Chapitre 9. La Librairie Standard

Généralités sur la STL

La standardisation permet la portabilité sur l'ensemble des plateformes matérielles et logicielles… et la librairie standard fournit en outre une aide au développement logiciel en C++.

Généralités

- La librairie standard contient :
  - la librairie C standard
  - la classe string et les flots d'entrées/sorties (support des jeux de caractères internationaux et de la localisation des applications)
  - la STL (Standard Template Library) qui contient un ensemble de conteneurs (liste, ensemble, …) et d'algorithmes
  - le support pour le calcul numérique (vecteurs et complexes)
- Fonctions de la librairie standard définies dans l'espace de nommage `std`, présentées comme un ensemble de fichiers d'en-tête (pas d'accès à l'implémentation)

Slide 2

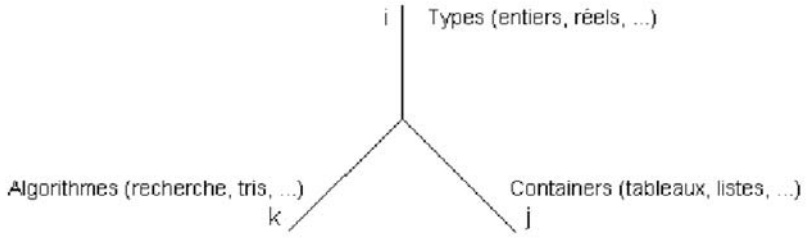
La STL est donc une sous-partie de la librairie standard de C++. Elle fournit, entre autres, des patrons de classes avec les méthodes adaptées pour les manipuler, mais aussi des algorithmes et des conteneurs.

La STL : description

- STL = Standard Template Library
- Fournit aux programmeurs un ensemble de structures de données classiques (liste, pile, …) et d'algorithmes
- Basée sur les patrons de classes
- Fournit des patrons de classes et les méthodes pour les manipuler (amélioration des performances)
- Liens :
  - guide du programmeur : http://www.sgi.com/tech/stl/
  - page oueb de B. Stroustrup : http://www2.research.att.com/~bs/

Slide 3

La STL : intérêt



- axe des i : types de données
- axe des j : structures de données
- axe des k : algorithmes
- Tous les cas possibles : i*j*k versions différentes de code pour un même algorithme.
- Avec la généricité, on peut supprimer l'axe des i ⇒ j*k versions seulement.
- Si les algorithmes fonctionnent avec toutes les structures de données ⇒ j+k versions. **C'est ce que fait la STL.**

Slide 4

La STL permet de proposer moins de versions différentes pour un même algorithme, indépendamment de la structure de données considérée et du type des données traitées.

La STL : composants

- Six composants au total
- Trois composants principaux :
  - les conteneurs (structures de données)
  - les algorithmes (traitements)
  - les itérateurs (manipulation des données)
- Trois autres composants :
  - les objets fonction (encapsulent une fonction dans un objet)
  - les adaptateurs (changent l'interface d'une classe)
  - Les allocateurs (gèrent le modèle mémoire de la machine)

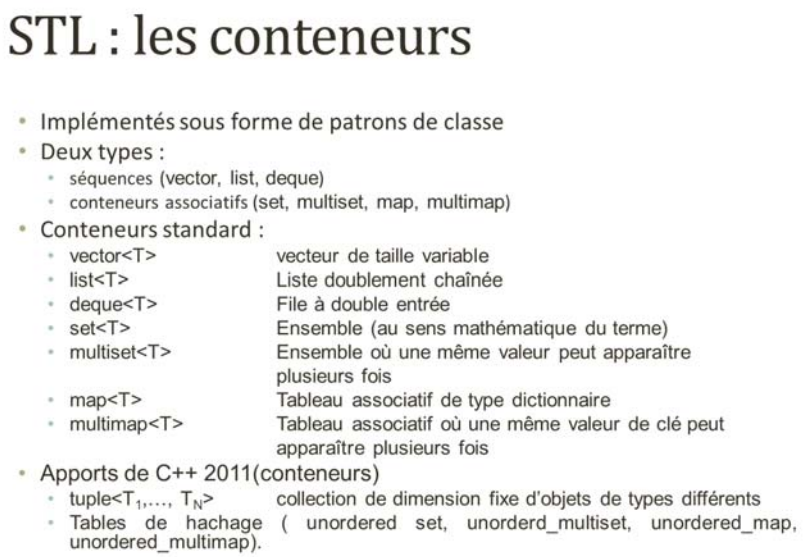
Slide 5

On va s'intéresser plus particulièrement aux conteneurs, itérateurs et algorithmes.
Les itérateurs permettent de parcourir les conteneurs et de manipuler les données qui y sont stockées.
NB : un algorithme prévu pour un itérateur donné peut s'appliquer à tous les conteneurs supportant ce type d'itérateur.

Les conteneurs, quant-à eux, sont des modèles de structures de données présentés sous forme de patrons de classe (donc génériques), instanciables, entre autres, par le type de données stockées.

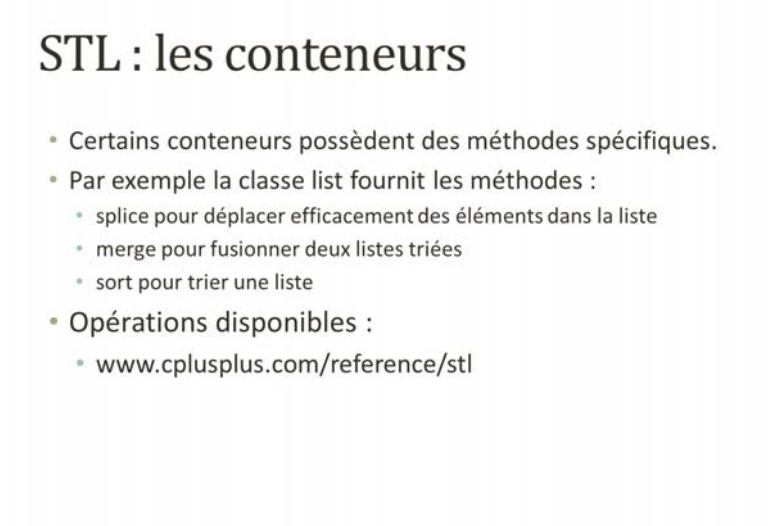
Les conteneurs de la STL

Les conteneurs sont des classes dont l'objectif principal est de stocker des objets.



Slide 6

La classe Vector se rapproche d'un type tableau, plus flexible et avec des méthodes spécifiques.
La classe List fonctionne comme la structure de données Liste vue en cours d'algo l'an dernier.
La classe Deque se comporte comme la structure de données File vue en cours d'algo l'an dernier.
Les maps ou multimaps sont des tableaux associatifs faisant appel à la notion de clé.
Un objet de type map représente un tableau associatif de type dictionnaire, *i.e.* un ensemble d'associations clé/valeur.



Slide 7

Exemple d'utilisation de la classe Vector de la STL

STL : exemple d'utilisation de la classe vector

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

int main ( )
{
  vector<int> v ;
  for (int i=0 ; i<100 ; i++) v.push_back(i) ;
  cout<<v[10]<<endl; //affiche 10 à l'écran
  cout << v.at(10) << endl; //affiche 10 à l'écran
  return 0;
}
```

Slide 8

vector<int> v ; est l'instanciation de la classe vector par un type entier : v est un tableau d'entiers.
v.push_back(i); ajoute i en fin de séquence (push_back ajoute les éléments à la fin du vecteur)
cout<<v[10]<<endl; on voit ici que l'opérateur [] a été défini pour le type vector, permettant d'accéder directement à un élément quelconque du vecteur. De même, l'opérateur << est défini pour Vector et pour le type qu'il contient.
cout<<v.at(10)<<endl; at est l'accesseur de v et joue le même rôle que l'opérateur []. L'opérateur [] a été surchargé pour permettre une utilisation plus intuitive de l'objet Vector.

Exemple d'utilisation de la classe map

STL : exemple d'utilisation de la classe map

```
#include <iostream>
#include <string>
#include <map>

using namespace std ;

int main( )
{
  map< string, string, less<string> > m ;
  //   less   est   l'opérateur   de   comparaison   pour
  //   le type de la clé ( de type string ici)

  m["France"] = "Paris";
  m["Espagne"]="Madrid";
  cout<<m["France"]<<endl; //Paris
  return 0 ;
}
```

Slide 9

map< string, string, less<string> > m ;
Le premier string correspond à la clé, le second au type de données T, et la partie **less<string>** correspond à l'opérateur de comparaison pour la clé(cf. l'en-tête du type map).
Donner l'opérateur de comparaison garantit qu'on comparera les clés entre elles de façon pertinente (permet le tri de la map ou de se déplacer dans une map triée pour y trouver un élément, par exemple).
Là encore, l'opérateur << est défini pour le type map et considéré comme défini pour le type contenu dans la Map.

Autre exemple pour la classe map

STL : exemple d'utilisation de la classe map

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

int main( ){
  map <string, int> mousquetaires;

  mousquetaires["Athos"  ] = 1;
  mousquetaires["Porthos"] = 2;

  pair<string, int> p = make_pair("Artagnan", 4);
  mousquetaires.insert(p);
  mousquetaires.erase("Athos"); // suppression
  if (mousquetaires.count("Aramis") ==0 )
              mousquetaires["Aramis" ] = 3;
  return 0;
}
```

Slide 10

Le type map a des méthodes associées (par exemple count, ici, qui permet de savoir combien d'instances sont associées à mousquetaires["Aramis"]) et il peut en outre être initialisé avec des types compatibles avec ceux qui ont servi à l'instancier.

Les itérateurs de la STL

STL : les itérateurs, définition (1/2)

- Permettent de rendre les algorithmes indépendants des conteneurs
- Généralisation de la notion de pointeur (* pour déréférencer, ++ pour passer au suivant)
- Accès uniforme aux conteneurs de la STL, aux flots d'E/S et aux tableaux C
- Chaque conteneur fournit deux méthodes qui retournent des itérateurs
  - `begin( )` : itérateur sur le premier élément
  - `end()` : itérateur après le dernier élément

Slide 11

L'utilisation d'itérateurs pour les conteneurs de la STL permet d'appliquer les mêmes opérations quel que soit le conteneur utilisé, ce qui permet d'avoir une bonne homogénéité des programmes et que les changements de conteneurs soient transparents.

Attention : on ne doit pas déréférencer un itérateur qui pointe après le dernier élément (celui renvoyé par end()), puisqu'il « pointe » *après* le dernier élément.

STL : les itérateurs, définition (2/2)

- 5 catégories d'itérateurs :
  - Entrée et Sortie (non détaillés)
  - Accès aléatoire
  - Bidirectionnel
  - Monodirectionnel
- Fichier d'en-tête : **`<iterator>`**
- Nom de l'itérateur associé à un conteneur :
  `nom_conteneur::iterator`
  ou **`nom_conteneur::const_iterator`**

Slide 12

Liste des différents itérateurs :
input_iterator <T,Distance> / input_iterator : itérateur d'entrée
output_iterator : itérateur de sortie
forward_iterator : itérateur monodirectionnel (ou itérateur "en avant")
bidirectional_iterator : itérateur bidirectionnel
random_access_iterator : itérateur à accès aléatoire

STL : itérateurs monodirectionnels

- Parcours unidirectionnel d'un conteneur
- Déplacement séquentiel
- Un seul opérateur dispo : ++
- Utilisé par les algorithmes qui ont besoin de faire plusieurs passes.

```
list<int> l;
l.push_back(10); l.push_back(45); l.push_back(12);
l.push_back(15);
for (list<int>::const_iterator it = l.begin();
          it != l.end(); ++it)
    {
          cout << *it << endl;
    }
```

Slide 13

Les itérateurs monodirectionnels sont aussi appelés des itérateurs "en avant" ; seul l'opérateur ++ (avancer de 1) est défini sur ces itérateurs : le déplacement est séquentiel à l'intérieur du conteneur.
Dans l'exemple proposé, la déclaration et l'initialisation de l'itérateur se fait dans la commande du for (l'itérateur est la variable it) :
Déclaration : `list<int>::const_iterator it`
`Initialisation : it = l.begin();`

NB : l'opérateur de comparaison = et != ont été définis pour l'itérateur en avant.
`*it` déréférence l'itérateur it et permet d'accéder à la valeur qu'il « pointe » dans la liste l.

STL : itérateurs bidirectionnels

- Cf. monodirectionnel, mais aussi…
- L'opérateur -- est défini et permet de revenir en arrière

```
list<int>::const_iterator it_end = l.begin();
--it_end;
for (--it; it != it_end; --it) {
    cout << *it << endl;
}
```

Slide 14

Ici, les opérateurs ++ et -- sont définis, ce qui permet de se déplacer séquentiellement (donc d'un seul élément à chaque fois) dans le conteneur, mais dans les deux sens.

STL : itérateurs à accès aléatoire

- supportent toute une gamme d'opérateurs (+,+=,…)

```
template<class IT> void bubblesort(IT debut, IT fin) {
  bool stop = false;
  while (!stop)
  {    stop = true;
       for (IT it = debut; it != fin - 1; ++it)
       {if (*it > *(it + 1))
              {swap(*it, *(it + 1));
               stop = false; }/* fin if*/  } /* fin for */
  }// while
}

vector<int> v;
v.push_back(12); v.push_back(69); v.push_back(5);
  v.push_back(1);
bubblesort(v.begin(), v.end());
for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
{ cout << *it << endl;}
```

Slide 15

Dans le cas des itérateurs à accès aléatoire, on peut parcourir le conteneur de façon séquentielle en avant ou en arrière, accéder directement à un élément donné, et se déplacer de façon "aléatoire" (i.e. choisir le "pas" dont on se déplace) dans le conteneur.

L'exemple proposé ici est en fait une fonction de tri prenant en paramètre des itérateurs. Elle est donc générique et indépendante de la structure de données. Il s'agit ici d'un tri à bulle.

La méthode définie ici trie les éléments situés entre deux itérateurs (donc sans se préoccuper du type du conteneur). Elle utilise des méthodes définies pour les conteneurs et s'appuyant sur les itérateurs (swap, par exemple)
Il s'agit d'une méthode générique, instanciée par le type des itérateurs qui seront passés en entrée.

NB : la ligne `*it > *(it + 1)` suppose que l'opérateur > est défini pour le type contenu dans le conteneur manipulé et « pointé » par l'itérateur. Idem pour la méthode swap utilisée à la ligne suivante

STL : les itérateurs, résumé

catégories	Monodirectionnel	Bidirectionnel	A accès aléatoire
Ecriture	*p=	*p=	*p=
Itération	++	++, --	++, --, +, -, +=, -=
Comparaison	==, !=	==, !=	==, !=, <, >, >=, <=

Slide 16

Auto-typage/typage automatique et conteneurs

À partir de la norme 2011 : auto-typage et parcours de conteneurs

- Mot-clé **auto**
  - Prend la place du type dans la déclaration
  - Le type d'une variable est décidé par correspondance avec le type retourné de l'objet utilisé pour l'initialisation
- Si l'on couple avec les itérateurs, on simplifie les parcours de conteneurs

```
template class<T> void f1 (vector<T> &arg)
    {
       for (vector<T> ::iterator p = arg.begin();p!=arg.end();p++)
              *p = 7
       for (auto p = arg.begin();p!=arg.end();p++)
              *p = 7
       }
```

- Boucles basées sur les intervalles
  - apparition d'un opérateur de type « foreach » pour simplifier la syntaxe du for

```
int mon tableau[5] = {1,2,3,4,5};
for (int &x : mon_tableau){
  x*=2;
}
```

Slide 17

Le mot-clé auto change de sémantique avec C++ 2011.
Exemple :
int a1 = 123;
char a2 = 123;

auto a3 = 123; // ici, a3 sera de type entier (type par défaut)
On peut du coup simplifier l'utilisation des itérateur grâce à ce typage automatique pour parcourir les conteneurs de façon plus robuste et plus lisible !

Par ailleurs, on peut utiliser une syntaxe simplifiant les parcours de type boucle for.
Dans l'exemple, on multiplie par deux le contenu de chaque case du tableau.
L'entier x est défini pour le corps de la boucle for et référence successivement chacun des éléments du tableau.
Cette syntaxe est utilisable avec : les listes classiques, mais aussi sur les conteneurs de la STL définissant les fonctions membres begin() et end()

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

int main() {
   vector<int> nombres = { 1, 2, 3, 4, 5 };
   for (int &element : nombres) {
      cout << element << endl;
   }
   return 0;
}
```

Dans la STL, sont mis à disposition non seulement des itérateurs, mais également des algorithmes s'appuyant sur les itérateurs pour manipuler les données des conteneurs sans connaître le type de ces derniers.

Les algorithmes de la STL

STL : les algorithmes

- Traitements effectués sur les conteneurs
- Algorithmes génériques paramétrés par un type d'itérateur : indépendants des conteneurs
- Implémentés sous forme de patrons de fonctions (dans `<algorithm>`)
- Instanciables également avec des pointeurs de type C
- 4 groupes d'algorithmes (cf. doc STL) :
  - traitements ne modifiant pas la séquence des éléments (`count`, `for_each`,...)
  - traitements modifiant la séquence des éléments (`copy`, `reverse`,...)
  - tris (`sort`, `merge`,...)
  - traitements numériques (`accumulate`,...)

Slide 18

Les algorithmes proposés dans la STL sont 100% génériques, puisqu'ils prennent en paramètres des types d'itérateurs : ils sont donc indépendants de la structure de données contenant les informations à traiter.
Les algorithmes de la STL peuvent également être instanciés avec des pointeurs de type C car ils ont les mêmes fonctionnalités que les itérateurs à accès aléatoire (on parle d'arithmétique des pointeurs) : on peut donc utiliser les algorithmes de la STL sur les tableaux.
Les différents algorithmes disponibles sont fournis à part.
NB :

- Les algorithmes portant le suffixe _copy créent une nouvelle séquence,
- Les algorithmes portant le suffixe _if s'utilisent avec un prédicat

STL : les algorithmes, exemple

```
#include <iostream>
#include <algorithm>
#include <iterator>

int main( )
{
  int T[]={1,2,5,9,7,3,5,4,7,8};
  std::sort(&T[0], &T[10]);
  //la fin est positionnée un élément après le dernier.
  std::copy(&T[0], &T[10],
  std::ostream_iterator<int>(std::cout, ","))
  // affichage : 1,2,3,…
  return 0;
}
```

Slide 19

Dans cet exemple, on utilise l'algorithme sort de la STL pour trier les éléments d'un tableau. On passe les points de début et de fin du tableau à l'algorithme comme s'il s'agissait d'itérateurs.

La ligne suivante :

```
     std::copy(&T[0], &T[10],std::ostream_iterator<int>(std::cout, ",",))
```

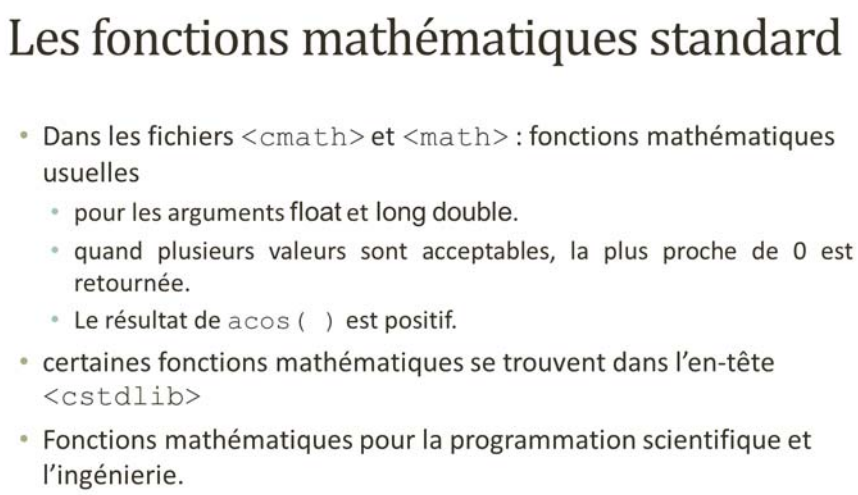
copie le conteneur délimité par **`&T[0] et &T[10]`** vers un itérateur de sortie référençant le flot de sortie standard stdout, en le formatant avec une virgule comme caractère de séparation.

Autres fonctionnalités de la librairie standard : maths et calcul numérique

Le calcul numérique

- Dans la librairie standard : classes permettant de réaliser des opérations mathématiques plus évoluées que l'arithmétique de base
- Limites des types :
  - patron `numeric_limits` (dans `<limits>`) précise pour chaque type un certain nombre d'informations (minimum, maximum, ...).
  - chaque implémentation de la bibliothèque standard fournit une spécialisation de `numeric_limits` pour chacun des types fondamentaux

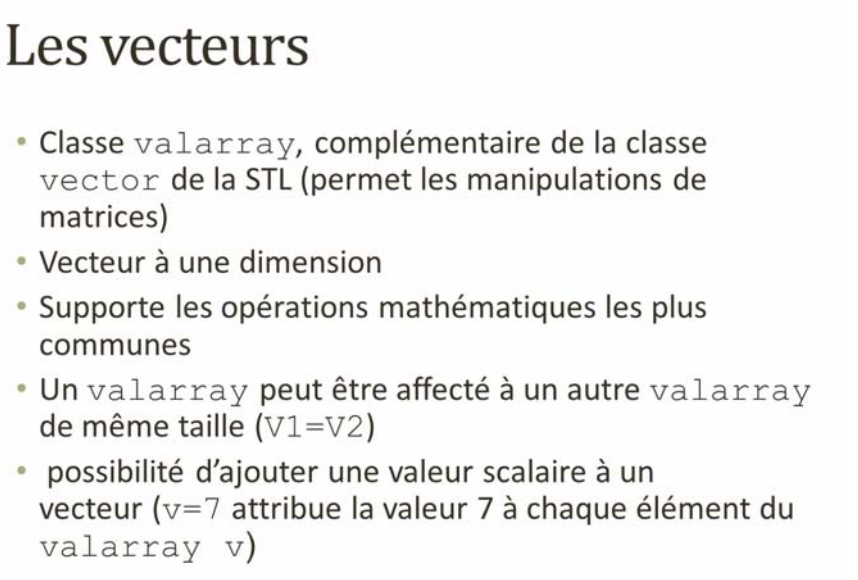
Slide 20



Slide 21

Le cas particulier des vecteurs

Lorsqu'on veut manipuler des vecteurs au sens mathématique du terme, ce n'est pas la classe Vector qu'on utilise, mais une classe spécialement définie pour des manipulations mathématiques.



Slide 22

Exemple d'utilisation des vecteurs

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

int main( )
{
  std::valarray <double> v1(2), v2(2);
  v1[0] = 1.2; v1[1] = 2.3;
  v2[0] = 5.1; v2[1] = 9.1;
  // on peut faire des opérations de masse :
  std::valarray<double> v = v1*3.0+v2/v1;
  std::cout << v[0] << ", " << v1[1] << std::endl;
  return 0;
}
```

Slide 23

Lors de la déclaration, le vecteur est instancié par le type de données contenu, et on passe sa taille au constructeur.
On voit dans l'exemple que l'on peut faire des opérations de masse sur les données (additions, multiplications, affectations, …) : **`std::valarray<double> v = v1*3.0+v2/v1;`**
Remarque : l'opérateur << a été surchargé pour la classe valarray.

NB : les objets de type valarray sont destinés à faire des opérations de masse sur les données, tandis que les objets de type vector sont plutôt destinés à stocker des objets en en facilitant l'accès.

De même, il existe une classe spécifique pour la représentation et la manipulation des matrices…

Les matrices

- Manipulation de matrices de dimensions quelconques en utilisant les classes `slice`, `slice_array` ou `gslice`.
  - `slice` : abstraction permettant de manipuler efficacement un vecteur sous la forme d'une matrice de dimension arbitraire. Représente tout n^{ième} élément d'une partie d'un `valarray`
  - `slice_array` : permet de faire référence aux sous-ensembles du tableau décrit par un `slice`.
  - `gslice` : « slice généralisé » contenant presque toutes les informations contenues dans n `slices`.

slide 24

Rmq : on peut accéder à un sous-ensemble d'un valarray en utilisant la classe mask_array

Les nombres complexes

- Patron `complex` (dans `<complex>`) instanciable avec des types flottants avec :
  - Fonctions unaires et binaires habituelles (+, -, *, /, = =, et !=)
  - fonctions de coordonnées
  - fonctions mathématiques
  - flots d'entrées/sorties (voir aide de C++ ou Stroustrup)

```
#include <iostream>
#include <complex>
int main( ) {
  std::complex<double> dc1(1.2, 2.3), dc2(5.6, 8.4), r;
  r = dc1 + std::sinh(dc2);
  std::cout << r <<std::endl;
  return 0;
}
```

Slide 25

Les nombres aléatoires

- Base pour la génération de nombres aléatoires dans : <cstdlib> et <stdlib.h>
- Possibilité de choisir la distribution de probabilité (loi normale, Bernoulli, Poissson, etc)

Exemple :

```
#define RAND_MAX implementation_defined
  /*entier positif très grand*/

int rand( ) ;  // nombre pseudo-aléatoire entre 0 et RAND_MAX
int srand(int i) ;
  // alimente le générateur de nombres aléatoires par i
  // débute une nouvelle séquence de nombres aléatoires
  // à partir de i (utile pour le débogage).
```

- `( double(rand( )) / RAND_MAX )*n` : bons résultats pour un nombre aléatoire entre 0 et n-1.
- Pour une valeur réellement aléatoire avec **`srand`** utiliser l'horloge.

slide 26

La STL permet aussi la génération de nombres aléatoires, à partir de la librairie C stdlib, entre autres…
Mais il y a également des fonctionnalités plus avancées permettant l'usage d'outils statistiques. Il s'agit d'une fonctionnalité découpée en deux parties :

- Un moteur de génération contenant l'état du générateur qui produit des nombres pseudo-aléatoires : au moins 3 algorithmes de génération sont proposés
- Une distribution qui détermine les valeurs que le résultat peut prendre, ainsi que sa loi de probabilité (normale, bernoulli, poisson, etc…)