

# STL

Xavier JUVIGNY

ONERA

23 Aout 2017

# Plan du cours

- 1 Présentation STL
- 2 Les conteneurs
- 3 Utilitaires système
- 4 algorithm
- 5 Parallélisme à mémoire partagée

# Standard Template Library (STL)

## Historique

- **1971** : Première recherche en programmation générique par Dave Musser
- **1979** : Alexander Stepanov publie ses premières idées de programmation générique;
- **1987** : A. Stepanov met en œuvre ses idées en ADA;
- **1992** : Sur demande d'HP, Stepanov avec Musser porte le code ADA en C++ avec l'aide de Meng Lee;
- **1993** : l'ISO demande à HP et Stepanov la possibilité de normaliser leur bibliothèque;
- **Mars 1994** : Première version normalisée de la STL pour le C++;
- **Août 1994** : HP met leur implémentation de la STL libre de droit.

## Caractéristiques

- Bibliothèque uniquement basée sur les templates, libre de droit;
- Les différentes implémentations de la STL sont relativement bien optimisées;
- Temps relativement long à la compilation, mais techniques possibles pour optimiser le temps de compilation;
- Divisée en plusieurs parties :
  - Les conteneurs;
  - Général;
  - Localisation;
  - Chaînes de caractères;
  - Flux, entrées et sorties;
  - Support pour le langage;
  - Support du multithreading;
  - Librairies numériques;
- À partir de C++ 17, certains algorithmes de la STL ont une version parallèle.

# Les itérateurs

## Notion d'itérateur

- Objets permettant de parcourir les éléments d'un conteneur, indépendamment de la structure sous-jacente;
- On pourra ainsi parcourir de la même façon un tableau, une liste, un arbre, un dictionnaire, etc.
- Différents itérateurs en C++ :

- Itérateurs bidirectionnel : permet d'accéder à l'élément suivant ou précédent :

```
std::list<int>::iterator itL = ... ; itL++ /* élt suivant */; itL-- /* élt précédent */;
```

- Itérateurs à accès aléatoire : Permet de pouvoir sauter autant d'éléments ( précédents ou suivant ) que l'on souhaite :

```
std::vector<int>::iterator itV = ...; itV += 5; /* Cinq éléments après */
```

- Itérateurs inverses : Permet de parcourir un conteneur à l'envers

```
std::vector<int>::reverse_iterator itV = ...; itV ++; /* élt précédent */
```

- Itérateurs constants : Permet de parcourir les éléments d'un conteneur sans pouvoir modifier les valeurs contenues.

```
std::vector<int>::const_iterator itV = ...; double a = *itV; *itV = 3 /* Erreur ! */;
```

- Un itérateur peut avoir plusieurs des qualificatifs cités au dessus.

```
std::vector<int>::const_reverse_iterator itV = ...; /* It. inverse constant à accès aléatoire*/
```

# Les itérateurs (suite)

## Utilisation des itérateurs

- Les itérateurs comparés avec l'opérateur !=, et l'opérateur \* permet l'accès à l'élément courant pointé par l'itérateur;
- Les conteneurs de la STL ont tous une méthode
  - `begin` renvoyant un itérateur pointant sur le premier élément du conteneur;
  - `end` renvoyant un itérateur pointeur sur un élément après le dernier élément du conteneur.

```
std::vector<int> tableau = { 2, 3, 5, 7, 11, 13, 17, 19 };  
for ( auto it = tableau.begin(); it != tableau.end(); ++it ) std::cout << *it << std::endl;
```

- Permet aussi certaines manipulations sur les conteneurs : insertion, suppression, etc.

```
std::vector<std::string> tab;  
tab.push_back(" et "); tab.push_back(" milou.");  
tab.insert(tab.begin(), "Tintin");// Insère Tintin au début du tableau  
tab.erase(tab.begin());// On supprime le premier mot...
```

## Fonctions `begin` et `end`

- Deux fonctions `begin` et `end` renvoyant un itérateur sur le premier ou après le dernier élément du conteneur...

```
int foo[] = {1,2,3,4,5}; std::vector<int> bar;  
for ( auto it = std::begin(foo); it != std::end(foo); ++it ) ...  
for ( auto it = std::begin(bar); it != std::end(bar); ++it ) ...
```

# Méthodes communes aux conteneurs

## Capacité

- `empty` teste si le conteneur est vide ou non;
- `size` retourne la taille actuelle du conteneur;
- `max_size` retourne la taille maximale possible pour le conteneur.

## Accesseurs

- La méthode `front` permet d'accéder au premier élément du conteneur;
- La méthode `back` permet d'accéder au dernier élément du conteneur;

Et seulement pour les conteneurs à accès direct :

- L'opérateur `[]` permet d'accéder directement au  $i^{\text{ème}}$  élément;
- La méthode `at` permet d'accéder directement au  $i^{\text{ème}}$  élément mais vérifie la validité de l'indice;
- La méthode `data` renvoie un pointeur C sur l'adresse du premier élément du conteneur.

## Modificateurs pour conteneurs dynamiques

- `insert` : Insère un élément avant la position de l'itérateur passé en argument
- `erase` : Enlève l'élément pointé par un itérateur ou les éléments encadrés par deux itérateurs `[first,last[`;
- `swap` : Échange les éléments de deux conteneurs
- `clear` : Enlève tous les éléments du conteneur;
- `emplace` : Construit un nouveau élément avant l'itérateur

# std::array

## Caractéristiques

- Conteneur statique à accès direct;
- Défini un tableau de taille fixe contiguë en mémoire;
- Fichier inclu : `#include <array>`
- Création tableau statique : `std::array<T,N>` où T type contenu et N nombre d'éléments;
- Possibilité d'avoir tableau taille zéro mais éviter accès aux éléments;
- itérateurs à accès aléatoires; itérateur inverse possible
- `fill` : méthode permettant de remplir le tableau avec une valeur unique
- Comparaisons de deux tableaux éléments par éléments, du premier vers le dernier

```
std::array<int,10> tab = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 27 };
decltype(tab) tab2= tab;
for ( auto rit = tab.cbegin(); rit != tab.crend(); ++rit ) std::cout << ' ' << *rit;
std::cout << std::endl;
tab[9] = 29; // Ah oui, 27 n'est pas premier, c'est 29 qui l'est !
for ( const auto& val : tab ) std::cout << ' ' << val; std::cout << std::endl;
std::cout << "Nbre éléments du tableau : " << tab.size() << " == " << tab.max_size() << std::endl;
std::cout << "1er elt : " << tab.front() << ", last elt : " << tab.back() << std::endl;
try { tab.at(10) = 31; } catch(std::out_of_range& err)
{ std::cerr << "Pfff, pas de onzième élément." << std::endl; tab.fill(0); }
std::get<4>(tab) = -11; // adresse de tab[4] calculée à la compilation.
if (false==tab.empty()) { if (tab == tab2) std::cerr << "No, it can't be !" << std::endl; }
assert(tab != tab2);
```

# std::vector

## Caractéristiques

- Conteneur dynamique à accès direct;
- Création tableau dynamique avec ou sans initialisation;
- itérateurs à accès aléatoires, itérateur inverse possible,
- Possibilité réservation mémoire supérieure à taille courante tableau;
- Gestion allocation mémoire très optimisée,
- Comparaison lexicographique, `std::fill`, ...

```
#include <vector>
#include <iostream>
#include <algorithm>
int main() {
    std::vector<std::size_t> itab(20), itab2; itab2.reserve(20);
    for ( int i = 0; i < 20; ++i ) itab[i] = ((1<<i)+1);
    for ( int i = 0; i < 20; ++i ) {
        bool is_mul = false;
        for ( auto v : itab2 ) if ( itab[i] % v == 0 ) { is_mul = true; break; }
        if ( is_mul == false ) itab2.push_back(itab[i]);
    }
    itab.swap(itab2);
    std::cout<<"nb nombre fermat : "<<itab.size()<< " sur " <<itab.capacity()<<std::endl;
    std::for_each(itab.begin(), itab.end(), [](std::size_t i){std::cout << i << ' ';});
    std::cout << std::endl;
    itab.shrink_to_fit(); std::cout<<"nvelle capacite : " <<itab.capacity()<<std::endl;
    return 0;
}
```



# std::list

## Caractéristiques

- Conteneur dynamique à accès séquentiel;
- Liste à double liens
- Création avec liste d'initialisation ou non;
- Itérateur uniquement séquentiel, possibilité reverse;
- Insertion au milieu de la liste efficace;
- Existe une version `std::forward_list` à lien unique, avec pas d'itérateur inverse.

```
# include <list>
int main()
{
    const int n = 10;
    std::list<std::vector<int>> pascal{ {std::vector<int>{1,1} } };
    for ( int i = 1; i <= n; ++i ) {
        const std::vector<int>& curdev = pascal.back();
        std::vector<int> devel(i+2, 1);
        for ( int j = 0; j < i; ++j ) devel[j+1] = curdev[j]+curdev[j+1];
        pascal.push_back(devel);
    }
    for ( const auto& pol : pascal ) {
        int deg = pol.size()-1;
        std::cout << "Dév. de (x+y)^" << pol.size()-1 << " = " << pol[0] << ".x^" << deg << " + ";
        for ( int d=1; d<deg; ++d ) std::cout << pol[d] << ".x^" << (deg-d) << ".y^" << d << " + ";
        std::cout << pol[deg] << ".y^" << deg << std::endl;
    }
}
```

# Ensembles

## std::set : Ensemble à valeurs uniques

- Conteneur dynamique associatif garantissant exemplaire unique de chaque objet;
- Possibilité définir opération comparaison permettant de trier éléments;
- Itérateur séquentiel avec possibilité itérateur inverse.

```
#include <set>
struct compObj {
    bool operator () ( int i, int j ) const {
        if ( i<j ) std::swap(i,j);
        int remain = i%j; int val = j;
        while ( remain != 0 ) { int tmp = remain; remain = val % remain; val = tmp; }
        if ( val != 1 ) return false; else return true;
    }
};
int main()
{
    std::set<int,compObj> numberprimes;
    for ( int i = 2; i < 1000; ++i ) numberprimes.insert(i);
    for ( int v : numberprimes ) std::cout << v << " ";
    std::cout << std::endl;
}
```

## std::multiset : Ensemble à valeurs multiples

- Mêmes fonctionnalités que `std::set` mais ne garantit pas unicité;
- Une version non ordonnée existe également.

# Autres conteneurs

## Conteneur de pile

- Permet de gérer une pile LIFO ( Last in, First out );
- Pas d'itérateurs, mais accès au dernier élément empilé;
- On ne peut qu'empiler ou dépiler.

```
# include <stack>
int main() {
    std::stack<std::string> pile;
    pile.emplace("Milou\n"); pile.emplace("et"); pile.emplace("Tintin");
    while (pile.empty() == false) { std::cout << pile.top() << " "; pile.pop(); }
```

## Conteneur queue et priority\_queue

- queue : Gère une queue FIFO ( First in First out );
- priority\_queue : Gère une queue dont le premier élément est le plus grand élément de l'ensemble.

```
# include <queue>
int main() {
    std::default_random_engine gen;
    std::uniform_int_distribution<int> dist(1,6); auto de=std::bind(dist,gen);
    std::priority_queue<int> lances; for ( int i=0; i<10; ++i ) lances.push(de());
    while (lances.empty() == false) { std::cout << lances.top() << " "; lances.pop(); }
    std::cout << std::endl;
```

# Mesure du temps

## Présentation générale

- Un seul header : `chrono`;
- Tous les éléments de ce header appartiennent à l'espace de nommage `std::chrono`;
- Repose sur trois concepts :
  - **Durée** : Mesure temps écoulé. Représenté par la classe template `duration` ayant pour paramètre `type pour compter` et `période de précision`. Exemple : `10 millisecondes`
  - **Localisation temporelle** : Une localisation précise dans le temps exprimé par la classe template `time_point` qui contient une durée relative à une époque ( qui est un lieu du temps commun à tous les objets utilisant la même horloge );
  - **Les horloges** : Permet de relier une localisation temporelle à un temps physique réel. Trois types d'horloges proposés : `system_clock`, `steady_clock` et `high_resolution_clock`.

## Les différentes horloges

- `system_clock` : Horloge système temps réel permettant de le convertir en représentation calendrier
- `steady_clock` : Horloge permettant de mesurer des intervalles de temps
- `high_resolution_clock` : Horloge possédant la période d'horloge la plus petite possible.

# Horloge système

## Propriétés

- Représente le temps réel;
- Permet de faire de l'arithmétique signée;

```
# include <chrono>
int main() {
    using std::chrono::system_clock;
    auto one_day = std::chrono::hours(24);
    system_clock::time_point today = system_clock::now();
    system_clock::time_point tomorrow = today + one_day;
    system_clock::time_point yesterday = today - one_day;
    std::time_t tt;
    tt = system_clock::to_time_t ( today );
    std::cout << "today is: " << ctime(&tt);
    tt = system_clock::to_time_t ( tomorrow );
    std::cout << "tomorrow will be: " << ctime(&tt);
    tt = system_clock::to_time_t ( yesterday );
    std::cout << "Yesterday was: " << ctime(&tt);
}
```

# Autres horloges

## steady\_clock

- Permet de calcul un temps écoulé entre deux appels;

```
using namespace std::chrono;
steady_clock::time_point beg = steady_clock::now();
auto approx_pi = approximate_pi(1000000);
steady_clock::time_point end = steady_clock::now();
auto time_span = duration_cast<duration<double>>(end-beg);
std::cout << "pi approx = " << approx_pi << std::endl;
std::cout << "Temps pris calcul pi : " << time_span.count() << std::endl;
```

## Horloge haute précision

- Horloge avec la plus grande précision
- Souvent un synonyme de `steady_clock`

# Tirages pseudo aléatoires

## Présentation

- Dans `random`, permet de produire des nombres aléatoires en utilisant une combinaison de générateurs et de distributions
- **Générateurs** : Objets qui génèrent des nombres distribués uniformément;
- **Distributions** : Objets qui transforment des suites de nombres générés par un générateur en suite de nombres qui suivent une loi de distribution, tel que `uniform`, `Normal` ou `Binomial`.
- Les objets de distribution génère des nombres aléatoires à l'aide de l'opérateur `()` qui prend un générateur en argument

```
std::default_random_engine generator;  
std::uniform_int_distribution<int> distribution(1,6);  
int dice_roll = distribution(generator);
```

- On peut utiliser `std::bind` pour lier la distribution au générateur :

```
auto dice = std::bind( distribution, generator );  
int wisdom = dice() + dice() + dice();
```

- Tous les générateurs ormi `random_device` utilisent une graine comme origine de l'aléatoire,

# Les générateurs

## Générateur pseudo-aléatoire (classes templates)

- `linear_congruential_engine`
- `mersenne_twister_engine`
- `subtract_with_carry_engine`

## Adaptateurs ( modifie le comportement du générateur )

- `discard_block_engine` : utilise  $r$  nombre sur  $p$  nombres;
- `independent_bits_engine` : génère qu'avec un nombre de bits spécifiques;
- `shuffle_order_engine` : stocke  $r$  éléments de la suite et prend une valeurs au hasard.

## Générateur non déterministe

`random_device`

## Instantiations de générateurs

- `default_random_engine` : Générateur aléatoire par défaut
- `minstd_rand` :  $x = x * 48271 \% 2147483647$
- `minstd_rand0` :  $x = x * 16807 \% 2147483647$
- `mt19937` : Mersenne Twister avec état sur 19937 bits
- `mt19937_64` : idem en 64 bits
- `ranlux24_base` : `subtract_with_carry_engine` sur 24 bits
- `ranlux48_base` : `subtract_with_carry_engine` sur 48 bits
- `ranlux24` : idem que `ranlux24_base` avec adaptateur `discard_block_engine`
- `ranlux48` : idem que `ranlux48_base` avec adaptateur `discard_block_engine`
- `knuth_b` : `random_device` + `shuffle_order_engine`



# Distributions aléatoires

## Distribution uniforme

- `uniform_int_distribution` : distribution sur un ensemble dénombrable
- `uniform_real_distribution` : distribution sur un ensemble réel

## Bernoulli

- `bernoulli_distribution`
- `binomial_distribution`
- `geometric_distribution`
- `negative_binomial_distribution`

## Basé sur des proportions

- `poisson_distribution`
- `exponential_distribution`
- `gamma_distribution`
- `weibull_distribution`
- `extreme_value_distribution`

## Loi normale

- `normal_distribution`
- `lognormal_distribution`
- `chi_squared_distribution`
- `cauchy_distribution`
- `fisher_f_distribution`
- `student_t_distribution`

## Par morceau

- `discrete_distribution`
- `piecewise_constant_distribution`
- `piecewise_linear_distribution`

# Exercices

## Nuage de points “distants”

- Générer un nuage de points 2D tel que la distance minimum entre deux points soit de un;
- Créer un nouveau conteneur permettant de parcourir ces points dans l'ordre croissant des abscisses.
- Pour créer un point aléatoirement, utiliser la bibliothèque `random` et

`chrono` :

```
typedef std::chrono::high_resolution_clock myclock;  
myclock::duration d = myclock::now().time_since_epoch();  
unsigned seed = d.count();  
std::default_random_engine generator(seed);  
std::uniform_real_distribution<double> distribution(-100.0,100.0);  
double x1 = distribution(generator);  
double y1 = distribution(generator);  
...  
double xn = distribution(generator);  
double yn = distribution(generator);
```

# Gestion de la mémoire

## Pointeur unique

- Permet de créer un objet dynamiquement pointé par un pointeur unique;
- Ce pointeur peut changer par déplacement ( pas de copie possible );
- Objet pointé détruit automatiquement quand plus de pointeur dessus.
- Création de l'objet pointé à l'aide de `std::make_unique` (C++14)
- Peut pointer sur le pointeur nul ( `nullptr` )
- Respecte les méthodes virtuelles.

```
std::unique_ptr<SymmetricMatrix> f(std::unique_ptr<SquareMatrix> M) { ... }  
...  
auto sq_mat = std::make_unique<SquareMatrix>(3, { 1., 3., 5.,  
                                                  2., 4., 0.,  
                                                  3., 5., 1. });  
  
auto mat = std::move(sq_mat); // Déplacement, mat pointe sur la matrice, sq_mat sur nullptr  
auto sym_mat = f(mat); // après appel, mat = nullptr et la SquareMatrix est détruite  
                        // et sym_mat pointe sur la matrice symétrique retournée  
int dim = sym_mat->dimension(); // Accès comme pour un pointeur classique  
double a = (*sym_mat)(0,0); // Avec l'opérateur () virtuel -> ok appel de la bonne méthode  
std::unique_ptr<SquareMatrix> sq_matsym(std::move(sym_mat)); // Convertit en pointeur sur matrice  
carrée
```

# Gestion de la mémoire

## Pointeur partagé

- Permet de créer un objet pointé par un ou plusieurs pointeurs;
- L'objet est détruit quand plus aucun pointeur pointe dessus;
- `use_count` permet de savoir le nombre de pointeur
- Création du pointeur et de l'objet pointé à l'aide de `std::make_shared` (C++11 et au delà)
- Peut pointer sur l'objet null ( `nullptr` );
- Plus lourd de gestion pour l'exécutable que `unique_ptr`;
- Conversions entre types possible à l'aide de `static_pointer_cast`, `dynamic_pointer_cast`,...;
- Respecte les méthodes virtuelles.

```
std::shared_ptr<SymmetricMatrix> f(std::shared_ptr<SquareMatrix> M) { ... }  
...  
auto sq_mat = std::make_shared<SquareMatrix>(3, { 1., 3., 5.,  
                                                  2., 4., 0.,  
                                                  3., 5., 1. });  
  
auto mat = sq_mat; // Copie, mat et sq_mat pointent sur la matrice  
std::cout << mat.use_count() << std::endl; // Affiche 2  
auto sym_mat = f(mat); // après appel, mat pointe sur la SquareMatrix  
                    // et sym_mat pointe sur la matrice symétrique retournée  
int dim = sym_mat->dimension(); // Accès comme pour un pointeur classique  
double a = (*sym_mat)(0,0); // Avec l'opérateur () virtuel -> ok appel de la bonne méthode  
std::shared_ptr<SquareMatrix> sq_matsym = std::static_pointer_cast<SquareMatrix>(sym_mat); //  
    Convertit en pointeur sur matrice carrée
```

# Expressions régulières

## Définition

- Moyen standard d'exprimer des motifs de comparaison pour les chaînes de caractères;
- Différent de la syntaxe shell
- Très difficile à lire : `"\s*(?P<header>[^\s]*)\s*:(?P<value>.*?)\s*"`
- Très puissant

## Syntaxe de base

- Caractère hors caractères spéciaux : représente lui-même;
- Caractère quelconque symbolisé par le point .;
- Répétitions caractère : symbole \* (0-N), + (1-N), ? : (0-1)

## Exemples

reg	ok	pas ok
a.a	aaa aba a a	aa a baa
a+	a aaaa	aaabaa
ab?a+	aba abaaa aaaa	acaaa abba
a*b.c	ab c bxc aaaaaaabdc	abc bc b

# Expressions régulières

## Syntaxe ( suite )

- Le début `^`
- La fin `$`
- Les ensembles : encadré par `[]`

## Exemples

reg	ok	pas ok
<code>[abc]+</code>	a abac cab	za +
<code>[0-9], [0-9]+</code>	0,1 2,345	0 23,4
<code>[A-Z]_[^ =()]+</code>	A_Data X_42	AB_Data A
<code>^[^#]+#.*\$</code>	A = 4 # Commentaire shell function() # Fonction shell	A=4 # Commentaire ligne

# Expressions régulières en C++

## Présentation

- Trouvé dans l'entête `regex`
- Les types de paramètres employés sont :
  - **La suite ciblée** : la suite de caractères dans laquelle on recherche le motif
  - **l'expression régulière** : Le motif qu'on recherche dans la suite ciblée
  - **Tableaux de correspondance** : Plusieurs opérations permettent d'avoir des informations sur les correspondances. Ces informations sont stockées dans des tableaux de type `match_results`;
  - **Chaîne de remplacement** : Plusieurs opérations permettent de remplacer des motifs.

## Opérations possibles

- `regex_match` : Recherche si la suite ciblée correspond au motif;
- `regex_search` : Recherche si un sous-ensemble de la suite ciblée correspond au motif;
- `regex_replace` : Remplace tous les sous-ensemble de la suite ciblée correspondant au motif par une autre suite de caractères.

```
std::string s ("this subject has a submarine as a subsequence");
std::smatch m;
std::regex e ("\\b(sub)([~]*)"); // matches words beginning by "sub"
std::cout << "The following matches and submatches were found:" << std::endl;
while (std::regex_search (s,m,e)) {
    for (auto x:m) std::cout << x << " ";
    std::cout << std::endl; s = m.suffix().str(); }
std::cout << std::regex_replace (s,e,"sub-$2");
std::string result;
std::regex_replace (std::back_inserter(result), s.begin(), s.end(), e, "$2");
std::cout << result;
```

# Algorithmes

## Présentation

- C++ propose un ensemble d'algorithmes pouvant traiter un ensemble de valeurs dans `algorithm`
- Ces algorithmes peuvent se classer en plusieurs catégories :
  - Les opérations sur les suites qui ne les modifient pas;
  - Celles qui les modifient;
  - Les algorithmes de partitionnement;
  - Les algorithmes de tri;
  - Les algorithmes de recherche;
  - Les algorithmes ensemblistes;
  - Les algorithmes de tas;
  - Les algorithmes de min/max;
  - Des algorithmes non classifiables...
- Ces algorithmes opèrent sur les valeurs de la séquence, via les itérateurs;
- Ils n'affectent en rien la structure même d'un conteneur, bien qu'ils puissent y permuter des éléments.



# Algorithmes ne modifiant pas les séquences

## Algorithmes de parcours

Une seule fonction : `for_each` qui permet d'appliquer une fonction passée en paramètre aux valeurs de la séquence.

```
std::vector<double> u{1.,3.,5.,7.,11.,13.};  
std::for_each(u.begin(),u.end(), [](double x) { std::cout << x << ' '; });
```

## Algorithmes de test

- Test sur les éléments d'une séquence : `all_of`, `any_of`, `none_of`
- Test de comparaison entre deux séquences : `mismatch`, `equal`, `is_permutation`

```
std::vector<int> iu{3,5,7,9,11};  
if (std::all_of(iu.begin(),iu.end(),[](int i) {return (i%2!=0);} )) std::cout<<"Tous impairs,";  
if (std::any_of(iu.begin(),iu.end(),[](int i) {return (i%3==0);} )) std::cout<<"un div par 3,";  
if (std::none_of(iu.begin(),iu.end(),[](int i){return (i%13==0);} )) std::cout<<"aucun div par 13";  
std::cout << std::endl;  
int iv[] = {3,5,9,11,7};  
auto diff = std::mismatch(iu.begin(),iu.end(), iv);  
if (diff.first != iu.end())  
    std::cout << "Premier elt diff : " << *diff.first << " != " << *diff.second << std::endl;  
if (std::is_permutation(iu.begin(),iu.end(),iv)) std::cout << "iv permut. de iu !" << std::endl;
```

# Algorithmes modifiant les séquences

## Copies, déplacements et suppressions

- Algorithmes de copie : `copy`, `copy_n`, `copy_if` et `copy_backward`
- Algorithmes de déplacement : `move`, `move_backward`
- Algorithme d'échange : `swap`, `swap_ranges`, `iter_swap`
- Algorithme de remplacement : `replace`, `replace_if`, `replace_copy`, `replace_copy_if`
- Algorithme de suppression : `remove`, `remove_if`, `remove_copy`, `remove_copy_if`
- Algorithme unicité : `unique`, `unique_copy`

## Algorithme de modifications

- Transformation : `transform`
- Remplissage : `fill`, `fill_n`
- Génération : `generate`, `generate_n`
- Inversion : `reverse`, `reverse_copy`
- Décalage : `rotate`, `rotate_copy`
- Mélange : `random_shuffle`, `shuffle`

# Autres algorithmes

## Partitionnement

- `is_partitioned`, `partition`, `stable_partition`, `partition_copy`, `partition_point`

## Tri

- `sort`, `stable_sort`, `partial_sort`, `partial_sort_copy`, `is_sorted`, `is_sorted_until`, `nth_element`

## Recherche binaire

- `lower_bound`, `upper_bound`, `equal_range`, `binary_search`

## Opérations ensemblistes

- `merge`, `inplace_merge`, `includes`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`

## Tas (heapsort)

- `push_heap`, `pop_heap`, `make_heap`, `sort_heap`, `is_heap`, `is_heap_until`

## Min/Max

- `min`, `max`, `minmax` : pour deux variables ou une liste d'initialisation
- `min_element`, `max_element`, `minmax_element` : pour un conteneur

# Modèle multithreading

## Multithreading en C++

- Permet une gestion multithread d'un programme indépendamment du système d'exploitation;
- Accès à des utilitaires bas niveau pour une performance optimale des threads ( atomic, etc. )
- Programmation simplifiée par rapport aux threads posix;
- Peut appeler des objets avec la méthode d'évaluation ();
- Objet `std::thread` est un objet uniquement déplaçable, pas copiable;
- `std::thread::hardware_concurrency()` peut donner le nombre de thread optimal à lancer pour une machine donnée;
- Possibilité de programmer des fonctions asynchrones.

## Exemple basic de programmation multithread

```
#include <iostream>
#include <thread>

int main()
{
    std::thread t([&std::cout] () { std::cout << "Hello World!" << std::endl; });
    std::cout << "Hello for main program ;-)" << std::endl;
    t.join(); // Attends que le thread est fini de s'exécuter
    return EXIT_SUCCESS;
}
```

# Passage d'arguments

## Modalité de passage par argument

```
void comp_boundary_condition( int boundary_id, Field& fld ) { ... } /* (1) */  
void wrong_code(Boundary& bnd) {  
    Field fld;  
    std::thread t(update_boundary_condition, bnd.id(), fld ); /* (2) */  
    comp_internal_nodes(...);  
    t.join();  
    update_boundary(bnd, fld); /*(3)*/  
}
```

- La fonction en (1) attend une référence en deuxième paramètre;
- Le constructeur du thread en (2) ne peut pas le savoir et copie `fld` aveuglement;
- On utilisera donc le mauvais `fld` lors de la mise à jour en (3);
- Il faut donc préciser qu'on veut passer une référence au thread :

```
void comp_boundary_condition( int boundary_id, Field& fld ) { ... } /* (1) */  
void right_code(Boundary& bnd) {  
    Field fld;  
    std::thread t(update_boundary_condition, bnd.id(), std::ref(fld) ); /* (2) */  
    comp_internal_nodes(...);  
    t.join();  
    update_boundary(bnd, fld); /*(3)*/  
}
```

# Appel méthode d'objet dans un thread et thread encapsulé

## Appel méthode objet dans une thread

- Même mécanisme que `std::bind` ( qui a aussi même problème pour les références... );
- On doit passer la méthode, un pointeur sur l'objet puis les arguments de la méthode.

```
class X { ...  
    void method_of_x(int i, double x) { .... }  
};  
X objX; ...          std::thread(&X::method_of_X, &objX, 2, 3.5);
```

## Thread encapsulé

- Possibilité d'encapsuler un thread dans une classe

```
class scoped_thread {  
    std::thread t;  
public:  
    explicit scoped_thread(std::thread t_) : t(std::move(t_))  
    { if(!t.joinable()) throw std::logic_error("No thread"); }  
    ~scoped_thread() { t.join(); }  
    scoped_thread(scoped_thread const&)=delete;  
    scoped_thread& operator=(scoped_thread const&)=delete; };  
    void f() { ...  
        scoped_thread t(std::thread(func(...))); // func = "function"  
        do_something_in_current_thread(); }  
};
```

# Gestion de plusieurs threads

## Gérer et synchroniser plusieurs threads

- `std::thread` est utilisable avec les conteneurs;

```
auto f = [] (int i){std::cout <<"Hello from thread "<< i << std::endl;};
const int nbThreads = std::thread::hardware_concurrency();
std::vector<std::thread> threads;
threads.reserve(nbThreads);
for ( int i = 0; i < nbThreads; ++i )
    threads.emplace_back(f,i);
std::cout << "Hello for main program ;-)" << std::endl;
for ( auto& t : threads ) t.join();
```

## Identification des threads

- Le type de l'identifiant est `std::thread::id`;
- Par défaut, il s'initialise avec une valeur "not any thread"
- Objet hashable pour être utilisé avec les conteneurs non triés associatifs...

```
std::thread::id master_id;
if ( std::this_thread::get_id() == master_id ) ...
```

# Gestion des conflits mémoires

## mutex

- Comme pour les posix threads, on peut gérer les conflits mémoires à l'aide de mutex;
- Plusieurs classes utilitaires en facilite l'emploi

```
class Heap {...  
    void push( const K& obj ) {  
        std::lock_guard<std::mutex> guard(m_serial_mutex); ...  
    }  
    K pop() {  
        std::lock_guard<std::mutex> guard(m_serial_mutex); ...  
    }  
private:  
    std::mutex m_serial_mutex; ...  
};
```

- Possibilité de locker plusieurs mutex à la fois :

```
std::mutex rhs_m, lhs_m; ...  
std::lock(rhs_m, lhs_m);  
std::lock_guard<std::mutex> lock_a(lhs_m, std::adopt_lock);  
std::lock_guard<std::mutex> lock_b(rhs_m, std::adopt_lock);
```

- `std::unique_lock` permet quant à lui de transférer un mutex d'un scope à un autre et de relâcher ou remettre le lock à volonté



# Autres possibilités de synchronisation

## Appeler une fonction qu'une fois

- `std::call_once` permet de n'appeler une fonction qu'une fois parmi les threads;

```
std::shared_ptr<SomeObj> ptr_obj;  
std::once_flag rsc_flg;  
...  
std::call_once(rsc_flg, init_ptr_obj, ptr_obj);
```

## Mutex récursif

- Mutex pouvant être locker plusieurs fois par le même thread sans blocage;
- Utile pour des fonctions récursives ou pour un chaînage d'appels de méthodes d'une classe;
- `std::recursive_mutex` permet d'utiliser ce mutex récursif;
- Préférable de l'utiliser conjointement avec `std::lock_guard` ou `std::unique_lock`;
- À utiliser le plus rarement possible, car en général, devoir l'utiliser montre une mauvaise conception du logiciel...
- Par exemple, dans le cas d'un appel chaîné de méthodes de classe, meilleurs de définir une fonction privée appelée par les deux méthodes et qui ne fait pas de lock...

# Variables de conditions

## Synchronisation avec une variable de condition

- Permet à un thread d'attendre qu'une condition soit vérifiée tout en ne bloquant pas de mutex;
- Thread en état de sommeil en attendant, besoin de notifier le thread pour le "réveiller";
- Deux types de condition en C++ : `std::condition_variable` et `std::condition_variable_any` ( plus général et peut utiliser des classes se comportant comme des mutex );

```
std::mutex mut; std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
void prepare_data() {
    while (more_data_to_prepare()) {
        data_chunk data = prepare_datum();
        std::lock_guard<std::mutex> lk(mut); data_queue.push(data);
        data_cond.notify_one(); // notify_one pour 1 thread en attente, notify_all pour tous en attentes
    }
}
void data_process() {
    while(true) {
        // std::unique_lock nécessaire pour que thread relâche le lock quand en sommeil
        std::unique_lock<std::mutex> lk(mut);
        // Retourne si cond vérifiée, sinon en sommeil jusqu'à notification, et reteste...
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        lk.unlock();
        ...
    }
}
```

# Queue multithreadable

## Queue dans contexte multithreadée

- Mettre au point une queue utilisable dans un contexte multithreadé;
- Rajouter une méthode `try_pop` qui tente de dépiler une donnée et renvoie une erreur si la queue est vide;
- Rajouter une méthode `wait_and_pop` qui attend que la queue ne soit plus vide pour dépiler une donnée.

# Tâches asynchrones

## Retour de valeurs de tâches en arrière plan

- 1 `std::thread` ne fournit pas de mécanisme pour retourner la valeur d'une fonction;
- 2 `std::async` permet de commencer une tâche asynchrone et retourne la valeur dans un `std::future`
- 3 Passage d'arguments identique au passage arguments pour `std::thread`;
- 4 La méthode `get` de `std::future` attend jusqu'à ce que la valeur de retour soit prête

```
# include <future>
# include <iostream>
double very_long_computation();
void other_funny_computation();
int main() {
    std::future<double> answer = std::async(very_long_computation);
    other_funny_computation();
    std::cout << "Result of long computation : " << answer.get() << std::endl;
    return EXIT_SUCCESS;
}
```

- 5 Possibilité de modifier comportement de `std::async` en rajoutant un paramètre en premier argument :
  - 6 `std::launch::async` ( par défaut ) pour lancer la fonction dans un nouveau thread;
  - 7 `std::launch::deferred` pour exécuter la tâche que lorsqu'on fait appel à la méthode `get` ou `wait` de la classe `std::future`.

# Autres utilitaires pour fonctions asynchrones

## Associer une tâche avec un futur

- `std::packaged_task<signature fonction>` permet d'associer une fonction avec un `std::future`
- Le `std::future` est construit à partir de la signature de la fonction;
- On peut l'invoquer à l'aide de la méthode `get_future()` de `std::packaged_task`.
- Un `std::packaged_task` peut être wrappé dans un `std::function` ou passé à un thread.

## Stocker une valeur lue plus tard par un `std::future`

- `std::promise` permet de stocker un résultat récupérable dans un `std::future`;
- Un thread peut y stocker une valeur qui sera récupérée par un autre thread à l'aide d'un `get` du `std::future` récupéré à l'aide de la méthode `get_future` de `std::promise`

## Lire la valeur d'un future par plusieurs threads

- `std::future` ne permet qu'à un seul thread de lire la valeur retournée par une fonction asynchrone ( `std::future` n'est pas copiable );
- `std::shared_future` permet à plusieurs threads de lire cette valeur;
- Néanmoins, il faudra synchroniser l'accès à la lecture ou passer une copie du `std::shared_future` aux autres threads.

# Opérations atomiques

## Modèle C++ des atomiques

- `std::atomic<T>` permet de synchroniser des variables de façon atomique;
- Propose des méthodes `load`, `store` et `exchange`;
- Opérations addition, soustraction, et, ou et ou exclusif permis;
- Méthode `is_lock_free` permet de tester si notre variable atomique est vraiment libre de verrous !
- Seul, `std::atomic_flag` est garantit sans verrous, mais en général, `std::atomic<integral>` sans verrous aussi ( mais pas imposé par la norme... );
- `std::atomic_flag` très basique. Opérations permises : le détruire, le mettre à faux, le mettre à vrai en demandant sa valeur précédente

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;
void f(int n) {
    for(int cnt = 0; cnt < 100; ++cnt) {
        while(lock.test_and_set(std::memory_order_acquire)) // acquire lock
            ; // spin
        std::cout << "Output from thread " << n << '\n';
        lock.clear(std::memory_order_release); // release lock
    }
}
int main() {
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) { v.emplace_back(f, n); }
    for (auto& t : v) { t.join(); }
}
```

# En conclusion

## Librairie standard

- Bibliothèque très riche en fonctionnalités;
- Beaucoup d'autres fonctionnalités omises dans cette présentation :
  - les complexes;
  - La bibliothèque mathématiques ( fonctions et `valarray` );
  - les dictionnaires non triés avec hachage;
  - etc.

## Maîtrise du C++

- Un débutant connaît 10% du langage;
- Un développeur avec un peu d'expérience : 40%
- Un expert dans le langage : 70%
- Bjarne Stroustrup, le créateur du langage prétend en connaître 75% !