# Pourquoi le C++ est-il un langage plus adapté pour les débutants que le C ?

Un adage bien connu dit qu'enseigner, c'est répéter. Ceux qui fréquentent depuis quelque temps le forum C++ de Developpez le savent très bien : on revoit les mêmes discussions revenir régulièrement. Ce billet de blog va tenter d'analyser un peu les arguments concernant l'apprentissage du C++, en se focalisant plus particulièrement sur les difficultés d'utilisation. En particulier le raisonnement suivant, que l'on entend souvent : « il est préférable d'apprendre le C avec le C++ », ainsi que l'affirmation suivante, souvent pas comprise : « le C++ est un meilleur langage pour débuter que le C ».

## Complexité du langage ou de l'apprentissage

Cette partie a été écrite par JolyLoic.

Lorsque l'on compare différents langages, il existe plusieurs notions de complexité, dont la confusion est une des causes qui fait que ces comparaisons s'enflamment.

Il y a au moins :

- la complexité intrinsèque d'un langage, celle qui correspond à l'effort à réaliser pour y devenir un gourou qui connait tous les détails les plus arcanes. Cette complexité peut pratiquement se mesurer au nombre de pages d'un document décrivant le langage. Sur ce plan, le C++ est beaucoup plus complexe que le C, c'est indéniable ;
- la complexité au jour le jour d'un langage, qui est liée au niveau de discipline mentale nécessaire pour coder, ainsi qu'à la puissance des abstractions que l'utilisateur peut mettre en

œuvre pour l'aider. Je suis convaincu que sur ce plan, le C++ est beaucoup plus simple que le C ;
- la complexité d'apprentissage. Elle correspond à la difficulté du parcours d'un étudiant ne connaissant rien et devant apprendre assez du langage pour en devenir professionnel. Je pense là aussi que le C++ est plus simple que le C, mais la réponse est moins tranchée.

La complexité intrinsèque et la complexité au jour le jour sont bien évidemment en opposition, puisque si l'on complexifie un langage, ce n'est pas par pur plaisir masochiste, mais bien pour le rendre plus puissant, donc plus efficace au fur et à mesure. La complexité d'apprentissage est liée aux deux autres, mais aussi à la notion de progressivité.

Cette notion de progressivité de l'apprentissage me semble importante. À part à un haut niveau post-bac, où l'on suppose les étudiants comme des experts dans l'art d'ingurgiter des connaissances (et où à mon avis, on se trompe de méthode, mais c'est un autre sujet), l'ordre d'apprentissage n'a rien à voir avec un ordre logique, qui irait du plus théorique au plus pratique (ce qui consisterait pour notre langue maternelle à faire de la linguistique, de la grammaire et à apprendre le dictionnaire dans l'ordre, avant de dire papa-maman, et dans la programmation à préconiser l'apprentissage du théorème du point fixe, du lambda calculus et autres modélisations mathématiques des langages avant de faire un hello world), ou encore, pour parler d'informatique du plus proche du matériel au plus éloigné (qui consisterait à faire de l'électronique, puis de l'assembleur, puis du C puis du C++).

Non, l'apprentissage se fait de manière pragmatique, en parcourant le sujet à enseigner de manière en apparence erratique, mais avec pour objectifs :

- que chaque pas ne soit pas trop gros, afin de ne perdre personne ;
- que le parcours soit à chaque étape intéressant, afin de motiver les gens.

Un des points de passage de plus haute complexité du C comme du C++ est la notion de pointeurs, avec ses corollaires d'allocation dynamique de la mémoire (mais même sans ça, la notion de pointeur et de valeur pointée pose bien des problèmes). Or, en C, impossible d'écrire un programme un minimum intéressant sans utiliser explicitement des pointeurs. En C++ (ou dans d'autres langages comme java ou C#), l'utilisation de ces pointeurs peut être cachée pendant assez longtemps et on a donc bien plus de liberté pour choisir un parcours le long duquel la courbe d'apprentissage ne présente pas de murs trop importants.

## Apprentissage d'un langage ou de la programmation

Avant d'expliquer et critiquer les phrases indiquées dans l'introduction, il convient de préciser quel est l'objectif de l'apprentissage. Si l'on est dans le cadre de l'apprentissage d'un langage précis, par exemple dans le cadre d'un cours scolaire, dans lequel le langage à utiliser est imposé, alors bien évidemment, la question de savoir si le C++ est plus adapté pour les débutants n'a pas lieu d'être. Nous nous placerons donc dans le cas où la personne qui souhaite apprendre le C ou le C++ est libre de son choix.

Quelle est la différence entre apprendre un langage et apprendre la programmation ? Dans le premier cas, l'attention se porte sur l'apprentissage de la syntaxe du langage, ce que l'on fait de ce langage compte peu finalement. Dans le second cas, l'objectif est de savoir concevoir efficacement un programme permettant de répondre à une problématique donnée. Que ce soit dans le cadre amateur ou professionnel, ce dernier objectif est beaucoup plus intéressant et c'est dans cette optique que se place ce billet de blog.

Précisons ce que signifie ce que l'on entend par « efficacement ». Dans la création d'un programme, on pense en général à la phase de codage et l'on comprend facilement qu'entre deux langages, si le premier permet d'obtenir le même résultat que le second, mais en utilisant 10 fois moins de lignes de code, alors il sera beaucoup plus efficace d'écrire un programme dans ce langage. Cependant, la création d'une application ne se limite pas à écrire du code et l'expérience montre que l'on passe

beaucoup de temps sur d'autres tâches : la conception du programme, la correction des bugs, la validation du programme (vérifier que le programme donne le résultat correct). Le terme « efficacement » doit donc être considéré en analysant le temps global de développement et pas simplement la phase de codage.

### Conception impérative et objet

La comparaison entre les différents paradigmes de programmation, leurs intérêts et inconvénients respectifs, leurs efficacités, n'est pas le propos de cet article, je ne vais donc pas m'attarder sur ce point. Signalons simplement que la raison même de la création de la programmation objet est de faciliter la création d'application, on peut s'attendre à ce que la conception en C++ utilisant des objets (correctement…) soit plus efficace qu'en C.

### Erreurs à la compilation et l'exécution

## Problématique

Probablement tous les développeurs expérimentés ont déjà eu le problème suivant, d'un bug dont l'origine était indéterminée et qui a nécessité plusieurs heures de recherche pour le corriger. Partant de ce constat, on peut tirer la règle suivante : plus la détection des erreurs est précoce, plus il sera facile de les corriger. Autrement dit, les erreurs survenant à la compilation seront beaucoup plus simples à corriger que les erreurs survenant lors de l'exécution (et l'une des forces des templates en C++ est de pouvoir justement réaliser beaucoup de vérifications à la compilation, sans surcoût à l'exécution, mais cela sort du sujet de ce blog).

Pour être plus concret, voyons un exemple avec la vérification de la constance des variables (« const-correctness »). Le mot-clé `const`, utilisé pour le passage de paramètres dans les fonctions, est un contrat que

passe l'utilisateur d'une fonction avec le créateur de cette fonction. Il permet de dire : cette variable ne sera pas modifiée dans cette fonction.

En théorie…

Voyons ce que cela donne en pratique. Imaginons une fonction `foo()` prenant une chaîne de caractères constante et pouvant modifier cette chaîne. Le respect de cette constance implique donc que la fonction ne doit pas travailler sur la chaîne passée en argument de la fonction (potentiellement non modifiable), mais sur une copie modifiable de cette chaîne.

## Le cas du C

Voyons par exemple le code suivant, qui convertit un simple const `char*` en `char*` (donc qui supprime le mot-clé `const`) :

```
void foo(const char* cstr) {
    char* str = cstr;
    bar(str); // modifie potentiellement str
}

int main()
{
   foo("une chaîne de caractères");
   return 0;
}
```

Lorsque l'on compile avec un compilateur C, celui-ci ne donne qu'un simple avertissement sur le fait que l'on modifie une variable constante en non constante. Malheureusement, les *warnings* sont souvent ignorés par les débutants et le code ne sera pas corrigé. C'est une mauvaise pratique largement répandue et c'est souvent une bonne chose d'activer l'option `-Werror` dans gcc, pour que les *warnings* deviennent des erreurs). On va pouvoir observer des comportements différents en fonction du code de `bar()`. Si la fonction `bar()` ne modifie pas la chaîne, par exemple avec le code suivant, alors il n'y aura pas de problèmes, le

programme s'exécutera sans erreurs :

```
void bar(char* str) {
    printf("%s" , str);
}
```

En revanche, si la fonction bar() modifie la chaîne, on va obtenir une erreur à l'exécution. Par exemple, le code suivant :

```
void bar(char* str) {
    str[0] = 'X'; // on suppose que str existe et contient
au moins un caractère
    printf("%s" , str);
}
```

Ce code va générer l'erreur suivante :

```
result: runtime error  time: 0.02s  memory: 1676 kB  signal:
11 (SIGSEGV)
```

Ce message d'erreur ne nous apprend pas grand-chose. Il indique simplement qu'il y a un problème d'accès sur une mémoire invalide, mais n'indique pas que l'on ne respecte pas le const-correctness et ne précise pas la ligne de code où se situe le problème. On imagine bien alors que dans un programme contenant plusieurs milliers de lignes de code, il sera difficile de trouver l'erreur.

Même si le premier code de `bar()` ne provoque pas d'erreurs à l'exécution, il n'en reste pas moins que le code de `foo()` est la source du problème. Cependant, il n'y a pas de validation réalisée par le compilateur et il est facilement possible dans ces conditions d'avoir du code potentiellement problématique qui fonctionne... quelque temps. Et à l'occasion d'une modification mineure, ce code problématique va déclencher une erreur d'exécution et la recherche de l'erreur sera fastidieuse. On dit souvent, de façon humoristique, qu'un code problématique et qui fonctionne, « que le programme tombe en marche ».

Sans cette vérification à la compilation, il est facile de se tromper, par exemple en écrivant le nom des fonctions. Supposons que l'on a deux

fonctions, une version qui ne modifie pas la chaîne et utilise donc directement la chaîne passée en argument ; une version qui modifie la chaîne, par exemple en appelant `realloc()`. Normalement, il faudrait récupérer le pointeur retourné par la fonction, mais comme on appelle la mauvaise fonction, on oublie de le faire :

```
// version constante
// s'utilise de la façon suivante : c_bar(str);
void c_bar(const char* str) {
    do_something(str);
}

// version non constante
// s'utilise de la façon suivante : str = c_bar(str);
char* bar(char* str) {
    char* str2 = realloc(str, strlen(str)*2);
    if (str2) { str = str2; } else { printf("Error de
réallocation"); }
    do_something(str);
    return str;
}

void foo(const char* cstr) {
    // allocation et copie
    size_t len = strlen(cstr) + 1;
    char* str = malloc(len);
    strcpy(str, cstr);

    // appel de bar() au lieu de c_bar()
    bar(str); // problème !

    // libération
    free(str);
}
```

En C, lors de la compilation, il y aura un simple *warning*, qui est facilement manqué. À l'exécution, le message d'erreur n'indique pas l'origine du problème. Plus que le fait que le code à écrire est plus long qu'en C++, il y aura une perte de temps très importante, plusieurs heures voire plusieurs jours, pour trouver l'origine des erreurs.

## Le cas du C++

En C++, la situation sera beaucoup plus simple : ce code génère une erreur suivante, indiquant que l'on essaie de convertir une variable constante en variable non constante :

```
prog.cpp: In function 'void foo(const char*)':
prog.cpp:5: error: invalid conversion from 'const char*' to
'char*'
```

L'erreur est simple (profitons-en, ce n'est pas toujours le cas en C++) et parfaitement localisée, la correction du code sera directe et rapide.

### Détection des fuites mémoires

## Problématique

Commençons par un rappel : qu'est-ce qu'une fuite mémoire ? Lorsque l'on alloue une variable dynamique sur le Tas, le bloc mémoire alloué doit être détruit à un moment donné, quand on en a plus besoin, pour éviter de saturer la mémoire disponible. L'adresse mémoire est sauvegardée dans un pointeur et il suffit d'appeler la fonction adéquate pour libérer la mémoire, en fonction de la fonction utilisée pour l'allouer : `free()` avec `malloc` ou `calloc`, `delete` avec `new`, `delete[]` avec `new[]`. Une fuite mémoire survient lorsque cette libération n'est pas réalisée, par exemple si l'on perd le pointeur, si l'on n'appelle pas la fonction de libération adéquate, voire qu'on n'appelle pas du tout cette fonction.

La difficulté avec les fuites mémoires est que c'est un problème qui survient exclusivement à l'exécution, on retombe sur le souci décrit dans le chapitre précédent : la localisation pourra être complexe et longue. Il existe des outils d'analyse des allocations, tel que valgrind, qui permet de rechercher ce type de problème (et il est recommandé de l'utiliser

régulièrement), mais s'il est possible de minimiser les risques lors de la phase de codage, cela pourra améliorer l'efficacité.

## Le cas du C

Revenons sur le code précédent de `foo()` et corrigeons le problème de const-correctness en copiant la chaîne constante dans une nouvelle chaîne non constante. Pour simplifier le code, il est possible d'écrire deux fonctions d'aide pour gérer la mémoire : `createString()`, pour allouer la mémoire et copier la chaîne, et `destroyString()`, pour libérer la mémoire. Une implémentation de ces fonctions peut-être la suivante :

```
char* createString(const char* cstr) {
    size_t len = strlen(cstr) + 1;
    char* str = malloc(len);
    strcpy(str, cstr);
    return str;
}

void destroyString(char* str) {
    free(str)
}
```

Le code de la fonction `foo()` est modifié de la façon suivante :

```
void foo(const char* cstr) {
    char* str = createString(cstr);
    bar(str); // modifie potentiellement str
    destroyString(str);
}

int main()
{
   foo("une chaîne de caractères");
   return 0;
}
```

Dans un code aussi simple, il est peu probable que le développeur oublie

de libérer la mémoire à la fin de la fonction `foo()`. Par contre, dans une situation plus complexe, dans laquelle la création de la chaîne est réalisée à un endroit du programme, l'utilisation à d'autres endroits et la libération encore ailleurs, il peut être difficile de savoir si la mémoire est correctement libérée.

De plus, le code précédent n'apporte pas de garanties fortes sur la libération. Par exemple, si le code entre `createString()` et `destroyString()` contient des `return`, la fonction `destroy()` ne sera pas appelée. Il est de la responsabilité du développeur de penser à appeler `destroy()` avant chaque `return`, ce qui peut être facilement oublié. Si la fonction manipule plusieurs objets dynamiques, il faut savoir à tout moment quelles sont les variables allouées et celles qui ne le sont pas.

```
void foo(const char* cstr) {
    char* str = createString(cstr);
    if (test)
        return; // problème !
    destroyString(str);
}
```

Plus pernicieux, le code peut allouer un nouveau bloc mémoire et attribuer l'adresse dans un pointeur, ce qui invalidera celui-ci et provoquera une fuite.

```
void foo(const char* cstr) {
    char* str = createString(cstr);
    if (test)
        str = createString("un nouvelle chaîne"); //
problème !
    destroyString(str);
}
```

En général, on trouve deux profils sur le forum : ceux qui croient qui ne feront jamais d'erreurs aussi grossières et ceux qui savent qu'il peut arriver de faire de telles erreurs. En général aussi, les premiers viennent sur le forum parce qu'ils ont des problèmes avec leur code, les seconds viennent pour aider les premiers.

Choisissez votre camp. Et si vous faites partie du premier groupe, posez-vous peut-être des questions...

## Le cas du C++

La situation pourrait sembler plus compliquée en C++. En effet, le C++ ajoute un système de gestion d'erreurs, absent du C : les exceptions. Ce système peut briser l'ordre séquentiel d'exécution du code et il peut être difficile de garantir la robustesse du code.

Cependant, le C++ propose également un système qui permet de transférer la responsabilité de la libération de la mémoire. En effet, grâce au RAII (acquisition de ressources à l'initialisation), la libération est sous la responsabilité d'une classe au lieu d'être sous celle de l'utilisateur. Le RAII est un principe qui peut être implémenté par les concepteurs d'une classe utilisant des ressources dynamiques, mais pour simplifier le travail, la bibliothèque standard STL fournit déjà de nombreuses classes encapsulant le RAII : `std::string` pour les chaînes, `std::vector` pour les tableaux, `std::shared_ptr` pour les pointeurs, etc.

Voyons par exemple une fonction utilisant une chaîne de caractères en C++ :

```
void foo(const char* cstr) {
    std::string str(cstr);
    // code quelconque pouvant modifier str
}
```

Ce code, en plus du fait d'être beaucoup plus simple, nous garantit que, quelles que soient les lignes de code contenues dans la fonction, la chaîne de caractères `str` sera correctement libérée lorsque l'on sort de la fonction, quelle que soit la façon dont on sort de cette fonction (par `return`, par exception). L'attention du développeur peut se focaliser sur le problème à résoudre et plus sur les détails de gestion de la mémoire.

Pour en savoir plus sur le RAII, vous pouvez lire cet article : Pourquoi le RAII est-il fondamental en C++ ?.

## Conclusion

D'autres arguments peuvent être abordés pour justifier l'apprentissage du C avant le C++ (on parle de l'approche historique de l'apprentissage du C++), mais une analyse peut facilement les mettre à mal. Par exemple, l'argument selon lequel l'apprentissage bas-niveau en C permet de mieux comprendre ensuite la programmation plus haut-niveau en C++. Outre le fait que l'apprentissage de la gestion manuelle de la mémoire n'est pas un prérequis pour comprendre les notions d'allocation dynamique, de Pile et de Tas, la pratique montre que ceux qui abordent l'apprentissage par cette approche se perdent dans les détails et le débogage et ont plus de mal à assimiler les concepts plus haut-niveau (conception objet, encapsulation, principes de programmations objet, etc.)

On pourrait croire que mes propos sont exagérés, qu'il est rare de faire des erreurs aussi grossières, que cela n'empêche pas de comprendre les concepts de programmation objet. En pratique, on voit régulièrement sur le forum les étudiants faire les mêmes erreurs d'apprentissage (un exemple récent), mais ce n'est pas une fatalité et il est possible de corriger notre façon d'enseigner la programmation en C++.

## Remerciements

Merci à JolyLoic pour la partie qu'il a rédigée et ses remarques, à Winjerome, Kalith et LittleWhite pour leur relecture et leurs remarques, à Winjerome pour ses corrections orthographiques.