C++ expert, les avancées du langage

VEC

1

Sommaire

FABIEN.BRISSONNEAU@GMAIL.COM

2

2

L’avènement du C++11

Les différentes normes C++98, C++03,C++0x, C++11

Les nouveautés du C++11 et les objectifs de cette norme

Le devenir de BOOST, STL

La gestion de la comptabilité des codes anciens

La disponibilité des outils de développement (compilateurs, débogueurs, IDE…)

Les différentes normes

Création en 1979, par Bjarne Stroustrup, basé sur Simula67

La première version Cfront translate le C++ en C. Abandonné en 1993.

1985 : version commerciale

1989 : ajout de l'héritage multiple

1991 : comité ISO C++

1992 : première STL

1998 : C++98, première version ISO, intègre la STL

2003 : C++03 corrections mineures

2005 : émission du TR1, présentant les nouveautés à venir

Les normes récentes

2011 : C++11, prenant en compte TR1, Boost, C99

BOOST : ensemble de bibliothèques, fondée en 1999, visant à apporter des idées au futur standard C++

2012 : création du Standard C++ Foundation

2014 : C++14, considérée comme une version mineure, impacte le langage et la bibliothèque

2015 : TS, filesystem

2017 : C++17, prenant en compte TS, C11, Boost, modifie langage et bibliothèque

Les différentes normes

C++0x : en attente d'une version majeur avant 2010

C++11 : auto, decltype, default et delete fonctions, final et override, rvalue, move, enums class, constexpr, listinitializer, constructeurs délégués et hérités, utilisation de {}, nullptr, long long, char16_t, char32_t, aliases, template variadic, lambdas expressions, noexcept, range for, static_assert

Opération atomic, initializer_list,forward_list, chrono, thread, améliorations des itérateurs

7

FABIEN.BRISSONNEAU@GMAIL.COM

7

Les différentes normes

C++14 : std::make_unique, …

Déduction du type de return …

C++17 : class template argument deduction, std::variant, filesystem, std::any,

C++ est C++11, espaces de noms stdx réservés, scoped_lock, shared_ptr pour tableau, shared_mutex, clamp, std::string_view

8

FABIEN.BRISSONNEAU@GMAIL.COM

8

Les nouveautés et les objectifs de C++11

Le C++11 introduit des nouveautés dans le langage, et complète la bibliothèque.
Objectifs : code plus rapidement, de façon plus élégante, en écrivant du code maintenable
Modifications :
- garder la compatibilités avec C++98 et le C
- faire évoluer la bibliothèque avant le langage
- préférer les changements qui font évoluer les techniques de programmation
- faciliter la mise en place de bibliothèques
- augmenter la protection des types
- augmenter la capacité de travailler directement avec le matériel
- proposer des solutions propres aux problèmes actuels
- implémenter le zéro-overhead (ça coûte uniquement si on s'en sert)
- rendre le C++ plus facile à apprendre

FABIEN.BRISSONNEAU@GMAIL.COM

Boost

La Pointer Container Library
Les boost::any et boost::variant
Programmation événementielle
Gestion des processus

FABIEN.BRISSONNEAU@GMAIL.COM

Pointer Container Library

Palier le problème des conteneurs de la STL qui stocke les objets concrets

Les conteneurs de le PCL contiennent des pointeurs vers des objets dynamiquement alloués, mais les désallouent à la fin de leur vie

boost::ptr_array, boost::ptr_vector, boost::ptr_list

boost::ptr_set, boost::ptr_map, boost::ptr_multiset, boost::ptr_multimap

boost::ptr_unordered_set, boost::ptr_unordered_map, boost::ptr_ unordered_ multiset, boost::ptr_ unordered_ multimap

Exemple de ptr_vector

Les destructeurs sur Fille et Fille2 sont appelés

```
boost::ptr_vector<Mere> ptrVec;

ptrVec.push_back(new Fille);
ptrVec.push_back(new Fille2);

Mere& m = ptrVec.front();
```

Boost Variant

Repose sur des templates

Défini un ensemble de types possibles

<boost/variant.hpp>

```
boost::variant< int, std::string>  valeur;

valeur = 1;

valeur = "Hello";

std::cout << boost::get<int>(&valeur);
```


FABIEN.BRISSONNEAU@GMAIL.COM

Boost Any

A besoin des RTTi

<boost/any.hpp>

```
boost::any v1, v2, v3;

v1 = "hello";
v2 = 42;
v3 = std::string("bonjour");

try {
    std::cout << boost::any_cast<const char*> (v1) << std::endl;
    std::cout << boost::any_cast<int> (v2) << std::endl;
    std::cout << boost::any_cast<std::string> (v3) << std::endl;
}
catch (std::exception & e) {
    std::cout << e.what() << std::endl;
}
```


FABIEN.BRISSONNEAU@GMAIL.COM

La programmation événementielle

- Signals et Slots
- Les connect peuvent être ordonnés
- Arguments possibles
- Un signal, plusieurs slots connectés

```
struct Hello
{
    void operator()() const
    {
        std::cout << "Bonjour Boost";
    }
};
```

```
boost::signals2::signal<void()> sig;

sig.connect(Hello());

sig();
```


C++11

C++11 core language features

C++11 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	Nvidia nvcc	HP aCC	Digital Mars C++	[Collapse]
C99 preprocessor	N1653	4.3	Yes	19.0* (partial)*	Yes	4.1	11.1	10.1	5.9	Yes	8.4	2015	7.0	A.06.25	Yes	
`static_assert`	N1720	4.3	2.9	16.0*	Yes	4.1	11.0	11.1	5.13	Yes	8.4	2015	7.0	A.06.25	8.52	
Right angle brackets	N1757	4.3	Yes	14.0*	Yes	4.1	11.0	12.1	5.13	Yes	8.4	2015	7.0			
Extended `friend` declarations	N1791	4.7	2.9	16.0* (partial) 18.0*	Yes	4.1	11.1* 12.0	11.1	5.13	Yes	8.4	2015	7.0	A.06.25		
`long long`	N1811	Yes	Yes	14.0*	Yes	Yes	Yes	Yes	Yes	Yes	8.4	2015	7.0	Yes	Yes	
Compiler support for type traits	N1836	4.3	3.0	14.0*	Yes	4.0	10.0	13.1.3	5.13	Yes	8.4	2015		6.16		
`auto`	N1984	4.4	Yes	16.0*	Yes	3.9	11.0 (v0.9) 12.0	11.1	5.13	Yes	8.4	2015	7.0	A.06.25		
Delegating constructors	N1986	4.7	3.0	18.0*	Yes	4.7	14.0	11.1	5.13	Yes	8.4	2015	7.0	A.06.28		
`extern template`	N1987	3.3	Yes	12.0*	Yes	3.9	9.0	11.1	5.13	Yes	8.4	2015	7.0	A.06.25		

C++11

C++11 library features

C++11 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC Standard Library	Apple Clang	Sun/Oracle C++ Standard Library	Embarcadero C++ Builder Standard Library	Cray C++ Standard Library	[Collapse]
Type traits	N1836	4.3	3.0	14.0*	Yes	5.13	Yes	8.4	
Garbage Collection and Reachability-Based Leak Detection (library support)	N2670	6 (no-op)	3.4 (no-op)	19.0* (no-op)	Yes (no-op)				
Money, Time, and hexfloat I/O manipulators	N2071 N2072	5	3.8	19.0*	Yes	5.15			

FABIEN.BRISSONNEAU@GMAIL.COM

C++14

C++14 core language features

C++14 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	Nvidia nvcc	[Collapse]
Tweaked wording for contextual conversions	N3323	4.9	3.4	18.0*	Yes	4.9	16.0	13.1.2*	5.15	10.3	8.6	16.1	9.0	
Binary literals	N3472	4.3 (GNU) 4.9	2.9	19.0*	Yes	4.10	11.0	13.1.2*	5.14	10.3	8.6	2015	9.0	
`decltype(auto)`, Return type deduction for normal functions	N3638	4.8 (partial)* 4.9	3.3 (partial)* 3.4	19.0*	Yes	4.9	15.0	13.1.2*	5.15	10.3	8.6	16.1	9.0	
Initialized/Generalized lambda captures (init-capture)	N3648	4.5 (partial) 4.9	3.4	19.0*	Yes	4.10	15.0		5.15	10.3	8.6	16.1	9.0	
Generic (polymorphic) lambda expressions	N3649	4.9	3.4	19.0*	Yes	4.10	16.0	13.1.2*	5.15	10.3	8.6	16.1	9.0	
Variable templates	N3651	5	3.4	19.0*	Yes	4.11	17.0	13.1.2*	5.15	10.3	8.6	17.4	9.0	
Extended constexpr	N3652	5	3.4	19.10*	Yes	4.11	17.0	13.1.2*	5.15	10.3	8.6	17.4	9.0	
Member initializers and aggregates (NSDMI)	N3653	5	3.3	19.10*	Yes	4.9	16.0		5.14	10.3	8.6	16.1	9.0	
Clarifying memory														

FABIEN.BRISSONNEAU@GMAIL.COM

C++14

C++14 library features

C++14 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC Standard Library	Apple Clang	Sun/Oracle C++ Standard Library	Embarcadero C++ Builder Standard Library	Cray C++ Standard Library	[Collapse]
constexpr for <complex>	N3302	5	3.4	19.0*	Yes	5.15	10.3	8.6	
std::result_of and SFINAE	N3462	5	Yes	19.0*	Yes	5.15	10.3	8.6	
constexpr for <chrono>	N3469	5	3.4	19.0*	Yes	5.15	10.3	8.6	
constexpr for <array>	N3470	5	3.4	19.0*	Yes	5.15	10.3	8.6	
constexpr for <initializer_list>, <utility> and <tuple>	N3471	5	3.4	19.0*	Yes	5.15	10.3	8.6	
Improved std::integral_constant	N3545	5	3.4	19.0*	Yes	5.15	10.3	8.6	
User-defined literals for <chrono> and <string>	N3642	5	3.4	19.0*	Yes	5.15	10.3	8.6	
Null forward iterators	N3644	5 (partial)	3.4	19.0*	Yes	5.15	10.3	8.6	
std::quoted	N3654	5	3.4	19.0*	Yes	5.15	10.3	8.6	
Heterogeneous associative lookup	N3657	5	3.4	19.0*	Yes	5.15	10.3	8.6	
std::integer_sequence	N3658	5	3.4	19.0*	Yes	5.15	10.3	8.6	
std::shared_timed_mutex	N3659	5	3.4	19.0*	Yes	5.15	10.3	8.6	
std::exchange	N3668	5	3.4	19.0*	Yes	5.15	10.3	8.6	
fixing constexpr member functions without const	N3669	5	3.4	19.0*	Yes	5.15	10.3	8.6	
std::get<T>()	N3670	5	3.4	19.0*	Yes	5.15	10.3	8.6	
Dual-Range std::equal, std::is_permutation, std::mismatch	N3671	5	3.4	19.0*	Yes	5.15	10.3	8.6	

19

FABIEN.BRISSONNEAU@GMAIL.COM

C++17

C++17 core language features

C++17 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	Nvidia nvcc	[Collapse]
New auto rules for direct-list-initialization	N3922	5	3.8	19.0*	Yes	4.10.1	17.0			10.3		17.7		
static_assert with no message	N3928	6	2.5	19.10*	Yes	4.12	18.0			10.3		17.7		
typename in a template template parameter	N4051	5	3.5	19.0*	Yes	4.10.1	17.0			10.3		17.7		
Removing trigraphs	N4086	5	3.5	16.0*	Yes	5.0	19.0.1			10.3				
Nested namespace definition	N4230	6	3.6	19.0*	Yes	4.12	17.0			10.3		17.7		
Attributes for namespaces and enumerators	N4266	4.9 (partial)* 6	3.6	19.0*	Yes	4.11	17.0			10.3		17.7		

20

FABIEN.BRISSONNEAU@GMAIL.COM

C++17

C++17 library features

C++17 feature	Paper(s)	GCC libstdc++	Clang libc++	MSVC Standard Library	Apple Clang	Intel Parallel STL	Sun/Oracle C++ Standard Library	Embarcadero C++ Builder Standard Library	Cray C++ Standard Library	[Collapse]
std::void_t	N3911	6	3.6	19.0*	Yes	N/A		10.3		
std::uncaught_exceptions()	N4259	6	3.7	19.0*	Yes	N/A				
std::size(), std::empty() and std::data()	N4280	6	3.6	19.0*	Yes	N/A		10.3		
Improving std::pair and std::tuple	N4387	6	4	19.0*	Yes	N/A		10.3		
std::bool_constant	N4389	6	3.7	19.0*	Yes	N/A		10.3		
std::shared_mutex (untimed)	N4508	6	3.7	19.0*	Yes	N/A		10.3		
Type traits variable templates	P0006R0	7	3.8	19.0*	Yes	N/A		10.3		

Exercices

Les améliorations du langage

Les énumérations fortement typées

Les tableaux à taille fixe

Le mot-clé auto pour simplifier le typage

La boucle basée sur un intervalle

Autres améliorations : templates à arguments variables, pointeur nul, …

Les enum class

Enum classiques

```
enum type_enum { LUNDI, MARDI, MERCREDI };

int valeur = type_enum::LUNDI;

std::cout << "valeur enum " << valeur << std::endl;
```

Avec « enum class », impossible de confondre enum et int

```
enum class type_enum_c11 { LUNDI, MARDI, MERCREDI};


type_enum_c11 jour = type_enum_c11::LUNDI;

int autre= type_enum_c11::LUNDI;
```

nullptr

Pour identifier un pointeur null, on utilisait 0 ou #define NULL 0

La valeur nullptr est plus claire

```
int* ptr = nullptr;

ptr = appel_fonction();

if (ptr == nullptr) {
    std::cout << " le ptr est encore null " << std::endl;
}
```


Le mot clé auto

L'inférence de type permet de simplifier le code

Laisse le compilateur déduire le type de la variable

```
auto valeur = 42;
// valeur est un int

auto now = std::chrono::system_clock::now();
// std::chrono::system_clock::time_point now = std::chrono::system_clock::now();
```

Nouvelle syntaxe des fonctions

Le type de retour est après la spécification de fonction

```
// int ma_fonction(int)  {   }

auto ma_fonction(int param) -> int {   }
```

```
class typeexterne {
    class typeinterne {};

    auto ma_fonction(int param)->typeinterne;
};
```

```
// typeexterne::typeinterne typeexterne::ma_fonction(int) { }

auto typeexterne::ma_fonction(int) -> typeinterne {  }
```


Auto ...

```
auto fontion_retourne_int()
{
    int valeur{ 3 };

    return valeur;
}
```

```
int fontion_retourne_int()
{
    int valeur{ 3 };

    return valeur;
}
```

```
auto& fontion_retourne_intref (int& valeur) {

    return valeur;
}
```

```
int& fontion_retourne_intref (int& valeur) {

    return valeur;
}
```

Auto …

```
auto fonction_avec_const_auto() {
    int a{ 4 };
    int b{ 2 };
    const auto & v = a + b;

}
```

```
template<auto n>
auto f() -> std::pair<decltype(n), decltype(n)>
{
    return { n, n };
}
```


La boucle for-range

Pour le parcours d'un conteneur

```
int tab[] = { 4,5,6,7,8 };

for (auto v : tab) {
    std::cout << " la valeur est " << v;
}
```

Evite la déclaration d'une variable de boucle

Evite d'expliciter le début et la fin

Utiliser auto& pour obtenir une référence

Le mot-clé constexpr

Déclare une expression constante à la compilation

```
constexpr unsigned int taille() { return 42; }

int main() {
    int tableau_constante[42];
    int tableau_fonction[taille()];
    const int valeur = 42;
    int tableau_const[valeur];
```


Expression if-constexpr

C++17

```
if constexpr (std::is_pointer_v<T>)
```

Si une branche n'est pas atteinte, elle n'est pas compilée

Template variadic

Nombre de paramètres template variable

```
template <typename T, typename... Rest>
double somme(T t, Rest... rest) {
    return t + somme(rest...);
}
```


Sizeof ...

Usage avec assert_static

```
template <typename T, typename... Rest>
double somme(T t, Rest... rest) {

    assert_static(sizeof ...(Rest) < 3, "Erreur, trop de params");

    return t + somme(rest...);
}
```

Fold expression

C++17

```
template <typename... Rest>
double somme2( Rest... rest) {
    return (rest + ...);
}
```


Les attributs

Soit standards, soit spécifiques à une implémentations (constructeur)

Standard attributes

Only the following attributes are defined by the C++ standard.

`[[noreturn]]`	indicates that the function does not return
`[[carries_dependency]]`	indicates that dependency chain in release-consume `std::memory_order` propagates in and out of the function
`[[deprecated]]`(C++14) `[[deprecated("reason")]]`(C++14)	indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some *reason*
`[[fallthrough]]`(C++17)	indicates that the fall through from the previous case label is intentional and should not be diagnosed by a compiler that warns on fall-through
`[[nodiscard]]`(C++17)	encourages the compiler to issue a warning if the return value is discarded
`[[maybe_unused]]`(C++17)	suppresses compiler warnings on unused entities, if any

Exercices

Les classes

La délégation de constructeurs, les contraintes liées à l'héritage

La nouvelle sémantique du déplacement et le constructeur de move

Adaptation de la forme normale aux nouveautés

Les directives =delete et =default

Les initialisateurs de conteneurs

Les données membres

Initialisation des membres

Initialisation des membres directement lors de la déclaration

```
class List {
    std::vector<int> data{ 4,5,6,7 };
    int taille_max { 42 };
    std::string label = "list_int";
```


Délégation de constructeurs

Un constructeur en rappelle un autre

```
class voiture {

    std::string immat;
public :

    voiture(std::string immat) : immat(immat) {}
    voiture() : voiture("AC-671-YN") {}
};
```

Les directives =delete et =default

Par défaut, le compilateur génère plusieurs fonctions membres

Exemple avec le constructeur de copie

```
class classe_copiable {

};
```

```
classe_copiable c;

classe_copiable d = c;
```

Avec =delete, la fonction est supprimée

Avec =default, la fonction existe

```
class classe_copiable {
public:
    classe_copiable() = default;
    classe_copiable(const classe_copiable&) = delete;
};
```

```
impossible de faire référence à fonction "classe_copiable::classe_copiable(const classe_copiable &)" (déclaré à la ligne 109) -- fonction supprimée
'classe_copiable::classe_copiable(const classe_copiable &)' : tentative de référencement d'une fonction supprimée
```


FABIEN.BRISSONNEAU@GMAIL.COM

Les r-values et le move

Une r-value est une expression qui s'utilise en partie droite d'une affectation

La sémantique du move doit éviter la copie lorsque l'objet qui possédait la ressource va disparaître

```
class mon_conteneur {
    const ma_data* ptr;

public:
    mon_conteneur(const ma_data* ptr) : ptr(ptr) {}
    mon_conteneur(const mon_conteneur& d) { ptr = new ma_data{ *d.ptr }; }
    ~mon_conteneur() { delete ptr; }
};
```

Le constructeur de copie est indispensable pour éviter l'aliasing de pointeurs

FABIEN.BRISSONNEAU@GMAIL.COM

La copie / r-value

Traces dans la ressource manipulée

```
class ma_data {
public:
    ma_data() { std::cout << "ctor de ma_data" <<std::endl; }
    ma_data(const ma_data&) { std::cout << "cpy ctor de ma_data" << std::endl; }
    ~ma_data() { std::cout << "dtor de ma_data" << std::endl; }
};
```

La fonction qui génère la copie

```
mon_conteneur createur() {
    mon_conteneur retval(new ma_data());
    return retval;
}
```

```
mon_conteneur p2{ createur() };          // cpy ctor
std::cout << " utilisation des donnees" << std::endl;
```


FABIEN.BRISSONNEAU@GMAIL.COM

La copie / le move

Sans le move

- Le constructeur de copie est appelé
- On duplique les données

```
ctor de ma_data
cpy ctor de ma_data
dtor de ma_data
 utilisation des donnees
dtor de ma_data
```

Avec le constructeur de move

```
mon_conteneur(mon_conteneur&& d) noexcept {
    ptr = d.ptr;
    d.ptr = nullptr;
}
```

```
ctor de ma_data
 utilisation des donnees
dtor de ma_data
```


FABIEN.BRISSONNEAU@GMAIL.COM

Forme normale des classes

Le C++ 11 propose 6 fonctions générées par défaut

```
class normale {
public:
    normale() ;                             // constructeur par défaut
    ~normale();                             // destructeur

    normale(const normale&);                // constructeur de copie
    normale(normale&&) noexcept;            // constructeur de move

    normale& operator=(const normale&);     // opérateur d'affectation
    normale& operator=(normale&&) noexcept; // opérateur de move
};
```


Liste d'initialisation

Par analogie avec

```
un_conteneur_struct d1{ 2,3,4 };
d1.print();

un_conteneur_struct d2 = { 2,3,4 };
d2.print();
```

```
struct un_conteneur_struct {
    int data1;
    int data2;
    int data3;

    void print() {
        std::cout << data1 << " " << data2 << " " << data3 << std::endl;
    }
};
```

Utiliser std::initializer_list

Un constructeur avec paramètre std::initializer_list

```
un_conteneur_class d3{ 2,3,4 };
d3.print();

un_conteneur_class d4 = { 2,3,4 };
d4.print();
```

```
class un_conteneur_class {
    int data1;
    int data2;
    int data3;
public:
    un_conteneur_class(std::initializer_list<int> liste) {
        auto it = liste.begin();
        data1 = *it; ++it;
        data2 = *it; ++it;
        data3 = *it;
    }
    void print() {
        std::cout << data1 << " " << data2 << " " << data3 << std::endl;
    }
};
```


FABIEN.BRISSONNEAU@GMAIL.COM

Exercices

FABIEN.BRISSONNEAU@GMAIL.COM

L'utilisation des threads

Déclaration et exécution d'un thread. Attente de fin d'exécution avec join()

Récupérer un résultat avec future<> et async()

Obtenir les capacités d'exécution de la plateforme avec hardware-concurrency()

FABIEN.BRISSONNEAU@GMAIL.COM

La gestion des threads

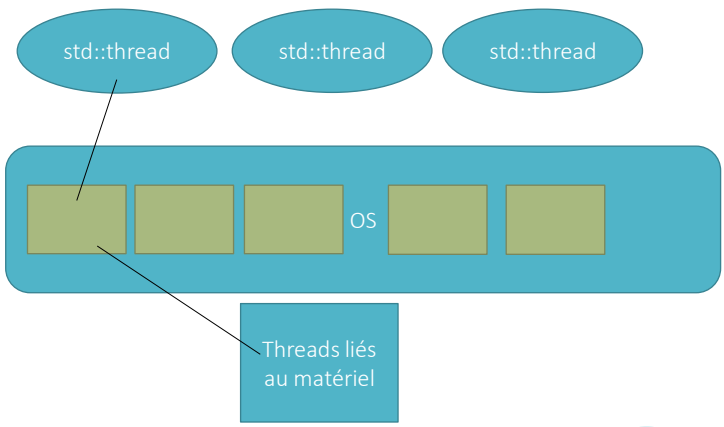
Thread et tâches

Passer des arguments

Partager des données

Attendre des événements

Niveau conceptuel



FABIEN.BRISSONNEAU@GMAIL.COM

Les threads

La CPU fournit les threads hardware, unités d'exécution dans la CPU
La fonction hardware_concurrency() indique le nombre

Le système d'exploitation fournit les threads software. Ils sont sous le contrôle du scheduler, exécutés sur les threads hardware.

La librairie C++ fournit des objets thread, associés ou non aux thread OS. Un thread n'est pas joignable si :
- construit par défaut (pas de fonction à exécuter)
- moved (n'est plus lié à la fonction à exécuter)
- detaché (non lié au main)
- déjà joined (un appel à join() déjà fait)

FABIEN.BRISSONNEAU@GMAIL.COM

Thread et tâches

#include <thread>

La classe thread représente un thread système
Le constructeur de l'objet thread prend en paramètre une fonction

```
std::thread t1{ fonc1 };
std::thread t2{ fonc2 };

t1.join();
t2.join();
```

Lorsque l'objet thread est détruit, il doit :
- Être joined
- Être detaché

Sinon, std::terminate sera appelée

La fonction membre join est bloquante. Elle attend que le thread termine
La fonction detach() permet au main de terminer sans considérer la fin des autres threads

FABIEN.BRISSONNEAU@GMAIL.COM

Thread joignable

```
for(std::thread& t  : threads) {
    if (t.joinable()) {
        t.join();
    }
}
```

Appeler join() sur le thread est nécessaire avant que l'objet ne soit détruit.

Passer des arguments

Dans <functional>
std::ref force un passage en référence

```
void fonc2(std::vector<int>& vec) {
    std::cout << " fonction 2" << std::endl;
    for (auto v : vec) {
        std::cout << " vec " << v << std::endl;
    }
}
```

```
std::vector <int> vecteur{ 1,2,4,5,8,6 };

std::thread t1{ fonc1, ref(vecteur)};
std::thread t2{ fonc2, ref(vecteur)};

t1.join();
t2.join();
```

Un thread exécute une fonction, éventuellement en passant les paramètres nécessaires. Ces paramètres peuvent représentés des données partagées par les threads.
Chaque thread va lire les données sans synchronisation avec les autres.

Mutex

```
#include <mutex>

std::mutex le_mutex;
int la_variable = 0;
```

Unlock appelé en sortie de bloc

Sur lock_guard

```
std::lock_guard<std::mutex> garde(le_mutex);

// accès aux donnnées partagées
la_variable = la_variable < 10 ? la_variable + 1 : la_variable;

std::cout << " thread " << std::this_thread::get_id() << "leaving" << std::endl;
```

Pour protéger les données partagées, une première approche est de verrouiller les accès des threads
Sur des portions de code.
Le mutex est objet sur lequel on appelle les fonctions lock() et unlock() resp. avant et après le code à protéger
Pour éviter que unlock() ne soit pas appelé (par exemple en cas de levée d'exception), utiliser lock_guard :
- dans son constructeur, appel de lock()
- dans son destructeur, appel de unlock()

Partager des données

Protéger les accès avec un mutex et scoped_lock

```
std::mutex m;

void fonc1(std::vector<int>& vec) {
    std::scoped_lock lck{ m };

    std::cout << " fonction 1" << std::endl;
    for (auto v : vec) {
        std::cout << " vec " << v << std::endl;
    }
}
```

Ou dans une autre sémantique
shared_lock (transfert de lock)
unique_lock (1 seul mutex, transfert)
scoped_lock (plusieurs mutex)

Possibilité de lock/unlock explicit

Les verrous

Scoped_lock : déverrouille les mutex à la fin de la portée, mais peut être créé sur plusieurs mutex

Unique_lock : verrouille l'accès à un mutex en exclusivité, fonctions supplémentaires par rapport à lock_guard et gère les transferts de responsabilité

Shared_lock : verrouille l'accès à un mutex mais permet plusieurs accès, gère les transferts de responsabilité

Mutex avec 2 niveaux d'accès

Niveau shared : accès simultané

Niveau exclusive : accès unique

```
mutable std::shared_mutex le_mutex;

unsigned int lire() const {
    std::shared_lock lock(le_mutex);

    return la_valeur;
}

void incremente() {
    std::unique_lock lock(le_mutex);
    la_valeur++;
}
```

Condition variables

Une condition variable fait qu'un thread peut attendre jusqu'à ce qu'une condition particulière soit atteinte

Permet aussi à un thread d'envoyer un signal à d'autres threads

Pattern classique : consommateur / producteur

Nécessite l'utilisation d'un verrou (ie unique_lock) avant d'entrer dans le wait(…)

Le wait va déverrouiller le lock passé en paramètre

Plusieurs versions de wait existent (avec ou sans condition)

FABIEN.BRISSONNEAU@GMAIL.COM

Attendre des événements

Attendre une durée

```
std::this_thread::sleep_for(milliseconds{ 20 });
```

Utiliser <condition_variable>

```
std::queue<std::string> mqueue;
std::condition_variable mcond;

void consumer() {
    while (true) {
        std::unique_lock lck{ m };
        mcond.wait(lck, [] {return !mqueue.empty(); });
        auto msg = mqueue.front();
        mqueue.pop();
```

```
mqueue.push("hello " + cpt);
mcond.notify_one();
```


FABIEN.BRISSONNEAU@GMAIL.COM

Niveau conceptuel

Utiliser
- future, promise,
  - *Passer des valeurs entre tâches*
- packaged_task,
  - *Wrapper pour gérer promise et future*
- async
  - *Traiter une tâche comme une fonction*

Utiliser future et promise permet de ne pas mettre en œuvre variables globales et verrous (mutex, locks)
Les futures et promises représentent les 2 côtés d'une valeur partagée par des threads
Le future est le côté recevant la valeur
La promise est le côté retournant la valeur

L'objectif est de na pas avoir à gérer bas niveau les threads et les synchronisations

Future et promise

Include <future>

Future<T>
- get pour récupérer la valeur attendue

Promise<T>
- set_value : positionner une valeur
- set_exception : positionner une exception

La fonction get() attend la valeur et doit être placée dans un try, pour récupération des exceptions dans un catch

Future et promise

```cpp
#include <future>

void la_fonction(std::promise<int> promesse) {

    promesse.set_value(5);
}
```

```cpp
std::promise<int> promesse;
std::future<int> futur = promesse.get_future();

std::thread le_thread(la_fonction, std::move(promesse));

std::cout << futur.get() << std::endl;

le_thread.join();
```

Dans cet exemple de future/promise
La promesse permet de générer une valeur à partir d'un thread
La future permet de récupérer la valeur

Le couple future/promise gère le passage de valeurs entre 2 threads

Packaged_task

Ces tâches automatisent les créations des futures/promises

Encapsuler les tâches

Démarrer un thread pour la tâche

```cpp
using FType = int(int, int);

std::packaged_task<FType> pt0{ calcul };
std::packaged_task<FType> pt1{ calcul };

std::future<int> f0{ pt0.get_future() };
std::future<int> f1{ pt1.get_future() };

std::thread t1{ move(pt0), 1,25 };
std::thread t2{ move(pt1), 26,50 };

std::cout << "resultat " << f0.get() + f1.get() << std::endl;
```

Dans cet exemple, les tâches permettent de
- Créer les futures qui récupèrent les données
- Créer automatiquement les promises à partir du retour de la fonction calcul

Le type FType est défini comme un type fonction
qui prend 2 entiers en paramètre et retourne un paramètre
C'est le type de la fonction calcul()

La fonction async

Traiter les tâches comme des fonctions

Pas de maîtrise des threads

```
int calcul(int a, int b) {
    int retval = 0;
    for (int i = a; i < b; i++) retval += i;
    return retval;
}
```

```
auto f0 = std::async(calcul, 1, 25);
auto f1 = std::async(calcul, 26, 50);

std::cout << "resultat " << f0.get() + f1.get() << std::endl;
```

Avec la fonction template std::async, nous nous débarrassons de la gestion manuelle des threads
Conceptuellement nous passons du multi-threads au multi-tâches

Capacité

Nombre de threads exécutables

```
unsigned long const hard_threads = std::thread::hardware_concurrency();
```

La fonction hardware_concurrency() peut retourner 0, signifiant qu'elle n'est pas implémentée pour cette plateforme

Variables atomiques

Existe pour les types scalaires std::atomic<T>

Applicable sur les types utilisateurs, si copiables

```
std::atomic_bool valeur{ true };

std::atomic<int> entier{ 42 };
```

```
if (valeur.is_lock_free()) {
    std::cout << " bool lock free " << std::endl;
}
```

Une variable atomique est une variable qui peut être modifiée et utilisée de plusieurs threads simultanément
Suivant les plateformes, une variable atomique peut ou non utiliser un verrou

FABIEN.BRISSONNEAU@GMAIL.COM

Compare-and-swap

```
bool val = valeur.load();

if (valeur.compare_exchange_weak(val, true)) {
    std::cout << "valeur vaut " << std::boolalpha << valeur.load() << std::endl;
}
```


FABIEN.BRISSONNEAU@GMAIL.COM

Algorithmes parallèles C++17

Choix d'une politique d'exécution

```
std::for_each(std::execution::seq, begin(vec), end(vec),
    [](int i) {/*traitement*/ });

std::for_each(std::execution::par, begin(vec), end(vec),
    [](int i) {/*traitement*/ });
```

En C++17, certains algorithmes peuvent supporter une exécution soit séquentielle, soit parallèle

Version C++11

std::thread
std::mutex
std::recursive_mutex
std::condition_variable
std::condition_variable_any
std::lock_guard
std::unique_lock
std::packaged_task
std::async
std::future

Version C++14

std::shared_lock

std::shared_timed_mutex

Version C++17

std::shared_mutex

std::scoped_lock

Organisation

<thread>

Classes : std::thread et sous std::this_thread -> yield, get_id, sleep_for, sleep_until

<mutex>

Classes : mutex, timed_mutex, recursive_mutex, recursive_timed_mutex, lock_guard, scoped_lock, unique_lock

Fonctions : try_lock, lock, call_once

<shared_mutex>

Classes : shared_mutex, shared_timed_mutex, shared_lock

FABIEN.BRISSONNEAU@GMAIL.COM

Organisation

<future>

Classes : promise, packaged_task, future, shared_future

Fonctions : async

<condition_variable>

Classes : condition_variables, condition_variable_any

Fonctions : notify_all_at_thread_exit

FABIEN.BRISSONNEAU@GMAIL.COM

Exercices

Autres nouveautés de la bibliothèque

La gestion du temps avec le namespace chrono()

Le nouveau conteneur tuple()

Les conteneurs …

Evolutions

Boost : usage commercial ou non

Bibliothèques en partie intégrées à C++11

En 2005, ensemble de spécifications concernant la bibliothèque standard

Technical Report 1

Normalisé en 2007

FABIEN.BRISSONNEAU@GMAIL.COM

Les nouveaux conteneurs

Std::forward_list

Std::priority_queue

Std::unordered_map

Std::unordered_multimap

Std::unordered_set

Std::unordered_multiset

FABIEN.BRISSONNEAU@GMAIL.COM

Le conteneur std::foward_list

Include <forward_list>

Liste simplement chainée

Optimisé pour des petites tailles

Possède un itérateur forward

Fonctions membres : front(), pop_front(), push_front() …

Le conteneur std::priority_queue

Include <queue>

Fonctions push, pop, top

Une fonction de comparaison permet de garder un ordre dans la queue

Les conteneurs std::unodered_xxx

Reprennent les noms _map, _set, _multimap, _multiset

Includes <unordered_map> et <unordered_set>

Basés sur des tables de hashage

Utilisent une fonction std::hash<T>

Les clés ne sont pas triées

Conteneur	Perf de operator [ ]	Itérateurs
vector	const	Random
list		Bi
forward_list		For
deque	const	Random
stack		
queue		
priority_queue		
map	O(log(n))	Bi
multimap		Bi
set		Bi
multiset		Bi
unordered_map	const	For
unordered_multimap		For
unordered_set		For
unordered_multiset		For

La classe tuple

Include <tuple>

Conteneur pour des objets de types différents

```
std::tuple<std::string, int, double> personne{ "Fabrice",32, 2574 };

std::cout << " Nom " << std::get<0>(personne)
    << " age " << std::get<1>(personne)
    << " salaire " << std::get<2>(personne)
    << std::endl;
```


Les smart pointers

Include <memory>

std::unique_ptr

std::shared_ptr

std::weak_ptr

Le smart pointer std::unique_ptr

Garantit la propriété unique de la ressource

Détruit la ressource lorsque unique_ptr devient inaccessible

```
std::unique_ptr<int> ptr{ new int(42) };

std::cout << *ptr << std::endl;
```

Fonctions membre release, reset, get, …

Opérateurs *, ->, []

Le smart pointer shared_ptr

Compte le nombre de références sur une ressource

Fonctions get, use_count, unique

Opérateurs *, ->, comparaisons

```
std::shared_ptr<int> ptr{ new int(42) };
auto ptr2 = ptr;

std::cout << *ptr << std::endl;
std::cout << ptr.use_count() << std::endl;
```

Le smart pointer weak_ptr

Référence temporaire

Doit être transformé en shared_ptr avant utilisation

```
std::shared_ptr<int> ptr{ new int(42) };
std::weak_ptr<int> ptr2 = ptr;

std::cout << *ptr2.lock() << std::endl;
```

Fonctions lock, unique, reset, use_count

Les foncteurs

Plus, minus, multiplies, divides, modulus, negate

Equal_to, not_equal_to, greater, less, greater_equal, less_equal

Les binders

Include <fonctional>

std::bind

bind1st et bind2nd obsolètes

```
bool predicat(int a, int b) {
    return a < b;
};
```

```
auto plus_court = std::bind(predicat, 1, 2);

std::cout << plus_court() << '\n';
```


std::bind

Lier un paramètre

```
auto plus_court = std::bind(predicat, std::placeholders::_1, 2);

std::cout << plus_court(4) << '\n';
```

Ou deux

```
auto plus_court = std::bind(predicat, std::placeholders::_1, std::placeholders::_2);

std::cout << plus_court(4,2) << '\n';
```

Dans un autre sens

```
using namespace std::placeholders;

auto plus_court = std::bind(predicat, _2, _1);

std::cout << plus_court(4,2) << '\n';
```

Exemple random

#include <random>

```
using moteur = std::default_random_engine;
using distribution = std::uniform_int_distribution<>;

moteur le_moteur {};
distribution la_distribution{ 1,6 };

int tirage = la_distribution(le_moteur);
```

```
moteur le_moteur {};
le_moteur.seed(time(0));
```


FABIEN.BRISSONNEAU@GMAIL.COM

Exemple chrono

#include <chrono>

```
using namespace std::chrono;

time_point<system_clock> tp = system_clock::now();
std::cout << " heure " <<  tp.time_since_epoch().count();

time_point<system_clock> tp2 = system_clock::now();
auto duree = duration_cast<milliseconds>(tp2 - tp);
std::cout << " duree " << duree.count();
```

A la place de time(0)

```
moteur le_moteur {};
le_moteur.seed(system_clock::now().time_since_epoch().count());
```


FABIEN.BRISSONNEAU@GMAIL.COM

La classe tuple

Include <tuple>

Conteneur pour des objets de types différents

```
std::tuple<std::string, int, double> personne{ "Fabrice",32, 2574 };

std::cout << " Nom " << std::get<0>(personne)
    << " age " << std::get<1>(personne)
    << " salaire " << std::get<2>(personne)
    << std::endl;
```


Contraintes sur template

Détecter à la compilation

```
static_assert (booleen, "message");
```

Obtenir des caractéristiques sur les types

```
template<typename T> class addition {
    static_assert(std::is_arithmetic<T>(), " Le type ne peut pas supporter l'addition");
public :
    T ajoute(T a, T b) {
        return a + b;
    }
};
```

Autre type_traits

Application du principe « substitution failure is not an error »

```
template<typename T> class addition {

public :

    template<typename R>
    typename std::enable_if_t<std::is_arithmetic_v<R>,R> ajoute(R a, R b) {
        return a + b;
    }

};
```

```
addition<int> add;
add.ajoute(4, 5);

addition<std::string> add_string;

add_string.ajoute("e", "f");
```


FABIEN.BRISSONNEAU@GMAIL.COM

SFINAE

Ici la fonction n'est générée que si le type sur lequel elle est appliqué a un membre « type »

```
struct UnType {
    using type=int;
};

template<typename T, typename = typename T::type> void une_fonction(int);
```

```
une_fonction<int>(5);       // pas de membre "type" dans int

une_fonction<UnType>(5);    // il y a un membre "type" dans UnType
```


FABIEN.BRISSONNEAU@GMAIL.COM

Utiliser std::variant

Détermine un ensemble de types possibles

```
std::variant<Chat, Chien> get_variant(unsigned int valeur) {
    if (valeur < 5) return Chat();
    else return Chien();
}


std::variant<Chat, Chien> bete = get_variant(2);

try {
    std::cout << " la variant contient " << bete.index() << std::endl;
    std::get<Chat>(bete).parle();
}
catch (std::bad_variant_access) {
    std::cout << " erreur ! ";
}
```


FABIEN.BRISSONNEAU@GMAIL.COM

Utiliser std::any

Include <any>

Représente une donnée quelconque

```
std::any retourne(unsigned int valeur ) {
    if (valeur != 42) {
        return 0;
    }
    else {
        return std::string("C'est la réponse !");
    }
}
```

```
std::any v = retourne(42);
std::string reponse = std::any_cast<std::string>(v);
```

En cas d'erreur, lève une exception std::bad_any_cast

FABIEN.BRISSONNEAU@GMAIL.COM

Système de fichier C++17

Représentation d'un chemin : path

Erreur liée au filesystem : filesystem_error

Répertoire : directory_entry

Les itérateurs : directory_iterator et recursive_directory_iterator

Les infos sur un fichier : file_status

Les espaces disponibles sur un filesystem : space_info

Des énumérés : file_type, perms, copy_options, directory_options

Exercices

La programmation fonctionnelle

Déclaration, typage, implémentation et utilisation

L'intérêt d'auto avec les lambdas expressions

La gestion des fermetures avec capture par valeur ou par référence

Définition

Paradigme de programmation dans lequel le programme est codé comme une évaluation d'expressions, sans changement d'état ni données mutables.

Principes : fonction pure, immutable data, récursion, fonction de haut niveau, etc

Les lambdas expressions

Expression d'un fonction, très courte

Pour

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

std::for_each(begin(conteneur), end(conteneur), ma_fonction);
```

Au lieu de

```
void ma_fonction(int v) {
    std::cout << v << ", ";
}
```

Faire

```
std::for_each(begin(conteneur), end(conteneur), [](int v) {std::cout << v << ", "; });
```


Concept function

Un wrapper de fonctions :

```
std::function<void(int)> ref;

ref = [=](int valeur) {  if (valeur > variable_locale)  std::cout << valeur; };
```

Fonctions de haut niveau

Transforment les conventions d'appels

```
auto f = std::bind(&affiche, "Brissonneau", "Fabien", 51);
f();
```

```
using namespace std::placeholders;
auto f = std::bind(&affiche, "Brissonneau", _1, _2);
f("Fabien",51);
```

```
using namespace std::placeholders;
auto f = std::bind(&affiche, "Brissonneau", _2, _1);
f(51,"Fabien");
```


Les lambdas et la capture

Pour utiliser des variables locales

```
std::string sep = ",";
std::for_each(begin(conteneur), end(conteneur), [&sep](int v) {std::cout << v << sep; });
```

Possibilité de les modifier

```
std::string sep = ",";
int compteur = 0;

std::for_each(begin(conteneur), end(conteneur),
    [&sep, &compteur](int v) {
                                std::cout << v << sep;
                                compteur++;
                            });
```

Les lambdas et la capture

Possibilités : [] -> rien, [&] -> tout par référence, [=] -> tout par valeur, [&a,=b] -> a par réf, et b par valeur, [=,&a] -> tout par valeur sauf a par réf

```
std::vector<int> v{ 4,5,6,7,8,9 };

int variable_locale = 5;
std::for_each(begin(v), end(v), [=](int valeur) {  if (valeur > variable_locale)  std::cout << valeur; });
```


Lambdas et types

```
std::function <int(int)> creer_fonction(int seuil) {

    return [=](int valeur) { std::cout << valeur;  return seuil + valeur; };
}
```

```
auto fct = creer_fonction(5);

std::for_each(begin(v), end(v), std::ref(fct));
```

Lambdas polymorphes

Les types des paramètres sont quelconques

```
auto f = [](auto n) { std::cout << " hello" << n; };

f(4);
f("coucou");
```


Opérateurs unaires et binaires

Un opérateur est une fonction membre ou non-membre

Représentée par un symbole

Il est possible de surcharger la plupart des opérateurs
Sauf ::, ., .*, ?:, sizeof, typeid, <xx>_cast

Syntaxe	Fonctions
a@	a.operator@() // fct membre
	operator@(a) // fct non-membre
a@b	a.operator@(b) // fct membre
	operator@(a,b) // fct non-membre

Les foncteurs

Un foncteur est un objet qui possède l'opérateur ()
  void operator() (int v) {}

Permet une syntaxe d'appel identique à une fonction

Mais possède des attributs

Les foncteurs

Pour le code suivant
  La valeur du seuil est globale

```
int seuil = 6;
void applique_regle(int v) {
    if (v < seuil)  std::cout << " valeur sélectionnnée " << std::endl;
}

int main() {

    std::vector<int> data = { 4,5,7,8,9 };

    std::for_each(begin(data), end(data), applique_regle);

    return 0;
}
```

Les foncteurs

Classe avec opérateur ()

```
int main() {

    std::vector<int> data = { 4,5,7,8,9 };

    foncteur_regle fct(6);

    std::for_each(begin(data), end(data), fct);

    return 0;
}
```

```
class foncteur_regle {
    int seuil;
public:
    foncteur_regle(int s) : seuil(s) {}
    void operator() (int v) {
        if (v < seuil)  std::cout << " valeur sélectionnnée " << std::endl;
    }
};
```


Exemples avec les algorithmes

Read-only
Copie : copy, move, transform
Write-only
Tri
Swap, Reverse, Partition
Rotation, Permutation
Tas et tri
Merge et mergesort
Lower_bound
Remove_if

Read only

```
std::cout << "Distance  "
    << std::distance(begin(mon_vecteur), end(mon_vecteur));

std::cout << "Nombre d'elements sup à 3 "
    << std::count_if(begin(mon_vecteur), end(mon_vecteur), [](int v) {return v > 3; });

std::cout << "Count  "
    << std::count(begin(mon_vecteur), end(mon_vecteur),242);

std::cout << "Find  "
    << *std::find(begin(mon_vecteur), end(mon_vecteur),5);

std::cout << "Find if "
    << *std::find_if(begin(mon_vecteur), end(mon_vecteur), [](int v) {return v > 5; });
```


Read only

Algos all_of, none_of, any_of

```
std::cout << "None of "
    << std::boolalpha << std::none_of(begin(mon_vecteur), end(mon_vecteur), [](int v) {return v > 5; });
```

Find_first_of

```
std::istream_iterator<char> it{ std::cin};
std::istream_iterator<char> end{};
std::string str{ "fin" };

auto resp = std::find_first_of(it, end, str.begin(), str.end());
std::cout << " trouve : " << *resp;
```

Copie

Std::copy

```
std::vector<int> mon_vecteur;
for (int i = 1; i < 11; i++) {
    mon_vecteur.push_back(i);
}
std::vector<int> ma_copie(10);

std::copy(mon_vecteur.begin(), mon_vecteur.end(), ma_copie.begin());

for (auto v : ma_copie) {
    std::cout << v << std::endl;
}
```


Copie

Std::back_inserter

Les algorithmes n'allouent pas de mémoire

```
std::vector<int> ma_copie;

std::copy(mon_vecteur.begin(), mon_vecteur.end(), std::back_inserter(ma_copie));
```

Move, Transform

Sémantique du move : std::move

Copie avec transformation

```
std::transform(mon_vecteur.begin(), mon_vecteur.end(), std::back_inserter(ma_copie), [](int v) {return v + 100; });
```


Write-only

Std::fill

```
std::vector<int> mon_vecteur(10);

std::fill(begin(mon_vecteur), end(mon_vecteur), 42);
```

Tri

Std::sort

```
std::sort(begin(mon_vecteur), end(mon_vecteur), [](int a, int b) {return a % 6 < b % 6; });
```

Std::stable_sort

Autres algorithmes

Swap, reverse

Rotate

Make_heap

Remove + erase

Les algorithmes remove marquent les éléments à supprimer

Il faut ensuite appeler erase sur le conteneur

```
auto ptr = std::remove_if(begin(mon_vecteur), end(mon_vecteur), [](int v) { return v > 3; });

mon_vecteur.erase(ptr, mon_vecteur.end());
```


Paramétrages par foncteurs

Un foncteur est un objet possédant l'operateur ()

```
struct foncteur {
    int seuil;
    bool operator() (int v) {
        return v > seuil;
    }
};

auto ptr = std::remove_if(begin(mon_vecteur), end(mon_vecteur), foncteur{ 3 });
```

Les adaptateurs

Std::back_inserter

Std::front_inserter

```
std::copy(begin(mon_vecteur), end(mon_vecteur), std::back_inserter(ma_copie));
```

Std::inserter

```
auto ptr = std::find(begin(ma_copie), end(ma_copie), 5);
std::copy(begin(mon_vecteur), end(mon_vecteur), std::inserter(ma_copie, ptr));
```


FABIEN.BRISSONNEAU@GMAIL.COM

Les itérateurs

Homogènes aux pointeurs

Permettent d'accéder aux conteneurs

Sont les paramètres des algorithmes génériques

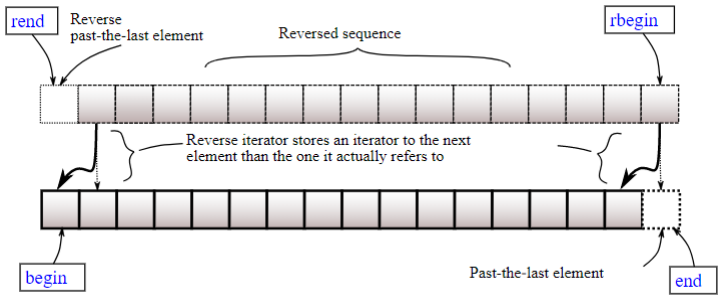
Représentent une position dans le conteneur

Abstraits par rapport à la nature du conteneur

FABIEN.BRISSONNEAU@GMAIL.COM

Les itérateurs sur les conteneurs

Membres begin, end, cbegin, cend, rbegin,rend,crbegin,crend



FABIEN.BRISSONNEAU@GMAIL.COM

Catégorie d'itérateurs

6 catégories d'itérateurs :

- Input, Forward, Bidirectional, Random
- Output
- Contiguous (c++17)

La catégorie de l'itérateur dépend des possibilités du conteneur, et représente ce dont un algorithme a besoin

C++20 change la façon de voir les itérateurs

FABIEN.BRISSONNEAU@GMAIL.COM

Exercices

La gestion de la mémoire et conteneurs

Les smart pointers :
- shared_ptr,
- unique_ptr,
- weak_ptr,
- auto_ptr

Usage conjoint avec la STL

Raii avec auto_ptr

Sorte de smart pointer

Supporte le transfert de responsabilité de la destruction de la ressource

```
std::auto_ptr<res> ptr1{ new res };

std::auto_ptr<res> ptr2 = create();

manip(ptr2);

std::cout << "fin du bloc" << std::endl;
```

```
std::auto_ptr<res> create() {
    return std::auto_ptr<res>{new res};
}

void manip(std::auto_ptr<res> p) {

}
```

```
dtor de res
fin du bloc
dtor de res
```


Les smart pointers

Include <memory>

std::unique_ptr

std::shared_ptr

std::weak_ptr

Le smart pointer std::unique_ptr

Garantit la propriété unique de la ressource

Détruit la ressource lorsque unique_ptr devient inaccessible

```
std::unique_ptr<int> ptr{ new int(42) };

std::cout << *ptr << std::endl;
```

Fonctions membre release, reset, get, …

Opérateurs *, ->, []

Le smart pointer std::shared_ptr

Compte le nombre de références sur une ressource

Fonctions get, use_count, unique

Opérateurs *, ->, comparaisons

```
std::shared_ptr<int> ptr{ new int(42) };
auto ptr2 = ptr;

std::cout << *ptr << std::endl;
std::cout << ptr.use_count() << std::endl;
```

Le smart pointer std::weak_ptr

Référence temporaire

Doit être transformé en shared_ptr avant utilisation

```
std::shared_ptr<int> ptr{ new int(42) };
std::weak_ptr<int> ptr2 = ptr;

std::cout << *ptr2.lock() << std::endl;
```

Fonctions lock, unique, reset, use_count

Les expressions régulières

Include <regex>

Classes template typées du type de la chaîne

std::match représente une correspondance de motif

std::regex_search recherche une occurrence du motif

std::regex_match fait correspondre une chaîne

std::regex_replace recherche et remplace

std::regex_iterator itère sur les correspondances

std::regex_token_iterator itère sur les non-correspondances

Les expressions régulières

Recherche de motif dans une chaîne

```
std::regex motif(R"(\w{2}\s*\d{3}\s*\w{2})");

std::string chaine_a_tester = "df 456 rf";

std::smatch correspondance;

if (std::regex_search(chaine_a_tester, correspondance, motif))
    std::cout << " voici " << correspondance[0];
```


Les expressions régulières

Vérifier la correspondance

```
if (std::regex_match(chaine_a_tester, motif))
    std::cout << " la chaîne correspond !";
```

Itérer

```
for (std::sregex_iterator it(str.begin(), str.end(), motif); it != std::sregex_iterator{}; ++it)
        std::cout << (*it)[0];
```

La bibliothèque chrono

- Définition de durées, horloges, d'instants
- Classe duration
- Classes system_clock, steady_clock, high_resolution_clock
- Classe time_point

```
std::chrono::time_point<std::chrono::system_clock> debut, fin;

debut = std::chrono::system_clock::now();
```


La bibliothèque chrono

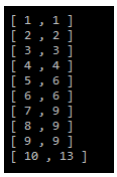
- Récupération de durées

```
int ecoule_en_secondes =
std::chrono::duration_cast<std::chrono::seconds>(fin - debut).count();

std::time_t fin_time = std::chrono::system_clock::to_time_t(fin);
```

Capacité et taille

```
for (int i = 1; i < 11; i++) {
    mon_vecteur.push_back(i);
    std::cout << " [ " << mon_vecteur.size() << " , " << mon_vecteur.capacity() << " ]" << std::endl;
}
```



FABIEN.BRISSONNEAU@GMAIL.COM

Les allocateurs et les conteneurs

L'allocateur est l'objet qui va gérer la mémoire pour le conteneur

Sauf std::array

La valeur par défaut est std::allocator<T>

FABIEN.BRISSONNEAU@GMAIL.COM

Créer son propre allocateur

```
template <class T>
struct MyAllocator {
    using value_type = T;

    // default constractor
    // copy constractor
    // move constractor
    MyAllocator() { ... }

    template <class U>
    MyAllocator(const MyAllocator<U>&) { ... }
```

```
T* allocate(std::size_t n)
{
    std::cout << " allocate " << n << std::endl;
    return reinterpret_cast<T*>(std::malloc(sizeof(T) * n));
}

void deallocate(T* p, std::size_t n)
{
    std::cout << " deallocate " << n << std::endl;
    std::free(p);
}
```


Utiliser son allocateur

Paramètre template

```
std::vector<int, MyAllocator<int>> mon_vecteur;
```

```
for (int i = 1; i < 11; i++) {
    mon_vecteur.push_back(i);
    std::cout << " [ " << mon_vecteur.size() << " , " << mon_vecteur.capacity() << " ]" << std::endl;
}
```

```
allocate 1
[ 0 , 0 ]

allocate 1
[ 1 , 1 ]
allocate 2
deallocate 1
[ 2 , 2 ]
allocate 3
deallocate 2
```

Exercices

Fin