées :

| | |
|---|---|
| `Type& front()` | retourne le premier élément de la liste |
| `void push_front(Type)` | ajoute un élément en tête de liste |
| `void pop_front()` | supprime le premier élément |
| `void insert(iterator, Type)` | insertion avant un élément de la liste désigné par un itérateur |

# Tableaux dynamiques : petit complément

Pour accéder directement à un élément d'un tableau dynamique (`vector`) on utilise l'opérateur `[]` : `tab[i]`.

Il existe une autre méthode pour cet accès : `at(n)` qui, à la différence de `[n]`, lance une l'exception `out_of_range` (de la bibliothèque `<stdexcept>`) si `n` n'est pas un index correct.

Exemple :

```
#include <vector>
#include <stdexcept>
...
vector<int> v(5,3); // 3, 3, 3, 3, 3
int n(12);
try {
   cout << v.at(n) << endl;
}
catch (out_of_range) {
   cerr << "Erreur : " << n << " n'est pas correct pour v"
        << endl
        << "qui ne contient que " << v.size()
                << " elements." << endl;
}
```







# Ensembles – Exemple

Les ensembles (au sens mathématique) sont implémentés dans la bibliothèque `<set>`. Ils ne peuvent cependant contenir que des éléments du même type, lesquels sont ordonnés par `operator<`.

(Pour des éléments de même type mais non ordonnés, i.e. *sans* `operator<`, on utilisera un `unordered_set`.)

On déclare un ensemble comme les autres containers, en spécifiant le type de ses éléments, par exemple :
`set<char> monensemble;`

Les ensembles n'étant pas des SDA séquentielles, l'accès direct à un élément n'est pas possible.

(quelques) méthodes des ensembles :

| | |
|---|---|
| `insert(Type)` | insère un élément s'il n'y est pas déjà |
| `erase(Type)` | supprime l'élément (s'il y est) |
| `find(Type)` | retourne un itérateur indiquant l'élément recherché |

a

À noter que la bibliothèque `<algorithm>` fournit des fonctions pour faire la réunion, l'intersection et la différence d'ensembles.







# Ensembles – Exemple

```
#include <set>
...
set<char> voyelles;

voyelles.insert('a');
voyelles.insert('b');
voyelles.insert('e');
voyelles.insert('i');
voyelles.erase('b');
voyelles.insert('e'); /* n'insere pas 'e' car *
                       * il y est deja */
```

Comment parcourir cet ensemble ?

```
for (unsigned int i(0); i < voyelles.size(); ++i)
        cout << voyelles[i] << endl;
```

ne fonctionne pas car c'est une SDA non-indexé (et même non-séquentielle).







# Ensembles – parcours

Comment parcourir cet ensemble ?

En C++11 c'est facile :

```
for (auto const v : voyelles)
    cout << v << endl;
```

Il y a aussi un autre moyen, plus avancé :

☞ utilisation d'**itérateurs**

# Itérateurs

Les **itérateurs** sont une SDA **généralisant** d'une part des accès par index (SDA séquentielles) et d'autre part les pointeurs, dans le cas de **containers**.

Ils permettent :

- de parcourir de façon itérative les containers
- d'indiquer (i.e. de pointer sur) un élément d'un container

Il existe en fait 7 sortes d'itérateurs, mais nous ne parlons ici que de la plus générale, qui permet de tout faire : lecture et écriture du containers, aller en avant ou en arrière (accès quelconque en fait).







# Itérateurs (2)

Un itérateur associé à un container `C<type>` se déclare simplement comme `C<type>::iterator nom;`

Exemples :

```
vector<double>::iterator i;

set<char>::iterator j;
```

Il peut s'initialiser grâce aux méthodes `begin()` ou `end()` du container, voire d'autres méthodes spécifiques, comme par exemple `find` pour les containers non-séquentiels.

Exemples :

```
vector<double>::iterator i(monvect.begin());

set<char>::iterator j(monset.find(monelement));
```

L'élément indiqué par l'itérateur `i` est simplement `*i`, comme pour les pointeurs.







# Retour sur l'exemple des ensembles

Pour parcourir notre ensemble précédent, nous devons donc faire :

```
for (set<char>::iterator i(voyelles.begin());
       i != voyelles.end(); ++i)
    cout << *i << endl;
```

Exemple d'utilisation de `find` :

```
set<char>::iterator i(voyelles.find('c'));

if (i == voyelles.end())
    cout << 'c' <<  "n'est pas dans l'ensemble" << endl;
else
    cout << *i << "est dans l'ensemble" << endl;
```







# Code complet de l'exemple

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

int main() {
  set<char> voyelles;
  voyelles.insert('a');
  voyelles.insert('b');
  voyelles.insert('e');
  voyelles.insert('i');
  voyelles.insert('a'); // ne fait rien car 'a' y est deja
  voyelles.erase('b');  // supprime 'b'

  // parcours l'ensemble
  for (set<char>::iterator i(voyelles.begin()); i!=voyelles.end(); ++i)
    cout << *i << endl;

  // recherche d'un element
  set<char>::iterator element(voyelles.find('c'));
  if (element == voyelles.end())
    cout << "l'element n'est pas dans l'ensemble" << endl;
  else
    cout << *element << " est dans l'ensemble" << endl;

  return 0;
}
```

# Suppression d'un élément d'un container

On a vu que tout container possédait une méthode

```
iterator erase(it)
```

permettant de supprimer un élément, mais...

**Attention !**on **ne** peut **pas** continuer à utiliser l'itérateur `it` sans autre !
(plus exactement : `erase` rend invalide tout itérateur et référence situé(e) au dela du premier point de suppression)

Exemple d'**erreur** classique :

```
vector<double> v;
...
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) v.erase(i);
```

(avec `bool cond(double);`)
**n'est pas** correct («`Segmentation fault`»)

pas plus que :

```
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) i = v.erase(i);
```







# Suppression d'un élément d'un container (2)

Ce qu'il faut faire c'est :

```
vector<double>::iterator next;
for (vector<double>::iterator i(v.begin()); i != v.end();
     i = next) {
  if (cond(*i)) { next = v.erase(i); }
  else          { next = ++i;        }
}
```

ou mieux en utilisant `remove_if` (ou `remove`) de `<algorithm>` :

```
v.erase(remove_if(v.begin(), v.end(), cond), v.end());
```

mais qui sont de toutes façons «**coûteux**» ($\mathcal{O}(\texttt{v.size()}^2)$) (voir transparent suivant)







# Suppression d'un élément d'un container (3)

En effet, un tableau dynamique **n'est pas la bonne SDA** si l'on veut détruire un élément au milieu **et** garder l'ordre
(utiliser plutôt des *listes chaînées* pour cela)

Note : si l'on ne tient pas à garder l'ordre, on peut toujours faire :

```
for (unsigned int i(0); i < v.size(); ++i)
    if (cond(v[i])) {
        v[i] = v[v.size()-1];
        v.pop_back();
        --i;
    }
```







# Tables associatives

Les **tables associatives** sont une généralisation des tableaux où les index ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple
`tab["Informatique"]`

On parle d'« associations clé–valeur »

Les tables associatives sont définies dans la bibliothèque `<map>`.

Elles nécessitent deux types pour leur déclaration : le type des « clés » (les index) et le type des éléments indexé.

Par exemple, pour indexer des nombres réels par des chaînes de caractères on déclarera :

```
map<string,double> une_variable;
```

Si l'ordre (`operator<`) des clés n'importe pas, on utilisera une `unordered_map`.

# Tables associatives – exemple

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

int main()
{
  map<string,double> moyenne;

  moyenne["Informatique"] = 5.5;
  moyenne["Physique"] = 4.5;
  moyenne["Histoire des maths"] = 2.5;
  moyenne["Analyse"] = 4.0;
  moyenne["Algebre"] = 5.5;

  // parcours de tous les elements
  for (map<string,double>::iterator i(moyenne.begin());
       i != moyenne.end(); ++i)
    cout << "En " << i->first << ", j'ai " << i->second
         << " de moyenne." << endl ;

  // recherche
  cout << "Ma moyenne en Informatique est de ";
  cout << moyenne.find("Informatique")->second << endl;

  return 0;
}
```







# Piles et files

Les piles ont déjà été vues au dernier cours. Pour utiliser celles de la STL : `#include <stack>`

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Elles sont définies dans la bibliothèque `<queue>`.

Une pile de type *type* se déclare par `stack<type>` et une file d'attente par `queue<type>`. Par exemple :

```
stack<double> une_pile;
queue<char> attente;
```

méthodes :

| | |
|---|---|
| `Type top()` | accède au premier élément (sans l'enlever) |
| `void push(Type)` | empile/ajoute |
| `void pop()` | dépile/supprime |
| `bool empty()` | teste si la pile/file est vide |







# Piles – exemple

reprise de l'exemple du dernier cours :

```
#include <stack>
using namespace std;
...
bool check(string s) {
  stack<char> p;
  for (unsigned int i(0); i < s.size(); ++i) {
    if ((s[i] == '(') || (s[i] == '['))
      p.push(s[i]);
    else if (s[i] == ')') {
      if ((!p.empty()) && (p.top() == '('))
        p.pop();
      else
        return false;
    } else if (s[i] == ']') {
      if ((!p.empty()) && (p.top() == '['))
        p.pop();
      else
        return false;
    }
  }
  return p.empty();
}
```







# Algorithmes

La bibliothèque `algorithm` (i.e. `#include <algorithm>`) définit différents types d'algorithmes généraux :

- de séquencement
  quelques exemples : `for_each`, `find`, `random_shuffle`, `copy`
- de tris
  `sort`, mais aussi bien d'autres
- numériques
  `inner_product`, `partial_sum`, `adjacent_difference`

3 exemples ici :

- `find`
- `copy` et les `output_iterators`
- `sort`

pour les autres : référez-vous à la documentation

# find

`find` est un algorithme général permettant de faire des recherches dans (une partie d')un container.
Son prototype général est :

```
iterator find(iterator debut, iterator fin, Type valeur);
```

qui cherche `valeur` entre `debut` (inclu) et `fin` (exclu). Il retourne un itérateur sur le contenu correspondant à la valeur recherchée ou `fin` si cette valeur n'est pas trouvée.

Exemple :

```
list<int> uneliste;

uneliste.push_back(3);
uneliste.push_back(1);
uneliste.push_back(7);

list<int>::iterator result(find(uneliste.begin(),
                                uneliste.end(), 7));
if (result != uneliste.end()) cout << "dans la liste";
else                          cout << "pas dans la liste";
cout << endl;
```







# copy

`copy` est un algorithme général pour copier (une partir d')un container dans un autre.

Son prototype général est :

```
OutputIterator copy(InputIterator debut, InputIterator fin,
                    OutputIterator resultat);
```

qui copie le contenu compris entre `debut` (inclus) et `fin` (exclus) vers `resultat` (inclus) et les positions suivantes (itérateurs).

La valeur de retour est `resultat + (fin - debut)`.

**Attention !**Notez bien que cela *copie* des éléments, mais ne fait pas d'insertion : il faut absolument que `resultat` ait (i.e. pointe sur) la place nécessaire !

Exemple :

```
copy(unensemble.begin(), unensemble.end(), untableau.begin());
```

Notez que l'on peut ainsi copier des données d'une SDA dans une autre SDA d'un autre type.







# copy (2)

`copy` peut être très utile pour afficher le contenu d'un container sur un flot en utilisant un `ostream_iterator` (je ne donne qu'un exemple ici :)

```
copy(container.begin(), container.end(),
     ostream_iterator<int>(cout, ", "));
```

`container` contenant ici des `int`, son contenu sera affiché sur `cout`, séparé par des ', '.







# copy – Exemple complet

```
#include <iostream>
#include <set>
#include <vector>
#include <iterator>
using namespace std;
int main() {
  set<double> unensemble, unautre;

  unensemble.insert(1.1);
  unensemble.insert(2.2);
  unensemble.insert(3.3);

  // copy(unensemble.begin(), unensemble.end(), unautre.begin());
  // ne fonctionne pas ("assignment of read-only location")
  // car unautre n'a pas la taille suffisante.

  vector<double> untableau(unensemble.size()); // prevoit la place

  copy(unensemble.begin(), unensemble.end(), untableau.begin());

  // output
  cout << "untableau = ";
  copy(untableau.begin(), untableau.end(),
       ostream_iterator<double>(cout, ", "));
  cout << endl;

  return 0;
}
```

# sort

`sort` permet de trier des SDA implémentées sous forme de containers

La version la plus simple de tri est (il y en a d'autres) :

```
void sort(iterator debut, iterator fin)
```

qui utilise `operator<` des éléments contenus dans la partie du container indiquée par `debut` et `fin`
(les objets qui y sont stockés doivent donc posséder cet opérateur)

Exemple :

```
list<double> uneliste;
...
sort(uneliste.begin(), uneliste.end());
```







# Nombres Complexes

La bibliothèque `<complex>` définit les nombres complexes.

Ils se déclarent par `complex<double>`. Ils possèdent un constructeur à 2 arguments permettant de préciser les parties réelle et imaginaire, e.g.

```
complex<double> c(3.2,1.4), i(0,1);
```

Par contre, il n'existe pas de constructeur permettant de créer un nombre complexe à partir de ses coordonnées polaires.

En revanche, la fonction `polar`, qui prend comme paramètres la norme et l'argument du complexe à construire, permet de le faire. Cette fonction renvoie le nombre complexe nouvellement construit :

```
c = polar(sqrt(3.0), M_PI / 12.0);
```

Les méthodes des nombres complexes sont `real()` qui retourne la partie réelle, `imag()` qui retourne la partie imaginaire, et bien sûr les operateurs usuels.







# Nombres Complexes (2)

Ce qui est plus inattendu c'est que les opérations de norme, argument, et conjugaison n'ont pas été implémentées sous forme de méthodes, mais de **fonctions** :

| | |
|---|---|
| `double abs(const complex<double>&)` | retourne la **norme** (au sens français du terme) du nombre complexe |
| `double norm(const complex<double>&)` | retourne le **carré** de la norme |
| `double arg(const complex<double>&)` | retourne l'argument du nombre complexe |
| `complex<double> conj(const complex<double>&)` | retourne le complexe conjugué |

La bibliothèque fournit de plus les extensions des fonctions de base (trigonométriques, logarithmes, exponentielle) aux nombres complexes.







# Détails de `<cmath>`

Quelques fonctions définies dans la bibliothèque `<cmath>` :

| | |
|---|---|
| `abs` | valeur absolue |
| `acos` | arccos |
| `asin` | arcsin |
| `atan` | arctan |
| `ceil` | $\lceil x \rceil$, entier supérieur |
| `cos` | cos |
| `cosh` | cosinus hyperbolique |
| `exp` | exp |
| `floor` | $\lfloor x \rfloor$, entier inférieur |
| `log` | ln, logarithme népérien |
| `log10` | log, logarithme en base 10 |
| `pow(x,y)` | $x^y = \exp(y \ln x)$ — (préférez la multiplication pour les faibles puissances entières) |
| `sin` | sin |
| `sinh` | sinus hyperbolique |
| `sqrt` | $\sqrt{}$ |
| `tan` | tan |
| `tanh` | tangente hyperbolique |

# Détails de <cmath> (2)

Quelques constantes souvent fournies (mais non ISO) :

| | |
|---|---|
| $e$ | `M_E` |
| $\log_2(e)$ | `M_LOG2E` |
| $\log_{10}(e)$ | `M_LOG10E` |
| $\ln(2)$ | `M_LN2` |
| $\ln(10)$ | `M_LN10` |
| $\pi$ | `M_PI` |
| $\frac{\pi}{2}$ | `M_PI_2` |
| $\frac{\pi}{4}$ | `M_PI_4` |
| $\frac{1}{\pi}$ | `M_1_PI` |
| $\frac{2}{\pi}$ | `M_2_PI` |
| $\frac{2}{\sqrt{\pi}}$ | `M_2_SQRTPI` |
| $\sqrt{2}$ | `M_SQRT2` |
| $\frac{1}{\sqrt{2}}$ | `M_SQRT1_2` |







# Nombres aléatoires

La génération de nombres au hasard sur ordinateur se fait avec des *générateurs* dit « pseudo-aléatoires » qui pour une valeur initiale donnée (appelée « **graine** » [« seed » en anglais]) donnent toujours la même séquence « aléatoire » (suivant une *distribution* de probabilité choisie).

Utiliser la même graine peut être utile pour déverminer un programme utilisant des nombres aléatoires.

Pour avoir une série de nombres aléatoires différente à chaque utilisation du programme, il faut utiliser une graine différente à chaque fois.

[Même si ce n'est pas terrible,] Cela se fait souvent en utilisant comme graine la valeur de l'horloge de l'ordinateur à cet instant.

Une autre solution consiste à tirer la graine (voire la séquence elle-même) depuis un **périphérique** matériel suffisement **aléatoire** (« *random device* ») : (micro-)déplacement de la souris, température du processeur, ...







# Nombres aléatoires (2)

Avant C++11, le seul outil standard pour la génération de nombres aléatoires était `rand`, un générateur uniforme de nombres **entiers**. Des outils non standards venaient compléter la matériel à disposition.

`int rand()` tire un nombre entier entre 0 et `RAND_MAX` (2147483647 dans l'état actuel)

La graine de `rand()` se change avec

```
void srand(unsigned int graine)
```

Exemple d'initialisation avec l'horloge :

```
#include <ctime>
#include <cstdlib>
using namespace std;
...
srand(time(0));
```







# C++11 Nombres aléatoires (3)

Dans la bibliothèque `<random>` (C++11), il existe différents **générateurs** de nombres pseudo-aléatoires et différentes **distributions** de probabilités.

Les deux doivent être combinés pour pouvoir effectuer une série de tirage.

Ci-après un exemple simple pour tirer de façon uniforme un nombre aléatoire entier entre `min` et `max`.

# C++11 Nombres aléatoires : exemple

```cpp
#include <iostream>
#include <functional> // pour bind()
#include <random>
using namespace std;

int main()
{
  // par exemple (un d'e ?)
  int min(1);  int max(6);

  // distribution uniforme entre min et max
  uniform_int_distribution<int> distribution(min, max);

  random_device rd;          // pour la graine
  unsigned int graine(rd());

  // choix du generateur et initialisation (graine)
  default_random_engine generateur(graine);

  auto tirage(bind(distribution, generateur)); // esoterisme

  for (int i(0); i < 10; ++i) { // 10 tirages
    cout << tirage() << endl;
  }
  return 0;
}
```







# Ce que j'ai appris aujourd'hui

Qu'il existe beaucoup d'outils prédéfinis dans la bibliothèque standard de C++

Le but n'est évidemment pas les connaître tous par cœur, mais de **savoir qu'ils existent** pour penser aller chercher dans la documentation les informations complémentaires.

---

## La suite

- Révisions