GBDIVERS

CC BY NC SA

Geste de savoir

C++ TL ;DR news 2

8 juin 2021

Table des matières

Second billet de cette série et déjà du retard 🍏 Il va falloir que je prenne l'habitude de publier le dimanche tel quel, peu importe le nombre de revues faites.

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

Au programme de cette semaine:

— [Pour tous] une petite annonce pour la conférence C++ FrUG de demain.
— [Pour tous] une revue rapide du livre "Expert C++".
— [C++17] un article qui présente le concept `std::optional::value_or` appliqué aux pointeurs.
— [c++17/20] un article sur 2 règles de bonnes pratiques pour les paramètres template.
— [C++20] un très long résumé sur les coroutines, pour présenter les concepts de base.

---

1. CPPFrUG 45

Cette semaine, je commence par une annonce. Le C++ French User Group organise régulièrement des soirées, avec des présentations et discussions sur le C++. Ce groupe se réunit normalement sur Paris, mais depuis le covid, les soirées sont organisées en ligne.

La prochaine réunion est demain (mardi 25 mai). Pour s'inscrire: https://www.meetup.com/fr-FR/User-Group-Cpp-Francophone/events/278281513/

Et les présentations de la dernière fois sont disponibles sur youtube https://www.youtube.com/channel/UCJlLdNmBQH9Rg-NxR68RYeQ

2. Expert C++ : Become a proficient programmer by learning coding best practices wit

Note: cette review est basée sur un survol rapide du PDF du livre, pas une lecture complète. Mon avis sur ce livre pourra changer après une lecture complète de celui-ci.

Globalement, sur un survol rapide du PDF, je dirais que c'est un livre avec une approche ascendante (du bas niveau - processus de compilation, optimisations, call stack, etc - vers le haut). Par exemple, quand il parle de la POO dans le chapitre 3, il s'intéresse à l'organisation des données en mémoire, pas aux questions de sémantiques de la POO.

Et il considère qu'il faut déjà connaître le C++ a priori. Les syntaxes sont utilisées sans être présentées (par exemple les classes).

En termes de structure du livre et de pédagogie, il me semble qu'il n'est pas complexe à lire, bien organisé, beaucoup d'illustrations, avec une progression logique. C'est pas une structure que l'on rencontre et conseille en général dans les cours débutant, mais pour le propos de ce livre, elle me semble cohérente.

La question est peut être quel est le public visé? Difficile à dire. Très clairement, ce n'est pas un livre que je conseille aux débutantes et débutants (mon conseil est de commencer par "C++ Primer" ou "Tour of C++", suivi de "Professional C++"), mais peut-être comme 3ème livre, pour celles et ceux qui veulent approfondir le bas niveau. Mais celles et ceux qui sont intéressé par ça peuvent aussi regarder des livres comme "Performance Analysis and Tuning on Modern CPUs" de Denis Bakhvalov, plus technique mais aussi plus avancé. Je dirais que l'avantage de ce livre est de regrouper pleins de petites informations que l'on trouve habituellement dans des livres différents. L'inconvénient est que vouloir couvrir pleins de thématiques fait que celle-ci sont survolés de très loin.

3. A Default Value to Dereference Null Pointers

Lire l'article original: https://www.fluentcpp.com/2021/05/14/a-default-value-to-dereference-null-pointers/

Dans certains cas, on souhaite pouvoir indiquer quand une valeur correspond à un état spécifique, par exemple une erreur ou que la valeur n'est pas déterminée. Des exemples concrets:

```
1 std::pair<iterator, bool> std::map::insert(value);   // (1)
2 size_t std::string::find(str, pos);                  // (2)
3 GLenum glGetError();                                 // (3)
4 QVariant QVariant::fromValue(value);                 // (4)
```

Dans ces codes, il est nécessaire de distinguer quand la fonction retourne une valeur invalide. Dans (1), le booléen indique si l'insertion a réussie. Dans (2), la valeur retournée correspond a la position dans la chaine ou a `std::string::npos` si la recherche a echouee. Dans (3), la valeur retournée est une énumération correspondant à une erreur ou `GL_NO_ERROR` s'il n'y a

pas d'erreur. Dans (4), le `QVariant` retourné contient la valeur si la conversion est possible ou est invalide.

La classe `std::optional` permet de manipuler un objet qui peut être indéterminée ("nullable object"), avec une sémantique spécifique pour gérer le cas où la valeur est indéterminée. Dit autrement, `std::optional<T>` peut contenir toutes les valeurs possible de `T`, plus un état supplémentaire `std::nullopt`.

```
std::optional<int> f()
{
    if (thereIsAnError) return std::nullopt;
    // happy path now, that returns an int
}

auto result = f();
std::cout << (result ? *result : 42) << '\n';

std::cout << f().value_or(42) << '\n';
```

Les pointeurs sont des objets qui peuvent être nullable aussi, mais la dernière syntaxe n'est pas utilisable. Comment ajouter `value_or` pour les pointeurs? La solution proposée est d'utiliser une fonction libre:

```
template<typename T, typename U>
decltype(auto) value_or(T* pointer, U&& defaultValue)
{
    return pointer ? *pointer : std::forward<U>(defaultValue);
}
```

Le type de retour (rvalue ou lvalue) va dépendre de la catégorie de valeur de la valeur par défaut.

4. Function Templates - More Details about Explicit Template Arguments and Concept

Lire l'article d'origine: http://www.modernescpp.com/index.php/function-templates-more-details

Cet article présente deux nouvelles "règles" de bonne pratique, sur les arguments template (C++17) et les concepts (C++20).

4.1. Explicitly Specifying the Template Arguments

Cette ”règle” est très simple:

```
std::vector<int> myVec{1, 2, 3, 4, 5};  // avant C++17
std::vector myVec{1, 2, 3, 4, 5};       // depuis le C++17
```

Il faut préférer la seconde syntaxe au lieu de la première.

Cette règle peut surprendre, mais elle est en fait logique: elle est l'équivalent de la même règle pour les fonctions, appliquée aux classes. Pour une fonction template, on va généralement préférer la déduction des arguments template selon les arguments de la fonction:

```
template <typename T>
T max(const T& lhs,const T& rhs);

auto res1 = max<float>(5.5, 6.0);   // non
auto res2 = max(5.5, 6.0);          // oui
```

Dans le cas d'un overload de fonctions template et non template:

```
double max(const double& lhs, const double& rhs);

template <typename T>
T max(const T& lhs,const T& rhs);

auto res1 = max(5.5, 6.0);    // (1)
auto res2 = max<>(5.5, 6.0);  // (2)
```

Ce code n'est pas ambigüe, du fait que la ligne (1), qui peut utiliser les 2 fonctions, va préférer la fonction non template et la ligne (2) ne va considérer que la fonction template.

4.2. Overloading with Concepts

La seconde ”règle” consiste simplement à contraindre par défaut les paramètres template en utilisant les concepts.

```
MyClass max(MyClass lhs, MyClass rhs);

template <std::totally_ordered T>
T max(const T& lhs,const T& rhs)

```

```
6  template <typename T>
7  T max(const T& lhs,const T& rhs);
8
9  auto value2 = max(MyClass{1}, MyClass{2});    // (1)
10 auto value2 = max(1, 2);                      // (2)
```

Dans ce code, la ligne (1) n'est pas ambigue, puisqu'elle va préférer la fonction non template. La ligne (2) n'est pas non plus ambigue, puisqu'elle va préférer la fonction template avec contrainte (par le concept `std::totally_ordered`).

5. C++20 Coroutine : Under The Hood

Lire l'article d'origine: http://www.vishalchovatiya.com/cpp20-coroutine-under-the-hood/

Note: pour cette article, j'ai volontairement fait un revue plus détaillée. Je n'avais pas encore expliqué les concepts de base des coroutines, donc il me semblait intéressant d'écrire un résumé plus long. Cependant, l'article original contient des informations supplémentaires, n'hésitez pas à aller le lire si vous voulez plus de détails, en particulier sur le fonctionnement interne des coroutines.

Cet article fait suite à un article pour faire des "coroutine" en C, avec les fonctions systèmes. Le lien est donné dans l'article. Quelques notions sont présentées dans ce premier article.

Une coroutine est une fonction qui peut être suspendue et reprise. Elle peut être vue comme un intermédiaire entre les fonctions et les threads. Comparé aux threads:

— pas la lourdeur du contexte de thread, ni du changement de contexte.
— Et contrairement aux threads, qui peuvent être suspendu à n'importe quel moment par le scheduler du système, les coroutines sont suspendus à un point déterminé du code (donc beaucoup plus simple a gérer la concurrence).

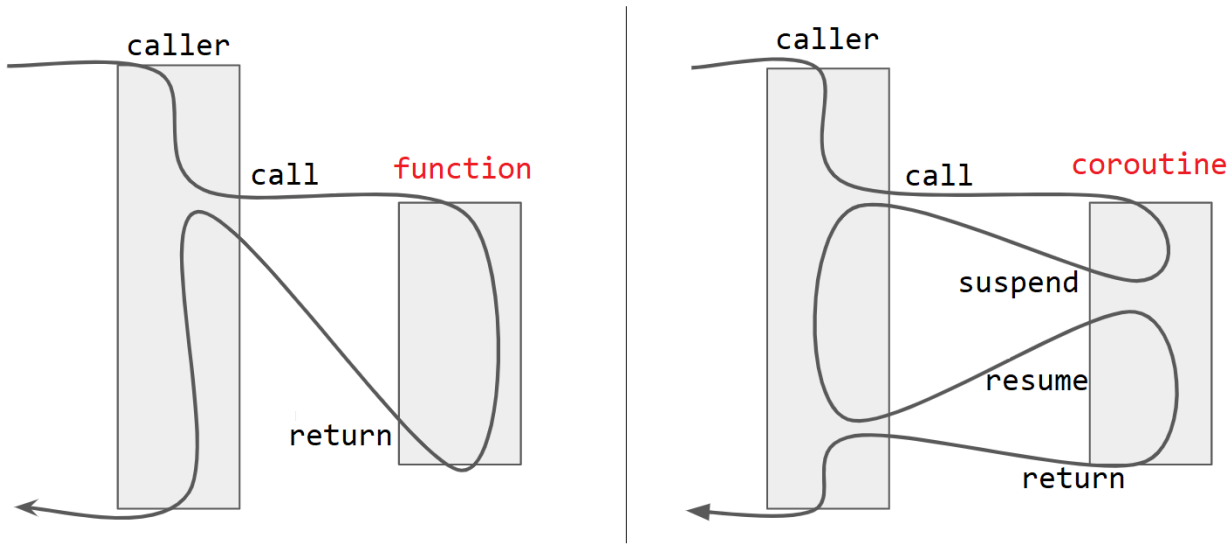

FIGURE 5.1. – Image provenant de l'article cité

En pratique, qu'est-ce qu'une coroutine en C++20? Une fonction qui contient `co_await`, `co_yield` et/ou `co_return`, et qui peut retourner `std::promise`.

Du point de vue haut niveau, une coroutine est:

- un promise, qui contrôle le comportement de la coroutine et sert d'intermédiaire entre le code appelant et le code appelé
- un awaiter, qui contrôle la suspension et la reprise de la coroutine
- un coroutine handler, qui contrôle l'exécution

L'article donne des liens vers 2 exemples d'utilisation de coroutines, dans le design pattern Iterator et dans un générateur de séquence d'entiers.

5.1. Suspendre une coroutine

```
1  struct HelloWorldCoro {
2      struct promise_type { // compiler looks for `promise_type`
3          HelloWorldCoro get_return_object() { return this; }
4          std::suspend_always initial_suspend() { return {}; }
5          std::suspend_always final_suspend() { return {}; }
6      };
7      HelloWorldCoro(promise_type* p) :
           m_handle(std::coroutine_handle<promise_type>::from_promise(*p))
           {}
8      ~HelloWorldCoro() { m_handle.destroy(); }
9      std::coroutine_handle<promise_type>      m_handle;
10 };
```

Dans ce code:

- Promise i.e. promise_type, doit contenir certaines fonctions
- Awaiter i.e. std::suspend_always
- Coroutine handle i.e. std::coroutine_handle

Utilisation:

```
1  HelloWorldCoro print_hello_world() { // ligne 1
2      std::cout << "Hello "; // ligne 2
3      co_await std::suspend_always{}; // ligne 3
4      std::cout << "World!" << std::endl; // ligne 4
5  } // ligne 5
6
7  int main() {
8      HelloWorldCoro mycoro = print_hello_world(); // ligne A
9      mycoro.m_handle.resume(); // ligne B
```

```
10      mycoro.m_handle.resume(); // ligne C
11      return EXIT_SUCCESS; // ligne D
12  }
```

Pour résumer ce qu'il se passe, le premier `resume` affiche `Hello` , le second affiche `World!`. Pour entrer plus dans les détails, le flux d'exécution suit les étapes suivantes:

- à la ligne A dans la fonction `main`, la fonction `print_hello_world` est appelée.
- à la ligne 1 de la fonction `print_hello_world`, le contexte de la coroutine est créé (`HelloWorldCoro`) puis la coroutine est suspendue en retournant ce contexte.
- à la ligne B, l'exécution de la coroutine est reprise à la ligne 2. Le texte "Hello " est affiché.
- à la ligne 3, la coroutine est de nouveau suspendue.
- à la ligne C, l'exécution est reprise à la ligne 4 et le texte "World!" est affiché.
- à la ligne 5, le contexte est détruit et la coroutine se termine.
- le programme se termine à la ligne D.

L'article entre plus en détail sur les codes intermédiaires qui sont générés lors de la compilation.

Note: l'exécution de la coroutine est suspendue juste après le lancement de celle-ci et la ligne 2 n'est pas appelée avant le premier `resume`. Cela est dû au fait que la fonction `initial_suspend` retourne `std::suspend_always`. Il est possible de changer ce comportement, pour que la ligne 2 soit exécutée dès l'appel à la coroutine, en utilisant le type standard `std::suspend_never`.

5.2. Retourner une valeur depuis une coroutine

Comme une coroutine doit retourner le "promise" pour contrôler le flux d'exécution, il n'est pas possible de retourner directement une valeur, comme pour une fonction normale. Pour cela, il faut `co_return` et `return_value`:

```
1  struct HelloWorldCoro {
2      struct promise_type {
3          int m_value;
4          void return_value(int val) { m_value = val; }
5          ...
6      };
7  };
8
9  HelloWorldCoro print_hello_world() {
10     std::cout << "Hello ";
11     co_await std::suspend_always{ };
12     std::cout << "World!" << std::endl;
13     co_return -1;
14 }
15
16 int main() {
17     HelloWorldCoro mycoro = print_hello_world();
```

```
18      mycoro.m_handle.resume();
19      mycoro.m_handle.resume();
20      assert(mycoro.m_handle.promise().m_value == -1);
21      return EXIT_SUCCESS;
22  }
```

Dans ce code, la valeur de retour est déclarée dans la structure `promise_type`, avec la fonction `return_value`. Dans la coroutine, une valeur est retournée par `co_return`, puis cette valeur est récupérée via le promise `mycoro.m_handle.promise().m_value`.

5.3. Rendre une valeur de Coroutine

La valeur retournée par la coroutine ne peut être modifiée qu'une seule fois, lors de l'appel à `co_return`, ce qui stop l'exécution de la coroutine. Si la ligne contenant `co_return` dans le code précédent est déplacée après la ligne `suspend_always`, le texte "World !" ne sera jamais affiché.

Mais dans un code asynchrone comme celui-ci, il peut être intéressant de retourner une valeur à chaque fois que la coroutine est suspendue. Pour cela, il faut utiliser `co_yield` et `yield_value`:

```
1   struct HelloWorldCoro {
2       struct promise_type {
3           int m_val;
4           std::suspend_always yield_value(int val) {
5               m_val = val;
6               return {};
7           }
8           ...
9       };
10  };
11
12  HelloWorldCoro print_hello_world() {
13      std::cout << "Hello ";
14      co_yield 1;
15      std::cout << "World!" << std::endl;
16  }
17
18  int main() {
19      HelloWorldCoro mycoro = print_hello_world();
20      mycoro.m_handle.resume();
21      assert(mycoro.m_handle.promise().m_val == 1);
22      mycoro.m_handle.resume();
23      return EXIT_SUCCESS;
24  }
```

Contrairement à `co_return` qui stoppait l'exécution de la coroutine, `co_yield` suspend simplement la coroutine en retournant une valeur. La coroutine peut être reprise ensuite.

5.4. Terminologie utilisée avec les coroutine en C++20

- awaitable: type qui accepte l'opérateur unaire `co_await`. Dans les codes d'exemple précédant, le type standard `std::suspend_always` a été utilisé, mais il est possible de déclarer son propre type.
- awaiter: type qui implemente les fonctions `await_ready`, `await_suspend` et `await_resume`. Dans les codes d'exemple précédant, c'était le type standard `std::suspend_always`. Il existe aussi le type standard `std::suspend_never`.
- promise: type qui implemente le type `promise_type`, qui contient un certain nombre de fonctions définies.
- coroutine handle: permet de contrôler l'exécution de la coroutine. Le type standard `std::coroutine_handle` dans les codes d'exemple précédents.
- coroutine state (ou context object): contient les informations du contexte de la coroutine (promise object, paramètres d'appel, variables locales, informations de contrôle de l'exécution), sur la Pile.

Les opérateurs unaires:

- `co_await` suspend l'execution et s'appelle avec un "awaiter".
- `co_yield` suspend l'execution et retourne une valeur.
- `co_return` stop l'exécution et retourne une valeur.

L'article contient plus de détails sur le fonctionnement interne et l'implémentation possible des coroutines en C++20. Si vous voulez entrer dans les profondeurs des coroutines, vous pouvez lire la série d'articles de Raymond Chen: https://devblogs.microsoft.com/oldnewthing/20210504–01/ ?p=105178

Il y a malheureusement trop d'articles publiés chaque semaine pour que je puisse faire un résumé de chacun. Surtout si j'inclue Qt, la 3D, des livres, des videos, etc.

Comme il faut que cette revue soit rapide à lire (et accessoirement à écrire), il faudra que je sois plus sélectif sur la liste d'articles. Mais je trouvais intéressant de faire un résumé plus long pour l'article sur les coroutines. Ca aurait pu faire l'objet d'un tutoriel indépendant, probablement.

Si vous avez des commentaires ou si vous voulez proposer des articles à relire, vous pouvez le faire sur le GitHub de NaN: https://github.com/NotANameServer/discord/issues/30