Zeste de savoir

[CppCon 2015] Retour sur la conférence de B. Stroustrup

12 août 2019

Table des matières

La conférence annuelle CppCon 2015 a eu lieu la semaine dernière du 21 au 25 Septembre. CppCon est la grande-messe du C++ où se rencontrent de nombreux utilisateurs et concepteurs du langage pour échanger et discuter. La semaine fut rythmée par de nombreuses conférences qui sont enregistrées et mises à disposition sur Youtube .

Les premières vidéos de cette session 2015 sont déjà disponibles : on y trouve la keynote de Bjarne Stroustrup, « Writing Good C++14 » ainsi que la vidéo « Writing Good C++14... By Default » de Herb Sutter. D'autres devraient arriver d'ici peu !

Dans sa keynote d'ouverture, Bjarne Stroustrup (le créateur du langage) a développé de nombreux points (dont le projet CoreGuideLine qui sera développé dans un prochain article), mais on va ici s'attarder sur deux idées : `owner` et `not_null`. Mais avant de voir de quoi il en retourne, regardons ce qui compose la grande majorité des bugs en C++ et quels problèmes ces outils vont résoudre.

1. Les erreurs classiques

Le C++ possède (en raison de son histoire) une forte compatibilité avec le C, qui peut se traduire par un certain nombre de problèmes de design si on cherche à calquer l'utilisation du C++ sur le C. Ce *mauvais* C++ est souvent qualifié de *C++ old school*, *C with classes* par opposition au « C++ moderne » qui vise à promouvoir des outils et *patterns* efficaces en terme de sécurité, maintenabilité et facilité de lecture du code.

1.1. Mauvaise utilisation des pointeurs

Parmi les points les plus critiqués du C++ *old school* on trouve les pointeurs et les accès aux tableaux. Pour rappel, pour utiliser un pointeur, celui-ci doit être valide, c'est-à-dire qu'il doit pointer sur un objet valide. Les erreurs proviennent du fait qu'un pointeur vaut `nullptr` (qui correspond à une adresse invalide) ou que le pointeur pointe sur un objet invalide (*dangling*

pointer, soit parce que le pointeur n'est pas correctement initialisé, soit parce que l'objet a été détruit).

```
Object* p = nullptr;
p->f();  // erreur, appel sur nullptr

Object* pp;
pp->f(); // erreur, pointeur invalide

Object* ppp = new Object;
delete ppp;
ppp->f();  // erreur, pointeur invalide
```

Bien sûr, sur ces codes aussi simples, l'erreur est évidente (bien que beaucoup de personnes ignorent en fait qu'un pointeur non initialisé ne vaut pas `nullptr`, mais prend une valeur aléatoire). Mais on rencontre souvent ces erreurs dans de vrais projets, ce ne sont pas que des problèmes théoriques.

Le second type d'erreur sont les accès en dehors des limites d'un tableau.

```
int array[5];
array[10] = 0;  // erreur, accès hors limite
```

Avec les tableaux C++ (`std::vector`, `std::array`, etc.), il est possible de tester facilement ce problème en utilisant la fonction membre `size` et une assertion.

```
std::vector<int> array(5);
assert(10 < array.size());  // produira un crash ici
array[10] = 0;
```

Il est possible (et je recommande) de toujours mettre un `assert` devant un accès à un tableau.

Le problème est plus compliqué avec les tableaux de style C. Ceux-ci ne conservent pas leur taille et il est facile de perdre cette information. Un dernier type d'erreur est la fuite de mémoire (*memory leak*). Cela arrive lorsqu'un objet créé dynamiquement n'est pas correctement libéré.

```
int* p = new int;
p = new int;  // perte du pointeur sur le premier objet créé

int* p = new int[10];
delete p;  // appel de delete au lieu de delete[]
```

1.2. Undefined Behavior

Beaucoup de débutants (ou plus expérimentés) font une erreur classique : les problèmes décrits précédemment ne produisent pas d'erreur de compilation. Et ne produisent pas non plus de crash. Pas toujours en tout cas. Et jamais en indiquant la ligne de code qui pose problème. Ce type d'erreur produit ce que l'on appelle un comportement indéfini (*Undefined Behavior* ou *UB* dans le jargon). Cela signifie que le comportement du programme n'est pas garanti par la norme. Il peut continuer à avoir un comportement normal, ne pas crasher, mais donner des résultats faux, ou crasher à n'importe quel moment. Le comportement observé peut changer en fonction du compilateur ou des options de compilation, ce qui rend ce type d'erreur très difficile à diagnostiquer et parfois à corriger.

Accéder à une zone mémoire invalide est un exemple d'*undefined behavior* et ceci a plus de chances de se produire quand on manipule des pointeurs nus, car on dispose de moins d'informations sur ce dernier (est-ce un tableau ? dois-je libérer la zone pointée ?) et le programmeur est donc plus susceptible de commettre une erreur.

1.3. Une solution ?

À cause de ces problèmes et de la difficulté pour les diagnostiquer, le C++ est souvent considéré comme un langage de programmation complexe. La solution proposée par B. Stroustrup est finalement assez simple.

Ne faites pas cela !

Quand on rencontre des problèmes avec une approche, un concept, une syntaxe, le plus simple pour éviter les erreurs est de ne pas faire cela.

Bon, ok, c'est un peu facile. Qu'est-ce que l'on doit faire à la place alors ?

Il existe beaucoup de nouveaux concepts en « C++ moderne », mais ces concepts ne sont pas liés à une norme du langage en particulier (C++03, C++11 ou plus). Un exemple classique sont les pointeurs « intelligents », beaucoup pensent que c'est un des ajouts majeurs du C++11. Mais ce n'est pas le cas ! La nouveauté du C++11 est simplement de proposer une implémentation de ces pointeurs dans la bibliothèque standard. Les pointeurs intelligents sont utilisables en C++03, en utilisant des bibliothèques (Boost, Qt, etc.) ou en les implémentant soi-même.

Pourtant, nombre de ces concepts ne sont pas encore assimilés par les développeurs C++ (débutants ou anciens). Les raisons sont multiples, mais il y a un point cité par B. Stroustup qui m'intéresse particulièrement. Dans de nombreux cas, on a tendance, dans les *guidelines* ou sur les forums, à donner des règles de façon assez directive (« ne fais pas cela ») sans forcement expliquer le pourquoi et donner l'approche correcte. Alors qu'il est plus logique de montrer le bon comportement dès le début plus que dire ce qu'on ne doit pas faire.

2. Quelques approches originales

Je ne vais pas entrer dans le détail de toutes les solutions en « C++ moderne » permettant d'éviter ces problèmes (c'est l'objet de mon cours C++), mais détailler deux concepts présentés par B. Stroustrup : `owner` et `not_null`. Une proposition d'implémentation par Microsoft est publiée sur GitHub.

2.1. Owner

L'implémentation de ce type est tellement simple que je me permets de copier ici le code provenant du GitHub de Microsoft.

```
1 template <class T>
2 using owner = T;
```

Là, normalement, vous vous dites.

> ? Ils se moquent de moi ? Ce type ne fait absolument rien !

Et vous auriez parfaitement raison

Mais analysons la situation plus en détail. Ce type a pour objectif de résoudre le problème de la libération de la mémoire.

Généralement, l'acquisition d'une ressource n'est pas la source principale de problèmes dans un programme. Le plus simple est que chacun est responsable de ses ressources, on les alloue avant de les utiliser et on les libère lorsque l'on n'en a plus besoin. Plus souvent, on va utiliser une ressource qui a été allouée par quelqu'un d'autre. Dans les cas extrêmes, il y aura un responsable dédié pour un type de ressource, tous ceux qui veulent ce type de ressource doivent passer par lui.

Par contre, la libération est plus problématique : personne ne va prendre la responsabilité de libérer la ressource, tout le monde va considérer qu'un autre le fera. Ce problème survient parce que personne n'est clairement désigné pour être le propriétaire d'une ressource, c'est-à-dire celui qui est responsable de la libérer (*ownership* ou « propriété » en français).

> *i* À mon sens, l'apport principal des pointeurs intelligents du C++11 n'est pas les pointeurs eux-mêmes, mais la réflexion que cela a engendré sur l'importance de définir clairement qui est le propriétaire.

Le problème est simple : on doit toujours libérer les ressources. Mais comment savoir qui doit le faire et quand le faire ? Prenons un code d'exemple.

```
void f(int* p) {
    // On reçoit une ressource du code appelant,
    // doit-on la libérer ?

    int* pp = g();
    // Autre ressource reçue, doit-on la libérer ?

    int* ppp = new int;
    h(ppp);
    // On donne une ressource à un autre. Comment
    // dire que l'on souhaite qu'il la libère ?
    // Ou lui dire que l'on va la libérer
    // nous-même ?
}
```

À chaque fois, il manque un moyen de dire qui est le propriétaire de la ressource, si on devient propriétaire ou pas d'une ressource que l'on reçoit, ou si on donne la propriété ou non d'une ressource à un autre. C'est donc un simple problème d'expressivité, pouvoir dire ce que l'on souhaite faire. Qui n'a pas été confronté un jour à un code écrit par un autre et s'est demandé ce qu'il voulait faire ?

Le rôle de `owner` est de simplement exprimer cette intention sur la propriété. L'idée est qu'une ressource transmise en utilisant `owner` transmet la propriété, une ressource transmise par un pointeur nu ne la transmet pas.

```
void f(int* p);
    // f ne prend pas la propriété, si on lui donne
    // une ressource, il ne va pas la libérer.

void g(owner<int*> p);
    // g prend la responsabilité de la ressource. On
    // n'est plus propriétaire de celle-ci et on ne pas la
    // libérer.

int* h();
    // h transmet une ressource, mais pas la
    // propriété, on ne doit pas la libérer.

owner<int*> i();
    // i transmet une ressource et la propriété,
    // il faut libérer la ressource.
```

Bien sûr, ce type est purement indicatif pour le développeur, une mauvaise utilisation ne va pas provoquer d'erreur de compilation ou d'exécution.

Il est par contre possible d'utiliser des outils d'analyse statique, qui pourraient prendre en compte ce type d'information.

En suivant une règle simple, on peut alors améliorer la qualité du code, et éviter les problèmes de libération.

> Quand on est propriétaire d'une ressource (que ce soit une ressource que l'on a allouée soi-même ou que l'on a reçue), lorsqu'on arrive en fin de portée du pointeur, soit on transmet la propriété à un autre (dans une fonction que l'on appelle ou en retour de fonction), soit on libère la ressource.

Il suffit parfois d'améliorer l'expressivité pour améliorer la qualité d'un code.

Certains l'auront compris, `owner` est une transmission de propriété non partagée. S'il faut avoir plusieurs propriétaires, `owner` ne pourra pas être utilisé, il faudra utiliser par exemple `std::shared_ptr` ou trouver un autre moyen d'indiquer la propriété. Je précise quand même que le partage de la propriété devrait être une situation d'exception plus que la règle.

2.2. Not_null

La seconde classe n'est pas tellement plus compliquée, elle tient en quelques lignes. Je vous laisse aller voir le code sur le GitHub de Microsoft. L'idée est assez simple : on ne doit pas utiliser un pointeur `nullptr` ? Il suffit d'interdire à un pointeur de l'être !

Pour cela, la classe `not_null`, qui possède une sémantique de pointeur, interdit explicitement l'utilisation de `nullptr`, `0` ou `NULL`. À la construction.

```
1 not_null(std::nullptr_t) = delete;
2 not_null(int) = delete;
```

Et pour l'affectation.

```
1 not_null<T>& operator=(std::nullptr_t) = delete;
2 not_null<T>& operator=(int) = delete;
```

Ces opérations supprimées permettent de produire une erreur lorsque l'on essaie de créer un pointeur nul.

```
int main() {
    not_null<int*> p = 0;
}
```

Ce code affiche ce qui suit.

```
main.cpp:59:20: error: conversion function from 'int
' to 'not_null<int *>' invokes a deleted function
  not_null<int*> p = 0;
                 ^    ~
main.cpp:12:5: note: 'not_null' has been explicitly
marked deleted here
  not_null(int) = delete;
  ^
1 error generated.
```

Les opérations arithmétiques sont aussi interdites (cette classe ne doit pas servir pour un pointeur sur un tableau).

```
not_null<T>& operator++() = delete;
not_null<T>& operator--() = delete;
not_null<T> operator++(int) = delete;
not_null<T> operator--(int) = delete;
not_null<T>& operator+(size_t) = delete;
not_null<T>& operator+=(size_t) = delete;
not_null<T>& operator-(size_t) = delete;
not_null<T>& operator-=(size_t) = delete;
```

Il est bien sûr assez facile de « tromper » le compilateur, en faisant une conversion ou en passant par une variable intermédiaire.

```
int main() {
    not_null<int*> p = static_cast<int*>(nullptr);
}
```

Mais c'est du ressort explicite du programmeur que de faire une telle chose.

La sécurité de `not_null` n'est pas non plus garantie par le compilateur ou lors de l'exécution. C'est surtout une amélioration de l'expressivité du code. Si on manipule un `not_null`, on sait que l'on ne doit pas passer un pointeur nul ou qui peut être nul. Basiquement, cela veut dire que l'on ne doit pas affecter un pointeur nu à un `not_null`.

```
int* p = f();
not_null<int*> q = p;  // violation du contrat
                       // de not_null
```

3. Conclusion

On voit par ces deux exemples simples l'importance de l'expressivité. Une erreur dans un code peut survenir parce que l'on ne sait pas tout de suite ce que l'on doit faire, quand on doit libérer une ressource, quand un pointeur est nul.

L'amélioration du code « C++ moderne » n'est pas simplement une affaire de syntaxe et notions complexes à maîtriser, mais aussi transmettre les bonnes attitudes et questionnements aux développeurs C++, comme se poser la question de la propriété d'une ressource ou de la nullité d'un pointeur.

Ou de ne pas utiliser de pointeurs lorsque cela n'est pas nécessaire (utiliser des références, des passages par valeur ou par déplacement).

Note : cet article est directement inspiré du billet de blog suivant [CppCon 2015] Stroustrup : Don't do that ! et adapté par Davidbrcz pour Zeste de Savoir.

Merci à Dominus Carnufex pour la lecture orthographique et grammaticale.