PIERRE_24

CC BY

Zeste de savoir

J’aurais aimé t’aimer, C++

jeudi 09 janvier 2025

Table des matières

Introduction . . . 1
1. Pourquoi ? . . . 2
2. C++ = C with classes (pun intended) . . . 4
2.1. Rétro-compatibilité, quand tu nous tiens . . . 4
2.2. Et expressif, avec ça ? . . . 5
2.3. Et donc … . . . 6
3. Et puis viennent les templates . . . 7
3.1. La compilation . . . 8
3.2. … Interlude : `-tidy-` … . . . 10
3.3. … Et l'héritage, donc. . . . 11
3.4. Et donc ? . . . 11
Conclusion . . . 12
Contenu masqué . . . 14

Introduction

Bonne année 2025 à vous !

Comme vous le savez peut-être, je passe mes journées à faire de la chimie quantique. En pratique, il s'agit de lancer des calculs via des programmes bien particuliers, voir parfois de créer lesdits programmes. Que font ces programmes ? Basiquement, de l'algèbre linéaire sur des matrices (multiplications, résolutions de système d'équations, etc) de tailles relativement importantes. Comment on fait ça ? En essayant d'utiliser un maximum de puissance de calcul. On peut pour cela utiliser du *multi-threading* (plutôt du multi-coeurs, en fait), demander à plusieurs machines de travailler sur le même problème, voire utiliser un (ou plusieurs !) GPU(s).

De manière logique, là ou le *mult-threading* est de plus en plus courant dans les langages de programmation (et même pour les plus vieux, il y a moyen de faire quelque chose facilement avec OpenMP), le fait de synchroniser plusieurs machines en même temps ou d'utiliser des GPUs est moins répendu. Et ce qui l'est encore moins, c'est des bibliothèques d'algèbres linéaires qui permettent de faire cela (parce qu'en vrai, c'est pas facile à faire).

Bref, me voilà avec pour bonne résolution pour cette année 2025, celle de réécrire *from scratch* un programme en C++ afin d'utiliser une telle bibliothèque. Et … Je pense que ça mérite que je partage mon expérience.

Le contenu du billet, qui est un billet d'opinion, est un peu provocateur. N'en prenez pas ombrage : chaque langage possède ces qualités et ces défauts. En plus, malgré tout ce

que je peux dire ici, je vais apprendre de mes erreurs et **vraiment** le coder en C++, ce programme

1. Pourquoi ?

TL ;DR : Vu que je fais du calcul intensif, j'ai pas beaucoup d'autres choix, et j'aime la POO.

Quand je vous ai dis plus haut, qu'il n'existait pas beaucoup de bibliothèques qui permettait de faire de l'algèbre linéaire, je ne vous ai malheureusement pas menti. Oui, il y a `Eigen` , mais elle ne fait pas du multi-noeuds.

Historiquement, tout commence avec une collection de routines écrites en **Fortran 77** distribuées entre dévellopeurs, qui finira par s'appeler BLAS . Vu qu'il s'agit de routines relativement modulables et utiles dans le petit monde du calcul intensif, BLAS est devenu un standard *de facto*. On lui a ensuite accolé un petit copain, LAPACK (qui contient des routines plus évoluées), mais qui a également été victime de son succès. Ce qui signifie qu'un **très** grand nombre de bibliothèques d'algèbre linéaire "modernes" sont soit : a) des *wrappers* plus ou moins *fancy* autour de BLAS/LAPACK (c'est le cas de numpy en Python, dont on ne dira jamais assez combien elle est géniale), ou b) des réimplémentations pour des usages spécifiques, mais tout en conservant l'API (par exemple cuBLAS , qui permet de faire de telles opérations sur des GPUs NVIDIA).

Pour faire communiquer plusieurs machines en même temps, on utilise MPI , un protocole qui étant les fonctionnalités des *sockets* UNIX. La fusion de celui-ci avec LAPACK donne scaLAPACK , une bibliothèque qui, si elle est correctement utilisée, permet de réaliser un calcul en parallèle sur plusieurs milliers de machines en simultanés. Comme un super-calculateur n'est qu'un agrégat de plusieurs milliers de machines,[1] ça tombe bien pour moi

Vu que le nombre de bibliothèques et d'applications de calculs intensifs qui ne sont pas basées sur le couple BLAS/LAPACK est ridiculement faible, je me permets donc d'affirmer que même en 2024, le Fortran 77 est toujours utilisé et que votre vie en est impactée. Deux exemples : la météo (de la bonne mécanique des fluides qui tache bien) et l'IA (qui fait plein d'algèbre linéaire en sous-main). Et ce n'est que les deux premiers exemples qui me viennent en tête.
Le pire ? Bah c'est vieillot (et les fonctions prennent des tonnes d'arguments parce que les structures n'existaient pas en Fortran à l'époque) mais ça marche bien, en fait. La preuve ? Les applications sus-mentionnées.

Le défaut ? Bah à moins d'utiliser un *wrapper* (dont la qualité varie en fonction du langage), un code de calcul intensif, ça s'écrit en donc Fortran, en C, ou en C++ (le Fortran s'interfaçant assez bien avec les deux dernier, même si tout ce passe par référence, en Fortran, donc par pointeur dans les deux autres).[2]

1. C'est un peu plus compliqué que ça, ne le dite pas à mon *system engineer*, mais vous avez l'idée
2. ... Hum, j'oubliais : "Beuuurk, des pointeurs nus".

En particulier, ça ne s'écrit pas en Python (à mon grand regret), ou le coup de l'interprétation reste prohibitif.[3] Ça ne s'écrit pas non plus en Rust (alors que ça devrait), vu que les garanties du Rust sont plutôt inutiles quand le *wrapper* pointe vers quelque chose d'écrit dans un langage (le Fortran, donc) qui ne partage pas lesdites garanties. Et bizarrement, autant pour ré-implémenter le noyau Linux en Rust, il y a du monde, autant pour l'algèbre linéaire, y'a plus personne[4] Et malheureusement, ça ne s'écrit pas en Julia non plus, parce que le *wrapper* pour scaLAPACK y est très basique.

Problème ? J'aime bien la POO,[5] et je trouve qu'une matrice, c'est quand même un très bel *objet* (au sens du "O" de POO). Donc autant je suis d'accord avec @Spacefox quand il nous dit dans son très bon billet que quand on a un marteau, tout ressemble à un clou, autant j'ai l'impression d'être dans un cas ou la POO s'applique bien (une matrice, c'est avant tout un grand tableau de nombres à gérer, donc le planquer dans un objet, c'est quand même confortable). Autrement dit, quand j'ai codé pour la première fois le fameux projet qui m'a ammené à écrire ce billet, je l'ai écrit en C et … J'ai fait de la POO en C. Rien de mal à ça, mais gérer la mémoire, c'est pas fun, en fait. Et *je suis pas venu ici pour souffrir*.[6]

Donc quand j'ai découvert qu'il existait finalement une vraie ré-implémentation de scaLAPACK en C++ qui apporte un coup de jeune au domaine (pas un bête *wrapper*), bah j'ai sauté sur l'occasion. Et c'était le début de mes ennuis.

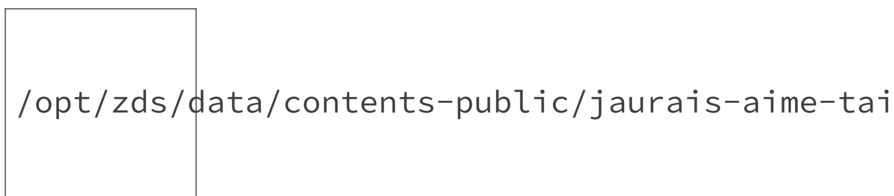


FIGURE 1.1. – Mon état d'esprit.

Mais avant de râler, voici une liste "défauts" (ou pas) que je ne vais pas traiter :

1. L'absence de gestionnaire de packages : bien que la présence d'un gestionnaire de package est un confort certains pour installer des dépendances, la gestion desdites dépendances peut vite être un cauchemar. L'exemple pathologique, c'est `npm` (pour éviter de m'énerver, je vous renverrai vers un autre billet de @SpaceFox, auquel je n'ai pas grand chose de plus à ajouter). Par ailleurs, j'utilise Meson , un CMake en moins énervé et cryptique, qui permet dans une certaine mesure d'installer des dépendances.
2. "La POO, c'est le mal". Plus spécifiquement, deux critiques inhérentes à son implémentation que j'ai pu voir à plusieurs endroits utilisées contre C++. En résumé : a) l'héritage n'est généralement pas une *zero cost abstraction* parce qu'elle oblige a maintenir une liste de fonctions à appeler en fonction des cas quand ce cas ne peut pas être déterminé à la compilation, et b) le fait que la gestion des exceptions introduits à la compilation des morceaux de code pas ouf pour les performances. Bien que ce soit des critiques

3. Je comprends pas pourquoi une bonne partie de l'IA s'écrit en Python, à vrai dire.
4. En vrai, Rust ou pas, réimplémenter correctement (sca)LAPACK sans tuer ces performances, c'est compliqué. C'est aussi ce qui explique que personne peu de monde s'y est frotté
5. Et globalement, je code en Python quand j'ai pas besoin de performances, donc j'ai facilement les réflexes de la POO.
6. En vrai, avec un peu de rigueur, ça se fait très bien, c'est juste … Looooooooooong et chiaaaaaaaaaaaant !

pertinentes, elles me semblent liées à la POO dans C++ donc … Si je choisis la POO dans C++, je choisis de faire avec. De toute façon, je sais d'avance que mon *bottleneck* au niveau performances ne se situera pas là.
3. Il n'y a pas de *garbadge collector* en C++ : ça serait confortable, mais c'est pas indispensable. En tout cas, pas à mon niveau (par contre, j'imagine pas sur des énormes applications).

De manière générale, dans ce billet d'humeur, je souhaite adresser des choix de *design* spécifiques à C++ que **je** trouve critiquables.

2. C++ = C with classes (pun intended)

TL ;DR : trop de rétro-compatibilité, les bonnes fonctionnalités sont inutilement compliquées, donc trop facile d'écrire du mauvais code.

Ce titre se réfère, ici à trois choses :

1. Le fait qu'il fut un temps ou le C++ était réellement transpilé en C ;
2. *Instant nostalgie* : le fait que comme un certain nombre de personnes de ma génération, j'ai appris le C++ avec le tuto de matéo21 sur feu le Site du Zéro. Ce tuto, dans sa première version, ne faisait pas grand chose pour te dissuader que le C++, c'était juste du C avec des classes (ça a ensuite été corrigé quand le tuto C et C++ ont été séparés, mais le mal était fait me concernant) ; et
3. Ce dont je veux parler dans cette section, le fait que le C++ ne fait pas grand-chose pour te forcer à écrire du C++ et qu'il te laisse facilement écrire du mauvais *C with classes*. Tout tient à la volonté du développeur et aux commentaires du type "ton code est bien, mais enfin, tu n'utilises pas cette fonctionnalité de C++20 qui n'est même pas disponible dans ton compilateur, c'est donc du mauvais C++".[1]

[a] Vécu avec `clang-tidy` qui propose des `std::ranges` quand on le pousse un peu trop, vu sur StackOverflow dans d'autres contextes.

Plus sérieusement, j'ai l'impression, mais je me trompe peut-être, que ce dernier point est dû à deux choses : a) C++ est **maladivement** rétro-compatible (même avec C, donc, jusqu'à un certain point), et b) C++ est devenu ultra expressif.

2.1. Rétro-compatibilité, quand tu nous tiens

Parlons tout d'abord du premier point.

Par exemple, la STL contient **toujours** un *subset* de la librairie standard C, qui pour ce que j'en ai vu, contient un joli `namespace std` et quelques surcharges, et c'est tout. Pas de C++ moderne, donc ? Et non : pour prendre un exemple qui me concerne directement, la norme C++

n'a seulement pensé à définir `template<typename T> T sqrt(T nb)` (en mieux, évidement) que depuis C++26 (source).[2]

Et pour du SIMD sur les opérations mathématiques ? Pareil, c'est seulement pour 2026 (je sais, il y a `std::for_each`, j'y viens). Mais bon, je peux pas leur en vouloir, les compilateurs n'utilisent toujours pas AVX par défaut

Pour les nostalgiques du C (coucou @Taurre), ne vous inquiétez pas, `FILE*` est toujours là, prêt à être utilisé. Je le sais parce que `tempfile()` en renvoi toujours un et que personne chez C++ c'est dit que ça serait bien d'en créer une version moderne. StackOverflow ne propose pas franchement mieux (la réponse la plus votée propose quand même d'utiliser des *file descriptors*, quand t'en revient à appeler l'API UNIX, tu sens que t'as raté un truc, surtout quand d'autres langages, il y a au moins un *wrapper*).

Pire encore, dans un monde ou tu es caillassé en place publique pour utiliser des *pointeurs nus* (les fameux), `new` et `malloc()` sont toujours là. Des fois que tu aimerais te tirer des balles dans le pied (mais pas de *kink shaming*). Vous ne voulez pas de pointeurs nus ? Baaaah … ne permettez pas d'en faire par défaut (et, aller, prévoyez une espèce `#pragma unsafe`). Sincèrement, je préfère me faire engueuler par mon compilateur que par des devs sur un forum

Bref, je trouve qu'il est beaucoup trop facile d'écrire du "mauvais C++", ce qui est étonnant dans un langage qui est à ce point obsédé par la *safety* au sens large du terme (à la Rust, mais sans s'en donner les moyens).

Je vais évidement éviter d'utiliser `new`. Ceci dit, l'alternative n'est pas tendre avec le dévellopeur. En fait, je trouve qu'il n'est pas simple d'écrire du bon C++, et c'est mon second point.

2.2. Et expressif, avec ça ?

Dire que C++ est *verbeux* par rapport à d'autres langages me semble raisonnable. On me répondra que cette *verbosité* se justifie par son expressivité, qui permet d'être très précis sur ce qu'on souhaite en tant que développeur. Et je suis d'accord. Par contre, j'ai l'impression que ça vient avec une autre règle, qui est que "le comportement par défaut n'est probablement pas le bon" (qui vient du fait que le comportement par défaut est rétro-compatible, donc probablement incorrect en C++ moderne). Et une troisième qui est basiquement "plus c'est long, plus c'est bon". Et là, à un moment, c'est comme le chocolat, ça devient écœurant.

Comparons les deux lignes suivantes :

```cpp
int* A_new = new int; *A_new = 3;
auto A_unique = std::make_unique<int>(3);
```

Les deux lignes font la même chose, et la seconde est beaucoup plus explicite. Par contre, c'est déjà même plus long, et encore, j'ai utilisé `auto`. Et c'est long *partout* ou on ne peut pas deviner `std::unique_ptr<int>` à partir du conctexte : dans les membres d'une classe, dans les définitions des constructeurs et des fonctions fonctions, et même parfois à l'intérieur : `auto` n'as qu'une utilité restreinte (et c'est normal, d'ailleurs, je ne remet pas ça en question).

2. J'écris ce billet en 2025, rappelons-le.

Toujours aussi long, le *casting* : `(int) 3.4` devient `static_cast<int>(3.4)` : plus d'expressivité, certes, mais plus long. Et quand tu commences à les enchainer, ça commence à déborder de ton écran : `static_cast<double>(n) / static_cast<double>(N) * 100` (paye tes pourcentages) par ci, `static_cast<int>(size)` parce que ta bibliothèque prend des entiers mais que les *containers* te renvoient du `size_t` par là, etc, etc. Rendez-moi `double(n)` !

Et trop, ça devient trop : il y a aussi toute la classe des algorithmes, qui sous couvert d'exhaustivité prennent toujours des itérateurs, et donc 2 arguments par *containers*. Pourquoi ne pas avoir fait une surcharge pour pouvoir écrire `std::sort(x)` plutôt que `std::sort(x.begin(), x.end())` ?[3] Aaaaah, mais non, j'ai pas compris : on va plutôt les mettres dans une AUUUUTRE bibliothèque, `std::ranges` (celle qui est pas dans clang, là), SAAANS déprécier la première. D'ailleurs quand on cherche sur le sujet, on peut constater la détresse des développeurs de clang et de MSVC qui ne parviennent pas à implémenter ladite bibliothèque. Je vous ai dit que le C++, ça se méritais ?

Passons vite fait sur les `#include` : trop aussi. Des choses aussi utiles que `vector`, `move` ou `make_unique`/`make_shared` sont dans 3 *includes* différents. C'était un problème avec C, pourquoi chercher à le reproduire ? À quand un `#include <stl>` ?[4] Même boost ne pousse pas le vice aussi loin, c'est dire !

Et finalement, le pire pour moi au niveau expressivité, c'est la sémantique de mouvement . J'ai beau en comprendre l'intérêt, 4 constructeurs,[5] sérieusement ? Des `std::move` partout, des `std::swap`, allez quoi ? Pourquoi je dois autaaaaant écrire pour faire les choses bien ? Dans une norme ou on a quand défini `T&& x` (ça commence à faire beaucoup de `&`), pourquoi il n'y a pas un sucre syntaxique pour `std::move`, par exemple ? Si le *perfect forwarding* c'est si bien, pourquoi est ce que j'ai l'impression de devoir le mériter ?

> ?
> Je me permets une digression. Est-ce qu'une partie du problème ne viendrait pas que le C++ a gardé le choix du C de tout passer par valeur ? Du peu que j'ai pratiqué le Java (qui est aussi furieusement ~~verbeux~~ expressif), j'ai l'impression que la sémantique de mouvement était une non-question. Ou je me suis trompé ?

2.3. Et donc ...

~~Est ce que les concepteurs de la norme sont payés plus si les fichiers sont plus longs ?~~

Je finirai sur un exemple qui résume toute ma frustration récente :

> Typiquement, j'ai découvert au détour d'un commentaire que `push_back()` dans un `std::vector`, c'est dépassé ; il faut utiliser `emplace_back`. Parce que sémantique de mouvement . Pourquoi ne pas avoir changé le comportement de `push_back` plutôt que de créer une nouvelle fonction ? Bonne question (et, comme d'habitude, la nouvelle fonction est plus longue à écrire). Pourquoi ne pas avoir déprécié `push_back` ? Bonne question.
> Est ce que `clang-tidy` reporte un *warning* ? Peut-être. Pourquoi je dois utiliser `clang-tidy`

3. J'espère qu'à ce prix là, c'est vectorisé, d'ailleurs !
4. Je parie que c'est à cause de l'usage des *templates*. J'y viens
5. Je sais qu'il ne faut pas les implémenter à chaque fois

> pour appendre ça ? Je sais pas.
>
> *Moi, il y a quelques jours.*

… Tout ça me laisse avec la très sincère impression qu'il faut écrire des kilomètres de code pour écrire du *bon* C++. Les cas nominaux ne sont déjà pas simples à écrire, mais le bon C++, ça se mérite.

Soit, mais la conséquence, c'est qu'il est très facile de générer de la dette technique en faisant des cochonneries par simplicité (genre jouer avec `container.data()` alors qu'il faudrait pas, ou à écrire des `for(auto x : container)` à la place de ces fameux `std::ranges`).[6] Pourquoi, si ce n'est que pour des raisons de rétro-compatibilité, des choses *correctes* ne seraient pas simples à écrire ? (genre, `make_shared` ou `make_unique` par défaut, par exemple)

Pour parler de ce que je connais, autant on a pu critiquer le choix de Python de casser la rétro-compatibilité entre Python 2 et 3, autant ils ont ensuite appris de leurs erreurs et déprécient désormais de manière plus ou moins raisonnable fonctionnalités et libraires. Et d'ailleurs, en Python le fait de recevoir des *warnings* de dépréciation dépend de la librairie, pas de la bonne volonté du compilateur, ou pire, de celle du dévellopeur d'activer les bons *flags*. Encore une fois, je préfère me faire engeuler par mon compilateur qui de toute façon ne me donne pas le choix ou m'oblige à activer des flags indiquant clairement que je fais quelque chose de mal (un truc bieeeen long qui te fait sentir bien mal, genre `-funsafe-very-unsafe-allow-new-but-scream-like-hell-when-see-one-because-its-unsafe`), qu'un dévellopeur.

Je pense sincèrement que le C++ gagnerai à sérieusement déprécier (d'après ce post , ça a déjà été le cas, mais c'est mineur) puis a remettre du sucre syntaxique. Ou faire une Kotlin (donc pas un autre langage, mais une version du C++ ou les bonnes choses seraient le défaut, et les mauvaises difficiles à faire).

Bref, enfin avoir un C++ qui ne serait pas un *C with classes and stuffs that are not mandatory anyway*.

3. Et puis viennent les templates

TL ;DR : ”Vous n'avez aucun pouvoir ici, Gandalf le gris.” (Saroumane, dans *La communauté de l'anneau* [le film])

> *i* Aller : on est en 2025, le support de C++20 avance, les IDEs font de l'auto-complétions et se permettent même des suggestions de style (jusqu'ici, la majorité de mes `std::move` m'ont été suggéré, peut être que j'ai pas tout à fait compris la sémantique de mouvement, en fait). `clang-tidy` … Existe (j'y viens) et en cherchant bien, on finit par tomber sur des ressources pour apprendre le C++ vraiment moderne (malheureusement, le *moderne* dépendant de quand a été écrit la ressource, actuellement c'est C++20, mais beaucoup de résultats sont toujours à C++11 voire à C++0x). Donc bon, à condition de se souvenir dans quel *include* est quoi, il y a moyen de commencer à s'amuser. D'ailleurs, merci à @informaticienzero et @mehdidou99 pour leur tuto

6. Et, me concernant, une certaine frustration de faire ”pas assez bien”.

> Sauf que mon projet est un peu particulier, parce que j'aimerai bien pouvoir facilement passer de la double précision à la simple précision (GPUs, tout ça). Et donc là, sans crier gare, je me prends d'envie de faire de la programmation générique (aka mettre des `template <typename output_type>` où ça va bien). Et c'est là où je commence à ~~courir après un poulet pour faire un sacrifice rituel au dieu de la santé mentale~~ sincèrement à remettre mes choix en question. Comprenez : c'est à ce moment-là que j'ai réellement considéré d'écrire ce billet.

<exagération> Les *templates* réussissent là ou les autres ont échoué, en cassant deux choses qui fonctionnait très bien par ailleurs : la compilation, et l'héritage. Rien que ça. </exagération>

3.1. La compilation

La compilation, d'abord. On apprend très vite, en C (et en C++, parce que rétro-compatibilité) à séparer la définition (dans un *header*) et l'implémentation. Même si c'est pas forcément naturel (ça a moins de sens en Python, par exemple, même si ça se fait), on est récompensé par le fait que les implémentations sont généralement compilées une fois, et assemblées à la fin. Autrement dit, quand on a compilé une première fois son projet, le re-compiler après une modification est généralement assez rapide (sauf si c'est dans un *header*, mais vu qu'il s'agit de définitions, c'est plus rare).

Sauf que voilà, en C++ (et dans d'autres langages, genre Java), une *template* se résout à la compilation. La conséquence de ça, c'est que désormais, on ne peut plus séparer définition et implémentation, et tout (!) doit aller dans le *header*.[1] Encore une fois, ça serait tout à fait normal dans d'autres langages, donc pourquoi pas.

Ou est mon problème dans tout ça ? On va faire une petite observation. Voici le temps de compilation du projet original, en C :

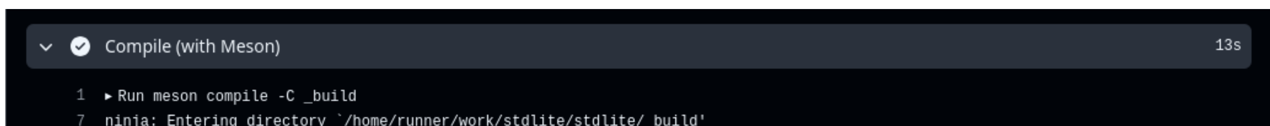
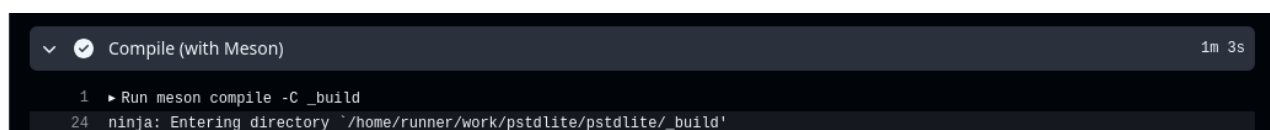


Figure 3.2. – Dans cet exemple précis, j'utilise `gcc`. C'est de toute façon à peu près la même avec `clang`.

Vous aurez reconnu l'interface de *GitHub Actions*. Je ne triche pas : c'est bien le temps de compilation à froid, à partir des sources fraichement téléchargées. Il y a bien entendu une déviation standard, mais elle est de quelques secondes. Et maintenant, voici le temps de compilation, dans les mêmes conditions, du projet C++, où j'ai peut-être implémenté quelque chose comme 10% des fonctionnalités :



1. Sauf si on fait des déclarations explicites en début de `.cpp`. Vu mon cas d'usage, il est probable que je me tourne vers ça.

Figure 3.3. – `g++`, cette fois. `clang++` fait pareil, si jamais.

4 (quatre !) fois plus. Je vais me faire déchirer dans les commentaires, mais sincèrement, je triche pas : les dépendances sont à peu près les mêmes (les options de compilation aussi, autant que faire ce peux) et j'ai pas encore eu besoin de rajouter Boost, par exemple (mais vu les chiffres, je vais y réfléchir à deux fois). Et c'est pas juste à froid : bien entendu, certains fichiers `.cpp` sont toujours compilés, ce qui permet de gagner du temps, mais à chaaaaaaaaaque changement dans une template, le compilateur repasse sur touuuuuuut le projet ou presque, et même avec un PC pas dégeu du tout, je sens bien la différence.

On me répondra que c'est le prix à payer pour avoir un code *safe* et *optimisé*, ce qui n'est pas forcément le cas de mon code C. Ok … Mais quatre fois plus long ?

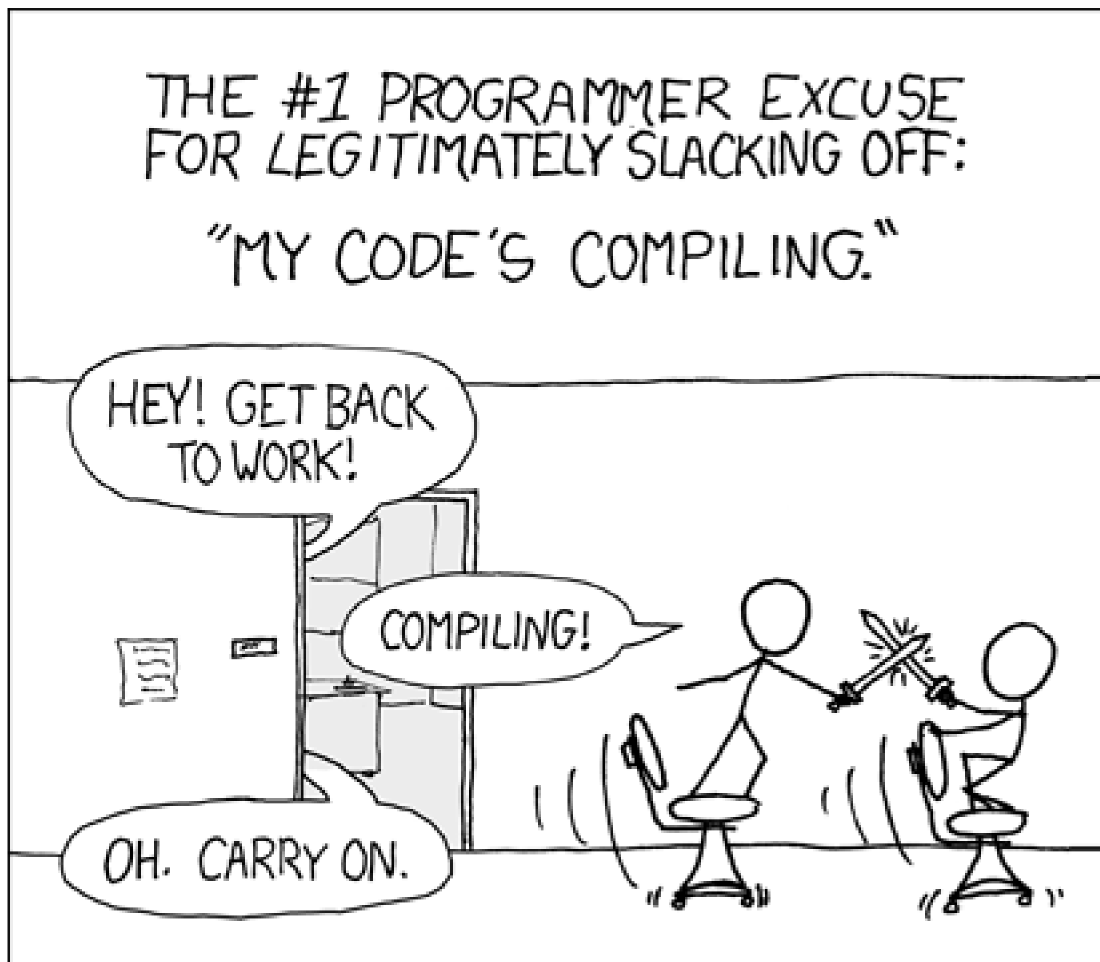


Figure 3.4. – Le nécessaire xkcd de circonstance.

Et gare à l'overdose : ce faisant, le compilateur réaffiche les mêeeeemes *warnings* (parce que oui, j'ai activé des *flags* pour essayer d'améliorer mon code) sur les dépendances sur lesquelles je n'ai paaaaaaaaas la main, d'autant que dès qu'il s'agit de *templates*, `gcc` ce sens obligé d'être très ~~verbeux~~ explicite lui aussi. Par exemple, pour un paramètre non-utilisé dans une dépendance, 35 (!) lignes de *warnings* :

> … Que voici pour la beauté du geste. C'est pas très intéréssant, par contre.

Dans un mode idéal, ce fameux paramètre inutilisé faisant partie de l'implémentation d'une fonction, il aurait dû arrêter de m'embêter à partir du moment ou j'ai eu fini de compiler ladite dépendance. Mais non, il reste là pour me hanter continuellement.[2]

Ça donne envie d'activer les *flags* pour améliorer son code, hein ?

Et si j'ai le malheur de faire une erreur, je suis bon pour scroller frénétiquement, parce qu'elle est enfuie au milieu de … tout ça. Encore heureux que `gcc` la met en rouge !

Bon, soyons honnêtes 30 secondes : j'ai un peu exagéré, parce que `clang` est plus sobre[3]. N'empêche, je ne compte plus le nombre de `device.hh:580:14: warning: unused parameter 'queue' [-Wunused-parameter]` qui sont dans mon log de compilation, ce qui montre bien que les compilateurs repassent encore et encore sur les même *headers*, *ad nauseam*.

L'histoire d'être constructif : cher Père Noël, est ce qu'il y aurai moyen de rajouter un mécanisme qui pré-processerai les *header* un maximum, de manière à pouvoir malgré tout pouvoir résoudre la *template* tout en accélérant la compilation ?

3.2. … Interlude : `clang-tidy` …

Ça va aller vite : `clang-tidy` est un *linter* qui permet de se concentrer sur l'amélioration du code. Il est très complet, mais affreusement lent.

Pourquoi ? Pareil, les *headers* de la STL qui contiennent des tonnes de *templates*, qu'il passe soigneusement en revue, des fois que les personnes chargées d'implémenter la STL soient eux aussi des mauvais programmeurs. Résultat, après 15 (!) bonnes secondes :

```
Suppressed 116461 warnings (116421 in non-user code, 39 NOLINT, 1 with check filters).
Use -header-filter=.* to display errors from all non-system headers. Use -system-headers to display errors from system headers as well.
```

Et ça, c'était UN fichier. En plus, il reporte une erreur parce qu'il trouve pas un *header* que le compilateur arrive manifestement à trouver. Quand je vous dit que le bon C++ ça se mérite.

En pratique, j'utilise donc `cpplint` qui implémente une partie des règles de Google. C'est toujours mieux que rien. Et de temps à autre, je lance `clang-tidy` et je vais prendre un café

---

2. À ce niveau là, c'est du même niveau que *dependabot* qui t'annonce qu'une dépendance d'une de tes dépendances contient une faille de sécurité. Dans les deux cas, ça me fait une belle jambe. Du coup, ça m'en fait deux. Soit.

3. je passe de l'un à l'autre quand GitHub Actions me reporte une erreur spécifique à l'un ou l'autre. Oui, ça arrive, `clang++` a tendance à plus coller à la norme, là ou `gcc` aime bien rajouter ces trucs et être plus permissif.

3.3. … Et l'héritage, donc.

Mais ce qui m'as achevé, c'est ce bout de code. Et plus particulièrement sa ligne 12 :

```
template<typename T>
class A {
  protected:
    T _x;
    T get() { return _x; }
  public:
    A(T x): _x{x} {}
};

template<typename T>
class B: public A<T> {
  using A<T>::_x;
  public:
    B(T x): A<T>(x) {}
    void set(T y) { _x = y; }
    void eq(T y) { return y == A<T>::get(); }
};
```

Ce code ne compile pas sans la ligne 12. Je répète lentement : malgré que `_x` est `protected` (et … juste au dessus), IL. **FAUT.** QUE. JE. DÉCLARE. **EXPLICITEMENT.** Q'IL. EXISTE. DANS. LA. CLASSE. PARENTE. À ce niveau là, je me fout qu'il existe une règle qui justifie l'existence de ce truc.[4] Tout l'intérêt de `protected` est de pouvoir accéder aux éléments de la classe parente à bas cout. Mais non, vous commencez à connaitre le refrain : c'est C++ et c'est une des *feature* les plus utile, ça serait donc dommage qu'elle soit facile à utiliser…

Sincèrement : soit je rajoute ce fameux `using`, soit je crée un *setter* dans la classe parente en plus du *getter*. Dans les deux cas, c'est du code qui serait inutile s'il ne s'agissait pas d'une *template*. Il ne s'agit pas d'être explicite ou de favoriser un bon comportement, là. Il s'agit de … Faire le travail du compilateur à sa place ?

Et quitte à finir en beauté : regardez le nombre de `A<T>::` qu'il faut écrire dans cet exemple. Je vous promet qu'ils sont tous nécessaires (et que `A::` tout seul ne fonctionne pas, mais ça à la limite c'est logique). Manifestement, déclarer `public A<T>` n'est pas suffisant pour qu'il soit évident que je ne parle pas de `std::vector<T>` ou de `A<float>` (à vrai dire, je suis pas assez versé en magie noire pour savoir qu'est ce qui serait valide à la place de `A<T>::` dans cette exemple). Encore des trucs qui seraient inutiles sans *templates*.

3.4. Et donc ?

Franchement, j'ai beau me douter qu'il existe une justification à tout ça, je ne peux pas m'empêcher de penser que le C++ est fait pour ne pas simplifier la vie du développeur. ~~Ce qui lui fait un point commun avec Java.~~[5]

4. C'est comme le 49.3, la règle existe mais c'est une mauvaise règle, voilà.
5. En vrai, je crois pas me souvenir que les *templates* en Java étaient aussi pénibles. Mais c'est loin.

Bon, évidement, j'ai exagéré : l'héritage n'est pas réellement *cassé* et le compilateur fini toujours bien par sortir un programme valide en sortie malgré qu'il tourne en rond sur les mêmes *templates* encore et encore.

Mais est ce que tout ça en vaut la peine ? Franchement, pour une classe qui en pratique ne sera utilisé qu'avec `float` et `double`, je suis à deux doigts d'utiliser des `#define`. Mais bon, c'est comme tout le reste, je garde la foi

Conclusion

Et c'est tout le temps comme ça. Plus je tente des trucs, plus je découvre à quel point mes pré-conceptions du C++ sont à l'opposé de ce qui fait le *bon* C++, et à quel point des choses qui sont simples sur le papier (ou, pour ce que ça vaut, d'autres langages) sont inutilement compliquées à mettre en pratique. C'est un peu décourageant. D'ailleurs, à force de naviguer dans les messages d'erreurs et les suggestions, j'ai fini par tomber sur la FQA, qui est une réponse point par point à (une ancienne version de) la FAQ C++. C'est disponible ici . Certaines critiques sont légitimes, d'autres le sont un peu moins (et tombent parfois dans "la POO, c'est mal" que je mentionnais plus haut). Certaines sont également datées.

Au delà de ça, tout n'est quand même pas toujours difficile en C++, et je retrouve quand même certains *patterns* que j'ai déjà éprouvés. Juste plus long.

Dit autrement, il est toujours facile de faire "du mauvais *X* en *X*" (ou *X* est le langage de votre choix), mais certains langages forcent quand même plus la main au développeur, par exemple en proposant du sucre syntaxique qui aide à faire les choses bien simplement. Un exemple (controversé), c'est les *list-comprehension* en Python. Un autre (tout aussi controversé), c'est l'*autoboxing* en Java. Etc. En C++, le seul truc qui simplifie vraiment les choses, c'est `auto`, à tel point que pour une fois c'est C qui c'est inspiré de C++ et pas l'inverse (promis, après ça j'arrête le sarcasme)

Par contre, le message de ce billet **n'est pas** de dire "*X* est mieux que C++" (d'ailleurs, si je cite Python, c'est juste parce que Python est le langage que je maitrise le plus, Python a évidemment bien des défauts et je pourrai écrire un billet similaire dessus). Chaque langage possède ces qualités et ces défauts, et c'est en âme et conscience que j'ai choisi C++ (voir partie 1). Ça ne sert d'ailleurs à rien de dire "t'aurais dû coder en *X*" : pareil, j'ai fait mon choix et j'ai déjà expliqué pourquoi je n'en changerai pas.

Pour le reste, j'espère sincèrement que ça va s'arranger. Pour C++ et pour moi

D'ailleurs, si vous avez des conseils, je suis preneur. Sincèrement

PS : ce billet a été écrit sur plusieurs jours afin d'éviter que je ne sois trop dans l'émotionnel.

PPS : Des remarques très intéréssantes m'ont été faites dans les commentaires, que je vous invite à lire également

---

Avant de vous quitter, il faut que j'adresse un dernier point, que je ne sais pas comment exprimer convenablement. Donc respirez un grand coup avec moi, et allons-y : la communauté C++ me semble un peu malsaine. C'est hautement subjectif, totalement basée sur mon expérience

personnelle, et évidement, ça ne concerne pas tous les développeurs. Et c'est probablement pas uniquement dans la communauté C++.

Néanmoins, ça va des infos *officielles* (celles qui sont écrites en partie ~~par les créateurs du langage~~ EDIT : pas tout à fait, merci pour l'info dans les commentaires) de C++ qui *trollent* ouvertement :[1]

> C.12 : Don't make data members const or references in a copyable or movable type Reason const and reference data members are not useful in a copyable or movable type, and make such types difficult to use by making them at least partly uncopyable/unmovable **for subtle reasons**.
>
> *La règle C.12 des "C++ core guidelines". Le gras est de moi.*

> Why am I getting errors when my template-derived-class uses a nested type it inherits from its template-base-class ? (...) **This might hurt your head ; better if you sit down.**
>
> *La "C++ Super-FAQ", plus particulièrement là . Le gras est toujours de moi.*

(merci de me rappeler que le C++ c'est compliqué, j'avais pas remarqué. M'enfin si même les concepteurs le disent ...)

... Et donc, vu que c'est compliqué et que personne est d'accord, ça va jusqu'aux développeurs eux-mêmes, qui partent en débat interminables dès qu'il y a une question sur "pourquoi ça marche pas" ou "pourquoi il ne faudrait pas faire comme ça ?" (celle sur lesquelles on tombe quand on se pose une question légitime sur le langage, donc). Un exemple parmi tant d'autres : une discussion sur reddit à propos de la règle C.12 que je mentionne ci-dessus. Après avoir lu un certain nombre de commentaires, je ne sais toujours pas quelle est la bonne solution.

Mais le pire, c'est que souvent ça s'échauffe (parce qu'il s'agit limite de dogmatisme) et que ça fini par se mettre dessus plus ou moins gentiment :

> Congratulations, **you've wasted everyone's time with your extra steps.** This data is const FOR YOU, for a specific use case, but not mine. I need to transform it, adapt it, and convert the buffer name to 1337 c0d3.
>
> *Ce commentaire dans la discussion sus-nommée. Le gras est de moi.*

Et c'est très souvent comme ça. Même la FQA que je cite plus haut tombe parfois (souvent ?) dans ce travers.

Sincèrement, je viens de passer 10 jours à réapprendre le C++, et je ne compte plus le nombre de commentaires désobligeant que j'ai pu lire au cours de mes lectures (avec, évidement, des "pointeurs nus ? Beurk" à toutes les sauces, en français ou en anglais[2]). Calmez-vous, les gens, on est là pour apprendre 😍

Voilà. C'est pas un morceau qui m'a fait plaisir à écrire, mais il fallait que j'évacue ça. Cœurs sur vous, les *devs* C++ 🙂

---

1. La FAQ a un ton très bizarre, d'ailleurs, qui passe du très formel à l'étrangement familier. C'est ... Particulier pour un truc qui se veux officiel.
2. Je pense que c'est devenu une sorte de *running gag*, à force, m'enfin quand même.

Contenu masqué

Contenu masqué n°1 :
… Que voici pour la beauté du geste. C'est pas très intéréssant, par contre.

```
In file included from /opt/slate/include/blas/device_blas.hh:6,
                 from /opt/slate/include/blas.hh:74:
/opt/slate/include/blas/device.hh: In instantiation of ‘void
    blas::device_memcpy(T*, const T*, int64_t, Queue&) [with T =
    double; int64_t = long int]’:
/opt/slate/include/slate/Tile.hh:966:42:   required from ‘void
    slate::Tile<scalar_t>::copyData(slate::Tile<scalar_t>*,
    blas::Queue&, bool) const [with scalar_t = double]’
  966 |             blas::device_memcpy<scalar_t>(
      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
  967 |                 dst_tile->data_, data_, size(), queue );
      |                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/slate/include/slate/BaseMatrix.hh:2497:27:   required from
    ‘void slate::BaseMatrix<scalar_t>::tileCopyDataLayout(slate::T
    ile<src_scalar_t>*, slate::Tile<src_scalar_t>*, blas::Layout,
    bool) [with scalar_t = double]’
 2497 |         src_tile->copyData( dst_tile, *queue, async );
      |         ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/slate/include/slate/BaseMatrix.hh:2704:9:   required from
    ‘void slate::BaseMatrix<scalar_t>::tileGet(int64_t, int64_t,
    int, slate::LayoutConvert, bool, bool, bool) [with scalar_t =
    double; int64_t = long int]’
 2704 |         tileCopyDataLayout( src_tile, dst_tile,
     target_layout, async );
      |         ^~~~~~~~~~~~~~~~~~
/opt/slate/include/slate/BaseMatrix.hh:2868:12:   required from
    ‘void slate::BaseMatrix<scalar_t>::tileGetForWriting(int64_t,
    int64_t, int, slate::LayoutConvert) [with scalar_t = double;
    int64_t = long int]’
 2868 |     tileGet(i, j, device, layout, true, false, false);
      |     ~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/opt/slate/include/slate/BaseMatrix.hh:396:26:   required from
    ‘void slate::BaseMatrix<scalar_t>::tileGetForWriting(int64_t,
    int64_t, slate::LayoutConvert) [with scalar_t = double;
    int64_t = long int]’
  396 |         tileGetForWriting( i, j, HostNum, layout );
      |         ~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~
../include/pstdlite/slate_utils.hpp:278:34:   required from ‘void
    pstdlite::slateutils::blow(slate::HermitianMatrix<scalar_t>&,
    slate::Matrix<scalar_t>&, bool, int) [with scalar_t = double]’
  278 |             dst.tileGetForWriting(i, j,
      slate::LayoutConvert::None);
```

```
23        |
              ~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24 ../tests/tests_system.cpp:50:19:   required from here
25    50 |   slateutils::blow(S, work);
26       |   ~~~~~~~~~~~~~~~~^~~~~~~~~
27 /opt/slate/include/blas/device.hh:580:14: warning: unused
       parameter ‘dst’ [-Wunused-parameter]
28   580 |     T*       dst,
29       |     ~~~~~~~~~^~~
```

Retourner au texte.