

C++

Grégory Lerbret

4 novembre 2023

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Rappels historiques

- Années 80 – « C with classes » par Bjarne Stroustrup aux Bell Labs
- 1983 – renommé C++
- 1985 – première version publique de CFront
- 1985 – première version de *The C++ Programming Language*
- 1998 – première normalisation
- 2003 – amendement
- 2007 – publication du premier *Technical Report* (TR1)
 - Partiellement implémenté par certains compilateurs ou Boost
 - Partiellement repris dans les normes suivantes et TS
- Projet de TR2 finalement transposé en *Technical Specification*

Philosophie du C++

- Multi-paradigme
- Typage statique déclaratif
- Généraliste
- Initialement, ajout des classes au C
- Vaste sous-ensemble commun (proche du C) entre C et C++
- *Zero-overhead abstraction*
- Compatibilité ascendante forte mais pas absolue
- Évolutions par les bibliothèques plutôt que par le langage
- Pas de « magie » dans la bibliothèque standard

Normalisation

- Normalisé par l'ISO (JTC1/SC22/WG21)
- Comité distinct de celui du C
- ... mais plusieurs membres en commun
- Pas de propriétaire du C++
- Actualité de normalisation, et du C++ en général : isocpp.org

isocpp.org n'est pas le site du comité

- Site de *Standard C++ Foundation* dont le but est la promotion du C++
 - Les deux sont cependant très proches et partagent de nombreux membres
-
- Dépôt GIT (brouillons et propositions)
 - Conférence annuelle cppcon

Norme & support

- Compilateurs
 - GCC – C++ Standards Support in GCC
 - Clang – C++ Support in Clang
 - Visual studio – Conformité du langage Microsoft C++
- Bibliothèques standards
 - GCC – status.html
 - Clang – C++ Standard Library
- Vision globale – C++ compiler support

Sites de référence C++

- cppreference.com
- hacking C++

Erreurs – Code retour

- Plusieurs variantes
 - Type de retour dédié
 - Valeur particulière notant un échec (`NULL`, `-1`)
 - Récupération de la dernière erreur (`errno`, `GetLastError()`)
- Nécessite « un test toutes les deux lignes »
- Gestion manuelle de la remontée de la pile d'appel
- Adapté au traitement local des erreurs, pas au traitement « plus haut »

Problèmes et limites

- Impact négatif sur la lisibilité
- Souvent délaissée dans un contexte d'enseignement ou de formation
- Beaucoup de code avec une gestion d'erreur déficiente

Erreurs – Exceptions

- Lancées par `throw`
- Attrapées par `catch()` depuis un bloc `try`

```
try {  
    ...  
    // Lancement d'une exception  
    throw logic_error("Oups !");  
    ...  
}  
catch(logic_error& e) {  
    // Traitement de l'exception  
    ...  
}
```

Erreurs – Exceptions

- Type quelconque
- Idéalement héritant de `std::exception` (via `std::logic_error`, `std::runtime_error` ou autres)
- `catch(...)` pour attraper les exceptions de tout type
- Compatibles avec le *stack unwinding*
- Pas de `finally`
- Appel de `std::terminate()` si une exception n'est pas attrapée
- Utilisées par la bibliothèque standard (p.ex. `std::bad_alloc`)

Erreurs – Critiques des exceptions

- Critiquées, voire interdites, par certaines normes de codage (p.ex. : Google C++ Style Guide)
- Arguments très variés
 - « Je ne comprends pas », « Ça ne sert à rien », ...
 - Impact négatif sur les performances

À nuancer

- Initialement vrai
- Actuellement, une exception non levée ne coute quasiment rien
- Souvent comparée à une non gestion d'erreur, est-ce pertinent ?

Erreurs – Critiques des exceptions

- Mauvais support par les différents outils

À nuancer

- Correctement supportées par les compilateurs actuels
 - Inégalement gérées par les outils d'analyse, de documentation, ...
- Code plus complexe à analyser
 - Difficiles à introduire dans une large base de code sans exception
 - Absence d'ABI normalisée

Erreurs – Exception safety

- *No-throw guarantee* : l'opération ne peut pas échouer

Do

- Destructeurs et `swap()` ne doivent pas lever d'exception
- *Strong exception safety* : pas d'effet de bord, pas de fuite, état conservé
- *Basic exception safety* : pas de fuite, invariants conservés
- *No exception safety* : aucune garantie

Erreurs – Exception safety

Do

- Privilégiez les garanties les plus fortes possibles

Don't

- Évitez la garantie faible
- Évitez absolument le *No exception safety*

Erreurs – Exception safety

Do

- Utilisez l'idiome *copy-and-swap* pour la *Strong exception safety*

```
class A {  
public:  
    A(const A&);  
    A& operator=(A);  
    friend void swap(A& lhs, A& rhs); // Nothrow  
};  
  
A& A::operator=(A other) {           // Copy  
    swap(*this, other);              // Swap  
    return *this;  
}
```

Erreurs – Exceptions et bonnes pratiques

Do

- *Throw by value, catch by const reference* ([C++ Coding Standards] chap. 73)

Do

- Utilisez des types dédiés héritant de `std::exception`
- Définissez des hiérarchies d'exceptions

Do

- Capturez uniquement là où vous savez traiter l'erreur

Erreurs – Exceptions et bonnes pratiques

Don't

- N'utilisez jamais les exceptions pour contrôler le flux d'exécution
- Ni pour gérer les « échecs attendus »
- Réservez les exceptions au signalement d'erreurs

Erreurs – assert

- Arrête le programme si l'expression est évalué à 0
- Affiche au moins l'expression, le fichier et la ligne

```
assert(expression);
```

- Sans effet lorsque `NDEBUG` est défini
 - Coût nul en *Release*
 - Inutilisable pour les erreurs d'exécution et le contrôle des entrées

Objectifs

- Traquer les erreurs de programmation et les violations de contrat interne

Erreurs – Conclusion

Do

- Utilisez exceptions et codes retour pour les erreurs d'exécution et la vérification des données externes
- Réservez `assert` aux erreurs de programmation et à la vérification des contrats internes

Do

- Préférez les exceptions aux codes retour ([C++ Coding Standards] chap. 72)

Don't

- Jamais d'`assert` pour les erreurs d'exécution et le contrôle des entrées

Ressources – Gestion manuelle

Comment gérer les erreurs ?

- Solution C : *Single Entry Single Exit*, bloc unique de libération

```
char* memory = malloc(50);  
if(!memory) goto err;  
...  
err:  
free(memory);
```

- Laborieux
- Difficile à mettre en place en présence d'exceptions

Ressources – Gestion manuelle

Quiz : Comment éviter les fuites mémoires ?

```
char* memory1 = NULL;  
char* memory2 = NULL;  
...  
memory1 = new char[50];  
...  
memory2 = new char[200];  
...  
delete[] memory1;  
delete[] memory2;
```

Ressources – Gestion manuelle

Comment copier des classes possédant des ressources ?

- Constructeurs et opérateurs générés copient les adresses des pointeurs
- Une double libération est une erreur

```
struct Foo {  
public:  
    Foo() : bar(new char[50]) {}  
    ~Foo() { delete[] bar; }  
  
private:  
    char* bar;  
};
```

Ressources – Gestion manuelle et bonnes pratiques

Do

Si une classe manipule une ressource brute, elle doit

- Soit définir constructeur de copie et opérateur d'affectation
- Soit les déclarer privés sans les définir (classe non copiable)

Big Rule of three

Si vous devez définir le constructeur de copie, l'opérateur d'affectation ou le destructeur, alors vous devriez définir les trois

Ressources – RAI

- Acquisition des ressources lors de l'initialisation de l'objet
- Libération automatique lors de sa destruction
- Propriété intrinsèque des objets par design
- Fonctionnement de la bibliothèque standard (conteneurs, fichiers, ...)
- Conséquences
 - Objets créés dans un état cohérent, testable et utilisable
 - Ressources automatiquement libérées à la destruction de l'objet
 - Capsules RAI copiables sans effort

Do

- Utilisez RAI pour vos objets

Ressources – RAI

Do

- Faites des constructeurs qui construisent des objets
 - Cohérents
 - Utilisables
 - Complètement initialisés

Don't

- Évitez les couples constructeur vide et fonction d'initialisation
- Évitez les couples constructeur vide et ensemble de mutateurs

Ressources – Limites du RAII

Gestion des erreurs

- Pas d'erreur ni d'exception dans les destructeurs
- La libération peut échouer (p.ex. `flush()` lors de la fermeture de fichier)

```
{  
    ifstream src("input.txt");  
    ofstream dst("output.txt");  
    copy_files(src, dst);  
}  
  
remove_file(src); // Potentielle perte de donnees
```

Ressources – Limites du RAI

`std::auto_ptr`

- Copiable
- La copie transfère la responsabilité de la ressource

```
void foo(auto_ptr<int> bar) {}
```

```
auto_ptr<int> bar(new int(5));
```

```
foo(bar);
```

```
cout << *bar << "\n"; // Erreur : bar n'est plus utilisable
```

Ressources – Loi de Déméter

- Principe de connaissance minimale
- Un objet A peut utiliser les services d'un deuxième objet B
- ... mais ne doit pas utiliser B pour accéder à un troisième objet
- En particulier, une classe n'expose pas ses données

Exceptions

- Agrégats et conteneurs dont le rôle est de contenir des données

Objectifs

- Mise en place du RAII
- Meilleure encapsulation
- Respect des *patterns* SOLID et GRASP
- Meilleure lisibilité, maintenabilité et réutilisabilité

Ressources – Loi de Déméter

Do, agrégats

- Préférez les structures aux classes
- Laissez les membres publics
- Fournissez, éventuellement, des constructeurs initialisant les données

Do, conteneurs

- Respectez l'interface et la logique des conteneurs standards

Do, classes de service

- Exposez des services, pas des données
- Pas de données publiques
- Limitez les accesseurs et les mutateurs

Ressources – Loi de Déméter

Conseil

- N'hésitez pas à étendre l'interface de classe avec des fonctions libres
- Pensez à l'amitié pour cette interface étendue
- Implémentez-la en terme de fonctions membres (p.ex. + à partir de +=)

```
class Foo {  
public:  
    Foo& operator+=(const Foo& other);  
};  
  
Foo operator+(Foo lhs, const Foo& rhs) {  
    return lhs += rhs;  
}
```

Ressources – Et le Garbage Collector ?

- Pas de GC dans le langage ni dans la bibliothèque standard
- Au moins un GC en bibliothèque tierce (Hans Boehm)
- ... mais limité par manque de support par le langage
- Non déterministe : adapté à la mémoire pas aux autres ressources
- Particulièrement adapté à la gestion des structures cycliques
- D'autres avantages pour la mémoire (compactage, recyclage, ...)

Wait and see

- Un complément à RAII, pas un concurrent ni un remplaçant
- Indisponible à ce jour

Ressources – Conclusion

Do, RAII

- Préférez les classes RAII de la bibliothèque standard aux ressources brutes
- Encapsulez les ressources dans des capsules RAII standards
- Concevez vos classes en respectant le RAII

Do, Déméter

- Respectez Déméter

Ressources – Conclusion

Don't

- Pas de `delete` dans le code applicatif

Attention

- Sous Linux, méfiez-vous de l'*Optimistic Memory Allocator*
- Pensez à paramétrer correctement l'OS

STL – Standard Template Library

- Partie de la bibliothèque standard comprenant
 - Conteneurs et `std::basic_string`
 - Itérateurs
 - Algorithmes manipulation les données des conteneurs via les itérateurs

Note

- Quelques algorithmes manipulant directement des données (p.ex. `std::min()`)
- Conçue initialement par Alexander Stepanov
 - Promoteur de la programmation générique
 - Sceptique vis à vis de la POO
- Basée sur les templates, pas sur la POO

STL – Standard Template Library

Intérêts

- n conteneurs et m algorithmes, seulement m implémentations
- Tout nouvel algorithme est disponible sur tous les conteneurs compatibles
- Tout nouveau conteneur bénéficie de tous les algorithmes compatibles
- Changement de conteneur à effort réduit

Pour aller plus loin

- [Effective STL] de Scott Meyers

STL – Standard Template Library

À nuancer

Algorithmes membres sur certains conteneurs

- Accès par itérateurs insuffisant (p.ex. `std::list`)
- Habitudes et historiques (p.ex. `std::string`)
- Performances (p.ex. `map.find()`)

STL Conteneurs – Généralités

- Contiennent des objets copiables et non constants
- ... qui peuvent être les adresses d'autres objets

Conteneurs de pointeurs

- Pas de libération automatique des objets pointés
- ... accessibles via un itérateur
- Fourniture possible d'une politique d'allocation
- Vu des algorithmes, ce qui fournit une paire d'itérateurs, est un conteneur

STL Conteneurs – Conteneurs séquentiels

- `std::vector`
 - Tableau de taille variable d'éléments contigus
 - Accès indexé
 - Croissance en temps amorti
 - Modifications en fin de vecteur (couteux ailleurs)
 - Compatible avec l'organisation mémoire des tableaux C

`std::vector<bool>` n'est pas un vecteur de booléen

- Ne remplit pas tous les pré-requis des conteneurs
- `operator[]` ne retourne pas le booléen mais un *proxy* vers celui-ci
- Voir [Effective STL] item 18

STL Conteneurs – Conteneurs séquentiels

- `std::list`
 - Liste doublement chaînée
 - Accès bidirectionnel non indexé
 - Modification n'importe où à faible coût
 - Plusieurs algorithmes membres (tri, fusion, suppression, ...)
- `std::deque`
 - *Double-ended queue*
 - Proche de `std::vector` mais extensible aux deux extrémités
 - Accès indexé
 - Éléments non nécessairement contigus
 - Non compatible avec l'organisation mémoire des tableaux C

STL Conteneurs – Conteneurs séquentiels

- `std::string`
 - Alias de `std::basic_string<char>`
 - Stockage de chaînes de caractères
 - Manipulation de *bytes* et non de caractères encodés

`std::string` et UTF-8

- `length()` et `size()` retournent le nombre de *bytes*, pas de caractères
 - Contiguïté non garantie, mais respectée en pratique
 - Variante `std::wstring` pour les caractères larges

API trop riche

- De nombreuses fonctions membres qui gagneraient à être libres et génériques
- Voir GotW #84 : Monoliths "Unstrung"

STL Conteneurs – Conteneurs associatifs

- Quatre saveurs
 - `std::map` – clés-valeurs, ordonné par la clé, unicité des clés
 - `std::multimap` – clés-valeurs, ordonné par la clé, multiplicité des clés
 - `std::set` – valeurs ordonnées et uniques
 - `std::multiset` – valeurs ordonnées et non-uniques

Implémentation

- Pas des tables de hachage
- Généralement des arbres binaires de recherche balancés
- Critère d'ordre configurable (strictement inférieur par défaut)

Attention

- Ordre strict
- Algorithmes membres (recherche) pour les performances

STL Conteneurs – Adaptateurs

- Basés sur un autre conteneur pour proposer une API simplifiée
- Avantages et inconvénients du conteneur sous-jacent
- `std::stack`
 - Pile LIFO
 - Basée sur `std::vector`, `std::list` ou `std::deque`
- `std::queue`
 - File FIFO
 - Basée sur `std::deque` ou `std::list`
- `std::priority_queue`
 - File dont l'élément de tête est le plus grand
 - Basée sur `std::vector` ou `std::deque`
 - Critère d'ordre configurable (strictement inférieur par défaut)

STL Conteneurs – Adaptateurs

```
stack<int, vector<int> > foo;
for(int i=0; i<5; ++i) foo.push(i);

// Affiche 4 3 2 1 0
while(!foo.empty()) {
    cout << ' ' << foo.top();
    foo.pop();
}
```



STL Conteneurs – conteneurs non-STL

- `std::bitset`
 - Tableau de bits de taille fixe
 - Conçu pour réduire l'empreinte mémoire
 - Pas d'itérateur ni d'interface STL

`std::bitset` VS. `std::vector<bool>`

Objectif de gain mémoire adressé par `std::bitset`, pourquoi `std::vector<bool>` n'est-il pas un vrai conteneur de booléen ?

- Conteneurs non-standards
 - Listes simplement chaînées
 - Tables de hachage
 - Tableaux de taille fixe
 - Tampons circulaires
 - Arbres et graphes
 - Variantes de conteneurs STL

STL Conteneurs – `std::pair`

- Couple de deux valeurs
- Pas un conteneur
 - Type de retour de la recherche sur les `std::map` (couple clé-valeur)
 - Candidat pour construire des vecteurs indexés par un non-numérique
- `std::make_pair` construit une paire

STL Conteneurs – Choix du conteneur

Do, par défaut

- `std::string` pour les chaînes de caractères
- `std::vector`

Do, performances

- Mesurez avec des données réelles sur la configuration cible

Flux d'octets

- Utilisez `std::vector<unsigned char>`
- Pas `std::vector<char>` encore moins `std::string`

STL Conteneurs – Choix du conteneur

Conseils

- Voir [Effective STL] item 1
- Voir Which C++ Standard Sequence Container should I use?
- Pensez à `reserve()`
- Une insertion en vrac suivie d'un tri peut être plus efficace qu'une insertion en place
- Un vecteur de paires peut être un bon choix pour un ensemble de clés-valeurs

STL Itérateurs – Généralités

- Abstraction permettant le parcours des collections d'objets
- Interaction entre conteneurs et algorithmes
- Interface similaire à celle d'un pointeur
- Quatre types
 - `iterator` et `const_iterator`
 - `reverse_iterator` et `const_reverse_iterator`
- Itérateurs sur un conteneur : `begin()` et `end()`
- Itérateurs inverses sur un conteneur : `rbegin()` et `rend()`
- Les itérateurs d'une paire doivent appartenir au même conteneur

Itérateurs de fin

- Pointent un élément après le dernier
- Ne doivent pas être déréférencés ni incrémentés

STL Itérateurs – Catégories et opérations

- Opérations communes : copie, affectation et incrémentation
- Hiérarchie de cinq catégories
 - *Input* : égalité (`==` et `!=`) et lecture
 - *Output* : écriture
 - *Forward* : Parcours multiples
 - *Bidirectional* : décrémentation
 - *Random access*
 - Déplacement d'un nombre arbitraire (`+`, `-`, `+=`, `-=` et `[]`)
 - Comparaison (`<`, `<=`, `>`, `>=`)



Attention

Seules les versions mutables de *Forward*, *Bidirectional* et *Random access* itérateurs sont des *Output* itérateurs.

STL Itérateurs – Catégories et conteneurs

Conteneur	Catégorie
<code>std::vector</code>	<i>Random access</i>
<code>std::deque</code>	<i>Random access</i>
<code>std::list</code>	<i>Bidirectionnal</i>
<code>std::map</code> et <code>std::multimap</code>	<i>Bidirectionnal</i>
<code>std::set</code> et <code>std::multiset</code>	<i>Bidirectionnal</i>

STL Itérateurs – Itérateurs d'insertion

- Adaptateur d'itérateurs
- De type *Output*
- Insertion de nouveaux éléments
 - En queue : `back_insérer`
 - En tête : `front_insérer`
 - À la position courante : `insérer`

STL Algorithmes – Foncteurs

- Instances de classe définissant `operator()`

```
class LessThan {  
public:  
    explicit LessThan(int threshold) : m_threshold(threshold) {}  
    bool operator() (int value) { return value <= m_threshold; }  
  
private:  
    int const m_threshold;  
};  
  
LessThan func(10);  
cout << func(5) << "\n";    // 1
```

STL Algorithmes – Foncteurs

- Possèdent des données membres
- Foncteurs standards : `std::plus`, `std::minus`, `std::equal`, `std::less`, ...
- Constructibles
 - Depuis des pointeurs de fonctions : `std::prt_fun`
 - Depuis des fonctions membres : `std::mem_fun`, `std::mem_fun1`, ...
 - En niant d'autres foncteurs : `std::not1`, `std::not2`
 - En fixant des paramètres : `std::bind1st`, `std::bind2nd`

STL Algorithmes – Prédicats

- Appelables retournant un booléen (ou un type convertible en booléen)
- Utilisés par de nombreux algorithmes
- De nombreux algorithmes utilisent un prédicat par défaut (p.ex. `<` ou `==`)

STL Algorithmes – Parcours

- `std::for_each()` parcourt un ensemble d'éléments
- ... et applique un traitement à chaque élément

```
void print(int i) { cout << i << ' '; }
```

```
vector<int> foo{4, 5, 9, 12};  
for_each(foo.begin(), foo.end(), print);
```

- Version du *map/apply* fonctionnel

STL Algorithmes – Parcours

- Retourne le foncteur passé en paramètre

```
struct Aggregate {  
    Aggregate() : m_sum(0) {}  
    void operator()(int i) { m_sum += i; }  
    int m_sum;  
};  
  
vector<int> foo{4, 5, 9, 12};  
for_each(foo.begin(), foo.end(), Aggregate()).m_sum; // 30
```

- Candidat pour le *fold/reduce* fonctionnel
- Pas de sémantique, faible utilité

STL Algorithmes – Recherche linéaire

- `std::find()` recherche une valeur
- ... et retourne un itérateur sur celle-ci
- ... ou l'itérateur de fin si la valeur n'est pas présente

```
vector<int> foo{4, 5, 9 ,12};  
vector<int>::iterator it1;  
vector<int>::iterator it2
```

```
it1 = find(foo.begin(), foo.end(), 5);    // it1 pointe sur foo[1]  
it2 = find(foo.begin(), foo.end(), 19);   // Et it2 sur foo.end()
```

STL Algorithmes – Recherche linéaire

- `std::find_if()` recherche depuis un prédicat

Variantes `_if`

- Les algorithmes suffixés par `_if` utilisent un prédicat
- `std::find_first_of()` recherche la première occurrence d'un élément
- `std::search()` recherche la première occurrence d'un sous-ensemble
- `std::find_end()` recherche la dernière occurrence d'un sous-ensemble
- `std::adjacent_find()` recherche deux éléments consécutifs égaux
- `std::search_n()` recherche la première suite de n éléments consécutifs égaux à une valeur



STL Algorithmes – Recherche dichotomique

- Pré-requis : ensemble trié
- `std::lower_bound()` retourne un itérateur sur le premier élément non strictement inférieur à la valeur recherchée
- ... et l'itérateur de fin si un tel élément n'existe pas

```
vector<int> foo{4, 5, 7, 9, 12};  
  
*lower_bound(foo.begin(), foo.end(), 6); // 7  
*lower_bound(foo.begin(), foo.end(), 9); // 9
```

STL Algorithmes – Recherche dichotomique

- `std::upper_bound()` retourne un itérateur sur le premier élément strictement supérieur à la valeur recherchée
- `std::equal_range()` retourne la paire (`std::lower_bound`, `std::upper_bound`)

Attention

- Le résultat retourné peut ne pas être la valeur recherchée
- `std::binary_search()` indique si l'élément cherché est présent

STL Algorithmes – Recherche dichotomique

Attention

- Pas de fonction de recherche dichotomique retournant l'élément cherché

```
vector<int>::iterator foo(vector<int> vec, int val) {  
    vector<int>::iterator it = lower_bound(vec.begin(), vec.end(), val);  
    if(it != vec.end() && *it == val) return it;  
    else return vec.end();  
}
```

```
vector<int> bar{1, 5, 8, 13, 25, 42};  
foo(bar, 12); // vec.end  
foo(bar, 13); // itérateur sur 13
```



STL Algorithmes – Comptage

- `std::count()` compte le nombre d'éléments égaux à la valeur fournie

```
vector<int> foo{4, 5, 3, 9, 5, 5 ,12};  
  
count(foo.begin(), foo.end(), 5); // 3  
count(foo.begin(), foo.end(), 2); // 0
```

- `std::count_if()` compte le nombre d'éléments satisfaisant le prédicat



STL Algorithmes – Comparaison

- `std::equal()` teste l'égalité de deux ensembles (valeur et position)

```
vector<int> foo{4, 5, 9, 12};  
vector<int> bar{4, 5, 12, 9};  
  
equal(foo.begin(), foo.end(), foo.begin()); // true  
equal(foo.begin(), foo.end(), bar.begin()); // false
```

STL Algorithmes – Comparaison

Attention

- `std::equal()` ne vérifie pas les tailles des deux ensembles

Et `operator==` ?

- `operator==` sur des conteneurs teste la taille et le contenu

Do

- Préférez `operator==` à `std::equal()` pour comparer un conteneur complet

STL Algorithmes – Comparaison

- `std::mismatch()` retourne une paire d'itérateurs sur les premiers éléments différents

```
vector<int> foo{4, 5, 9, 13};  
vector<int> var{4, 5, 12, 8};  
  
mismatch(foo.begin(), foo.end(), bar.begin()); // 9 12
```

- ... ou l'itérateur de fin en cas d'égalité



STL Algorithmes – Remplissage

- `std::fill()` remplit l'ensemble avec la valeur fournie

```
vector<int> foo(4);  
  
fill(foo.begin(), foo.end(), 12);  // 12 12 12 12
```

- `std::fill_n()` idem avec un ensemble défini par sa taille

Constructeur

- Remplissage des conteneurs séquentiels à la construction

```
vector<int> foo(4, 12);
```

STL Algorithmes – Remplissage

- `std::generate()` valorise les éléments à partir d'un générateur

```
int gen() {  
    static int i = 0;  
    i += 5;  
    return i;  
}
```

```
vector<int> foo(4);  
generate(foo.begin(), foo.end(), gen); // 5 10 15 20
```

- `std::generate_n()` idem avec un ensemble défini par sa taille



STL Algorithmes – Copie

- `std::copy()` copie les éléments du début vers la fin

```
vector<int> foo{4, 5, 9, 12};  
vector<int> bar;  
  
copy(foo.begin(), foo.end(), back_inserter(bar));
```

- `std::copy_backward()` copie les éléments de la fin vers le début

Attention

- À la taille du second ensemble
- Aux ensembles non-disjoints



STL Algorithmes – Échange

- `std::swap()` échange deux objets

```
int x=10, y=20;    // x:10 y:20  
swap(x,y);        // x:20 y:10
```

- `std::swap_ranges()` échange des éléments de deux ensembles

```
vector<int> foo (5,10); // foo: 10 10 10 10 10  
vector<int> bar (5,33); // bar: 33 33 33 33 33  
  
swap_ranges(foo.begin()+1, foo.end()-1, bar.begin());  
// foo : 10 33 33 33 10  
// bar : 10 10 10 33 33
```

- `std::iter_swap()` échange deux objets pointés par des itérateurs



STL Algorithmes – Remplacement

- `std::replace()` remplace toutes les occurrences d'une valeur par une autre

```
vector<int> foo{4, 5, 7, 9 ,12, 5};
```

```
replace(foo.begin(), foo.end(), 5, 8); // 4 8 7 9 12 8
```

- `std::replace_if()` remplace toutes les éléments vérifiant le prédicat par une valeur donnée

STL Algorithmes – Remplacement

- `std::replace_copy()` copie les éléments d'un ensemble en remplaçant toutes les occurrences d'une valeur par une autre

Variantes `_copy`

- Les algorithmes suffixés par `_copy` fonctionnent comme l'algorithme de base en troquant la modification en place contre une copie du résultat
- `std::replace_copy_if()` copie les éléments d'un ensemble en remplaçant toutes les éléments vérifiant le prédicat par une valeur donnée



STL Algorithmes – Suppression

- `std::remove()` élimine les éléments égaux à une valeur donnée

```
vector<int> foo{4, 5, 5, 5, 7, 9, 9, 5};  
  
remove(foo.begin(), foo.end(), 5);    // 4 7 9 9 ...
```

Pas de suppression

- Ramène les éléments à conserver vers le début de l'ensemble
- Retourne l'itérateur correspond à la nouvelle fin

Idiome *Erase-Remove*

Suppression via un appel à `erase()` sur les éléments après le nouvel itérateur de fin

```
foo.erase(remove(foo.begin(), foo.end(), 5), foo.end());
```


STL Algorithmes – Suppression

- `std::remove_if()` élimine les éléments vérifiant le prédicat
- `std::remove_copy()` copie les éléments différents d'une valeur donnée
- `std::remove_copy_if()` copie les éléments ne vérifiant pas le prédicat



STL Algorithmes – Suppression des doublons

- `std::unique()` élimine les éléments consécutifs égaux sauf le premier

```
vector<int> foo{4, 5, 5, 5, 7, 9, 9, 5};
```

```
unique(foo.begin(), foo.end()); // 4 5 7 9 5 ...
```

- `std::unique_copy()` copie l'ensemble en ne conservant que le premier des éléments consécutifs égaux



STL Algorithmes – Transformation

- `std::transform()` applique une transformation aux éléments d'un ensemble

```
int double_val(int i) { return 2 * i;}

vector<int> foo{4, 5, 7, 9};
vector<int> bar(4);
transform(foo.begin(), foo.end(), bar.begin(), double_val);
// 8 10 14 18
```

STL Algorithmes – Transformation

- Ou de deux ensembles en stockant le résultat dans un troisième

```
vector<int> foo{4, 5, 7, 9};  
vector<int> bar{2, 3, 6, 1};  
vector<int> baz(4);  
  
transform(foo.begin(), foo.end(), bar.begin(),  
          baz.begin(), plus<int>());  
// 6 8 13 10
```



STL Algorithmes – Rotation

- `std::rotate()` effectue une rotation de l'ensemble, le nouveau début étant fourni par un itérateur

```
vector<int> foo{4, 5, 7, 9, 12};  
  
rotate(foo.begin(), foo.begin() + 2, foo.end()); // 7 9 12 4 5
```

- `std::rotate_copy()` effectue une rotation et copie le résultat



STL Algorithmes – Partitionnement

- `std::partition()` réordonne l'ensemble pour que les éléments vérifiant le prédicat soit avant ceux ne le vérifiant pas ...

```
bool is_odd(int i) { return (i % 2) == 1; }  
vector<int> foo{4, 13, 28, 9 , 54};  
  
partition(foo.begin(), foo.end(), is_odd);  
// 9 13 28 4 54 ou 9 13 4 28 54 ou ...)
```

- ... et retourne un itérateur sur le début de la seconde partie

Attention

- Ordre relatif non conservé

STL Algorithmes – Partitionnement

- `std::stable_partition()` partitionne en conservant l'ordre relatif

```
vector<int> foo{4, 13, 28, 9 , 54};
```

```
stable_partition(foo.begin(), foo.end(), is_odd); // 13 9 4 28 54
```

Deux fonctions ?

- Stabilité coûteuse en temps et pas toujours nécessaire

STL Algorithmes – Partitionnement

- `std::nth_element()` réordonne les éléments
 - Élément sur l'itérateur pivot est celui qui serait à cette place si l'ensemble était trié
 - Éléments avant ne sont pas supérieurs
 - Éléments après ne sont pas inférieurs
 - Pas d'ordre particulier au sein des deux sous-ensembles

```
vector<int> foo{9, 8, 7, 6, 5, 4, 3, 2, 1};  
  
nth_element(foo.begin(), foo.begin() + 3, foo.end());  
// 2 1 3 4 5 9 6 7 8
```



STL Algorithmes – Tri

- `std::sort()` trie un ensemble

```
vector<int> foo{4, 13, 28, 9 , 54};
```

```
sort(foo.begin(), foo.end()); // 4 9 13 28 54
```

Attention

- Ordre relatif non conservé
- `std::stable_sort()` trie l'ensemble en conservant l'ordre relatif

STL Algorithmes – Tri

- `std::partial_sort()` réordonne l'ensemble de manière à ce que les éléments situés avant un itérateur pivot soient les plus petits éléments de l'ensemble ordonnés par ordre croissant ...

```
vector<int> foo{9, 8, 7, 6, 5, 4, 3, 2, 1};  
  
partial_sort(foo.begin(), foo.begin() + 3, foo.end());  
// 1 2 3 9 8 7 6 5 4
```

- ... les autres éléments n'ont pas d'ordre particulier
- `std::partial_sort_copy()` copie l'ensemble ordonné à l'image de `std::partial_sort()`



STL Algorithmes – Mélange

- `std::random_shuffle()` réordonne aléatoirement l'ensemble

```
vector<int> foo{9, 8, 7, 6, 5, 4, 3, 2, 1};  
  
random_shuffle(foo.begin(), foo.end());  
// 1 8 3 7 9 4 2 6 5  
// ou ...
```



STL Algorithmes – Fusion

- `std::merge()` fusionne deux ensembles triés dans un troisième

```
vector<int> foo{1, 5, 6, 8};  
vector<int> bar{2, 5};  
vector<int> baz;  
  
merge(foo.begin(), foo.end(), bar.begin(), bar.end(),  
      back_inserter(baz));    // 1 2 5 5 6 8
```

- `std::inplace_merge()` fusionne deux sous-ensembles sur place



STL Algorithmes – Opérations ensemblistes

Attention

- Ensembles sans répétition de valeur
- Ensembles triés
- `std::includes()` vérifie si tous les éléments sont présents dans un autre ensemble

```
vector<int> foo{1, 5, 6, 8};  
vector<int> bar{2, 5};  
vector<int> baz{1, 6};
```

```
includes(foo.begin(), foo.end(), bar.begin(), bar.end()); // faux  
includes(foo.begin(), foo.end(), baz.begin(), baz.end()); // vrai
```

STL Algorithmes – Opérations ensemblistes

- `std::set_union()` : union de deux ensembles

```
vector<int> foo{1, 5, 6, 8};  
vector<int> bar{2, 5};  
vector<int> baz;  
  
set_union(foo.begin(), foo.end(), bar.begin(),  
          bar.end(), back_inserter(baz)); // 1 2 5 6 8
```

- `std::set_intersection()` : intersection de deux ensembles
- `std::set_difference()` : différence de deux ensembles
- `std::set_symmetric_difference()` : différence symétrique de deux ensembles



STL Algorithmes – Gestion de tas

Tas

- Structure permettant la récupération de l'élément de plus grande valeur
- `std::make_heap()` forme un tas depuis un ensemble
- `std::pop_heap()` déplace l'élément de plus haute valeur en fin d'ensemble
- `std::push_heap()` ajoute l'élément en fin d'ensemble au tas

Structure

- `std::pop_heap()` et `std::push_heap()` maintiennent la structure de tas
- `std::sort_heap()` tri le tas



STL Algorithmes – Min-max

- `std::min()` détermine le minimum de deux éléments
- `std::max()` détermine le maximum de deux éléments

```
min(52, 6); // 6  
max(52, 6); // 52
```

- `std::min_element()` détermine le plus petit élément d'un ensemble

```
vector<int> foo{18, 5, 6, 8};  
  
min_element(foo.begin(), foo.end()); // Sur 5
```

- `std::max_element()` détermine le plus grand élément d'un ensemble



STL Algorithmes – Numérique

- `std::accumulate()` « ajoute » tous les éléments de l'ensemble

```
vector<int> foo{18, 5, 6, 8};
```

```
accumulate(foo.begin(), foo.end(), 1, multiplies<int>()); // 4320
```

- Opérateur et valeur initiale configurables
- *Reduce/fold* fonctionnel

STL Algorithmes – Numérique

- `std::adjacent_difference()` « différence » entre chaque élément et son prédécesseur

```
vector<int> foo{18, 5, 6, 8};  
vector<int> bar;  
  
adjacent_difference(foo.begin(), foo.end(),  
                    back_inserter(bar), minus<int>()); // 18 -13 1 2
```

- Opérateur configurable

STL Algorithmes – Numérique

- `std::inner_product()` : « produit scalaire » de deux ensembles

```
vector<int> foo{1, 2, 3, 4};  
vector<int> bar{2, 3, 4, 5};  
  
inner_product(foo.begin(), foo.end(), bar.begin(), 0); // 40
```

- Opérateurs et valeur configurables

STL Algorithmes – Numérique

- `std::partial_sum()` : « somme » partielle d'un ensemble
- Chaque élément résultant est la somme des éléments d'indice inférieur ou égal de l'ensemble de départ

```
vector<int> foo{1, 2, 3, 4};  
vector<int> bar;  
  
partial_sum(foo.begin(), foo.end(), back_inserter(bar)); // 1 3 6 10
```

- Opérateur configurable



STL Algorithmes – Au delà des conteneurs

- Itérateurs définissables hors des conteneurs
 - Abstraction du parcours
 - Sémantique de pointeurs
- Algorithmes indépendants du conteneur
- Utilisables sur d'autres ensembles de données

STL Algorithmes – Au delà des conteneurs

- Tableaux C
 - Pas un conteneur
 - Sémantique : Tableau ou pointeur ? Statique ou dynamique ?
 - Service : Taille ? Copie ?
 - Simple pointeur comme itérateur
 - Début : adresse du premier élément
 - Fin : adresse suivant le dernier élément

```
int foo[4];
```

```
fill(foo, foo + 4, 5); // 5 5 5 5
```

STL Algorithmes – Au delà des conteneurs

- Flux

- `istream_iterator` : *input* itérateur
 - Début : depuis un flux entrant
 - Fin : constructeur par défaut
- `ostream_iterator` : *output* itérateur
 - Depuis un flux sortant, séparateur configurable

```
vector<int> foo{5, 6, 12, 89};  
ostream_iterator<int> out_it (cout, ", ");  
  
copy(foo.begin(), foo.end(), out_it); // 5, 6, 12, 89,
```

Attention

- Séparateur ajouté après chaque élément, y compris le dernier
- Buffers de flux : `istreambuf_iterator` et `ostreambuf_iterator`



STL Conclusion

Do

- Préférez les conteneurs aux tableaux C

Attention

- `operator[]` ne vérifie pas les bornes

Don't

- N'utilisez pas d'itérateur invalidé

Attention

- Pas objets polymorphiques dans les conteneurs
- Ou via des pointeurs intelligents

STL Conclusion

Do, performances

- Mesurez !

Conseils, performances

- Réfléchissez à votre utilisation des données
- Méfiez-vous des complexités brutes

Do

- Préférez les algorithmes standards aux algorithmes tierces et « maisons »

Bémol, performances

- Algorithmes standards généralement très bons
- Mais pas forcément optimaux dans une situation particulière

STL Conclusion

Do

- Faites vos propres algorithmes plutôt que des boucles
- Faites des algorithmes génériques et compatibles

Do, sémantique

- Le bon algorithme pour la bonne opération
- Définissez la sémantique de vos algorithmes et choisissez un nom explicite

Do

- Préférez les prédicats purs

STL Conclusion

Do

- Vérifiez que les ensembles de destination aient une taille suffisante

Do

- Vérifiez les pré-conditions des algorithmes (p.ex. ensemble trié)
- Vérifiez le type d'itérateur requis
- Vérifiez les complexités garanties

Aller plus loin

- Voir STL Algorithms (Marshall Clow)

Sommaire

- 1 Retour sur C++98/C++03
- 2 **C++11**
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Approuvé le 12 août 2011
- Dernier *Working Draft* : N3337
- Standardisation laborieuse
 - Sortie tardive (C++0x)
 - Périmètre initial trop ambitieux (retrait des concepts en 2009)
- Changement de fonctionnement du comité
 - Utilisation de *Technical Specification* et de groupes de travail dédiés
 - Pilotage par les dates et pas les fonctionnalités
 - Des versions fréquentes (3 ans : 2011, 2014, 2017, 2020, ...)
 - Voir Trip report: Winter ISO C++ standards meeting
- Support quasi-complet par GCC, Clang et Visual C++
- Objectifs : sûreté, simplicité, rapidité et meilleure détection d'erreur en *compile-time*

Dépréciations et suppressions

- Dépréciation de `register`

Export templates

- Suppression des *export templates*
- `export` reste un mot-clé réservé

Compatibilité

- Rupture de comptabilité ascendante
- Implémenté sur un unique compilateur et inutilisé en pratique

Motivations

- Voir N1426

Nouveaux types entiers

- Hérités de C99 (`cstdint` et `cinttypes`)

Depuis C99

- Ainsi que *variadic macro*, `__func__`, concaténation de chaînes littérales, ...
- `long long int` et `unsigned long long int`
 - Au moins aussi grand que `long int`
 - Plages garanties : $[-(2^{63} - 1), 2^{63} - 1]$ et $[0, 2^{64}]$
 - Extension de nombreux compilateurs bien avant C++11
- Types entiers le plus grand disponibles `intmax_t` et `uintmax_t`

Nouveaux types entiers

- Entiers de N bits `int<N>_t` et `uint<N>_t`
 - $N = 8, 16, 32$ ou 64
 - `int<N>_t` obligatoirement en complément à 2
 - Pas de bit de *padding*
 - Support optionnel
- Plus petits entiers d'au moins N bits `int_least<N>_t` et `uint_least<N>_t`
- Plus rapides entiers d'au moins N bits `int_fast<N>_t` et `uint_fast<N>_t`
- Entiers capables de contenir une adresse `intptr_t` et `uintptr_t`
 - Convertibles en `void*` avec une valeur égale au pointeur original
 - Support optionnel

Nouveaux types entiers

- Macros de définition des plages correspondantes
- Macros de construction depuis des entiers classiques
- Macros des spécificateurs pour `printf()` et `scanf()`
- Fonctions de manipulation de `intmax_t` et `uintmax_t`
- Surcharges de `abs()` et `div()` pour `intmax_t` si nécessaire

POD Généralisé – Rappels

- Types POD (*Plain Old Data*) : classes et structures POD, unions POD, types scalaires et tableaux de ces types
- Certaines constructions permises uniquement sur les types POD
 - Utilisation de `memcpy()` ou `memmove()`
 - Utilisation de `goto` au-delà de la déclaration d'une variable
 - Utilisation de `reinterpret_cast`
 - Accès au début commun d'une union par un membre non actif
 - Utilisation des fonctions C `qsort()` ou `bsearch()`
 - ...

POD Généralisé – Classe agrégat

- C++98
 - Pas de constructeur déclaré par l'utilisateur
 - Pas de donnée membre non-statique privée ou protégée
 - Pas de classe de base
 - Pas de fonction virtuelle

POD Généralisé – Classe agrégat

- C++11
 - Pas de constructeur fourni par l'utilisateur
 - Pas d'initialisation *brace-or-equal-initializers* des données membres non-statiques
 - Pas de donnée membre non-statique privée ou protégée
 - Pas de classe de base
 - Pas de fonction virtuelle

POD Généralisé – Classe POD C++98

- Classe agrégat
- Sans donnée membre non-statique de type non-POD
- Sans référence
- Sans opérateur d'affectation défini par l'utilisateur
- Sans destructeur défini par l'utilisateur

POD Généralisé – Classe POD C++11

- Contraintes réparties en trois sous-notions
- *trivially copyable*
 - Pas de constructeur de copie ou de déplacement non triviaux
 - Pas d'opérateur d'affectation non trivial
 - Destructeur trivial

Trivial

- Pas fournie par l'utilisateur
- Pas de fonction virtuelle ni de classe de base virtuelle
- Opération des classes de bases et des membres non-statiques est triviale

Autre formulation

- Copie, déplacement, affectation et destruction générés implicitement
- Pas de fonction ni de classe de base virtuelle
- Classes de base et membres non-statiques *trivially copyable*

POD Généralisé – Classe POD C++11

- *trivial*
 - *trivially copyable*
 - Constructeur par défaut trivial
 - Pas fourni par l'utilisateur
 - Pas de fonction virtuelle ni de classe de base virtuelle
 - Constructeur par défaut des classes de base et des membres non-statiques trivial
 - Pas d'initialisation *brace-or-equal-initializers* des données membres non-statiques

POD Généralisé – Classe POD C++11

- *Standard-layout*

- Pas de donnée membre non-statique non-*Standard-layout*
- Pas de référence
- Pas de classe de base non-*Standard-layout*
- Pas de fonction virtuelle
- Pas de classe de base virtuelle
- Même accessibilité de toutes les données membres non-statique
- Données membres non-statiques dans une unique classe de l'arbre d'héritage
- Pas de classe de base du type de la première donnée membre non-statique

En résumé

- Organisation mémoire similaire aux structures C

POD Généralisé – Classe POD C++11

- POD
 - *trivial*
 - *standard layout*
 - Pas de donnée membre non-statique non-POD
- Traits correspondants
 - `std::is_trivial`
 - `std::is_trivially_copyable`
 - `std::is_standard_layout`

POD Généralisé – Objectifs

- Opérations POD accessibles à la sous-notion correspondante
- Relâchement et adaptation de certaines contraintes
 - Constructeurs ou destructeurs `=default` autorisés
 - Données membres non-statiques plus nécessairement publiques
 - Classes de base non virtuelles autorisées

POD Généralisé – Conséquences

- *standard layout*
 - Utilisation de `reinterpret_cast`
 - Utilisation de `offsetof`
 - Accès au début commun d'une union par un membre non actif
- *trivially copyable*
 - Utilisation de `memcpy()` ou `memmove()`
- *trivial*
 - Utilisation de `goto` au-delà de la déclaration d'une variable
 - Utilisation de `qsort()` ou `bsearch()`

Unions généralisées

- Constructeurs, opérateurs d'assignation ou destructeurs définis par l'utilisateur acceptés sur les types membres d'une union
- ... mais les fonctions équivalentes de l'union sont supprimées
- Toujours impossible d'utiliser des types avec des fonctions virtuelles, des références ou des classes de base

inline namespace

- Injection des déclarations du namespace imbriqué dans le namespace parent

```
namespace V1 { void foo() { cout << "V1\n"; } }  
  
inline namespace V2 { void foo() { cout << "V2\n"; } }  
  
V1::foo(); // Affiche V1  
V2::foo(); // Affiche V2  
foo();    // Affiche V2
```

Motivation

- Évolution de bibliothèque et conservation des versions précédentes



0 ou NULL ?

- C++ 98 : 0 ou **NULL**
- Cohabite mal avec les surcharges

0 ou NULL ?

- C++ 98 : 0 ou **NULL**
- Cohabite mal avec les surcharges

Quiz : Quelle surcharge est éligible ?

```
void foo(char*) { cout << "chaine\n"; }  
void foo(int) { cout << "entier\n"; }
```

```
foo(0);  
foo(NULL);
```


0 ou NULL ? nullptr !

- C++ 11 : `nullptr`
 - Unique pointeur du type `nullptr_t`
 - Conversion implicite de `nullptr_t` vers tout type de pointeur

```
void foo(char*) { cout << "chaine\n"; }  
void foo(int) { cout << "entier\n"; }
```

```
foo(0);           // Version int  
foo(nullptr);    // Version pointeur
```

Do

- Utilisez `nullptr` plutôt que 0 ou `NULL`



static_assert

- Assertion vérifiée à la compilation

```
static_assert(sizeof(int) == 3, "Taille incorrecte");  
// Erreur de compilation indiquant "Taille incorrecte"
```

Do

- Utilisez `static_assert` pour vérifier à la compilation ce qui peut l'être

Do

- Préférez les vérifications *compile-time* ou *link-time* aux vérifications *run-time*



constexpr

- Indique une expression constante
- Donc évaluable et utilisable à la compilation
- Implicitement `const`
- Fonctions `constexpr` implicitement `inline`
- Contenu des fonctions `constexpr` limité
 - `static_assert`
 - `typedef`
 - `using`
 - Exactement une expression `return`

```
constexpr int foo() { return 42; }
```

```
char bar[foo()];
```

constexpr

```
constexpr int foo() { return 42; }
```

```
int a = 42;  
switch(a) {  
    case foo():  
        break;  
  
    default:  
        break;  
}
```



constexpr

- Sous certaines conditions restrictives, `const` sur une variable est suffisant

```
const int a = 42;  
char bar[a];
```

Variable-Length Array

- Pas de rapport entre VLA et `constexpr`
- VLA est un mécanisme *run-time*

Do

- Déclarez `constexpr` les constantes et fonctions évaluable en *compile-time*

Extended sizeof

- `sizeof` sur des membres non statiques

```
struct Foo { int bar; };
```

```
// Valide en C++11, mal-forme en C++98/03
```

```
cout << sizeof(Foo::bar);
```

Note

- En pratique, cet exemple compile en mode C++98 sous GCC

Sémantique de déplacement

- Deux constats
 - Copie potentiellement coûteuse ou impossible
 - Copie inutile lorsque l'objet source est immédiatement détruit

Optimisation des copies

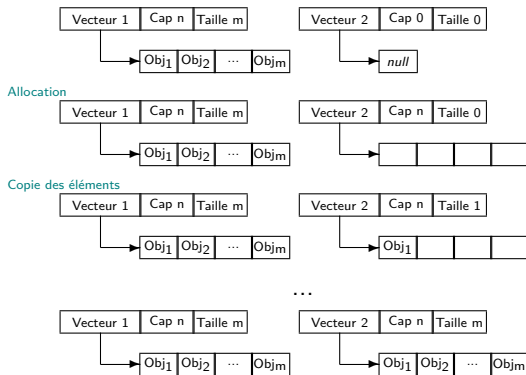
- Partiellement adressé en C++98/03 par l'élision de copie et (N)RVO
- Échange de données légères plutôt que copie profonde
- Déplacement seulement si
 - Type déplaçable
 - Instance sur le point d'être détruite ou explicitement déplacée

Attention

- Les données ne sont plus présentes dans l'objet initial

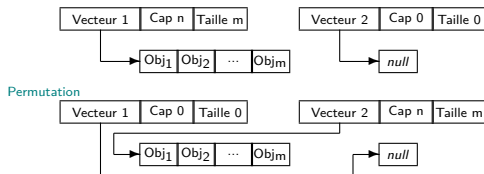
Sémantique de déplacement

• Copie



Sémantique de déplacement

- Déplacement



Sémantique de déplacement

- *rvalue reference*
 - Référence sur un objet temporaire ou sur le point d'être détruit
 - Noté par une double esperluette : `T&& value`
- Deux fonctions « de conversion »
 - `std::move()` convertit le paramètre en *rvalue*
 - `std::forward()` convertit le paramètre en *rvalue* s'il n'est pas une *lvalue reference*

rvalue, lvalue, ...?

- Voir N3337 §3.10

`std::forward()` ?

- *perfect forwarding* (Voir N1385)

Sémantique de déplacement

- Rendre une classe déplaçable
 - Constructeur par déplacement `T(const T&&)`
 - Opérateur d'affectation par déplacement `T& operator=(const T&&)`

Génération implicite

- Pas de constructeur par copie, d'opérateur d'affectation, de destructeur, ni l'autre déplacement *user-declared*

user-declared ? user-provided ?

- *user-declared* : la fonction est déclarée par l'utilisateur, y compris en `=default`
- *user-provided* : le corps de la fonction est fourni par l'utilisateur

Sémantique de déplacement

Rule of five

Si une classe déclare destructeur, constructeur par copie ou par déplacement, affectation par copie ou par déplacement, alors elle doit définir les cinq

Rule of zero

Lorsque c'est possible, n'en définissez aucune

Pour aller plus loin

- Voir *Élégance, style épuré et classe* (Loïc Joly)

Sémantique de déplacement

Dans la bibliothèque standard

- Nombreuses classes standards déplaçables (thread, flux, ...)
- Évolution de contraintes : déplaçable plutôt que copiable
- Implémentations utilisant le déplacement si possible

Initializer list

- Initialisation des conteneurs

```
vector<int> foo;  
foo.push_back(1);  
foo.push_back(56);  
foo.push_back(18);  
foo.push_back(3);
```

// Devient

```
vector<int> foo{1, 56, 18, 3};
```

Initializer list

- Classe `std::initializer_list` pour accéder aux valeurs de la liste

Accéder, pas contenir !

- `std::initializer_list` référence mais ne contient pas les valeurs
 - Valeurs contenues dans un tableau temporaire de même durée de vie
 - Copier un `std::initializer_list` ne copie pas les données
-
- Fonctions membres `size()`, `begin()`, `end()`
 - Construction automatique depuis une liste de valeurs entre accolades

Initializer list

- Constructeurs peuvent prendre un `std::initializer_list` en paramètre

```
MaClasse(initializer_list<value_type> itemList);
```

- Ainsi que toute autre fonction
- Intégré aux conteneurs de la bibliothèque standard



Initializer list

Do

- Préférez `std::initializer_list` aux insertions successives

Don't

- N'utilisez pas `std::initializer_list` pour copier ou transformer
- Utilisez les algorithmes et constructeurs idoines

Uniform Initialization

- Plusieurs types d'initialisation en C++98/03

```
int a = 2;  
int b(2);  
int c[] = {1, 2, 3};  
int d;
```

Uniform Initialization

- Mais aucune de générique

```
int a(2);           // Definition de l'entier a
int b();            // Declaration d'une fonction
int c(foo);         // ???
int d[] (1, 2);     // KO
```

```
int a[] = {1, 2, 3}; // OK

struct Foo { int a; };
Foo foo = {1};       // OK

vector<int> b = {1, 2, 3}; // KO
int c{8}             // KO
```

Uniform Initialization

- En C++ 11, l'initialisation via {} est générique

```
int a[] = {1, 2, 3};           // OK
Foo b = {5};                   // OK
vector<int> c = {1, 2, 3};     // OK
int d = {8};                   // OK
int e = {};
```

- Avec ou sans =

```
int a[]{1, 2, 3};              // OK
Foo b{5};                      // OK
vector<int> c{1, 2, 3};        // OK
int d{8};                      // OK
int e{};
```

Uniform Initialization

- Dans différents contextes

```
int* p = new int{4};  
long l = long{2};  
  
void f(int);  
f({2});
```

Uniform Initialization

Attention

- Pas de troncature avec {}

```
int foo{2.5}; // Erreur
```

Attention

- Si le constructeur par `std::initializer_list` existe, il est utilisé

```
vector<int> foo{2}; // 2  
vector<int> foo(2); // 0 0
```

Uniform Initialization

Contraintes sur l'initialisation d'agrégats

- Pas d'héritage
- Pas de constructeur fourni par l'utilisateur
- Pas d'initialisation *brace-or-equal-initializers*
- Pas de fonction virtuelle ni de membre non statique protégé ou privé

Do

- Préférez l'initialisation `{}` aux autres formes

auto

- Dédution (ou inférence) de type
- Type déduit de l'initialisation

Attention

- Inférence de type \neq typage dynamique
- Inférence de type \neq typage faible
- Typage dynamique \neq typage faible

Vocabulaire

- Statique : type porté par la variable et ne varie pas
- Dynamique : type porté par la valeur
- Absence : variable non typée, type imposé par l'opération

auto

- `auto` définit une variable dont le type est déduit

```
auto i = 2; // int
```

- Règles de déduction proches de celles des templates
- Listes entre accolades inférées comme des `std::initializer_list`

Attention

- Référence, `const` et `volatile` perdus durant la déduction

```
const int i = 2;  
auto j = i; // int
```

auto

- Combinaison possible avec `const`, `volatile` ou `&`

```
const auto i = 2;
```

```
int j = 3;
```

```
auto& k = j;
```

- Typier explicitement l'initialiseur permet de forcer le type déduit

```
// unsigned long
```

```
auto i = static_cast<unsigned long>(2);
```

```
auto j = 2UL
```

auto

- Tendance forte *Almost Always Auto* (AAA)

Pour aller plus loin

- Voir GotW 94 : AAA Style
- Plusieurs avantages
 - Variables forcément initialisées
 - Typage correct et précis
 - Garanties conservées au fil des corrections et refactoring
 - Généricité et simplification du code

auto

- Tendance forte *Almost Always Auto* (AAA)

Pour aller plus loin

- Voir GotW 94 : AAA Style
- Plusieurs avantages
 - Variables forcément initialisées
 - Typage correct et précis
 - Garanties conservées au fil des corrections et refactoring
 - Généricité et simplification du code

Quiz

- Type de retour de `std::list<std::string>::size()` ?

auto

- Limitations - solutions
 - Erreur de déduction - typage explicite de l'initialiseur
 - Initialisation impossible - `decltype`
 - Interfaces, rôles, contexte - concepts ?

Compatibilité

- `auto` présent en C++98/03 avec un sens radicalement différent

decltype

- Dédiction du type d'une variable ou d'une expression
- Permet donc la création d'une variable du même type

```
int a;  
long b;  
  
decltype(a) c;           // int  
decltype(a + b) d;       // long
```

- Généralement, déduction sans aucune modification du type
- Depuis une *lvalue* de type T autre qu'un nom de variable : T&

```
decltype( (a) ) e;       // int&
```

declval

- Utilisation de fonctions membres dans `decltype` sans instance
- Typiquement sur des templates acceptant des types sans constructeur commun mais avec une fonction membre commune

```
struct foo {  
    foo(const foo&) {}  
    int bar () const { return 1; }  
};  
  
decltype(foo().bar()) n2 = 5;           // Erreur  
decltype(std::declval<foo>().bar()) n2 = 5; // OK, int
```

Attention

- Uniquement dans des contextes non évalués

Dédution du type retour

- Combinaison de `auto` et `decltype`

```
auto add(int a, int b) -> decltype(a + b) {  
    return a + b;  
}
```

- Particulièrement utiles pour des fonctions templates

Quiz : T, U ou autre ?

```
template<typename T, typename U> ??? add(T a, U b) {  
    return a + b;  
}
```



Dédution du type retour

Solution

- Pas de bonne réponse en typage explicite
- Mais l'inférence de type vient à notre secours

```
template<typename T, typename U>  
auto add(T a, U b) -> decltype(a + b) {  
    return a + b;  
}
```

do

- Utilisez la déduction du type retour dans vos fonctions templates

std::array

- `std::array`
 - Tableau de taille fixe connue à la compilation
 - Éléments contigus
 - Accès indexé

```
array<int, 8> foo{2, 5, 9, 8, 2, 6, 8, 9};  
  
accumulate(foo.begin(), foo.end(), 0); // 49
```

```
array<int, 8> foo{2, 5, 9, 8, 2, 6, 8, 9, 17};  
// Erreur de compilation
```

std::array

- Vérification des index à la compilation

```
array<int, 8> foo{2, 5, 9, 8, 2, 6, 8, 9};  
  
get<2>(foo) << '\n'; // 9  
get<8>(foo) << '\n'; // Erreur de compilation
```



std::forward_list

- Liste simplement chaînée `std::forward_list`

```
forward_list<int> foo{2, 5, 9, 8, 2, 6, 8, 9, 12};  
  
accumulate(foo.begin(), foo.end(), 0); // 61
```



Conteneurs associatifs

- Conteneurs associatifs sous forme de tables de hachage
 - `std::unordered_map`
 - `std::unordered_multimap`
 - `std::unordered_set`
 - `std::unordered_multiset`
- Versions non ordonnées de `std::map`, `std::set`, ...

Unordered ?

- Nombreuses implémentations `hash_XXX` existantes
- Structures fondamentalement non ordonnées

shrink_to_fit()

- `shrink_to_fit()` réduit la capacité des `std::vector`, `std::deque` et `std::string` à leur taille

```
vector<int> foo{12, 25};
```

```
foo.reserve(15);      // Taille : 2, capacite : 15
```

```
foo.shrink_to_fit();  // Taille : 2, capacite : 2
```



data()

- `data()` récupère le « tableau C » d'un `std::vector`

`foo.data()` OU `&foo[0]` ?

- Comportement identique
- Préférez `foo.data()` sémantiquement plus clair

emplace()

- `emplace()`, `emplace_back()` et `emplace_front()` construisent dans le conteneur depuis les paramètres d'un des constructeurs de l'élément

```
class Point {  
public:  
    Point(int a, int b);  
};  
  
vector<Point> foo;  
foo.emplace_back(2, 5);
```

Objectif

- Éliminer des copies inutiles et gagner en performance



std::string

- Évolutions de `std::string`
 - Éléments obligatoirement contigus
 - `data()` retourne une chaîne C valide (synonyme à `c_str()`)
 - `front()` retourne le premier caractère d'une chaîne
 - `back()` retourne le dernier caractère d'une chaîne
 - `pop_back()` supprime le dernier caractère d'une chaîne

std::bitset

- Évolutions de std::bitset
 - all() teste si tous les bits sont levés
 - to_ulong() convertit en **unsigned long long**

```
bitset<5> foo;  
foo.all();           // False  
  
foo.set(2);  
foo.to_ulong();     // 4  
  
foo.set();  
foo.all();           // True  
foo.to_ulong();     // 31
```



Conteneurs - Choix

Do

- Préférez `std::array` lorsque la taille est fixe et connue
- Sinon préférez `std::vector`

Itérateurs

- Fonctions membres `cbegin()`, `cend()`, `crbegin()` et `rcend()` retournant des `const_iterator`
- Fonctions libres `std::begin()` et `std::end()`
 - Conteneur : appel des fonctions membres
 - Tableau C : adresse du premier élément et suivant le dernier élément

```
int foo[] = {1, 2, 3, 4};  
vector<int> bar{2, 3, 4, 5};  
  
accumulate(begin(foo), end(foo), 0); // 10  
accumulate(begin(bar), end(bar), 0); // 14
```



Itérateurs

- Compatibles avec les conteneurs non-STL proposant `begin()` et `end()`
- Surchargeable sans modification du conteneur pour les autres

```
class Foo {  
public:  
    char* first();  
    const char* first() const;  
};  
  
char* begin(Foo& foo) {  
    return foo.first();  
}  
  
const char* begin(const Foo& foo) {  
    return foo.first();  
}
```

Itérateurs

Conseils

- `using std::begin` et `using std::end` permet l'ADL malgré la surcharge

Don't

- N'ouvrez pas le namespace `std` pour spécialiser

Do

- Préférez `std::begin()` et `std::end()` aux fonctions membres

Itérateurs

- Fonctions libres `std::prev()` et `std::next()` pour retrouver l'itérateur suivant ou précédent
- `move_iterator` : adaptateur d'itérateur retournant des *rvalue reference* lors du déréférencement

```
vector<string> foo(3), bar{"one", "two", "three"};

typedef vector<string>::iterator Iter;

copy(move_iterator<Iter>(bar.begin()),
      move_iterator<Iter>(bar.end()),
      foo.begin());
// foo : "one" "two" "three"
// bar : "" "" ""
```



Foncteurs prédéfinis

- `std::bit_and()` : et bit à bit
- `std::bit_or()` : ou inclusif bit à bit
- `std::bit_xor()` : ou exclusif bit à bit

```
vector<unsigned char> foo{0x10, 0x20, 0x30};  
vector<unsigned char> bar{0xFF, 0x25, 0x00};  
vector<unsigned char> baz;  
  
transform(begin(foo), end(foo), begin(bar), back_inserter(baz),  
          bit_and<unsigned char>());    // baz : 0x10 0x20, 0x00
```


Algorithmes – Recherche linéaire

- `std::find_if_not()` recherche le premier élément ne vérifiant pas le prédicat

```
vector<int> foo{1, 4, 5, 9, 12};  
  
find_if_not(begin(foo), end(foo), is_odd); // 4
```



Algorithmes – Comparaison

- `std::all_of()` teste si tous les éléments de l'ensemble vérifient un prédicat
 - Retourne vrai si l'ensemble est vide

```
vector<int> foo{1, 4, 5, 9, 12};  
vector<int> bar{1, 5, 9};  
vector<int> baz{4, 12};
```

```
all_of(begin(foo), end(foo), is_odd); // False  
all_of(begin(bar), end(bar), is_odd); // True  
all_of(begin(baz), end(baz), is_odd); // False
```

Algorithmes – Comparaison

- `std::any_of()` teste si au moins un élément vérifie un prédicat
 - Retourne faux si l'ensemble est vide

```
vector<int> foo{1, 4, 5, 9, 12};  
vector<int> bar{1, 5, 9};  
vector<int> baz{4, 12};
```

```
any_of(begin(foo), end(foo), is_odd); // True  
any_of(begin(bar), end(bar), is_odd); // True  
any_of(begin(baz), end(baz), is_odd); // False
```

Algorithmes – Comparaison

- `std::none_of()` teste si aucun élément ne vérifie le prédicat
 - Retourne vrai si l'ensemble est vide

```
vector<int> foo{1, 4, 5, 9, 12};  
vector<int> bar{1, 5, 9};  
vector<int> baz{4, 12};
```

```
none_of(begin(foo), end(foo), is_odd); // False  
none_of(begin(bar), end(bar), is_odd); // False  
none_of(begin(baz), end(baz), is_odd); // True
```



Algorithmes – Comparaison

- `std::is_permutation()` teste si un ensemble est la permutation d'un autre

```
vector<int> foo{1, 4, 5, 9, 12};  
vector<int> bar{1, 5, 4, 9, 12};  
vector<int> baz{5, 4, 3, 9, 1};  
  
is_permutation(begin(foo), end(foo), begin(bar)); // true  
is_permutation(begin(foo), end(foo), begin(baz)); // false
```



Algorithmes – Copie

- `std::copy_n()` copie les `n` premiers éléments d'un ensemble

```
vector<int> foo{1, 4, 5, 9, 12}, bar;  
  
copy_n(begin(foo), 3, back_inserter(bar)); // 1 4 5
```

- `std::copy_if()` copie les éléments vérifiant un prédicat

```
vector<int> foo{1, 4, 5, 9, 12}, bar;  
  
copy_if(begin(foo), end(foo), back_inserter(bar), is_odd); // 1 5 9
```



Algorithmes – Déplacement

- `std::move()` déplace les éléments d'un ensemble du début vers la fin

```
vector<int> foo{4, 5, 9, 12};  
vector<int> bar;  
  
move(begin(foo), end(foo), back_inserter(bar));
```

- `std::move_backward()` déplace les éléments de la fin vers le début
- Versions « déplacement » de `std::copy()` et `std::copy_backward()`

Algorithmes – Partitionnement

- `std::is_partitioned()` indique si un ensemble est partitionné

```
vector<int> foo{4, 5, 9, 12};  
vector<int> bar{9, 5, 4, 12};  
  
is_partitioned(begin(foo), end(foo), is_odd); // false  
is_partitioned(begin(bar), end(bar), is_odd); // true
```


Algorithmes – Partitionnement

- `std::partition_copy()` copie l'ensemble en le partitionnant
- `std::partition_point()` retourne le point de partition d'un ensemble partitionné
 - C'est à dire le premier élément ne vérifiant pas le prédicat

```
vector<int> foo{9, 5, 4, 12};
```

```
partition_point(begin(foo), end(foo), is_odd); // 4
```



Algorithmes – Tri

- `std::is_sorted()` indique si l'ensemble est ordonné (ordre ascendant)

```
vector<int> foo{4, 5, 9, 12};  
vector<int> bar{9, 5, 4, 12};  
  
is_sorted(begin(foo), end(foo)); // true  
is_sorted(begin(bar), end(bar)); // false
```

- `std::is_sorted_until()` détermine le premier élément mal placé

```
vector<int> foo{4, 5, 9, 3, 12};  
  
is_sorted_until(begin(foo), end(foo)); // 3
```



Algorithmes – Mélange

- `std::shuffle()` mélange l'ensemble grâce à un générateur de nombre aléatoire uniforme

```
vector<int> foo{4, 5, 9, 12};  
unsigned seed = now().time_since_epoch().count();  
  
shuffle(begin(foo), end(foo), default_random_engine(seed));
```



Algorithmes – Gestion de tas

- `std::is_heap()` indique si l'ensemble forme un tas

```
vector<int> foo{4, 5, 9, 3, 12};  
  
is_heap(begin(foo), end(foo)); // false  
make_heap(begin(foo), end(foo));  
is_heap(begin(foo), end(foo)); // true
```

- `std::is_heap_until()` indique le premier élément qui n'est pas dans la position correspondant à un tas

Algorithmes – Min-max

- `std::minmax()` retourne la paire constituée du plus petit et du plus grand de deux éléments

```
minmax(5, 2); // 2 - 5
```

- `std::minmax_element()` retourne la paire constituée des itérateurs sur le plus petit et le plus grand élément d'un ensemble

```
vector<int> foo{18, 5, 6, 8};
```

```
minmax_element(foo.begin(), foo.end()); // 5 - 18
```



Algorithmes – Numérique

- `std::iota()` affecte des valeurs successives aux éléments d'un ensemble

```
vector<int> foo(5);
```

```
iota(begin(foo), end(foo), 50); // 50 51 52 53 54
```



Algorithmes – Conclusion

Do

- Continuez à suivre les règles C++98/03 à propos des algorithmes

Do

- Privilégiez la sémantique lorsque plusieurs algorithmes sont utilisables

Range-based for loop

- Itération sur un conteneur complet

```
vector<int> foo{4, 8, 12, 37};  
  
for(int var : foo)  
    cout << var << " ";    // Affiche 4 8 12 37
```

- Compatible avec `auto`

```
vector<int> foo{4, 8, 12, 37};  
  
for(auto var : foo)  
    cout << var << " ";    // Affiche 4 8 12 37
```

- Utilisable sur tout conteneur
 - Exposant `begin()` et `end()`
 - Utilisable avec `std::begin()` et `std::end()`

Range-based for loop

Modification des éléments

- La variable d'itération doit être une référence

```
vector<int> foo(4);  
  
for(auto& var : foo)  
    var = 5;    // foo : 5 5 5 5
```



Range-based for loop

Do

- Préférez *range-based for loop* aux boucles classiques et à `std::for_each()`

Conseils

- Contrairement à `for`, l'indice de l'itération n'est pas disponible
- Malgré tout, préférez la *range-based for loop* avec un indice externe à `for`

Do

- Utilisez l'inférence de type sur la variable d'itération

std::string & conversions

- Fonctions de conversion d'une chaîne de caractères en un nombre
 - std::stoi() vers **int**
 - std::stol() vers **long**
 - std::stoul() vers **unsigned long**
 - std::stoll() vers **long long**
 - std::stoull() vers **unsigned long long**
 - std::stof() vers **float**
 - std::stod() vers **double**
 - std::stold() vers **long double**

```
stoi("56"); // 56
```

- S'arrêtent sur le premier caractère non convertible

std::string & conversions

- `std::to_string()` convertit d'un nombre en une chaîne de caractères

```
to_string(56); // "56"
```

- `std::to_wstring()` convertit vers une chaîne de caractères larges

std::string & conversions

Attention

- Pas de fonction `std::stoui()` de conversion vers un **unsigned int**

Do

- Préférez `std::sto...()` à `sscanf()`, `atoi()` ou `strto...()`

Do

- Préférez `std::to_string()` à `snprintf()` ou `itoa()`

Alternative et complément

- `Boost.Lexical_cast` permet de telles conversions et quelques autres

Chaînes de caractères UTF

- `char` doit pouvoir contenir un encodage 8 bits UTF-8
- `char16_t` représente un code point 16 bits
- `char32_t` représente un code point 32 bits
- `std::u16string` spécialisation de `basic_string` pour caractères 16 bits
- `std::u32string` spécialisation de `basic_string` pour caractères 32 bits
- Même interface que `std::string`

Nouvelles chaînes littérales

- Chaînes littérales UTF-8, UTF-16 et UTF32

```
string u8str      = u8"UTF-8 string";  
u16string u16str = u"UTF-16 string";  
u32string u32str = U"UTF-32 string";
```

Nouvelles chaînes littérales

- Chaînes littérales brutes (sans interprétation des échappements)
 - Préfixées par R
 - Encadrées par une paire de parenthèses
 - Éventuellement complétées d'un délimiteur

```
// Affiche Message\n en une seule \n ligne  
cout << R"(Message\n en une seule \n ligne)";  
cout << R"--(Message\n en une seule \n ligne)--";
```

- Composition possible des deux type de chaînes littérales

```
u8R"(Message\n en une seule \n ligne)";
```



User-defined literals

- Possibilité de définir des littéraux « utilisateur »
- Nombre (entier ou réel), caractère ou chaîne suffixé par un identifiant
- Identifiants non standards préfixés par _
- Définit via `operator""` suffixe

```
class Foo {  
public: explicit Foo(int a) : m_a{a} {}  
private : int m_a;  
};  
  
Foo operator""_f(unsigned long long int a) {  
    return Foo(a);  
}  
  
Foo foo = 12;    // Erreur compilation  
Foo bar = 12_f;  // OK
```

User-defined literals

- Littéraux brutes : chaîne C entièrement analysée par l'opérateur

```
Foo operator""_b(const char* str) {  
    unsigned long long a = 0;  
    for(size_t i = 0; str[i]; ++i)  
        a = (a * 2) + (str[i] - '0');  
    return Foo(a);  
}
```

```
Foo foo = 0110_b; // 6
```

Restrictions

- Uniquement pour les littéraux numériques

User-defined literals

- Littéraux « préparés » par le compilateur
 - Littéraux entiers : `unsigned long long int`
 - Littéraux réels : `long double`
 - Littéraux caractères : `char`, `wchar_t`, `char16_t` ou `char32_t`
 - Chaînes littérales : couple pointeur sur caractères et `size_t`

Motivations

- Pas de conversion implicite
- Expressivité



std::tuple

- Collection d'objets de type divers
- Généralisation de std::pair

```
tuple<int, char, long> foo;
```

- std::make_tuple() construit un std::tuple

```
tuple<int, char, long> foo = make_tuple(5, 'e', 98L);
```

std::make_tuple ou constructeur ?

std::make_tuple() permet la déduction de types, pas le constructeur

```
auto foo{5, 'e', 98L};           // KO  
auto bar = make_tuple(5, 'e', 98L); // OK
```

std::tuple

- Fonction de déstructuration `std::tie()`
- Et une constante pour ignorer des éléments `std::ignore`

```
int a; long b;  
tie(a, ignore, b) = foo;
```

- `std::get<>()` accède aux éléments du `std::tuple` par l'indice

```
char c = get<1>(foo);
```

Attention

- Les indices commencent à 0

std::tuple

- `std::tuple_cat()` concatène deux `std::tuple`

```
auto foo = make_tuple(5, 'e');  
auto bar = make_tuple(98L, 'r');  
auto baz = tuple_cat(foo, bar);           // 5 'e' 98L 'r'
```

- Classe représentant la taille `std::tuple_size`

```
tuple_size<decltype(baz)>::value;         // 4
```

- Classe représentant le type des éléments `std::tuple_element`

```
tuple_element<0, decltype(baz)>::type first; // int
```



std::tuple

Don't

- N'utilisez pas `std::tuple` pour remplacer une structure
- `std::tuple` regroupe localement des éléments sans lien sémantique

Do

- Préférez un `std::tuple` de retour aux paramètres OUT

Constructeurs de fstream

- Construction depuis des `std::string`

```
string filename{"foo.txt"};

// C++ 98
ofstream file(filename.c_str());

// C++ 11
ofstream file{filename};
```


=default & =delete

- Applicables aux fonctions générées implicitement le compilateur
 - Constructeur par défaut, par copie et par déplacement
 - Destructeur
 - Opérateur d'affectation
 - Opérateur d'affectation par déplacement
- `=default` force le compilateur à générer l'implémentation triviale
- `=delete` désactive la génération implicite de la fonction
- `=delete` peut aussi s'appliquer aux fonctions héritées pour les supprimer

```
class Foo {  
    public: Foo(int) {}  
    public: Foo() = default;  
  
    private: Foo(const Foo&) = delete;  
    private: Foo& operator=(const Foo&) = delete; };
```

=default & =delete

Do

- Préférez `=default` à une implémentation manuelle avec le même effet

Do

- Préférez `=delete` à une déclaration privée sans définition

`=default` ou non définition ?

- Consensus plutôt du côté de la non-définition
- Intérêt documentaire réel à `=default`

Initialisation par défaut des membres

- Initialisation des membres lors de la déclaration

```
struct Foo {  
    Foo() {}  
    int m_a{2};  
};
```

Restriction

- Pas d'initialisation avec ()
- Initialisation avec = uniquement sur des types copiables

Do

- Préférez l'initialisation des membres à l'initialisation par constructeurs pour les initialisations avec une valeur connue à la compilation

Délégation de constructeur

- Utilisation d'un constructeur dans l'implémentation d'un second
- ... en « l'initialisant » dans la liste d'initialisation

```
struct Foo {  
    Foo(int a) : m_a(a) {}  
    Foo() : Foo(2) {}  
    int m_a;  
};
```



Délégation de constructeur

Do

- Utilisez la délégation de constructeur pour mutualiser le code commun

Don't

- Évitez la délégation pour l'initialisation constante de membres
- Préférez l'initialisation par défaut des membres

Héritage de constructeur

- Indique que la classe hérite des constructeurs de la classe mère
- Génération du constructeur correspondant par le compilateur
 - Paramètres du constructeur de base
 - Appelle le constructeur de base correspondant
 - Initialise les membres sans fournir de paramètres

```
struct Foo {  
    Foo() {}  
    Foo(int a) : m_a(a) {}  
    int m_a{2};  
};  
  
struct Bar : Foo {  
    using Foo::Foo;  
};
```

Héritage de constructeur

- Redéfinition possible dans la classe dérivée

```
struct Bar : Foo {  
    using Foo::Foo;  
    Bar() : Foo(5) {}  
};
```

Valeurs par défaut

- Génération de toutes les combinaisons de constructeurs sans valeur par défaut correspondantes au constructeur de base avec des valeurs par défaut

Héritage multiple

- Héritage impossible de deux constructeurs avec la même signature



override

- Indique la redéfinition d'une fonction d'une classe de base

```
struct Foo {  
    Foo() {}  
    virtual void f(int);  
};  
  
struct Bar : Foo {  
    Bar() {}  
    virtual void f(int) override;  
};
```


override

- Provoque une erreur de compilation si
 - La fonction n'existe pas dans la classe de base
 - La fonction de la classe de base n'est pas virtuelle

```
struct Foo {  
    Foo() {}  
    virtual void f(int);  
    virtual void g(int) const;  
    void h(int);  
};  
  
struct Bar : Foo {  
    Bar() {}  
    void f(float) override;    // Erreur  
    void g(int) override;     // Erreur  
    void h(int) override;     // Erreur  
};
```



override

Objectifs

- Documentaire
- Détection des non-reports de modifications lors d'un refactoring
- Détection des redéfinitions involontaires

Do

- Marquez `override` les fonctions que vous redéfinissez

Do

- Utilisez `virtual` à la base de l'arbre d'héritage
- Utilisez `override` sur les redéfinitions

final

- Indique qu'une classe ne peut pas être dérivée

```
struct Foo final {  
    virtual void f(int);  
};  
  
struct Bar : Foo {    // Erreur  
    void f(int);  
};
```

- Aussi bien via l'héritage public que privé

final

- Ou qu'une fonction ne peut plus être redéfinie

```
struct Foo {  
    virtual void f(int);  
};  
  
struct Bar : Foo {  
    void f(int) final;  
};  
  
struct Baz : Bar {  
    void f(int);           // Erreur  
};
```

Do

- Utilisez `final` avec parcimonie



Opérateurs de conversion explicite

- Extension de `explicit` aux opérateurs de conversion
- ... qui ne définissent alors plus de conversion implicite

```
struct Foo { operator int() { return 5; } };
```

```
Foo f;
```

```
int a = f; // OK
```

```
int b = static_cast<int>(f); // OK
```

```
struct Foo { explicit operator int() { return 5; } };
```

```
Foo f;
```

```
int a = f; // Erreur
```

```
int b = static_cast<int>(f); // OK
```



noexcept

- Indique qu'une fonction ne jette pas d'exception

```
void foo() noexcept {}
```

- Pilotable par une expression booléenne

```
void foo() noexcept(true) {}
```

Dépréciation

- Les spécifications d'exception sont dépréciées
- Voir A Pragmatic Look at Exception Specifications (Herb Sutter)

noexcept

- Opérateur `noexcept()` teste, au *compile-time*, si une expression peut ou non lever une exception
- Pour l'appel de fonction, teste si la fonction est `noexcept`

```
noexcept(foo()); // true
```

Do

- Marquez `noexcept` les fonctions qui sémantiquement ne jette pas d'exception

Conversion exception - pointeur

- `std::exception_ptr` quasi-pointeur à responsabilité partagée sur une exception
- `std::current_exception()` récupère un pointeur sur l'exception courante
- `std::rethrow_exception()` relance l'exception contenue dans `std::exception_ptr`
- `std::make_exception_ptr()` construit `std::exception_ptr` depuis une exception

Conversion exception - pointeur

```
void foo() { throw 42; }

try {
    foo();
}
catch(...) {
    exception_ptr bar= current_exception();
    rethrow_exception(bar);
}
```

Motivation

- Faire passer la barrière des threads aux exceptions

Nested exception

- `std::nested_exception` contient une exception imbriquée
- `nested_ptr()` récupère un pointeur sur l'exception imbriquée
- `rethrow_nested()` relance l'exception imbriquée
- `std::rethrow_if_nested()` relance l'exception imbriquée si elle existe
- `std::throw_with_nested()` lance une exception embarquant l'exception courante

```
void foo() {  
    try { throw 42; }  
    catch(...) { throw_with_nested(logic_error("bar")); }  
}  
  
try { foo(); }  
catch(logic_error &e) { std::rethrow_if_nested(e); }
```



Énumérations fortement typées `enum class`

- Énumérations mieux typées
- Sans conversions implicites
- Énumérés locaux à l'énumération

```
enum class Foo { BAR1, BAR2 };
```

```
Foo foo = Foo::BAR1;
```

- Possibilité de fournir le type sous-jacent

```
enum class Foo : unsigned char { BAR1, BAR2 };
```

- `std::underlying_type` permet de récupérer ce type sous-jacent



Énumérations fortement typées `enum class`

Do

- Préférez les énumérations fortement typées

Bémol

- Pas de méthode simple et robuste pour récupérer la valeur ou l'intitulé de l'énuméré

std::function

- Encapsule un callable de n'importe quel type

```
int foo(int, int);  
  
function<int(int, int)> bar = foo;
```

- Copiable
- Peut être passer en paramètre ou retourner par une fonction

std::mem_fn

- Convertit une fonction membre en *function object* prenant une instance en paramètre

```
struct Foo { int f(int a) { return 2 * a; } };
```

```
Foo foo;
```

```
function<int(Foo, int)> bar = mem_fn(&Foo::f);
```

```
bar(foo, 5); // 10
```

Note

- Type de retour non spécifié mais stockable dans std::function

Dépréciation

- Dépréciation de std::mem_fun, std::ptr_fun et consorts

std::bind

- Construction de *function object* en liant des paramètres à un callable
- *Placeholders* `std::placeholders::_1`, `std::placeholders::_2`, ... pour lier les paramètres du *function object* à l'appelable

```
int foo(int a, int b) { return (a - 1) * b; }
```

```
function<int(int)> bar = bind(&foo, _1, 2);  
bar(3);                // 4
```

```
auto baz = bind(&foo, _2, _1);  
baz(3, 2, 1, 2, 3);    // 3
```

Dépréciation

- Dépréciation de `std::bind1st` et `std::bind2nd`



lambda et fermeture

Vocabulaire

- Lambda : fonction anonyme
- Fermeture : capture des variables libres de l'environnement lexical
- `[capture](parametres) specificateurs -> type_retour {instructions}`

```
int bar = 4;
auto foo = [&bar] (int a) -> int { bar *= a; return a; };

int baz = foo(5); // bar : 20, baz : 5
```


lambda et fermeture

- Capture

- `[]` : pas de capture
- `[x]` : capture `x` par valeur
- `[&y]` : capture `y` par référence
- `[=]` : capture tout par valeur
- `[&]` : capture tout par référence
- `[x, &y]` : capture `x` par valeur et `y` par référence
- `[=, &z]` : capture `z` par référence et le reste par copie
- `[&, z]` : capture `z` par valeur et le reste par référence

- La capture de variables membres se fait par la capture de `this`

- Soit explicitement via `[this]`

Capture de `this`

- Capture du pointeur, non de l'objet

- Soit via `[=]` ou `[&]`

lambda et fermeture

- Préservation de la constante des variables capturées
- Pas de capture des variables globales et statiques

Attention

- Par défaut, les variables capturées par copie ne sont pas modifiables

```
int i = 5;  
  
auto foo = [=] () { cout << ++i << "\n"; };           // Erreur  
auto bar = [=] () mutable { cout << ++i << "\n"; };   // OK
```

lambda et fermeture

- Spécificateurs
 - `mutable` : modification possible des variables capturées par copie
 - `noexcept` : ne lève pas d'exception
- Omission possible du type de retour si
 - Unique instruction
 - Un `return`
- Omission possible d'une liste de paramètres vide

```
auto foo = [] { return 5; };
```

Exception

- Omission impossible si la lambda est `mutable`

lambda, std::function, ... - Conclusion

Do

- Préférez les lambdas aux `std::function`
- Préférez les lambdas à `std::bind()`

Motivations

- Lisibilité, expressivité et performances
- Voir Practical Performance Practices

Attention

- Prenez garde à la durée de vie des variables capturées par référence

std::reference_wrapper

- Encapsule un objet en émulant une référence
- Construction par `std::ref()` et `std::cref()`
- Copiable

```
int a{10};  
reference_wrapper<int> aref = ref(a);  
  
aref++; // a : 11
```



Double chevron

- C++98/03 : >> est toujours l'opérateur de décalage
- C++11 : peut être une double fermeture de template

```
vector<vector<int>>> foo;  
// Invalide en C++98/03  
// Valide en C++11
```

- Utilisation de parenthèses pour forcer l'interprétation en tant qu'opérateur

```
vector<array<int, (0x10 >> 3) >>> foo;
```

Alias de template

- En C++98/03, `typedef` définit des alias sur des templates
- ... seulement si tous les paramètres templates sont explicites

```
template <typename T, typename U, int V>  
class Foo;
```

```
typedef Foo<int, int, 5> Baz;    // OK
```

```
template <typename U>  
typedef Foo<int, U, 5> Bar;      // Incorrect
```

Alias de template

- `using` permet la création d'alias ne définissant que certains paramètres

```
template <typename U>  
using Bar = Foo<int, U, 5>;
```

`using` de types

- `using` n'est pas réservé aux templates

```
using Error = int;
```


Extern template

- Indique que le template est instancié dans une autre unité de compilation
- Inutile de l'instancier ici

```
extern template class std::vector<int>;
```

Objectif

- Réduction du temps de compilation

Variadic template

- Template à nombre de paramètres variable
- Définition avec `typename...`

```
template<typename... Args>  
class Foo;
```

- Récupération de la liste avec ...

```
template<typename... Args>  
void bar(Args... parameters);
```

Variadic template

- Récupération de la taille avec `sizeof...`

```
template<typename... Args>
class Foo() {
private :
    static const unsigned int size = sizeof...(Args);
};
```

Variadic template

- Utilisation récursive par spécialisation

```
// Condition d'arret
template<typename T>
T sum(T val) {
    return val;
}

template<typename T, typename... Args>
T sum(T val, Args... values) {
    return val + sum(values...);
}

sum(1, 5, 56, 9);           // 71
sum(string("Un"), string("Deux")); // "UnDeux"
```



Variadic template

- Ou expansion sur une expression et une fonction d'expansion

```
template<typename... T> void pass(T&&...) {}

int total = 0;
foo(int i) {
    total += i;
    return i;
}

template<typename... T>
auto sum(T... t) {
    pass((foo(t))...); return total;
}

sum(1,2,3,5); // 11
```

Variadic template

Contraintes de l'expansion

- Paramètre unique
 - Ne retournant pas `void`
 - Pas d'ordre garanti
-
- Candidat naturel `std::initializer_list`
 - ... constructible depuis un *variadic template*

```
template<typename... T>
auto foo(T... t) {
    initializer_list<int>{ t... };
}

foo(1,2,3,5);
```

Variadic template

- ... qui règle le problème de l'ordre

```
int total = 0;
foo(int i) {
    total += i; return i;
}

template<typename... T>
auto sum(T... t) {
    initializer_list<int>{ (foo(t), 0)... };
    return total;
}

sum(1,2,3,5); // 11
```

Variadic template

- ... sur n'importe quelle expression prenant un paramètre

```
template<typename... T>
auto sum(T... t) {
    typename common_type<T...>::type result{};
    initializer_list<int>{ (result += t, 0)... };
    return result;
}

sum(1, 2, 3, 5); // 11
```

```
template<typename... T>
void print(T... t) {
    initializer_list<int>{ (cout << t << " ", 0)... };
}

print(1, 2, 3, 5);
```



std::enable_if

- Classe template sur une expression booléenne et un type
- Définition du type seulement si l'expression booléenne est vraie
- Templates disponibles uniquement pour certains types

```
template<class T,  
typename enable_if<is_integral<T>::value, T>::type* = nullptr>  
void foo(T data) { }  
  
foo(42);  
foo("azert");    // Erreur
```



Types locaux en arguments templates

- Utilisation des types locaux non-nommés comme arguments templates

```
void bar(vector<int>& foo) {  
    struct Less {  
        bool operator()(int a, int b) { return a < b; }  
    };  
  
    sort(foo.begin(), foo.end(), Less());  
}
```

- Y compris des lambdas

```
sort(foo.begin(), foo.end(),  
    [] (int a, int b) { return a < b; });
```

Type traits – Helper

- Constante *compile-time* `std::integral_constant`
- `std::integral_constant` booléen vrai `true_type`
- `std::integral_constant` booléen faux `false_type`

```
template <unsigned n>
struct factorial
    : integral_constant<int, n*factorial<n-1>::value> {};

template <>
struct factorial<0>
    : integral_constant<int, 1> {};

factorial<5>::value; // 120 en compile-time
```

Type traits – Trait

- Détermine, à la compilation, les caractéristiques des types
- `std::is_array` : tableau C

```
is_array<int>::value;      // false  
is_array<int[3]>::value;   // true
```

- `std::is_integral` : type entier

```
is_integral<short>::value; // true  
is_integral<string>::value; // false
```



Type traits – Trait

- `std::is_fundamental` : type fondamental (entier, réel, `void` ou `nullptr_t`)

```
is_fundamental<short>::value;    // true
is_fundamental<string>::value;   // false
is_fundamental<void*>::value;    // false
```

- `std::is_const` : type constant

```
is_const<const short>::value;    // true
is_const<string>::value;         // false
```

Type traits – Trait

- `std::is_base_of` : base d'un autre type

```
struct Foo {};  
struct Bar : Foo {};  
  
is_base_of<int, int>::value;           // false  
is_base_of<string, string>::value;    // true  
is_base_of<Foo, Bar>::value;          // true  
is_base_of<Bar, Foo>::value;          // false
```

- Et bien d'autres ...

Type traits – Transformations

- Construction d'un type par transformation d'un type existant
- `std::add_const` : type `const`

```
typedef add_const<int>::type A;           // const int
typedef add_const<const int>::type B;     // const int
typedef add_const<const int*>::type C;     // const int* const
```

Type traits – Transformations

- `std::make_unsigned` : type non signé correspondant

```
enum Foo {bar};

typedef make_unsigned<int>::type A;           // unsigned int
typedef make_unsigned<unsigned>::type B;     // unsigned int
typedef make_unsigned<const unsigned>::type C; // const unsigned int
typedef make_unsigned<Foo>::type D;          // unsigned int
```

- Et bien d'autres ...

Pointeurs intelligents

- RAII appliqué aux pointeurs et aux ressources allouées
- Objets à sémantique de pointeur gérant la durée de vie des objets
- Garantie de libération
- Garantie de cohérence
- Historiquement
 - `std::auto_ptr`
 - `boost::scoped_ptr` et `boost::scoped_array`

Pointeurs intelligents – `std::unique_ptr`

- Responsabilité exclusive
- Non copiable, mais déplaçable
- Testable

```
unique_ptr<int> p(new int);  
*p = 42;
```

- `release()` relâche la responsabilité de la ressource
- `reset()` change la ressource possédée
- `get()` récupère un pointeur brut sur la ressource

Attention

- Ne pas utiliser le pointeur retourné par `get()` pour libérer la ressource

Pointeurs intelligents – `std::unique_ptr`

- Fourniture possible de la fonction de libération

```
FILE *fp = fopen("foo.txt", "w");  
unique_ptr<FILE, int(*)(FILE*)> p(fp, &fclose);
```

- Spécialisation pour les tableaux C
 - Sans * et ->
 - Mais avec []

```
std::unique_ptr<int[]> foo (new int[5]);  
for(int i=0; i<5; ++i) foo[i] = i;
```

Dépréciation

- Dépréciation de `std::auto_ptr`

Pointeurs intelligents – `std::shared_ptr`

- Responsabilité partagée de la ressource
- Comptage de références
- Copiable (incrémentation du compteur de références)
- Testable

```
shared_ptr<int> p(new int());  
*p = 42;
```

- `reset()` change la ressource possédée
- `use_count()` retourne le nombre de possesseurs de la ressource
- `unique()` indique si la possession est unique
- Fourniture possible de la fonction de libération

Pointeurs intelligents – `std::make_shared()`

- Allocation et construction de l'objet dans le `std::shared_ptr`

```
shared_ptr<int> p = make_shared<int>(42);
```

Objectifs

- Pas de `new` explicite, plus robuste

```
// Fuite possible en cas d'exception depuis bar()  
foo(shared_ptr<int>(new int(42)), bar());
```

- Allocation unique pour la ressource et le compteur de référence

Do

- Utilisez `std::make_shared()` pour construire vos `std::shared_ptr`

Pointeurs intelligents – `std::weak_ptr`

- Aucune responsabilité sur la ressource
- Collabore avec `std::shared_ptr`
- ... sans impact sur le comptage de références
- Pas de création depuis un pointeur nu

Objectif

- Rompre les cycles

```
shared_ptr<int> sp(new int(20));  
weak_ptr<int> wp(sp);
```

Pointeurs intelligents – `std::weak_ptr`

- Pas d'accès à la ressource
- Convertible en `std::shared_ptr` via `lock()`

```
shared_ptr<int> sp = wp.lock();
```

- `reset()` vide le pointeur
- `use_count()` retourne le nombre de possesseurs de la ressource
- `expired()` indique si le `std::weak_ptr` ne référence plus une ressource valide

Pointeurs intelligents – Conclusion

Don't

- N'utilisez pas de pointeurs bruts possédants

Do

- Réfléchissez à la responsabilité de vos ressources

Do

- Préférez `std::unique_ptr` à `std::shared_ptr`
- Préférez une responsabilité unique à une responsabilité partagée

Pointeurs intelligents – Conclusion

Do

- Brisez les cycles à l'aide de `std::weak_ptr`

Attention

- Passez par un `std::unique_ptr` temporaire intermédiaire pour insérer des éléments dans un conteneur de `std::unique_ptr`
- Voir Overload 134 - C++ Antipatterns

Do

- Transférez au plus tôt la responsabilité à un pointeur intelligent

Pointeurs intelligents – Conclusion

Pour aller plus loin

- Voir Pointeurs intelligents (Loïc Joly)

Sous silence

- Allocateurs, mémoire non-initialisée, alignement, ...

Mais aussi

- Support minimal des *Garbage Collector*
- Mais pas de GC standard

Attributs

- Syntaxe standard pour les directives de compilation *inlines*
- ... y compris celles spécifiques à un compilateur
- Remplace la directive *#pragma*
- Et les mots-clé propriétaires (`__attribute__`, `__declspec`)

```
[[ attribut ]]
```

- Peut être multiple

```
[[ attribut1, attribut2 ]]
```

Attributs

- Peut prendre des arguments

```
[[ attribut(arg1, arg2) ]]
```

- Peut être dans un namespace et spécifique à une implémentation

```
[[ vendor::attribut ]]
```

Par exemple

les attributs `gs1` des « C++ Core Guidelines Checker » de Microsoft

```
[[ gs1::suppress(26400) ]]
```

Attributs

- Placé après le nom pour les entités nommées

```
int [[ attribut1 ]] i [[ attribut2 ]];  
// Attribut1 s'applique au type  
// Attribut2 s'applique a i
```

- Placé avant l'entité sinon

```
[[ attribut ]] return i;  
// Attribut s'applique au return
```

Bonus

- Aussi une information à destination des développeurs

Attribut `[[noreturn]]`

- Indique qu'une fonction ne retourne pas

```
[[ noreturn ]] void f() { throw "error"; }
```

Attention

- Qui ne retourne pas
- Pas qui ne retourne rien

Usage

- Boucle infinie, sortie de l'application, exception systématique

Sous silence

- `[[carries_dependency]]`

Rapport

- `std::ratio` représente un rapport entre deux nombres
- Numérateur et dénominateur sont des paramètres templates
- `num` accède au numérateur
- `den` accède au dénominateur

```
ratio<6, 2> r;  
cout << r.num << "/" << r.den;    // 3/1
```

- Instanciations standards des préfixes du système international d'unités
 - yocto, zepto, atto, femto, pico, nano, micro, milli, centi, déci
 - déca, hecto, kilo, méga, giga, téra, péta, exa, zetta, yotta

Rapport

- Méta-fonctions arithmétiques

- `std::ratio_add()`, `std::ratio_subtract()`
- `std::ratio_multiply()`, `std::ratio_divide()`

```
ratio_add<ratio<5, 1>, ratio<3, 2>> r;  
cout << r.num << "/" << r.den;    // 13/2
```

- Méta-fonctions de comparaison

- `std::ratio_equal()`, `std::ratio_not_equal()`
- `std::ratio_less()`, `std::ratio_less_equal()`
- `std::ratio_greater()` et `std::ratio_greater_equal()`



Durées

- Classe template `std::chrono::duration`
- Unité dépendante d'un ratio avec la seconde
- Instanciations standards *hours*, *minutes*, *seconds*, *milliseconds*, *microseconds* et *nanosecond*

```
milliseconds foo(500); // 500 ms  
foo.count();           // 500
```

- `count()` retourne la valeur
- `period` est le type représentant le ratio

```
milliseconds foo(10000);  
foo.count() * milliseconds::period::num /  
               milliseconds::period::den; // 10
```

Durées

- Opérateurs de manipulation des durées (ajout, suppression, ...)

```
milliseconds foo(500);  
milliseconds bar(10);  
foo += bar;    // 510  
foo /= 2;      // 255
```

- Opérateurs de comparaison entre durées
- `zero()` crée une durée nulle
- `min()` crée la plus petite valeur possible
- `max()` crée la plus grande valeur possible



Temps relatif

- `std::chrono::time_point` temps relatif depuis l'époque

Epoch

- Origine des temps de l'OS (1 janvier 1970 00h00 sur Unix)
- `time_since_epoch()` retourne la durée depuis l'époque
- Opérateurs d'ajout et de suppression d'une durée
- Opérateurs de comparaison entre `time_point`
- `min()` retourne le plus petit temps relatif
- `max()` retourne le plus grand temps relatif

Horloges

- Horloge temps-réel du système `std::chrono::system_clock`
- `now()` récupère temps courant

```
system_clock::time_point today = system_clock::now();  
today.time_since_epoch().count();
```

- `to_time_t()` converti en `time_t`
- `fromtime_t()` construit depuis `time_t`

```
system_clock::time_point today = system_clock::now();  
time_t tt = system_clock::to_time_t(today);  
ctime(&tt);
```

Horloges

- Horloge monotone de mesure des intervalles `std::chrono::steady_clock`
- `now()` récupère temps courant

```
steady_clock::time_point t1 = steady_clock::now();  
...  
steady_clock::time_point t2 = steady_clock::now();  
duration<double> time_span =  
duration_cast<duration<double>>(t2 - t1);
```

Horloges

- Horloge avec le plus petit intervalle entre deux *ticks*
`std::chrono::high_resolution_clock`
- Possible synonyme de `std::chrono::system_clock` ou `std::chrono::steady_clock`

Do

- Préférez `std::clock::duration` aux entiers pour manipuler les durées

Attention

- N'espérez pas une précision arbitrairement grande des horloges

Thread Local Storage

- Spécifieur de classe de stockage `thread_local`
- Influant sur la durée de stockage
- Compatible avec `static` et `extern`
- Rend propres au thread des objets normalement partagés
- Instance propre au thread créée à la création du thread
- Valeur initiale héritée du thread créateur

```
thread_local int foo = 0;
```

Variables atomiques – std::atomic

- Encapsulation de types de base fournissant des opérations atomiques
- Atomicité de l'affectation, de l'incrément et de la décrémentation

```
atomic<int> foo{5};  
++foo;
```

- store() stocke une nouvelle valeur
- load() lit la valeur
- exchange() met à jour et retourne la valeur avant modification

Variables atomiques – std::atomic

- `compare_exchange_weak` et `compare_exchange_strong`
 - Si `std::atomic` est égal à la valeur attendue, il est mis à jour avec une valeur fournie
 - Sinon, il n'est pas modifié et la valeur attendue prends la valeur de `std::atomic`

```
atomic<int> foo{5};  
int bar{5};  
  
foo.compare_exchange_strong(bar, 10); // foo : 10, bar : 10  
foo.compare_exchange_strong(bar, 8);  // foo : 10, bar : 10
```

Variables atomiques – std::atomic

- `fetch_add()` addition et retour de la valeur avant modification

```
atomic<int> foo{5};  
  
cout << foo.fetch_add(10) << " ";  
cout << foo;           // Affiche 5 15
```

- `fetch_sub()` soustraction et retour de la valeur avant modification
- `fetch_and()` et binaire et retour de la valeur avant modification
- `fetch_or()` ou binaire et retour de la valeur avant modification
- `fetch_xor()` ou exclusif et retour de la valeur avant modification

Variables atomiques – `std::atomic`

- Plusieurs instantiations standards (`std::atomic_bool`, `std::atomic_int`, ...)

Mais aussi

- Plusieurs fonctions « C-style », similaires aux fonctions membres de `std::atomic`, manipulant atomiquement des données

Variables atomiques – `std::atomic_flag`

- Gestion atomique de *flags*
- Non copiable, non déplaçable, *lock free*
- `clear()` remet à 0 le *flag*
- `test_and_set()` lève le *flag* et retourne sa valeur avant modification

```
atomic_flag foo = ATOMIC_FLAG_INIT;
```

```
foo.test_and_set(); // 0
```

```
foo.test_and_set(); // 1
```

```
foo.clear();
```

```
foo.test_and_set(); // 0
```

Threads – std::thread

- Représente un fil d'exécution
- Déplaçable mais non copiable
- Constructible depuis une fonction et sa liste de paramètre

```
void foo(int);  
  
thread t(foo, 10);
```

- Thread initialisé démarre immédiatement
- `joinable()` indique si le thread est joignable
 - Pas construit par défaut
 - Pas été déplacé
 - Ni joint ni détaché

Threads – std::thread

- join() attend la fin d'exécution du thread
- detach() détache le thread

```
void foo(int imax) {  
    for(int i = 0; i < imax; ++i)  
        cout << "thread " << i << '\n';  
}  
  
int imax = 40;  
thread t(foo, imax);  
  
for(int i = 0; i < imax; ++i)  
    cout << "main " << i << '\n';  
t.join();
```

Threads – `std::this_thread`

- Représente le thread courant
- `yield()` permet de « passer son tour »
- `sleep_for()` suspend l'exécution sur la durée spécifiée

```
this_thread::sleep_for(chrono::seconds(5));
```

- `sleep_until()` suspend le thread jusqu'au temps demandé

Attention

- Ne vous attendez pas à des attentes arbitrairement précises

Attentes passives

- Les autres threads continuent de s'exécuter

Mutex – `std::mutex`

- Verrou pour l'accès exclusif à une section de code
- `lock()` verrouille le mutex (en attendant sa libération s'il est déjà verrouillé)
- `try_lock()` verrouille le mutex s'il est libre, retourne `false` sinon
- `unlock()` relâche le mutex

Attention

- `lock()` sur un mutex verrouillé par le même thread provoque un *deadlock*
- `std::recursive_mutex` variante verrouillable plusieurs fois par un même thread

Mutex – `std::timed_mutex`

- Similaire à `std::mutex`
- ... proposant en complément des *try lock* temporisés
- `try_lock_for()` attend, si le mutex est verrouillé, la libération de celui-ci ou l'expiration d'une durée
- `try_lock_until()` attend, si le mutex est verrouillé, la libération de celui-ci ou l'atteinte d'un temps
- `std::recursive_timed_mutex` est une variante de `std::timed_mutex` verrouillable plusieurs fois par un même thread

Mutex – `std::lock_guard`

- Capsule RAII sur les mutex
- Constructible uniquement depuis un mutex
- Verrouille le mutex à la création et le relâche à la destruction

```
mutex foo;  
{  
    lock_guard<mutex> bar(foo);  // Prise du mutex  
    ...  
}  // Liberation du mutex
```

Note

- Gestion du mutex entièrement confiée au *lock*

Mutex – `std::unique_lock`

- Capsule RAII des mutex
- Supporte les mutex verrouillés ou non
- Relâche le mutex à la destruction
- Expose les méthodes de verrouillage et libération des mutex

```
mutex foo;
{
    unique_lock<mutex> bar(foo, defer_lock);
    ...
    bar.lock(); // Prise du mutex
    ...
} // Liberation du mutex
```

Mutex – `std::unique_lock`

- Comportements multiples à de la création
 - Verrouillage immédiat
 - Tentative de verrouillage
 - Acquisition sans verrouillage
 - Acquisition d'un mutex déjà verrouillé
- `mutex()` retourne le mutex associé
- `owns_lock()` teste si le *lock* a un mutex associé et l'a verrouillé
- `operator bool()` encapsule `owns_lock()`

Note

- Gestion du mutex conservée, garantie de libération

Mutex – Gestion multiple

- `std::lock()` verrouille tous les mutex passés en paramètre
- ... sans produire de *deadlock*

```
mutex foo, bar, baz;  
lock(foo, bar, baz);
```

- `std::try_lock` tente de verrouiller dans l'ordre tous les mutex passés en paramètre
- ... et relâche les mutex déjà pris en cas d'échec sur l'un d'eux

Mutex – `std::call_once()`

- Garantit l'appel unique (pour un *flag* donnée) de la fonction en paramètre
- Si la fonction a déjà été exécutée, `std::call_once()` retourne sans exécuter la fonction
- Si la fonction est en cours d'exécution, `std::call_once()` attend la fin de cette exécution avant de retourner

```
void foo(int, char);  
  
once_flag flag;  
call_once(flag, foo, 42, 'r');
```

Cas d'utilisation

- Appelle par un unique thread d'une fonction d'initialisation

Variables conditionnelles – Principe

- Mise en attente du thread sur la variable conditionnelle
- Réveil du thread lors de la notification de la variable
- Protection par verrou
 - Prise du verrou avant l'appel à la fonction d'attente
 - Relâchement du verrou par la fonction
 - Reprise du verrou lors de la notification avant le déblocage du thread

Variables conditionnelles – `std::condition_variable`

- Uniquement avec `std::unique_lock`
- `wait()` met en attente le thread

```
mutex mtx;  
condition_variable cv;  
  
unique_lock<std::mutex> lck(mtx);  
cv.wait(lck);
```

Note

- Possibilité de fournir un prédicat
 - Blocage seulement s'il retourne `false`
 - Déblocage seulement s'il retourne `true`

Variables conditionnelles – `std::condition_variable`

- `wait_for()` met en attente le thread, au maximum la durée donnée
- `wait_until()` met en attente le thread, au maximum jusqu'au temps donné

Note

- `wait_for()` et `wait_until()` indique si l'exécution a repris suite à un timeout

Variables conditionnelles – `std::condition_variable`

- `notify_one()` notifie un des threads en attente sur la variable conditionnelle

Attention

- Impossible de choisir quel thread notifié avec `notify_one()`
- `notify_all()` notifie tous les threads en attente
- `std::condition_variable_any` similaire à `std::condition_variable`
- ... sans être limité à `std::unique_lock`
- `std::notify_all_at_thread_exit()`
 - Indique de notifier tous les threads à la fin du thread courant
 - Prend un verrou qui sera libéré à la fin du thread

Variables conditionnelles – `std::condition_variable`

```
mutex mtx;
condition_variable cv;

void print_id(int id) {
    unique_lock<std::mutex> lck(mtx);
    cv.wait(lck);
    cout << "thread " << id << '\n';
}

thread threads[10];
for(int i = 0; i<10; ++i)
    threads[i] = thread(print_id, i);
this_thread::sleep_for(chrono::seconds(5));
cv.notify_all();
for(auto& th : threads) th.join();
```

Futures & promise – Principe

- `std::promise` contient une valeur
 - Disponible ultérieurement
 - Récupérable, éventuellement dans un autre thread, via `std::future`
- `std::future` permet la récupération d'une valeur disponible ultérieurement
 - Depuis un `std::promise`
 - Depuis un appel asynchrone ou différé de fonction
- Mécanismes asynchrones
- `std::future` définissent des points de synchronisation

Note

- `std::promise` et `std::future` peuvent également manipuler des exceptions

Futures & promise – `std::future`

- Utilisable uniquement s'il est valide (associé à un état partagé)
- Construit valide que par certaines fonctions fournisseuses
- Déplaçable mais non copiable
- Prêt lorsque la valeur, ou une exception, est disponible
- `valid()` teste s'il est valide
- `wait()` attend qu'il soit prêt
- `wait_for()` attend qu'il soit prêt, au plus la durée donnée
- `wait_until()` attend qu'il soit prêt, au plus jusqu'au temps donné
- `get()` attend qu'il soit prêt, retourne la valeur (ou lève l'exception) et libère l'état partagé

Futures & promise – `std::future`

- `share()` construit un `std::shared_future` depuis le `std::future`

Attention

- Après un appel à `share()`, le `std::future` n'est plus valide
- `std::shared_future` similaires à `std::future`
 - Mais copiables
 - Responsabilité partagée sur l'état partagé
 - Valeur lisible à plusieurs reprises

Futures & promise – `std::async()`

- Appelle la fonction fournie
- Et retourne, sans attendre la fin de l'exécution, un `std::future`
- `std::future` permettant de récupérer la valeur de retour de la fonction

Note

- Deux politiques d'exécution de la fonction appelée
 - Exécution asynchrone
 - Exécution différée à l'appel de `wait()` ou `get()`
- Par défaut le choix est laissé à l'implémentation

Futures & promise – `std::async()`

```
int foo() {  
    this_thread::sleep_for(chrono::seconds(5));  
    return 10;  
}  
  
future<int> bar = async(launch::async, foo);  
...  
cout << bar.get() << "\n";
```


Futures & promise – `std::promise`

- Objet que l'on promet de valoriser ultérieurement
- Déplaçable mais non copiable
- Partage un état avec le `std::future` associé
- `get_future()` retourne le `std::future` associé

Attention

- Un seul `std::future` par `std::promise` peut être récupéré

Futures & promise – `std::promise`

- `set_value()` affecte une valeur et passe l'état partagé à prêt
- `set_exception()` affecte une exception et passe l'état partagé à prêt
- `set_value_at_thread_exit()` affecte une valeur, l'état partagé passera à prêt à la fin du thread
- `set_exception_at_thread_exit()` affecte une exception, l'état partagé passera à prêt à la fin du thread

Futures & promise – std::promise

```
void foo(future<int>& fut) {  
    int x = fut.get();  
    cout << x << '\n';  
}  
  
promise<int> prom;  
future<int> fut = prom.get_future();  
thread th1(foo, ref(fut));  
...  
prom.set_value(10);  
th1.join();
```

Futures & promise – `std::packaged_task`

- Encapsulation d'un callable similaire à `std::function`
- ... dont la valeur de retour est récupérable par un `std::future`
- Partage un état avec le `std::future` associé
- `valid()` teste s'il est associé à un état partagé (contient un callable)
- `get_future()` retourne le `std::future` associé

Attention

- Un seul `std::future` par `std::packaged_task` peut être récupéré

Futures & promise – `std::packaged_task`

- `operator()` appelle l'appelable, affecte sa valeur de retour (ou l'exception levée) au `std::future` et passe l'état partagé à prêt
- `reset()` réinitialise l'état partagé en conservant l'appelable

note

- `reset()` permet d'appeler une nouvelle fois l'appelable
- `make_ready_at_thread_exit()` appelle l'appelable et affecte sa valeur de retour (ou l'exception levée), l'état partagé passera à prêt à la fin

Futures & promise – std::packaged_task

```
void foo(future<int>& fut) {  
    int x = fut.get();  
    cout << x << '\n';  
}  
  
int bar() { return 10; }  
  
packaged_task<int>() tsk(bar);  
future<int> fut = tsk.get_future();  
thread th1(foo, std::ref(fut));  
...  
tsk();  
th1.join();
```

Conclusion

Do, dans cet ordre

- Évitez de partager variables et ressources
- Préférez les partages en lecture seule
- Préférez les structures de données gérant les accès concurrents
- Protégez l'accès par mutex ou autres barrières

Do

- Encapsulez les mutex dans des `std::lock_guard` ou `std::unique_lock`

Conclusion

Do

- Analysez vos cas d'utilisation pour choisir le bon outil

Attention

- Très faibles garanties de *thread-safety* de la part des conteneurs standards

Do

- Boost.Lockfree pour des structures de données *thread-safe* et *lock-free*

Pour aller plus loin

- [C++ Concurrency in action] d'Anthony Williams

Expressions rationnelles (regex)

- `std::basic_regex` représente une expression rationnelle
- Instanciations standards `std::regex` et `std::wregex`
- Construite depuis une chaîne représentant l'expression
- ... et des drapeaux de configuration
 - Grammaire : ECMAScript, basic POSIX, extended POSIX, awk, grep, egrep
 - Case sensitive ou non
 - Prise en compte de la locale
 - ...

```
regex foo("[0-9A-Z]+", icafe);
```

Expressions rationnelles (regex)

- `std::regex_search()` recherche

```
regex r("[0-9]+");  
regex_search(string("123"), r);           // true  
regex_search(string("abcd123efg"), r);    // true  
regex_search(string("abcdefg"), r);       // false
```

- `std::regex_match()` vérifie la correspondance

```
regex r("[0-9]+");  
regex_match(string("123"), r);             // true  
regex_match(string("abcd123efg"), r);      // false  
regex_match(string("abcdefg"), r);         // false
```

Expressions rationnelles (regex)

- Capture de sous-expressions dans `std::match_results`
- Instanciations standards `std::cmatch`, `std::wcmatch`, `std::smatch` et `std::wsmatch`
- `empty()` teste la vacuité de la capture
- `size()` retourne le nombre de captures
- Itérateurs sur les captures
- Sur chaque élément capturé
 - `str()` : la chaîne capturée
 - `length()` : sa longueur
 - `position()` : sa position dans la chaîne de recherche
 - `suffix()` : la séquence de caractères suivant la capture
 - `prefix()` : la séquence de caractères précédant la capture

Expressions rationnelles (regex)

```
string s("abcd123efg");  
regex r("[0-9]+");  
smatch m;
```

```
regex_search(s, m, r);  
m.size();           // 1  
m.str(0);           // 123  
m.position(0);      // 4  
m.prefix();         // abcd  
m.suffix();         // efg
```

Expressions rationnelles (regex)

- Fonction de remplacement : `std::regex_replace()`

```
string s("abcd123efg");  
regex r("[0-9]+");  
regex_replace(s, r, "-"); // abcd-efg
```



Expressions rationnelles (regex)

Do

- Préférez les expressions rationnelles aux analyseurs « à la main »

Don't

- N'utilisez pas les expressions rationnelles pour les traitements triviaux
- Préférez les algorithmes

Conseil

- Encapsulez les expressions rationnelles ayant une sémantique claire et utilisées plusieurs fois dans une fonction dédiée au nom évocateur

Performance

- Construction très couteuse de l'expression rationnelle

Nombres aléatoires

- Générateurs pseudo-aléatoires initialisés par une graine (congruence linéaire, Mersenne, ...)
- Générateur aléatoire

Attention

- Peut ne pas être présent sur certaines implémentations
- Peut être un générateur pseudo-aléatoire (entropie nulle) sur d'autres
- Distributions adaptant la séquence d'un générateur pour respecter une distribution particulière (uniforme, normale, binomiale, de Poisson, ...)
- Fonction de normalisation ramenant la séquence générée dans $[0,1)$

Nombres aléatoires

```
default_random_engine gen;  
uniform_int_distribution<int> distribution(0,9);  
gen.seed(system_clock::now().time_since_epoch().count());  
  
// Nombre aleatoire entre 0 et 9  
distribution(gen);
```

Do

- Préférez ces générateurs et distributions à `rand()`

Quiz

Comment générer un tirage équiprobable entre 6 et 42 avec `rand()`



Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Approuvé le 16 août 2014
- Dernier *Working Draft* : N4140
- Dans la continuité de C++11
- Changements moins importants
- Mais loin d'une simple version correctrice
- Support complet par GCC, Clang et Visual C++

constexpr

- Fonctions membres `constexpr` plus implicitement `const`
- Relâchement des contraintes sur les fonctions `constexpr`
 - Variables locales (ni `static`, ni `thread_local`, obligatoirement initialisées)
 - Objets mutables créés lors l'évaluation de l'expression constante
 - `if`, `switch`, `while`, `for`, `do while`
- Application de `constexpr` à plusieurs éléments de la bibliothèque standard

Généralisation de la déduction du type retour

- Utilisable sur les lambdas complexes

```
[](int x) {  
    if(x >= 0) return 2 * x;  
    else return -2 * x;  
};
```

- Mais aussi sur les fonctions

```
auto bar(int x) {  
    if(x >= 0) return 2 * x;  
    else return -2 * x;  
}
```

Généralisation de la déduction du type retour

- Y compris récursive

```
auto fact(unsigned int x) {  
    if(x == 0) return 1U;  
    else return x * fact(x-1);  
}
```

Contraintes

- Un `return` doit précéder l'appel récursive
- Tous les chemins doivent avoir le même type de retour



decltype(auto)

- Dédution du type retour en conservant la référence

```
string bar("bar");

string foo1() { return string("foo"); }
string& bar1() { return bar; }

decltype(auto) foo2() { return foo1(); } // string
decltype(auto) bar2() { return bar1(); } // string&
auto foo3() { return foo1(); }           // string
auto bar3() { return bar1(); }           // string
```

Aggregate Initialisation

- Compatible avec l'initialisation par défaut des membres
- Initialisation par défaut des membres non explicitement initialisés

```
struct Foo {int i, int j = 5};
```

```
Foo foo{42};    // i = 42, j = 5
```



Itérateurs

- Fonctions libres `std::cbegin()` et `std::cend()`
- Fonctions libres `std::rbegin()` et `std::rend()`
- Fonctions libres `std::crbegin()` et `std::crend()`
- *Null forward iterator* ne référençant aucun conteneur valide

```
auto ni = vector<int>::iterator();  
auto nd = vector<double>::iterator();  
  
ni == ni;    // true  
nd != nd;    // false  
ni == nd;    // Erreur de compilation
```

Attention

- *Null forward iterator* non comparables avec des itérateurs « classiques »

Recherche hétérogène

- Optimisation de la recherche hétérogène dans les conteneurs associatifs ordonnés
- Fourniture d'une classe exposant
 - Fonction de comparaison
 - *Tag* `is_transparent`
- Suppression de conversions inutiles

Algorithmes

- Surcharge de `std::equal()`, `std::mismatch()` et de `std::is_permutation()` prenant deux paires complètes d'itérateurs

```
vector<int> foo{1, 2, 3};  
vector<int> bar{10, 11};  
  
equal(begin(foo), end(foo), begin(bar), end(bar));
```

- `std::exchange()` change la valeur d'un objet et retourne l'ancienne

```
vector<int> foo{1, 2, 3};  
  
vector<int> bar = exchange(foo, {10, 11});  
// foo : 10 11, bar : 1, 2, 3
```

Dépréciation

- Dépréciation de `std::random_shuffle()`

Quoted string

- Insertion et extraction de chaînes avec guillemets

```
stringstream ss;  
string in = "String with spaces and \"quotes\"";  
string out;  
  
ss << quoted(in);  
cout << "in: " << in << "'\n"  
      << "stored as " << ss.str() << "'\n";  
// in : 'String with spaces and "quotes"'  
// stored as '"String with spaces and \"quotes\""'  
  
ss >> quoted(out);  
cout << "out: " << out << "'\n";  
// out: 'String with spaces, and "quotes"'
```



Littéraux binaires

- Support des littéraux binaires préfixés par `0b`

```
int foo = 0b101010; // 42
```



Séparateurs

- Utilisation possible de ' dans les nombres littéraux

```
int foo = 0b0010'1010; // 42  
int bar = 1'000;        // 1000  
int baz = 010'00;       // 512
```

Note

- Purement esthétique, aucune sémantique ni place réservée



User-defined literals standards

- Suffixe `s` sur les chaînes : `std::string`

```
auto foo = "abcd"s;    // string
```

Note

- Remplace `std::string{"abcd"}`

Attention

- Nécessite l'utilisation de `using namespace std::literals`



User-defined literals standards

- Suffixe h, min, s, ms, us et ns : `std::chrono::duration`

```
auto foo = 60s;    // chrono::seconds  
auto bar = 5min;   // chrono::minutes
```

User-defined literals standards

- Suffixe `if` : nombre imaginaire de type `std::complex<float>`
- Suffixe `i` : nombre imaginaire de type `std::complex<double>`
- Suffixe `il` : nombre imaginaire de type `std::complex<long double>`

```
auto foo = 5i; // complex<double>
```


Adressage des `std::tuple` par le type

- Utilisation du type plutôt que de l'indice

```
tuple<int, long, long> foo{42, 58L, 9L};  
  
get<int>(foo);           // 42
```

Attention

- Uniquement s'il n'y a qu'une occurrence du type dans le `std::tuple`

```
get<long>(foo);  // Erreur
```



Variable template

- Généralisation des templates aux variables
- Y compris les spécialisations

```
template<typename T>
constexpr T PI = T(3.1415926535897932385);

template<>
constexpr const char* PI<const char*> = "pi";

PI<int>;           // 3
PI<double>;        // 3.14159
PI<const char*>;   // pi
```



Generic lambdas

- Lambdas utilisables sur différents types de paramètres
- Déduction du type des paramètres déclarés `auto`

```
auto foo = [] (auto in) { cout << in << '\n'; };  
  
foo(2);  
foo("azerty"s);
```



Variadic lambdas

- Lambda à nombre de paramètres variable
- Suffixe ... à `auto`

```
auto foo = [] (auto... args) {  
    std::cout << sizeof...(args) << '\n';  
};
```

```
foo(2);           // 1
```

```
foo(2, 3, 4);     // 3
```

```
foo("azerty"s);  // 1
```



Capture généralisée

- Création de variables capturées depuis des variables locales ou des constantes

```
int foo = 42;

auto bar = [ &x = foo ]() { --x; };
bar(); // foo : 41

auto baz = [ y = 10 ]() { cout << y << '\n'; };
baz(); // 10

auto qux = [ z = 2 * foo ]() { cout << z << '\n'; };
qux(); // 82
```

Capture généralisée

- Capture par déplacement

```
auto foo = make_unique<int>(42);  
auto bar = [ foo = move(foo) ](int i) {  
    cout << *foo * i << '\n';  
};  
  
bar(5); // Affiche 210
```

- Capture des variables membres

```
struct Bar {  
    auto foo() { return [s=s] { cout << s << '\n'; }; }  
  
    string s;  
};
```



Améliorations des lambdas

- Type de retour complètement facultatif
- Conversion possible de lambda sans capture en pointeur de fonction

```
void foo(void(* bar)(int))  
  
foo([](int x) { cout << x << endl; });
```

- Peuvent être **noexcept**
- Ajout des paramètres par défaut aux lambdas

```
auto foo = [] (int bar = 12) { cout << bar << '\n'; };
```

std::is_final

- Indique si la classe est finale ou non

```
class Foo {};  
class Bar final {};  
  
is_final<Foo>::value;    // false  
is_final<Bar>::value;    // true
```


Alias transformation

- Simplification de l'usage des transformations de types
- Ajout du suffixe `_t` aux transformations
- Suppression de `typename` et `::type`

```
typedef add_const<int>::type A;  
typedef add_const<const int>::type B;  
typedef add_const<const int*>::type C;
```

// Deviennent

```
add_const_t<int> A;  
add_const_t<const int> B;  
add_const_t<const int*> C;
```

std::make_unique

- Allocation et construction de l'objet dans le `std::unique_ptr`

```
unique_ptr<int> foo = make_unique<int>(42);
```

Don't

- Plus de `new` dans le code applicatif

Note

- Utilisable pour construire dans un conteneur

Attribut `[[deprecated]]`

- Indique qu'une entité (variable, fonction, classe, ...) est dépréciée
- Émission possible d'avertissement sur l'utilisation d'une entité deprecated

```
[[ deprecated ]]  
void bar() {}  
  
class [[ deprecated ]] Baz {};  
  
[[ deprecated ]]  
int foo{42};
```

Attribut `[[deprecated]]`

- Possibilité de fournir un message explicatif

```
[[ deprecated("utilisez foo") ]]  
void bar() {}
```

```
warning: 'void bar()' is deprecated: utilisez foo
```



std::shared_timed_mutex

- Similaire à `std::timed_mutex` avec deux niveaux d'accès
 - Exclusif : possible si le verrou n'est pas pris
 - Partagé : possible si le verrou n'est pas pris en exclusif
- Même API que `std::timed_mutex` pour l'accès exclusif
- API similaire pour l'accès partagé
 - `lock_shared`
 - `try_lock_shared`
 - `try_lock_shared_for`
 - `try_lock_shared_until`
 - `unlock_shared`

Attention

- Un thread ne doit pas prendre un mutex qu'il possède déjà
- Même en accès partagé

std::shared_lock

- Capsule RAII sur les mutex partagés
- Support des mutex verrouillés ou non
- Relâche le mutex à la destruction
- Similaire à `std::unique_lock` mais en accès partagée

```
shared_timed_mutex foo;
{
    shared_lock<shared_timed_mutex> bar(foo, defer_lock);
    ...
    bar.lock(); // Prise du mutex
    ...
} // Liberation du mutex
```

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Approuvé en décembre 2017
- Dernier Working Draft : N4659
- Support complet du langage par Clang, GCC et Visual C++
- Très bon support de la bibliothèque par Clang, GCC et Visual C++
- Progression très rapide du support en parallèle de la normalisation

Note

- Voir Vidéos C++ Weekly (Jason Turner)

Fonctionnalités supprimées

- Suppression des trigraphes (non dépréciés)

Note

- Les digraphes ne sont pas concernés
- Suppression de `register` (qui reste un mot réservé)
- Suppression des opérateurs d'incrément sur les booléens
- Suppression de `std::auto_ptr`
- Suppression de `std::random_shuffle()`
- Suppression des anciens mécanismes fonctionnels : `std::bind1st()`, `std::bind2nd()`, ...
- Suppression des spécifications d'exception

__has_include

- Teste la présence d'un fichier d'en-tête
- Et donc la disponibilité d'une fonctionnalité

```
#if __has_include(<optional>)  
# include <optional>  
# define OPT_ENABLE  
#endif
```



inline variable

- Sémantique **inline** identique sur fonctions et variables
- Peut être définie, à l'identique, dans plusieurs unité de compilation
- Se comporte comme s'il n'y avait qu'une variable

```
inline int foo = 42;
```

- **constexpr** sur une donnée membre statique implique **inline**
- Utile pour initialiser des variables membres statiques non constantes

```
class Foo { static inline int bar = 42; };
```

Don't

- Ne justifie pas l'usage de variables globales

Nested namespace

- Simplification des imbrications de namespaces via l'opérateur ::

```
namespace A {  
namespace B {  
namespace C {  
...  
}}}
```

// Devient

```
namespace A::B::C {  
...  
}
```



static_assert sans message

- `static_assert` sans message utilisateur

```
static_assert(sizeof(int) == 3);  
// Erreur de compilation
```



if constexpr

- Branchement évalué à la compilation

```
if constexpr(cond)
{ ... }
else if constexpr(cond)
{ ... }
else
{ ... }
```

Motivation

- Conditions d'arrêt plus simple avec les *variadic template*
- Moins de spécialisations explicites

Note

- Conditions intégralement évaluables au *compile-time*, pas de court-circuit

if constexpr

```
template <typename T> auto foo(T t) {  
    if constexpr(is_pointer_v<T>)  
        return *t;  
    else  
        return t;  
}  
  
int a = 10, b = 5;  
int* ptr = &b;  
cout << foo(a) << ' ' << foo(ptr); // 10 5
```

if constexpr

Note

- Les branches doivent être syntaxiquement correctes
- ... mais pas nécessairement sémantiquement valides

Note

- Les branches peuvent avoir des types retour différents sans remettre en cause la déduction de type retour

Do

- Préférez `if constexpr` aux suites de spécialisations de template et SFINAE, aux imbrications de ternaires ou à *#if*

if constexpr

hello world de la récursion

```
template<int N>
constexpr int fibo(){ return fibo<N-1>()+fibo<N-2>(); }
template<>
constexpr int fibo<1>() { return 1; }
template<>
constexpr int fibo<0>() { return 0; }
```

// Devient

```
template<int N>
constexpr int fibo() {
    if constexpr (N>=2) return fibo<N-1>()+fibo<N-2>();
    else return N;
}
```



if init statement

- Initialisation dans le branchement
- Portée identique aux déclarations dans la condition

```
if(int foo = 42; bar) cout << foo;  
else                 cout << -foo;
```

- Sémantiquement équivalent à

```
{  
    int foo = 42;  
    if(bar)  cout << foo;  
    else    cout << -foo;  
}
```

if init statement

- Alternative à certaines constructions peu lisibles

```
if((bool ret = foo()) == true) ...
```

- ... injectant un symbole inutile au delà du branchement

```
bool ret = foo();  
if(ret) ...
```

- ... nécessitant l'introduction d'une portée supplémentaire

```
{  
    bool ret = foo();  
    if(ret) ...  
}
```



switch init statement

- Initialisation dans le `switch()`
- Utilisable dans le corps du `switch()`

```
switch(int foo = 42; bar) {  
    ...  
}
```



Structured binding

- Décomposition automatique des types composés en multiples variables

```
auto [liste de nom] = expression;
```

- Sur des types dont les données membres non statiques
 - Sont toutes publiques
 - Sont toutes des membres de l'objet ou de la même classe de base publique
 - Ne sont pas des unions anonymes
- Et sur les classes implémentant `get<>()`, `tuple_size` et `tuple_element`
- Notamment `std::tuple`, `std::pair`, `std::array`

Structured binding

```
tuple<int, long, string> foo();  
auto [x,y,z] = foo();
```

```
class Foo {  
    const int i = 42;  
    const string s{"Hello"};  
public: template <int N> auto& get() const {  
    if constexpr(N == 0) { return i; }  
    else { return s; } } };  
  
template<> struct tuple_size<Foo>  
    : integral_constant<size_t, 2> {};  
  
template<size_t N> struct tuple_element<N, Foo> {  
    using type = decltype(declval<Foo>().get<N>()); };  
  
auto [ i, s ] = Foo{};
```

Structured binding

- Compatible avec `const`

```
tuple<int, long, string> foo();  
const auto [x,y,z] = foo();
```

- Avec les références

```
auto& [refX,refY,refZ] = monTuple;
```

Attention

- La portée de l'objet référencé doit être supérieure à celle des références

Structured binding

- Avec *range-based for loop*

```
map<int, string> myMap;  
for(const auto& [k,v] : myMap)  
{ ... }
```

- Avec *if init statement*

```
if(auto [iter, succeeded] = myMap.insert(value); succeeded)  
{ ... }
```



Structured binding

Objectif

- Meilleure lisibilité
- Remplacement de `std::tie()`

Nom

- Déstructuration (*destructuring*) dans d'autres langages

Et ensuite ?

- Premier pas vers les types algébriques de données et le *pattern matching*

Limite

- Pas de capture de *structured binding* par les lambdas

Ordre d'évaluation

- Ordre d'évaluation fixé
 - De gauche à droite pour les expressions post-fixées
 - De droite à gauche pour les affectations
 - De gauche à droite pour les décalages

```
// a avant b  
a.b;  
a->b,  
b op= a;  
a[b];  
a << b;  
a >> b;
```

Ordre d'évaluation

- Évaluation complète d'un paramètre avant celle du suivant

```
f(a(x), b, c(y));  
// Lorsque x est evalue, a(x) l'est avant b, y ou c(y)
```

Ordre des paramètres

Ordre d'évaluation des paramètres toujours non fixé

Élision de copie

- Élision garantie pour les objets créés dans l'instruction de retour

```
T f() {  
    return T{}; // Pas de copie  
}
```

```
T g() {  
    T t;  
    return t; // Copie potentielle eludee  
}
```

Élision de copie

- Élision garantie lors de la définition d'une variable locale

```
T t = f(); // Pas de copie
```

- Même en l'absence de constructeur par copie

Note

- Élision de copies possibles avant C++17, garanties maintenant

Aggregate Initialisation

- Généralisation aux classes dérivées
- Incluant l'initialisation de la classe de base

```
struct Foo {int i;};  
struct Bar : Foo {double l;};  
  
Bar bar{{42}, 1.25};  
Bar baz{}, 1.25};    // Foo non initialise
```

Attention

- Uniquement sur de l'héritage public non virtuel
- Pas de constructeur fourni par l'utilisateur (y compris hérité)
- Pas de donnée membre non statique privée ou protégée
- Pas de fonction virtuelle



Déduction de type et Initializer list

- Évolution des règles de déduction sur les listes entre accolades
 - *Direct initialisation* : déduction d'une valeur
 - *Copy initialisation* : déduction d'un `initializer_list`

```
auto x1 = { 1, 2 };    // std::initializer_list<int>
auto x2 = { 1, 2.0 };  // Erreur
auto x3{ 1, 2 };       // Erreur : multiples elements
auto x4 = { 3 };       // std::initializer_list<int>
auto x5{ 3 };          // int
```

Initialisation des énumérations fortement typées

- Initialisation possible d'`enum class` avec une constante du type sous-jacent

```
enum class Foo : unsigned int { Invalid = 0 };  
Foo foo{42};  
Foo bar = Foo{42};
```


Initialisation des énumérations fortement typées

- Pas de relâchement du typage par ailleurs
- En particulier, pas de copie ni d'affectation depuis un entier

```
Foo foo;  
foo = 42;           // Erreur
```

- Ni d'initialisation avec la syntaxe =

```
Foo foo = 42;       // Erreur  
Foo bar = {42};     // Erreur
```



std::byte

- Stockage de bits
- Pas un type caractère ni arithmétique
- Remplace les solutions à base de **unsigned char**
- Supporte les opérations binaires (décalage, et, ou, non)
- Supporte les constructions depuis un type entier
- ... et les conversions vers des entiers (`std::to_integer`)
- Mais pas les opérations arithmétiques

```
std::byte b{5};  
b |= std::byte{2};  
b <= 2;  
std::to_integer<unsigned int>(b); // 28-1C
```



Déplacement de nœuds entre conteneurs associatifs

- Déplacement de nœuds entre conteneurs associatifs de même type
- Objet *node handle* : stockage et accès au nœud
 - Déplaçable mais non copiable
 - Modification possible de la clé
 - Destruction du nœud lors de sa destruction
- `extract()` extrait le nœud du premier conteneur
 - Nœud identifié par sa clé ou par un itérateur
 - Retourne un *node handle*
- Surcharge de `insert()` prenant un *node handle* en paramètre
 - Retourne une structure indiquant la réussite ou non de l'insertion
 - ... et, en cas d'échec, le *node handle*

Motivations

- Éviter des copies inutiles
- Modifier une clé dans une `std::map`

Déplacement de nœuds entre conteneurs associatifs

```
map<int, string> foo {{1,"foo1"}, {2,"foo2"}};  
map<int, string> bar {{2,"bar2"}};
```

```
bar.insert(foo.extract(1));  
// foo : {{2,"foo2"}}  
// bar : {{1,"foo1"}, {2,"bar2"}}
```

```
auto r = bar.insert(foo.extract(2)); // Echec  
// foo : {}  
// bar : {{1,"foo1"}, {2,"bar2"}}  
// r.inserted : false, r.node : {2,"foo2"}
```

```
r.node.key() = 3;  
bar.insert(r.position, std::move(r.node));  
// foo : {}  
// bar : {{1,"foo1"}, {2,"bar2"}, {3,"bar2"}}
```



Fusion de conteneurs associatif

- `merge()` fusionne le contenu de conteneurs associatifs

```
map<int, string> foo {{1, "foo1"}, {2, "foo2"}};  
map<int, string> bar {{3, "bar2"}};  
  
foo.merge(bar);  
// foo : {{1, "foo1"}, {2, "foo2"}, {3, "bar2"}}
```

std::map & std::unordered_map

- try_emplace() tente de construire en place
- ... sans effet, même pas un « vol » de la valeur, si la clé existe déjà
- insert_or_assign() ajoute ou modifie un élément

```
map<int, string> foo {{1, "foo1"}, {2, "foo2"}};  
foo.insert_or_assign(3, "foo3");  
// foo : {{1, "foo1"}, {2, "foo2"}, {3, "foo3"}}  
  
foo.insert_or_assign(2, "foo2bis");  
// foo : {{1, "foo1"}, {2, "foo2bis"}, {3, "foo3"}}
```



emplace_back(), emplace_front()

- Retournent une référence sur l'élément ajouté

```
vector<...> foo;  
  
foo.emplace_back(...);           // C++14 et precedents  
auto& val = foo.back();  
  
auto& val = foo.emplace_back(...); // C++17
```

```
vector<vector<int>> foo;  
foo.emplace_back(3, 1).push_back(42); // foo : {{1 1 1 42}}
```

Note

- emplace() renvoie toujours un itérateur

Fonctions libres de manipulation

- `std::size()`
 - Conteneurs et `initializer_list` : résultat de la fonction membre `size()`
 - Tableau C : taille du tableau
- `std::empty()`
 - Conteneurs : résultat de la fonction membre `empty()`
 - Tableau C : `false`
 - `initializer_list` : `size() == 0`
- `std::data()`
 - Conteneurs : résultat de la fonction membre `data()`
 - Tableau C : pointeur sur la première case
 - `initializer_list` : itérateur sur le premier élément

ContiguousIterator

- Basé sur `RandomAccessIterator`
- Mais sur des conteneurs à stockage contigu
- Itérateur associé à
 - `std::vector`
 - `std::array`
 - `std::basic_string`
 - `std::valarray`
 - Aux tableaux C

Motivations

- Utilisation avec des API C
- Utilisation de `memcpy` et `memset`

Limitation de plage de valeurs

- `std::clamp()` ramène une valeur dans une plage donnée
 - Retourne la borne inférieure si la valeur lui est inférieure
 - Retourne la borne supérieure si la valeur lui est supérieure
 - Retourne la valeur sinon

```
clamp(1, 18, 42);    // 18  
clamp(54, 18, 42);   // 42  
clamp(25, 18, 42);   // 25
```



std::to_chars() et std::from_chars

- Conversions entre chaînes C pré-allouées et nombre

```
char str[25];  
to_chars(begin(str), end(str), 12.5);  
  
double val;  
from_chars(begin(str), end(str), val);
```

- Retournent un pointeur sur la partie non utilisée de la chaîne
- Et un code erreur

API bas-niveau

- Pas d'exception, pas de gestion mémoire, pas de locale



std::variant

- Union *type-safe* contenant une valeur d'un type choisi parmi n
- Type contenu dépend de la valeur assignée

Restrictions

- Ne peut pas contenir de références, de tableaux C, `void` ni être vide
- `std::variant` *default-constructible* seulement si le premier type l'est

std::monostate

- Permet d'émuler des `std::variant` vides
- Rend un `std::variant` *default constructible*

Do

- Préférez `std::variant` aux unions brutes

std::variant

- `get<>()` récupère la valeur depuis l'index ou le nom du type
- Et lève une exception si le type demandé n'est pas correct
- `get_if<>()` retourne un pointeur sur la valeur ou `nullptr`
- `std::holds_alternative<>()` teste le type contenu
- `index()` retourne l'index d'un type donnée
- Construction en-place

```
variant<int, float, string> v{in_place_index<0>, 10};
```

std::variant

```
variant<int, float, string> v, w;  
v = "xyzy";           // string  
v = 12;                // int  
  
int i = get<int>(v); // ok  
  
w = get<int>(v);       // ok, assignation  
w = get<0>(v);         // ok, assignation  
w = v;                // ok, assignation  
  
get<double>(v);        // erreur de compilation  
get<3>(v);             // erreur de compilation  
  
get<float>(w);         // exception : w contient un int
```

std::variant

- `std::visit()` permet l'appel sur le type réellement contenu

```
vector<variant<int, string>> v{5, 10, "hello"};  
  
for(auto item : v)  
    visit([](auto&& arg){cout << arg;}, item);
```

Attention

- Appelable valide pour tous les types du `std::variant`

En attendant C++17

- Utilisez `Boost.Variant`



Pack expansion sur using

- Expansion du *parameter pack* dans les *using declaration*

```
struct Foo {  
    int operator()(int i) { return 10 + i; }  
};  
  
struct Bar {  
    int operator()(const string& s) { return s.size(); }  
};  
  
template <typename... Ts> struct Baz : Ts... {  
    using Ts::operator()...;  
};  
  
Baz<Foo, Bar> baz;  
baz(5);           // 15  
baz("azerty");   // 6
```



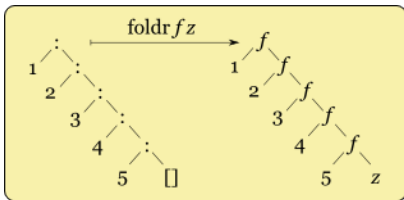
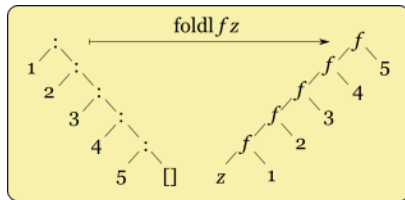
Fold expression

- Application d'un opérateur binaire à un *parameter pack*
- Support du *right fold* et du *left fold*

```
(pack op ...); // right fold
(... op pack); // left fold
```

- Éventuellement avec un valeur initiale

```
(pack op ... op init);
(init op ... op pack);
```



Fold expression

```
template<typename... Args>
bool all(Args... args) { return (... && args); }

bool b = all(true, true, true, false);
// ((true && true) && true) && false
```

```
template<typename... Args>
long long sum(Args... args) { return (args + ...); }

long long b = sum(1, 2, 3, 4);
// 1 + (2 + (3 + 4))
```

Fold expression

left fold ou right fold ?

```
template<typename... Args>  
double div(Args... args) { return (... / args); }
```

```
div(1.0, 2.0, 3.0);      // 0.166667  
// (1.0 / 2.0) / 3.0
```

```
template<typename... Args>  
double div(Args... args) { return (args / ...); }
```

```
div(1.0, 2.0, 3.0);      // 1.5  
// 1.0 / (2.0 / 3.0)
```

Fold expression

- Si le *parameter pack* est vide, le résultat est
 - `true` pour l'opérateur `&&`
 - `false` pour l'opérateur `||`
 - `void()` pour l'opérateur `,`

Attention

- Un *parameter pack* vide est une erreur pour les autres opérateurs

Fold expression

- Compatible avec des opérateurs non arithmétiques ni logiques

```
template<typename ...Args>
void FoldPrint(Args&&... args) {
    (cout << ... << forward<Args>(args)) << '\n'; }

FoldPrint(10, 'a', "ert"s);
```

- Y compris « , » qui va donner une séquence d'actions

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args) {
    (v.push_back(args), ...); }

vector<int> foo;
push_back_vec(foo, 10, 20, 56);
```



Contraintes du range-based for loop

- Utilisation possible de types différents pour `begin` et `end`
- Permet de traiter des paires d'itérateurs
- ... mais aussi un itérateur et une taille
- ... ou un itérateur et une sentinelle de fin
- Compatible avec les travaux sur Range TS

Héritage de constructeurs

- Visibilité des constructeurs hérités avec leurs paramètres par défaut
- Comportement identique aux autres fonctions héritées

Attention

- Casse du code C++11 valide

```
struct Foo { Foo(int a, int b = 0); };  
struct Bar : Foo { Bar(int a); using Foo::Foo; };  
struct Baz : Foo { Baz(int a, int b = 0); using Foo::Foo; };
```

```
Bar bar(0); // Ambigu (OK en C++11)
```

```
Baz baz(0); // OK (Ambigu en C++11)
```

noexcept

- `noexcept` fait partie du type des fonctions

```
void use_func(void (*func)() noexcept);  
void my_func();  
  
use_func(&my_func);           // Ne compile plus
```

- Les fonctions `noexcept` peuvent être convertie en fonctions non `noexcept`

std::uncaught_exceptions()

- Retourne le nombre d'exceptions lancées (ou relancées) et non encore attrapées du thread courant

```
if(uncaught_exceptions())  
{ ... }
```

Motivation

- Comportement différent d'un destructeur en présence d'exception

Caractères littéraux UTF-8

- Caractère UTF-8 préfixé par `u8`
- Erreur si le caractère n'est pas représentable par un unique code point UTF-8

```
char x = u8'x';
```

Déduction de template dans les constructeurs

- Déduction des paramètres templates d'une classe à la construction
- Plus de déclaration explicite des paramètres templates
- Ni de *make helpers*

```
pair<int, double> p(2, 4.5);  
auto t = make_tuple(4, 3, 2.5);
```

// Devient

```
pair p(2, 4.5);  
tuple t(4, 3, 2.5);
```

Déduction de template dans les constructeurs

- Permet de fournir une lambda en paramètre template sans la déclarer

```
template<class Func> struct Foo {  
    Foo(Func f) : func(f) {}  
    Func func;  
};  
  
Foo([&](int i) { ... });
```



Déduction de template dans les constructeurs

Note

- Rend obsolète plusieurs *make helper*

Attention

- Ne permet pas la déduction partielle

```
tuple<int> t(1, 2, 3); // Erreur
```

template <auto>

- Dédution du type des paramètres templates numériques

```
template <auto value> void foo() {}  
foo<10>(); // int
```

```
template <typename Type, Type value>  
constexpr Type F00 = value;  
constexpr auto const foo = F00<int, 100>;
```

// Devient

```
template <auto value> constexpr auto F00 = value;  
constexpr auto const foo = F00<100>;
```



Template & contraintes d'utilisation

- `typename` autorisé dans les déclarations de *template template parameters*

```
template <template <typename> typename C, typename T>
//
struct Foo { C<T> data; };

foo<std::vector, int> bar;
```

Template & contraintes d'utilisation

- Évaluation constante de tous les arguments templates « non-types »
- Y compris pointeurs, références, pointeurs sur membres, ...

```
template<int* P> struct Foo {  
    int operator()() { return *P; }  
};  
  
int N = 5;  
Foo<&N> foo;           // OK  
foo();                 // 5  
  
constexpr int* bar() { return &N; }  
Foo<bar()> foo2;       // OK  
foo2();               // 5
```


Capture de *this

- Capture `*this` par valeur

```
[*this]() { ... }  
[=, *this]() { ... }
```

```
struct Foo {  
    auto bar() { return [*this] { cout << s << endl; }; }  
    std::string s;  
};  
  
auto baz = Foo{"baz"}.bar();  
baz();      // Affiche baz
```

Lambdas et expressions constantes

- Lambdas autorisées dans les expressions constantes
- Si l'initialisation de chaque capture est possible dans l'expression constante

```
constexpr int AddEleven(int n) {  
    return [n] { return n+11; }();  
}
```

```
AddEleven(5);    // 16
```

Lambdas et expressions constantes

- Déclaration `constexpr` de lambda possible
- Explicitement via `constexpr`

```
auto ID = [] (int n) constexpr { return n; };  
constexpr int I = ID(3);
```

- Implicitement `constexpr` lorsque les exigences sont satisfaites

```
auto ID = [] (int n) { return n; };  
constexpr int I = ID(3);
```

Lambdas et expressions constantes

- Fermeture de type littéral si les données sont des littéraux

```
constexpr auto add = [] (int n, int m) {  
    auto L = [=] { return n; };  
    auto R = [=] { return m; };  
    return [=] { return L() + R(); };  
};  
  
add(3, 4)()    // 7
```



std::invoke()

- Appelle l'appelable fourni en paramètre
- ... en fournissant la liste de paramètres
- ... et en retournant le retour de l'appelable

```
int foo(int i) {  
    return i + 42;  
}
```

```
invoke(&foo, 8); // 50
```

std::invoke()

- Fonctionne également avec des fonctions membres
- ... le premier paramètre fourni est l'objet à utiliser

```
struct Foo {  
    int bar(int i) { return i + 42; }  
};
```

```
Foo foo;  
invoke(&Foo::bar, foo, 8); // 50
```

Motivation

- Syntaxe unique d'appel d'appelable



std::not_fn()

- Construction de *function object* en niant un callable

```
bool LessThan10(int a) { return a < 10; }  
  
vector foo = { 1, 6, 3, 8, 14, 42, 2 };  
count_if(begin(foo), end(foo), not_fn(LessThan10)); // 2
```

Dépréciation

- Dépréciation de std::not1 et std::not2

Alias de traits

- Ajout du suffixe `_v` aux traits de la forme `is_...`
- Suppression de `::value`

```
template <typename T>  
enable_if_t<is_integral<T>::value, T>  
sqrt(T t);
```

// Devient

```
template <typename T>  
enable_if_t<is_integral_v<T>, T>  
sqrt(T t);
```


Nouveaux traits

- Nouveaux traits

- `is_swappable_with`, `is_swappable`, `is_nothrow_swappable_with` et `is_nothrow_swappable` : objets échangeables
- `is_callable` et `is_nothrow_callable` : objet callable
- `void_t` conversion en `void`

- Méta-fonctions sur les traits

- `std::conjunction` : et logique entre traits
- `std::disjunction` : ou logique entre traits
- `std::negation` : négation d'un trait

```
// foo disponible si tous ls Ts... ont le meme type  
template<typename T, typename... Ts>  
enable_if_t<conjunction_v<is_same<T, Ts>...>>  
foo(T, Ts...) {}
```

Gestion des attributs

- Usage étendu aux déclarations de `namespace`

```
namespace [[ Attribut ]] foo {}
```

- Et aux valeurs d'une énumération

```
enum foo {  
    F00_1 [[ Attribut ]],  
    F00_2  
};
```

Gestion des attributs

- Attributs inconnus doivent être ignorés
- `using` des attributs non standards

```
[[ nsp::kernel, nsp::target(cpu,gpu) ]]  
foo();
```

// Devient

```
[[ using nsp: kernel, target(cpu,gpu) ]]  
foo();
```

Attribut `[[fallthrough]]`

- Dans un `switch` avant un `case` ou `default`
- Indique qu'un cas se poursuit intentionnellement dans le cas suivant
- Incitation à ne pas lever d'avertissement dans ce cas

```
switch(foo) {  
    case 1:  
    case 2:  
        ...  
    [[ fallthrough ]];  
    case 3:    // Idealement : pas de warning  
        ...  
    case 4:    // Idealement : warning  
        ...  
        break;  
}
```



Attribut `[[nodiscard]]`

- Indique que le retour d'une fonction ne devrait pas être ignorée

```
[[nodiscard]] int foo() { return 5; }
```

```
foo(); // Idealement : warning
```

- Incitation à lever un avertissement dans le cas contraire

Note

- Conversion implicite en `void` pour supprimer l'avertissement

```
(void)foo();
```

Attribut `[[nodiscard]]`

- Possible sur la déclaration d'un type (classe, structure ou énumération)
- Indique qu'un retour de ce type ne devrait jamais être ignoré

```
struct [[nodiscard]] Bar {};  
Bar baz() { return Bar{}; }
```

```
baz(); // Idéalement : warning
```



Attribut `[[maybe_unused]]`

- Sur une classe, structure, fonction, variable, paramètre, ...
- Indique qu'un élément peut ne pas être utilisé
- Incitation à ne pas lever d'avertissement en cas de non-utilisation

```
[[ maybe_unused ]]  
int foo([[ maybe_unused ]] int a,  
        [[ maybe_unused ]] long b) {}
```

- Ne devrait pas lever d'avertissement en cas d'utilisation

Avant C++17

- Ne pas nommer les paramètres non utilisés



Attributs C++17 - Conclusion

Do

- Utilisez les attributs pour indiquer vos intentions

Au delà du compilateur

- Prise en compte par d'autres outils souhaitable

std::shared_mutex

- Similaire à `std::mutex` avec deux niveaux d'accès
 - Exclusif : possible si le verrou n'est pas pris
 - Partagé : possible si le verrou n'est pas pris en exclusif
- API identique à `std::mutex` pour l'accès exclusif
- API similaire pour l'accès partagé
 - `lock_shared`
 - `try_lock_shared`
 - `unlock_shared`

Note

- Équivalent *non-timed* de `std::shared_timed_mutex`

std::scoped_lock

- Acquisition de plusieurs mutex

```
mutex first_mutex;  
mutex second_mutex;  
  
scoped_lock lck(first_mutex, second_mutex);
```

std::apply()

- Appel de fonction depuis un *tuple-like* d'argument

```
void foo(int a, long b, string c) { ... }  
  
tuple bar{42, 5L, "bar"s};  
apply(foo, bar);
```

- Fonctionne sur tout ce qui supporte std::get() et std::tuple_size
- Notamment std::pair et std::array

```
array<int, 3> baz{1, 54, 3};  
apply(foo, baz);
```

- std::make_from_tuple() permet de construire un objet depuis un *tuple-like*



std::optional

- Gestion d'objet dont la présence est optionnelle

Restriction

- Ne peut pas contenir des références, des tableaux C, `void` ni être vide
- Interface similaire à un pointeur
 - Testable via `operator bool()`
 - Accès à l'objet via `operator*`
 - Accès à un membre via `operator->`

Attention

- `operator*` ou `operator->` indéfini sur un `std::optional` vide
 - `std::nullopt` indique l'absence de l'objet
- `value()` retourne la valeur ou lève l'exception `std::bad_optional_access`
- `value_or()` retourne la valeur ou une valeur par défaut

std::optional

- Supporte la déduction de type

```
optional foo(10); // std::optional<int>
```

- Supporte la construction en-place

```
optional<complex<double>> foo{in_place, 3.0, 4.0};
```

- Y compris depuis un std::initializer_list

```
optional<vector<int>> foo(in_place, {1, 2, 3});
```

- Existence du *helper* std::make_optional

```
auto foo = make_optional(3.0);  
auto bar = make_optional<complex<double>>(3.0, 4.0);
```

std::optional

- Changement de la valeur via `reset()`, `swap()`, `emplace()` ou `operator=`
- Comparaison naturelle des valeurs contenues

```
optional<int> deux(2), dix(10);
```

```
dix > deux;      // true
dix < deux;      // false
dix == 10;      // true
```

- En prenant en compte `std::nullopt`

```
optional<int> none, dix(10);
```

```
dix > none;      // true
dix < none;      // false
none == 10;     // false
none == nullopt; // true
```

std::optional

```
std::optional<bool>? std::optional<T*>?
```

- Utilisez des booléens « trois états » (Boost.tribool)
- Utilisez des pointeurs bruts

Do

- Préférez std::optional aux pointeurs bruts pour les données optionnelles

En attendant C++17

- Utilisez Boost.Optional



std::any

- **void*** *type-safe* contenant un objet de n'importe quel type (ou vide)
- Implémentation de *Type-erasure*
- Type contenu dépend de la valeur assignée

```
any a = 1;    // int  
a = 3.14;    // double  
a = true;    // bool
```


std::any

- Supporte la construction en-place

```
any a(in_place_type<complex<double>>, 3.0, 4.0);
```

- *Helper* std::make_any

```
any a = make_any<complex<double>>(3.0, 4.0);
```

- Changement de valeur, éventuellement de type, via l'affectation

```
std::any a = 1;  
a = 3.14;
```

- ... ou emplace()

```
a.emplace<std::complex<double>>(3.0, 4.0);
```

std::any

- `any_cast<Type>()` récupère la valeur
- ... et lève une exception si le type demandé n'est pas correct

```
any a = 1;  
any_cast<int>(a);           // 1  
any_cast<bool>(a);          // Lance bad_any_cast
```

- ou récupère l'adresse
- ... et retourne `nullptr` si le type demandé n'est pas correct

```
any a = 1;  
int* foo = any_cast<int>(&a);  
int* foo = any_cast<bool>(&a); // nullptr
```

std::any

- `reset()` vide le contenu
- `has_value()` teste la vacuité
- `type()` récupère l'information du type courant

En attendant C++17

- Utilisez `Boost.Any`



std::string_view

- Vue sur une séquence contiguë de caractères
- Quatre spécialisations standards (une par type de caractères)
- Référence non possédante sur une séquence pré-existante
- Pas de modification de la séquence depuis la vue
- Constructible depuis `std::string`, une chaîne C ou un pointeur et une taille

Attention !

- Pas de 0 terminal systématique
- La chaîne référencée doit vivre au moins aussi longtemps que la vue

std::string_view

- Accès aux caractères : `operator[]`, `at()`, `front()`, `back()`, `data()`
- Modification des bornes : `remove_prefix()` et `remove_suffix()`
- Accès à la taille et à la taille maximale : `size()`, `length()` et `max_size()`
- Test de vacuité : `empty()`
- Construction d'une chaîne depuis la vue : `to_string()`
- Copie d'une partie de la vue : `copy()`
- Construction d'une vue sur une sous-partie de la vue : `substr()`
- Comparaison avec une autre vue ou une chaîne : `compare()`
- Recherche : `find()`, `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()`, `find_last_not_of()`
- Comparaison lexicographique : `==`, `!=`, `<=`, `>=`, `<` et `>`
- Affichage : `operator<<`

std::string_view

```
string foo = "Lorem ipsum dolor sit amet";

string_view bar(&foo[0], 11);
cout << bar.size() << " - " << bar << '\n'; // 11 - Lorem ipsum

bar.remove_suffix(6);
cout << bar.size() << " - " << bar << '\n'; // 5 - Lorem
```

Performances

- Souvent meilleures que les fonctionnalités équivalentes de string
- Mais pas toujours, donc mesurez



Mémoire

- `std::shared_ptr` et `std::weak_ptr` sur des tableaux

```
std::shared_ptr<int[]> foo(new int[10]);
```

Pas de `std::make_shared()`

- `std::make_shared()` ne supporte pas les tableaux en C++17
- Évolutions des allocateurs
- Classe de gestion de pools de ressources (synchronisés ou non)

Note

- Pointeur intelligent sans responsabilité dans le TS `observer_ptr`
- Mais pas dans le périmètre accepté pour C++17

Algorithmes

- Recherche d'une séquence dans une autre
 - Trois foncteurs de recherche : *default*, Boyer-Moore et Boyer-Moore-Horspoll
 - `std::search()` encapsule l'appel à un des foncteurs
- Échantillonnage
 - `std::sample()` extrait aléatoirement *n* éléments d'un ensemble

```
string in = "abcdefgh", out;  
sample(begin(in), end(in), back_inserter(out),  
       5, mt19937{random_device{}}());
```



PGCD et PPCM

- Ajout des fonctions gcd et lcm
- Initialement prévu pour des versions ultérieures
- ... mais suffisamment simples et élémentaires pour C++17

```
gcd(12, 18);    // 6  
lcm(12, 18);    // 36
```



Filesystem TS

- Gestion des systèmes de fichiers
- Adapté à l'OS et au système de fichiers utilisés
- Manipulation des chemins et noms de fichiers

```
path foo("/home/foo");  
path bar(foo / "bar.txt");  
bar.filename();      // bar.txt  
bar.extension();     // .txt  
bar.native();        // std::string  
bar.c_str();         // const char*
```

Filesystem TS

- Manipulation des répertoires, des fichiers et de leurs méta-datas
 - Copie : `copy_file()`, `copy()`
 - Création de répertoires : `create_directory()`, `create_directories()`
 - Création des liens : `create_symlink()`, `create_hard_link()`
 - Test d'existence : `exists()`
 - Taille : `file_size()`
 - Type : `is_regular_file()`, `is_directory()`, `is_symlink()`, `is_fifo()`, `is_socket()`, ...
 - Permissions : `permissions()`
 - Date de dernière écriture : `last_write_time()`
 - Suppression : `remove()`, `remove_all()`
 - Changement de nom : `rename()`
 - Changement de taille : `resize_file()`
 - Chemin du répertoire temporaire : `temp_directory_path()`
 - Chemin du répertoire courant : `current_path()`

Filesystem TS

- Parcours de répertoires
 - Entrée du répertoire : `directory_entry`
 - Itérateurs pour le parcours
 - Parcours simple : `directory_iterator`
 - Parcours récursif : `recursive_directory_iterator`
 - Construction de l'itérateur de début depuis le chemin du répertoire
 - Construction de l'itérateur de fin par défaut
- `std::fstream` constructible depuis `path`

Do

- Utilisez *Filesystem* plutôt que les API C ou systèmes

En attendant C++17

- Utilisez `Boost.Filesystem`

Parallelism TS

- Surcharges parallèles de nombreux algorithmes standards
- Politiques d'exécution (séquentielle, parallèle et parallèle+vectorisée)

```
void bar(int i);  
  
vector<int> foo {0, 5, 42, 58};  
for_each(execution::par, begin(foo), end(foo), bar);
```

Attention

- Pas de gestion intrinsèque des accès concurrents

Parallelism TS

- `std::for_each_n()` : variante de `std::for_each()` prenant l'itérateur de début et une taille et non une paire d'itérateurs
- `std::reduce()` « ajoute » tous les éléments de l'ensemble

`std::reduce()` OU `std::accumulate()` ?

- Ordre des « additions » non spécifié dans le cas de `std::reduce()`

Parallelism TS

- `std::exclusive_scan()` construit un ensemble où chaque élément est égal à la somme des éléments de rang strictement inférieur de l'ensemble initial et d'une valeur initiale

```
vector<int> foo {5, 42, 58}, bar;  
  
exclusive_scan(begin(foo), end(foo), back_inserter(bar), 8);  
// bar : 8 13 55
```

Parallelism TS

- `std::inclusive_scan()` construit un ensemble où chaque élément est égal à la somme des éléments de rang inférieur ou égal de l'ensemble initial et d'une valeur initiale (si présente)

```
vector<int> foo {5, 42, 58};  
vector<int> bar;  
  
inclusive_scan(begin(foo), end(foo), back_inserter(bar), 8);  
// bar : 13 55 113
```


Parallelism TS

- `std::transform_reduce()` : `std::reduce()` sur des éléments préalablement transformés
- `std::transform_exclusive_scan()` : `std::exclusive_scan()` sur des éléments préalablement transformés
- `std::transform_inclusive_scan()` : `std::inclusive_scan()` sur des éléments préalablement transformés

Note

- Transformation non appliquée à la graine



Mathematical Special Functions

- Une longue histoire datant du TR1
- Ajout de fonctions mathématiques particulières
 - Fonctions cylindriques de Bessel
 - Fonctions de Neumann
 - Polynômes de Legendre
 - Polynômes de Hermite
 - Polynômes de Laguerre
 - ...

Sommaire

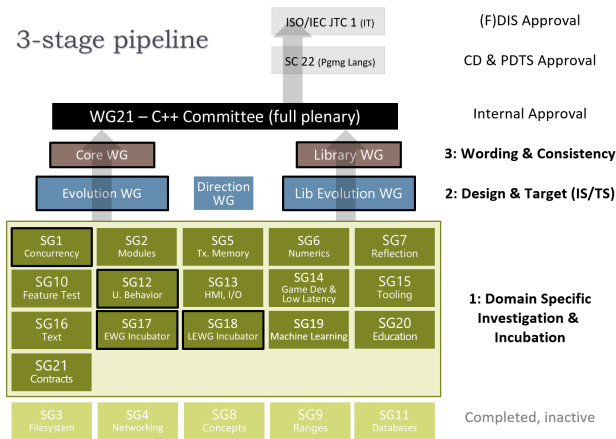
- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Approuvé en décembre 2020
- Dernier Working Draft : N4861
- Très bon support par GCC, Clang et Visual C++

Changements d'organisation du comité

- Création d'un *Direction Group*
- Création d'un *Study Group* pour l'éducation (SG20)



Dépréciations et suppressions

- Dépréciation du terme POD et de `std::is_pod()`
- Dépréciation partielle de `volatile`
- Dépréciation de l'usage de l'opérateur virgule dans les expressions d'indilage
- Suppression des membres dépréciés de `std::reference_wrapper` :
`result_type`, `argument_type`, `first_argument_type` et `second_argument_type`

Fonctionnalités

- `__has_cpp_attribute` teste le support d'un attribut
 - Similaire à `__has_include` pour la présence d'entête
 - Extensible aux attributs propriétaires d'une implémentation
- Macros testant le support de fonctionnalité du langage
 - `__cpp_decltype` : support de `decltype`
 - `__cpp_range_based_for` : support du *range-based for loop*
 - `__cpp_static_assert` : support de `static_assert`
 - ...
- Macros testant le support de fonctionnalités par la bibliothèque standard
 - `__cpp_lib_any` : support de `std::any`
 - `__cpp_lib_chrono` : support de `std::chrono`
 - `__cpp_lib_gcd_lcm` : support des fonctions `std::gcd()` et `std::lcm()`
 - ...

Valorisation

Année et mois de l'acceptation dans le standard ou de l'évolution



Information à la compilation

- Entête `<version>` : informations de version
 - Contenu *implementation-dependent*
 - Version du standard, de la bibliothèque, *release date*, copyright, ...
- `source_location` : position dans le code source
 - Fichier, ligne, colonne et fonction courante
 - Contenu *implementation-dependent*
 - Remplaçant de `__LINE__`, `__FILE__`, `__func__` et autres macros propriétaires



Compilation conditionnelle

- Ajout d'un paramètre booléen, optionnel, à `explicit`
 - Pilotage de `explicit` via un paramètre booléen *compile-time*
 - Possibilité de rendre des constructeurs templates explicites ou non en fonction de l'instanciation
 - Alternative à des constructions à base de macros de compilation ou de SFINAE

Types entiers

- Types entiers signés obligatoirement en compléments à 2

Situation pré-C++20

- Pas de contrainte en C++
- 3 choix en C : signe+mantisse, complément à 1 et complément à 2

Rupture de compatibilité ?

- En pratique, toutes les implémentations actuelles sont en complément à 2
- Précision de comportements sur des types entiers signés
 - Conversion vers non signé est toujours bien définie
 - Décalage à gauche : même résultat que celui du type non signé correspondant
 - Décalage à droite : décalage arithmétique avec extension du signe

Caractères

- Contraintes de `char16_t` et `char32_t` : caractères UTF-16 et UTF-32
- `char8_t` pour les caractères UTF-8
 - Pendant UTF-8 de `char16_t` et `char32_t`
 - Similaire en terme de taille, d'alignement, de conversion à `unsigned char`
 - Pas un alias sur un autre type
 - Prise en compte dans la bibliothèque standard
- Type `u8string` pour les chaînes UTF-8

Motivation

- Suppression de l'ambigüité caractère UTF-8 / littéral
- Suppression d'ambigüité sur les surcharges et spécialisation de template

Définition d'agrégat

- Modification de la définition d'agrégat :
 - C++17 : pas de constructeur *user-provided*
 - C++20 : pas de constructeur *user-declared*

```
// Agregat en C++17 pas en C++20  
struct S {  
    S() = default;  
};
```

Initialisation des agrégats

- Initialisation nommée des membres d'un agrégat ou d'une union

```
struct S { int a; int b; int c; };  
S s{.a = 1, .c = 2};
```

```
union U { int a; char* b };  
U u{.b = "foo"};
```

Restrictions

- Uniquement sur les agrégats et les unions
- Initialisation des champs dans leur ordre de déclaration
- Initialisation d'un unique membre d'une union



Initialisation des agrégats

- Initialisation des agrégats via des données parenthésées

ou {} ?

- {} permet l'utilisation d'*initializer list*
- () permet les conversions avec perte de précision

Motivations

- Fonctions transférant les arguments à un constructeur sur des agrégats
- Initialisation par défaut des champs de bits

```
struct S {  
    int a : 1 {0},  
    int b : 1 = 1;  
};
```

Endianness

- Énumération `std::endian`
 - `little` : *little-endian*
 - `big` : *big-endian*
 - `native` : *endianness* du système

using enum

- Utilisation d'`using` sur une `enum class`

```
enum class color { red, green, blue };  
using enum color;  
  
if(c1 == green) { ... }
```

- Sur une valeur de l'énumération

```
enum class color { red, green, blue };  
using enum color::green;  
  
if(c1 == green) { ... }
```

- Sur une *unscoped* `enum`

Conversion pointeur-booléen

- Conversion pointeur vers booléen devient *narrowing*
- `nullptr` reste autorisé dans les initialisations directes

```
struct Foo {  
    int i;  
    bool b;  
};  
  
void* p;  
Foo foo{1, p};           // erreur  
bool b1{p};              // erreur  
bool b2 = p;             // OK  
bool b3{nullptr};        // OK  
bool b4 = nullptr;       // erreur  
bool b5 = {nullptr};     // erreur  
if(p) { ... }           // OK
```

Spécifications d'exception et `=default`

- Définition possible de spécifications d'exception des fonctions `=default` différentes de celles de la fonction implicite

```
struct S {  
    // Valide en C++20  
    // Invalide en C++17 (constructeur implicite noexcept)  
    S() noexcept(false) = default;  
};
```

Sémantique de déplacement

- Davantage de déplacements possibles

```
unique_ptr<T> f0(unique_ptr<T> && ptr) { return ptr; }

string f1(string && x) { return x; }

struct Foo{};

void f2(Foo w) { throw w; }

struct Bar { B(Foo); };

Bar f3() {
    Foo w;
    return w;
}
```

operator<=>

- Effectue une « *Three-way comparison* »
 - $(a <=> b) < 0$ si $a < b$
 - $(a <=> b) > 0$ si $a > b$
 - $(a <=> b) == 0$ si a et b sont équivalents
- Trois types de retour possibles
 - `std::strong_ordering` : ordre et égalité
 - `std::weak_ordering` : ordre et équivalence
 - `std::partial_ordering` : ordre partiel
- Peut être généré par le compilateur (=default)
 - `operator<=>` des bases et membres
 - `operator==` et `operator>`

operator<=>

- `operator<=>` déclenche la génération
 - Opérateurs d'ordre (<, <=, > et >=) via `operator<=>`
 - `operator==` via `operator==` des bases et membres
 - `operator!=` via `operator==`

`operator==` et `operator!=`

- Uniquement pour les retours de type « égalité »
 - Pas de génération depuis `operator<=>`
-
- Possible de marquer ces autres opérateurs `=default`
 - Utilisation de l'opérateur binaire déclaré s'il existe
 - Supporté par la bibliothèque standard



Nested namespace

- Extension des *nested namespaces* aux *inline namespaces*

```
namespace A::inline B::C {  
    int i;  
}
```

// Equivalent a

```
namespace A {  
    inline namespace B {  
        namespace C {  
            int i;  
        }  
    }  
}
```

Modules – Présentation

- Alternative au mécanisme d'inclusion

Modules et `namespace`

- Ne remplace pas les `namespace`
 - Réduction des temps de compilation
 - Nouveau niveau d'encapsulation
 - Plus grande robustesse (isolation des effets des macros)
 - Meilleures prises en charge des bibliothèques par l'analyse statique, les optimiseurs, ...
 - Gestion des inclusions multiples sans garde
 - Compatible avec le système actuel d'inclusion

Bibliothèque standard

- En C++20, la bibliothèque standard n'utilise pas les modules

Modules – Interface Unit

- L'*Interface Unit* commence par un préambule
 - Nom du module à exporter
 - Suivi de l'import d'autres modules
 - Éventuellement ré-exportés par le module

```
export module foo;  
import a;  
export import b;
```

- Suivi du corps exportant des symboles via le mot-clé **export**

```
export int i;  
export void bar(int j);  
export {  
    void baz() { ... }  
    long l;  
}
```


Modules – Implementation Unit

- L'*Implementation Unit* commence par un préambule
 - Nom du module implémenté
 - Suivi de l'import d'autres modules
- Suivi du corps contenant les détails d'implémentation

```
module foo;  
void bar(int j) { return 3 * j; }
```

Note

- *Implementation Unit* a accès aux déclarations non exportées du module

Mais ...

- Mais pas les autres unités de compilation même si elles importent le module

Modules – Partitions

- Les modules peuvent être partitionnés sur plusieurs unités
- Les partitions fournissent alors un nom de partition

```
// Interface Unit  
export module foo:part;
```

```
// Implementation Unit  
module foo:part;
```

Primary Module Interface Unit

- Une et une seule *Interface Unit* sans nom de partition par module
- Un élément peut être déclaré dans une partition et défini dans une autre

Modules – Partitions

- Les partitions sont un détail d'implémentation non visibles hors du module
- Une partition peut être importée dans une *Implementation Unit*
- ... en important uniquement le nom de la partition

```
module foo;  
import :part;      // Importe foo:part  
import foo:part;   // Erreur
```

- Le *Primary Module Interface Unit* peut exporter les partitions

```
export module foo;  
export :part1;  
export :part2;
```

Modules – Export de namespace

- Un namespace est exporté s'il est déclaré **export**
- ... ou implicitement si un de ses éléments est exporté

```
export namespace A { // A est exporte
    int n;           // A::n est exporte
}

namespace B {
    export int n;    // B::n et B sont exportes
    int m;           // B::m n'est pas exporte
}
```

Modules – Export de namespace

- Les éléments d'une partie exportée d'un namespace sont exportés

```
namespace C { int n; }           // C::m est exporte  
  
export namespace C { int m; }    // mais pas C::n
```

Modules – Implémentation inline

- Interface et implémentation dans un unique fichier
- Implémentation dans un fragment `private`

```
export module m;  
struct s;  
export using s_ptr = s*;  
  
module :private;  
struct s {};
```

Restriction

- Uniquement dans une *Primary Module Interface Unit*
- Qui doit être la seule unité du module

Modules – Utilisation

- Import des modules via la directive `import`

```
import foo;  
  
// Utilisation des symboles exportés de foo
```

- Cohabitation possible avec des inclusions

```
#include <vector>  
import foo;  
#include "bar.h"
```

Modules – Code non-modulaire

- Inclusion d'en-têtes avant le préambule du module

```
module;  
#include "bar.h"  
export module foo;
```

- Ou import des en-têtes

```
export module foo;  
import "bar.h";  
import <version>;
```


Modules – Code non-modulaire

- Export possible des symboles inclus

```
module;  
#include "bar.h" // Definit X  
export module foo;  
export using X = ::X;
```

- Ou de l'en-tête dans son ensemble

```
export module foo;  
export import "bar.h";
```

Chaînes de caractères

- `std::basic_string::reserve()` ne peut plus réduire la capacité
 - Appel avec une capacité inférieure sans effet
 - Comportement similaire à `std::vector::reserve()`

Rappel

- Après `reserve()`, la capacité est supérieure ou égale à la capacité demandée
- Dépréciation de `reserve()` sans paramètre

Réduction à la capacité utile

- Utilisez `shrink_to_fit()` et non `reserve()`

Chaînes de caractères

- Ajout à `std::basic_string` et `std::string_view`
 - `starts_with()` teste si la chaîne commence par une sous-chaîne
 - `ends_with()` teste si la chaîne termine par une sous-chaîne

```
string foo = "Hello world";  
  
foo.starts_with("Hello");    // true  
foo.ends_with("monde");     // false
```

- `std::string_view` constructible depuis une paire d'itérateurs



Conteneurs associatifs

- `contains()` teste la présence d'une clé

```
map<int, string> foo{{1, "foo"}, {42, "bar"}};  
  
foo.contains(42); // true  
foo.contains(38); // false
```



Conteneurs associatifs

- Optimisation de la recherche hétérogène dans des conteneurs non-ordonnés
 - Fourniture d'une classe exposant
 - Différents foncteurs de calcul du hash
 - Tag `transparent_key_equal` sur une comparaison transparente
 - Suppression de conversions inutiles

```
struct string_hash {  
    using transparent_key_equal = equal_to<>;  
    size_t operator()(string_view txt) const {  
        return hash_type{}(txt); }  
    size_t operator()(const string& txt) const {  
        return hash_type{}(txt); }  
    size_t operator()(const char* txt) const {  
        return hash_type{}(txt); } };  
  
unordered_map<string, int, string_hash> foo = ...;  
foo.find("abc");  
foo.find("def"sv);
```

std::list et forward_list

- `remove()`, `remove_if()` et `unique()` retourne le nombre d'éléments supprimés

std::array

- `std::to_array()` construit un `std::array` depuis un tableau C

```
auto foo = to_array({1, 2, 5, 42});  
  
long foo[] = {1, 2, 5, 42};  
auto bar = to_array(foo);  
  
auto foo = to_array<long>({1, 2, 5, 42});
```

- Y compris une chaîne C

```
auto foo = to_array("foo");
```

0 terminal

- Le 0 terminal est un élément du tableau



Suppression d'éléments

- `std::erase()` supprime les éléments égaux à la valeur fournie
- `std::erase_if()` supprime les éléments satisfaisant le prédicat fourni

```
vector<int> foo {5, 12, 2, 56, 18, 33};
```

```
erase_if(foo, [](int i) { return i > 20; }); // 5 12 2 18
```

- Remplacement de l'idiome « *Erase-remove* »
- Remplacement de la fonction membre `erase()`



std::span

- Vue sur un conteneur contigu
- Similaire à `std::string_view`
- Constructible depuis
 - Conteneur
 - Couple début / taille
 - Couple début / fin
 - Range
 - Autre `std::span`

```
array<int, 5> foo = {0, 1, 2, 3, 4};  
span<int> s1{foo};  
span<int> s2(foo.data(), 3);
```

std::span

- `begin()`, `end()`, ... : itérateurs sur le `std::span`
- `size()`, `empty()` : taille et vacuité
- `operator[]`, `front()`, `back()` : accès à un élément

```
array<int, 5> foo = {0, 1, 2, 3, 4};  
span<int> bar{ foo.data(), 4 };  
  
bar.front(); // 0
```

- `first()`, `last()` : construction de *sous-span*

```
span<int> baz = bar.first(2); // 0, 1
```

- *structured binding* sur des `std::span` de taille fixe



Décalages d'éléments

- `std::shift_left()` décale les éléments vers le début de l'ensemble
- `std::shift_right()` décale les éléments vers la fin de l'ensemble
- ... retournent un itérateur vers la fin (resp. début) du nouvel ensemble

Taille et décalage

- Opération sans effet si le décalage est supérieur la taille de l'ensemble

```
vector<int> foo{5, 10, 15, 20};  
shift_left(foo.begin(), foo.end(), 2);    // 15, 20
```

```
vector<int> bar{5, 10, 15, 20};  
shift_right(bar.begin(), bar.end(), 1);    // 5, 10, 15
```



Manipulation de puissances de deux

- `std::has_single_bit()` teste si un entier est une puissance de deux
- `std::bit_ceil()` plus petite puissance de deux non strictement inférieure
- `std::bit_floor()` plus grande puissance de deux non strictement supérieure
- `std::bit_width()` plus petit nombre de bits pour représenter un entier

```
has_single_bit(4u); // true
has_single_bit(7u); // false
bit_ceil(7u);      // 8
bit_ceil(8u);      // 8
bit_floor(7u);     // 4
```

Restriction

- Uniquement sur des entiers non signés



Manipulation binaire

- `std::rotl()` et `std::rotr()` rotations binaires
- `std::countl_zero` nombre consécutif de bits à zéro depuis le plus significatif
- `std::countl_one` nombre consécutif de bits à un depuis le plus significatif
- `std::countr_zero` nombre consécutif de bits à zéro depuis le moins significatif
- `std::countr_one` nombre consécutif de bits à un depuis le moins significatif
- `std::popcount` nombre de bit à un

```
rotl(6u, 2);    // 24  
rotr(6u, 1);    // 3  
popcount(6u);  // 2
```

Restriction

- Uniquement sur des entiers non signés



Conversion binaire

- `std::bit_cast` ré-interprétation d'une représentation binaire en un autre type
 - Conversions bit-à-bit
 - Alternative plus sûre à `reinterpret_cast` ou `memcpy()`
 - Conversion `constexpr` si possible

Restriction

- Uniquement sur des types *trivially copyable*

Comparaison d'entiers

- Ajout de fonctions de comparaison d'entier : `std::cmp_equal()`, `std::cmp_not_equal()`, `std::cmp_less()`, `std::cmp_greated()`, `std::cmp_less_equal()` et `std::cmp_greater_equal()`
- Permettent la comparaison signé / non signé sans promotion

Mathématiques

- Définition des constantes mathématiques e , $\log_2 e$, $\log_{10} e$, π , $\frac{1}{\pi}$, $\frac{1}{\sqrt{\pi}}$, $\ln 2$, $\ln 10$, $\sqrt{2}$, $\sqrt{3}$, $\frac{1}{\sqrt{3}}$, γ , φ
- `std::midpoint()` : demi-somme de deux valeurs (entières ou flottantes)

Règle d'arrondi

- La demi-somme d'entiers est entière et arrondie vers le premier paramètre

```
midpoint(2, 4); // 3  
midpoint(2, 5); // 3  
midpoint(5, 2); // 4
```



Mathématiques

- `std::lerp()` interpolation linéaire entre deux valeurs flottantes

```
lerp(10, 20, 0);    // 10  
lerp(10, 20, 0.1); // 11  
lerp(10, 20, 0.2); // 12  
lerp(10, 20, 0.3); // 13  
lerp(10, 20, 0.4); // 14  
lerp(10, 20, 0.5); // 15  
lerp(10, 20, 0.6); // 16  
lerp(10, 20, 0.7); // 17  
lerp(10, 20, 0.8); // 18  
lerp(10, 20, 0.9); // 19  
lerp(10, 20, 1);   // 20
```



Évolutions de la bibliothèque standard

- Utilisation de l'attribut `[[nodiscard]]`
- Davantage de `noexcept`
- Optimisation d'algorithmes numériques via `std::move()`

Ranges – Présentation

- Abstraction de plus haut niveau que les itérateurs
- Manipulation d'ensemble au travers d'algorithmes et de *range adaptators*
- Vivent dans le namespace `std::ranges`

Pour aller plus loin

- « Iterators Must Go » d'Andrei Alexandrescu
- Blog d'Eric Niebler

Ranges – Concepts

- Range
 - Permet la manipulation uniforme des éléments d'une structure de données
 - Itérateur de début
 - Sentinelle de fin
 - Itérateur
 - Valeur particulière
 - `std::default_sentinel_t` : itérateurs gérant la limite du range
- Conteneur : range possédant ses éléments
- View
 - range ne possédant pas les éléments pointés par `begin()` et `end()`
 - Copie, déplacement et affectation en temps constant
- SizedRange : taille en temps constant
- ViewableRange : range convertible en View
- CommonRange : itérateur et sentinelle de même type

Ranges – Concepts

- `InputRange` : fournit `input_iterator`
- `OutputRange` : fournit `output_iterator`
- `ForwardRange` : fournit `forward_iterator`
- `BidirectionalRange` : fournit `bidirectional_iterator`
- `RandomAccessRange` : fournit `random_access_iterator`
- `ContiguousRange` : fournit `contiguous_iterator`

En résumé

- Conteneurs : possession, copie profonde
- Vues : référence, copie superficielle

Ranges – Itérateurs

- `std::common_iterator` : adaptateur d'itérateurs/sentinelles permettant de représenter un *non-common* range comme un `CommonRange`
- `std::counted_iterator` : adaptateur d'itérateurs reprenant le fonctionnement de l'itérateur sous-jacent mais conservant la distance à la fin du range

Ranges – Opérations

- `begin()`, `end()`, `cbegin()`, `cend()`, ... retournent itérateurs et sentinelles
- `size()` retourne la taille du range
- `empty()` teste la vacuité
- `data()` et `cdata()` retournent l'adresse de début du range

Restrictions

- `data()` et `cdata()` uniquement sur des `ContiguousRange`
- Surcharges de différents algorithmes prenant un range en paramètre

Ranges – Factory

- `std::views::empty` crée une vue vide
- `std::views::single` crée une vue d'un unique élément
- `std::views::iota` crée une vue en incrémentant une valeur initiale

```
for(int i : views::iota(1, 10))  
    cout << i << ' ';    // 1 2 3 4 5 6 7 8 9
```

- `std::views::counted` crée un range depuis un itérateur et un nombre d'éléments

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
for(int i : views::counted(a, 3))  
    cout << i << ' ';    // 1 2 3
```


Ranges – range adaptators

- Appliquent filtres et transformations aux `range`
- Associés, pour certains, à un *range adaptor closure object*
 - Prends un unique paramètre `viewable_range`
 - Retourne une `view`
- Évaluation paresseuse des `view`

Ranges – range adaptators

- Peuvent être chaînés avec une syntaxe « appel de fonction »

```
D(C(R));
```

- Ou une syntaxe « pipeline »

```
R | C | D;
```

- Peuvent prendre plusieurs arguments

```
adaptor(range, args...);  
adaptor(args...)(range);  
range | adaptor(args...);
```

Ranges – range adaptators

- Plusieurs adapteurs fournis par la bibliothèque standard
 - `all_view` : tous les éléments du range
 - `ref_view` : références sur les éléments du range
 - `filter_view` : tous les éléments satisfaisants un prédicat

```
vector<int> ints{0, 1, 2, 3, 4, 5};  
auto even = [](int i){ return (i % 2) == 0;},  
  
auto rng = ints | view::filter(even);           // 0, 2, 4
```

- `transform_view` : les éléments transformés par l'application d'une fonction

```
vector<int> ints{0, 1, 2, 3, 4, 5};  
auto double = [](int i){ return 2 * i;},  
  
auto rng = ints | view::transform(double);      // 0, 2, 4, 6, 8, 10
```

Ranges – range adaptators

- `take_view` : les N premiers éléments
- `take_while_view` : les éléments jusqu'au premier ne satisfaisant pas un prédicat
- `drop_view` : tous les éléments sauf les N premiers
- `drop_while_view` : tous les éléments depuis le premier ne satisfaisant pas un prédicat
- `common_view` convertit une vue en `common_range`
- `reverse_view` : éléments en sens inverse
- `istream_view` : vue par application successive de `operator>>` sur un flux

Ranges – range adaptators

- `join_view` « aplati » les éléments d'un range

```
vector<string> ss{"hello", " ", "world", "!"};  
join_view greeting{ss};  
for(char ch : greeting)  
    cout << ch; // hello world!
```

- `split_view` sépare un range en élément sur un délimiteur donné

```
string str{"the quick brown fox"};  
split_view sentence{str, ' '};  
for(auto word : sentence) {  
    for(char ch : word)  
        cout << ch;  
    cout << " *"; }  
// the *quick *brown *fox *
```

Ranges – range adaptators

- `elements_view` : vue des N^e éléments de chaque *tuple* d'une vue de *tuple-likes*

```
auto figures = map {  
    {"Lovelace"s, 1815}, {"Turing"s, 1912},  
    {"Babbage"s, 1791}, {"Hamilton"s, 1936}  
};  
  
auto years = figures | views::elements<1>; // 1791 1936 1815 1912
```

- `keys_view` : vue des clés de chaque `std::pair` d'une vue de `std::pair`
- `values_view` : vue des valeurs de chaque `std::pair` d'une vue de `std::pair`
- Possible d'utiliser les algorithmes opérants sur les `range`

Ranges – Examples

```
vector<int> ints{0, 1, 2, 3, 4, 5};  
  
auto even = [](int i){ return (i % 2) == 0; };  
auto square = [](int i) { return i * i; };  
  
for(int i : ints | views::filter(even) | views::transform(square))  
    cout << i << ' '; // 0 4 16
```



Ranges – Projections

- Paramètres des algorithmes pré-traitant les éléments du range

```
vector foo{-1, 2, -3, 4, -5, 6};  
  
sort(foo, {}, [](int i) { return abs(i); }); // -1 2 -3 4 -5 6
```

- Ou extrayant une données des éléments du range

```
struct Task { string desc;  
               unsigned int priority { 0 }; };  
  
vector<Task> tasks { {"AAA", 10}, {"BBB", 5},  
                    {"CCC", 8}, {"DDD", 12} };  
  
sort(tasks, greater{}, &Task::priority); // DDD, AAA, CCC, BBB
```


Gestion des flux

- Flux synchrones

- Classe tampon synchrone : `std::basic_syncbuf`
- Classe flux bufferisé synchrone : `std::basic_ostream`
- `emit()` transfère le tampon vers le flux de sortie

```
{ ostream s(cout);  
  s << "Hello," << '\n'; // no flush  
  s.emit(); // characters transferred, cout not flushed  
  s << "World!" << endl; // flush noted, cout not flushed  
  s.emit(); // characters transferred, cout flushed  
  s << "Greetings." << '\n'; // no flush  
} // characters transferred, cout not flushed
```

- Limitation de la taille lue dans les flux avec `std::setw()`

```
// Seuls 24 caracteres sont lus  
cin >> setw(24) >> a;
```

std::format – Présentation

- API de formatage inspiré de la bibliothèque {fmt}

Motivations

- Formatage « à la C » non extensible et peu sûr
- Flux complexes, peu performants, peu propices à l'internationalisation et la localisation, formateurs globaux
- Formatage *locale-specific* ou *locale-independent*
- Format sous forme de chaînes utilisant {} comme *placeholder*

En attendant C++20

- Utilisez {fmt} ou Boost.Format

Voir aussi

- Overload 166

std::format – API

- `format()` retourne une chaîne

```
format("{} ", "a"); // "a "
```

- `format_to()` formate dans un `output_iterator`

```
vector<char> foo;
```

```
format_to(back_inserter(foo), "{} ", "a");
```

- `format_to_n()` formate dans un `output_iterator` avec une taille limite

```
array<char, 4> foo;
```

```
format_to_n(foo.data(), foo.size(), "{} ", "a");
```

std::format – API

- `formatted_size()` retourne la taille nécessaire au formatage
- `vformat()` et `vformat_to()` arguments regroupés dans un *tuple-like*

```
vformat("{} ", make_format_args("a"));
```

- Variantes `wchar` et `locale`

std::format – Placeholder

- Format général : `{[arg-id][:format-spec]}`
 - `arg-id` : index, optionnel, de l'argument de la liste de paramètres
 - `format-spec` : spécifications, optionnelles, du format

Séquences d'échappement

- `{{` affiche `{`
- `}}` affiche `}`

std::format – Identifiant d'arguments

- Valeur optionnelle indiquant l'index du paramètre à afficher
- Débute à 0

```
format("{1} et {0}", "a", "b"); // "b et a"  
format("{0} et {0}", "a");      // "a et a"
```

- En cas d'absence, les paramètres sont pris dans l'ordre d'apparition

```
format("{} et {}", "a", "b"); // "a et b"
```

Limite

- Impossible d'en omettre que certains

std::format – Spécification de format

- Format général : `[[fill]align][sign][#][0][width][prec][L][type]`
 - `fill` et `align` : gestion de l'alignement
 - `sign` : gestion du signe
 - `#` : forme alternative
 - `0` : gestion des zéros non significatifs
 - `width` : taille minimal du champ
 - `prec` : précision du champ
 - `L` : prise en compte de la locale
 - `type` : type à afficher

std::format – Alignement

- Alignement par défaut dépendant du type

```
format("{:6}", 42);    // "    42"  
format("{:6}", 'x');   // "x     "
```

- Fourniture du caractère de *padding*

```
format("{:06}", 42);   // "000042"
```

- Choix de l'alignement

```
format("{:*<6}", 'x'); // "x*****"  
format("{:*>6}", 'x'); // "*****x"  
format("{:*^6}", 'x'); // "**x***"
```


std::format – Taille minimale

- Fournit la taille minimal du champ
- Si le champ est plus long, il n'est pas tronqué

```
// "| 10| |          10|"
format("|{0:4}| |{0:12}|", 10);
// "|10000000| |    10000000|"
format("|{0:4}| |{0:12}|", 1000000);
```

- Possible de fournir la taille en paramètre via un *placeholder*

```
// "| 10| |          10|"
format("|{0:{1}}| |{0:{2}}|", 10, 4, 12);
```

std::format – Précision

- Introduit par un .
- Uniquement sur
 - Les nombres flottants

```
format("{:.6f}", 392.65); // "392.650000"
```

- Les chaînes de caractères : troncature

```
format("{:.6}", "azertyuiop"); // "azerty"
```

- Possible de fournir la taille en paramètre via un *placeholder*

std::format – Signe

- Uniquement sur les négatifs : '-'
- Sur toutes les valeurs : '+'
- Uniquement sur les négatifs en réservant l'espace : ''

```
format("{0:},{0:},{0:-},{0: }", 1);    // "1,+1,1, 1"  
format("{0:},{0:},{0:-},{0: }", -1);   // "-1,-1,-1,-1"
```

std::format – Zéros non significatifs

- Affichage des zéros non significatifs

```
format("{: +06d}", 120); // "+00120"
```

std::format – Format

- Entiers : décimal, octal, binaire ou hexadécimal

```
format("{:d}", 42);           // "42"  
format("{:x} {:X}", 42, 42); // "2a 2A"  
format("{:b}", 42);          // "101010"  
format("{:o}", 42);          // "52"
```

- Caractères : valeur numérique ou caractère

```
format("{:X}", 'A');          // "41"  
format("{:c}", 'A');          // "A"
```

- Booléens : chaîne ou nombre

```
format("{:d}", true);         // "1"  
format("{:s}", true);         // "true"
```

std::format – Format

- Flottants : fixe, court, scientifique ou hexadécimal

```
format("{:.6f}", 392.65);    // "392.650000"  
format("{:.6g}", 392.65);    // "392.65"  
format("{:.6e}", 392.65);    // "3.9265e+02"  
format("{:.6E}", 392.65);    // "3.9265E+02"  
format("{:.6a}", 42.);       // "1.500000p+5"
```

- Chaîne de caractère

```
format("{:s}", "azerty");    // "azerty"
```

std::format – Forme alternative

- Affichage de la base des entiers

```
format("{:#x}", 42);      // "0x2a"  
format("{:#X}", 42);      // "0X2a"
```

- Affichage du point décimal et de l'ensemble de la précision des flottants

```
format("{:.6g}", 392.65);  // "392.65"  
format("{:#.6g}", 392.65); // "392.650"
```



std::format – Dates et heures

- Format basé sur `strftime`
 - `%y` : année sur deux digits
 - `%m` : mois
 - `%d` : jour dans le mois
 - `%u`, `%w` : jour dans la semaine
 - `%H`, `%I` : heure (format 24h ou 12h)
 - `%M` : minutes
 - `%S` : secondes
 - ...

```
format("{:%F %T}", chrono::system_clock::now());  
// AAAA-MM-JJ HH:mm:ss"
```


std::format – Gestion des erreurs

- Exception `std::format_error`
 - Chaîne de format invalide
 - Spécificateurs non cohérents avec le type fournit
 - Absence de valeur
 - Exception levée par un formateur

Valeur surnuméraire

- Les valeurs surnuméraires ne sont pas des erreurs et sont ignorées

std::format – Types utilisateur

- Par spécialisation de `std::formatter<>`

```
template<>
struct formatter<T> {
    template <class ParseContext>
    auto parse(ParseContext& parse_ctx);

    template <class FormatContext>
    auto format(const T& value, FormatContext& fmt_ctx);
};
```

std::format – Types utilisateur

```
struct MyComplex { double real; double imag; };

template <>
struct formatter<MyComplex> {
    constexpr auto parse(format_parse_context& ctx) {
        return ctx.begin();
    }

    auto format(const MyComplex& value, format_context& ctx) {
        return format_to(ctx.out(), "{}+{}i", value.real, value.imag);
    }
};

format("{} ", MyComplex{1, 2}); // "(1+2i)"
```

Tableaux

- Support des tableaux par `std::make_shared()`

```
shared_ptr<double[]> foo = make_shared<double[]>(1024);
```

- Dédution de la taille des tableaux par `new()`

```
double* a = new double[]{1, 2, 3};
```

Destruction

- `std::destroying_delete_t` : pas de destruction avant l'appel à `delete()`

Intérêt

- Conserver des informations nécessaire à la libération

```
struct Foo {  
    void operator delete(Foo* ptr, destroying_delete_t) {  
        const size_t realSize = ...;  
        ptr->~Foo();  
        ::operator delete(ptr, realSize);  
    }  
};
```

Ne pas oublier

- La destruction doit être appelée explicitement

Horloges

- Nouvelles horloges
 - `std::chrono::utc_clock`
 - Temps universel coordonné
 - Epoch : 1 janvier 1970 00:00:00
 - Support des secondes intercalaires
 - `std::chrono::gps_clock`
 - Epoch : 6 janvier 1980 00:00:00 UTC
 - Pas de seconde intercalaire
 - `std::chrono::tai_clock`
 - Temps atomique universel
 - Epoch : 31 décembre 1957 23:59:50 UTC
 - Pas de seconde intercalaire
 - `std::chrono::file_clock` : alias vers le temps du système de fichier



Horloges

- Conversion des horloges vers et depuis UTC
- Conversion de `std::chrono::utc_clock` vers et depuis le temps système
- Conversion des horloges entre-elles

Conversion de `std::chrono::file_clock`

- Support optionnel des conversions entre `std::chrono::file_clock` et `std::chrono::utc_clock` ou `std::chrono::system_clock`
- Pseudo-horloge `std::chrono::local_t` temps dans la *timezone* locale

Évolution de `std::chrono::duration`

- *Helper* pour le jour, la semaine, le mois ou l'année
- `to_stream()` affiche une `std::chrono::duration`
- `from_stream()` lit une `std::chrono::duration`
- Utilisation de chaîne de format utilisant des séquences préfixées par %
 - `%H,%I` : heure (format 24h ou 12h)
 - `%M` : minutes
 - `%S` : secondes
 - `%Y, %y` : année (4 ou 2 chiffres)
 - `%m` : numéro du mois
 - `%b, %B` : nom du mois dans la locale (abrégé ou complet)
 - `%d` : numéro du jour dans le mois
 - `%U` : numéro de la semaine
 - `%Z` : abréviation de la *timezone*
 - ...

Calendrier

- Gestion du calendrier grégorien
 - Différentes représentations
 - Année, mois
 - Jour dans l'année, dans le mois
 - Dernier jour du mois
 - Jour dans la semaine, n^{e} jour de la semaine dans le mois

Convention anglo-saxonne

- Le premier jour de la semaine est le dimanche
 - Et différentes combinaisons permettant de construire une date complète

Calendrier

- Constantes représentant les jours de la semaine et les mois
- Suffixes littéraux y et d pour les années et les jours
- `operator/` pour construire une date depuis un format humain

```
auto date1 = 2016y/May/29d;  
auto date2 = Sunday[3]/May/2016y;
```



Timezone

- Gestion des *timezones*
 - Gestion de la base de *timezones* de l'IANA
 - Récupération de la *timezone* courante
 - Recherche d'une *timezone* depuis son nom
 - Caractéristique d'une *timezone*
 - Informations sur les secondes intercalaires
 - Récupération du nom d'une *timezone*
 - Conversion entre *timezone*
 - Gestion des ambiguïté de conversion

```
// 2016-05-29 07:30:06.153 UTC
auto tp = sys_days{2016y/may/29d} + 7h + 30min + 6s + 153ms;
// 2016-05-29 16:30:06.153 JST
zoned_time zt = {"Asia/Tokyo", tp};
```



Date et heure

En attendant C++20

- Utilisez Boost.DateTime

Pour aller plus loin

- ICU supporte de nombreux calendriers et mécanismes de localisation

Range-based for loop

- Initialisation dans les range-based for loop

```
vector<int> foo{1, 8, 5, 56, 42};  
for(size_t i = 0; const auto& bar : foo) {  
    cout << bar << " " << i << "\n";  
    ++i;  
}
```

- Seuls des couples `begin()`, `end()` cohérents sont utilisés
 - « Début » et « début + taille »
 - fonctions membres `begin()` et `end()`
 - fonctions libres `std::begin()` et `std::end()`

Intérêt

- Itération (via des fonctions libres) d'éléments ayant une fonction membre `begin()` ou `end()` mais pas les deux



constexpr

- `constexpr` impose une évaluation *compile-time*
 - `constexpr` implique `inline`

```
constexpr int sqr(int n) { return n * n; }  
constexpr int r = sqr(100); // OK  
int x = 100;  
int r2 = sqr(x);           // Erreur
```

Restriction

- Pas de pointeur dans des contextes `constexpr`



constinit

- `constinit` impose une initialisation durant la phase *static initialization*
 - Uniquement sur des objets dont la *storage duration* est *static* ou *thread*
 - Mal-formé en cas d'initialisation dynamique
 - Adresse le *static initialization order fiasco*

constexpr

- Initialisation triviale dans des contextes `constexpr`
- `std::is_constant_evaluated()` pour savoir si l'évaluation est *compile-time*
- Prise en compte accrue dans la bibliothèque standard
- Assouplissement des restrictions
 - Fonctions virtuelles `constexpr`
 - Utilisation d'`union`
 - Utilisation de `try {} catch()`
 - Se comporte comme *no-ops* en *compile-time*
 - Ne peut pas lancer d'exception *compile-time*
 - Utilisation de `dynamic_cast` et `typeid`
 - Utilisation de `asm`
 - Si le code `asm` n'est pas évalué en *compile-time*

Structured binding

- Extension à tous les membres visibles
- Plus proche de variables classiques
 - Capture par les lambdas (copie et référence)

```
tuple foo{5, 42};  
  
auto [a, b] = foo;  
auto f1 = [a] { return a; };  
auto f2 = [=] { return b; };
```

- Déclaration `inline`, `extern`, `static`, `thread_local` ou `constexpr` possible
- Possibilité de marquer `[[maybe_unused]]`



Structured binding

- Recherche de `get()` : seules les fonctions membres templates dont le premier paramètre template n'est pas un type sont retenues

Motivation

- Utiliser des classes possédant un `get()` indépendant de l'interface *tuple-like*

```
struct X : shared_ptr<int> { string foo; };
```

```
template<int N> string& get(X& x) {  
    if constexpr(N==0) return x.foo; }  
template<> class tuple_size<X> :  
    public integral_constant<int, 1> {};  
template<> class tuple_element<0, X> {  
    public: using type = string; };
```

```
X x;  
auto& [y] = x;
```

Non-Type Template Parameters

- Utilisation possible de classes
 - *strong structural equality*
 - Classes de base et membres non statiques avec une *defaulted* **operator==**
 - Pas de référence
 - Pas de type flottant
 - Pas d'union

```
template<chrono::seconds seconds>  
class fixed_timer { ... };
```

```
template<fixed_string Id>  
class entity { ... };  
  
entity<"hello"> e;
```

Templates

- **typename** optionnel lorsque seul un nom de type est possible
- Spécialisation possible sur des classes internes privées ou protégées
- `std::type_identity<>` désactive la déduction de type

```
template<class T>
void f(T, T);

f(4.2, 0); // erreur, int ou double
```

```
template<class T>
void g(T, type_identity_t<T>);

g(4.2, 0); // OK, g<double>
```



Templates

- Dédution de type sur les alias de template

```
template<typename T>
using IntPair = std::pair<int, T>;

// C++ 17
IntPair<double> p0{1, 2.0};

// C++ 20
IntPair p1{1, 2.0};    // std::pair<int, double>
```



Paramètres auto

- Création de fonctions templates via `auto`

```
void foo(auto a, auto b) { ... };
```

- Similaire à la création de lambdas polymorphiques



Concepts – Présentation

- Histoire ancienne et mouvementée
 - Prévu initialement pour C++0x
 - ... et cause des décalages successifs
 - Retrait à grand bruit de C++11
 - Finalement Concept lite TS publié en 2015
 - Intégration du TS acceptée en juillet 2017
- Définition de contraintes sur les paramètres templates et l'inférence de type
 - Meilleurs diagnostics
 - Meilleure documentation du code
 - Aide à la déduction de type
 - Aide à la résolution de spécialisation
- Propositions abandonnées / mises de côté
 - *Axiom* : spécification de propriétés sémantiques d'un concept
 - *Concept map* : transformation d'un type non-compatible vers un concept

Concepts – Utilisation template

- Utilisable via une *Requires clause*

```
template<typename T> requires Incrementable<T>  
void foo(T);
```

- ... via une *Trailing requires clause*

```
template<typename T>  
void foo(T) requires Incrementable<T>;
```

- ... via des paramètres templates contraintes

```
template<Incrementable T>  
void foo(T);
```

- ... ou via des combinaisons de ces syntaxes

Concepts – Utilisation template

- Utilisable depuis un concept nommé

```
template<typename T> requires Addable<T>  
T add(T a, T b) { return a + b; }
```

- ... ou depuis des expressions

```
template<typename T> requires requires (T x) { x + x; }  
T add(T a, T b) { return a + b; }
```

```
template<typename T> requires (sizeof(T) > 1)  
void foo(T);
```

Concepts – Utilisation template

- Peuvent être composés

```
template<typename T>  
requires (sizeof(T) > 1 && sizeof(T) <= 4)  
void foo(T);
```

```
template<typename T>  
requires (sizeof(T) == 2 || sizeof(T) == 4)  
void foo(T);
```

Concepts – Utilisation template

- Support des *parameters pack*

```
template<Concept... T>  
void foo(T...);
```

```
template<typename... T>  
requires Concept<T> && ... && true  
void foo(T...);
```

Concepts – Utilisation inférence de type

- Contraintes sur les paramètres (lambdas et fonctions templates)

```
[] (Constraint auto a) { ... };  
void foo(Constraint auto a) { ... };
```

- Contraintes sur les types de retour

```
Constraint auto foo();  
auto bar() -> Constraint decltype(auto);
```

Concepts – Utilisation inférence de type

- Contraintes sur les variables

```
Constraint auto bar = foo();  
Constraint decltype(auto) baz = foo();
```

- Contraintes sur les *non-type template parameters*

```
template<Constraint auto S>  
void foo();
```

- Support des *parameters pack*

```
void foo(Constraint auto... T);
```

Concepts – Standard

- Nombreux concepts standards

- Relations entre types : `same_as`, `derived_from`, `convertible_to`, `common_with`, ...
- Types numériques : `integral`, `signed_integral`, `unsigned_integral`, `floating_point`, ...
- Opérations supportées : `swappable`, `destructible`, `default_constructible`, `move_constructible`, `copy_constructible`, ...
- Catégories de types : `movable`, `copyable`, `semiregular`, `regular`, ...
- Comparaisons : `boolean`, `equality_comparable`, `totally_ordered`, ...
- *Callable concepts* : `invocable`, `predicate`, `strict_weak_order`, ...
- ...

Concepts – Définition

- Peuvent être définis depuis des expressions

```
template<typename T>  
concept Addable = requires (T x) { x + x; };
```

```
template <class T, class U = T>  
concept Swappable = requires(T&& t, U&& u) {  
    swap(forward<T>(t), forward<U>(u));  
    swap(forward<U>(u), forward<T>(t)); };
```

Concepts – Définition

- Y compris en retirant des qualifieurs

```
template<class T>
concept Addable = requires(
    const remove_reference_t<T>& a,
    const remove_reference_t<T>& b) { a + b; };
```

- Ou en imposant les types de retour

```
template<class T>
concept Comparable = requires(T a, T b) {
    { a == b } -> boolean;
    { a != b } -> boolean; };
```


Concepts – Définition

- Depuis des *traits*

```
template<class T>  
concept integral = is_integral_v<T>;
```

```
template<class T, class... Args>  
concept destructible_from =  
    destructible<T> && is_constructible_v<T, Args...>;
```

Concepts – Définition

- Depuis d'autres concepts

```
template<class T> concept semiregular =  
    copyable<T> && default_constructible<T>;
```

- En combinant différentes méthodes

```
template<class T> concept totally_ordered =  
    equality_comparable<T> &&  
    requires(const remove_reference_t<T>& a,  
             const remove_reference_t<T>& b) {  
        { a < b } -> boolean;  
        { a > b } -> boolean;  
        { a <= b } -> boolean;  
        { a >= b } -> boolean;  
    };
```



Attributs

- Ajout d'attributs
 - `[[likely]]` et `[[unlikely]]` probabilité de branches conditionnelles
 - `[[no_unique_address]]` membre statique ne nécessitant pas une adresse unique
- Extension de `[[nodiscard]]` aux constructeurs
 - Marquage `[[nodiscard]]` des constructeurs autorisé
 - Vérification également lors des conversions via les constructeurs
- Possibilité d'associer un message à `[[nodiscard]]`



Lambdas

- Utilisables dans des contextes non évalués
- Utilisation de paramètres templates pour les lambdas génériques
 - En complément de la syntaxe avec `auto`
 - Permet de récupérer le type

Usage

- Spécification de contraintes sur paramètres : types identiques, itérateur, ...

```
auto foo = []<typename T>(vector<T> const& vec) {  
    cout<< size(vec) << '\n';  
    cout<< vec.capacity() << '\n';  
};
```



Lambdas

- Lambda *stateless* assignables et constructibles par défaut

```
auto greater = [](auto x, auto y) { return x > y; };  
  
map<string, int, decltype(greater)> foo;
```

- Dépréciation de la capture implicite de `this` par `[=]`
 - Capture explicite par `[=, this]`
 - Capture implicite par `[&]` toujours présente
- Capture de *structured binding*

Lambdas

- Expansion des *parameter packs* lors de la capture

```
template<class F, class... Args>
auto delay_invoke(F f, Args... args) {
    return [f=move(f),...args=move(args)]()->decltype(auto) {
        return invoke(f, args...);
    }
}
```

- Peuvent être `constexpr`

Binding

- `std::bind_front()` assigne les arguments fournis aux premiers paramètres de l'appelable

```
int foo(int a, int b, int c, int d) { return a * b * c + d; }
```

```
auto baz = bind_front(&foo, 2, 3, 4);  
baz(7); // 31
```

// Equivalent a

```
auto bar = bind(&foo, 2, 3, 4, _1);  
bar(6); // 30
```

- `std::reference_wrapper` accepte les types incomplets



std::atomic

- `std::atomic<std::shared_ptr<T>>`
- `std::atomic<>` sur les types flottant
- Initialisation par défaut de `std::atomic<>`
- `std::atomic_ref` applique des modifications atomiques sur des données non-atomiques qu'il référence
- `wait()`, `notify_one()` et `notify_all()` pour attendre le changement d'état d'un `std::atomic`

Thread

- Nouvelle variante de thread `std::jthread`
 - Automatiquement arrêté et joint lors de la destruction

```
int main() { thread t(foo); } // Erreur (terminate)
```

```
int main() { jthread t(foo); } // OK
```

- Peut être arrêté par l'appel à `request_stop()`

```
void foo(stop_token st) {  
    while(!st.stop_requested()) { ... }  
}  
  
jthread t(foo);  
...  
t.request_stop();
```



synchronisation – sémaphores

- `std::counting_semaphore`
 - Création avec la valeur maximale de possesseurs
 - `max()` retourne le nombre maximal de possesseurs
 - `release()` relâche, une ou plusieurs fois, le sémaphore
 - `acquire()` prend le sémaphore en attendant si besoin
 - `try_acquire()` tente de prendre le sémaphore et retourne le résultat de l'opération
 - `try_acquire_for()` tente de prendre le sémaphore en attendant la durée donnée si besoin
 - `try_acquire_until()` tente de prendre le sémaphore en attendant jusqu'à un temps donné si besoin
- `std::binary_semaphore` instantiation de `std::counting_semaphore` pour un unique possesseur



synchronisation – latch

- `std::latch` compteur descendant permettant de bloquer des threads tant qu'il n'a pas atteint zéro
 - Création avec la valeur initiale du compteur
 - `count_down()` décrémente le compteur
 - `try_wait()` indique si le compteur a atteint zéro
 - `wait()` attend jusqu'à ce que le compteur atteigne zéro
 - `arrive_and_wait()` décrémente le compteur et attend qu'il atteigne zéro

Pas d'incrément

- Impossible d'incrémenter un `std::latch` et de revenir à sa valeur initiale



synchronisation – barrière

- `std::barrier` attend qu'un certain nombre de threads n'atteigne la barrière
 - Création avec le nombre de threads attendus
 - `arrive()` décrémente le compteur
 - `wait()` attend que le compteur atteigne zéro
 - `arrive_and_wait()` décrémente le compteur et attend qu'il atteigne zéro
 - `arrive_and_drop()` décrémente le compteur ainsi que la valeur initiale
 - Une fois zéro atteint, les threads en attente sont débloqués et le compteur reprend la valeur initiale décrémentée du nombre de threads « *droppés* »



Politique d'exécution

- Nouvelle politique d'exécution vectorisé `std::unsequenced_policy`

std::coroutine – Présentation

- Fonction dont l'exécution peut être suspendue et reprise
- Simplification du développement de code asynchrone
- TS publié en juillet 2017

std::coroutine – Définition

- Fonctions contenant
 - `co_await` suspend l'exécution
 - `co_yield` suspend l'exécution en retournant une valeur
 - `co_return` termine la fonction
- Restrictions
 - Pas de `return`
 - Pas d'argument *variadic*
 - Pas de déduction de type sur le retour
 - Pas sur les constructeurs, destructeurs, fonctions `constexpr`

std::coroutine – Mécanismes

- *Promise* utilisée pour renvoyer valeurs et exceptions
- *Coroutine state* interne contenant promesse, paramètres, variables locales et état du point de suspension
- *Coroutine handle* non possédant pour poursuivre ou détruire la coroutine
 - `operator bool()` indique si le *handle* gère effectivement une coroutine
 - `done()` indique si la coroutine est suspendue dans son état final
 - `operator()` et `resume()` poursuit la coroutine
 - `destroy()` détruit la coroutine
- Spécialisation de *coroutine handle* sur une *promise*
 - `promise()` accès à la promesse

std::coroutine – Exemple

```
struct generator {  
    ...  
    bool next() { return cor ? (cor.resume(), !cor.done()) : false; }  
    int value() { return cor.promise().current_value; }  
  
    coroutine_handle<promise_type> cor;  
};  
  
generator f() { co_yield 1; co_yield 2; }  
  
auto g = f();  
while(g.next()) cout << g.value() << endl;
```

std::create_directory()

- Échec de std::create_directory() si l'élément terminal existe et n'est pas un répertoire

```
create_directory("a/b/c");  
// C++17  
// Erreur si a ou b existe mais ne sont pas des répertoires  
// Pas d'erreur si c existe mais n'est pas un répertoire  
  
// C++20  
// Erreur dans les deux cas
```

Constructeur de `std::variant`

- Contraintes sur le constructeur et l'opérateur d'affectation de `std::variant`
 - Pas de conversion en `bool`

```
variant<string, bool> x = "abc"; // C++17 : bool, C++20 : string
```

- Pas de *narrowing conversion*

```
variant<float, long> v;  
v = 0; // C++17 : erreur, C++20 : long
```

std::visit()

- Possibilité d'explicitement le type de retour de `std::visit()`
 - Via un paramètre template
 - Sinon déduit de l'application du visiteur au premier paramètre

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Travaux techniques terminés en février 2023
- Document final en cours de validation internationale
- Dernier *Working Draft* : n4917

Changements d'organisation du comité

- Création d'un *ABI Review Group* : étude des impacts des évolutions sur l'ABI
- Création d'un *Study Group* pour la liaison C/C++ (SG22)
- Création d'un *Study Group safety and security* (SG23)

Dépréciations et suppressions

- Suppression des fonctionnalités liées au support d'un GC
- Dépréciation de `std::aligned_storage` et `std::aligned_union`
- Dépréciation de `std::numeric_limits::has_denorm`

Espaces en fin de ligne

- Espaces ignorés après le \ de séparation de ligne

```
// Toujours une chaîne vide en C++23  
auto str = "\<space>  
";
```

Label

- Label autorisé en fin de bloc
- Reprise d'une évolution C2X

```
void foo(void) {  
    int x;  
    x = 1;  
last:  
}
```

Compilation conditionnelle

- Ajout de *#elifdef* et *#elifndef*
- Équivalents à *#elif defined* et *#elif not defined*

```
#ifdef FOO  
...  
#elifdef BAR  
...  
#endif
```

- Se combinent avec *#if* et *#elif*



Avertissement

- `#warning` génère un avertissement à la compilation



Gestion explicite de la durée de vie

- `std::start_lifetime_as` et `std::start_lifetime_as_array` indiquent qu'un objet est créé mais sans initialisation

```
struct X { int a, b; };
```

```
X* p = start_lifetime_as<X>(malloc(sizeof(struct X)));  
p->a = 1;  
p->b = 2;
```

Types flottants étendus

- `std::float16_t`, `std::float32_t`, `std::float64_t`, `std::float128_t`
 - Types IEEE N-bit
 - Support optionnel
- `std::bfloat16_t`
 - Type IEEE binary16
 - Support optionnel
- Suffixes littéraux correspondants (`f16`, `f32`, `f64`, `f128` et `bf16`)
- Prise en compte par `std::format`, `std::ostream` et `std::istream`
- Prise en compte par `std::numeric_limits` et `std::is_floating_point`
- Ajout de surcharges dans `<cmath>`, `<complex>` et `<atomic>`

Types indépendants

Types indépendants (pas d'alias) de `float`, `double` ou `long double`

Évolutions de `char8_t`

- Initialisation d'un tableau de `char` ou d'`unsigned char` depuis une chaîne littérale UTF-8

Relâchement des contraintes de `wchar_t`

- Suppression de la contrainte

The values of type `wchar_t` can represent distinct codes for all members of the largest extended character set specified among the supported locale

- Permet l'utilisation de `wchar_t` pour représenter des caractères UTF-16 ou UCS-2 sur des systèmes supportant UTF-8

Conversions

- Ajout d'une conversion implicite en booléen
 - Dans les `static_assert`
 - Dans les `if constexpr`

```
// Valide en C++23, pas en C++20  
if constexpr(flags & 0x01) { ... }  
else { ... }
```

```
// Valide en C++23, pas en C++20  
template <std::size_t N>  
class Foo { static_assert(N, "Message"); };
```

- `auto(x)` et `auto{x}` convertissent `x` en *prvalue*

Énumérations

- `std::to_underlying` convertit une énumération vers le type sous-jacent

```
enum class F00 : uint32_t { A = 0xABCDEF };  
  
auto bar = to_underlying(F00::A); // uint32_t
```



constexpr

- Relâchement de contrainte sur les fonctions `constexpr`
 - Code non évalué au *compile-time*
 - Variables non littérales
 - Utilisation de `goto`
 - Retour non littéral
 - Paramètres non littéraux
 - Appel de fonctions non `constexpr`
 - Code non évalué au *compile-time* ou utilisable dans un contexte constant
 - Variables `static` ou `thread_local`
 - Valeur non utilisée
 - Utilisation de pointeurs ou références inconnus
- Conversion implicite de fonctions `constexpr` en `constexpr`
- Davantage de `constexpr` dans la bibliothèque standard

if constexpr

- Branche prise en compte si le code est évalué au *compile-time*
- Peut appeler des fonctions immédiate
- **else** pour le code évalué au *run-time*

```
constexpr int foo(int i) { return i; }
```

```
constexpr int bar(int i) {  
    if constexpr { return foo(i) + 1; }  
    else { return 42; } }
```



if consteval

- Négation possible

```
if not consteval { ... }
```

// Ou

```
if ! consteval { ... }
```

Attention

- Accolades obligatoires, même avec une unique instruction

Évolutions de la sémantique de déplacement

- Simplification des règles de déplacement implicite
- `std::move_only_function` équivalent *move-only* de `std::function`

Durée de vie des temporaires

- Extension de la durée de vie des objets temporaires créés dans l'initialisation d'un *range-based for loop* jusqu'à la fin de la boucle

```
const vector<int>& foo(const vector<int>& t) { return t; }  
vector<int> bar( return vector<int>{1, 2, 3}; );  
  
// Valide, duree de vie du retour de bar est etendu  
for (auto e : foo(bar())) { ... }
```

init-statement

- `using` possible dans l'*init-statement* de `if`, `switch` et `for`

```
for(using T = int; T e : v)  
{ ... }
```



Encodage

- Support des fichiers sources en UTF-8
- Encodage identique entre le préprocesseur et le code C++

Suffixes littéraux

- Suffixe `uz` pour `size_t`
- Suffixe `z` pour le type entier signé correspondant à `size_t`
- `z` utilisable pour les littéraux binaires, octaux ou hexadécimaux de `size_t`



Chaînes littérales

- Plus de concaténation de chaînes littérales adjacentes d'encodage différent

```
L"" u""; // Invalide  
L"" u8""; // Invalide  
L"" U""; // Invalide  
u8"" L""; // Invalide  
u8"" u""; // Invalide  
u8"" U""; // Invalide  
u"" L""; // Invalide  
u"" u8""; // Invalide  
u"" U""; // Invalide  
U"" L""; // Invalide  
U"" u""; // Invalide  
U"" u8""; // Invalide
```

Et si ?

Si une des chaînes n'a pas d'encodage, on utilise celui de la seconde

Caractères littéraux

- Caractères Unicode conservés durant la phase du préprocesseur

```
#define S(x) # x  
// C++23 : "Köppe"  
const char * s1 = S(Köppe);  
const char * s2 = S(K\u00f6ppe);
```

- Suppression des caractères littéraux larges non codables ou multi-caractères
- Ajout de séquences d'échappement délimitées
 - `\u{}` prenant un nombre arbitraire de chiffres hexadécimaux
 - `\x{}` prenant un nombre arbitraire de chiffres hexadécimaux
 - `\o{}` prenant un nombre arbitraire de chiffres octaux
- Ajout de séquences d'échappement nommés `\N{...}`

```
cout << "\N{GREEK SMALL LETTER ETA WITH PSILI}";
```

Évolutions des opérateurs d'égalité

- Modification des règles de résolution de `operator==` et `operator!=`
- Corrige des ambiguïtés introduites par la réécriture de `==` et `!=` en C++20
- `operator==` est utilisé pour réécrire `operator!=` et la forme inverse de `operator==` uniquement si `operator!=` n'existe pas

```
struct Foo {  
    bool operator==(const Foo&) { return true; }  
    bool operator!=(const Foo&) { return false; }  
};
```

// Ambigu en C++20

```
bool b = Foo{} != Foo{};
```

operator[] multidimensionnel

- Définition de `operator[]` avec aucun ou plusieurs arguments
- Y compris des arguments *variadic*

```
T& operator[] ();  
T& operator[] (size_t x, size_t y, size_t z);  
  
foo[3, 2, 1] = 42
```

Au-delà de C++23

- Réécritures
 - De `a[x][y][z]` en `a[x, y, z]`
 - De `a(x, y, z)` en `a[x][y][z]` (et `a(x)` en `a[x]`)
 - De `a[x, y, z]` en `a[x][y][z]`
- Extension aux tableaux C, aux conteneurs standards existants et aux `operator[]` non-membres



Opérateurs static

- Possibilité de déclarer `static` des `operator()`

```
struct Foo {  
    static constexpr bool operator()(int i, int j) { return i < j; }  
};  
  
static_assert(Foo::operator()(1, 2));
```

- Possibilité de déclarer `static` des `operator[]`

```
struct Foo {  
    static int operator[](int i) { return v[i]; }  
    static constexpr array<int, 4> v{5, 8, 9, 12};  
};  
  
cout << Foo::operator[](2) << "\n";
```



Évolutions des lambdas

- () optionnelles en l'absence de paramètres dans les lambdas mutables
- Utilisation du *name lookup* du corps de la lambda pour son retour

```
// Ne compile pas en C++20 et précédents  
auto foo = [j=0]() mutable -> decltype(j) { return j++; };
```

- Ajout du support d'attributs pour les lambdas

```
[] [[ attr ]] () ->int { return 42; };
```


Évolutions des lambdas

- Support des attributs `[[nodiscard]]`, `[[deprecated]]`, `[[noreturn]]`
- Lambdas `static` : `operator()` de l'objet généré est `static`

Limites

- `static` et `mutable` sont mutuellement exclusifs
- Liste de capture vide

`std::invoke_r()`

- Similaire à `std::invoke()`
- Retour convertit vers le premier paramètre template
- Ou ignoré si le premier paramètre template est `void`

Évolutions des attributs

- Duplication possible d'un attribut dans une liste d'attributs

```
// Valide en C++23, pas en C++20  
[[nodiscard, nodiscard]]  
int foo();
```

Nouveaux attributs

- `[[assume(expression)]]` permet au compilateur d'optimiser en supposant la véracité de l'expression

Contrainte

- Expression doit être vraie à l'emplacement de `assume`

Layout

- Suppression de la possibilité donnée aux compilateurs de réordonner les données d'accessibilité différente

Paramètre `this` explicite / deducing `this`

- Objectif : limiter les surcharges `const` / non `const` de fonctions membres
- Utilisation d'un premier paramètre, préfixé `this`, notant l'instance de classe

```
struct Foo {  
    void bar(this Foo const&);  
}
```

Restrictions

- Ne peuvent pas être `virtual` ni `static`
- Ne peuvent pas avoir de *cv-qualifier* ni de *ref-qualifier*

Paramètre `this` explicite / deducing `this`

- Utilisation des règles classiques de déduction de types

```
struct Foo {  
    template <typename Self>  
    void bar(this Self&&, int);  
};  
  
void ex(Foo& foo, D& d) {  
    foo.bar(1);           // Self=Foo&  
    move(foo).bar(2);     // Self=Foo  
}
```

Paramètre `this` explicite / deducing `this`

- Permet le passage de `this` par valeur

```
struct Foo {  
    void bar()(this Foo, int i);  
};  
  
Foo{}(4);
```


Déduction dans les constructeurs hérités

- Déduction des paramètres templates d'un constructeur hérité

```
template <typename T> struct A {  
    A(T);  
};  
template <typename T> struct B : public A<T> {  
    using A<T>::A;  
};  
  
B b(42);           // OK B<int>
```

noexcept

- Ajout de `noexcept` à plusieurs fonctions de la bibliothèque standard

Traits

- `std::is_scoped_enum` indique si un type est un `enum class`

```
class A {};  
enum E {};  
enum struct Es {};  
enum class Ec : int {};  
  
is_scoped_enum_v<A>;      // Faux  
is_scoped_enum_v<E>;      // Faux  
is_scoped_enum_v<Es>;     // Vrai  
is_scoped_enum_v<Ec>;     // Vrai  
is_scoped_enum_v<int>;    // Faux
```



Traits

- `std::is_implicit_lifetime` indique si un objet à une durée de vie implicite
- `std::reference_constructs_from_temporary` et `std::reference_converts_from_temporary` indiquent si la référence est construite depuis un temporaire

Chaînes de caractères

- `contains()` teste la présence d'une sous-chaîne dans une chaîne ou une vue

```
string foo = "Hello world";  
foo.contains("Hello");    // true  
foo.contains("monde");    // false  
  
string_view bar = foo;  
bar.contains("Hello");    // true  
bar.contains("monde");    // false
```

- Interdiction de la construction de `std::string` depuis `nullptr`
- Construction de `std::string_view` depuis un range
- Ajout de la contrainte trivialement copiable à `std::string_view`



Chaînes de caractères

- `resize_and_overwrite()` redimensionne et met à jour une chaîne
 - Allocation d'un tableau de `count + 1` caractères
 - Copie du contenu de la chaîne dans ce tableau
 - Appel à la fonction pour valoriser les caractères et déterminer la taille finale
 - Mise à jour du contenu de la chaîne avec celui du tableau

```
string foo = "Hello ", bar = "world!";

foo.resize_and_overwrite(20,
    [sz = foo.size(), bar] (char* buf, size_t buf_size) {
    auto to_copy = min(buf_size - sz, bar.size());
    memcpy(buf + sz, bar.data(), to_copy);
    return sz + to_copy; }); // Hello world!
```

Motivation

Éviter des initialisations, des tests et des copies inutiles



std::span

- Ajout de la contrainte trivialement copiable

std::pair

- Construction depuis un *braced initializers*

```
pair<string, vector<string>> foo("hello", {});
```

- Construction depuis std::tuple, std::array ou un *tuple-like*

```
tuple foo{1, 3.0};  
pair bar{foo};
```

```
array foo{1, 3};  
pair bar{foo};
```


std::stack & std::queue

- Création de std::stack et std::queue depuis une paire d'itérateurs

```
vector<int> v{1, 3, 7, 13};  
queue q(begin(v), end(v));  
stack s(begin(v), end(v));
```



Conteneurs associatifs

- Surcharge de `erase()` et `extract()` ne créant pas de clés temporaires
- Adaptateurs de conteneurs associatifs
 - `std::flat_map` et `std::flat_multimap`
 - Interface similaire à `std::map` et `std::multimap`
 - Davantage *cache-friendly*
 - Clés et valeurs stockées dans deux conteneurs différents
 - `std::flat_set` et `std::flat_multiset`

std::mdspan

- Vues multidimensionnelles
- Possibilité de fournir un *layout* configurable
- Trois *memory layouts* standards
 - `layout_right` : *layout* du C et du C++, lignes puis colonnes
 - `layout_left` : *layout* de Fortran ou Matlab, colonnes puis lignes
 - `layout_stride`
- Accès à un élément via `operator[]` multi-paramètres (`[x,y,z]`)

Évolutions des itérateurs

- Correction de `iterator_category` et `counted_iterator`
- `std::move_iterator<T*>` doit être un *random access iterator*
- Modification des exigences sur les itérateurs des algorithmes « non ranges » pour permettre l'utilisation de vues

std::byteswap()

- Inverse les octets d'un entier

```
uint16_t i = 0xCAFE;  
byteswap(i);    // 0xFECA  
  
uint32_t j = 0xDEADBEEFu;  
byteswap(j);    // 0xEFBEADDE
```



Évolutions des flux

- `spanstream` remplaçant de `strstream` utilisant un `std::span` comme buffer
- Support du mode exclusif à `std::fstream`

Évolutions de `std::format`

- Ajout du concept `formatable`
- Vérification des chaînes de format au *compile-time*
- Réduction de la taille du binaire de `format_to`
- Support des types `std::generator-like` par `std::format`
- Formateur de `std::chrono` *locale-independent* par défaut

```
format("{:%S}", sec(4.2)); // C++20 : 04,200 / C++23 : 04.200  
format("{:L%S}", sec(4.2)); // C++20 : exception / C++23 : 04,200
```

Évolutions de `std::format`

- Ajout du type `?` pour afficher les chaînes échappées
- Formatage des ranges
- Formatage des `std::pair` et `std::tuple`
- Formateur pour `std::vector<bool>::reference`
- Formatage des conteneurs (si les éléments sont formatables)
 - `std::map` et équivalent : `{k1: v1, k2: v2}`
 - `std::set` et équivalent : `{v1, v2}`
 - `std::vector`, `std::list`, ... : `[v1, v2]`
- Formatage des `std::thread::id`
- Formatage des `std::stacktrace`

std::print

- Écriture dans `std::cout` depuis une chaîne de format `std::format`

```
cout << format("Hello, {}!", name);
```

```
// Devient
```

```
print("Hello, {}!", name);
```

out_ptr et inout_ptr

- Abstractions entre *smart pointers* et API C modifiant un pointeur
 - Création d'un pointeur de pointeur temporaire depuis le *smart pointer*
 - Automatisation des appels à `reset()` et `release()`
 - *Exception-safe* : *smart pointer* rétabli au retour de l'API C
 - Permet le passage comme pointeur C `void*` ou `void**`
 - Permet la conversion vers un type de pointeur arbitraire
- `out_ptr` permet à l'API C de modifier l'adresse contenu dans le *smart pointer* sans l'utiliser
- `inout_ptr` permet à l'API C de modifier l'adresse contenu dans le *smart pointer* mais aussi de l'utiliser

Bibliothèque de Stacktrace

- Basée sur Boost.stacktrace
- `current()` récupère la *stacktrace* courante
- Manipulation d'une *stacktrace*
 - `empty()` teste la présence d'entrée
 - `size()` retourne le nombre d'entrée de la *stacktrace*
 - `begin()`, `end()`, ... retournent les itérateurs sur les entrées
 - `operator[]` accède à une entrée donnée
 - `to_string()` retourne la description de la *stacktrace*
 - `operator<<` affiche la *stacktrace*
- Manipulation des entrées de la *stacktrace*
 - `description()` retourne la description de l'entrée
 - `source_file()` retourne le nom de la fonction
 - `source_line()` retourne la ligne

Bibliothèque de Stacktrace

```
auto trace = stacktrace::current();  
for(const auto& entry: trace) {  
    cout << "Description: " << entry.description() << "\n";  
    cout << "file: " << entry.source_file() << "\n";  
    cout << "line: " << entry.source_line() << "\n";  
    cout << "-----" << "\n";  
}
```



`std::unreachable()`

- `std::unreachable()` indique que la localisation n'est pas atteignable
- Permet au compilateur d'optimiser en supposant que le code ne sera pas atteint
- Comportement indéfini si `std::unreachable()` est appelé



Atomiques

- Support des `atomics` C

time_point::clock

- Relâchement des contraintes sur `time_point::clock`
 - Plus grande flexibilité du type d'horloge
 - Horloges *stateful*, horloges externes
 - Représentation d'un *time of day* par un `time_point` particulier

std::variant

- Héritage possible de `std::variant`
- `std::visit()` restreints aux `std::variant`

Opérations monadiques de `std::optional`

- `transform()` modifie la valeur contenu dans un `std::optional`
 - Retourne un `std::optional` vide s'il n'y a pas de valeur stockée
 - Retourne le résultat de la fonction sinon

```
optional<string> foo = "Abcdef", bar;
```

```
foo.transform([](auto&& s) { return s.size(); }); // 6
```

```
bar.transform([](auto&& s) { return s.size(); }); // Vide
```

Opérations monadiques de `std::optional`

- `and_then()` dérive une fonction pour retourner un `std::optional`

```
auto func = [] (int i) -> optional<int> { return 2 * i; };  
optional<int> foo = 42, bar;  
  
foo.and_then(func); // 84  
bar.and_then(func); // Vide
```

Retour de fonction

Le retour de la fonction doit être une spécialisation de `std::optional`

Opérations monadiques de `std::optional`

- `or_else()`
 - Retourne le `std::optional` s'il a une valeur
 - Appelle une fonction sinon

```
auto func = [] -> optional<string> { return "Oups!"; };
```

```
optional<string> foo = "Abcdef", bar;
```

```
foo.or_else(func); // Abcdef
```

```
bar.or_else(func); // Oups!
```

Retour de fonction

Le retour de la fonction doit être une spécialisation de `std::optional`

std::expected

- Classe template `std::expected<T, E>` contenant
 - Soit une valeur de type `T`
 - Soit une erreur de type `E`
- `operator bool()` et `has_value()` indiquent si l'objet contient une valeur
- `operator->` et `operator*` accèdent à la valeur
- `value()` retourne la valeur
- `error()` retourne l'erreur

```
expected<int, string> foo(int i) { ... }
```

```
expected<int, string> e = foo(5);
```

```
if(e)
```

```
    cout << e.value();
```

```
else
```

```
    cout << e.error();
```



std::expected

- `value_or()` retourne
 - La valeur si présente
 - La valeur reçue en paramètre sinon
- `transform()` modifie la valeur contenu dans un `std::expected`
- `and_then()` dérive une fonction pour retourner un `std::expected`
- `or_else()`
 - Retourne la valeur si elle est présente
 - Appelle une fonction avec l'erreur sinon

Retour de fonction

Le retour de `and_then()` et `or_else()` doit être `std::expected`



std::expected

- `error_or()` retourne
 - L'erreur si la valeur n'est pas présente
 - Le paramètre sinon
- `transform_error()`
 - Retourne la valeur si elle est présente
 - Appelle une fonction avec l'erreur sinon



std::unexpected

- Classe template `std::unexpected<E>` contenant une erreur
- `error()` retourne l'erreur
- Permet de construire un `std::expected` indiquant une erreur

```
expected<double, int> foo = unexpected(3);
```

```
// Vrai
```

```
if (!foo) { ... }
```

```
// Vrai
```

```
if (foo == unexpected(3)) { ... }
```

Évolutions des ranges et vues

- Ajout de `starts_with()` et `ends_with()` aux ranges
- Ajout de `contains()` aux ranges

```
auto foo = view::iota(0, 50);  
auto bar = view::iota(0, 30);  
  
if(ranges::starts_with(foo, bar)) { ... }
```

- Relâchement des contraintes sur les *range adaptors* pour accepter les types *move-only*
- Relâchement des contraintes sur `join_view` permettant le support de davantage de ranges
- Suppression de la contrainte *default constructible* pour les vues
- `std::ranges::to<>()` construit un conteneur depuis un range

Nouveaux ranges et range adaptors

- Amélioration de `std::views::split()`
- `std::views::lazy_split()`
- `std::views::zip()` et `std::views::zip_transform()`

```
auto x = vector{1, 2};  
auto y = list<string>{"Aa", "Bb", "Cc"};  
auto z = array{'A', 'B', 'C', 'D'};  
  
// 1 Aa A  
// 2 Bb B  
for(tuple<int&, string&, char&> e : views::zip(x, y, z)) {  
    cout << get<0>(e) << ' ' << get<1>(e) << ' ' << get<2>(e) << '\n';  
}
```

- `std::views::adjacent()` et `std::views::adjacent_transform()`

Nouveaux ranges et range adaptors

- `std::ranges::iota()`
- `std::ranges::shift_left()` et `std::ranges::shift_right()`
- `std::views::join_with()` transforme un range de ranges en un range

```
vector<string> vs = {"the", "quick", "brown", "fox"};
```

```
vs | join_with('-'); // the-quick-brown-fox
```

- `std::views::chunk()` coupe un range en blocs de N éléments
- `std::views::slide()` : `std::views::adjacent()` avec une taille *run-time*
- `std::views::chunk_by()` découpe un range en fonction d'un prédicat

```
vector v = {1, 2, 2, 3, 0, 4, 5, 2};
```

```
v | chunk_by(less_equal{})); // [[1, 2, 2, 3], [0, 4, 5], [2]]
```

Nouveaux ranges et range adaptors

- `std::views::find_last()`
- `std::ranges::stride_view()` conserve un élément sur n

```
iota(1, 13) | stride(3); // 1 4 7 10
```

- `std::ranges::fold()` équivalent range de `std::accumulate`

```
vector<double> v = {0.25, 0.75};  
  
auto r = ranges::fold(v, 1, plus()); // 2
```

- `std::views::cartesian_product` construit une vue sur le produit cartésien de plusieurs conteneurs

Nouveaux ranges et range adaptors

- `std::views::as_rvalue()`
- `std::views::repeat()` répète n fois une valeur

```
views::repeat(17, 4); // 17 17 17 17
```

- Correction de `std::ranges::istream_view()`
- `std::views::enumerate()` range index/valeur depuis un range de valeurs
 - Manipulation d'un index dans un *range-based for loop* sans gestion explicite
 - Construction de `std::map` depuis un `std::vector` avec l'index pour clé

borrowed_range

- Nouveau concept de range : `borrowed_range`
- Range dont les itérateurs sur celui-ci reste valide après sa destruction
- Des ranges inconditionnellement *borrowed* : `ref_view`, `string_view`, `empty_view` et `iota_view`
- Des ranges conditionnellement *borrowed*, selon la vue sous-jacente : `take_view`, `drop_view`, ...

Range adaptors définis par l'utilisateur

- Classe de base `std::ranges::range_adaptor_closure<t>`
- Adaptateur de fonction `std::bind_back()`

```
bind_back(f, ys...)(xs...);
```

```
// Equivalent a
```

```
f(xs..., ys...);
```

Modules

- Module `std` pour tout le namespace `std` depuis les en-têtes C++ et C
- Module `std.compat` : module `std` et le namespace global des en-têtes C

std::generator

- Générateur de coroutines synchrones

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- Début formel des travaux en juin 2023

Vérification statique

- Support de messages construits par `static_assert`

```
static_assert(sizeof(S) == 1,  
format("Attendu 1, obtenu {}", sizeof(S)));
```

Compile-time

- Uniquement des valeurs connues au *compile-time*

Lexer

- Suppression de comportements indéfinis
 - *Universal characters* sur plusieurs lignes autorisés

```
int \\  
u\  
0\  
3\  
9\  
1 = 0;
```

- Construction possible d'*universal characters* par des macros

```
#define CONCAT(x, y) x ## y  
int CONCAT(\\, u0393) = 0;
```

- Une chaîne non terminée est une erreur

Encodage

- Ajout de @, \$ et ` au jeu de caractères de base
- Caractères non-encodables sont mal formés
- Notion de *literal encoding* et *environnement encoding* et API d'interrogation

Placeholders

- Joker `_` pour des variables inutilisées

```
auto _ = foo(); // Equivalent a [[maybe_unused]] auto _ = foo();
```

```
std::lock_guard _(mutex);  
auto [x, y, _] = f();
```

Conteneurs

- Nouveaux conteneurs
 - Vecteur de capacité fixée en *compile-time* `std::inplace_vector`
- Possibilité d'utiliser `std::weak_ptr` en tant que clé de conteneur associatif
- `std::submdspan()` retourne une vue sur un sous-ensemble d'un `std::mdspan`

Chaînes de caractères

- Support de `std::string_view` par `std::stringstream`
- Interfaçage de `std::bitset` avec `std::string_view`

Initializer-list

- *static storage* possible pour les *braced-initializer-list*

std::visit()

- Versions membres de `std::visit()` et `std::visit_format_arg()`

Ratio

- Ajout des préfixes quecto, ronto, ronna et quetta

constexpr

- Davantage de `constexpr` dans la bibliothèque standard
- Conversion depuis `void*` dans des contextes `constexpr`
 - `std::format()` au compile-time
 - `std::function_ref`, `std::function` et `std::any` `constexpr`

lifetime

- `std::is_within_lifetime()` indique si l'objet pointé est vivant
- ... en particulier si un membre d'une union est active

Gestion mémoire

- *hazard pointers* : unique écrivain, multiples lecteurs
- Structure de donnée *read-copy update* : planification d'actions (p.ex. suppression) à réaliser plus tard

Type callable

- Ajout de `std::copyable_function` pour les fonctions copiables
- Ajout de `std::function_ref`
 - Type référence pour le passage d'appelable à une fonction
 - Plus générique et moins gourmand que `std::function` et équivalents

std::format

- Redéfinition de `std::to_string` en terme de `std::format`
- Davantage de vérifications *compile-time* du type des arguments
 - Déjà le cas de la majorité des erreurs
 - ... mais pas de toutes

```
format("{:>{}}", "hello", "10");  
// Erreur run-time
```

- Formatage des pointeurs

```
format("{:#018X}", reinterpret_cast<uintptr_t>(&i));  
// 0X00007FFE0325C4E4
```


Durées et temps

- Spécialisation de `std::hash` pour `std::chrono`

Accès bas-niveaux aux IO

- Alias sur le descripteur de fichier de la plateforme : `native_handle_type`
- `native_handle()` retourne ce descripteur

Sommaire

- 1 Retour sur C++98/C++03
- 2 C++11
- 3 C++14
- 4 C++17
- 5 C++20
- 6 C++23 « *Pandemic Edition* »
- 7 C++26
- 8 Et ensuite ?

Présentation

- C++23 ne marque pas la fin des évolutions du C++
- Plusieurs sujets proposés et non pris en compte dans les versions actuelles
- Plusieurs TS publiés et non intégrés ou en cours d'étude

TS – Contracts

- Retiré du *draft* C++20 et création d'un groupe d'étude en juillet 2019
- Support de la programmation par contrat
- Remplace la vérification via `assert`
- Et la documentation via commentaires `@pre`, `@post` et `@invariant`
- Plusieurs propositions initiales concurrentes
- ... mais un compromis à émerger
- Les contrats de fonctions membres publiques peuvent utiliser des membres privés ou protégés
- Intégration des contrats à la bibliothèque standard

TS – Contracts

- Diverses propositions de déclaration
 - Attributs : `[[pre:...]]`, `[[post:...]]` et `[[assert:...]]`
 - Conditions : `precond(...)`, `postcond(...)` et `incond(...)`
 - Fermetures : `pre {...}`, `post(...) {...}`
 - Alternatives : `[{pre:...}]`, `@(pre: ...)`
- Plusieurs comportements
 - `ignore` : contrat non vérifié
 - `enforce` : appel au *handler* de violation de contrat et terminaison
 - `observe` : appel au *handler* de violation de contrat et poursuite

TS – Networking TS

- Publié en avril 2018
- Partiellement basé sur Boost.Asio
- Gestion de *timer*
- Gestion de tampon et de flux orientés tampon
- Gestion de *sockets* et de flux *socket*
- Gestion IPv4, IPv6, TCP, UDP
- Manipulation d'adresses IP
- Pas de protocoles de plus haut niveau actuellement
- Demande post-TS : gestion de la sécurité (a priori pas possible)
- Modèle asynchrone, différent de celui déjà présent en C++

TS – Pattern matching

- Utilisation du mot clé `inspect` (ou `switch`) et du *wildcard* `--` (ou `_`)
- Utilisable sur
 - Entiers

```
inspect(x) {  
  0  => { cout << "Aucun"; }  
  1  => { cout << "Un"; }  
  -- => { cout << "Plusieurs"; }  
}
```

- Chaînes de caractères

```
inspect(x) {  
  "zero" => { cout << "Aucun"; }  
  "un"   => { cout << "Un"; }  
  --     => { cout << "Plusieurs"; }  
}
```


TS – Pattern matching

- `std::tuple`, `std::pair`, `std::array` et tuple-like

```
inspect(p) {  
    [0, 0] => { cout << "on origin"; }  
    [0, y] => { cout << "on y-axis"; }  
    [x, 0] => { cout << "on x-axis"; }  
    [x, y] => { cout << x << ',' << y; }  
}
```

- `std::variant` et `std::any`

```
inspect(v) {  
    <int> i    => { cout << "Entier " << i; }  
    <float> f => { cout << "Reel " << f; }  
}
```

TS – Pattern matching

- Types polymorphiques

```
inspect(shape) {  
  <Circle> [r]      => { cout << 3.14 * r * r; }  
  <Rectangle> [w, h] => { cout << w * h; }  
}
```

- Support des gardes

```
inspect(p) {  
  [x, y] if(x > y) => { cout << x << "superieur a" << y; }  
}
```

Attention

Prise en compte de la première correspondance et non de la meilleure

TS – Library fundamentals 2

- Partiellement intégré en C++17 et C++20
- `std::is_detected` indique si un *template-id* est bien formé
- Wrapper `std::propagate_const` pour les pointeurs et *pointer-like*
- Pointeurs intelligents non possédants `std::observer_ptr`
- `std::ostream_joiner` écrit des éléments dans un flux de sortie

```
int foo[] = {1, 2, 3, 4, 5};  
copy(begin(i), end(i), make_ostream_joiner(cout, ", "));  
// "1, 2, 3, 4, 5"
```

- Générateur aléatoire propre au thread `std::default_random_engine` initialisé dans un état non prédictif
 - `std::randint()` génère un nombre entier dans une plage spécifiée
 - `std::reseed()` modifie la graine de génération
 - `std::sample()` choisit aléatoirement n élément d'une séquence
 - `std::shuffle()` réordonne aléatoirement les éléments d'un range

TS – Library fundamentals 3

- *Scope Guard* : enregistrement d'un foncteur appelé
 - appelé à la sortie du scope : `std::scope_exit`
 - appelé à la sortie du scope par une exception : `std::scope_fail`
 - appelé à la sortie du scope hors exception : `std::scope_success`
- *RAII wrapper* `std::unique_resource`

TS – Parallelism 2

- Exception levée durant une exécution parallèle
- Politique d'exécution `vector_policy`
- Support de SIMD (*Single Instruction on Multiple Data*)

TS – Concurrency

- Partiellement intégré à C++20, C++23 et C++26
- Versions de `std::future` et `std::shared_future` supportant les continuations
 - `is_ready()` indique si l'état partagé est disponible
 - `then()` attache une continuation à la future
- `std::when_any` crée une future disponible lorsqu'une des futures contenues devient disponible
- `std::when_all` crée une future disponible lorsque toutes les futures contenues sont disponibles
- `std::make_ready_future()` crée une future contenant une valeur immédiatement disponible
- `std::make_exceptional_future()` crée une future contenant une exception immédiatement disponible

TS – Transactional Memory

- Blocs synchronisés
- Blocs atomiques
- Fonction *transaction-safe*
- Attributs `[[optimize_for_synchronized]]`

TS – Reflection

- Réflexion statique uniquement
- Introspection
- Méta-programmation et code *compile-time*
- Injection
- Méta-classes
 - Construction de types de classes (dont les classes elles-mêmes) ayant
 - Des contraintes
 - Des comportements par défaut
 - Des opérations par défaut
 - `class`, `struct`, `enum class`, `interface`, `value type`

Dépréciation

- Dépréciation des modes d'arrondi (`fesetround()`)

Suppression

- Suppression d'éléments précédemment dépréciés
 - Conversion arithmétique d'énumération
 - Comparaison de tableau C (se fait sur les adresses)
 - `volatile`
 - `std::strstream`
 - `std::allocator`
 - API d'accès atomique à `std::shared_ptr`
 - `std::basic_string::reserve()` sans argument
 - *Unicode Conversion Facets*
 - `wstring_convert`
 - *Locale Category Facets for Unicode*

Erroneous behavior

- Ajout d'un nouveau type de comportement : *erroneous behavior*
- Indique un code incorrect, mais bien défini (dont *implementation-defined* et *unspecified behavior*)
- Recommandation au compilateur de lever un warning
- Compilateur peut rejeter le code
- Applicable aux lectures de variables non initialisées

Vérification statique

- *Procedural function interfaces*
 - Annotations de types *claim* / *assertion*
 - Recouvre des points du contract TS mais plus ambitieux

Mots-clés

- Conversion de macros en mots-clés
 - `assert`
 - `offsetof`

Encodage

- Ajout des algorithmes Unicode
- Support d'Unicode (UTF-8, UTF-16 et UTF-32) dans la bibliothèque standard

Types

- Relâchement des restrictions sur les `typedef _t`
- Mécanismes *compile-time* vérifiant que deux types ont la même représentation mémoire
- Ajout d'un type « *fixed point decimal* »
- Entiers larges `wide_integer<128, unsigned>`
- Nombres rationnels
- Possibilité de définir des objets `constexpr`
- *Zero-initialisation* des objets *automatic storage duration*
- Entiers non signés pour lesquels l'*overflow* est un UB
- Rendre les `std::initializer_list` déplaçables
- Détection et gestion des débordements
- Gestion des arrondis

Support des unités physiques

- Gestion des quantités et dimensions
- Supports des unités de base, dérivées, multiples et sous-multiples
- Conversions et opérations entre unités

```
static_assert(10km / 2 == 5km);

static_assert(1h == 3600s);
static_assert(1km + 1m == 1001m);

static_assert(1km / 1s == 1000mps);
static_assert(2kmph * 2h == 4km);
static_assert(2km / 2kmph == 1h);

static_assert(1000 / 1s == 1kHz);

static_assert(10km / 5km == 2);
```


Représentation mémoire

- Accès aux octets sous-jacents d'un objet
 - Nouvelle catégorie d'objet *contiguous-layout*
 - Uniquement des types scalaires et des classes sans fonction ni base virtuelle
 - N'hérite pas d'objet non *contiguous-layout*
 - Contiguïté garantie
 - Représentation sous forme de tableau
 - Obtention d'un pointeur sur la représentation via `reinterpret_cast` vers `char`, `unsigned char` ou `std::byte`
 - Conversion pointeur sur représentation vers pointeur sur objet via `reinterpret_cast`

Relocate

- Opération *relocate* (déplacement puis destruction)
 - Définition de la notion *relocatable*
 - Concept Relocatable
 - Définition de la notion de *trivial relocatability*
 - Traits `std::is_relocatable`, `std::is_nothrow_relocatable` et `std::is_trivially_relocatable`
 - Attribut `[[trivially_relocatable]]`
 - Algorithmes gérant cette opération : `std::relocate_at()`, `std::uninitialized_relocate()` et `std::uninitialized_relocate_n()`

Shadowing

- Levée plusieurs restrictions
 - Masquage avec un type **void** pour empêcher l'utilisation de la variable masquée
 - Initialisation de la nouvelle variable avec l'ancienne variable de même nom
 - Masquage sans création d'une nouvelle portée
 - Conversion conditionnelle

```
auto foo = optional<string>{"Foo"};  
if(foo as string) { /* foo: string& */ }  
else { /* foo: optional<string> */ }
```

- Constification d'un conteneur dans un *range-based for loop*

```
vector<string> foo{"1", "2", "3"};  
cfor(auto &bar : foo) { /* foo est constant */ }
```

std::ignore

- `std::ignore` pour ignorer une valeur de retour

```
ignore = printf("Hello\n");
```

Contrôle de flux

- Ajout d'une instruction à **break** appelé lors de la sortie de la boucle
- Ajout d'une boucle `do_until`
- Version *generator-base* de *for loop*

```
struct generator { ... }  
  
for(int i: generator())  
{ ... }
```

do expression

- Ajout des « *do expression* » : instructions traités comme une expression

```
int x = do { do return 42; };
```

- Améliorations et simplifications des coroutines, du *pattern matching*, ...
- Introduit un nouveau scope mais pas de nouveau *function scope*
- `do return` pour retourner une valeur dans un *do expression*
- Possibilité d'explicitement le type de retour

static_assert

- Retarder à l'instanciation l'échec de `static_assert(false)` dans des templates

```
// C++20 : echec de compilation systematique  
template<typename T> int my_func(const T&) {  
    if constexpr(is_integral_v<T>) { return 1; }  
    else if constexpr(is_convertible_v<string, T>) { return 2 ; }  
    else { static_assert(false); }  
}
```

Évolutions des fonctions

- *Unified Call Syntax*

- $x.f(\dots)$ tente d'appeler $f(x, \dots)$ si $x.f(\dots)$ n'est pas valide
- $p \rightarrow f(\dots)$ tente d'appeler $f(p, \dots)$ si $p \rightarrow f(\dots)$ n'est pas valide
- Si $f(x, \dots)$ n'est pas valide, $f(x, \dots)$ tente d'appeler
 - $x \rightarrow f(\dots)$ si `operator->` existe pour x
 - $x.f(\dots)$ sinon
- Généralisation de `std::begin()` et co. dans le langage

- Possibilité pour les fonctions `va_start` de ne prendre aucun argument
- Élision de copie des objets de retour nommés (NRVO) garantie
- Paramètres `constexpr` et « *maybe constexpr* »
- Fonctions *heap-free*
- Retour `std::move(x)` éligible au NRVO si x l'est

Évolutions des fonctions

- Possibilité de déterminer l'appelant
- Arguments nommés

```
void foo(int a, int b, int c, int d, bool e = false);
```

```
foo(b: 10, a: 100, c: 640, d: 480);
```

```
foo(100, 10, d: 480, e: false, c: 640);
```

Opérateurs

- Surcharge de `operator`.
 - Si l'opérateur est défini, les opérations sont transférés à son résultat
 - ... sauf celles définies comme fonctions membres
 - Réalisation de *smart reference* (p.ex. *proxy*)
- Surcharge de `operator?`:
- `operator??` pour tester `std::expected`
- Évolutions des opérateurs de comparaison et de `operator<=>`
 - Dépréciation des conversions entre énumération et flottant
 - Dépréciation des conversions entre énumérations
 - Dépréciation de la comparaison « *two-way* » entre types tableaux
 - Comparaison *three-way* entre *unscoped* énumération et type entier
- Interdiction de l'appel de `operator=` sur des temporaires
- Possibilité d'utiliser `auto` ou `auto&` comme retour d'opérateur `=default`

Opérateurs

- Génération d'opérateurs à la demande via `=default`
 - `operatorX=` à partir de `operatorX`
 - incrément et décrétement préfixés à partir de l'addition et de la soustraction
 - incrément et décrétement postfixés à partir des versions préfixés
 - `operator-` et `operator->` à partir de `operator*` et `operator.`
- Ajout de `operator[]` à `std::initializer_list`
- Opérateur pipeline `operator|>`

```
x|>f(y);
```

```
// Equivalent a
```

```
f(x, y);
```

Opérateurs

- `operator template()` : extension du support des *non-type template parameters*
- Opérateur d'implication `operator==>()`

```
p ==> q;
```

```
// Equivalent a
```

```
!p || q;
```

- Opérateur `nameof`

Structured binding

- Support du *structured binding* sur `std::extents`

Classes

- Qualifieurs autorisés sur les constructeurs
 - Constructeurs `const` pour construire systématiquement des objets constants
 - Constructeurs non `const` peuvent construire des objets constants ou non
- Déduction template dans les constructeurs d'agrégats et les alias
- *Layout* des classes
 - Contrôle du *layout* pour privilégier taille, ordre de déclaration, visibilité, vitesse, ordre alphabétique, lignes de cache ou règles d'une version antérieure du C++ ou d'un autre langage
 - Contrôle de l'alignement (remplaçant de `#pragma pack(N)`)
- Constructeurs par déplacement =bitcopies
- Extension de `=delete` à d'autres construction (variables template)
- `=delete` avec un message pour le diagnostic de compilation
- Classes de base `std::noncopyable` et `std::nonmovable`
- Mécanisme de conversion tableau de structures vers structure de tableaux

Classes

- Possibilité de déclarer **friend** un *parameter pack*

```
template <typename T>
class Foo {
    friend T;          // OK en C++23 et avant
    ...
};

template <typename... Ts>
class Bar {
    friend Ts...;      // Invalide en C++23
    ...
};
```

Énumération

- Ajout d'énumérations « *flag-only* »
- Fonctions membres sur les énumérations

Gestion d'erreur

- Exceptions légères (*Zero-overhead deterministic exceptions*)
- Objet standard pour le retour d'erreur (`status_code` et `error`)
- Récupération des informations de l'exception contenue dans un `std::exception_ptr`

Conteneurs

- Nouveaux conteneurs

- Tableaux multidimensionnels `std::mdarray`
- Queue concurrente
- *Bucket array* `std::hive` : plusieurs blocs d'éléments liés entre eux avec un indicateur sur l'état de chaque élément (actif / effacé)
- Vecteur utilisant un buffer externe
- Conteneurs intrusifs : conteneurs non possédants
- Conteneurs `inplace` avec un buffer de taille fixe

Conteneurs

- `span` de taille fixe
- Ajout de `at()` à `std::span`
- Relâchement des contraintes sur les tableaux C
 - Initialisation des tableaux d'agrégats
 - Copies de tableaux
 - Tableau comme type de retour
- Correction de dysfonctionnements de `std::flat_map` et `std::flat_set`

Chaînes de caractères

- Construction de `std::string_view` depuis des chaînes implicites
- Prise en charge de `std::string_view` par `std::from_chars`
- Concaténation de `std::string` et `std::string_view`
- Modification du constructeur de `std::string` depuis un caractère pour interdire les autres numériques (entiers ou flottants)
- Voire dépréciation de la construction d'un `std::string` depuis un caractère

Tuples

- Récupération d'un index depuis un type pour `std::variant` et `std::tuple`
- Utilisation de tableaux C comme *tuple-like*
- Utilisation d'*aggregates* comme *tuple-like*
- Amélioration de l'ergonomie d'accès aux champs des `std::tuple`

```
t[0ic]
```

```
// Equivalent a
```

```
std::get<0>(t)
```

- `std::complex` deviennent des *tuple-like*

Itérateurs

- API « itérateurs » de génération des nombres aléatoire
- `std::iterator_interface` pour la définition de nouveaux itérateurs

Algorithmes

- `std::find_last()` recherche depuis la fin d'un conteneur
- `std::is_uniqued` test l'absence de deux valeurs consécutives identiques
- Gestion des UUID
- Fonctions statistiques (moyenne, médiane, variance, ...)
- Améliorations du générateur aléatoire
- Algèbre linéaire
 - Basé sur un sous-ensemble de BLAS (vecteur, matrice, ...)

Ranges

- Ajout d'un paramètre « pas » à `std::iota_view`
- Utilisation de `std::get_element<>` comme point de configuration

```
// Tri sur le premier element du tuple  
vector<tuple<int, int>> v{{3,1},{2,4},{1,7}};  
  
ranges::sort(v, less{}, get_element<0>);
```

- Plusieurs nouveaux adaptateurs : `adjacent_filter`, `adjacent_remove_if`, `c_str`, `generate`, ...
- Ajout de `ranges::size_hint` permettant aux ranges de réserver de la mémoire

Traits

- Trait `std::is_narrowing_convertible`
- Traits et fonctions pour garantir des conversions sans perte
- Trait indiquant si un type *trivially default constructible* peut être initialisé en mettant tous les octets à 0
- Amélioration de l'ergonomie de `std::integral_constant<int>`
- Trait `std::is_virtual_base_of` indiquant si une classe est une classe de base virtuelle d'une autre

Lambdas

- Capture mutable partielle par les lambdas

std::function

- `std::inplace_function` : pendant de `std::function` sans allocation
- `std::function_ptr_t` : pointeur générique sur une fonction

Attributs

- Réserve des attributs sans namespace et avec le namespace `std`
- Possibilité d'implémenter des attributs utilisateurs
- Nouveaux attributs
 - `[[invalidate_dereferencing]]` : `*ptr` et `ptr->` inutilisables après l'appel
 - `[[invalidate]]` : `ptr`, `*ptr` et `ptr->` inutilisables
 - `[[no_copy]]` : types et fonctions ne permettant pas la copie (mais le déplacement et le RVO)
 - `[[rvo]]` : fonctions utilisables uniquement dans un contexte RVO
 - `[[side_effect_free]]` ou `[[pure]]`
 - `[[trivially_relocatable]]`
 - `[[discard]]` indique qu'un retour de fonction est volontairement ignoré

Expansion statement

- Répétition d'une expression au *compile-time*

```
auto foo = make_tuple(0, 'a', 3.14);  
  
for... (auto elem : tup)  
    cout << elem << "\n";
```

- Duplication de l'expression pour chaque élément (pas de boucle)
- Éléments de type différent
- Utilisable sur `std::tuple`, `std::array`, classes destructurables, ...

Parameters pack

- Généralisation et simplification des *parameters pack*
 - Déclaration possible partout où une variable peut être déclarée

```
template <typename... Ts>  
struct Foo { Ts... elems; };
```

- Indexation des *packs*

```
struct tuple_element<I, tuple<Ts...>> { using type = Ts...[I]; };
```

- *Slicing* de *packs*

```
auto x = Foo(a1, [:]t1..., [3:]t2..., a2);  
bar([1:]t1..., a3, [0]t1);
```

Parameters pack

- *Pack* de taille fixe

```
template<unsigned int N> struct my_vector {  
    my_vector(int...[N] v) : values{v...} {}  
};
```

- *Variadic function* homogène

```
template <class T>  
void f(T... vs);
```

- *Unpack* de `std::tuple` à la volée

```
int sum(int x, int y, int z) { return x + y + z; }  
  
tuple<int, int, int> point{1, 2, 3};  
int s = sum(point.elements...);
```

Structured bindings

- Utilisation de *parameters pack* dans les *structures bindings*

```
tuple<X, Y, Z> f();  
  
auto [...xs] = f();  
auto [x, ...rest] = f();  
auto [x,y,z, ...rest] = f();  
auto [x, ...rest, z] = f();  
auto [...a, ...b] = f(); // ill-formed
```


std::format

- Amélioration du support de `std::filesystem::path`
 - Présence de caractères d'échappement (p.ex. `\n`)
 - Support de caractère UTF-8
- Amélioration du support de `std::chrono::time_point`
- Ajout de formateurs
 - Valeurs atomiques
 - Générateurs aléatoires et distributions
 - *Smart pointers*
 - `std::optional`, `std::variant`, `std::any` et `std::expected`
 - `std::mdspan`, `std::flat_map` et `std::flat_set`
- Possibilité de fournir une chaîne de format au *runtime*

std::scan

- Pendant du formatage de texte introduit en C++20
- Alternative sûre et robuste à `sscanf()`
- Extensible aux types utilisateurs
- Compatible avec les itérateurs et les ranges

```
string key;  
int value;  
scan("answer = 42", "{} = {}", key, value);  
//      ~~~~~      ~~~~~      ~~~~~  
//      entree      format      arguments  
// key : "answer", value : 42
```

```
string key;  
chrono::seconds time;  
scan("start = 10:30", "{0} = {1:%H:%M}", key, time);
```

Templates

- Instanciation possible de templates au *runtime* (JIT limité aux templates)
- Paramètre template universel
- Templates dans les classes locales
- Rendre les `<>` vides optionnels

Concepts

- Concept pour les algorithmes numériques

Type erasure

- Programmation polymorphique via *type erasure* : *Proxy*, *Facade*, *Addresser*

Références

- Ajout de références possédantes, $T\sim$, gérant la destruction de l'objet référencé
- *Reallocation constructor* transférant la responsabilité de l'objet initial à l'objet créé : $T::T(T\sim)$

Pointeurs

- Suppression de `NULL` et interdiction de `0` comme pointeur nul
- Surcharge de `new` retournant la taille réellement allouée

Pointeurs intelligents

- `std::retain_ptr` pointeur intrusif manipulant le comptage de référence interne
- Création de pointeurs intelligents avec une valeur par défaut
- Comparaison entre pointeurs intelligents et pointeurs nus
- Retour covariant avec `std::unique_ptr<T>` (comme `T*`)

Contrôle mémoire

- Ajout de mécanismes de sécurité de l'usage mémoire
 - *Aliasing*
 - Suivi des dépendances
 - Annotation de types
 - Gestion de *lifetime*
 - ...
- Accès à la taille réellement allouée
- Spécificateur de stockage des temporaires
 - `constinit`
 - `variable_scope`
 - `block_scope` : durée de vie des littéraux C
 - `statement_scope` : durée de vie des temporaires en C++
- Seuils d'allocation SOO (*Small Object Optimization*)

Concurrence

- Version `atomic` de minimum et maximum
- Obtention de l'adresse de l'objet référencé par `std::atomic_ref` via `data()`
- Invocation concurrente
- `std::volatile_load<T>` et `std::volatile_store<T>`
- Gestion des processus, de la communication avec ceux-ci et des *pipes*
- `std::fiber_context` : changement de contexte *stackfull* sans besoin de *scheduler*
- Ajout d'un nom aux threads et mutex
- Contrôle de la priorité et de la taille de pile des threads

Coroutines

- Bibliothèques de support des coroutines
- `std::lazy<T>` : coroutines permettant l'évaluation différée
- Unification et amélioration des API asynchrones

Regex

- Ajout de regex *compile-time*

Interface utilisateur

- Support des entrées/sorties audio
- `std::web_view` API fournissant une fenêtre dans laquelle le programme peut injecter des composants web (ou être appelé via *callback*)

Debug

- `std::breakpoint()` : point d'arrêt dans le programme
- `std::breakpoint_if_debugging` : point d'arrêt si l'exécution se fait dans un debugger
- `std::is_debugger_present()` permet de savoir si l'exécution se fait dans un debugger

Module

- Communication d'informations aux outils de *build* par les modules
- Gestion de la compatibilité ascendante via la configuration d'un *epoch* au niveau d'un module pour activer des évolutions brisant la compatibilité

Compilation & implémentation

- `std::embed()` ressources externes disponibles au *runtime*
- Remplaçant à *`#ifdef ... #endif`*
- API d'interaction avec le système de build et le compilateur

Des questions ?

Bibliographie

[C++ Coding Standards] Herb Sutter et Andrei Alexandrescu

C++ Coding Standards : 101 Rules, Guidelines, and Best Practices

Addison Wesley Professional 0-321-11358-6

[Exceptional C++] Herb Sutter

Exceptional C++ : 47 Engineering Puzzles, Programming Problems, and Solutions

Addison Wesley 0-201-61562-2

[Exceptional C++ Style] Herb Sutter

Exceptional C++ Style 40 New Engineering Puzzles, Programming Problems, and Solutions

Addison Wesley 0-201-76042-8

[More Exceptional C++] Herb Sutter

More Exceptional C++

Addison Wesley 0-201-70434-X

Bibliographie (suite.)

[Effective C++] Scott Meyers

Effective C++ : 55 Specific Ways to Improve Your Programs and Designs
Addison Wesley 0-321-33487-6

[More Effective C++] Scott Meyers

More Effective C++ : 35 New Ways to Improve Your Programs and Designs
Addison Wesley 0-201-63371-X

[Effective STL] Scott Meyers

Effective STL : 50 Specific Ways to Improve Your Use of the Standard
Template Library
Addison Wesley 0-201-74962-9

[Effective Modern C++] Scott Meyers

Effective Modern C++ : 42 Specific Ways to Improve Your Use of C++11
and C++14
Addison Wesley 1-491-90399-6

Bibliographie (suite.)

[C++ Concurrency in action] Anthony Williams

C++ Concurrency in Action - Practical Multithreading

Manning 9781933988771

[CppCoreGuidelines] isocpp

C++ Core Guidelines

[https://github.com/isocpp/CppCoreGuidelines/blob/master/](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md)

[CppCoreGuidelines.md](https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md)

[isocpp C++ FAQ] isocpp

C++ FAQ

<https://isocpp.org/faq>

[Chaîne Youtube cppcon] cppcon

Vidéo CppCon

<https://www.youtube.com/user/CppCon/featured>

Bibliographie (suite.)

[Overload] ACCU

Overload

https://accu.org/journals/nonmembers/overload_issue_members/

[Guru of the Week] Herb Sutter

Guru of the Week

<http://www.gotw.ca/gotw/>

[C++11 Faq] Bjarne Stroustrup

C++11 - the new ISO C++ standard

<http://www.stroustrup.com/C++11FAQ.html>

[More C++ Idioms]

More C++ Idioms

https://en.wikibooks.org/w/index.php?title=More_C%2B%2B_Idioms

Bibliographie (suite.)

[C++ now]

C++ now

<http://cppnow.org/>

[C++ now GitHub]

GitHub C++ now

<https://github.com/boostcon>

[Boost C++ Libraries] Boris Schäling

The Boost C++ Libraries

<http://theboostcpplibraries.com/>

[C++17 features in "Tony Tables"] Tony Van Eerd

C++17 features in "Tony Tables"

https://github.com/tvaneerd/cpp17_in_TTs/blob/master/ALL_IN_ONE.md

Bibliographie (suite.)

[Changes between C++14 and C++17 DIS] Thomas Köppe

Changes between C++14 and C++17 DIS

<https://isocpp.org/files/papers/p0636r0.html>

[7 Features of C++17 that will simplify your code] Bartek

7 Features of C++17 that will simplify your code

<https://tech.io/playgrounds/2205/>

[7-features-of-c17-that-will-simplify-your-code/introduction](https://tech.io/playgrounds/2205/7-features-of-c17-that-will-simplify-your-code/introduction)

[C++ Stories] Bartek

C++ Stories

<https://www.cppstories.com/>

[Oleksandr Koval's blog] Koval

All C++20 core language features with examples

<https://oleksandrkv1.github.io/2021/04/02/cpp-20-overview.html>