Geste de savoir

C++ TL ;DR news 3

8 juin 2021

Table des matières

1. A list of bad practices commonly seen in industrial projects 1
2. GotW #102 Solution : Assertions and “UB” (Difficulty : 7/10) 2
   2.1. Quelques définitions 2
   2.2. Codes d’exemple 3
3. 17 Smaller but Handy C++17 Features 4
   3.1. Pour le langage 4
   3.2. Pour la bibliothèque standard 5
   3.3. Fonctionnalités supprimées 8
4. C++ vs Rust—simple polymorphism comparison 8
5. [Livre] Advanced C++ 10

Pas de billet la semaine dernière. Il y a eu un long week-end chez moi (memorial day), donc c’était vacances. On reprend le billet cette semaine.

Lisez les autres C++ TL ;DR news: https://zestedesavoir.com/billets/3947/c-tl-dr-news/

— [Pour tous] les mauvaises pratiques que l’on rencontre régulièrement dans des projets industriels.
— [Pour tous] la solution du Guru of the Week 102.
— [C++17] quelques petites fonctionnalités utiles du C++17.
— [Pour tous] la comparaison du polymorphisme entre le C++ et Rust.
— [Pour tous] le livre ”Advanced C++”

---

1. A list of bad practices commonly seen in industrial projects

Lire l’article: https://belaycpp.com/2021/06/01/a-list-of-bad-practices-commonly-seen-in-industrial-projects/

Voici un article qui se lit facilement, sur les mauvaises pratiques que l’on rencontre régulièrement dans des projets industriels. Il n’y a rien de nouveau, mais c’est bien de faire des rappels de temps en temps.

— avoir des fonctions très très longues, par exemple plusieurs centaines de lignes. ”Pour résoudre un gros problème, divisez-le en plusieurs petits problèmes.”
— créer des classes lorsque ce n’est pas nécessaire. Par exemple une classe qui ne contient que des fonctions `static` (utilises un espace de noms) ou une classe avec principalement que des getters et setters (fais une structure et des attributs publiques).
— implémenter des comportements indéfinis, basés sur ce qu’on suppose que le compilateur fera.
— comparer des nombres signés et non signés.

— essayer d’optimiser ton code pendant que tu l’écris…
— … mais ne sois pas idiot non plus! N’écris pas du code qui serait sous-performant sans raison.
— trop en faire, au cas oú tu en aurais besoin plus tard. Si tu penses que tu auras besoin de quelque chose plus tard, il faut se poser la question du coût de l’évolution future versus le coût de le faire maintenant.

Si tu veux en savoir plus sur la qualité du code, je te conseille les livres de C. Martin (Clean code, Clean coder, Clean architecture).

2. GotW #102 Solution : Assertions and “UB” (Difficulty : 7/10)

Lire l’article: https://herbsutter.com/2021/06/03/gotw-102-solution-assertions-and-ub-difficulty-7–10/

Herb Sutter est l’un des experts C++ les plus influents: membre du comité de normalisation du C++, auteur de plusieurs livres très connus, speaker dans plusieurs conférences (je te conseille ses talks à la CppCon), développeur chez Microsoft.

Cet article est un Guru of the Week (GotW). Les GotW sont des articles de blog qu’il écrit depuis des dizaines d’années. Le principe est de poser une série de questions sur un thématique dans un premier article, les lecteurs apportent des éléments de réponse aux questions, puis un second article ”GotW Solution” présente les explications.

Cette article est la solution du GotW 102: https://herbsutter.com/2021/05/25/gotw-102-assertions-and-ub-difficulty-7–10/

2.1. Quelques définitions

2.1.1. Comportement indéfini (undefined behavior ou UB)

Est ”ce qu’il se passe lorsque votre programme essaie de faire quelque chose dont le sens n’est pas du tout défini dans le langage ou la bibliothèque standard C++ (code et/ou données illégaux)”. Le compilateur peut faire ce qu’il veut.

2.1.2. Comportement non spécifié (unspecified behavior)

Est ”ce qu’il se passe lorsque votre programme fait quelque chose pour lequel la norme C++ ne documente pas les résultats”. Le résultat est valide mais tu ne sais pas à l’avance ce qu’il sera.

2.1.3. Comportement défini par l’implémentation (Implementation-defined behavior)

Dépend du compilateur.

2.2. Codes d’exemple

2.2.1. Comportement indéfini

Exemple avec le déréférencement d’un pointeur null.

```
void deref_and_set( int* p ) {
    assert( p );
    *p = 42;
}
```

2.2.2. Comportement non spécifié

Exemple avec le calcul d’un point médian.

```
int midpoint( int low, int high ) {
    assert( low <= high );
    return low + (high-low)/2;
        // less overflow-prone than “(low+high)/2”
        // more accurate than “low/2 + high/2”
}
```

Guideline: un comportement non spécifié peut devenir un comportement indéfini si tu utilises le résultat sans vérifier que le résultat retourné ne produise pas un comportement indéfini.

Guideline: ne spécifie pas le comportement de ta fonction (postconditions) pour les entrée invalides (préconditions).

2.2.3. Comportement défini par l’implémentation

Dans le code d’exemple suivant, le comportement de la fonction ne devrait pas être valide si x est nul (assert). Mais un test est ajouté (programmation défensive) pour que le comportement soit quand même valide en release si l’utilisateur de la fonction fait une erreur.

```
some_result_value DoSomething( int x ) {
    assert( x != 0 );
    if ( x == 0 ) { return error_value; }
    return sensible_result(x);
}
```

Guideline: ne pas respecter une assertion n’est pas nécessairement un comportement indéfini.

Guideline: toujours documenter les contraintes sur les entrées de fonction (pré-conditions). L'utilisateur de la fonction doit savoir quelles valeurs d'entrée sont invalides.

Guideline: toujours respecter les préconditions quand tu utilises une fonction.

3. 17 Smaller but Handy C++17 Features

Lire l'article: https://www.cppstories.com/2019/08/17smallercpp17features/

Le titre de cet article est explicite: il présente 17 fonctionnalités mineurs mais utiles du C++17. Quelques fonctionnalités concernent le langage, mais la grande majorité concerne la bibliothèque standard. Les fonctionnalités majeures (structured bindings, filesystem, parallel algorithms, `if constexpr`, `std::optional`, `std::variant`, etc.) ne sont pas abordées ici.

3.1. Pour le langage

3.1.1. L'allocation dynamique pour les données alignées, avec `new`

Par exemple pour les données SIMD. Dans le code suivant, en C++11/14, l'alignement n'est pas garantie. En C++17, l'alignement est garantie.

```
class alignas(16) vec4
{
    float x, y, z, w;
};

auto pVectors = new vec4[1000];
```

3.1.2. Les variables membres statiques peuvent être `inline` en C++17

```
class MyClass {
    static inline std::string startName = "Hello World";
};
```

3.1.3. Ajout de la direction du pré-processeur `__has_include`

Pour vérifier si un header a déjà été inclus ou pas.

```
#if defined __has_include
    if __has_include(<charconv>)
        define has_charconv 1
        include <charconv>
    endif
#endif
```

3.2. Pour la bibliothèque standard

3.2.1. Ajout des variables template dans les traits

En C++14, le suffixe `_t` a été ajouté pour les traits pour remplacer `::type`. En C++17, le suffixe `_v` est ajouté pour remplacer `::value`.

```
std::is_integral_v<T>
std::is_class_v<T>
```

3.2.2. Ajout des méta-fonctions pour les opérateurs logiques.

- ET logique: `template<class... B> struct conjunction;`
- OU logique: `template<class... B> struct disjunction;`
- NON logique: `template<class B> struct negation;`

```
#include<type_traits>

template<typename... Ts>
std::enable_if_t<std::conjunction_v<std::is_same<int, Ts>...> >
PrintIntegers(Ts ... args) {
    (std::cout << ... << args) << '\n';
}
```

3.2.3. Ajout de `std::void_t`

Ce type peut être utile dans certaines implémentation de méta-fonctions.

```
template< class... >
using void_t = void;
```

3.2.4. Ajout de conversion de chaînes `std::from_chars`

Bas niveau et rapide, mais nécessite un peu plus de code.

```
const std::string str { "12345678901234" };
int value = 0;
const auto res = std::from_chars(str.data(), str.data() +
    str.size(), value);
```

3.2.5. Extraire un noeud d'une map ou d'un set

Permet de simplifier et optimiser le déplacement des noeuds.

```
inSet.emplace("John");
auto handle = inSet.extract("John");
outSet.insert(std::move(handle));
```

3.2.6. Ajout de `try_emplace()` dans `std::map` et `std::unordered_map`

`emplace` prend les arguments, même si l'insertion échoue, alors que `try_emplace` ne fait rien en cas d'échec.

```
std::map<std::string, std::string> m;
m["Hello"] = "World";

m.emplace("Hello", std::move(s)); // échec de l'insertion
// s a été move après emplace

m.try_emplace("Hello", std::move(s)); // échec de l'insertion
// s n'a pas été move après try_emplace
```

3.2.7. Ajout de `insert_or_assign()` dans `std::map` et `std::unordered_map`

`insert` n'ajoute pas l'élément si la clé existe déjà, `insert_or_assign` ajoute l'élément dans tous les cas.

```
myMap.insert_or_assign("c", "cherry"); // map contient "cherry"
myMap.insert_or_assign("c", "clementine"); // map contient
    "clementine"
```

3.2.8. `emplace` retourne le nouvel élément ajouté

```
// since C++11 and until C++17 for std::vector
template< class... Args >
void emplace_back( Args&&... args );

// since C++17 for std::vector
template< class... Args >
reference emplace_back( Args&&... args );
```

3.2.9. Ajout d'un nouvel algorithme `std::sample`

Pour extraire aléatoirement n éléments.

3.2.10. Ajouts de plusieurs fonctions mathématiques.

Par exemple `gcd()`, `lcm()`, `clamp()`, etc.

3.2.11. Support des tableaux dans `std::shared_ptr`

Avant, c'était possible uniquement avec `std::unique_ptr`.

```
std::shared_ptr<int[]> ptr(new int[10]);
```

3.2.12. Ajout de `std::scoped_lock`

Pour lock plusieurs mutex en même temps.

```
std::scoped_lock lck(first_mutex, second_mutex);
```

3.2.13. Ajoute de `std::invoke`

Pour invoquer des fonctions.

```
// a regular function:
std::cout << std::invoke(foo, 10, 12) << '\n';
```

```
3
4 // a lambda:
5 std::cout << std::invoke([](double d) { return d*10.0;}, 4.2) <<
      '\n';
```

3.3. Fonctionnalités supprimées

3.3.1. Suppression de `std::auto_ptr`

Enfin!

3.3.2. Suppression des anciens utilitaires pour les fonctions

`unary_function()`, `binary_function()`, ptr_fun()‘, etc.

4. C++ vs Rust— simple polymorphism comparison

Lire l’article: https://itnext.io/c-vs-rust-simple-polymorphism-comparison-e4d16024b57

L’auteur de cet article compare comment le polymorphisme est implémenté en C++ et en Rust. Le propos du polymorphisme est de pouvoir accéder à un type concret, dérivé d’un type de base, à l’éxecution, via un pointeur ou une référence.

Par exemple, des classes `Dog` et `Cat` qui dérivent de `Animal`.

Le code d’exemple:

```
1  #include <iostream>
2  #include <memory>
3  #include <vector>
4
5  class Animal {
6  public:
7      virtual ~Animal() = default;
8      virtual void talk() const = 0;
9  };
10
11 class Dog final : public Animal {
12 public:
13     void talk() const override { std::clog << "I'm a dog\n"; }
14 };
15
16 class Cat final : public Animal {
17 public:
```

```
18      void talk() const override { std::clog << "I'm a cat\n"; }
19  };
20
21  int main() {
22      std::vector<std::unique_ptr<Animal>> animals;
23      animals.emplace_back(std::make_unique<Dog>());
24      animals.emplace_back(std::make_unique<Cat>());
25      for (const auto& animal: animals) {
26          animal->talk();
27      }
28  }
```

Affiche:

```
1  I'm a dog
2  I'm a cat
```

La version Rust:

```
1   trait Animal {
2       fn talk(&self);
3   }
4
5   struct Dog;
6   struct Cat;
7
8   impl Animal for Dog {
9       fn talk(&self) {
10          println!("I'm a dog");
11      }
12  }
13
14  impl Animal for Cat {
15      fn talk(&self) {
16          println!("I'm a cat");
17      }
18  }
19
20  fn main() {
21      let animals : Vec<Box<dyn Animal>> = vec![Box::new(Dog{}),
        Box::new(Cat{})];
22      for animal in animals.iter() {
23          animal.talk();
24      }
25  }
```

Les différences entre les versions C++ et Rust:

— la classe `Animal` est implémentée sous forme d'un trait et les classes `Dog` et `Cat` implémentent ce trait.
— l'objet courant est passé en paramètre de la fonction via `&self`.
— les classes `Dog` et `Cat` sont vides et ne font rien. Tout se passe via les implémentations de la fonction `talk()` dans le trait `Animal`.
— `Box` est équivalent à `std::unique_ptr`, `dyn` indique que l'allocation est dynamique.

5. [Livre] Advanced C++

"Advanced C++ : Master the technique of confidently writing robust C++ code", de Gazihan Alankus, Olena Lizina et Rakesh Mane.

TL;DR: beaucoup de choses intéressante dans ce livre et beaucoup d'exercices proposés. Je le recommande, par exemple comme second livre après un cours débutant.

De nombreux chapitres ne concernent pas directement la syntaxe du C++, mais tout l'écosystème autour du C++:

— éditeur (avec Eclispe)
— outils de build (cmake)
— outils de tests (gtest)
— debug (dans Eclipse)
— les bonnes pratiques (outil de formatage dans Eclispe, nommage des variables et fonctions, lisibilité du code, conception avec les règles des 3/5/0)
— la documentation (cppreference.com)

Je ne suis pas un grand fan d'Eclipse pour le C++, je ne l'ai pas testé depuis très très longtemps. Par contre, je préfère très largement qu'un auteur se positionne clairement pour un outil, qu'il connaît et utilise tous les jours (peu importe que cet outil ne soit pas mon préféré), plutôt que de rester généraliste et vague (et donc inutile), sous pretexte de "laisser le choix aux lecteurs de choisir les outils qu'ils préfèrent".

Les explications et étapes pour reproduire les exemples sont très détaillées. Il y a de très nombreux exercices, qui sont fournis dans un dépôt sur GitHub: https://github.com/TrainingByPackt/Advanced-CPlusPlus .

Le code est moderne (C++17), je n'ai pas vu de problème dans les codes présentés.

Quelques exemples de notions abordées dans de livre:

Une présentation du processus de compilation, qui est illustrée avec des captures d'écran de Compiler Explorer. C'est, pour moi, un exemple qui montre que les auteurs sont très probablement des vrais développeurs, qui utilisent professionnellement le C++: ils présentent cet outil parce que c'est comme ça qu'ils travaillent et qu'ils ont un vraie connaissance pratique du C++. C'est le genre d'outils qu'on s'attend à ce qu'un vrai développeur connaissance, du fait de sa popularité dans les conférences C++, et qui est un outil pratique auquel on a recours assez facilement dans du développement de tous les jours, quand on se pose des questions sur des détails syntaxiques du langage.

Une présentation des types, pourquoi c'est important pour permet d'éviter ou détecter certains types d'erreurs.

Les auteurs présentent plusieurs syntaxe pour initialiser: sans valeur, avec `=` ou avec `{}` mais pas la syntaxe avec `()`. Cela me conforte dans l'idée qu'ils connaissent réellement le C++ et qu'ils omettent volontairement cette syntaxe possible à cause du https://en.wikipedia.org/wiki/Most_vexing_parse . C'est la différence avec un cours écrit par une personne qui n'est pas développeur (un enseignant par exemple, ou un développeur qui utilise pleins de langage et ne maîtrise pas le C++), qui va souvent faire un catalogue de syntaxes possibles, sans avoir le recul (et faire le tri) que fait un vrai développeur C++. Dit autrement, un vrai dev C++ ne cherchera pas à présenter toutes les syntaxes possibles, mais uniquement celles qui ont un sens dans un vrai code professionnel.

Un autre exemple de bonne approche pédagogique, au lieu d'expliquer directement la syntaxe des exceptions, les auteurs abordent progressivement les choses:

1. pourquoi les exceptions.
2. qu'est-ce que cela change pour le workflow.
3. comment utiliser `throw` et `catch`.
4. comment gérer la mémoire avec le RAII.
5. les exceptions et les classes RAII dans la STL.
6. question plus generale: qui est responsable (ownership).
7. la move semantic et les pointeurs intelligents (écrire ses propres classes de pointeurs et utiliser ceux de la STL).
8. impact sur le passage de paramètres de fonction.

Pleins de sujets abordés: le name lookup (ADL), l'organisation d'un projet et Pimpl, les threads, la gestion de fichiers, les tests, les performances.

Quelques détails pédagogiques qui me semblent critiquable, mais qui ne sont pas majeurs:

— les pointeurs nus sont présentés assez tôt, sans entrer dans les détails. Ce qui n'est pas catastrophique, vu que ce n'est pas un cours pour débutants.
— "The first and least preferred initialization mechanism…": si c'est moins préférée, pourquoi la présenter? Et en premier?

C'est une critique pédagogique habituelle d'enseigner en premier des syntaxes qui ne sont pas utilisées en vrai dans le monde professionnel. Mais c'est surtout quand le lecteur va avoir le temps de s'habituer à ces syntaxes, parce que cela nécessite un désapprentissage. Et on sait que les gens vont utiliser ce dont ils ont l'habitude. Quand la "mauvaise" syntaxe est apprise en même temps que la "bonne" et que cette dernière est celle que va utiliser en pratique celui qui apprend, ca ne sera pas forcément un problème.

D'autant plus qu'ici, l'initialisation des attributs lors de la déclaration de la classes est présentée juste après et il est bien précisé que c'est la méthode recommandée.

Ce livre et le livre suivant ("Expert C++) ont changé mon point de vue sur ce type d'éditeur. J'avais un a priori assez négatif, basé sur mes anciennes reviews. Mais j'ai été surpris de la qualité de ces livres. Ma conclusion: lisez des livres écrits par des développeurs et pas par des

enseignants qui ne pratiquent pas. Les enseignants n'écrivent pas des livres significativement mieux pédagogiquement, par contre ils sont souvent moins bon techniquement.

Mes autres critiques de livres: https://guillaumebelz.github.io/critiques.html

---

Cette semaine, les articles sont un peu plus long, mais relativement faciles à lire.

À partir de maintenant, je vais publier le lundi plutôt que le dimanche (fuseau horaire du pacifique). C'est plus simple pour moi.

À la semaine prochaine!