C++

Grégory Lerbret

13 novembre 2019

Sommaire

Sommaire

Rappels historiques

- Développement de « C with classes » par Bjarne Stroustrup dans les années 80 aux Bell labs
- Renommé C++ en 1983
- Première version publique de CFront et de *The C++ Programming Language* en 1985
- Première normalisation en 1998
- Amendement en 2003
- Un *Technical Report* (TR1 publié en 2007) :
  - Partiellement implémenté par certains compilateurs ou Boost
  - Repris en partie dans les normes suivantes (C++11, 14 et 17) et TS
- Un projet de TR2 finalement transposé en *Technical Specification*

La « philosophie » du C++

- Langage multi-paradigme, impératif à typage statique déclaratif
- A visée généraliste
- Initialement, ajout des classes (Simula) au C : le *C with classes*
- Forte compatibilité avec le C : vaste sous-ensemble commun proche du C
- *Zero-overhead abstraction* : Ne payons pas pour ce que nous n'utilisons pas
- Large panel d'outils variés pour des développeurs responsables
- 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 : `http://www.open-std.org/JTC1/SC22/WG21/`
- Comité distinct de celui du C, mais des membres en commun dont certains pour l'échange entre groupes
- Pas de propriétaire du C++
- Actualité de normalisation, et du C++ en général, sur isocpp.org
- Une conférence annuelle : cppcon

isocpp.org n'est pas le site du comité

isocpp.org n'est pas le site officiel du comité de normalisation mais celui de *Standard C++ Foundation* dont le but est la promotion du C++
Les deux sont cependant très proches et partagent de nombreux membres

Norme & support

- Compilateur :
  - GCC : gcc.gnu.org/projects/cxx-status.html
  - Clang (LLVM) : clang.llvm.org/cxx_status.html
  - Visual studio : msdn.microsoft.com/fr-fr/library/hh567368.aspx
- Bibliothèque standard :
  - GCC : gcc.gnu.org/onlinedocs/libstdc++/manual/status.html
  - Clang : `http://libcxx.llvm.org/cxx1z_status.html`
- Vision globale : en.cppreference.com/w/cpp/compiler_support

Erreurs - Code retour 1/2

- Plusieurs variantes :
  - Code retour dédié
  - Valeur particulière indiquant un échec (`NULL`, `-1`)
  - Récupération de la dernière erreur (`errno`, `GetLastError()`)
- « Un test toutes les deux lignes de code »
- Nécessite de gérer « à la main » la remontée de la pile d'appel
- Adapté au traitement local des erreurs, pas au traitement « plus haut »

Erreurs - Code retour 2/2

Problèmes

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

Erreurs - Exception

- Lancée par un `throw`...
- ... et attrapée un `catch()` depuis un bloc `try`

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

Erreurs - Exception

- De n'importe quel type (y compris entier) ...
- ... mais il est recommandé qu'elles héritent de `std::exception` (via `std::logic_error`, `std::runtime_error` ou autre)
- `catch(...)` pour attraper les exceptions de tout type
- Les exceptions non attrapées terminent le programme
- Pas de `finally`
- Utilisées par la bibliothèque standard (p.ex. `std::bad_alloc`)

Erreurs - Critiques des exceptions 1/2

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

Erreurs - Critiques des exceptions 1/2

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

Non, pas vraiment

- Recevable à l'origine, mais maintenant une exception non levée ne coute quasiment rien
- Souvent étayée par une comparaison avec une non gestion d'erreur : est-ce pertinent ?

Erreurs - Critiques des exceptions 2/2

- Des arguments très variés :
  - Mauvais support par les différents outils

Très variable

- 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 1/2

- *No-throw guarantee* : l'opération ne peut pas échouer (p.ex. `swap`)
- *Strong exception safety* : pas d'effet de bord, état conservé
- *Basic exception safety* : invariants conservés, pas de fuite
- *No exception safety* : aucune garantie

Do

Privilégiez les garanties les plus fortes possibles (*no-throw* et *strong*)

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);                     // And swap
  return *this;}
```

Erreurs - Exceptions et bonnes pratiques

Do

Throw by value, catch by reference (« C++ Coding Standard » chap. 73)

Do

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

Don't

- N'utilisez jamais les exceptions pour contrôler le flux d'exécution
- Réservez les exceptions au signalement d'erreurs

Erreurs - Et `assert` ?

- Arrête le programme si l'expression est évalué à 0 . . .
- . . . et 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 ou codes retour pour les erreurs d'exécution et la vérification des données externes
- Réservez `assert` pour les erreurs de programmation et la vérification des contrats internes

Do

Préférez, pour les erreurs, les exceptions aux codes retour (« C++ Coding Standard » chap. 72)

Don't

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

Gestion des ressources - Gestion manuelle 1/3

Premier problème

Comment gérer les erreurs ?

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

```
FILE* file = NULL;
char* memory = NULL;
...
memory = malloc(50);
if(!memory) goto err;
...
file = fopen("bar.txt", "r");
if(!file) goto err;
...
err:
free(memory);
if(file) fclose(file);
```

Ressources - Gestion manuelle 2/3

- Laborieux et source d'erreur
- Difficile à mettre en place en présence d'exception (pas de SESE)

Ressources - Gestion manuelle 2/3

- Laborieux et source d'erreur
- Difficile à mettre en place en présence d'exception (pas de SESE)

Quiz : Comment éviter les fuites mémoires ici ?

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

Ressources - Gestion manuelle 3/3

Second problème

Comment copier des classes possédant des ressources ?

- Constructeurs et opérateurs générés copient les adresses des pointeurs
- La 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)

Do

Big Rule of three : si vous devez définir l'une des trois fonctions de base que sont le constructeur de copie, l'opérateur d'affectation ou le destructeur, alors vous devriez définir les trois

Ressources - RAII

- *Resource Acquisition Is Initialization*
- Acquisition des ressources lors de l'initialisation de l'objet ...
- ... et libération automatique lors de sa destruction
- Propriété intrinsèque des objets « par design »
- Fonctionnement de la bibliothèque standard (conteneurs, fichiers, `auto_ptr`)
- Conséquences :
  - Les objets sont créés dans un état cohérent, testable et utilisable
  - Les ressources sont automatiquement libérées à la destruction de l'objet
  - Les capsules RAII se copient sans effort

Ressources - RAII 2/4

Do

Faites des constructeurs qui construisent des objets :

- Cohérents
- Utilisables
- Si possible, complètement initialisés

Don't

Évitez les couples constructeur « vide » et fonction d'initialisation

Don't

Évitez les couples constructeur « vide » et ensemble de mutateurs

Ressources - RAII

Attention : signalement d'erreur

- Pas d'erreur ni d'exception pour les destructeurs
- Libération de ressources peut échouer (p.ex. `flush` lors de la fermeture de fichier)

```
{
  ifstream src("input.txt");
  ofstream dst("output.txt");
  copyFiles(src, dst);
}

removeFile(src);
// Potentielle perte de donnees
```

Ressources - RAII

Attention : std::auto_ptr

- std::auto_ptr est copiable
- Cette copie transfère la responsabilité de la ressource

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

auto_ptr<int> bar(new int(5));
foo(bar);
// Erreur : bar n'est plus utilisable
cout << *bar << "\n";
```

Ressources - Loi de Déméter 1/3

- . . . Ou principe de connaissance minimale
- « Ne parlez qu'à vos amis proches »
- 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

Les agrégats et les conteneurs dont le rôle est de contenir des données

Objectifs

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

Ressources - Loi de Déméter 2/3

Do, agrégats

- Utilisez des structures plutôt que des classes
- Laissez les membres publics
- Fournissez, éventuellement, des constructeurs initialisant les données

Do, conteneurs

Respectez l'interface et la logique des conteneurs standard

Do, classes de service

- Exposez des services non des données
- Pas de données publiques
- Limitez les accesseurs et encore plus les mutateurs

Ressources - Loi de Déméter 3/3

Conseil

- N'hésitez pas à étendre l'interface de classe avec des fonctions libres
- Pensez à l'amitié pour implémenter 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 Garage Collector ?

- Pas de GC dans le langage ni dans la bibliothèque standard
- ... mais un sujet de discussion récurrent
- Au moins un GC en bibliothèque tierce (Hans Boehm) ...
- ... mais limité par manque de support par le langage
- Fondamentalement non déterministe : acceptable pour la mémoire pas pour d'autres ressources
- Beaucoup plus 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 mais peut-être plus tard ...

Ressources - Conclusion 1/2

Do

Utilisez RAII

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

Do

Respectez Déméter

Ressources - Conclusion 2/2

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 1/3

- *Standard Template Library* : partie de la bibliothèque standard comprenant :
  - Classes conteneurs et `std::basic_string` : données
  - Itérateurs : parcours des conteneurs
  - Algorithmes : manipulation des données via les itérateurs

Note

Également 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 de POO

STL 2/3

Intérêts

- n conteneurs et m algorithmes, seulement m implémentations
- Tout nouvel algorithme est disponible sur tous 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 3/3

A nuancer

Malgré tout des algorithmes en fonction interne 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

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

Conteneurs de pointeurs

Pas de libération des objets « pointés »

- . . . accessibles via un itérateur
- Possibilité, en général, de fournir une politique d'allocation
- Vu des algorithmes, tout ce qui fournit une paire d'itérateurs est un conteneur

STL Conteneurs - Conteneurs séquentiels 1/3

- `std::vector`
  - Tableau de taille variable
  - Éléments contigus
  - Accès indexé
  - Croissance en temps amorti
  - Modifications en fin de vecteur (couteux ailleurs)
  - Compatible avec l'organisation mémoire des tableaux C (`T* ptr = &vec[0];`)

Attention : `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 objet *proxy* vers celui-ci
- Voir [Effective STL] item 18

Le conteneur par défaut

STL Conteneurs - Conteneurs séquentiels 2/3

- `std::list`
  - Liste doublement chaînée
  - Accès bidirectionnel non indexé
  - Modification n'importe où à faible coût
  - Plusieurs algorithmes membres (tri, fusion, renversement, suppression, élimination de doublon)
- `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 3/3

- `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 la taille en nombre de *bytes* non de caractères

- Contiguïté non garantie en C++98/03 (mais respectée en pratique)
- Un cousin peu utilisé pour les caractères larges : `std::wstring`

Une 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

Pas des tables de hachage

Généralement implémentés sous forme d'arbres binaires de recherche balancés (*red-black tree* le plus souvent)

- Le critère d'ordre est configurable (strictement inférieur par défaut)

Attention

Le critère d'ordre est un ordre strict

- Algorithmes membres (recherche) avec pour motivation les performances

STL Conteneurs - Adaptateurs 1/2

- 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 toujours le plus grand
  - Basée sur `std::vector` ou `std::deque`
  - Critère d'ordre configurable (strictement inférieur par défaut)

STL Conteneurs - Adaptateurs 2/2

```
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 (paramètre template)
  - Conçu pour réduite l'empreinte mémoire
  - Pas d'itérateur ni d'interface « STL »

`std::bitset` et `std::vector<bool>`

L'objectif de gain mémoire étant déjà adressé par `std::bitset` plus adapté, pourquoi `std::vector<bool>` n'est-il pas un vrai conteneur de booléen ?

- Conteneurs non-standard :
  - Listes simplement chaînées
  - Tables de hachage
  - Tableaux de taille fixe
  - Tampons circulaires
  - Arbres et graphes
  - Des variantes des conteneurs STL (p.ex. les « *ropes* »)

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 1/2

Do, par défaut

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

Do, performances

Une seule règle : mesurez avec des données réelles sur la configuration cible

Flux d'octets

Un flux d'octets est un `std::vector<unsigned char>`, pas un `std::vector<char>` et encore moins `std::string`

STL Conteneurs - Choix du conteneur 2/2

Conseils

- Voir [Effective STL] item 1
- 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

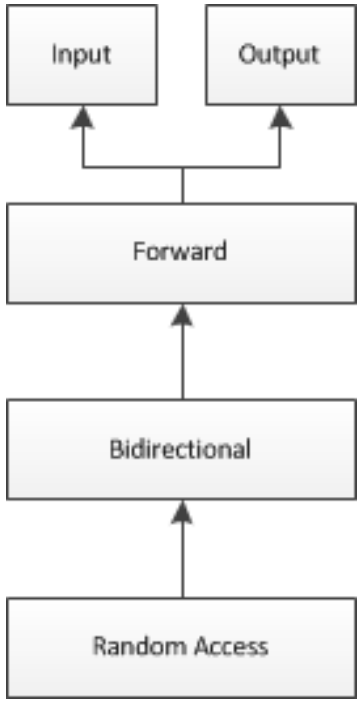
- Abstraction permettant de parcourir des collections d'objets
- Interaction entre les conteneurs et les algorithmes
- Interface similaire à celle d'un pointeur
- Quatre types d'itérateurs :
  - `iterator` et `const_iterator`
  - `reverse_iterator` et `const_reverse_iterator`
- Itérateurs sur un conteneur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

Attention

Les itérateurs de fin pointent un élément après le dernier (« *past the end* ») et ne doivent pas être déréférencés

STL Itérateurs - Catégories et opérations

- Opérations communes : copie, affectation et incrémentation
- Une hiérarchie de cinq catégories d'itérateurs :
  - *Input* : égalité (== et !=) et lecture
  - *Output* : écriture
  - *Forward* : Parcours multiples possibles
  - *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
`std::vector`	*Random access*
`std::deque`	*Random access*
`std::list`	*Bidirectionnal*
`std::map` et `std::multimap`	*Bidirectionnal*
`std::set` et `std::multiset`	*Bidirectionnal*

STL Itérateurs - Itérateur d'insertion

- Itérateurs de type *Output*
- Insertion de nouveaux éléments plutôt que modification d'éléments existants
- Trois types :
  - Insertion en queue de conteneur : `back_inserter`
  - Insertion en tête : `front_inserter`
  - Insertion à la position courante : `inserter`

STL Algorithmes - Foncteurs 1/2

- Un foncteur est une instance de classe définissant un `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";    // Affiche 1
```

STL Algorithmes - Foncteurs 2/2

- Avantage : possèdent données membres
- Plusieurs foncteurs standard encapsulant les opérateurs : `plus`, `minus`, `equal`, `less`, ...
- Construction de foncteur :
  - Depuis des pointeurs de fonctions : `prt_fun`
  - Depuis des fonctions membres : `mem_fun`, `mem_fun1`, ...
  - En niant d'autres foncteurs : `not1`, `not2`
  - En fixant des paramètres : `bind1st`, `bind2nd`

STL Algorithmes - Prédicats

- Un prédicat est un « appelable » (foncteur ou pointeur de fonction) 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 ==) qui peut être remplacé

STL Algorithmes - Parcours 1/2

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

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

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

Syntaxe

Les exemples utilisent une initialisation de conteneur introduite pas C++11

- Version du map/apply fonctionnel

STL Algorithmes - Parcours 2/2

- 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 1/2

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

STL Algorithmes - Recherche linéaire 2/2

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

Variante « _if »

Les algorithmes suffixés par _if utilise un prédicat plutôt qu'une valeur

- `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 donnée

STL Algorithmes - Recherche dichotomique 1/3

- Pré-requis : ensemble trié
- `std::lower_bound()` retourne un itérateur sur le première 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};

// Affiche 7
cout << *lower_bound(foo.begin(), foo.end(), 6);
// Affiche 9
cout << *lower_bound(foo.begin(), foo.end(), 9);
```

STL Algorithmes - Recherche dichotomique 2/3

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

Attention

`std::lower_bound()`, `std::upper_bound()` et `std::equal_range()` retourne un résultat qui peut ne pas être la valeur recherchée

- `std::binary_search()` indique si l'élément cherché est présent

STL Algorithmes - Recherche dichotomique 3/3

Attention

Pas de fonction de recherche dichotomique retournant l'élément cherché s'il existe, il faut bâtir cette recherche sur ces fonctions élémentaires

```
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);  // iterateur 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};

// Affiche 3
cout << count(foo.begin(), foo.end(), 5);
// Affiche 0
cout << count(foo.begin(), foo.end(), 2);
```

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

STL Algorithmes - Comparaison 1/3

- `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(), var.begin());  // false
```

STL Algorithmes - Comparaison 2/3

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 l'opérateur == à `std::equal()` pour comparer un conteneur complet

STL Algorithmes - Comparaison 3/3

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

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

pair<vector<int>::iterator,
vector<int>::iterator> res;
res = mismatch(foo.begin(), foo.end(), bar.begin());
// Affiche 9 12
cout << *res.first << " " << *res.second;
```

- Ou l'itérateur de fin en cas d'égalité

STL Algorithmes - Remplissage 1/2

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

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

fill(foo.begin(), foo.end(), 12);
// foo : 12 12 12 12
```

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

Constructeur et remplissage

Les constructeurs des conteneurs séquentiels permettent de remplir le conteneur avec une valeur donnée : `vector<int> foo(4, 12);`

STL Algorithmes - Remplissage 2/2

- `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 1/2

- `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);
// foo : 4 8 7 9 12 8
```

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

STL Algorithmes - Remplacement 2/2

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

Variante « _copy »

Les algorithmes suffixés par _copy fonctionne 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 vérifiés par le prédicat par une valeur donnée

STL Algorithmes - Suppression 1/2

- `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);     // foo : 4 7 9 9 ...
```

STL Algorithmes - Suppression 1/2

- `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);     // foo : 4 7 9 9 ...
```

`std::remove()` ne supprime pas

`std::remove()` ramène simplement les éléments à conserver vers le début de l'ensemble et retourne l'itérateur correspond à la nouvelle fin

STL Algorithmes - Suppression 1/2

- `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);     // foo : 4 7 9 9 ...
```

`std::remove()` ne supprime pas

`std::remove()` ramène simplement les éléments à conserver vers le début de l'ensemble et retourne l'itérateur correspond à la nouvelle fin

L'idiome *Erase-Remove*

La suppression réelle des éléments passent par un appel à la fonction membre `erase()` sur les éléments situés après l'itérateur retourné par `std::remove()`

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

STL Algorithmes - Suppression 2/2

- `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());
// foo : 4 5 7 9 5 ...
```

- `std::unique_copy()` copie l'ensemble en ne conservant qu'un des éléments consécutifs égaux

STL Algorithmes - Transformation 1/2

- `std::transform()` applique une transformation à tous les éléments d'un ensemble ...

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

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

STL Algorithmes - Transformation 2/2

- ... ou de deux 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>());
// baz : 6 8 13 10
```

STL Algorithmes - Rotation

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

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

rotate(foo.begin(), foo.begin() + 2, foo.end());
// foo : 7 9 12 4 5
```

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

STL Algorithmes - Partitionnement 1/3

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

```
bool isOdd(int i) { return (i%2)==1; }

vector<int> foo{4, 13, 28, 9 , 54};
partition(foo.begin(), foo.end(), isOdd);
// foo : 9 13 28 4 54 (ou {9 13 4 28 54} ou ...)
```

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

Attention

L'ordre relatif n'est pas conservé

STL Algorithmes - Partitionnement 2/3

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

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

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

STL Algorithmes - Partitionnement 2/3

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

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

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

Pourquoi deux fonctions ?

La performance, la stabilité est couteuse en temps et pas toujours nécessaire

STL Algorithmes - Partitionnement 3/3

- `std::nth_element()` réordonne les éléments de sorte que :
  - L'élément situé sur l'itérateur pivot soit celui qui serait à cette place si l'ensemble était trié
  - Que les éléments situés avant ne soient pas supérieurs
  - Que les éléments situés après ne soient 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());
// foo : 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());
// foo : 4 9 13 28 54
```

Attention

L'ordre relatif des éléments égaux n'est pas 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());
// foo : 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());
// foo : 1 8 3 7 9 4 2 6 5
```

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));
// baz : 1 2 5 5 6 8
```

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

STL Algorithmes - Opérations ensemblistes 1/2

- `std::includes()` vérifie si tous les éléments d'un ensemble trié 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 2/2

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

```
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));
// baz : 1 2 5 6 8
```

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

STL Algorithmes - Gestion de « tas »

- Le tas (*heap*) est une structure permettant la récupération rapide 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

push, *pop* et structure de tas

`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 1/4

- `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 2/4

- `std::adjacent_difference()` effectue la « différence » entre chaque élément de l'ensemble et celui qui le précède

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

adjacent_difference(foo.begin(), foo.end(),
                    back_inserter(bar), minus<int>());
// bar : 18 -13 1 2
```

- Opérateur configurable

STL Algorithmes - Numérique 3/4

- `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 4/4

- `std::partial_sum()` « somme » partielle d'un ensemble
- Chaque élément de l'ensemble de sortie est égal à 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));
// bar : 1 3 6 10
```

- Opérateur configurable

STL Algorithmes - Au delà des conteneurs 1/3

- 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 2/3

- 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);  // foo : 5 5 5 5
```

STL Algorithmes - Au delà des conteneurs 3/3

- 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,
```

- Buffers de flux : `istreambuf_iterator` et `ostreambuf_iterator`

Attention

Le séparateur est ajouté après chaque élément, y compris le dernier

STL Conclusion 1/4

Do

Préférez les conteneurs aux tableaux C

Attention

`operator[]` ne vérifie pas les bornes

Don't

N'utilisez pas d'itérateur invalidé

Attention

Ne stockez pas objets polymorphiques dans les conteneurs ou uniquement via des pointeurs intelligents

STL Conclusion 2/4

Do, performances

Mesurez !

Conseils, performances

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

Do

Préférez les algorithmes standard aux algorithmes tierces, aux algorithmes « maisons » et aux boucles

Un petit bémol

En terme de performance, les algorithmes standard sont généralement très bons mais, étant génériques, pas forcément optimaux dans pour situation particulière

STL Conclusion 3/4

Do

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

Do

Respectez la sémantique des algorithmes :

- 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

Présentation

- Approuvé unanimement 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 plutôt que par les fonctionnalités (*train model*)
  - Des versions fréquentes (3 ans : 2011, 2014, 2017, . . .)
  - Alternance majeures/mineures ou intermédiaires ?
  - Voir Trip report: Winter ISO C++ standards meeting
- Très bon support par les versions récentes de GCC, Clang et VC++
- Objectifs C++11 et suivants : plus sûr, plus simple, aussi rapide que possible, meilleure détection d'erreur au compile-time

Nouveaux types entiers 1/3

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

Intégration depuis C99

Outre les nouveaux types : *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
- `intmax_t` et `uintmax_t` : types entiers le plus grand disponibles

Nouveaux types entiers 2/3

- `int<N>_t` et `uint<N>_t` : types entiers de N bits
  - N = 8, 16, 32 ou 64
  - Types signés obligatoirement en complément à 2
  - Pas de bit de padding
  - Optionnels (uniquement si un type compatible existe)
- `int_least<N>_t` et `uint_least<N>_t` : plus petits types entiers d'au moins N (8, 16, 32 ou 64) bits
- `int_fast<N>_t` et `uint_fast<N>_t` : plus rapides types entiers d'au moins N (8, 16, 32 ou 64) bits
- `intptr_t` et `uintptr_t` : type entier capable de contenir une adresse
  - Doit pouvoir être reconvertit en `void*` avec une valeur égale au pointeur original
  - Optionnels

Nouveaux types entiers 3/3

- 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` (`imaxabs`, `imaxdiv`, `strtoimax`, `strtoumax`, `wcstrtoimax` et `wcstrtoumax`)
- 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 ne sont permises que pour les types POD
  - Utilisation de `memcpy()` ou `memmove()`
  - Utilisation de `goto` « au-dessus » 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
- 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

En C++14

Suppression de la contrainte « Pas d'initialisation *brace-or-equal-initializers* des données membres non-statiques »

POD Généralisé - Classe POD C++98

- Classe agrégat
- Pas de donnée membre non-statique de type non-POD (ou de tableau de non-POD) ni de référence
- Pas d'opérateur d'assignation défini par l'utilisateur
- Pas de destructeur défini par l'utilisateur

POD Généralisé - Classe POD C++11 1/3

- 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

POD Généralisé - Classe POD C++11 1/3

- 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
- L'opération des classes de bases et des membres non-statiques est triviale

POD Généralisé - Classe POD C++11 1/3

- 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
- L'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 2/3

- *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 3/3

- *Standard-layout* :
  - Pas de donnée membre non-statique non-*Standard-layout* ni de référence
  - Pas de classe de base non-*Standard-layout*
  - Pas de fonction virtuelle ni de classe de base virtuelle
  - Même accessibilité de toutes les données membres non-statique
  - Les données membres non-statiques localisées dans une unique classe de l'arbre d'héritage
  - Pas de classe de base du type de la première donnée membre non-statique
- POD :
  - *trivial*
  - *standard layout*
  - Pas de donnée membre non-statique non-POD
- Ajout des traits correspondants : `std::is_trivial`, `std::is_trivially_copyable` et `std::is_standard_layout`

POD Généralisé - Objectifs

- Certaines opérations réservées au POD deviennent accessibles aux classes remplissant les contraintes de la sous-notion correspondante
- Relâchement / adaptation de certaines contraintes par rapport à C++98 :
  - Constructeurs ou destructeurs déclarés =`default` autorisés
  - Données membres non-statiques ne sont plus nécessairement publiques
  - Classes de base non virtuelles autorisées

POD Généralisé - Quelques conséquences

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

Unions généralisées

- Les types avec des constructeurs, opérateurs d'assignation ou destructeurs définis par l'utilisateur sont maintenant acceptés comme membre 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` ? `nullptr` ! 1/2

- C++ 98, pointeur nul : `0` ou `NULL`
- Cohabite mal avec les surcharges

`0` ou `NULL` ? `nullptr` ! 1/2

- C++ 98, pointeur nul : `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 ! 2/2

- C++ 11, pointeur nul : `nullptr`
  - Unique pointeur du type `nullptr_t`
  - Conversion implicite `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` pour désigner les pointeurs nuls

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

Plus généralement, 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 inlines
- 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];
```

Attention

Les extensions types VLA n'ont aucun rapport avec `constexpr`, elles prennent place au *run-time*

Do

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

Extended `sizeof`

- Permet l'appel de `sizeof` sur des membres non statiques

```
struct Foo { int bar; };

// Valide en C++11 mais 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 1/7

- Deux constats :
  - Copie peut être couteuse ou impossible
  - Copie inutile lorsque l'objet source est immédiatement détruit

Optimisation des copies

Partiellement adressé en C++98/03 par des optimisations classiques : élision de copie et (N)RVO

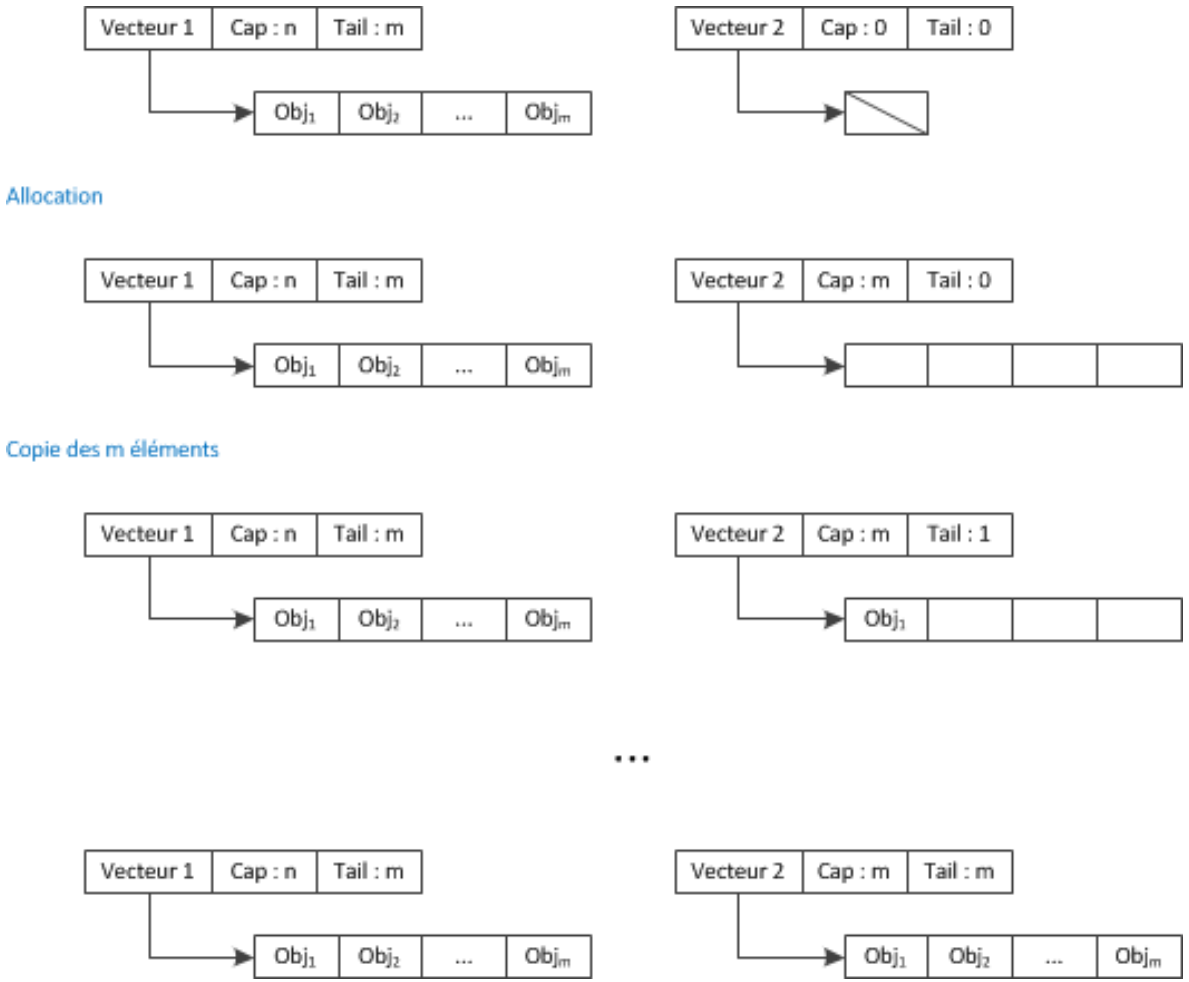
- « Échange » de données légères plutôt que copie profonde
- Déplacement seulement si
  - Type déplaçable et
  - Instance sur le point d'être détruite ou explicitement déplaçable

Attention

Les données ne sont plus présentes dans l'objet initial

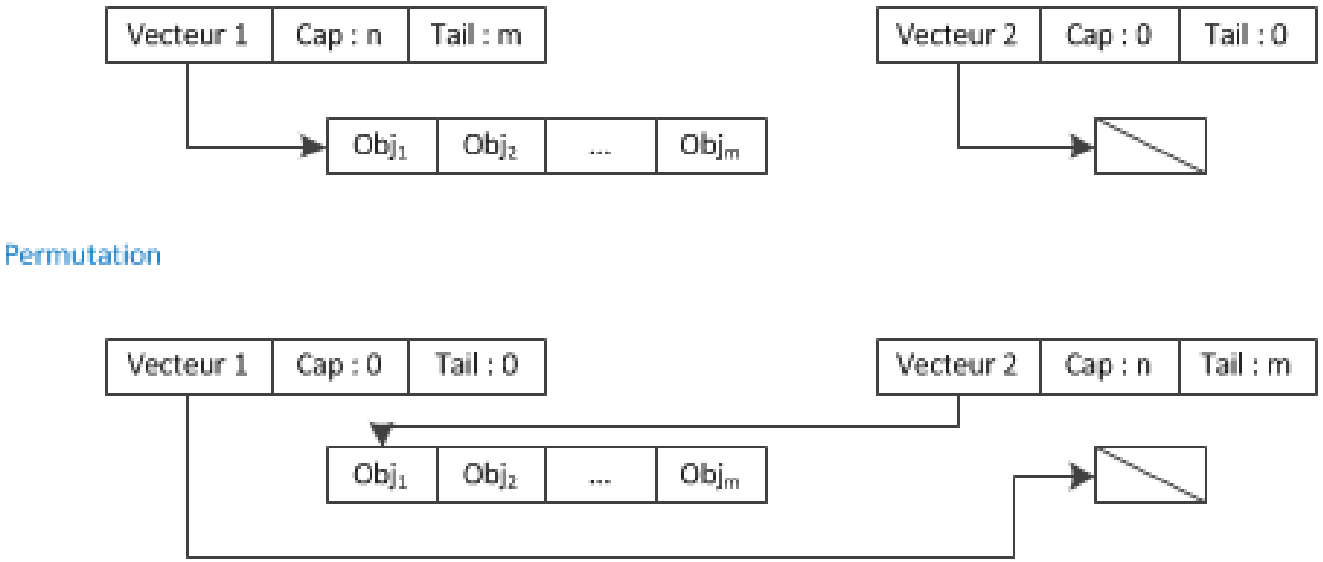
Sémantique de déplacement

- Copie :



Sémantique de déplacement

- Déplacement :



Sémantique de déplacement 4/7

- *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()` ?

Objectif : le *perfect forwarding* (Voir N1385)

Sémantique de déplacement 5/7

- 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 « opérateur » de 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 6/7

Rule of five

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

Rule of zero

Lorsque c'est possible, n'en définissez aucune

Aller plus loin

Voir Élégance, style épuré et classe (Loïc Joly)

Sémantique de déplacement 7/7

Sémantique de déplacement dans la bibliothèque standard

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

Bonnes pratiques ?

Nombreux débats sur les bonnes pratiques (existantes et nouvelles)

Initializer list

- En C++98/03, initialisation de conteneurs avec les valeurs impossible

```
vector<int> foo;

foo.push_back(1);
foo.push_back(56);
foo.push_back(18);
foo.push_back(3);
```

- C++11 le permet

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

Initializer list

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

Attention : Accéder, pas contenir !

- `std::initializer_list` référence mais ne contient pas les valeurs
- Contenues dans un tableau temporaire de même durée de vie
- Copier un `std::initializer_list` ne copie pas les données

- Trois fonctions membres :
  - `size()` : nombre d'éléments
  - `begin()` : itérateur de début de liste
  - `end()` : itérateur de fin de liste
- 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

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
```

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 = {};                       // OK
```

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 = {};                       // OK
```

- 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{};                          // OK
```

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
```

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 6/6

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 virtuel ni de membres non statiques protégés ou privés

Do

Préférez l'initialisation `{}` aux autres formes

Do

Prenez garde à la différence entre `std::vector<int> foo(2)` et `std::vector<int> foo{2}`

auto

- Déduction/inférence de type
- Type se déduit de l'initialisation

`auto`

- Déduction/inférence de type
- Type se déduit de l'initialisation

Inférence de type ≠ typage dynamique

Deux notions orthogonales, le typage reste statique

Inférence de type ≠ typage faible

Encore deux notions orthogonales

typage dynamique ≠ typage faible

Toujours deux notions orthogonales

`auto`

Typage & vocabulaire

- Statique : type porté par la variable et ne varie pas
- Dynamique : type porté par la valeur, le type de la variable change au fil des affectations
- Absence : variable non typée, le type est imposé par l'opération
- Parfois une distinction *compile-time* / *run-time*

auto

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

```
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` sont 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;
```

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

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

auto

- Une tendance forte : *Almost Always Auto* (AAA)
- Voir GOTW 94 : AAA Style
- Plusieurs avantages (robustesse, performances, maintenabilité, . . .)
  - Variables forcément initialisées
  - Typage correct et précis
  - Garanties conservées au fil des corrections et refactoring
  - Généricité et simplification du code

`auto`

- Une tendance forte : *Almost Always Auto* (AAA)
- Voir GOTW 94 : AAA Style
- Plusieurs avantages (robustesse, performances, maintenabilité, . . .)
  - Variables forcément initialisées
  - Typage correct et précis
  - Garanties conservées au fil des corrections et refactoring
  - Généricité et simplification du code

Quiz

Quelle est le type de retour de la fonction membre `size()` d'une `std::list<std::string>` ?

`auto`

- Une tendance forte : *Almost Always Auto* (AAA)
- Voir GOTW 94 : AAA Style
- Plusieurs avantages (robustesse, performances, maintenabilité, . . .)
  - Variables forcément initialisées
  - Typage correct et précis
  - Garanties conservées au fil des corrections et refactoring
  - Généricité et simplification du code

Quiz

Quelle est le type de retour de la fonction membre `size()` d'une `std::list<std::string>` ?

Réponse

```
std::list<std::string>::size_type
```

`auto` 6/6

- Des limitations
  - Erreur de déduction de type : typage explicite de l'initialiseur
  - Pas d'initialisation possible : `decltype`
  - Interfaces, rôles, contexte : concepts ?

Mon point de vue sur AAA

- Arguments en faveur de AAA convaincants
- Mais les vieilles habitudes sont dures à perdre

Attention

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

Dépréciation

Mot-clé `register` également déprécié

decltype

- `decltype` déduit le type d'une variable ou d'une expression
- Et permet donc de créer 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

- Permet l'utilisation de fonctions membres dans `decltype` sans appel au constructeur
- Typiquement sur des templates acceptant des types sans constructeur commun mais avec une fonction membre commune

```
struct foo {
  NonDefaufoo(const foo&) { }
  int bar () const { return 1; } };

decltype(foo().bar()) n2 = 5;                    // Erreur
decltype(std::declval<foo>().bar()) n2 = 5; // OK, int
```

Attention

Uniquement utilisable dans des contextes non évalués

Déduction du type retour 1/2

- `auto` et `decltype` permettent de déduire le type retour d'une fonction

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

- Particulièrement utiles pour des fonctions template

Déduction du type retour 1/2

- `auto` et `decltype` permettent de déduire le type retour d'une fonction

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

- Particulièrement utiles pour des fonctions template

Quiz : Que mettre à la place des XXX ? T, U, autre chose ?

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

Déduction du type retour 2/2

Solution

- Pas de bonnes réponses 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

Conteneurs

- `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
```

  - Permet la vérification des index à la compilation

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

Conteneurs

- `std::forward_list` : liste simplement chaînée

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

Conteneurs

- Nouveaux 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::multimap`, `std::set` et `std::multiset`

Pourquoi unordered ?

- Structures fondamentalement non ordonnées
- Trop nombreuses implémentations `hash_XXX` existantes

Conteneurs 4/6

- `shrink_to_fit()` réduit la capacité des `vector`, `deque` et `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()` récupère le « tableau C » d'un `vector`

`foo.data()` ou `&foo[0]`

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

Conteneurs

- `emplace()`, `emplace_back()` et `emplace_front()` construisent directement 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 les copies inutiles restantes (malgré élision de copie, (N)RVO et sémantique de déplacement) et gagner en performance

Conteneurs 6/6

- `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`
  - `all()` teste si tous les bits sont levés
  - `to_ullong()` converti en `unsigned long long`

Do

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

Itérateurs

- Fonctions membres `cbegin()`, `cend()`, `crbegin()` et `rcend()` permettant d'obtenir systématiquement des `const_iterator`
- Fonctions libres `std::begin()` et `std::end()`
  - conteneur : équivalente aux 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

- Fonctions libres `std::begin()` et `std::end()` (cont.)
  - compatible avec les conteneurs non-STL proposant les fonctions membres `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 3/4

Conseils

`using std::begin` et `using std::end` pour permettre à l'ADL de fonctionner 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()` permettant de retrouver l'itérateur suivant ou précédant un itérateur
- Famille d'itérateur : `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), isOdd); // 4
```

Algorithmes - Comparaison 1/4

- `std::all_of()` indique si tous les éléments de l'ensemble vérifient un prédicat

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

- Retourne vrai si l'ensemble est vide

Algorithmes - Comparaison

- `std::any_of()` indique si au moins un élément de l'ensemble vérifie un prédicat

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

- Retourne faux si l'ensemble est vide

Algorithmes - Comparaison 3/4

- `std::none_of()` indique si aucun élément de l'ensemble ne vérifie le prédicat

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

- Retourne vrai si l'ensemble est vide

Algorithmes - Comparaison 4/4

- `std::is_permutation()` indique 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
```

- Égalité des éléments mais pas de leur ordre

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), isOdd);
// 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 1/2

- `std::is_partitioned()` indique si un ensemble est partitionné, c'est à dire si les éléments vérifiant un prédicat sont avant ceux ne le vérifiant pas

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

is_partitioned(begin(foo), end(foo), isOdd); // false
is_partitioned(begin(bar), end(bar), isOdd); // true
```

Algorithmes - Partitionnement 2/2

- `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), isOdd); // 4
```

Algorithmes - Tri

- `std::is_sorted()` indique si l'ensemble est ordonnée (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 non ordonné

```
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 1/3

- 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 << " ";     // Idem
```

Range-based for loop 2/3

Range-based for loop et modification

Pour modifier les éléments du conteneur la variable d'itération doit être une référence

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

for(auto& var : foo)
  var = 5;     // foo : 5 5 5 5
```

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

Range-based for loop 3/3

Do

Préférez *range-based for loop* aux boucles for classiques et à l'algorithme `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 au for classique
- Plus robuste, plus sûr

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

```
cout << stoi("56"); // Affiche 56
```

- S'arrêtent sur le premier caractère non convertible

std::string & conversions

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

```
cout << to_string(56); // Affiche 56
```

- std::to_wstring() : conversion 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<X>()` à `sscanf()`, `atoi()` ou `strto<X>()`

Do

Préférez `std::to_string()` à `s(n)printf()` ou `itoa()`

Alternative et complément

Boost.Lexical_cast permet également 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
- `u16string` spécialisation de `basic_string` pour caractères 16 bits
- `u32string` spécialisation de `basic_string` pour caractères 32 bits
- Même interface que `std::string`

Nouvelles *strings literals* 1/2

- Des 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 *strings literals* 2/2

- Des 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)--";
```

- Les deux se composent

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

User-defined literals 1/3

- Possibilité de définir des littéraux « utilisateur »
- Nombre (entier ou réel), caractère ou chaîne suffixé par un identifiant
- Identifiants « utilisateur » préfixés par _
- Définissable 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 2/3

- 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

Ne fonctionne que pour les littéraux numériques

User-defined literals 3/3

- 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;
```

- Fonction de construction : `std::make_tuple()`

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

std::make_tuple ou constructeur ?

`std::make_tuple()` permet de déduire les 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`
  - En fait, construction d'un `std::tuple` de référence sur les paramètres fournis

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

- Fonction template d'accès aux éléments du tuple par l'indice

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

Attention

Les indices commencent à 0

std::tuple

- Fonction de concaténation : std::tuple_cat()

```
auto foo = make_tuple(5, 'e');
auto bar = make_tuple(98L, 'r');
auto baz = tuple_cat(foo, bar); // baz : 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` permet de regrouper localement des éléments sans lien sémantique

Do

Préférez `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 qui aurait le même effet

Do

Préférez =delete des constructeurs de copie et opérateur d'affectation à une déclaration privée sans définition pour rendre une classe non copiable

=default ou non définition ?

- Consensus plutôt du côté de la non-définition
- Mais un intérêt documentaire réel à =default

Initialisation par défaut des membres

- Initialisation des membres d'une classe lors de leur 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 1/2

- 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 2/2

Do

Utilisez la délégation de constructeur pour mutualiser le code commun aux constructeurs d'une classe

Don't

Évitez la délégation pour l'initialisation constante commune de membres, préférez l'initialisation d'attributs

Héritage de constructeur 1/2

- Indique que la classe hérite des constructeurs de la classe mère
- Le compilateur génère le constructeur correspondant
  - 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 2/2

- Possible de redéfinir un des constructeurs dans la classe dérivée

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

Attention : valeurs par défaut

Les constructeurs ayant des paramètres par défaut produisent toutes les combinaisons de constructeurs sans valeur par défaut correspondantes

Restriction (héritage multiple)

Il n'est pas possible d'hériter de deux constructeurs ayant la même signature

override

- Indique qu'une classe dérivée redéfinie une fonction d'une classe de base

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

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

override

- Et provoque une erreur de compilation si la fonction n'existe pas dans la classe de base ou 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` 3/3

Objectifs

- Documentaire
- Détecter les non-reports de modification lors d'un refactoring
- Permettre aux outils d'analyse de code de détecter des redéfinitions involontaires

Do

Marquez `override` les fonctions que vous redéfinissez

Do

Utilisez `virtual` uniquement à la base de l'arbre d'héritage et `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

- Le spécificateur `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, en compile-time, si une expression ne peut pas lever une exception
- Dans le cas d'un appel de fonction, revient à tester si la fonction est `noexcept`

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

Do

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

« Conversion » exception / pointeur 1/2

- `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 2/2

```
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, ne fait rien sinon
- `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 1/2

- Des 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 2/2

Do

Préférez les énumérations fortement typées aux énumérations classiques

Bémol

Pas de manière simple et robuste de récupérer la valeur ou l'intitulé de l'énuméré

std::function

- Wrapper encapsulant un appelable 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

Note

Les foncteurs ne sont pas transmis aux algorithmes par ce mécanisme mais par des paramètres templates identifiés aux types internes du compilateur

std::mem_fn

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

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

Foo foo;
std::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 consort

std::bind

- Construit un *function object* en liant des paramètres à un appelable
- *Placeholders* std::placholders::_1, std::placholders::_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, std::bind2nd et consort

lambda et fermeture 1/3

Vocabulaire

- Lambda : fonction anonyme
- Fermeture : capture des variables libres de l'environnement lexical

lambda et fermeture

Vocabulaire

- Lambda : fonction anonyme
- Fermeture : capture des variables libres de l'environnement lexical

- Syntaxe : `[capture](parametres) -> type_retour {instructions}`
- Capture :
  - [ ] : pas de capture
  - [x] : capture x par valeur
  - [&y] : capture y par référence
  - [&] : capture tout par référence
  - [=] : capture tout par valeur
  - [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

lambda et fermeture


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

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

- 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 `[&]`
- La capture préserve la constante des variables capturées
- Les variables globales et statiques ne peuvent pas être capturées

lambda et fermeture

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
```

- Le type de retour peut être omis s'il n'y a qu'une instruction et qu'il s'agit d'un `return`
- Une liste de paramètres vide peut être omise

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

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

Do

Préférez les lambdas aux `std::function`

Do

Préférez les lambdas à `std::bind()`

Motivations

Lisibilité, expressivité et performances
Voir practical_performance_practices.pdf

Attention

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

std::reference_wrapper

- Encapsule un objet en émulant un référence
- std::ref() et std::cref() pour construire
- Copiable

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

aref++;    // a : 11
```

Double chevron

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

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

- Possible d'utiliser des 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 ...
- ... mais 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 de créer des alias ne définissant que certains paramètres

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

Alias de template

- `using` permet de créer des alias ne définissant que certains paramètres

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

- `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 1/7

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

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

- Récupération de liste avec `...`

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

Variadic template 2/7

- 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 4/7

- Ou en utilisant l'expansion sur une expression et une fonction d'expansion prenant un *variadic template* en paramètre

```
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 5/7

Contraintes

- 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 6/7

- ... 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

- ... et travaille 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
- Et définissant son type que si l'expression booléenne est vraie
- Le type est alors égal au type fourni
- Permet de rendre un template 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
```

Suppression des export templates

- Suppression de l'export template
- `export` reste un mot-clé réservé

`export` et compatibilité

Rupture de comptabilité ascendante
Fonctionnalité implémentée sur un unique compilateur et inutilisée en pratique

Motivations

Voir N1426

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

- `std::integral_constant` type représentant une constante *compile-time*
- `true_type` : `std::integral_constant` booléen vrai
- `false_type` : `std::integral_constant` booléen faux

```
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 3/3

- `std::is_base_of` : type de 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 1/2

- Construit un nouveau type en transformant un type existant
- `std::add_const` constifie le type

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

Type traits - Transformations 2/2

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

```
enum Foo {bar};

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

- 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` 1/2

- Responsabilité exclusive
- Non copiable mais déplaçable
- Testable (conversion en booléen)

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

- `release()` pour relâcher la responsabilité de la ressource
- `reset()` pour changer la ressource possédée
- `get()` pour récupérer un pointeur brut sur la ressource

Attention

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

Pointeurs intelligents - std::unique_ptr

- Possibilité de fournir la fonction de libération

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

- Spécialisation pour les tableaux C
  - Sans les opérateurs * et ->
  - Mais avec l'opérateur []

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

Dépréciation

std::auto_ptr est déprécié au profit de std::unique_ptr

Pointeurs intelligents - `std::shared_ptr`

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

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

- `reset()` pour changer la ressource possédée
- `use_count()` retourne le nombre de possesseurs de la ressource
- `unique()` indique si la possession de la ressource est unique
- Possibilité de fournir la fonction de libération

Pointeurs intelligents - `std::make_shared()`

- Alloue et construit l'objet dans le `std::shared_ptr`

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

Objectifs

- Pas de `new` explicite, et donc plus de robustesse

```
// 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` 1/2

- 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` 2/2

- Pas d'accès à la ressource (ni `*` ni ->)
- Mais une conversion en `std::shared_ptr` via `lock()`

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

- `reset()` pour vider 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 1/3

Don't

N'utilisez pas de pointeurs bruts possédants, utilisez des pointeurs intelligents

Do

Réfléchissez à la responsabilité de vos ressources

Do

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

Pointeurs intelligents - Conclusion 2/3

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 la responsabilité des objets alloués à un pointeur intelligent le plus tôt possible

Pointeurs intelligents - Conclusion 3/3

Aller plus loin

Voir Pointeurs intelligents (Loïc Joly)

Sous silence . . .

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

Mais aussi . . .

Des réflexions et contraintes sur les 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 (p.ex. `__attribute__` ou `__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 `gsl` des « C++ Core Guidelines Checker » de Microsoft

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

Attributs 3/3

- 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

Bien souvent, également une information à destination des développeurs

Attribut [[noreturn]]

- Indique qu'une fonction ne retourne pas

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

Attention

Fonction qui ne retourne pas, et non qui ne retourne rien

Usage

Boucle infinie, sortie de l'application, exception systématique

Sous silence . . .

```
[[ carries_dependency ]]
```

Rapport 1/2

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

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

- 20 instanciations standard représentant les préfixes du système international d'unités (de yocto à yotta)

Rapport

- Méta-fonctions arithmétiques : `std::ratio_add()`, `std::ratio_substract()`, `std::ratio_multiply()` et `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 1/2

- Classe template `std::chrono::duration` représente une durée
- Unité dépendante d'un ratio (paramètre template) avec la seconde
- Six instanciations standard : *hours*, *minutes*, *seconds*, *milliseconds*, *microseconds* et *nanosecond*

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

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

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

Durées 2/2

- Opérateurs d'ajout, suppression, incrémentation, décrémentation, multiplication, ... des durées

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

- Opérateurs de comparaison entre durée
- `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'epoch

Note

Epoch est l'origine des temps de l'OS (1 janvier 1970 00h00 sur Unix)

- `time_since_epoch()` retourne la durée depuis l'epoch
- 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

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

```
system_clock::time_point today = system_clock::now();
cout << today.time_since_epoch().count() << "\n";
```

- `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);
cout << ctime(&tt) << "\n";
```

Horloges

- `std::chrono::steady_clock` : horloge monotone dédiée à la mesure des intervalles de temps
- `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

- `std::chrono::high_resolution_clock` : horloge avec le plus petit intervalle entre deux *ticks*
- Peut être un 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

- Nouveau « spécifieur de classe de stockage » `thread_local`
- Influant sur la durée de stockage
- Compatible avec `static` et `extern` pour spécifier le type de lien
- 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` 1/4

- Encapsule les types de base (booléens, nombres entiers, caractères et pointeurs) en fournissant des opérations atomiques
- Atomicité de l'affectation, de l'incrémentation 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 : 5

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` 4/4

- Plusieurs instanciations standard (p.ex. `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;

cout << foo.test_and_set() << "\n";   // 0
cout << foo.test_and_set() << "\n";   // 1
foo.clear();
cout << foo.test_and_set() << "\n";   // 0
```

Threads - `std::thread` 1/2

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

```
void foo(int);

std::thread t(foo, 10);
```

- Thread initialisé démarre immédiatement
- `joignable()` indique si le thread est joignable
  - N'est pas construit par défaut
  - N'a pas été déplacé
  - N'a pas été 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

```
// Pause de 5 secondes
this_thread::sleep_for(chrono::seconds(5));
```

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

Attention

Ne vous attendez pas à des attentes ultra-précises

Note

`sleep_for()` et `sleep_until()` sont des 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 et retourne `false` dans le cas contraire
- `unlock()` relâche le mutex

Attention

`lock()` d'un mutex verrouillé par le même thread provoque un deadlock

- `std::recursive_mutex` est une variante de `std::mutex` verrouillable plusieurs fois par un même thread

Mutex - `std::timed_mutex`

- Similaire à `std::mutex` ...
- ... mais proposant en complément des *try lock* temporisés
- `try_lock_for()` attend, si le mutex est déjà verrouillé, jusqu'au la libération de celui-ci ou l'expiration de la durée passée en paramètre
- `try_lock_until()` attend, si le mutex est déjà verrouillé, jusqu'au la libération de celui-ci ou l'atteinte du temps passé en paramètre
- `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 sur les 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` 2/2

- Plusieurs comportements lors 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 l'appel à `owns_lock()`

Note

Gestion du mutex conservée, garantie de libération

Mutex - Gestion multiple

- `std::lock()` verrouille tous les mutex passés en paramètre
- ...en ne produisant aucun 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()`

- Garantie d'un 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

- Le thread se met en attente sur la variable conditionnelle
- Et est réveillé lorsqu'un autre thread notifie cette variable
- Protection par verrou
  - Le thread prends le verrou avant d'appeler la fonction d'attente
  - ... celle-ci le relâche en attendant
  - ... et le reprend à la réception de la notification avant de débloquer le thread

Variables conditionnelles - `std::condition_variable` 1/4

- Uniquement avec `std::unique_lock`
- `wait()` mise en attente du 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` 2/4

- `wait_for()` mise en attente du thread, au maximum de la durée fournie
- `wait_until()` mise en attente du thread, au maximum jusqu'au temps fourni

Note

`wait_for()` et `wait_until()` indique en retour si l'exécution a repris suite à un timeout ou non

Variables conditionnelles - `std::condition_variable` 3/4

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

Attention

Impossible de choisir quel thread notifié avec `notify_one()`

- `std::condition_variable_any` similaire à `std::condition_variable`
- ... mais 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
  - Prends un verrou qui sera libéré à la fin du thread

Variables conditionnelles - std::condition_variable 4/4

```
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

- *Promise* contient une valeur
  - Fournie ultérieurement
  - Récupérable ultérieurement, éventuellement dans un autre thread, via un *future*
- *Future* permet de récupérer une valeur disponible ultérieurement
  - Depuis un *promise*
  - Depuis un appel asynchrone ou différé de « fonction »
- Mécanismes asynchrones
- *Futures* définissent des points de synchronisation

Note

Promise et *future* peuvent également manipuler des exceptions

Futures & promise - `std::future` 1/2

- Utilisable uniquement lorsqu'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 spécifiée
- `wait_until()` attend qu'il soit prêt, au plus jusqu'au temps spécifié
- `get()` attend qu'il soit prêt, retourne la valeur (ou lève l'exception) et libère l'état partagé

Futures & promise - `std::future` 2/2

- `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 sont copiables
  - ... ont une responsabilité partagée sur l'état partagé
  - ... la valeur peut être lue à plusieurs reprises

Futures & promise - `std::async()` 1/2

- Appelle la fonction fournie en paramètre
- 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()` 2/2

```
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` 1/3

- Objet que l'on promet de valoriser ultérieurement
- ... et dont la valeur est récupérable via un `std::future`
- Déplaçable mais non copiable
- Partage un état partagé avec le `str::future` associé
- `get_future()` retourne le `std::future` associé

Attention

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

Futures & promise - `std::promise` 2/3

- `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 3/3

```
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` 1/3

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

Attention

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

Futures & promise - `std::packaged_task` 2/3

- `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 3/3

```
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 1/2

Do

Pour l'accès concurrent aux ressources, dans l'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 2/2

Do

Analysez vos cas d'utilisation pour choisir le bon outil

Attention

Très faibles garanties de thread-safety de la part des conteneurs standard

Do

Regardez du côté de 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) 1/6

- `std::basic_regex` représente une expression rationnelle
- Deux instanciations standard `std::regex` et `std::wregex`
- Construite depuis une chaîne représentant l'expression elle-même ...
- ... et des drapeaux de configuration :
  - Grammaire utilisée : ECMAScript (par défaut), basic POSIX, extended POSIX, awk, grep, egrep
  - Case sensitive ou non
  - Prise en compte de la locale dans les plages de caractères
  - ...

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

Expressions rationnelles (regex) 2/6

- `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()` : vérification de 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) 3/6

- Possible de capturer des sous-expressions dans un `std::match_results`
- Quatre instanciations standard `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) 4/6

```
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) 5/6

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

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

Expressions rationnelles (regex) 6/6

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

Nombres aléatoires 1/2

- Des générateurs pseudo-aléatoires initialisés avec une graine (p.ex. congruence linéaire, Mersenne, ...)
- Un générateur aléatoire

Attention

Peut ne pas être présent sur certaines implémentations
Peut être un générateur pseudo-aléatoire (entropie nulle) sur d'autres

- Des distributions adaptant la séquence d'un générateur pour respecter une distribution particulière (p.ex. uniforme, normale, binomiale, de Poisson, ...)
- Une fonction de normalisation ramenant la séquence générée dans [0,1)

Nombres aléatoires 2/2

```
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()`

Nombres aléatoires 2/2

```
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

Présentation

- Approuvé le 16 août 2014
- Dernier *Working Draft* : N4140
- Dans la continuité de C++11
- Changement moins important
- Mais loin d'une simple version correctrice
- Très bon support par les versions récentes de GCC, Clang et Visual C++

constexpr

- Fonctions membres `constexpr` ne sont plus implicitement `const`
- Relâchement des contraintes sur les fonctions `constexpr`
  - Variables locales (ni `static`, ni `thread_local` et 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 1/2

- Utilisable sur les lambdas complexes

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

Généralisation de la déduction du type retour 1/2

- 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 2/2

- 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)

- Détermine le 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

- Devient compatible avec l'initialisation par défaut des membres
- ... les membres non explicitement initialisés le sont par défaut

```
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 »

Algorithmes

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

Note

Il n'est donc plus nécessaire de tester la taille auparavant

- `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

`std::random_shuffle()` est déprécié

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 grâce au préfixe « 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 standard

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

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

Note

Remplace avantageusement `std::string("abcd")` dans de nombreux contextes (p.ex. assertions cppunit)

User-defined literals standard 2/3

- Suffixe « h », « min », « s », « ms », « us » et « ns » : `std::chrono::duration` dans l'unité correspondante

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

Note

Suffixe « s » utilisé pour `std::string` et pour les secondes mais sans ambiguïté car dépendant du type de littéral auquel il s'applique

User-defined literals standard

- 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 tuples par le type

- Utilisation du type plutôt que de l'indice

```
tuple<int, long, long> foo{42, 58L, 9L};

cout << get<int>(foo);      // 42
```

Adressage des tuples par le type

- Utilisation du type plutôt que de l'indice

```
tuple<int, long, long> foo{42, 58L, 9L};

cout << get<int>(foo);      // 42
```

Attention

Uniquement s'il n'y a qu'une occurrence du type dans le tuple

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

Variable template

- Généralisation des templates aux variables
- Y compris les spécialisations

```
template<typename T>
constexpr T PI = T(3.14159265358979323385);

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

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

Sous silence ...

`std::integer_sequence`

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);
```

Mais aussi

Ajout des paramètres par défaut aux lambdas

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

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 depuis les variables locales

```
int foo = 42;

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

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

Capture généralisée 2/2

- 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'; }; }

  std::string s;};
```

Améliorations des lambdas

- Type de retour complètement facultatif
- Une lambda qui ne capture rien peut être convertie en pointeur de fonction

```
void foo(void(* bar)(int))

foo([](int x) { std::cout << x << std::endl; });
```

- Peuvent être `noexecpt`

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

- Alloue et construit 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 ]]` 1/2

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

```
[[ deprecated ]]
void bar() {}

class [[ deprecated ]] Baz { };

[[ deprecated ]]
int foo{42};
```

Attribut [[deprecated]] 2/2

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

std::shared_lock

- Capsule RAII sur les mutex partagés
- Supporte les 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

Présentation

- Approuvé en décembre 2017
- Dernier Working Draft : N4659
- Bon support par CLang 4, GCC 8 et Visual C++ 2017
- 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 pour l'instant

- 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

Mais ...

les fonctions ne levant pas d'exception peuvent être marquées `noexcept()`

Disponibilité des en-têtes : `__has_include`

- Permet de savoir si un fichier d'en-tête est présent ...
- ... et donc si une fonctionnalité est disponible

```
#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

Pas une justification aux variables globales

Nested namespace

- Nouvelle manière de définir 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 (*static-if*)

```
if constexpr(cond)
  statement1;
else if constexpr(cond)
  statement2;
else
  statement3;
```

- 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 deux branches doivent être syntaxiquement correctes mais pas nécessairement sémantiquement valides

Note

Les deux branches peuvent avoir des types retour différents sans remettre en cause la déduction de type retour

Do

Remplace avantageusement certaines constructions basées sur une suite de spécialisations de template et SFINAE, une imbrication illisible d'opérateur ternaire ou l'utilisation de `#if`

if constexpr

Le *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 1/2

- 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 2/2

- Alternative intéressante à certaines constructions peu lisibles

```
if((bool ret = foo()) == true) ...
```

- ... ou injectant un symbole inutile au delà du branchement

```
bool ret = foo();
if(ret) ...
```

- ... ou nécessitant l'introduction d'une portée supplémentaire

```
{
  bool ret = foo();
  if(ret) ...
}
```

switch init statement

- Pendant du *if init statement*
- Initialisation dans le `switch()`
- Utilisable dans le corps du `switch()`

```
switch(int foo = 42; bar) {
  case ...
}
```

structured binding 1/5

- Décompose automatiquement des types composés en de multiples variables

```
auto [liste de nom] = expression;
```

- Sur des types dont les données membres non statiques
  - Sont toutes publiques
  - Sont toutes des membres directs du type 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`
  - tableaux C

structured binding 2/5

```
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 3/5

- 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 4/5

- ...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 5/5

Objectif

Meilleure lisibilité
Remplace des usage de `std::tie()`

Nom

Appelé déstructuration (*destructuring*) dans d'autres langage

Et ensuite ?

Un premier pas vers les types algébriques de données et le *pattern matching*

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

Élision de copie garantie

- Élision de copie 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 garantie 2/2

- Élision de copie garantie lors de l'initialisation 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 intialise
```

Attention

- Uniquement sur de l'héritage public et 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 liste entre accolade
  - *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 1/2

- Possibilité d'initialiser un `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 2/2

- 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
```

Ajout de `std::byte`

- Stockage de bits
- Pas un type caractère ni « arithmétique »
- Remplace les solutions à base de `unsigned char`
- Globalement un `enum class` construit sur un `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 ne supporte 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 1/2

- Déplacement de nœuds entre conteneurs associatifs de même type
- Objet *node handle* pour le stockage et l'accès au nœud
  - Déplaçable mais non copiable
  - Permet la modification de la clé
  - Détruit le 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*
- Nouvelle surcharge de `insert()`
  - Prend en paramètre un *node handle*
  - 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 map

Déplacement de nœuds entre conteneurs associatifs 2/2

```
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 : modification et ajout

- try_emplace() : tentative de construction « 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()` et conteneurs séquentiels

- `emplace_back()` et `emplace_front()` retournent une référence sur l'élément ajouté dans un conteneur séquentiel

```
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

Nouvelle catégorie d'itérateur : `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

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

`variant` 1/3

- Union *type-safe* contenant une valeur d'un type choisi parmi n
- Issue de Boost.Variant
- Type contenu dépend de la valeur assignée
- `get<>()` récupère la valeur ...
- ... et lève une exception si le type demandé n'est pas correct
- `get_if<>()` retourne un pointeur sur la valeur ou `nullptr`

Do

Préférez `variant` aux unions brutes

Restrictions

Ne peut pas contenir des références, des tableaux, `void` ni être vide
Le premier type doit être *default-constructible* pour que le `std::variant` le soit

variant


```
variant<int, float, string> v, w;
v = "xyzzy";              // 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
```

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

Le *callable* doit être valide pour tous les types du `std::variant`

En attendant C++17 ...

Utilisez Boost.Variant

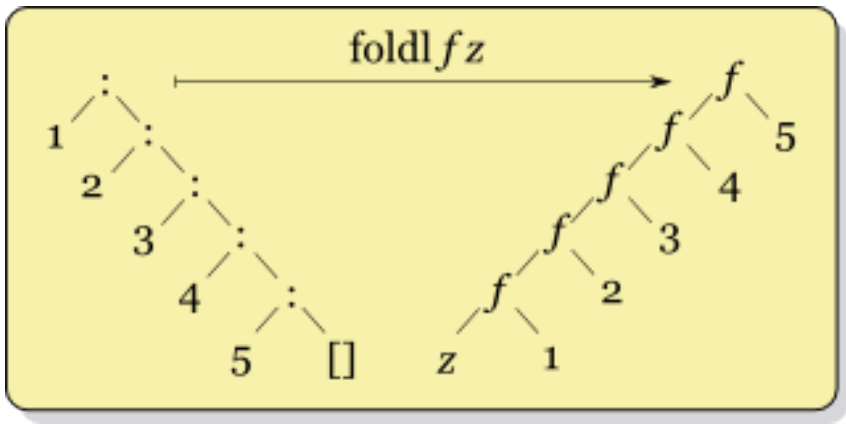
Pack expansion sur `using`

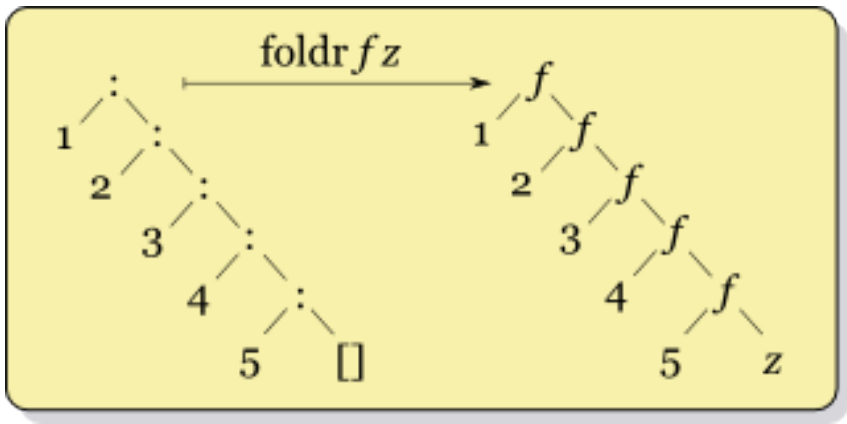
- Expansion du *parameter pack* dans les *using declaration*

```
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

- Applique un opérateur binaire à un *parameter pack*
- Support du *right fold* `(pack op ...)`
- `...` et du *left fold* : `(... op pack)`
- Éventuellement avec un valeur initiale : `(pack op ... op init)` ou `(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 5/5

- 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 de type *range-based for loop*

- Utilisation possible de types différents pour `end` et `begin`
- 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

Modifications de l'héritage de constructeur

- Constructeurs hérités visibles 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` dans le typage

- `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()

- std::uncaught_exceptions() retourne le nombre d'exceptions lancées (ou relancées) et non encore attrapées du thread courant

```
if(uncaught_exceptions())
{ ... }
```

Motivation

Obtenir un comportement différent d'un destructeur en présence d'exception (p.ex. rollback)

Caractères littéraux UTF-8

- Écriture de caractère UTF-8 préfixé par `u8`
- Lève une 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 1/3

- Déduction des paramètres templates d'une classe à la construction
- Plus de déclaration explicite des paramètres template ...
- ... 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 2/3

- 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 3/3

Note

Rend obsolète plusieurs *make helper* (`make_pair`, `make_tuple`, etc.)

Attention

Ne permet pas la déduction partielle

```
std::tuple<int> t(1, 2, 3);   // Erreur
```

template <auto>

- Déduction du type des paramètres templates numériques

```
template <auto value> void foo() { }
foo<10>();  // int
```

```
template <typename Type, Type value>
  constexpr Type FOO = value;
constexpr auto const foo = FOO<int, 100>;

// Devient

template <auto value> constexpr auto FOO = value;
constexpr auto const foo = FOO<100>;
```

Template & contraintes d'utilisation 1/2

- 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 2/2

- 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
- Utilisation de `*this` dans la spécification de capture

```
[*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 1/3

- Lambdas autorisées dans les expressions constantes ...
- ...si l'initialisation de chaque donnée capturée est possible dans l'expression constante

```
constexpr int AddEleven(int n) {
  return [n] { return n+11; }();  }

AddEleven(5);    // 16
```

Lambdas et expressions constantes 2/3

- Déclaration `constexpr` d'une lambda possible
- Définit explicitement un appel `constexpr` ...

```
auto ID = [] (int n) constexpr { return n; };
constexpr int I = ID(3);
```

- ... appel implicitement `constexpr` lorsque les exigences sont satisfaites

```
auto ID = [] (int n) { return n; };
constexpr int I = ID(3);
```

Lambdas et expressions constantes 3/3

- 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() 1/2

- Appelle le *callable* fourni en paramètre
- ...en fournissant la liste de paramètres
- ...et en retournant le retour du *callable*

```
int foo(int i) {
  return i + 42;}

cout << invoke(&foo, 8);  // 50
```

std::invoke() 2/2

- 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;
cout << invoke(&Foo::bar, foo, 8);  // 50
```

Motivation

Une syntaxe unique d'appel de *callable*

std::not_fn()

- Construit un *function object* en niant un appelable

```
bool LessThan10(int a)  {
  return a < 10; }

vector foo = { 1, 6, 3, 8, 14, 42, 2 };
auto n = count_if(begin(foo), end(foo), not_fn(LessThan10));
cout << n << '\n'; // 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 appelable
  - `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>
std::enable_if_t<std::conjunction_v<std::is_same<T, Ts
    >...>>
foo(T, Ts...) { }
```

Gestion des attributs 1/2

- Usage étendu aux déclarations de `namespace`

```
namespace [[ Attribut ]] foo {}
```

- . . . Et aux valeurs d'une énumération (énumérateurs)

```
enum foo {
  FOO_1 [[ Attribut ]],
  FOO_2 };
```

Gestion des attributs 2/2

- Attributs inconnus sont ignorés
- `Using` des attributs non standard

```
[[ nsp::kernel, nsp::target(cpu,gpu) ]]
foo();

// Devient

[[ using nsp: kernel, target(cpu,gpu) ]]
foo();
```

Attribut [[fallthrough]]

- Placé dans un `switch` avant un `case` ou `default`
- Indique qu'un cas se poursuit intentionnellement dans le cas suivant
- Incitation à ne pas lever de warning dans ce cas

```
switch(foo) {
  case 1:
  case 2:
    ...
[[ fallthrough ]];
  case 3:   // Idealement : pas de warning
    ...
  case 4:   // Idealement : warning
    ...
    break; }
```

Attribut [[nodiscard]] 1/2

- Indique que le retour d'une fonction ne devrait pas être ignorée

```
[[ nodiscard ]] int foo() {return 5;}

foo();  // Idealement : warning
```

- Incitation à lever un warning dans le cas contraire

Note

Conversion implicite en `void` pour supprimer le warning

```
(void)foo();
```

Attribut [[nodiscard]] 2/2

- 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();  // Idealement : warning
```

Attribut `[[ maybe_unused ]]` 1/2

- Utilisable sur une classe, structure, fonction, variable, paramètre, ...
- Indique qu'un élément peut ne pas être utilisé
- Incitation à ne pas lever de warning en cas de non-utilisation

```
[[ maybe_unused ]]
int foo([[ maybe_unused ]] int a,
        [[ maybe_unused ]] long b) {}
```

- Ne devrait pas lever de warning en cas d'utilisation

Attribut [[maybe_unused]] 2/2

Avant C++17

La méthode « classique » pour supprimer le warning sur la non-utilisation de paramètres consiste à ne pas les nommer

```
int foo(int, long) {}
```

Attributs C++17 - Conclusion

Do

Utilisez les attributs pour indiquer vos intentions

Au delà du compilateur

Prise en compte par d'autres outils (générateurs de documentation, analyseurs statique de code) 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

Attention

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

Note

Équivalent non « timed » de std::shared_timed_mutex

std::scoped_lock

- std::scoped_lock peut acquérir plusieurs mutex

```
mutex first_mutex;
mutex second_mutex;

scoped_lock lck(first_mutex, second_mutex);
```

std::apply()

- Appel de fonction depuis un tuple 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);
```

- De même, std::make_from_tuple() permet de construire un objet depuis un *tuple-like*

std::optional

- Gestion d'objet dont la présence est optionnelle
- Issue de Boost.optional
- Interface similaire à un pointeur
  - Testable via `operator bool()`
  - Accès à l'objet via `operator*()`
  - Accès au membre de l'objet via `operator->()`

Attention

L'appel de `operator*()` ou `operator->()` sur un `std::optional` vide est indéfini

  - `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);

cout << (dix > deux) << '\n';    // true
cout << (dix < deux) << '\n';    // false
cout << (dix == 10) << '\n';     // true
```

- ...En prenant en compte std::nullopt

```
optional<int> none, dix(10);

cout << (dix > none) << '\n';          // true
cout << (dix < none) << '\n';          // false
cout << (none == 10) << '\n';          // false
cout << (none == nullopt) << '\n';     // true
```

std::optional

std::optional<bool> et std::optional<T*> pertinents ?

Probablement plus pertinent d'utiliser :

- Des booléens « trois états » (Boost.tribool)
- Des pointeurs bruts

Do

Préférez `optional` aux pointeurs bruts pour gérer des données optionnelles

En attendant C++17 ...

Utilisez Boost.Optional

std::any

- `void*` *type-safe* contenant un objet de n'importe quel type (ou vide)
- Introduction d'une forme de typage dynamique au sein de C++
- Issue de Boost.Any
- 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);
```

- Existence du *helper* std::make_any

```
any a = make_any<complex<double>>(3.0, 4.0);
```

- Changement de valeur (et 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;
cout << any_cast<int>(a) << '\n';  // 1
cout << any_cast<bool>(a) << '\n'; // 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

- std::basic_string_view référence une séquence contiguë de caractères
- Quatre spécialisations standard (une pour chaque type de caractère)
- Référence non possédante sur une séquence pré-existante
- Pas de modification de la séquence depuis la vue

Attention !

- Pas de \0 terminal systématique
- La chaîne référencée doit vivre au moins aussi longtemps que la vue

`std::string_view` 2/3

- 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

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

Présence dans le TS d'un pointeur intelligent sans responsabilité (observateur) : `observer_ptr`, mais n'est 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
- Issue de Boost.Filesystem
- 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 2/3

- Manipulation des répertoires, des fichiers et de leurs métadatas
  - 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 3/3

- 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 (ou une autre bibliothèque tierce équivalente)

Parallelism TS

- Ajout de surcharges « parallèles » à de nombreux algorithmes standard
- 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

Accès concurrents non gérés intrinsèquement par l'exécution parallèle
Responsabilité du développeur de choisir des structures de données et des foncteurs adressant ce point

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

Différence entre `std::reduce()` et `std::accumulate()` ?

L'ordre des « additions » n'est pas 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

La transformation n'est pas 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
  - ...

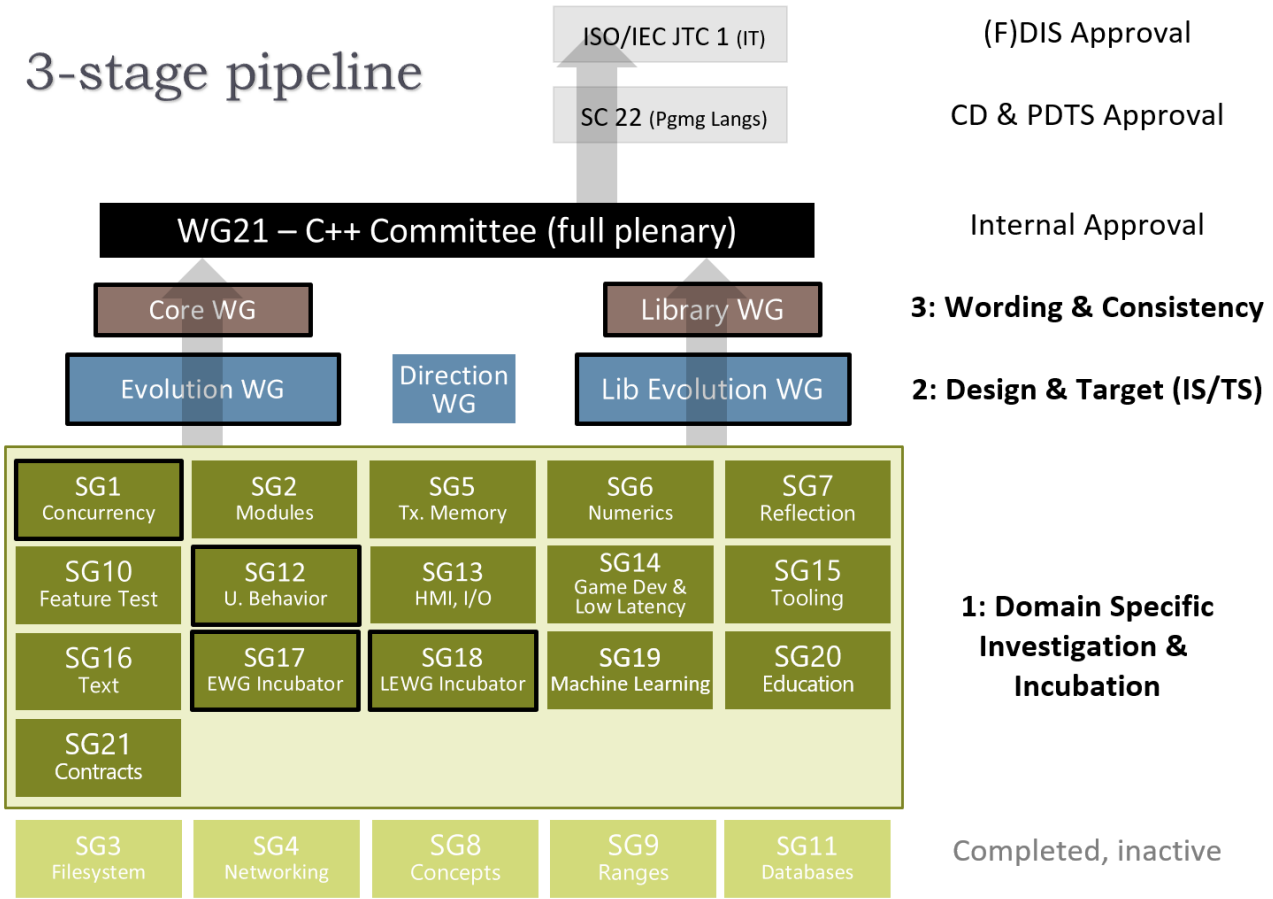
Sommaire

Présentation

- Travaux lancés en juillet 2017
- Périmètre figé en juillet 2019
- Revue du *Commitee Draft* en cours par les comités nationaux
- Version finale prévue pour février 2020
- Dernier Working Draft : N4830

Changements d'organisation du comité

- Création d'un *Direction Group*
- Création d'un Study Group pour l'éducation (SG20) : aide à l'apprentissage et à l'adoption des évolutions



Dépréciations et suppressions

- Dépréciation du terme POD et de `std::is_pod()`
- Dépréciation partielle de `volatile`
- Suppression des membres dépréciés de `std::reference_wrapper` : `result_type`, `argument_type`, `first_argument_type` et `second_argument_type`

Fonctionnalités

- `__has_cpp_attribute` permet de tester 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 au mois de l'acceptation dans le standard ou de l'évolution

Information à la compilation

- Entête `<version>` : informations de version
  - Contenu *implementation-dependent*
  - Typiquement : version du standard, version 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 actuelle

- 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 défini
  - Décalage à gauche : même résultat que celui du type non signé correspondant
  - Décalage à droite : décalage arithmétique avec extension du signe

Caractères

- Type `char8_t` pour les caractères
  - 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
class S {
  S() = default; };
```

Initialisation des agrégats 1/2

- 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, pas sur toutes les classes
- Initialisation des champs dans leur ordre de déclaration
- Initialisation d'un unique membre d'une union

Initialisation des agrégats 2/2

- Initialisation des agrégats via des données parenthésées

Différences entre `{}` et `()`

- `{}` permet l'utilisation d'*initializer list*
- `()` permet les conversions avec perte de précision

Motivations

Utilisation des fonctions transférant les arguments à un constructeur sur des agrégats

- Initialisation par défaut des champs de bits

endianess

- Définition d'une énumération `std::endian`
  - `little` : *little-endian*
  - `big` : *big-endian*
  - `native` : l'*endianess* du système

`operator<=>()` 1/2

- 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
- Cinq types de retour possibles :
  - `std::strong_ordering` : ordre et égalité
  - `std::strong_equality` : égalité
  - `std::weak_ordering` : ordre et équivalence
  - `std::weak_equality` : équivalence
  - `std::partial_ordering` : ordre partiel
- Peut être généré par le compilateur (`=default`)
  - `operator<=>()` des bases et membres
  - `operator==()` et `operator>()`

`operator<=>()` 2/2

- `operator<=>()` déclenche la génération par le compilateur des autres opérateurs de comparaison en fonction du type de retour
  - Opérateurs d'ordre (<, <=, > et >=) via `operator<=>()`
  - `operator==()` via `operator==()` des bases et membres
  - `operator!=()` via `operator==()`

`==, !=, <=>`

`operator==()` et `operator!=()` ne sont pas générés à partir de `operator<=>()`

- Possible de marquer ces autres opérateurs `=default`
- Utilisation de l'opérateur binaire déclaré s'il existe
- Supporté par la bibliothèque standard

Fun fact

Cet opérateur est surnommé « *spaceshift operator* »

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

Et les `namespace` ?

Ne replace pas les `namespace`

- Réduction des temps de compilation
- Nouveau niveau d'encapsulation
- Plus grande robustesse (isolation des effets des macros)
- Meilleurs 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

Une *Implementation Unit* a accès aux déclarations non exportées du module

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

Une *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 1/2

- 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

Chaque module doit contenir un et un seul *Interface Unit* sans nom de partition

- Un élément peut être déclaré dans une partition et défini dans une autre

Modules - Partitions 2/2

- 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 nom de *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
```

- Les éléments d'une partie exportée d'un *namespace* sont exportés

```
// C::m est exporte mais pas C::n
namespace C { int n; }

export namespace C { int m; }
```

Modules - Implémentation *inline*

- Interface et implémentation dans un unique fichier
- En séparant les deux parties

```
export module m;
struct s;
export using s_ptr = s*;

module : private;
struct s {};
```

Modules - Implémentation *inline*

- Interface et implémentation dans un unique fichier
- En séparant les deux parties

```
export module m;
struct s;
export using s_ptr = s*;

module : private;
struct s {};
```

Restriction

Uniquement dans une *Primary Module Interface Unit* qui devrait être la seule unité du module

Modules - Utilisation

- Import des modules via la directive `import`

```
import foo;

// Utilisation des symboles exportes de foo
```

- Cohabitation possible avec des inclusions

```
#include <vector>
import foo;
#include "bar.h"
```

Modules - Code non-modulaire 1/2

- 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 2/2

- 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 1/2

- `std::basic_string::reserve()` ne peut plus réduire la capacité
  - L'appel avec une capacité inférieure n'a pas d'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
```

Conteneurs associatifs

- `contains()` teste la présence d'une clé

```
map<int, string> foo{{1, "foo"}, {42, "bar"}};

cout << foo.contains(42) << "\n";  // true
cout << foo.contains(38) << "\n";  // false
```

Conteneurs associatifs 2/2

- Optimisation de la recherche hétérogène dans des conteneurs ordonnés
  - Fourniture d'une classe exposant
    - Les différents foncteurs de calcul du hash
    - Le tag `transparent_key_equal`
  - Suppression des conversions de type 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> map = ...;
map.find("abc");
map.find("def"sv);
```

std::list et forward_list

- remove(), remove_if() et unique() retourne le nombre d'éléments supprimés

Suppression d'éléments

- `erase()` supprime les éléments égaux à la valeur fournie
- `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
```

```
map<int, int> bar{{5, 1}, {12, 2}, {2, 3}, {42, 4}};
erase_if(bar, [](pair<int, int> i) {return i.first >
    20;});
// 2-3 5-1 12-2
```

- Remplacent l'idiome « *Erase-remove* » et l'utilisation de la fonction membre `erase()`

std::span

- Fournit une vue sur un ensemble contigu
- Similaire à std::string_view
- Constructible depuis un ensemble, début/taille, début/fin ou 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 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 };

cout << bar.front() << "\n";     // 0
```

- `first()`, `last()` : construction de *sous-span*

```
array<int, 5> foo = {0, 1, 2, 3, 4};
span<int> bar{ foo.data(), 4 };

span<int> baz = bar.first(2);    // 0, 1
```

- *structured binding* sur des *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

Si le décalage est plus grand que la taille de l'ensemble, l'opération est sans effet

```
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::ispow2()` teste si un entier est une puissance de deux
- `std::ceil2()` plus petite puissance de deux non strictement inférieure
- `std::floor2()` plus grande puissance de deux non strictement supérieure
- `std::log2p1()` plus petit nombre de bits nécessaire pour représenter un entier

```
ispow2(4u);    // true
ispow2(7u);    // false
ceil2(7u);     // 8
ceil2(8u);     // 8
floor2(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*

Mathématiques

- `std::lerp()` interpolation linéaire entre deux valeurs flottantes
- `std::midpoint()` : demi-somme de deux valeurs (entières ou flottantes)

Règle d'arrondi

La demi-somme d'entiers est entière et arrondie, si nécessaire, vers le premier paramètre

```
midpoint(2, 4);      // 3
midpoint(2, 5);      // 3
midpoint(5, 2);      // 4
```

- Définition de 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

Évolutions de la bibliothèque standard

- Utilisation de l'attribut `[[nodiscard]]`
- Utilisation de `noexcept`
- Optimisation d'algorithmes numériques via `std::move()`

Ranges - Présentation

- Abstraction de plus haut niveau que les itérateurs
- Manipulation d'ensemble d'éléments au travers d'algorithmes et de *range adaptators*
- Vivent dans le `namespace std::ranges`

Pour aller plus loin

- « Iterators Must Go » d'Andrei Alexandrescu
- Le blog d'Eric Niebler

Ranges - Itérateurs

- `std::common_iterator` : adaptateur d'itérateur/sentinelle représentant un range itérateur/sentinelle de types différents en un range de types similaires
- `std::counted_iterator` : adaptateur d'itérateur avec un fonctionnement similaire à l'itérateur sous-jacent mais conservant la distance à la fin du range

Ranges - Concepts 1/2

- `Range`
  - Un itérateur de début
  - Une sentinelle de fin
    - Une valeur particulière
    - Un autre itérateur
    - Le type vide `std::default_sentinel_t` marquant la fin d'un range et utilisable avec des itérateurs gérant la limite du range
- `SizedRange` : *range* fournissant sa taille en temps constant
- `View` : *range* fournissant copie, déplacement et affectation en temps constant
- `ViewableRange` : *range* convertible en `View`
- `CommonRange` : *range* dont itérateurs et sentinelle ont le même type

Ranges - Concepts

- `InputRange` : *range* fournissant des `input_iterator`
- `OutputRange` : *range* fournissant des `output_iterator`
- `ForwardRange` : *range* fournissant `forward_iterator`
- `BidirectionalRange` : *range* fournissant `bidirectional_iterator`
- `RandomAccessRange` : *range* fournissant `random_access_iterator`
- `ContiguousRange` : *range* fournissant `contiguous_iterator`

En résumé

- Conteneurs : possession, copie profonde
- Vues : référence, copie superficielle

Ranges - Opérations

- `begin()`, `end()`, `cbegin()`, `cend()`, `...` : récupération des itérateurs
- `size()` : récupération de la taille
- `empty()` : teste la vacuité
- `data()` et `cdata()` : récupération de l'adresse de début de la plage

Restrictions

`data()` et `cdata()` sur des *contiguous range* uniquement

- Surcharges des différents algorithmes pour prendre des *ranges* 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})
  std::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 : std::views::counted(a, 3))
  std::cout << i << ' ';
  // 1 2 3
```

Ranges - *range adaptators* 1/6

- Appliquent filtres et transformations aux *ranges*
- Associés, pour certains, à un *range adaptor closure object*
  - Prends un unique paramètre `viewable_range`
  - Retourne une `view`

Ranges - *range adaptators* 2/6

- Peuvent être chaînés avec une syntaxe « appel de fonction » ...

```
D(C(R));
```

- ...Ou une syntaxe « pipeline »

```
R | C | D;
```

- Évaluation paresseuse des pipelines
- Peuvent prendre plusieurs arguments

```
adaptor(range, args...);
adaptor(args...)(range);
range | adaptor(args...);
```

Ranges - *range adaptators* 3/6

- Plusieurs *adaptors* 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;},

//0, 2, 4, 6, 8, 10
auto rng = ints | view::transform(double);
```

Ranges - *range adaptators* 4/6

- Plusieurs *adaptors* fournis par la bibliothèque standard :
  - `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* 5/6

- Plusieurs *adaptors* fournis par la bibliothèque standard :
  - `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* 6/6

- Plusieurs *adaptors* fournis par la bibliothèque standard :
  - `elements_view` : la 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` : la vue des clés de chaque `std::pair` d'une vue de `std::pair`
  - `values_view` : la vue des valeurs de chaque `std::pair` d'une vue de `std::pair`
- Possible d'utiliser les algorithmes opérants sur les *ranges*

Ranges - Exemples

```
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 |
            view::filter(even) |
            view::transform(square))
  cout << i << ' '; // 0 4 16
```

```
vector<int> foo{1, 1, 2, 2, 3, 3, 5, 5, 5, 6, 9}, bar;

auto odd = [](int i){ return (i % 2) == 1; };

copy(foo | view::filter(odd) | unique{},
     back_inserter(bar)); // 1 3 5 9
```

Gestion des flux

- Flux synchrones
  - Classe tampon synchrone : `std::basic_syncbuf`
  - Classe flux bufferisé synchrone : `std::basic_osyncstream`
  - `emit()` transfère le buffer vers le flux de sortie

```
{ osyncstream 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`

- API de formatage inspiré de la bibliothèque {fmt}

Motivations

- Le formatage « à la C » ne supporte pas les types utilisateurs et est peu sûr
- Les flux sont complexes et peu propices à l'internationalisation et la localisation

- `std::format()` et `std::vformat()` retournent une chaîne de caractères
- `std::format_to()` et `std::vformat_to()` écrivent dans un flux
- Prise en compte de *locale*
- Format sous forme de chaînes utilisant {} comme *placeholder*

En attendant C++20

Utilisez {fmt}, Boost.Format ou une bibliothèque tierce équivalente

std::format

- Deux types d'indexation :
  - Automatique

```
format("{} et {}", "a", "b"); // "a et b"
```

  - Manuelle

```
format("{1} et {0}", "a", "b"); // "b et a"
format("{0} et {0}", "a");      // "a et a"
```

- Un ensemble de *formatters* standard :
  - Alignement

```
format("{:6}", 42);     // "    42"
format("{:6}", 'x');    // "x     "
format("{:*<6}", 'x'); // "x*****"
format("{:*>6}", 'x'); // "*****x"
format("{:*^6}", 'x'); // "**x***"
```

std::format

- Un ensemble de *formatters* standard :
  - Présence du signe pour les numériques

```
format("{0:},{0:+},{0:-},{0: }", 1); // "1,+1,1, 1"
format("{0:},{0:+},{0:-},{0: }", -1);// "-1,-1,-1,-1"
```

  - Format des numériques

```
format("{:+06d}", 120);       // "+00120"
format("{:#06x}", 0xa);       // "0x000a"
// "101010 42 52 2a"
format("{0:b} {0:d} {0:o} {0:x}", 42);
format("{0:#x} {0:#X}", 42); // "0x2a 0X2A"
```

- Possibilité de créer ses propres *formatters*

Gestion mémoire

- Support des tableaux par `std::make_shared()`

```
shared_ptr<double[]> foo = make_shared<double[]>(1024);
```

- Déduction de la taille des tableaux par `new()`

```
double* a = new double[]{1, 2, 3};
```

Nouvelles horloges

- Ajout de nouvelles horloges
  - `std::chrono::utc_clock` : temps universel coordonné
  - `std::chrono::gps_clock`
  - `std::chrono::tai_clock` : temps atomique universel
  - `std::chrono::file_clock` : alias vers le temps du système de fichier
- 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`

Le support des conversions entre `std::chrono::file_clock` et `std::chrono::utc_clock` ou `std::chrono::system_clock` est optionnel

- Pseudo-horloge `std::chrono::local_t` temps dans la *timezone* locale

Évolution de `std::chrono::duration`

- Ajout de *helper* pour le jour, la semaine, le mois ou l'année
- Ajout de `to_stream()` pour afficher une `std::chrono::duration`
- Ajout de `from_stream()` pour lire une `std::chrono::duration`
- Utilisation de chaîne de format utilisant des séquences préfixées par `%`
  - `%H` et `%I` : l'heure (au format 24h ou 12h)
  - `%M` : les minutes
  - `%S` : les secondes
  - `%Y` et `%y` : l'année (4 ou 2 chiffres)
  - `%m` : le numéro du mois
  - `%b` et `%B` : le nom du mois dans la locale (abrégé ou complet)
  - `%d` : le numéro du jour dans le mois
  - `%U` : le numéro de la semaine
  - `%Z` : l'abbréviation 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 les différentes combinaisons permettant de construire une date complète
  - Constantes représentant les jours de la semaine et les mois
  - Suffixes littéraux `y` et `d` marquant 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 1/2

- 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 ambigüité 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};
```

Timezone 2/2

En attendant C++20

Utilisez Boost.Date_Time (ou une bibliothèque tierce équivalente)

Pour aller plus loin

ICU supporte de nombreux calendriers et mécanismes de localisation

Évolutions des *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; }
```

- Cohérence entre *begin* et *end* :
  - « Début » et « début + taille »
  - fonctions membres `begin()` et `end()`
  - fonctions libres `std::begin()` et `std::end()`

consteval

- Spécificateur `consteval` : impose une évaluation *compile-time*
  - `consteval` implique `inline`

```
consteval 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 `consteval`

`constinit`

- Spécificateur `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*

Évolutions de `constexpr` 1/2

- Initialisation triviale dans des contextes `constexpr`
- `std::is_constant_evaluated()` pour savoir si l'évaluation est *compile-time*
- Prise en compte étendue de `constexpr` dans la bibliothèque standard

Évolutions de `constexpr` 2/2

- Assouplissement des restrictions de `constexpr`
  - Utilisation d'`union` dans du code `constexpr`
  - Utilisation de `try {} catch()` dans du code `constexpr`
    - Comporte comme *no-ops* en *compile-time*
    - Ne peut pas lancer d'exception *compile-time*
  - Utilisation de `dynamic_cast` et `typeid` dans du code `constexpr`
  - Déclaration de fonctions virtuelles `constexpr`
  - Utilisation de `asm`

Évolutions des *structured binding* 1/2

- Extension à tous les membres visibles (et plus uniquement publics)
- 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; };
```

  - Possibilité de les déclarer `inline`, `extern`, `static`, `thread_local` ou `constexpr`
  - Possibilité de les marquer `[[maybe_unused]]`

Évolutions des *structured binding* 2/2

- 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<std::chrono::seconds seconds>
class fixed_timer { /* ... */ };
```

```
template<fixed_string Id>
class entity { /* ... */ };

entity<"hello"> e;
```

Évolutions des 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, std::type_identity_t<T>);

g(4.2, 0); // OK, g<double>
```

Paramètres `auto`

- Création de fonctions templates via l'usage d'`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
- Définir des contraintes sur les paramètres templates et l'inférence de type
  - Diagnostics plus clair
  - Meilleur documentation du code
  - Aide à la déduction de type
  - Aide à la résolution de spécialisation
- Propositions visiblement abandonnées
  - *Axiom* : spécification de propriétés sémantiques d'un concept
  - *Concept map* : transformation entre un concept et un type ne le satisfaisant pas

Concepts - Utilisation template 1/4

- Utilisable via la liste des paramètres template

```
template<Decrementable T>
void foo(T);
```

- . . . Ou via une clause `requires`

```
template<typename T> requires Incrementable<T>;
void foo(T)
```

- . . . Ou les deux

```
template<Decrementable T>
void foo(T) requires Incrementable<T>;
```

Concepts - Utilisation template

- Utilisable depuis un concept nommé

```
// On suppose le concept Addable existant
template<typename T> requires Addable<T>
T add(T a, T b) { return a + b; }
```

- . . . 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 3/4

- 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<typename... T>
requires Concept<T> && ... && true
void foo(T...);
```

```
template<Concept... T>
void foo(T...);
```

Concepts - Utilisation inférence de type 1/2

- 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 2/2

- 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

- De nombreux concepts définis dans la bibliothèque standard
  - 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(std::forward<T>(t), std::forward<U>(u));
  swap(std::forward<U>(u), std::forward<T>(t)); };
```

Concepts - Définition

- Y compris sans qualificateur

```
template<class T> concept Addable = requires(
  const remove_reference_t<T>& a,
  const remove_reference_t<T>& b) { a + b; };
```

- Ou sur les types de retour

```
template<class T> concept Comparable = requires(
{ 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 constructible_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; };
```

Évolutions des Attributs

- Ajout de nouveaux attributs
  - `[[likely]]` et `[[unlikely]]` : probabilité de branches conditionnelles

Avec parcimonie

Les compilateurs savent déjà déterminer les branches les plus probables, généralement mieux que nous

  - `[[no_unique_address]]` : l'adresse d'un membre peut être partagée
- Extension de `[[nodiscard]]` aux constructeurs
  - marquage `[[nodiscard]]` des constructeurs est autorisé
  - Vérification lors des conversions via les constructeurs
- Possibilité d'associer un message à `[[nodiscard]]`

Évolutions des lambdas 1/2

- Utilisables dans des environnements non évalués
- Utilisation de paramètres templates pour les lambdas génériques

```
auto foo = []<typename T>(vector<T> bar) { ... };
```

  - 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<< std::size(vec) << '\n';
  cout<< vec.capacity() << '\n'; };
```

Évolutions des lambdas 2/2

- 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
- 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...);};}
```

Binding

- `std::bind_front()` attache 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 bar = bind(&foo, 2, 3, 4, _1);
bar(6);  // 30

auto baz = bind_front(&foo, 2, 3, 4);
baz(7);  // 31
```

- `std::reference_wrapper` accepte les types incomplets

std::atomic

- Ajout de std::atomic<std::shared_ptr<T>>
- Ajout de std::atomic<> sur les types flottant
- 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`
  - Peut être arrêté par l'appel à `request_stop()`
  - Automatiquement arrêté et joint lors de la destruction

synchronisation - sémaphores

- `std::counting_semaphore`
  - Création avec la valeur maximale 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_until()` tente de prendre le sémaphore en attendant 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

Il n'est pas possible d'incrémenter un `std::latch` ni 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()` attends que le compteur atteigne zéro
  - `arrive_and_wait()` décrémente le compteur et attends 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 reprends la valeur initiale décrémentée du nombre de *threads* « *droppés* »

Politique d'exécution

- Ajout d'une 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

- Fonction contenant
  - `co_await` suspend l'exécution
  - `co_yield` suspend l'exécution en retournant une valeur
  - `co_return` termine la fonction
- Des 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()` : le *handle* gère effectivement une coroutine
  - `done()` : 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()

- std::create_directory() échoue si l'élément terminal existe et n'est pas un répertoire

```
create_directory("a/b/c");
// Erreur en C++17 si a ou b existe mais ne sont pas des
    repertoires
// Pas d'erreur en C++17 si c existe mais n'est pas un
    repertoire

// Erreur en C++20 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'expliciter le type de retour de `std::visit()`
  - Via un paramètre template
  - Sinon déduit de l'application du visiteur au premier paramètre

Sommaire

Présentation

- C++20 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

Propositions diverses

- Surcharge de l'opérateur point (`operator.()`)
  - Si l'opérateur est défini, les opérations sont transférés à son résultat
  - . . . sauf celles spécifiquement déclarées membres
  - Réalisation de « *smart reference* » (p.ex. proxy)
- *Unified Call Syntax*
  - `f(x,y)` appelle `x.f(y)` si `f(x,y)` n'est pas trouvé
  - Généralisation de `std::begin()` et co. directement dans le langage
- `std::expected` contenant un statut et une valeur optionnelle : retour d'un compte rendu d'exécution de la fonction et, éventuellement, d'une valeur
- *Procedural function interfaces* : framework pour la vérification statique de partie du programme
- Support de l'Unicode
- Support des entrées/sorties audio
- Dépréciation de l'usage de l'opérateur virgule dans les expressions d'indiçage

Propositions diverses

- Ajout de nouveaux « conteneurs » :
  - Adaptateurs `std::flat_map` et `std::flat_multimap` : map depuis une paire de conteneurs séquentiels
  - `std::mdspan` : vues multidimensionnels (et indiçage associé `[x,y,z]`)
- Nouveaux pointeurs intelligents :
  - `out_ptr` : manipulation de `T**` en paramètres de retour des API C
  - `retain_ptr` : pointeur intrusif manipulant le comptage de référence interne d'un objet
- Création de pointeurs intelligents avec une valeur par défaut
- 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 type tableau
  - Comparaison « *three-way* » entre *unscoped* énumération et type entier

Propositions diverses

- *Pattern matching* via `inspect` :

```
inspect (p) {      // std::pair
  [0, 0]: cout << "on origin";
  [0, y]: cout << "on y-axis";
  [x, 0]: cout << "on x-axis";
  [x, y]: cout << x << ',' << y;}

inspect (shape) { // Sous-type
  (as<Circle> ? [r]): cout << 3.14 * r * r;
  (as<Rectangle> ? [w, h]): cout << w * h;}
```

Mais aussi

Sur les types entiers, les chaînes, les *tuples*, les `std::variant`, ...

Propositions diverses

- Ajout de `volatile_load<T>` et `volatile_store<T>`
- *Compile Time Regular Expression*
- `std::embed()` : rendre disponible au *runtime* des ressources externes
- Gestion des UUID
- Amélioration de la déduction template dans les constructeurs : agrégats, alias et constructeurs hérités
- Implémentations *freestanding* : intégration du plus grand sous-ensemble possible de la bibliothèque standard qui ne présente pas de *memory overhead* ni ne nécessite de support de l'OS
- Contrôle du *layout* des classes

Propositions diverses

- Méta-classes pour construire des types de classes (dont les classes elles-mêmes) ayant des contraintes, des comportements par défaut et des opérations par défaut (`class`, `struct`, `enum class`, *interface*, *value*)
- Répétition compile-time d'une expression : *Expansion statement*

```
auto foo = make_tuple(0, 'a', 3.14);
for... (auto elem : tup)
  cout << elem << "\n"
```

  - Pas une boucle : duplication de l'expression pour chaque élément
  - Éléments de type différent
  - Utilisable sur `std::tuple`, `std::array`, classes destructurables, ...
- Exceptions légères
- Ajout *floating-point types* de plus petite taille

TS - Contracts

- Retiré du draft C++20 et création d'un groupe d'étude en juillet 2019
- Support de la programmation par contrat
- Remplacement de la vérification à coup d'assert et de la documentation via commentaire @pre, @post et @invariant
- Initialement, plusieurs propositions « concurrentes »
- . . . mais un compromis à émerger
- Utilisation d'attributs `[[assert:x]]`,`[[expects:x]]` et `[[ensure:x]]`
- Possibilité de les marquer `audit` pour ne les activer qu'à la demande
- Possibilité de les marquer `axiom` pour ne pas générer de code *runtime* (*compile-time* uniquement)
- Les contrats de fonctions membres publiques peuvent utiliser des membres privés ou protégés
- Intégration des contrats à la bibliothèque standard

TS - Networking TS

- Publié en avril 2018
- Très probablement intégré à C++23
- Partiellement basé sur Boost.Asio
- Modèle asynchrone
- Gestion de timer
- Gestion de buffer et de flux orientés buffer
- Gestion de sockets et de flux « socket »
- Gestion d'IPv4, IPv6, TCP, UDP
- Manipulation d'adresses IP
- Pas de protocoles de plus haut niveau actuellement

TS - Les autres

- *Library fundamentals 2* : évolutions de la bibliothèque standard
  - Pointeurs intelligents non possédant
  - Nouveaux algorithmes
- *Library fundamentals 3* :
  - *Generic Scope Guard*
  - *RAII wrapper*
- *Parallelism* 2 : publié en juin 2018
- *Transactional Memory* : publié
- *Numerics*
- *Array extension* : taille non connue à la compilation
- *2D Graphics* (io2d) : API C++ au dessus de Cairo, différé
- *Reflection* : *feature-complete*, TS en « C++20 », probablement en C++ 23

TS - Les autres

- *Concurrency* 1 : publié
  - `future.then()`
- *Concurrency* 2
  - *Synchronic* : meilleure abstraction pour atomique permettant de tirer partie des caractéristiques logicielles et matérielles de la plateforme
  - *Executor* permettant de spécifier où s'exécute telle tâche (*a priori* pour C++23)

Sommaire

Présentation

- Ensemble de bibliothèques
- Domaines très variés
- Antichambre de la bibliothèque standard
- Licence permissive (proche MIT ou BSD)
- Portable
- Haut niveau d'exigence (y compris documentaire)
- Processus de revue strict et transparent
- Conférence annuelle (BoostCon puis C++now)
- Une version tout les 4 à 5 mois
- Compatibilité ascendante non garantie

www.boost.org

Boost.Optional, Boost.Any et Boost.Variant

- Intégrées à C++17
- Fonctionnement identique entre Boost et C++17
- ... à deux ou trois détails syntaxiques prés
  - Changement de namespace
  - `std::nullopt` devient `boost::none`
  - `polymorphic_relaxed_get()` devient `std::get_if()`

Do

Utilisez Boost.Optional, Boost.Any et Boost.Variant si votre bibliothèque standard n'est pas C++17

Boost.Filesystem

- Intégrée à C++17
- Fonctionnement identique entre Boost et C++17
- Ajout de `boost::filesystem::fstream` compatible avec `std::fstream` pour ouvrir un fichier depuis un *path*

```
path p{"test.txt"};
ofstream ofs{p};
ofs << "Hello, world!\n";
```

Boost.DateTime

- Intégrée, en grande partie, à C++11 et C++20
- Manipulation de dates et heures
- Gestion des temps POSIX (`ptime`)
- Gestion des *timezones* et des temps locaux
- Gestion du calendrier grégorien (`date`)
- Gestion des durées (`time_duration`)

Boost.DateTime

- Récupération de la date et heure courante dans la timezone locale

```
ptime date = second_clock::local_time();
```

- ... ou en UTC

```
ptime date = second_clock::universal_time();
```

Boost.DateTime

- Construction depuis une chaîne

```
std::string ts("2002-01-20 23:59:59.000");
ptime t(time_from_string(ts))

std::string ts("20020131T235959");
ptime t(from_iso_string(ts))
```

Boost.DateTime

- Écriture sous forme de chaîne

```
std::string ts("2002-01-01 10:00:01.123456789");
ptime t(time_from_string(ts))

to_simple_string(ptime);
// 2002-Jan-01 10:00:01.123456789
to_iso_string(ptime);
// 20020131T100001,123456789
to_iso_extended_string(ptime);
// 2002-01-31T10:00:01,123456789
```

Boost.DateTime

- Accesseurs sur un élément de la date

```
ptime now = second_clock::local_time();

now.date().year();
now.date().month();
now.date().day();
now.date().day_of_week();
now.date().day_of_year();
now.date().week_number();
now.time_of_day().hours();
now.time_of_day().minutes();
now.time_of_day().seconds();
```

Boost.DateTime

- Fonctions de conversion
  - `end_of_month()` date du dernier jour du mois
  - `julian_day()` jour julien correspondant
  - `utc_time()` conversion local vers UTC
  - `local_time()` conversion UTC vers local
  - `to_tm()` conversion en une structure `tm`
- Opérateurs de comparaison
- Différence entre deux dates
- Ajout et soustraction d'une durée à une date ou à une autre durée
- Ajout et soustraction de jours à une date
- Multiplication et division d'une durée par un entier

Boost.DateTime 7/7

Do

Préférez `Boost.DateTime` aux classes *home-made*

Pour aller plus loin

- `Boost.Locale` pour gérer le formatage de `time_t` dans les flux
- ICU (*International Components for Unicode*) supporte de multiples calendriers (musulman, hébreu, chinois, perse, ...)

Boost.Format

- Alternative intégrée à C++20
- Formatage de chaînes de caractères
- Proche de `printf()` mais *type-safe* et extensible
- Basé sur une chaîne de format et la surcharge de l'opérateur `%`
- *Placeholders* numérotés (`%X%` ou `%|X$|`) indiquant la donnée à utiliser

```
cout << format{"%2%/%1%/%3%"} % 12 % 5 % 2014 << '\n';
// Affiche 5/12/2014
```

- *Placeholders* non-numérotés (`%||`) prenant les données dans l'ordre

```
cout << format{"%|| %|| %||"} % 12 % 5 % 2014 << '\n';
// Affiche 12 5 2014
```

Boost.Format

- Spécification dans la chaîne de format à la `printf()`
  - Alignement, présence du signe ou de la base et padding
  - Taille et précision
  - Type (uniquement pour le format de sortie)

Note

- Contrairement à `printf()`, le type dans le spécifieur de format n'impose pas le type de la variable
- Les `h`, `l` et `L` dans le type sont acceptés mais n'ont aucun effet

```
cout << format{"%|1$+|"} % 12 << '\n';
// Affiche +12
cout << format{"%|1$#x|"} % 12 << '\n';
// Affiche 0xc
```

Boost.Format

- Spécification sur la valeur avec `io::group()`

Note

S'applique à toutes les occurrences de la valeur dans la chaîne

```
cout << format{"%1% %2% %1%"} %
        io::group(showpos, 1) % 2 << '\n';
// Affiche +1 2 +1
```

Boost.Format

Motivations

- Plus expressif et souple que le formatage des flux
- *Type-safe* et extensible

Alternatives

`{fmt}`, SafeFormat, FastFormat, tinyformat

Boost.StringAlgorithms

- Manipulation des chaînes de caractères
- Changement de casse
  - Avec modification de la chaîne ou génération d'une nouvelle chaîne
  - Avec ou sans prise en compte de la locale

```
string foo{"Boost"};
cout << to_upper_copy(foo) << '\n'; // BOOST
```

Boost.StringAlgorithms

- Suppression de caractères
  - Avec modification de la chaîne ou génération d'une nouvelle chaîne
  - Suppression de la première, dernière, i^e ou toutes occurrences d'un caractère
  - Suppression de n caractères en début ou fin de chaîne

```
string foo{"Boost"};
cout << erase_all_copy(foo, "o") << '\n'; // Bst
```

Boost.StringAlgorithms

- Recherche de sous-chaîne
  - Recherche de la première, dernière ou i^{e} occurrence
  - Récupération des n premiers ou derniers caractères de la chaîne
- Concaténation de chaînes

```
vector<string> foo{"foo1", "foo2", "foo3"};
cout << join(foo, "-");   // foo1-foo2-foo3
```

- Découpage de chaîne

```
string foo = "Boost C++ Libraries";
vector<string> bar;
split(bar, foo, is_space());
// bar : "Boost", "C++", "Libraries"
```

Boost.StringAlgorithms

- Remplacement de caractères
  - Avec modification de la chaîne ou génération d'une nouvelle chaîne
  - Remplacement de la première, dernière, i^{e} ou toutes occurrences d'un caractère
  - Remplacement de n caractères en début ou fin de chaîne

```
string foo{"Boost"};
cout << replace_all_copy(foo, "o", "0") << '\n';
// B00st
```

Boost.StringAlgorithms

- *Trimming*
  - Avec modification de la chaîne ou génération d'une nouvelle chaîne
  - A droite, à gauche ou aux deux extrémités
  - Variante éliminant les doublons « d'espaces » dans la chaîne
  - Variante remplaçant les « espaces » dans la chaîne par une autre séquence de caractères
  - Variante prenant un prédicat de choix des caractères

```
string foo{"    Boost    "};
cout << trim_left_copy(foo) << '\n';  // "Boost    "
cout << trim_right_copy(foo) << '\n'; // "    Boost"
cout << trim_copy(foo) << '\n';       // "Boost"

string foo{"    Boost    Lib    "};
cout << trim_all_copy(foo) << '\n';   // "Boost Lib"
```

Autres doublons avec la bibliothèque standard

- `Boost.Regex` : gestion d'expressions rationnelles
  - Sensiblement identique aux *regex* de C++11
- `Boost.Bind` similaire `std::bind()`
- `Boost.LexicalCast` : conversion entre chaîne et nombre
  - API totalement différente de `std::to_string()` et `std::stoX()`

Do

Préférez la bibliothèque standard à Boost lorsque c'est possible

Boost.CircularBuffer

- Buffer circulaire dont la taille est définie à la création
- API compatible avec la bibliothèque standard
- `push_back()` permet d'ajouter un élément au buffer
- ... en écrasant le plus ancien si besoin

```
unsigned int CpuAlarm::getCurrentAlarmLevel() {
  m_lastCpuUseValues.push_back(getCurrentValue());
  unsigned int averageCpu = 0;

  for(auto it : m_lastCpuUseValues)
    averageCpu += it;

  averageCpu /= m_lastCpuUseValues.size();
  return averageCpu; }
```

Boost.LockFree

- Conteneur lock-free
  - `lockfree::queue`
  - `lockfree::stack`
  - `lockfree::spsc_queue` : file *lock-free* optimisée pour le cas « producteur unique / consommateur unique »
- API compatible avec la bibliothèque standard

Autres conteneurs de Boost

- `Boost.Array` similaire à `std::array`
- `Boost.Unordered` similaire aux tables de hachage de la bibliothèque standard
- `Boost.Heap` *priority queues* (plus riches que `std::priority_queue`)
- `Boost.MultiIndex` conteneurs indexés selon plusieurs critères
- `Boost.Bimap` map indexée par les deux entrées
- `Boost.MultiArray` tableaux multi-dimensionnels
- `Boost.Intrusive` support à la création de conteneurs intrusifs

Boost.Tokeniser

- *Tokenisation* d'une chaîne (ou d'une séquence) de caractères
- Itération sur les *token*

```
char_separator<char> sep("[]");
tokenizer<char_separator<char>> tokens(data, sep);

for(const std::string& text : tokens)
{ ... }
```

Boost.Asio

- Gestion des entrées/sorties en mode synchrone ou asynchrone
- Encapsulation des sockets bas-niveau

```
io_service io_service;
tcp::acceptor acceptor(io_service,
                       tcp::endpoint(tcp::v4(), 13));

for(;;) {
  tcp::socket socket(io_service);
  acceptor.accept(socket);
  ...}
```

Boost.Asio


```
io_service io_service;
tcp::resolver resolver(io_service);
tcp::resolver::query query("127.0.0.1", "daytime");
tcp::resolver::iterator endpoint_iterator = resolver.resolve
    (query);
tcp::socket socket(io_service);
connect(socket, endpoint_iterator);
```

- Mais également des ports séries et des timers

Et bien d'autres

- Pointeurs intelligents
- Graphes
- *Ranges*
- Algorithmes
- Tribool
- Programmation parallèle & communication inter-processus
- Système de signaux
- Écriture de parseurs et générateurs
- Programmation fonctionnelle
- Support à la méta-programmation
- Logs
- Options en ligne de commande
- Sérialisation
- ...

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 (cont.)

[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 (cont.)

[C++ Concurrency in action] Anthony Williams
C++ Concurrency in Action - Pratical Multithreading
Manning 9781933988771

[CppCoreGuidelines] isocpp
C++ Core Guidelines
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 (cont.)

[Overload] ACCU
Overload
https://accu.org/index.php/journals/c78/

[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 (cont.)

[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 (cont.)

[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