INFO0004-2: Retour sur les projets

Sami BEN MARIEM, Cyril SOLDANI, Pr Laurent MATHY

25 juin 2020

Table des matières

1 Introduction

Les sections suivantes comprennent une série de recommandation pour améliorer votre code, sur base des erreurs ou mauvaises pratiques relevées dans vos projets. Les recommandations sont classées en trois grandes catégories : les conseils pour du code plus facile à utiliser (plus flexible) et à maintenir (plus lisible), les conseils pour du code plus robuste (et sécurisé), et les conseils pour du code plus efficace.

Notez que pour conserver une taille raisonnable, ce document prends quelques raccourcis. Certaines recommandations illustrées sur un exemple donné (*e.g.* fonctions), s'appliquent plus généralement (*e.g.* aussi pour les méthodes). Il faut prendre ces recommandations à l'esprit plutôt qu'à la lettre. Tout est par ailleurs question de compromis. Ces recommandations se basent sur les défauts les plus souvent observés dans vos projets, mais presque aucune ne devrait être poussée aveuglément à l'extrême. "Foolish Consistency is the Hobgoblin of Little Minds" (Guido van Rossum, créateur de Python).

Enfin, notez également que certains des points discutés ici sont déjà discutés, souvent plus en détail, **dans le cours** !

2 Du code facile à utiliser et facile à maintenir

2.1 Écrire des fonctions courtes

Sauf exceptions (*e.g.* un `switch` avec beaucoup de cas), essayez de limiter la taille de vos fonctions à une cinquantaine de lignes au maximum. Si vos fonctions sont trop longues, il y de grandes chances qu'elles fassent trop de choses différentes (violant ainsi le SRP). Cela rend votre architecture fragile au changement, et limite le potentiel de réutilisation. Même si votre fonction n'effectue qu'une seule tâche (complexe), essayez de décomposer celle-ci en sous-tâches.

Garder les fonctions courtes est aussi bon pour la lisibilité, car le lecteur peut garder l'ensemble de la fonction à l'écran. Il peut aussi voir clairement les étapes principales sans être noyé dans les détails.

2.2 Écrire des fonctions simples

Écrivez des fonctions *simples*, à la complexité cyclomatique faible. Outre les bénéfices associés à des fonctions plus *courtes* (voir 2.1), réduire la complexité des fonctions tend à réduire le nombre de bugs dans le code.

Des fonctions simples sont aussi souvent, assez logiquement, plus facile à comprendre.

"Ce qui se conçoit bien s'énonce clairement" disait Boileau. Cet adage s'applique aussi en programmation. Si vos fonctions sont complexes, demandez-vous d'abord si votre compréhension du problème ne devrait pas être améliorée.

Enfin, conserver un code simple permet au compilateur de suivre, et de vous donner des messages pertinents. *E.g.* dans le code suivant, `clang++` génère un avertissement à la ligne 8 indiquant que `x` est peut-être utilisée avant d'être initialisée dans l'exemple de gauche, mais pas dans celui de droite (parfaitement équivalent, mais plus simple) :

```
1 int x;
2 for (int i = 0; i < 8; ++i) {
3   if (i == 0) {
4     x = 21;
5   }
6   // Other code using x
7 }
8 int y = x * 2;
```

```
1 int x = 21;
2 for (int i = 0; i < 8; ++i) {
3
4
5
6   // Other code using x
7 }
8 int y = x * 2;
```

Voir aussi le point 2.19 pour un des moyen de rendre votre code plus simple.

2.3 Don't Repeat Yourself (DRY)

Si à ce stade vous ne savez pas encore pourquoi vous ne devez pas vous répéter dans votre code, vous avez un sérieux problème. Je vous invite d'urgence à lire un livre comme "The Pragmatic Programmer" (Andy Hunt et Dave Thomas), ou "Clean Code" (Robert Cecil Martin).

Le code redondant est plus long à écrire, nuit à la lisibilité, à l'efficacité (*code bloat*), et à la robustesse (vous finirez *nécessairement* par faire un changement dans une partie du code en oubliant les autres endroits où vous auriez dû faire la même modification).

Du code redondant n'apparaît pas nécessairement par copier-coller. Il apparaît aussi au fur et à mesure du développement, quand plusieurs parties du code se révèlent a posteriori devoir faire la même chose. Vous devez donc être constamment attentif aux opportunités de factoriser du code (ce qui est plus facile si vous gardez des fonctions courtes (2.1) et simples (2.2), car le code à réutiliser sera déjà probablement dans sa propre fonction).

Être attentif au fur et à mesure n'est pas suffisant. Vous devez aussi périodiquement prendre du recul, regarder votre code et vous demander s'il y a des choses redondantes qui pourrait être factorisées.

2.4 Keep It Simple Stupid (KISS)

En général, implémentez la solution la plus simple qui fonctionne.

Si vous avez un problème de performance, utilisez un algorithme plus efficace plutôt que des *trucs*, qui vous paraissent malin aujourd'hui mais vous paraîtront obscurs demain, et risquent de rendre l'évolution du code plus difficile.

En particulier, éviter de grouper des modules ou fonctions qui devraient être distinct(e)s au motif que ça permet de faire une économie de calcul minimale. Chaque module/fonction doit avoir une responsabilité unique. Éviter aussi les encodages trop sophistiqués de vos données pour permettre plusieurs usage différents : mieux vaut deux tableaux clairs et simple à utiliser plutôt qu'un seul demandant à être utilisé avec grande précaution pour ne pas commettre d'erreur.

2.5 Évitez les variables indexées

Évitez d'indexer manuellement des variables : `var1`, `var2`, *etc.* (ou `one`, `two`, *etc.*).

En général, cela indique soit de mauvais noms si ces variables désignent des choses différentes, soit des répétitions (et donc une violation du point 2.3). En effet, il est difficile d'écrire du code générique faisant intervenir plusieurs variables différentes. Pour le compilateur, `var1` est aussi différente de `var2` que `trucmuche` de `tartempion`.

Utilisez des conteneurs à la place, même s'il n'y a que quelques éléments.

2.6 Évitez les valeurs inutilisées

Assignez à une variable une valeur qui n'est jamais lue n'est pas efficace. De manière plus importante, ça peut indiquer un problème logique dans votre code (après tout, si vous lui donnez une valeur, il doit bien y avoir une raison).

Vous pourriez être tenté d'initialiser une variable avec une valeur *sentinelle* en attendant une initialisation ultérieure si la variable doit être utilisée. L'inconvénient de cette approche est que le compilateur ne pourra plus vous avertir que vous utilisez potentiellement une variable non-initialisée. Pour lui, la valeur sentinelle est une valeur comme une autre. Par ailleurs, cela indique peut-être un problème de portée (voir point 2.7).

Enfin, si vous affectez une valeur à votre variable, le compilateur considérera que celle-ci est utilisée, et ne vous avertira pas si cette variable n'est en fait pas ou plus nécessaire.

2.7 Déclarer les variables près de leur utilisation

C++ ne vous limite pas à déclarer les variables en début de fonction. Déclarez-les juste avant leur première utilisation. Le code sera plus lisible (les utilisations seront sur le même écran que la déclaration), et vous pourrez plus souvent les initialiser directement avec une valeur pertinente (voir point 2.6).

En outre, limiter la portée des variables peut permettre la génération d'un code plus efficace.

2.8 Évitez les espaces en fin de ligne

Outre le fait qu'ils ne servent à rien, des espaces en fin de ligne peuvent donner lieu à des problèmes avec le système de contrôle de version (*e.g.* `git`). Une fois ces espaces enregistrés dans `git`, une modification ultérieure du fichier dans un autre éditeur qui supprime automatiquement ces espaces (ce qui est le cas de beaucoup d'IDEs) va entraîner une série de modifications dans le fichier sur des lignes que vous n'avez pas touchées. Ça rend la lecture des `diff` difficile (d'autant plus que les espaces sont par définition invisibles), et peut induire des conflits artificiels en cas de *merge*. Ce n'est pas très critique quand vous travaillez seul, mais devient un vrai problème quand vous travaillez en équipe.

Configurez une fois pour toute votre éditeur pour afficher ou supprimer ces espaces. Un outil comme `clang-format` peut aussi être utilisé. Enfin, prenez l'habitude de faire un `git diff` avant un `git add`. `git` vous affichera en rouge les espaces posant problème.

2.9 Créez des fichiers d'entête autoportants

Pour pouvoir utilisez un de vos modules facilement, il faut qu'il suffise d'inclure son fichier d'entête, sans devoir inclure d'autre fichiers d'entête. Si l'inclusion de votre entête requiert l'inclusion préalable d'autres fichiers d'entête (*e.g.* vous utilisez `std::vector` mais n'incluez pas `<vector>` de sorte que l'utilisateur doit inclure lui-même `<vector>` avant d'inclure votre entête), le code du client sera :

- lent à écrire, avec des aller-retours entre la documentation, ou les message d'erreur du compilateur, et le code ;
- moins synthétique, avec de nombreuses inclusions qui noient les dépendances importantes dans le bruit de ce qui leur est nécessaire ;
- plus fragile : simplement changer l'ordre des fichiers d'entête (*e.g.* classés alphabétiquement par l'IDE) peut changer un code qui compile en un code qui ne compile pas. Idem si une

des autres dépendances change et cesse d'inclure les éléments qui étaient aussi nécessaires (mais non inclus) par votre module.

Assurez-vous donc que vos fichier d'entête incluent tout ce qui leur est nécessaire (voir point 2.10).

2.10 Un fichier d'implémentation doit inclure son entête en premier

Dans vos fichiers d'implémentation, incluez le fichier d'entête correspondant en premier. Si ce dernier n'est pas autoportant (voir point 2.9), vous aurez une erreur de compilation en compilant votre fichier d'implémentation.

2.11 N'utilisez pas using dans les entêtes

N'utilisez pas `using` dans vos fichiers d'entête. Ça pollue le `namespace` de l'appelant avec de nouveaux noms dans la portée lexicale principale, qui peuvent amener des conflits de noms avec le code de l'utilisateur, ou celui d'autres modules également inclus par l'utilisateur dans le même fichier d'implémentation (avec des messages d'erreur particulièrement obscurs). En somme, utiliser `using` dans un fichier d'entête revient à supprimer tout l'avantage d'avoir des `namespace`s en C++.

2.12 Protégez vos fichiers d'entête contre l'inclusion multiple

Vos fichiers d'entête sont susceptibles d'être inclus plusieurs fois par un même fichier d'implémentation (*e.g.* parce que votre module C est inclus à la fois par un module A et un module B, tous deux inclus par l'utilisateur).

Certaines déclarations (*e.g.* de classe) vont donner lieu à une erreur de compilation si elles sont répétées.

Aussi, protégez vos fichiers d'entête contre l'inclusion multiple à l'aide d'une garde d'inclusion :

```
1 #ifndef MY_MODULE_HH
2 #define MY_MODULE_HH
3
4 // Header contents ...
5
6 #endif // MY_MODULE_HH
```

2.13 N'exposez que votre API public

Un module doit avoir un interface clair, dont toutes les fonctions (et méthodes publiques) doivent être directement utiles à l'utilisateur. Cet interface ne devrait pas dépendre des détails d'implémentation, vous autorisant à changer ceux-ci par la suite sans nécessiter de modification du code des utilisateurs.

En particulier, les fonctions auxiliaires nécessaires à l'implémentation de votre module, mais qui ne sont pas supposées être appelées directement par l'utilisateur, ne doivent pas apparaître dans votre fichier d'entête. Placez ces fonctions uniquement dans votre fichier d'implémentation, et rendez-les `static` (voir point 4.1).

De la même manière, n'incluez dans votre fichier d'entête que les autres fichiers d'entête nécessaire à son interface (*e.g.* types nécessaires pour les prototypes de ses fonctions publiques). Les autres fichiers d'entête uniquement nécessaires à son implémentation doivent allez dans le fichier d'implémentation.

2.14 Placez la documentation de l'API public dans le fichier d'entête

Les fichiers d'entête de vos modules sont le premier endroit où l'utilisateur va regarder ce qui est disponible, et chercher la documentation. Les fichiers d'entête d'une librairie sont également toujours disponibles, même si la librairie elle-même est déjà compilée (*e.g.* sous la forme d'un fichier `.so`) et que le code de l'implémentation n'est pas disponible. Enfin, de nombreux IDE peuvent extraire automatiquement la documentation des fichiers d'entête pour la présenter dynamiquement à l'utilisateur.

Il est donc préférable de commenter vos interfaces dans les fichiers d'entête plutôt que dans les fichiers d'implémentation, et de ne garder dans ces derniers que les commentaires propres aux détails de l'implémentation.

2.15 Ne réinventez pas la roue

Utilisez la STL autant que possible. Utiliser `std::count_if` ou `std::accumulate` à la place d'une boucle manuelle ne vous fera gagner que 2 lignes, mais :

- ces algorithmes nommés rendent l'intention du code plus explicite et donc le code plus lisible (du moins, pour ceux qui connaissent bien la STL) ;
- ça diminue le risque de bug ;
- ces lignes gagnées ici et là vont s'additionner pour donner des fonctions plus courtes (voir point 2.1) ;
- ces fonctions sont souvent plus efficaces qu'une boucle manuelle (*e.g.* elles peuvent être implémentées avec des instructions de vectorisation comme SIMD).

2.16 Évitez les identifiants réservés

En C++, les identifiants commençant par deux *underscores* ou un underscore suivi d'une majuscule sont *réservés*. L'utilisateur ne peut pas créer de tel identifiants, juste utiliser ceux déjà existants (*e.g.* `__cplusplus`).

Ça vaut aussi pour les macros. En particulier, n'utilisez pas d'underscore en position préfixe dans vos gardes d'inclusion :

```
1 #ifndef _MY_MODULE_HH_ // Invalid C++
```

2.17 Utilisez des énumérations pour les cas

Il est généralement préférable de définir des énumérations plutôt que d'utiliser quelques nombres ou caractères pour identifier plusieurs cas. *E.g.* utilisez

```
1 enum Suit { SPADES, HEARTS, DIAMONDS, CLUBS };
```

plutôt que `'s'`, `'h'`, `'d'` et `'c'` pour identifier la couleur d'une carte.

Outre le côté plus explicite, ça permet au compilateur de détecter si vous avez oublié de gérer un des cas dans un `switch` (n'utilisez pas de cas `default`), et conduit à du code plus efficace.

2.18 Évitez les long if-else-if

Évitez les longues chaînes de `if-else-if`, qui ne sont ni très lisibles, ni très efficaces (nombreux tests et branchements).

Si chaque branche gère un cas particulier, essayer d'utiliser plutôt un `switch` (en ajoutant une `enum` au besoin). Il se peut aussi que ce soit révélateur d'un code trop complexe (2.2) ou d'une mauvais structure.

En particulier, s'il s'agit d'un simple lookup, il pourrait être préférable d'utiliser une structure de donnée à la place (*e.g.* un `std::map`).

2.19 Optimisez vos conditions booléennes

Quand un problème a plusieurs cas qui dépendent de façon complexe de plusieurs facteurs, prenez le temps de réfléchir au problème avant de vous lancer dans le code. Dessiner un arbre de décision, faire appel à des tables de vérités et autres outils théoriques (*e.g.* table de Karnaugh) peut vous aidez à produire un code beaucoup plus court et plus clair.

En particulier, si vous avez du code répété à l'intérieur de plusieurs conditionnels, il y a fort à parier que vos conditions peuvent être simplifiées pour ne plus avoir le code qu'une seule fois. *E.g.*, les deux code suivants sont parfaitement équivalent, mais celui de droite est beaucoup plus clair, et plus efficace.

```
if (a) {
  if (c) {
    foo();
  }
} else if (b) {
  if (c) {
    foo();
  } else {
    bar();
  }
}
if ((a && !c) || (!a && !b)) {
  bar();
}
```

```
if ((a || b) && c) {
  foo();
} else {
  bar();
}
```

Voir aussi 2.22.

2.20 Évitez les nombres magiques

Évitez de faire appel à des *nombres magiques*, *i.e.* des valeurs numériques spécifiques, dans votre code. Utilisez des constantes nommées à la place.

Si certains valeurs constantes sont calculées à partir d'autres valeurs constantes, utilisez une fonction `constexpr` pour faire le calcul rendra la relation explicite dans le code, et le code plus robuste (si une constant primaire est modifiée, les constantes qui en dépendent le seront aussi **automatiquement**).

2.21 Choisissez des noms descriptifs adaptés au domaine

Évitez les noms génériques comme `data`, `tmp`, ou `arg` (à part dans du code très générique).

Préférez des noms descriptifs qui indiquent au lecteur de quoi il s'agit (pour les variables et types), ou quelle est l'action effectuée (pour les fonctions). Choisissez ces noms spécifiques au *domaine métier*, *i.e.* au problème que vous êtes en train de résoudre, pour faciliter le parallèle entre le code et le problème qu'il résout.

2.22 Utilisez les expressions booléennes directement

Plutôt que d'utiliser des conditionnels pour calculer des expressions booléennes, utilisez des expressions booléennes directement. *E.g.* préférez le code de droite au code de gauche :

```
if (cond && other_cond) {
  return true;
} else {
  return false;
}
```

```
return cond && other_cond;
```

Voir aussi le point 2.19.

2.23 Préférez des constantes typées aux macros

Écrivez

```
static const size_t N = 8;
```

plutôt que

```
#define N 8
```

- Les macros ont une portée lexicale globale et s'appliquent partout, alors que les constantes peuvent être restreintes à un namespace, une classe, *etc*.
- Comme les macros sont remplacées par le C preprocessor (CPP) avant l'appel au compilateur proprement dit, les messages d'erreurs sont bien plus intelligibles quand vous utilisez des constantes plutôt que des macros.
- Le type associé à la constante permet au compilateur de détecter des erreurs qui ne seraient pas détectées avec une macro.
- Il est possible de prendre l'adresse d'une constante, mais pas d'une macro.
- Les constantes n'ont aucun coût supplémentaire (tant que vous n'en prenez pas l'adresse).
- Le support pour le C preprocessor pourrait être totalement supprimé des futures versions de C++.

2.24 Utiliser explicit pour éviter les conversions hasardeuses

Un constructeur avec un seul argument, ou dont les arguments suivant le premier ont une valeur par défaut, devient aussi par défaut un opérateur de conversion implicite depuis le type de l'argument vers le type de l'objet, *i.e.*

```
class MyType {
  MyType(Other other);
  // ...
}
```

permet alors d'écrire `MyType mt = other` où `other` est de type `Other`, sans aucun avertissement de la part du compilateur.

Pour éviter ce comportement, vous pouvez utiliser le mot-clé `explicit` devant la déclaration de votre constructeur. Le compilateur exigera alors que vous écriviez `MyType mt(other)` *explicitement*, et pourra vous avertir si vous confondez un objet de type `MyType` avec objet de type `Other` (ou autre erreur similaire).

C'est particulièrement important avec les types supportant déjà d'autres conversions implicites. Par exemple, `MyType mt = true` serait accepté s'il y a un constructeur `MyType(double x)` car le booléen peut être converti en entier, qui peut lui-même être converti en `double`.

Le comportement `explicit` devrait être le comportement par défaut. Il ne l'est pas pour des raisons historiques, alors c'est à vous d'être attentif et de l'utiliser systématiquement, sauf si vous voulez justement créer un opérateur de conversion implicite.

3 Du code sécurisé et robuste

3.1 Activez les avertissements et faites en sorte de ne pas en avoir

C++ est un langage très permissif, qui vous permet de faire a peu prêt n'importe quoi, en ce compris beaucoup de choses stupides. Pour vous aidez à détecter les choses stupides au delà des erreurs de compilation, les (bons) compilateurs disposent de nombreux avertissements : utilisez-les ! *E.g.*, un bon point de départ est `clang++ -std=c++14 -Wall -Wextra -pedantic -Wno-c++98-compat -Wno-c++98-compat-pedantic`. De temps à autre, vous pouvez aussi jeter un œil à `clang++ -Weverything` (mais ce dernier donne trop d'avertissements pour un usage normal, sauf à retirer les messages non-pertinents, comme nous l'avons fait avec `-Wno-c++98-compat` ci-dessus). Voyez la documentation fournie avec votre compilateur.

Une fois les avertissements activés, corrigez votre code en conséquence **jusqu'à ce qu'il n'y en ait plus !** Si vous conservez des avertissements bénins (ou erronés, *e.g.* voir point 2.2), le bruit va rapidement finir par noyer le signal, et vous allez rater les avertissements importants qui apparaîtront quand vous modifierez votre code. Ça ne donne pas non plus une très bonne idée de la qualité de votre code à ceux qui le compile avec avertissements.

Pour supprimez les avertissements :

1. Corrigez votre code quand il y a un problème réel, ou qu'il peut être amélioré en tenant compte de l'avertissement.
2. Si vous estimez (après réflexion) que l'avertissement n'est pas pertinent, vous pouvez souvent le désactiver en utilisant le langage lui-même (*e.g.* `[[maybe_unused]]`, un cast explicite). Simplifier votre code peut aussi aider à satisfaire le compilateur (voir 2.2).
3. Si vous n'arrivez pas à supprimer un avertissement en n'utilisant que le langage, vous pouvez aussi utiliser une directive pour le compilateur (*e.g.* `__attribute__((unused))` indique qu'un argument est inutilisé, aussi bien avec `g++` qu'avec `clang++`).

Pour aller plus loin, vous pouvez aussi utiliser des outils d'analyse statique de code comme `cppcheck` ou `clang-tidy`.

3.2 Évitez le shadowing

En C++, évitez le *shadowing*, *i.e.* la déclaration d'une nouvelle variable avec le même nom qu'une autre variable (ou argument) déjà accessible dans sa portée. Non seulement ça rend la variable originale inaccessible, mais c'est surtout très propice aux confusions ultérieures entre les deux variables (*e.g.* modifier une variable locale temporaire en croyant modifier une variable globale).

Ça rend le code plus difficile à comprendre et plus fragile, et ça indique très souvent une erreur (8 fois sur 10, quand il y a du shadowing dans un de vos projets, il y a un bug associé).

3.3 Attention au fallthrough dans les switch

Dans un `switch` en C/C++, le code continue a s'exécuter avec le cas suivant à la fin d'un cas, sauf s'il y a un `break` ou un `return` (*fallthrough*). Soyez donc attentif à ajouter ce mot-clé systématiquement si seul un des cas doit s'exécuter.

3.4 Utilisez const là où c'est possible

Le mot-clé `const` permet d'éviter des modifications accidentelles, et vous encourage à écrire du code dans un style plus *fonctionnel*, plus facile à analyser et à comprendre.

`const` rend aussi votre interface plus explicite, car l'appelant peut compter sur le fait qu'un argument `const` ne sera pas modifié par l'appelé et n'a donc pas besoin d'être copié si l'objet original doit être conservé tel quel.

Enfin, `const` permet de générer du code plus efficace. Le compilateur a plus d'optimisation à sa disposition quand il peut déterminer qu'une valeur ne sera pas modifiée.

3.5 Un fichier d'implémentation doit toujours inclure son propre entête

Ainsi, le compilateur pourra détecter et vous avertir de certaines inconsistances éventuelles entre l'interface que vous donnez aux utilisateurs de votre module et l'implémentation correspondante.

Vous devriez aussi inclure ce fichier d'entête en premier (voir 2.10).

3.6 Évitez les conversions hasardeuses

Méfiez-vous des conversions implicites entre nombres en virgule flottante et nombres entiers, entre entiers de différentes tailles et entre entiers signés et non-signés. Ces conversions sont sources de nombreux bugs (troncation, underflow, overflow, interprétation d'un nombre négatif comme un grand nombre positif ou l'inverse).

Utilisez les avertissements (3.1) pour détecter ces conversions implicites hasardeuses.

Vous pouvez souvent éviter ces conversions en utilisant les types appropriés (*e.g.* pour indexer un `std::vector`, utilisez un `std::vector::size_type` plutôt qu'un `int`). Vous pouvez aussi parfois éviter d'indexer tout court, en utilisant plutôt un itérateur ou la syntaxe `for (auto &x : xs)` à la place.

Vous pouvez souvent reformuler du code pour éviter des overflow/underflow ou conversions, *e.g.*

```
for (size_t i = 0; i < n - 1; ++i) {
  // Will run 18446744073709551615 times if n == 0
}

for (size_t i = 0; i + 1 < n; ++i) {
  // Will correctly not run at all if n == 0
}
```

3.7 Préférez les casts C++ aux cast C

Les casts C du type `(SomeType) valueOfOtherType` peuvent faire des choses très différentes en fonction des types `SomeType` et `OtherType`. Ils ne font en outre quasiment aucune vérification, laissant passer silencieusement de nombreuses erreurs.

Les casts C++ `static_cast`, `dynamic_cast`, *etc.* ont le mérite d'être beaucoup plus explicites sur ce que vous voulez faire, et les vérifications qui doivent être faites par le compilateur (ou même le programme durant l'exécution). Ces vérifications sont en outre plus expressives (*e.g.* vérifier qu'une classe dérive d'une autre).

3.8 Attention à l'invalidation d'itérateurs

Certaines opérations sur les conteneurs, comme par exemple `std::vector::erase`, invalident les itérateurs existants sur ce conteneurs. Ceux-ci ne doivent plus être utilisés après l'opération.

E.g., le code suivant essaie naïvement de supprimer les valeurs 42 d'un vecteur d'entiers

```
void remove42s(std::vector<int>& v) {
  auto end = v.end();
  for (auto it = v.begin(); it != end; ++it)
    if (*it == 42)
      v.erase(it)
}
```

Après avoir supprimé un élément, `it` peut maintenant pointer vers l'élément suivant du vecteur (puisque les valeurs suivantes vont typiquement être recopiées une case plus tôt dans le tableau sous-tendant le vecteur). Après incrémentation, `it` pointera vers l'élément après-suivant et vous aurez passé un élément (conservant un 42 dans le vecteur si c'en était un). Plus grave, l'itérateur `end` pointe maintenant plus d'un élément après la fin du vecteur, et vous aurez donc un *buffer overflow*, ouvrant la voie à des failles de sécurité, des comportements erronés et des crashs.

3.9 N'oubliez pas les constructeurs, destructeurs ou opérateurs nécessaires

Si votre classe gère manuellement de la mémoire (*i.e.* elle utilise `new`), vous devriez probablement utiliser des composants de la librairies à la place : conteneurs à la place de tableaux primitifs, smart pointers, *etc*.

Si vous devez **vraiment** gérer manuellement une ressource comme de la mémoire, alors n'oubliez pas que vous avez **besoin** d'un :

- destructeur ;
- constructeur par copie ;
- opérateur d'assignation.

Ces éléments doivent être soit écrit (correctement), soit désactivé (par exemple en les rendant privés, ou en utilisant `= delete`), mais la version générée par défaut **ne convient pas**.

Si le destructeur par défaut est conservé, vous aurez une fuite mémoire. Si vous gardez la version par défaut du constructeur par copie, ou de l'opérateur d'assignation, vous aurez un problème de double `delete` s'il est utilisé, conduisant à un crash ou une corruption de la mémoire.

3.10 Attention aux références vers la pile

Soyez attentif à ne pas renvoyer de pointeur ou de référence vers une variable allouée localement sur la pile, puisque ce pointeur (ou référence) sera invalide dès le retour de la fonction.

Si ce problème apparaît explicitement en C où il n'y a que des pointeurs, il peut se produire plus insidieusement en C++ avec les références, *e.g.*

```
int *c_like() {
  int i = 42; // On the stack
  // Explicit & => obvious
  return &i;
}
```

```
int& cpp_like() {
  int i = 42; // On the stack
  // Looks like we just return an int
  return i;
}
```

Si ça reste visible ici, ça l'est beaucoup moins quand la fonction est plus longue. Moralité : soyez attentif à ce problème quand vous écrivez le prototype (*i.e.* type) de vos fonctions.

3.11 Attention aux champs non initialisés

Si un constructeur n'initialise pas certains champs d'un objet, ceux-ci feront l'objet d'une construction par défaut. Pour les objets, ça appellera le constructeur par défaut, mais pour les types primi-

tifs (*e.g.* `int`), ça dépend du contexte et du compilateur (voir cours pour les détails). La plupart du temps, ils ne sont pas initialisés du tout.

Vous devez donc initialiser **tous** les champs de type primitif, ne comptez pas sur le fait qu'ils seront à zéro.

3.12 Évitez d'utiliser une variable en cours de définition (singleton)

Dans les codes tentant d'utiliser le pattern du *singleton*, j'ai souvent vu quelque chose d'équivalent à

```
1 Game *game = game->get_instance();
```

où `Game::get_instance()` est une méthode statique retournant un pointeur vers l'instance unique de `Game`.

Le problème avec ce code, c'est qu'au moment où vous exécutez `game->get_instance()`, `game` n'est pas encore initialisée, puisque l'assignation n'a pas encore eu lieu. Si ce code peut fonctionner par chance en fonction des optimisations du compilateur, il peut aussi crasher ou faire n'importe quoi. N'utilisez jamais la variable que vous êtes en train de définir dans sa définition.

À la place, appeler la méthode statique (qui n'est jamais qu'une fonction) directement :

```
1 Game *game = Game::get_instance();
```

4 Du code efficace

4.1 Rendre les fonctions auxiliaires statiques

Quand une fonction n'est utilisée que dans un seul fichier (unité de compilation), ajoutez le mot-clé `static` à sa définition. Cela indique au compilateur que cette fonction n'a pas besoin d'être disponible pour les autres unités de compilation (*i.e.* de générer un symbole pour cette fonction dans le fichier objet). Cela permet au compilateur d'être plus agressif dans ses optimisations (*e.g.* une fonction normalement trop large pour faire l'objet d'un *inlining* pourrait tout de même l'être si elle n'est utilisée qu'une fois).

Notez que vous n'avez pas besoin d'une déclaration séparée pour vos fonctions statiques si elles apparaissent dans le fichier avant leur utilisation. On place donc généralement ces fonctions au dessus de leur utilisation, et on utilise des déclarations séparée uniquement si certaines d'entre elles sont mutuellement récursives.

4.2 Passez les types structurés par référence

Passez les types structurés comme `std::vector` ou `std::string` par référence (constante) plutôt que par valeur, pour évitez les copies. En effet, par valeur, non seulement la structure sera copiée (quelques octets), mais les données le seront également !

Même si vous avez besoin d'une copie modifiable de l'objet reçu dans le code appelé, il peut être plus efficace de prendre une référence constante en argument et de la copier explicitement, plutôt que d'avoir une copie implicite lors de l'appel. Ce sera en outre plus explicite et plus lisible.

4.3 Pre-incrémenter est plus efficace que post-incrémenter

`++i` est plus efficace que `i++`. En effet, ce dernier doit non seulement incrémenter `i`, mais aussi retenir la valeur qu'il avait précédemment (puisque `i++` renvoie la valeur que `i` avait **avant** l'incrémentation).

Le compilateur pourra souvent optimiser `i++` en `++i` tout seul quand `i` est entier, mais ne le fera pas avec les types définis par l'utilisateur (notamment les itérateurs, qui devront donc être copiés en notation postfixe). Prenez donc l'habitude de pre-incrémenter quand vous n'avez pas besoin de conserver la valeur précédente.

La même remarque vaut également pour la décrémentation.

4.4 Ne faites pas un find avant un insert

Quand vous voulez insérer une valeur dans un `std::map` (ou équivalent), il n'est pas nécessaire de vérifier qu'une valeur n'est pas déjà présente avec `find` avant de l'ajouter avec `insert`. En effet, `insert` ne fait rien si la valeur est déjà présente, et vous économisez ainsi un parcours de la structure (souvent $\mathcal{O}(\log n)$ où n est le nombre d'éléments).

4.5 Attrapez les exceptions par référence plutôt que par valeur

Ce point est similaire au point 4.2. Évitez de faire des copies inutiles.

En outre, c'est encore plus important pour les exceptions, car les opérations de copie pourraient elles-même déclencher une exception dans l'exception (*e.g.* s'il n'y a plus assez de mémoire pour copier l'exception).

4.6 Initialiser les champs dans la liste d'initialisation

Quand vous écrivez un constructeur comme

```
1 C(int i) {
2   slow_heavy_object = SlowHeavyObject(i);
3 }
```

L'objet `slow_heavy_object` sera construit deux fois chaque fois que vous construisez un objet de classe `C`. Une première fois avec le constructeur par défaut, puis une seconde avec l'argument `i` dans le corps du constructeur.

Quand c'est possible, il est plus efficace d'utiliser une *liste d'initialisation* à la place :

```
1 C(int i) : slow_heavy_object(i) {}
```

Ici, le constructeur par défaut de `SlowHeavyObject` n'est pas appelé, `slow_heavy_object` sera initialisé directement avec l'appel à `SlowHeavyObject(i)`.

Depuis C++11, vous pouvez aussi utiliser la syntaxe qui assigne directement une valeur aux champs de la classe.

4.7 Évitez les variables et fonctions inutilisées

Á part si vous écrivez une librairie, et que le code qui utilisera le vôtre n'est pas encore connu, toutes vos variables et fonctions devraient être utilisées, au moins une fois.

Si ce n'est pas le cas, ça peut indiquer une erreur logique dans le programme. Si une variable ou une fonction a été écrite mais n'est pas utilisée, c'est peut-être parce qu'une autre variable ou fonction a été utilisée erronément à la place (ce qui est d'autant plus facile à faire si vous faites du *shadowing*, voir point 3.2).

D'autre part, ça va ralentir la compilation et souvent augmenter la quantité de code machine produite, conduisant à des programmes plus lourds et plus lents (moins bonne localité du cache d'instructions).

Si vous avez développé une fonction qui n'est plus ou pas encore nécessaire, mais dont vous pourriez avoir besoin dans le futur, utiliser un gestionnaire de version ! Supprimez cette fonction de votre code. Si jamais vous en aviez besoin plus tard, il suffira de la récupérer avec git (ou autre).