RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

C++ Moderne
Librarie STL

Xavier JUVIGNY, SN2A, DAAA, ONERA
xavier.juvigny@onera.fr

Formation C++
- 8 Septembre 2022 -

[1] ONERA, [2] DAAA

Plan du cours

Sommaire

Quelques fonctions utilitaires

Dans la librairie `utility` :

- `T exchange( T& obj, U&& newval)` : Remplace la valeur contenue par la variable `obj` par la nouvelle valeur `newval` et renvoie l'ancienne valeur de `obj` ; (C++ 14)
- `cmp_equal, cmp_not_equal, cmp_less, ...` : Compare deux entiers avec garantie sans problèmes avec non signés (C++ 20) ;
- `template<R,T> in_range(T t)` : Renvoie vrai si la conversion de `t` en type `R` se fait sans perte de données (exemple quand entier long vers entier court) (C++ 20) ;
- `template<F,Tuple> decltype(auto) apply(F&& f,Tuple&& t)` : Appel la fonction `f` avec pour argument les valeurs dans le tuple `t` (C++ 17) ;
- `template<T,Tuple> T make_from_tuple( Tuple&& t )` : Construit un objet de type `T` en utilisant les valeurs dans le tuple `t` (C++ 17) ;

Localisation des sources (C++ 20)

Utilitaire dans <source_location> permettant de localiser l'endroit où on se trouve dans le code. Ne marche qu'avec g++ version 11 ou supérieure (MSVC ?).

Exemple

```
int une_fonction() {
    std::source_location location = std::source_location::current();
    std::cout << "Je suis dans la fonction " << location.function_name() << std::endl;
    std::cout << "A la ligne : " << std::source_location::current().line() << std::endl;
    return int(std::source_location::current().column());
}

int main() {
    std::source_location location = std::source_location::current();
    std::cout << "Je suis dans le fichier " << location.file_name() << std::endl;
    int col = une_fonction();
    std::cout << "Retour à la colonne " << col << std::endl;
```

Variable optionnelle (C++ 17)

Librairie `optional`

Utilité

Variable contenant ou non une valeur de type *T*. Passer arguments optionnels ou retour d'une valeur optionnelle. Remplace l'utilisation du pointeur.

Exemple

```
template<std::integral I> std::pair<I,std::optional<I>> divmod( I const& a, I const& b ) {
    std::pair<I, std::optional<I>> resultat;
    resultat.first = a/b;
    I residu   = a - resultat.first*b;
    if (residu != 0) resultat.second = residu;
    return resultat;
}

    auto res1 = divmod(7,3);
    std::cout << "7/3 = " << res1.first;
    if (res1.second.has_value()) std::cout << "[" << res1.second.value() <<"]";
```

Variant (C++ 17)

Principe

Remplace l'union du C avec les avantages suivants :

- Support complet du cycle de vie des types complexes : si on change de type, le destructeur est appelé ;
- Service pour connaître le type actuel employé.

Exemple

▸ code de l'exemple

Code exemple sur variant

```
std::variant<double, int, std::string> unionVariable;
static_assert(std::variant_size_v<decltype(unionVariable)> == 3);

unionVariable = 3.14;
std::visit([](auto var) { std::cout << var; }, unionVariable);
std::cout << "\nIndex du type utilisé : " << unionVariable.index() << std::endl;

unionVariable = "Tintin"s;
std::visit([](auto var) { std::cout << var; }, unionVariable);
std::cout << "\nIndex du type utilisé : " << unionVariable.index() << std::endl;

if (const auto intPtr (std::get_if<int>(&unionVariable)); intPtr)
    std::cout << "int : " << *intPtr << std::endl;

if (std::holds_alternative<std::string>(unionVariable)) std::cout << "Contient un std::string." << std::endl;

try {
    auto f = std::get<double>(unionVariable);
    std::cout << "J'ai bien un réel : " << f << std::endl;
} catch(std::bad_variant_access&)
{   std::cout << "Impossible d'afficher un réel." << std::endl;    }
```

Retour page variant

Objet de type faible (C++ 17)

Principe

Le type d'un objet mute selon le type de la valeur affectée.

Exemple

```
std::any a_variable; a_variable = 3;
std::cout<<demangle(a_variable.type().name())<<" : "<<std::any_cast<int>(a_variable)<<std::endl;
a_variable = std::vector<double>{2.,3.,5.,7.,11.};
std::cout << demangle(a_variable.type().name()) << " : "
          << std::any_cast<std::vector<double>>(a_variable)[0] << std::endl;
std::cout << std::boolalpha << "a_variable contient un vecteur de double ? "
          << (typeid(std::vector<double>) == a_variable.type()) << std::endl;
try { a_variable = "tintin"s; std::cout << std::any_cast<float>(a_variable) << std::endl; }
catch(const std::bad_any_cast& e) { std::cout << e.what() << std::endl; }
a_variable.reset();
if (a_variable.has_value())
    std::cout<<demangle(a_variable.type().name())<<" : "<<std::any_cast<int>(a_variable)<<std::endl;
std::cout << "type de a = void ? " << (a_variable.type() == typeid(void)) << std::endl;
```

Gestion des fichiers et répertoires (C++ 17)

Principe

- Gérer des fichiers et des répertoires en étant portable entre systèmes d'exploitation ;
- Utilise la bibliothèque `filesystem` ;
- Définir un `path` générique (posix) ou dépendant du système, un répertoire (et sa nature : symbolique, normal, etc.) ;
- etc.

Gestion des paths

Path générique ou natif au choix :

```
namespace fs = std::filesystem;
fs::path p1="/usr/lib"; // Générique
fs::path p2="C:\\TMP\\"; // Natif
std::cout << std::boolalpha << "p1 contient un fichier ? " << p1.has_filename() << std::endl;
```

Création/suppression de répertoire et parcours de répertoire

Exemple

```
const fs::path sandbox{"sandbox"};
fs::create_directories(sandbox/"dir1"/"dir2");
std::ofstream{sandbox/"file1.txt"};
std::ofstream{sandbox/"file2.txt"};

std::cout << "directory_iterator:\n";
// Itérateur sur un répertoire :
for (auto const& dir_entry : fs::directory_iterator{sandbox})
{    std::cout << dir_entry.path() << '\n';    }

std::cout << "\nrecursive_directory_iterator:\n";
for (auto const& dir_entry : std::filesystem::recursive_directory_iterator{sandbox})
{   std::cout << dir_entry << '\n';   }
// Supprime le répertoire sandbox ainsi que tout ses fichiers et sous-répertoires :
fs::remove_all(sandbox);
```

Sommaire

Généré son propre pointeur partagé (memory)

Principe

Permet à un objet géré par un pointeur partagé de générer lui-même des pointeurs partagés sur lui-même

Exemple

```
struct Node : public std::enable_shared_from_this<Node> {
    virtual ~Node() = default;
    virtual std::ostream& display(std::ostream& out) const = 0;
    std::shared_ptr<Node> getptr() { return shared_from_this(); }
};

class Addition : public Node  {
public:
    static std::shared_ptr<Node> create() { return std::shared_ptr<Addition>(new Addition()); }
    virtual ~Addition() = default;
    std::ostream& display(std::ostream& out) const override
    {   out << "+"; return out;   }
private:
    Addition() = default;     };
```

Alignement mémoire

Principe

Pouvoir interroger le système sur l'alignement mémoire de certains types et pouvoir aligner les données d'une structure.

- `alignof(type)` : Retourne l'alignement mémoire en octets d'un type. Pour une structure, retourne l'alignement max des attributs ;
- `alignas(octets)` : Demande à ce qu'une structure/classe s'aligne sur un certain nombre d'octets

Exemple

```
struct alignas(alignof(long double)) BigPoint
{ double x,y,z;  }; // alignof(BigPoint) == 16 octets
```

Construire ou détruire à une adresse donnée

Aligner la mémoire

```
std::array<std::byte, 4096> bufferMemory;
void* memoireAlignee = bufferMemory.data(); std::size_t espace_restant=4096;
Point* poolPoints = (Point*)std::align(128,bufferMemory.size()-128, memoireAlignee, espace_restant);
```

Construire à une adresse donnée

```
auto vecteur = std::construct_at(poolPoints, 3.4, 1.5, 2.6, 3.14);
```

Détruire à une adresse donnée

```
std::destroy_at(vecteur);
```

Remarque : Si on veut détruire plusieurs objets créés dans une zone mémoire débutant à une adresse donnée, on peut utiliser `std::destroy` ou `std::destroy_n`.

Sommaire

Introspection

Permet au code de s'inspecter pour changer le comportement du code selon le résultat de l'introspection. Inclure `type_traits`.

Exemple

```
template<typename CArray, int dim, std::size_t... dims>
auto get_dimensions( std::integer_sequence<std::size_t,dims...> ) {
    if constexpr (dim > 0) {
        return get_dimensions<CArray, dim-1>( std::integer_sequence<std::size_t,
                                              std::extent_v<CArray,dim>, dims...>{} );
    } else { return std::integer_sequence<std::size_t, std::extent_v<CArray,0>, dims...>{}; }
}
template<typename CArray> void displayCArray( CArray const& t_array ) {
    if constexpr ( std::is_bounded_array_v<CArray> ) {
        constexpr auto ndims = std::rank_v<CArray>;
        auto dimensions = get_dimensions<CArray, ndims-1>(std::integer_sequence<std::size_t>{});
        std::cout << "Tableau"; displayDimensions(dimensions);
    } else { std::cout << "Tableau à dimension indéfinie" << std::endl;  }
}
```

Template pour compilation conditionnelle

Principe

- `std::enable_if<bool,T>` permet de restreindre l'utilisation des paramètres templates ;
- Indispensable en C++ 11, a perdu de l'intérêt avec C++ 17 et surtout avec les concepts du C++ 20 ;
- Mais encore utile dans certains cas !
- Un peu délicat à utiliser ! Se base sur le principe du **SFINAE**.

Exemple simple

```
template<typename T, std::enable_if_t<std::is_integral<T>::value,bool> = true >
void incrementCounter( T& t_counter )
{   ++t_counter;    }
```

Compilation conditionnelle (suite)

Exemple

```
template<typename T, int ndims, std::enable_if_t<(ndims>0),int> = 0>
class NdArray {
public:
    NdArray( std::array<std::size_t, ndims> const& t_ndims )
        :   m_dimensions(t_ndims)    {
        std::size_t nbCoefs = 1;
        for ( std::size_t indDim = 0; indDim<ndims; ++indDim )
            nbCoefs *= t_ndims[indDim];
        std::vector<T>(nbCoefs).swap(m_coefficients);
    }
    template<std::size_t indice0, std::size_t... indicesn,
             std::enable_if_t<(sizeof...(indicesn)<ndims),bool> =true>
    constexpr std::size_t globalIndex(std::size_t index = 0)    {
        std::size_t globIndex = indice0*m_dimensions[index];
        if constexpr (sizeof...(indicesn) > 0) globIndex += globalIndex<indicesn...>(index+1);
        return globIndex;
    }
```

▸ suite du code

Compilation conditionnelle (suite 2)

Exemple(suite)

```
    template<std::size_t... indices,
             std::enable_if_t<(sizeof...(indices)==ndims),bool> =true>
    T const& get() const {   return m_coefficients[globalIndex<indices...>()];    }

    template<std::size_t... indices,
             std::enable_if_t<(sizeof...(indices)==ndims),bool> =true>
    T& get() {   return m_coefficients[globalIndex<indices...>()];    }

private:
    std::vector<T> m_coefficients{};
    std::array<std::size_t,ndims> m_dimensions{};
};
```

Retour début du code

Sommaire

Fonctions disponibles

- Opérations arithmétiques : `plus,minus,multiplies,divides,modulus,negate`

```
std::vector valeurs{2.,3.,5.,7.,11.,13.};
```

- Opérations de comparaison : `equal_to,not_equal_to,greater,less,greater_equal,less_equal`

```
std::copy(valeurs.begin(), valeurs.end(), std::ostream_iterator<double>(std::cout, ";")); std::cout << std
```

- Opérations logiques : `logical_and,logical_or,logical_not`

```
std::transform(valeurs.begin(), valeurs.end(), std::back_inserter(flags), [](double x){ return (int(x)%2==
```

- Arithmétique binaire : `bit_and,bit_or,bit_xor,bit_not`
- Opérations réduction logique : `std::all_of,std::any_of,std::none_of`

Curryfication de fonctions

Définition

Procédé fonctionnel consistant à fixer un ou plusieurs paramètres d'une fonction pour définir une nouvelle fonction

- `std::bind` permet de curryfier ;
- `std::placeholders::_1,...std::placeholders::_N` : Permet d'associer les paramètres de la nouvelle fonction avec les paramètres de l'ancienne fonction.

Exemples

```
double normeEuclidienne( double x, double y, double z ) {   return std::sqrt(x*x+y*y+z*z);   }
auto norme2D = std::bind(normeEuclidienne,_1, _2, 0.);
std::cout << "||(3,4)|| = " << norme2D(3.,4.) << std::endl;
auto invminus = std::bind(std::minus<double>(), _2, _1);
std::cout << "invminus(3,4) = " << invminus(3,4) << std::endl;
```

Curryfication d'une méthode

Principe

- Curryfier une méthode appliquée à un objet ;
- Passer un pointeur sur méthode et en 1er argument un pointeur sur l'objet sur lequel on applique la méthode.

Exemple

```
auto popFunc = std::bind(&std::list<double>::pop_front, &pList);
popFunc();
```

Cas où la méthode est surchargée

Exemple

```
using pushbackType = void (std::list<double>::*)(const double&);
auto pushFunc = std::bind(static_cast<pushbackType>(&std::list<double>::push_back), &pList, _1);
pushFunc(11.);
```

Sommaire

Nouveaux conteneurs

Variant conteneurs STL

- `std::forward_list` : Liste simplement chaînée
- `std::unordered_map` et `std::unordered_multimap` : Dictionnaires basés sur fonctions hashage.
- `std::unordered_set` et `std::unordered_multiset` : Ensembles basés sur fonctions hashage.

Vue modifiable sur une zone mémoire (C++ 20)

Vue sur une zone mémoire contigu

- Moyen léger d'itérer ou d'indexer des objets stockés en mémoire contigu ;
- Plus sûr que d'utiliser un pointeur ou des indices ;
- La taille de la zone mémoire peut être dynamique ou statique.
- **Syntaxe** : `std::span<T,Extent=dynamic_extent>`
- `dynamic_extent` signifie que la taille de la zone sera spécifiée à l'exécution
- Sinon `Extent` sera un entier donnant le nombre d'éléments vus par le span.
- Il est prévu pour C++ 23 de définir un `mdspan` permettant une vue multidimensionnelle.

Exemple d'utilisation de span

Exemple simple

```
template<typename T, std::size_t N>
void displayMemoryZone( std::span<T,N> const& zone ) {
    for ( auto const& value: zone )
        std::cout << value << " ";
    std::cout << std::endl;
}

int main() {
    double tableauC[] = {2.,3.,5.,7., 11.};
    std::vector tableauV{2.,3.,5.,7.,11.};

    double* cppDynamicTab = new double[5];
    cppDynamicTab[0] = 2.; cppDynamicTab[1] = 3.; cppDynamicTab[2] = 5.;
    cppDynamicTab[3] = 7.; cppDynamicTab[4] = 11.;
    std::span<double> zoneCpp(cppDynamicTab, 5);
    auto subZone = zoneCpp.subspan(1, 3);

    displayMemoryZone( std::span{tableauC} );
    displayMemoryZone( std::span{tableauV} );
    displayMemoryZone( zoneCpp );
    displayMemoryZone( subZone );
```

Les plages (C++ 20)

Présentation

La bibliothèque `range` proposée par le C++ 20 est une extension et une généralisation des algorithmes et itérateurs en les rendant plus puissants et moins sujets aux erreurs.
On peut créer et manipuler des vues, c'est à dire des objets peu coûteux qui représentent des séquences itérables (les plages). Les plages sont des abstractions de

- Une paire d'itérateur (début,fin)
- Une paire (début,taille) pour des séquences dont la taille est connue ;
- Une paire (début,condition) pour des séquences se terminant sur une condition donnée ;
- Une valeur (début...) pour une séquence non bornée comme une plage retournée par `view::iota`

Remarque : À ma connaissance, la bibliothèque `range` ne compile pas avec clang...

Un premier exemple comme mise en bouche

Exemple canonique

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};

auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
                       | std::views::transform([](int n){ return n * 2; });

for (auto v: results) std::cout << v << " ";     // 4 8 12
```

Exemple : Construction ensemble triplets pythagoriciens

Emploi simple de l'évaluation paresseuse

```
{
    auto entiers = std::ranges::views::iota(1);
    for ( auto value : entiers | std::ranges::views::take(10) )
        std::cout << value << " ";
```

Utilisation des filtres et des transformations

```
{
    auto entiers = std::ranges::views::iota(1);
    auto entiersfiltre = entiers | std::views::filter([](int n){ return n % 2 == 0; })
                                 | std::views::transform([] (int n) { return n*n+1; });

    for ( auto value : entiersfiltre | std::ranges::views::take(10) )
        std::cout << value << " ";
```

Exemple : Construction ensemble triplets pythagoriciens (suite)

Applatissement d'un conteneur de conteneur

```
{
    std::vector tabtab{ std::vector{1.,2.,5.}, std::vector{2.,5.,7.,9.},
                        std::vector{2.,5.   }, std::vector{3.,7.,8.}    };
    auto flatten = tabtab | std::views::join;
    for ( auto elts : flatten ) std::cout << elts << " ";
```

Génération de paires d'entiers

```
{
    auto paires = std::ranges::views::iota(1) | std::views::transform(  [](int z)
                  {    return std::ranges::views::iota(1,z)
                            | std::views::transform( [=](int x){ return std::pair{x,z}; } );
                  } ) | std::views::join;

    for (auto pairvalue : paires | std::ranges::views::take(10) )
        std::cout << pairvalue.first << "," << pairvalue.second << " ";
```

Exemple : Construction ensemble triplets pythagoriciens (suite)

Génération de triplets d'entiers

```
{
    auto triplets = std::ranges::views::iota(1)
                  | std::views::transform([](int z) {
                     return std::ranges::views::iota(1,z) | std::views::transform([=](int x) {
                         return std::ranges::views::iota(1,x) |
                                std::views::transform( [=](int y){ return std::make_tuple(y,x,z); } );
                         }) | std::views::join;
                    } ) | std::views::join;

    for (auto v : triplets | std::ranges::views::take(10) )
        std::cout << std::get<0>(v) << "," << std::get<1>(v) << "," << std::get<2>(v) << " ";
```

Exemple : Construction ensemble triplets pythagoriciens (suite)

Génération de triplets d'entiers (variante)

```
{
    auto triplets = std::ranges::views::iota(1)
                | std::views::transform([](int z) {
                    return std::ranges::views::iota(1,z) | std::views::transform([=](int x) {
                        return std::ranges::views::iota(1,x) |
                               std::views::transform( [=](int y){ if (x*x+y*y==z*z)
                                                                    return std::make_tuple(y,x,z);
                                                                  else
                                                                    return std::make_tuple(0,0,0); } );
                        }) | std::views::join;
                 } ) | std::views::join;

    for (auto vals : triplets | std::ranges::views::take(200) )
        std::cout << "[" << std::get<0>(vals) << "," << std::get<1>(vals)
                  << "," << std::get<2>(vals) << "] ";
```

Exemple : Construction ensemble triplets pythagoriciens (suite)

Génération des triplets pythagoriciens

```
auto triplets = std::ranges::views::iota(1)
              | std::views::transform([](int z) {
                  return std::ranges::views::iota(1,z) | std::views::transform([=](int x) {
                      return std::ranges::views::iota(1,x) |
                             std::views::transform( [=](int y){ if (x*x+y*y==z*z)
                                                                  return std::make_tuple(y,x,z);
                                                                else
                                                                  return std::make_tuple(0,0,0); } );
                      }) | std::views::join;
                } ) | std::views::join |
                      std::views::filter([](std::tuple<int,int,int> t){ return std::get<0>(t) != 0; });

for (auto vals : triplets | std::ranges::views::take(100) )
    std::cout << std::get<0>(vals) << "^2 + " << std::get<1>(vals) << "^2 = "
              << std::get<2>(vals) << "^2" << std::endl;
```

Sommaire

Nombres remarquables (C++ 20)

Les nombres remarquables (π, e, ...) sont enfin définis en template dans la norme dans la librairie `numbers`.
Nombres définis dans l'espace de nommage `std::numbers`. Une version double est toujours proposée en spécialisation du nombre templaté en enlevant le suffixe `_v`. Par exemple `pi_v<double>` peut s'écrire plus simplement `pi`.

- $e \equiv$ `e_v<T>`
- $\frac{1}{\log(2)} \equiv$ `log2e_v<T>` et $\frac{1}{\log(10)} \equiv$ `log10e_v<T>`
- $\pi \equiv$ `pi_v<T>`, $\frac{1}{\pi} \equiv$ `inv_pi_v<T>`, $\frac{1}{\sqrt{\pi}} \equiv$ `inv_sqrt_pi_v<T>` ;
- $\log(2) \equiv$ `ln2_v<T>` et $\log(10) \equiv$ `ln10_v<T>` ;
- $\sqrt{2} \equiv$ `sqrt2_v<T>`, $\sqrt{3} \equiv$ `sqrt3_v<T>` et $\frac{1}{\sqrt{3}} \equiv$ `inv_sqrt3_v<T>` ;
- Constante de Euler-Mascheroni : $\gamma \equiv$ `egamma_v<T>`
- Nombre d'or : $\varphi \equiv$ `phi_v<T>`.

Les nouvelles fonctions mathématiques

- `std::div` : division et reste euclidiens ;
- `std::fma` : Fused multiply addition op.
- `std::exp2` : 2^x
- `std::expm1` : $e^x - 1$
- `std::log2` : $\frac{\log(x)}{\log(2)}$
- `std::crbt` : $\sqrt[3]{x}$
- `std::hypot(x,y[,z]` : $\sqrt{x^2 + y^2[+z^2]}$
- `std::asinh, std::acosh; std::atanh`
- `std::erf` : fonction erreur $\frac{2}{\sqrt{\pi}} \int_0^x \exp^{-t^2} dt$
- `std::tgamma` : Fonction gamma $\int_0^{+\infty} t^{z-1} e^{-t} dt$

Les fonctions mathématiques spéciales (C++ 17)

- `std::laguerre(n,x)` : Évalue le polynôme de Laguerre $L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}(x^n e^{-x})$
- `std::assoc_laguerre(n,m,x)` $= (-1)^m \frac{d^m}{dx^m} L_{n+m}(x)$
- `std::legendre(n,x)` : $P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}(x^2-1)^n$
- `std::assoc_legendre(n,m,x)` : $= (1-x^2)^{\frac{m}{2}}\frac{d^m}{dx^m}P_n(x)$
- `std::beta(x,y)` $= \int_0^1 t^{x-1}(1-t)^{y-1}dt$
- `std::comp_ellint_1(k)` $= \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1-k^2\sin^2\theta}}$
- `std::comp_ellint_2(k)` $= \int_0^{\frac{\pi}{2}} \sqrt{1-k^2\sin^2\theta}d\theta$
- `std::comp_ellint_3(k)` $= \int_0^{\frac{\pi}{2}} \frac{d\theta}{(1-n\sin^2\theta)\sqrt{1-k^2\sin^2\theta}}$
- `std::cyl_bessel_i`, `std::cyl_bessel_j` et `std::cyl_bessel_k`
- `std::cyl_neumann`
- `std::sph_bessel`, `std::sph_legendre` et `std::sph_neumann`
- `std::riemann_zeta`

Opérations numériques sur les tableaux

De nouvelles opérations sont proposées sur les tableaux de valeurs :

- `iota(beg,end,initvalue)` : Rempli le tableau entre les itérateurs `beg` et `end` en incrémentant séquentiellement une valeur commençant par `initvalue` ;
- `reduce` : Comme `accumulate` mais plus efficace en parallèle (out of order)...
- `transform_reduce` : Transforme le tableau avant de faire une réduction ;
- `gcd` : Calcul le pgcd de deux nombres entiers ;
- `lcm` : Calcul le ppcm de deux nombres entiers.
- `midpoint` : Calcul soit la moyenne de deux entiers soit le median des valeurs dans une zone comprise entre deux pointeurs.

Sommaire

Gestion du calendrier (C++ 20)

Inclure `chrono`

Définition de la date et du temps

- Possibilité de définir l'année : `std::chrono::year(2022)` ou `2022y` ;
- Possibilité de définir le mois : `std::chrono::January; std::chrono::September;` ;
- Possibilité de définir le jour : `std::chrono::day(22)` ou `22d` :
- Possibilité de définir l'heure/minute/secondes : `5h+3min+10s`
- On peut aussi être plus précis : `8ms + 10us + 60ns`.

Gestion du calendrier (C++ 20)...

Exemple gestion d'un calendrier

```
void displayDate( year_month_day const& date ) {
    std::cout << "date : " << static_cast<unsigned>(date.day()) << "."
                           << static_cast<unsigned>(date.month()) << "."
                           << static_cast<int>(date.year()) << std::endl;
}

    year_month_day date{2022y, September, 8d};
    displayDate(date);

    const std::chrono::time_point now{std::chrono::system_clock::now()};
    std::chrono::year_month_day today{std::chrono::floor<std::chrono::days>(now)};
    displayDate( today );
    today += months(5u);
    displayDate(today);
```

Remarque : Normalement, il existe un opérateur pour afficher une date, jour, etc. mais pas encore disponibles...

Question sans réponse : Comment rajouter des jours à une date ?

Mesure de la durée

Les horloges

Trois horloges à disposition :

- Horloge système temps réel : `system_clock`
- Horloge monotone s'incrémentant à intervalle régulier : `steady_clock`
- Horloge la plus précise : `high_resolution_clock` qui peut être un alias sur une des deux horloges précédentes.

Chacune de ces horloges possède une méthode statique `now` donnant un repère dans le temps.

Mesure de la durée

On utilise `duration<T,Periode>` qui permet de mesurer le temps écouler entre deux repères de temps selon la période considérée (seconde par défault).

Exemples de mesure de la durée

Exemple

```
auto start = std::chrono::high_resolution_clock::now();
std::vector<double> tab(size, 1.);
auto sum = std::accumulate(tab.begin(), tab.end(), 0.);
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> diffsec = end - start;
std::chrono::duration<double,std::micro> diffmsec = end - start;
std::chrono::duration<float, std::ratio<1,24>> frame = end - start;
std::cout << "Temps traitement avec taille " << size << " => "
          << diffsec.count() << " secondes soit " << diffmsec.count() << "msec"
          << "soit " << 1./frame.count() << " images par seconde" << std::endl;
```

Sommaire

Gestion des nombres aléatoires

Organisation de la bibliothèque

La bibliothèque est scindée en plusieurs parties :

- Des générateurs de nombres aléatoires, eux-mêmes divisés en :
  - Générateurs prédéfinis ;
  - Générateurs paramétrables ;
  - Des adaptateurs définissant des générateurs basés eux-mêmes sur des générateurs.
- Des lois de distributions utilisant un générateur de nombres aléatoires.

Remarque : Le C++ 20 introduit un générateur uniforme de bit aléatoires (concept) qui doit assurer que le générateur vérifiant ce concept fait un tirage aléatoire uniforme d'entiers compris dans un intervalle donné.

Les générateurs paramétrables

Ce sont des générateurs logiciels contrôlables par paramètres templates.

- `std::linear_congruential_engine<Type,a,c,m>` : Défini une suite de nombre aléatoire du type entier donné par le paramètre template : $x_{i+1} \leftarrow (a.x_i + c) \mod m$. Très rapide. Piètre qualité. Période dépend de m.
- `std::mersenne_twister_engine<Type,w,n,m,r,a,u,d,s,b,t,c,l,f>` (Voir Génère en entier dans un interval fermé $[0, 2^w - 1]$.CppReference pour le détail des paramètres !)
- `std::substract_with_carry_engine<Type,w,s,r>` : générateur de fibonacci retardé $S_n \equiv S_{n-j} \star S_{n-k} \mod m$ ($\star$ opérateur binaire).

Adaptateurs pour les générateurs

Permet de modifier le comportement d'un générateur aléatoire.

- `std::discard_block_engine<Generateur,p,r>` : Utilise un générateur pseudo-aléatoire générant un bloc p de nombres et n'en utiliser que r éléments.
- `std::independ_bits_engine<Generateur,w,Type non signé>` : Utilise un générateur pseudo-aléatoire pour généré des nombres entiers contenus dans un nombre spécifique de bits.
- `std::shuffle_order_engine<Generateur,k` : Utilise un générateur pour maintenir un buffer de k entiers tirés pseudo-aléatoirement et quand demandé, renvoie au hasard un de ces entiers qu'il remplace par un nouveau nombre.

Les générateurs prédéfinis

- `std::random_device` : non déterministe (source entropique hardware). Lent, haute qualité.
- `std::default_random_engine` : générateur par défaut. Utilise en général `std::mt19937`
- `std::minstd_rand0` ≡ `std::linear_congruence_engine<uint32,16807,0,2147483647>`
- `std::minstd_rand` ≡ `std::linear_congruential_engine<uint32, 48271, 0, 2147483647>`
- `std::knuth_b` ≡ `std::shuffle_order_engine<std::minstd_rand0,256>`
- `std::mt19937` ≡ `std::mersenne_twister_engine<uint32,...>`
- `std::mt19937_64` ≡ `std::mersenne_twister_engine<uint64,...>`
- `std::ranlux24_base` ≡ `std::subtract_with_carry_engine <uint32, 24, 10, 24> ;`
- `std::ranlux24` ≡ `std::discard_block_engine <ranlux24_base, 223, 23>`
- `std::ranlux48_base` ≡ `std::subtract_with_carry_engine <uint64, 48, 5, 12>`
- `std::ranlux48` ≡ `std::discard_block_engine <ranlux48_base, 389, 11>`

Distributions uniformes

Entités disponibles

- `std::uniform_int_distribution(a,b)` : $\forall i \in [a;b], P(i|a,b) = \frac{1}{b-a+1}$
- `std::uniform_real_distrution(a,b)` : $\forall x \in [a;b[, P(x|a,b) = \frac{1}{b-a}$

Exemples

```
std::random_device rd;
auto generator = std::mt19937(rd());
auto d20 = std::bind(std::uniform_int_distribution<std::uint8_t>(1,20),generator);
for ( int i=0; i<10; ++i )
    std::cout << "Je jette un dé 20 : " << int(d20()) << std::endl;

auto proba = std::uniform_real_distribution<double>(0.,1.);
for (int i=0; i<10; ++i)
    std::cout << "Tu as " << proba(generator) << " chances de réussir !" << std::endl;
```

Distributions de type Bernouilli

Entités disponibles

- `std::bernoulli_distribution(p)` : $P(b|p) = \begin{cases} p & \text{si } b \text{ est vrai} ; \\ 1-p & \text{sinon.} \end{cases}$;
- `std::binomial_distribution(t,p)` : $P(i|t,p) = \left(\begin{array}{c} t \\ i \end{array} \right) .p^i.(1-p)^{t-i}$;
- `std::negative_binomial_distribution(k,p)` : $P(i|k,p) = \left(\begin{array}{c} k+i-1 \\ i \end{array} \right) .p^k.(1-p)^i$;
- `std::geometric_distribution(p)` : $P(i|p) = p.(1-p)^i$

Exemple

```
std::bernoulli_distribution b(0.25);
for (int i=0; i<10; ++i) std::cout << b(generator) << " ";
```

Distributions de type Poissons

Entités disponibles

- `std::poisson_distribution(m)` : $P(i|m) = \frac{e^{-m}m^i}{i!}$;
- `std::exponential_distribution(l)` : $P(x|l) = l.e^{-lx}$;
- `std::gamma_distribution(a,b)` : $P(x|a,b) = \frac{e^{-\frac{x}{b}}}{b^a.\Gamma(a)}.x^{a-1}$
- `std::weilbull_distribution(a,b)` : $P(x|a,b) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} e^{-\left(\frac{x}{b}\right)^a}$
- `std::extreme_value_distribution(a,b)` : $P(x|a,b) = \frac{1}{b}e^{\frac{a-x}{b}-e^{\frac{a-x}{b}}}$

Distributions de type normales

Entités disponibles

- `std::normal_distribution(m,s)` : $P(x|m,s) = \frac{1}{s\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2}$
- `std::lognormal_distribution(m,s)` : $P(x|m,s) = \frac{1}{s.x.\sqrt{2\pi}} e^{-\frac{(\log(x)-m)^2}{2s^2}}$
- `std::chi_squared_distribution(n)` : $P(x|n) = \frac{x^{\frac{n}{2}-1} e^{-\frac{x}{2}}}{\Gamma(n/2).2^{n/2}}$
- `std::cauchy_distribution(a,b)` : $P(x|a,b) = \left(b\pi\left[1+\left(\frac{x-a}{b}\right)^2\right]\right)^{-1}$
- `std::fisher_f_distribution(m,n)` : $P(x|m,n) = \frac{\Gamma\left(\frac{m+n}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\Gamma\left(\frac{n}{2}\right)} \left(\frac{m}{n}\right)^{\frac{m}{2}} x^{\frac{m}{2}-1} \left(1+\frac{m}{n}x\right)^{-\frac{m+n}{2}}$
- `std::student_t_distribution(n)` : $P(x|n) = \frac{1}{\sqrt{n\pi}} . \frac{\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{n}{2}\right)} . \left(1+\frac{x^2}{n}\right)^{-\frac{n+1}{2}}$

Distributions discrètes

Entités disponibles

- `std::discrete_distribution({w1,...,wn})` : $P(i) = \frac{w_i}{\sum_i w_i}$
- `std::piecewise_constant_distribution({b0,b1,...,bn},{w1,...,wn})` :
  $P(x|b_{i-1} \leq x < b_i) = \dfrac{w_i}{(b_i - b_{i-1}) \displaystyle\sum_i w_i}$
- `std::piecewise_linear_distribution({b0,b1,...,bn},{w1,...,wn})` :
  $P(x|b_{i-1} \leq x < b_i) = p_i \dfrac{b_{i+1} - x}{b_{i+1} - b_i}$ avec $p_i = \dfrac{w_i}{\displaystyle\sum_i w_i}$.

Sommaire

Les expression régulières

Principe

Les expressions régulières est une grammaire permettant de définir des motifs, c'est à dire à une description d'un ensemble de chaînes de caractères possédant des caractéristiques communes.

Quelques définitions

- **Séquences cibles** : Chaînes sur lesquelles est appliquée une expression régulière ;
- **Motif** : Séquence de caractères décrivant ce qu'on cherche à identifier ;
- **Correspondance** : Sous-chaîne d'une séquence cible qui correspond au motif.

Exemple

Création expression régulière cherchant motif de type "jour/mois/année" appliquée à la chaîne "La date du 25/12/2014 était un Jeudi". La correspondance trouvée est "25/12/2014".

Les grammaires possibles

Les grammaires disponibles

Plusieurs grammaires disponibles en C++ :

- `std::regex::ECMAScript` : ECMAScript format international (défaut)
- `std::regex::basic` : grammaire POSIX basique
- `std::regex::extended` : grammaire POSIX étendue
- `std::regex::awk` : grammaire utilisée par awk
- `std::regex::grep` : grammaire utilisée par grep
- `std::regex::egrep` : grammaire utilisée par grep avec l'option `-E`.

Exemples

```
std::regex re1(".*(a|xayy)");// ECMAScript par défaut
std::regex re2(".*(a|xayy)", std::regex::extended);// Posix étendu
```

Options disponibles pour les grammaires

Options disponibles

On peut rajouter à ces grammaires différentes options (avec l'opérateur `|`) :

- `std::regex::icase` : Ne pas tenir compte de la casse ;
- `std::regex::nosubs` : Ne pas tenir compte des sous-expressions quand on recherche une correspondance.
- `std::regex::optimize` : Optimisation de l'expression régulière pour les correspondances au détriment du temps de construction de l'expression régulière.
- `std::regex::multiline` : Spécifie que `^` spécifie le début d'une ligne et `$` la fin d'une ligne si `ECMAScript` est utilisé.

Exemples

```
std::regex re2(".*(a|xayy)", std::regex::extended |
                             std::regex::icase);// Posix étendu
```

Expressions régulières

Caractéristiques

- Très difficile à lire : `"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*"`
- Très puissant
- 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

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

Expressions régulières en C++

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;
```

Sommaire

Utilisation des algorithmes parallèles de la STL (C++ 17)

Politique d'exécution

C++ 17 introduit des politiques d'exécutions permettant de choisir la façon d'exécuter une fonction :

- `sequenced_policy seq;` : Utiliser l'algorithme séquentiel pour la fonction
- `parallel_policy  par;` : Utiliser l'algorithme parallèle pour la fonction
- `std::parallel_unsequenced_policy par_unseq;` : Utilise l'algorithme parallèle avec appel non séquencé des fonctions ;
- `std::parallel_unsequenced unseq;` (C++20) : Utilise un algorithme séquentiel avec appel non séquencé des fonctions internes.

Exemple d'utilisation des algorithmes parallèles (C++ 17)

Exemple d'exécution en parallèle

```
template<typename ExecutionPolicy>
void performanceDuTri( ExecutionPolicy&& policy, std::string const& label,
                       std::vector<std::size_t> tableau) {
    auto start = std::chrono::high_resolution_clock::now();
    std::sort(policy, tableau.begin(), tableau.end());
    auto end   = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dt = end - start;
    std::cout << "Durée du tri " << label << " : " << dt.count() << std::endl;
}

    performanceDuTri( std::execution::seq, "séquentiel",tableau);
    performanceDuTri( std::execution::par, "parallèle", tableau);
    performanceDuTri( std::execution::unseq, "non séquencé", tableau);
    performanceDuTri( std::execution::par_unseq, "parallèle non séquencé", tableau);
```

Modèle multithreading

Multithreading en C++

- Permet une gestion multithread d'un programme indépendemment 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 de programm multithread

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 ait fini de s'exécuter
  return EXIT_SUCCESS;
}
```

Passage d'arguments

Syntaxe de std : :thread : `std::thread(fonction,arguments...);`

Exemple

```
void displayIdentite( int id, std::string const& name)
{   std::cout << "Identité de " << id << " : " << name << std::endl;    }

std::thread t(displayIdentite, 3, "maitre");
```

Mais attention aux références !

Programme avec erreur de compilation

```
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);
}
```

- La fonction en (1) attend une référence en deuxième paramètre ;
- Le constructeur en (2) passe par défaut les arguments par valeurs mais la fonction attend une référence ! Erreur de compilation.

Passage de références

Il est donc nécessaire de préciser le passage d'un argument par référence (ou par référence constante) :

Exemple corrigé

```
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);
}
```

Remarque : Pour passer un argument par référence constante, on utilise `std::cref`

Appel méthode d'un objet dans un thread

Appel méthode objet dans un thread

- Même mécanisme que `std::bind` (même problème pour les références...) ;
- On doit passer une adresse de 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);
```

Remarque : Même problématique si la méthode (ou la fonction) est surchargée.

Encapsulation de threads

Thread encapsulé

- Possibilité d'encapsuler un thread dans une classe

```
class scopedThread {
  std::thread t;
public:
  explicit scopedThread(std::thread t_) : t(std::move(t_))
  { if(!t.joinable()) throw std::logic_error("No thread"); }
  ~scopedThread() { t.join(); }
  scopedThread(scopedThread const&)=delete;
  scopedThread& operator=(scopedThread const&)=delete; };
...
};

void f() {
  ...
  scopedThread 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 ;

Exemple de gestion multithreads

```
std::vector<std::thread> threads; threads.reserve(10);
for (int i=0; i<10; ++i) {
    threads.emplace_back([&](int i) { std::cout << "Hello from " << i
                                                << "th thread" << std::endl; }, i+1);
}

for (int i=0; i<10; ++i)
    threads[i].join();
```

Gestion des threads

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...

Exemple

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

Mettre un thread en sommeil

- `std::this_thread::sleep_for(std::chrono::duration duree);` : Met le thread courant en sommeil pendant une durée déterminée ;
- `std::this_thread::sleep_until(std::chrono::time_point temps);` : Met le thread courant en sommeil jusqu'à une certaine date...

Synchroniser les sorties (C++ 20)

Principe

Bibliothèque `syncstream`.

- Wrapper permettant de synchroniser des threads écrivant sur la même sortie ;
- Les flush y sont notés mais pas immédiatement exécutés.

Exemple

```
std::osyncstream sout(std::cout);
void helloFromSync() {
    for (int i=0; i<10; ++i) sout << "Hello from " << std::this_thread::get_id() << std::endl;
}

template<typename HelloFromType>
void tenHelloFrom( HelloFromType&& helloFunc ) {
    auto nbthreads = std::thread::hardware_concurrency();
    std::vector<std::thread> threads; threads.reserve(nbthreads);
    for ( unsigned int i{0}; i<nbthreads; ++i) threads.emplace_back(helloFunc);
    for (auto& thread : threads) thread.join();
}
```

Gestion des conflits mémoires

Exclusion mutuelle (mutex)

- Notion provenant des threads posix ;
- Objets partagés pouvant être verrouillés et dévérouillés ;
- Si l'objet est verrouillé par un thread, les prochains threads voulant le verrouiller doivent attendre qu'il soit déverrouillé ;
- Attention, la procédure de verrouillage/déverrouillage est assez lente !

En C++

Objet de type `std::mutex`. Trois méthodes pour les gérer :

- `void lock()` : Verrouille le mutex. Si déjà verrouillé, bloque jusqu'à ce qu'il soit déverrouillé (par un autre thread) ;
- `void unlock()` : Déverrouille le mutex (généralement verrouillé par le même thread) ;
- `bool try_lock()` : Tente de verrouiller. Si déjà verrouillé, retourne sans avoir verrouillé en retournant `false`.

Remarque : Certains conflits mémoires sont vicieux. Par exemple, attention à l'emploi des champs de bits contigüs (voir l'exemple `STL/bitfield_race_condition.cpp`).

Exemple d'utilisation d'exclusion mutuelle

Exemple sur la recherche de nombres premiers jumeaux

```
void searchTwinPrimes(unsigned nbThreads, unsigned idThread, std::uint64_t max,
                      std::vector<std::pair<std::uint64_t, std::uint64_t>>& twins )
{
    static std::mutex updateTwins;
    std::uint64_t maxIndex = (max-1)/6+1;
    std::uint64_t begLoop = idThread + 1;
    std::uint64_t endLoop = maxIndex + 1;

    for (std::uint64_t index=begLoop; index<endLoop; index += nbThreads)
        if (isPrime(6*index-1) && isPrime(6*index+1))
            {
                updateTwins.lock();
                twins.emplace_back(std::pair{6*index-1,6*index+1});
                updateTwins.unlock();
            }
}
```

Exclusions mutuelles dérivées

C++ propose plusieurs services permettant un usage plus aisé des exclusions mutuelles

`std::timed_mutex`

Mutex proposant les mêmes services que `std::mutex` +

- `m.try_lock_for(std::chrono::duration dt)` : Essaie de verrouiller pendant une certaine durée puis retourne au bout d'un temps dt si échec ;
- `m_try_lock_for(std::chrono::time_point t)` : Essaie de verrouiller jusqu'à un certain temps t puis retourne faux si échec.

`std::recursive_mutex`

Mutex permettant à un thread de le verrouiller plusieurs fois. Le thread devra ensuite déverrouiller autant de fois le mutex qu'il a verrouillé. Même méthodes que `std::mutex`.

`std::recursive_timed_mutex`

Mutex unissant les services de `std::timed_mutex` et `std::recursive_mutex`.

Exclusions mutuelles partagées (C++ 14)

Se trouve dans la bibliothèque `shared_mutex`.

Mutex permettant soit d'avoir un seul thread pouvant le verrouiller (comme un mutex classique) soit plusieurs threads pouvant le verrouiller en partageant le verrou.
Méthodes proposées :

- `lock,try_lock,try_lock_for, try_lock_until, unlock` : Mêmes méthodes de verrouillage/déverrouillage exclusive que pour un mutex classique.
- `lock_shared, try_lock_shared, try_lock_shared_for, try_lock_shared_until, unlock_shared` : Méthodes de verrouillage pouvant être partagés par plusierus threads.

Remarque : Si un thread fait un verrouillage exclusif, les threads cherchant à faire un verrouillage partagé sont bloqués. De même, si un thread effecttue un verrouillage partagé, un thread cherchant à faire un verrouillage exclusif sera bloqué.

Exemple d'utilisation

Verrou pouvant être utilisé en exclusion pour un thread qui écrit dans un dictionnaire et en partagé pour lire dans le dictionnaire.

Remarque 2 : Il existe également un `std::shared_timed_mutex` mélangeant les services des mutex `std::shared_mutex` et `std::timed_mutex`.

Exemple de gestion de mutex partagé

```
struct Dictionnaire {
    void addWord( std::string const& mot, std::string const& definition )
    {  accessLock.lock();  m_dico[mot] = definition;  accessLock.unlock();  }
    void getDefinition( std::string const& mot, std::string& def ) const {
        accessLock.lock_shared();
        if (m_dico.find(mot) != m_dico.end()) {
            def = m_dico.at(mot);
            accessLock.unlock_shared();
        } else {
            def = "";
            accessLock.unlock_shared();
        }                                                          }
...
Dictionnaire dico; std::string def;
dico.addWord("mer", "Grande baignoire avec des vagues");
std::thread t1(&Dictionnaire::addWord, &dico, "océan", "Comme la mer, mais en plus froid" );
std::thread t2(&Dictionnaire::getDefinition, &dico, "mer", std::ref(def) );
std::thread t3(&Dictionnaire::addWord, &dico, "vent", "Substance que certains aiment brasser."   );
std::thread t4(&Dictionnaire::addWord, &dico, "alea jacta est", "Ils sont bavards à la gare de l'Est."   );
t1.join(); t2.join(); t3.join(); t4.join();
```

Verrous utilisant un mutex

Plusieurs verrous sont proposés permettant de faciliter l'utilisation des mutex.

`std::lock_guard<Mutex>(mutex[, strategie])`

Verrouille un mutex à sa création et le déverrouille à sa destruction.

`std::unique_lock<Mutex>(mutex[,strategie])`

Verrou sur un mutex qui essentiellement fait la même chose que `std::lock_guard<Mutex>(mutex)`.
Possibilité de verrouiller/déverrouiller le verrou durant sa durée de vie et tester l'état du verrouillage.

Stratégies possibles

Par défaut, à sa construction, le verrou essaie de verrouiller le mutex. Mais si on passe la stratégie en second argument :

- `std::defer_lock` : Ne tente pas de verrouiller le mutex pendant sa construction ;
- `std::try_to_lock` : Essaie de verrouiller le mutex pendant sa constrution sans blocage ;
- `std::adopt_lock` : Suppose que le thread a déjà verrouillé le mutex.

Interblocage

Définition

Situation où chaque thread cherche à verrouiller un mutex déjà verrouillé par un autre thread.

Exemple de code pouvant générer un interblocage

```
std::mutex m1, m2;
void f() {
  std::lock_guard g1(m1); std::lock_guard g2(m2);
  ... }
void g() {
  std::lock_guard g2(m2); std::lock_guard g1(m1);
  ... }
...
std::thread t1(f), t2(g);
t1.join(); t2.join();
```

Remarque : Intervient principalement quand on utilise plusieurs mutex...

Eviter l'interblocage

C++ propose divers mécanismes permettant d'éviter l'interblocage quand on verrouille plusieurs mutex :

- `std::lock(m1,m2,...)` : Permet de verrouiller un nombre quelconque de mutex, en s'assurant qu'il n'y ait pas d'interblocage.
- `std::scoped_lock(m1,m2,...)` : Verrouille les n mutex passés en paramétres à sa construction et les déverrouille à sa destruction (C++ 17)
- `std::try_lock(m1,m2,...)` : Essaie de verrouiller les mutex passés en paramètres à sa construction et les déverrouille à sa destruction.

Exemple de code ne pouvant pas générer un interblocage

```
std::mutex m1, m2;
void f() {
  std::scoped_guard g(m1,m2);
  ... }
void g() {
  std::scoped_guard g(m2,m1);
  ... }
...
std::thread t1(f), t2(g);
t1.join(); t2.join();
```

Variables et opérations atomiques

Définition opération atomique

Une opération atomique est une opération élémentaire qui sera toujours executée sans qu'aucun autre process ne soit capable de lire ou modifier la mémoire durant l'opération.

- Sur un système mono–cœur : Toute opération correspondant à une seule instruction CPU est atomique. On peut supposer que les opérations machines de type échange (xchg) ou incrément (inc) sont atomiques sur ces systèmes ;
- Sur un système multi-cœur : On utilise des instructions (sur x86 en particuliers) qui verrouillent le bus d'accès à la mémoire durant l'opération, soit systématiquement soit en préfixant l'instruction par un lock (xchg verrouille systèmatiquement, cmpxchg op1,po2 devra être préfixé par lock) ;

Définition variable atomique

Variable sur laquelle on peut effectuer des opérations atomiques (entiers, réels, ...)

Variables atomiques et méthodes associées

Un type template `template<typename T>struct std::atomic;` ayant pour méthodes :

- de stocker une valeur : `store(desire, memory_order=seq_cst)` ;
- lire une valeur : `T load(memory_order)` ;
- Stocker une nouvelle valeur et retourner l'ancienne valeur : `T exchange(desire, memory_order=seq_cst)` ;
- Comparaison égalité d'une valeur `attendu` avec la valeur atomique stockée, si égalité remplace valeur atomique par valeur `desiree` sinon modifier `attendu` à la valeur atomique stockée :
  `bool compare_exchange_weak(T& attendu, T desire,memory_order success, memory_order failure)` (possibilité fausse détection différence) ou
  `bool compare_exchange_strong(T& attendu, T desire,memory_order success, memory_order failure)`
- Attendre et notifier : Bloque jusqu'à ce qu'une notification débloque l'appel (C++ 20) :
  - `wait(old, memory_order)` : Si `old` égale à la valeur atomique, bloque l'appel jusqu'à une notification ou que le thread soit débloqué par erreur sinon retourne immédiatement.
  - `notify_one()` : Débloque par notification un des threads bloqués par un `wait` atomique ;
  - `notify_all()` : Débloque par notification tous les threads bloqués par un `wait` atomique.

Remarque : On va voir dans les prochains transparents ce que signifie le `memory_order` !

Ordre d'accès mémoire

Pour plus de détails : regarder cette vidéo ! ou cette là ou encore celle-ci.

- Ordre accès mémoire se fait lors des opérations atomiques ?
- Si aucune contrainte d'accès, ordre de modification effectué par un thread peut être vu dans des ordres différents par d'autres threads ;

Exemple en pseudo-code et explication

```
# Thread 1 :
write(&data, "Hello world !")
atomic_store(&has_data, True)
```

```
# Thread 2
if atomic_load(&has_data):
    d = read(&data)
    assert(d == "Hello world !")
```

- Le CPU (ou le compilateur) peut reordonner les instructions d'un thread pour optimisation ;
- Il peut très bien décider d'effectuer d'abord l'écriture atomique avant le write ;
- Le thread 2 va donc lire data avant qu'il soit modifié et l'assertion non vérifiée.
- Sans parler de la cohérence de cache qui ne se met pas forcément à jour suivant l'ordre d'écriture...

Contraintes sur l'ordonnancement mémoire

Dans l'exemple précédent, deux problèmes majeurs :

- Reordonnancement des instructions (CPU ou compilateur) ;
- Visibilité d'une modification par d'autres mémoires cache ;

Deux ordonnancement mémoire : acquire et release

- `std::memory_order::release` permet lors d'une opération atomique d'écriture (store) de s'assurer que toutes opérations sur la mémoire déclarées avant cette opération se passent bien avant ;
- `std::memory_order::acquire` permet lors d'une opération atomique de lecture (load) de s'assurer que toutes opérations sur la mémoire déclarées après cette opération se passent bien après.

Application à l'exemple précédent

```
// Thread 1 :
write(&data, "Hello world !");
std::atomic_store(&has_data, true,
                  std::memory_order::release);
```

```
// Thread 2
if (std::atomic_load(&has_data,
                     std::memory_order::acquire)
{
    d = read(&data);
    assert(d == "Hello world !");
}
```

Ordonnancement mémoire séquentiellement consistant

C'est donnée en C++ par `std::memory_order::seq_cst`.

Définition donnée par Leslie Lamport (1979)

...le résultat de toute exécution est le même que si les lectures et écritures avaient lieu dans un certain ordre, et les résultats de chaque processeur apparaissaient dans cette séquence dans l'ordre spécifié par le programme..

- C'est l'ordonnancement par défaut si on omet le paramètre d'ordonnancement ;
- Oblige le processeur et le compilateur a ne pas reordonné vos instructions ;
- Peut-être plus coûteux que si on laisse le reordonnancement se faire.
- **Vous assure de ne pas avoir d'effets secondaires** : Commencer toujours avec ce modèle avant de vouloir optimiser !.

Ordonnancement mémoire tolérant

C'est donnée en C++ par `std::memory_order::relaxed`.
Attention, ce modèle permet au compilateur et au CPU de pouvoir reordonner vos instructions.
Avec ce modèle, il n'est pas garanti que votre code fonctionne correctement. Par contre, permet une bonne optimisation pour le CPU et le compilateur.

Gestion des lignes de cache

Le standard C++ propose deux constantes permettant de gérer les lignes de caches (librairie `new`) :

- `std::hardware_constructive_interference_size` : Permet de connaître l'alignement nécessaire pour commencer les données d'une structure sur une ligne de cache ;
- `std::hardware_destructive_interference_size` : Permet de connaître l'alignement nécessaire des membres d'une structure pour **ne pas** partager une ligne de cache.

Exemple

```
// Permet d'aligner la structure afin qu'elle tient dans une ligne de cache
struct alignas(hardware_constructive_interference_size) InterneLigneCache_t {
    std::atomic_uint64_t x{};
    std::atomic_uint64_t y{};
} interneLigneCache;

// Permet que les deux membres de la structure ne soient pas dans la même ligne de cache.
struct EnDehorsLigneCache_t {
    alignas(hardware_destructive_interference_size) std::atomic_uint64_t x{};
    alignas(hardware_destructive_interference_size) std::atomic_uint64_t y{};
} enDehorsLigneCache;
```

Variables de conditions `std::condition_variable`

Variable conditionnelle

- Permet à un thread d'attendre qu'une condition soit vérifiée tout en ne bloquant pas de mutex ;
- Thread en état de sommeil, besoin de notifier le thread pour le "réveiller" ;

```
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();  ... } }
```

Exercice : Queue multithreadable

Queue dans contexte multithreadée

- Mettre en œuvre une queue prévue pour 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.

Remarque : Si on veut utiliser des conteneurs de type STL mais compatibles avec un environnement multithreadé, on peut utiliser la bibliothèque TBB d'Intel (gratuite, utilisée pour les algorithmes parallèles de STL !).

Tâches asynchrones (bibliothèque `future`)

Principe

Les threads ne permettent pas d'utiliser directement une fonction retournant une valeur :

```
double computeFoo();
double val;
std::thread t([](double& retValue){ retValue=computeFoo(); }, std::ref(val));
...
t.join();
```

Fonctions asynchrones permettent "facilement" d'appeler une fonction dans un thread puis de récupérer "plus tard" le résultat !

Qu'entend-on par "plus tard"

En fait, trois mécanismes possibles (à choisir) :

- `std::launch::async` : Fonction exécutée par un autre thread, synchronisée lors de la récupération de la valeur retournée ;
- `std::launch::deferred` : Fonction exécutée quand on récupére sa valeur de retour (évaluation paresseuse)...
- `std::launch::async|std::launch::deferred` : Fonction exécutée sur un autre thread et évaluée au moment de sa récupération (mode par défaut)

Lancer une tâche asynchrone

Comment lancer une tâche asynchrone ?

Une tâche asynchrone peut être exécutée à l'aide de la fonction
`std::async([std::launch policy,] function, arguments...) -> std::future`.

Comment accèder au retour de la fonction asynchrone ?

C++ propose un mécanisme utilisant une structure template `std::future<T>` permettant de pouvoir récupérer la valeur de la fonction asynchrone. Avec cette structure, on peut appeler les méthodes :

- Attendre et récupérer la valeur (`get()`) ;
- Tester si la valeur est récupérable (`valid() -> bool`) ;
- Attendre que la valeur soit récupérable (`wait()`) ;
- ou attende un certain temps ou jusqu'à une certaine heure que la valeur soit récupérable (`wait_for(durée)` et `wait_until(date)`.

Exemple de tâche asynchrone (reportée)

```
double computeDistances( Point const& t_target, std::future<Point> t_barycenter ) {
    std::cout << __PRETTY_FUNCTION__ << std::endl;
    auto barycenter = t_barycenter.get();
auto generateMasses( std::int64_t nbMasses ) -> std::vector<Point> {
Point computeBarycenter( std::future<std::vector<Point>>  t_masses ) {
    constexpr std::int64_t nbBodies = 100'000'000;
    std::cout << "Appel calcul des masses" << std::endl;
    auto masses = std::async(std::launch::deferred, generateMasses, nbBodies);
    std::cout << "Appel calcul du barycentre" << std::endl;
    auto bary   = std::async(std::launch::deferred, computeBarycenter, std::move(masses));
    std::cout << "Appel calcul des distances" << std::endl;
    Point target{ 2., 0., 0.};
    auto fdist   = std::async(std::launch::deferred, computeDistances, target, std::move(bary));
    std::cout << "Distance des points à la cible \n " << fdist.get() << std::endl;
```

Tâches asynchrone : partage de la valeur de retour

Et si plusieurs threads veulent accèder au résultat ?

Possibilité d'utiliser `std::shared_future` qui permet à plusieurs threads d'accèder au résultat en possédant leur copie propre de `std::shared_future`.

Exemple

```
for (auto centre : { Point{-2., -2., -2.}, Point{-2., -2., +2.},
                     Point{-2., +2., -2.}, Point{-2., +2., +2.},
                     Point{+2., -2., -2.}, Point{+2., -2., +2.},
                     Point{+2., +2., -2.}, Point{+2., +2., +2.}
                   })
{
    b.emplace_back(std::async(policy,generateMasses, centre, nbBodies).share());
}
```

Thread auto joignable (C++ 20)

Principe

- Créer un thread qui se synchronise automatiquement à sa destruction (satisfait au principe RAII) ;
- Permet de s'assurer de sa synchronisation même en cas d'exception du programme principal.
- Permet de gérer un thread pour qu'il quitte proprement une boucle infinie.

Exemple

```
void jthread_cancel() {
    using namespace std::literals::chrono_literals;
    //La fonction prend un jeton d'arrêt :
    std::jthread jt([](std::stop_token stoken) {
        while (!stoken.stop_requested()) {
        // Du travail... Simulation
        std::this_thread::sleep_for(1s);
    } });

    std::this_thread::sleep_for(5s);
    // Le Thread est arrêté et synchronisé à la destruction.
```

Jetons d'arrêts (C++ 20)

Principe

- Permet d'arrêter un thread ou collectivement plusieurs threads
- De définir une fonction appeler automatiquement lors de l'arrêt
- Utile pour arrêter proprement un thread attendant qu'une condition soit vérifiée !

Exemple

```
void jobPooler1(std::stop_token stoken) {
    // Enregistrer un callback stop
    std::stop_callback cb(stoken, []() { cv.notify_all(); });
    while (true) { ...
        cv.wait(lck, [stoken]() { return jobs.size() > 0 || stoken.stop_requested(); });
        if (stoken.stop_requested()) break;
        ...
    } // Fin boucle while
}
```

Suite de l'exemple utilisant un jeton d'arrêt

Exemple (suite...)

```
template<typename JobPoolerType> void poolManager( JobPoolerType&& t_pooler ) {
    std::stop_source ssource;
    std::thread esclave1(t_pooler, ssource.get_token()), esclave2(t_pooler, ssource.get_token());
    // Soumets quelques jobs
    for (int idJob = 0; idJob < 10; idJob++) {
        {
        std::unique_lock lck(mut);
        jobs.push(idJob);
        cv.notify_one(); //Wakes up only one worker
        }
    }
    // Arrêter tous les threads
    ssource.request_stop();
    // Synchronisation
    esclave1.join(); esclave2.join();
}
```

Simplification de l'exemple précédent

Principe

- Possibilité en C++ 20 de rajouter un jeton d'arrêt dans la méthode `wait` de `std::condition_variable_any` ;

Jobpooler simplifié

```
std::condition_variable_any cv_any;
void jobPooler2(std::stop_token stoken) {
    while (true) {
    int jobId = -1;
    { // Acquérir un job dans une section gardée
        std::unique_lock lck(mut);
        if(!cv_any.wait(lck, stoken, []() { return jobs.size() > 0; })) {
            break;
        }
        jobId = jobs.front(); jobs.pop();
    }
    ...
    }
}
```