

# Bonnes pratiques du C++ - Développement critique embarqué

(1 décembre 2023)

## TABLE DES MATIERES

Introduction.....	4
Logiciel critique pour la sécurité.....	4
Divergence des langages C et C++ .....	6
Principes de développement.....	8
Lire la documentation.....	8
Vérifier ses travaux .....	9
Principes génériques.....	9
Style de codage.....	9
Programmation défensive .....	12
Standard Template Library .....	13
Programmation.....	14
Interfaces.....	16
Isolation du code .....	16
Interface précise et fortement typée .....	16
Librairie ABI.....	16
Idiome Pimpl.....	17
Dual inheritance .....	20
Autres considérations.....	23
Fonctions .....	26
Définition d'une fonction.....	26
Passage d'arguments.....	26
Sémantique de possession .....	27
Classes .....	30
Type régulier.....	30
Rule of zero.....	31
Rule of six (ou five) .....	32
Hierarchies de classes.....	33
Instructions.....	38
Gestion des ressources.....	40
RAII.....	40
Smart pointers .....	41
Gestion d'erreur .....	45
Conception.....	45
Exception .....	49
Codes d'erreurs .....	50

Concurrence .....	54
Généralités.....	54
Section critique.....	59
Tâche .....	60
Lock-free programming .....	61
Performance .....	63
Mauvaises optimisations .....	63
Application des bonnes pratiques .....	66
Template.....	68
Usage .....	68
Interfaces.....	69
Définitions .....	70
Tests.....	77
Architecture.....	78
Définitions de la conception .....	78
Modification/Extension – Cas pratique .....	79
Abstraction .....	89
Temps réel .....	96
Outils.....	98
Annexes .....	99
Types forts .....	99
External Polymorphism et Type Erasure.....	103
Liens.....	109

## INTRODUCTION

Le développement embarqué d'un logiciel critique pour la sécurité est soumis à des contraintes spécifiques liées à l'environnement dans lequel le logiciel est utilisé. Pour cela, les critères attendus dans les développements sont :

- La sécurité : Les données utilisées par le logiciel sont sensibles.
- La robustesse : Le logiciel doit pouvoir être disponible dans des situations de stress matériel et opérateur.
- La maintenabilité : Le logiciel sont déployés sur du matériel ayant une longue durée de vie.

## LOGICIEL CRITIQUE POUR LA SECURITE

L'évolution rapide de la technologie a considérablement amplifié notre dépendance aux systèmes informatiques, faisant de la sécurité des logiciels un enjeu crucial. Les applications critiques, telles que celles utilisées dans les secteurs de la santé, de l'automobile, de l'aéronautique, de l'énergie et de la défense, requièrent une approche rigoureuse et méthodique pour répondre aux normes les plus élevées en matière de fiabilité et de résilience.

La sécurité logicielle implique donc sa prise en compte à tous les niveaux du processus de développement logiciel : La conception doit prévoir les vérifications nécessaires, l'implémentation doit prendre en compte les bonnes pratiques d'implémentation, les tests doivent permettre de couvrir les cas nominaux et les cas d'erreur.

Dans le présent document, une sélection des règles de base est présentée. Elle est très largement non exhaustive et elle s'inspire des C++ Core Guidelines [1] en ce qui concerne les règles d'implémentation. Les C++ Core Guidelines sont les bonnes pratiques de développement en langage C++. Elles sont accessibles à tous et ont pour but d'évoluer au fur et à mesure des évolutions du langage. En fin de paragraphes, des renvois vers les C++ Core Guidelines seront présentés pour approfondir le sujet.

Dans le développement de logiciel critique pour la sécurité, vous avez sans doute été contraint à respecter des règles de codage. Les règles de codage les plus connues sont certainement MISRA C++ :2008. Elles ont été publiées par la *Motor Industry Software Reliability Association*. Elles sont basées sur *MISRA C guidelines* de 1998. Initialement conçues pour l'industrie automobile, elles sont devenues de facto un standard dans le milieu industriel. Elles ont, cependant, un problème conceptuel : le langage C++ évolue et les règles MISRA-C++ ont déjà plusieurs générations de retard (\*).

Au vu de toutes les règles et contraintes évoquées, il est peut-être temps d'énoncer la core guideline « (In.0) : Don't panic ! ». Les bonnes pratiques insistent sur les problèmes connus du langage C++ avec des solutions techniques (déréférencement de pointeur nul, fuite mémoire, violation de mémoire, overflow, underflow, ...). Les bonnes pratiques ne sont pas des commandements à suivre aveuglément. Il est évident que l'objectif n'est pas de reprendre tout l'historique de 20 ans de code d'un logiciel pour corriger toutes les erreurs. De plus, certaines bonnes pratiques ne sont pas des règles vérifiables automatiquement. Ces bonnes pratiques nécessitent une part de réflexion quant à leurs applications. Parmi lesquelles, quelques-unes sont des directives de conception pour améliorer la maintenabilité, la testabilité, etc. Ainsi, écrivait Bjarne Stroustrup dans son livre « The C++ Programming Language » :

« *Design and programming are human activities ; forget that and all is lost.* »

(\*Une nouvelle version de MISRA-C++ est sortie au moment de la fin de l'écriture du guide : *MISRA C++:2023: Guidelines for the use of C++17 in critical systems Paperback* – 27 Nov. 2023. Cependant, la problématique des

*règles MISRA C++ reste identique car elle n'apporte qu'une liste de règles de codage à l'instant t pour un standard défini. Les règles de codage sont rédigées à partir d'un sous ensemble de bonnes pratiques des C++ Core Guidelines.)*

## DIVERGENCE DES LANGAGES C ET C++

Avant d'aller dans le vif du sujet, il est bien de se rappeler pourquoi le langage C++ a été créé. Pour répondre à ces questions, commençons par faire un peu d'histoire et par casser une idée reçue qui est que le C++ est une évolution du C avec l'ajout de fonctionnalités de programmation objet. Et bien non, le langage C et le langage C++ sont bien deux langages différents comme Neandertal et Homo sapiens sont deux espèces de genre homo différents (\*). Cette idée reçue, bien qu'historiquement correct (Le C++ a évolué d'une version ancienne du langage C), n'est plus vraie aujourd'hui. En fait, C et C++ sont séparés d'un ancêtre commun de plus de 40 ans et ont évolué séparément depuis (L'ancêtre commun est le « classic C » de 1978). Ces évolutions n'ont pas été faites en total isolation : certains concepts et fonctionnalités ont été échangés, ou bien, ont été conçus en étroite collaboration entre les comités de standardisation ISO C et C++ (Homo sapiens a bien quelques gènes de Neandertal dû aux croisements qui se sont opérés plus tard). Cependant, des différences subsistent, un code C avec le standard C17 [2] ne compilera pas avec un compilateur C++. Et plus généralement, les deux langages n'ont pas la même philosophie d'utilisation.

Le langage C est un langage de bas-niveau (logiciel sans système d'exploitation ou se situant en dessous du système d'exploitation : pilote matériel, boot loader, ...). Le développeur doit pouvoir manipuler la mémoire, transtyper des données, ... sans que le compilateur lui interdise (Le compilateur C prend comme hypothèse que le développeur sait ce qu'il fait).

Le langage C++ a été créé entre autres, pour apporter de la sécurité sur les types et sur la gestion des ressources. Il possède, pour cela, des mots clés et une librairie plus adaptée à la programmation haut-niveau.

Ainsi, lors du développement pour les logiciels critique pour la sécurité, ne mixez pas le langage C avec le langage C++ et préférez l'utilisation des fonctions de la Standard Template Library (STL) à la librairie standard du C. Les exemples de programmation « C-style » à éviter sont :

- L'utilisation des pointeurs (sauf pour exprimer l'absence de valeur)
- L'utilisation des tableaux []
- L'utilisation des structures
- L'utilisation des unions
- L'utilisation des énumérations non délimités
- L'utilisation des macros
- L'utilisation de fonctions variadics
- Les cast de type C
- Les sauts inconditionnels (goto, continue, break dans les boucles)
- L'utilisation des fonctions C (printf, scanf, strcpy, memcpy, malloc, ...)

Si le module doit posséder une interface « C », limitez l'utilisation des types C à l'API, convertissez les pointeurs en référence, les tableaux en `std::vector` ou `std::array`, et faites appels à des classes réalisant le traitement spécifique.

*(\*Ne faites pas dire ce que je n'ai pas dit, je ne pense pas que les langages C ou C++ soient voués à disparaître.)*



### C++ CORE GUIDELINES

[P.4: Ideally, a program should be statically type safe](#)

[I.13: Do not pass an array as a single pointer](#)

[F.15: Prefer simple and conventional ways of passing information](#)  
[F.60: Prefer `T\*` over `T&` when “no argument” is a valid option](#)  
[C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently](#)  
[C.8: Use `class` rather than `struct` if any member is non-public](#)  
[C.181: Avoid “naked” unions](#)  
[Enum.1: Prefer enumerations over macros](#)  
[Enum.2: Use enumerations to represent sets of related named constants](#)  
[Enum.3: Prefer `enum classes` over “plain” `enums`](#)  
[Enum.4: Define operations on enumerations for safe and simple use](#)  
[Enum.5: Don’t use `ALL CAPS` for enumerators](#)  
[Enum.6: Avoid unnamed enumerations](#)  
[Enum.7: Specify the underlying type of an enumeration only when necessary](#)  
[Enum.8: Specify enumerator values only when necessary](#)  
[ES.27: Use `std::array` or `stack\_array` for arrays on the stack](#)  
[ES.30: Don’t use macros for program text manipulation](#)  
[ES.31: Don’t use macros for constants or “functions”](#)  
[ES.34: Don’t define a \(C-style\) variadic function](#)  
[ES.47: Use `nullptr` rather than `0` or `NULL`](#)  
[ES.49: If you must use a cast, use a named cast](#)  
[ES.76: Avoid `goto`](#)  
[ES.77: Minimize the use of `break` and `continue` in loops](#)  
[CPL.1: Prefer C++ to C](#)  
[CPL.2: If you must use C, use the common subset of C and C++, and compile the C code as C++](#)  
[CPL.3: If you must use C for interfaces, use C++ in the calling code using such interfaces](#)

### LIRE LA DOCUMENTATION

La première bonne pratique paraît plutôt évidente : Lire la documentation. Cependant, cette activité est souvent sous-estimée soit par manque de temps, ou soit par péché d'orgueil (mais cela n'arrive jamais ! donc c'est toujours par manque de temps).

Pour évoquer l'importance de cette bonne pratique, je propose de l'illustrer par une mise en situation :

« Votre entreprise reçoit une commande de son client, de mise à jour logiciel et matériel. Pour des raisons techniques et de livraison, la commande est séparée en deux lots : la partie logicielle sera livrée dans un premier temps, et la partie matérielle, dans un second temps.

La mise à jour logicielle consiste à ajouter un bouton "annuler" sur une fenêtre d'attente d'une opération mécanique et la mise à jour matérielle consiste à changer la dalle tactile du calculateur. Le changement de la dalle tactile n'a aucun impact sur le logiciel.

Le chef de projet vous confie le développement. Au niveau du code, l'attente de l'opération mécanique est réalisée sur un fil d'exécution asynchrone et lorsque le temps d'attente arrive à échéance, un message est émis au matériel pour démarrer un moteur.

Vous voyez immédiatement ce qu'il faut faire au niveau de l'implémentation car il existe un autre traitement asynchrone qui peut être interrompu avant la fin de l'échéance : il faut utiliser une `std::condition_variable` pour gérer l'attente et le réveil du fil d'exécution. Vous vous inspirez du code existant, vous simplifiez un peu, ce n'est pas très compliqué : une méthode "wait" pour l'attente et une méthode "notify\_one" pour le réveil.

Vous faites vos tests : pas de problème, puis viennent les tests d'intégrations : pas de problème, les tests de validation sur banc : pas de problème et les tests système : pas de problème.

Et pourtant six mois plus tard, le client vous recontacte pour vous indiquer que le logiciel ne fonctionne plus parfois après un clic sur le bouton "annuler" et que parfois, le moteur démarre avant la fin de l'opération mécanique.

Après plusieurs heures d'analyses et de déplacements chez le client, vous comprenez que le problème se situe au niveau du "wait". Vous êtes victime de Lost Wakeup et de Spurious Wakeup. »

Suite à cette trop longue mise en situation, expliquons un peu la succession d'événements qui a mené à cette situation. Les tests sur banc n'ont révélé aucune anomalie car ils ne sont pas représentatifs d'un système en fonctionnement. Les tests système ont été faits avec une dalle tactile différente avec une latence différente de la nouvelle dalle. Toutes les heures passées à investiguer, à corriger, à justifier et à revalider auraient pourtant pu être évité dès le début. En effet, la documentation de `std::condition_variable` [3] indique clairement comment utiliser la fonction "wait" et que celle-ci doit être utilisée en combinaison d'un booléen protégé par un `std::mutex` pour éviter les Lost Wakeup et les Spurious Wakeup.

Pour résumer, lorsque vous implémentez une fonctionnalité,

- Vérifiez toujours la documentation lorsque vous utilisez une fonction qui vous n'est pas familière.
- Ne vous fiez pas à des utilisations existantes dans le code.

Et dans la documentation,

- Vérifiez si la fonction n'a pas été dépréciée



- Vérifiez si des contraintes existent selon des contextes d'utilisation
- Vérifiez si d'autres fonctions existent qui seraient plus appropriées à utiliser dans ce contexte d'utilisation
- Vérifiez si une bonne pratique ne limite pas l'utilisation de cette fonction

## VERIFIER SES TRAVAUX

Si vous avez une expérience dans le développement logiciel, vous connaissez certainement déjà cette bonne pratique qui s'impose lorsque l'on travaille en équipe. En effet, il n'y a pas plus rapide pour s'attirer les foudres de vos collègues que de se croire seul à développer sur le code. Cependant, il est bien de rappeler les vérifications à effectuer pour travailler en équipe :

- Vérifiez la compilation du projet. Si vous travaillez sur un module et que cela a un impact sur le logiciel appelant, copiez la dépendance dans le logiciel appelant et vérifiez la compilation du logiciel appelant.
- Si vous commencez des travaux sur un logiciel ou un module, incrémentez le numéro de version (dans le doute, vérifiez dans les documents de GCL ou auprès du chef de projet, quelle est la version attendue).
- Testez vos développements. Il n'y a pas mieux pour perdre du temps que de laisser vos collègues essayer de comprendre pourquoi le logiciel ne fonctionne plus lorsqu'ils intègrent leurs travaux.
- Utilisez un outil d'analyse statique de code sur les nouveaux fichiers et les fichiers modifiés (limitez la correction aux fonctions modifiées ou nouvelles) pour corriger un maximum d'erreurs d'implémentation.
- Relisez vos modifications avec un outil de diff. Cela permet de supprimer le code qui aurait été modifié pour faire du test, d'annuler les modifications accidentelles, de corriger vos commentaires. Cette phase est importante car elle permet de prendre un peu de hauteur et de vérifier que rien n'a été oublié.
- Rebasez et quashez vos travaux. Ne laissez pas vos travaux dormir dans une branche à part car plus vous attendez et plus la gestion des conflits prendra du temps et sera complexe.

## PRINCIPES GENERIQUES

Assez tourné autour du pot ! Commençons par poser les principes dans lequel les bonnes pratiques s'inscrivent. Les principes décrits ne sont pas des règles vérifiables mais ils donnent les lignes directrices.

### STYLE DE CODAGE

Les bonnes pratiques liées aux styles de codage recouvrent des règles purement esthétiques. Elles permettent néanmoins de faciliter le travail en équipe et la maintenabilité. Le style de codage est souvent prédéfini dans un projet existant. Les règles présentées dans les C++ Core Guidelines ou dans les Style Guide [4] ne sont à prendre à compte que si rien n'est encore défini. Cependant, voici quelques principes génériques pouvant s'appliquer à tous styles de codage :

- Choisissez avec attention le nom de vos déclarations. Le nommage est une activité difficile, celui-ci doit exprimer une intention. Voici quelques exemples pour clarifier ce principe.
  - o Essayez d'être précis dans le nommage pour éviter d'être mal-interprété. Utilisez des alias pour ajouter de la signification au type générique.

```
double getRatio(double val, double size) // imprécis
{
    return (val / size) * 100.0;
}
```

```
Percent calcPercentage(const Quantity part, const Quantity total); //
plus précis
```

- Restez homogène dans les termes, c'est-à-dire utilisez toujours le même nom pour une même donnée.

```
Shape diamond = rhombus;
Shape lozenge = rhombus;
// Les termes 'rhombus', 'diamond', 'lozenge' sont plus ou moins
synonyme en anglais mais ils peuvent prêter à confusion car parfois,
'diamond' et 'lozenge' désignent un type de losange spécifique. (Et,
si par malheur, le logiciel évolue pour gérer les types de losange
spécifiques, la lecture du code ne pourra pas permettre de déterminer
le type de losange manipulé.)
```

- Ayez des noms bien différents et sans faute d'orthographe, pour éviter les mauvaises lectures.

```
char distance = 'm';
double unite = 1.0;
double temp = 0.0; //temporaire
double dsitance = temp * unite;
double temps = 10.0;
double vitesse = distance / temp;
// utilisation des mauvaises variables distance et temp
```

- Définissez une longueur de nom de variable proportionnelle à la taille de la portée.

```
int g; // mauvais : variable globale
double sqrt(double nonNegativeNumericValue); // mauvais : paramètre de
fonction
```

- Limitez le nombre d'arguments.

- Regroupez les paramètres ayant une cohérence entre eux, votre code sera plus clair.

```
double computeDistance(double, double, double, double, double,
double); // imprécis latitude1, longitude1, altitude1, latitude2,
longitude2, altitude2 ou latitude1, latitude2, ...

double computeDistance(GeoPosition, GeoPosition); // plus précis
```

- Différenciez les paramètres en entrée et en lecture seule et les paramètres de sortie. Définissez vos paramètres en entrée avec le spécificateur « const » et vos paramètres de sortie en retour de fonction. Cerise sur le gâteau, le code gagne en clarté et il est plus optimisé [5].

```
void findFirstAboveBound(const std::vector<int>& range, bool& found,
int& element, const int bound = 0);
// imprécis : element et found sont à définir en entrée ou sont-ils
des éléments seulement en sortie ?

std::pair<bool, int> findFirstAboveBound(const std::vector<int>&
range, const int bound = 0);
// plus précis
```

```

// Note : Utilisez std::tuple si vous avez plus que 2 sorties
std::tuple<bool, std::string, int> readIntParameter();

// Note : Utilisez std::move seulement si le tuple/pair est initialisé
avec la copie de variables locales de grande taille
std::pair<bool, std::vector<uint8_t>> readBuffer()
{
    std::vector<uint8_t> buffer = Read(File);
    return {!buffer.empty(), std::move(buffer)};
}

// Note : Utilisez std::tie ou auto[...] (structured bindings) pour
récupérer les valeurs
// C++11
bool success;
std::vector<uint8_t> buffer;
std::tie(success, buffer) = readBuffer();
if (success)
{
    process(buffer);
}
// C++17
if (auto [success, buffer] = readBuffer(); success)
{
    process(buffer);
}

```

- Conservez le style de codage du projet. Dans le cas contraire et si vous modifiez une fonction ou une classe existante, le relecteur de votre code éprouvera sans doute un arrière-goût désagréable dans la bouche. Cette mauvaise sensation est dû au fait que si vous n'avez pas fait attention au style de codage existant, vous n'avez peut-être pas pris le temps d'étudier des éléments plus subtils comme l'architecture dans laquelle s'inscrit la fonction ou la classe et notamment pris en compte les principes « SOLID » et les niveaux d'abstraction. (Je vois votre perplexité dans vos yeux, j'y reviendrai dans la partie Architecture, promis.)



## C++ CORE GUIDELINES

[P.1: Express ideas directly in code](#)

[P.3: Express intent](#)

[NL.1: Don't say in comments what can be clearly stated in code](#)

[NL.2: State intent in comments](#)

[NL.3: Keep comments crisp](#)

[NL.4: Maintain a consistent indentation style](#)

[NL.5: Avoid encoding type information in names](#)

[NL.7: Make the length of a name roughly proportional to the length of its scope](#)

[NL.8: Use a consistent naming style](#)

[NL.9: Use ALL CAPS for macro names only](#)

[NL.10: Prefer underscore style names](#)

[NL.11: Make literals readable](#)

[NL.15: Use spaces sparingly](#)  
[NL.16: Use a conventional class member declaration order](#)  
[NL.17: Use K&R-derived layout](#)  
[NL.18: Use C++-style declarator layout](#)  
[NL.19: Avoid names that are easily misread](#)  
[NL.20: Don't place two statements on the same line](#)  
[NL.21: Declare one name \(only\) per declaration](#)  
[NL.25: Don't use `void` as an argument type](#)  
[NL.26: Use conventional `const` notation](#)  
[NL.27: Use a `.cpp` suffix for code files and `.h` for interface files](#)  
[F.20: For "out" output values, prefer return values to output parameters](#)  
[F.21: To return multiple "out" values, prefer returning a struct or tuple](#)

---

## PROGRAMMATION DEFENSIVE

Il existe pléthore de règles logicielles implémentées dans les logiciels d'analyse statique de code, pour vérifier la programmation défensive. Mon but n'est pas de faire une énumération de ces règles mais de rappeler quelques principes qui s'appliquent à tous les logiciels qu'il soit sécuritaire ou non :

- Vérifiez les données. Lorsque les données proviennent de l'extérieur du module ou du logiciel, ces données doivent être vérifiées (Zone de confiance et défense en profondeur [6]). Par exemple, on peut vérifier :
  - o les domaines de définitions pour les valeurs numériques
  - o le format pour les chaînes de caractères
  - o la taille pour les données non typées ou les tableaux
  - o la structure pour les données structurées
  - o le checksum
  - o ...
- Vérifiez les pointeurs. En C++, vous considérez un pointeur comme une valeur optionnelle (pour les autres cas d'utilisation des pointeurs, préférez utiliser une référence ou un smart pointer), donc, il faut tester si la valeur est différente de « `null_ptr` ».
- Encapsulez les objets dangereux (pointeurs alloués, handles windows, objets système comme les threads ou les objets de synchronisation). Utilisez toujours un mécanisme de RAII [7] pour manipuler des objets susceptibles de faire des fuites mémoires ou des blocages. Pour cela, n'oubliez pas de regarder la STL car elle propose déjà de nombreuses classes avec l'implémentation de ce mécanisme (`std::unique_ptr`, `std::shared_ptr`, `std::lock_guard`, `std::future`, `std::vector`, `std::string`, ...)
- Initialisez les données. Les données de type primitif ou de type pointeur doivent être initialisées, notamment à cause des vérifications nécessaires décrites ci-dessus. De plus, pour les codes d'erreur ou les booléens de validité, initialisez-les avec le cas d'erreur pour éviter des faux positifs dans les vérifications des données. Également, préférez l'initialisation avec les accolades pour détecter les cas de "narrowing conversions" et ajoutez le spécificateur « `const` » ou « `constexpr` » systématiquement si la donnée n'est pas modifiée.

```
constexpr int32_t size{16};  
const char* const name = "PROGRAM";  
T var{}; // initialisation avec la valeur par défaut
```

Le corollaire aux données « const » est d'ajouter systématiquement le spécificateur « const » aux méthodes de classes si elles ne modifient pas les données membres de la classe.

```
Class Point
{
    // ...
    int getX() const;
    int getY() const;
};

const Point p{16, 50}; // p est un objet « const »
std::cout << p.getX() << "," << p.getY(); // seules l'appel aux méthodes
« const » sont autorisées sur cet objet
```



## C++ CORE GUIDELINES

[P.5: Prefer compile-time checking to run-time checking](#)

[P.6: What cannot be checked at compile time should be checkable at run time](#)

[P.7: Catch run-time errors early](#)

[P.8: Don't leak any resources](#)

[P.10: Prefer immutable data to mutable data](#)

[Con.1: By default, make objects immutable](#)

[Con.2: By default, make member functions const](#)

[Con.3: By default, pass pointers and references to consts](#)

[Con.4: Use const to define objects with values that do not change after construction](#)

[Con.5: Use constexpr for values that can be computed at compile time](#)

---

## STANDARD TEMPLATE LIBRARY

Pour souffler un peu, débutons ce chapitre par deux proverbes de la communauté C++ [8] :

« *When you use explicit loops, you don't know the algorithm of the STL.* »

« *When your program has undefined behavior, your program has catch-fire semantics. This means your computer can catch fire* »

Le principe derrière le premier proverbe, est d'utiliser les algorithmes de la STL pour développer votre code. Les raisons sont que :

- cela permet d'éviter des bugs dûs à une implémentation compliquée.
- cela permet de gagner du temps, en ne réimplémentant pas du code qui existe
- les implémentations de la STL respectent les principes « SOLID » (\*1)
- les implémentations de la STL sont optimisées en temps et en mémoire (\*2)

(\* ok, à part quelques exceptions (1: std::string, 2: std::regex, ...), qui s'explique par des problématiques de compatibilité.)

Donc, j'insiste : il n'y a aucune bonne raison de ne pas utiliser la STL et de se lancer dans des implémentations « C-style ». S'il faut un exemple pour démontrer ce principe, considérons le code suivant :

```
char* str = "UPPERSTRING";
int offset = 'a' - 'A';
char* lowerStr = new char[sizeof(str)];
memset(lowerStr, 0, sizeof(str));
int i = 0;
while (i < strlen(str))
{
    lowerStr[i] = str[i] + offset;
    i++;
}
```

(Je comprends le malaise que l'on ressent devant ce code, mais j'ai le remède juste après.)

Le code ci-dessus, a de nombreux problèmes : il ne fonctionne qu'avec les caractères [A-Z], n'est pas optimisé en temps, ni en mémoire et a une fuite mémoire.

```
std::string str{"UPPERSTRING"};
std::transform(std::begin(str), std::end(str), std::begin(str), [](char c)
{
    return std::tolower(c);
});
```

L'implémentation ci-dessus, est beaucoup plus intéressante : elle fonctionne dans tous les cas, elle est optimisée et elle est sûre.

Le second proverbe souligne le fait de lire la documentation avec attention. En effet, certains cas d'utilisation des algorithmes de la STL sont indiqués comme ayant un comportement indéfini ou un comportement implémentation-défini. Il faudra dans ces cas-là, réaliser une vérification pour éviter le comportement indéfini ou implémentation-défini.



C++ CORE GUIDELINES

[P.2: Write in ISO Standard C++](#)

[P.3: Express intent](#)

[P.9: Don't waste time or space](#)

[P.11: Encapsulate messy constructs, rather than spreading through the code](#)

[P.13: Use support libraries as appropriate](#)

---

## PROGRAMMATION

Finissons, enfin, ce paragraphe par deux principes de programmation qui peuvent apparaître contradictoire au premier abord :

- DRY est l'acronyme de « *Don't repeat yourself* » et que l'on peut résumer par factoriser au maximum le code. Le bénéfice de factoriser est sans conteste car cela permet d'améliorer la maintenabilité (un

seul endroit à changer), de réutiliser du code au lieu de le réimplémenter. Il permet souvent de répondre au principe de responsabilité unique (le « S » de « SOLID »). Voici un exemple de code « DRY » d'une recherche d'un élément dans un conteneur :

```
auto it = std::find_if(std::cbegin(range), std::cend(range), CompareId);
```

Le type de retour est remplacé par auto : il est inutile de répéter le type du conteneur, savoir que std::find\_if retourne un itérateur est suffisant à la compréhension. Le prédicat est sous forme de fonction (CompareId) pour être réutilisé dans d'autres appels.

- « *Explicit is better than implicit* ». Pas besoin d'expliquer le proverbe : un code explicite est meilleur pour la lisibilité. Voici le même code mais avec une approche explicite :

```
std::vector<Element>::const_iterator iterator =  
std::find_if(std::cbegin(range), std::cend(range), [](const Element& e){  
    return e.Id == 0;  
});
```

Le type de retour est réécrit entièrement. Le prédicat est sous forme d'un lambda pour permettre de visualiser facilement la condition.

Maintenant que vous avez vu les deux exemples de code, vous vous apercevez qu'avoir une approche jusqu'au-boutiste d'un des principes ne semble pas être la bonne solution et qu'il faut avoir une approche mixte. Dans l'exemple, la réécriture du type de retour est peut-être excessive car le nom de la variable est suffisamment explicite (il peut être plus long, comme dans le cas d'une map avec des types template) et le lambda est plus lisible que la fonction (si le prédicat n'est utilisé qu'une ou deux fois et qu'il est assez court). Cette approche mixte s'inscrit dans ce qu'on peut appeler le principe YAGNI (« *You Aren't Gonna Need It* ») qui sera détaillé dans la partie Architecture. Le code exemple devient donc :

```
auto iterator = std::find_if(std::cbegin(range), std::cend(range), [](const auto&  
e){  
    return e.Id == 0;  
});
```

Notez que l'explication se concentre sur l'aspect de la lisibilité du code, quand il y a le choix. Il va sans dire que le principe « DRY » prévaut sur la duplication de code. Par exemple, évitez de créer une surcharge avec du code dupliqué :

```
int convertInt(const std::string& str)  
  
int convertInt(const std::string& str, const ENumericBase base)
```

Utilisez, dans ce cas, une valeur par défaut :

```
int convertInt(const std::string& str, const ENumericBase base =  
ENumericBase::Decimal)
```

## INTERFACES

Les C++ Core Guidelines indiquent plusieurs bonnes pratiques pour la conception d'interface. Typiquement parlant, certaines bonnes pratiques ne sont pas des règles de conception et encore moins d'architecture mais si elles ne sont pas respectées, cela peut nuire à la conception du logiciel et par voie de conséquence, à la sécurité, à la maintenabilité et à la testabilité du logiciel. Les interfaces sont probablement l'aspect le plus important de l'organisation du code. Une interface définit un contrat entre un fournisseur de service et un utilisateur de service.

## ISOLATION DU CODE

La première bonne pratique est plutôt évidente :

- Évitez les variables globales non const.
- Évitez les singletons.

Les inconvénients d'une variable globale sont bien connus mais toutefois voici une énumération pour se rafraîchir la mémoire :

- Conception : Les encapsulations de classes sont polluées par une dépendance qui souvent ne respecte pas les niveaux d'abstractions.
- Testabilité : On ne peut pas tester les unités (fonctions, classes, modules) en isolation.
- Factorisation : On ne peut pas séparer les responsabilités.
- Optimisation : On ne peut pas faire de calcul parallèle. De plus, l'utilisation de variables globales limite les optimisations du compilateur.
- Concurrence : Les variables globales sont par nature, non protégées aux appels concurrentiels.

Les singletons sont des variables globales non const déguisées et apportent plus de problématiques que de solutions. Outre les problèmes des variables globales, les problématiques du singleton sont :

- Qui est responsable de la destruction du singleton ?
- Peut-on étendre un singleton ?
- Comment initialiser un singleton ?

Par conséquent, vous devez retenir que le singleton **n'est pas** un patron de conception. La seule justification possible à son utilisation serait une problématique d'implémentation liée à une librairie tiers (Utilisation d'une interface non extensible, interface C, ...). (Donc, ne présentez pas le singleton comme une solution de conception si vous ne voulez pas être regardé de travers et pensez « injection de dépendance » à la place.)

## INTERFACE PRECISE ET FORTEMENT TYPEE

La seconde bonne pratique est un rappel sur la déclaration des fonctions, déjà évoqué dans la partie « Style de codage ». Pour résumer les points importants :

- Soyez précis dans le nommage. Utilisez des alias pour apporter de la signification aux types génériques.
- Limitez le nombre d'arguments. Regroupez les arguments qui sont liés entre eux. Différenciez les paramètres d'entrées en lecture seule et les paramètres de sortie.
- N'utilisez pas les types pointeurs \* ou tableaux [] (programmation « C-style ») pour passer un tableau de valeur. Utilisez les conteneurs de la STL et les références.

## LIBRAIRIE ABI



Ce chapitre est un peu plus conséquent donc prenez une pause, allez chercher un café (ou tout autre boisson non alcoolisée). Prêt, commençons, ABI signifie « Application Binary Interface ». Elle constitue l'interface entre deux programmes binaires. Pour des considérations de maintenabilité, une ABI doit être stable. En effet, la modification d'une dépendance ne doit pas entraîner la recompilation de l'intégralité du programme. Afin d'arriver à cette objectif, il est nécessaire de fournir une interface avec un couplage faible, séparant l'interface accessible au client et son implémentation. Il existe deux modèles de conception :

- Pimpl idiom
- Dual inheritance

---

## IDIOME PIMPL

Prenons l'exemple de modifications d'une librairie de base, c'est-à-dire une librairie utilisée dans plusieurs logiciels existant. Ces modifications consistent à la correction de bugs et à l'ajout d'une nouvelle donnée dans une structure en paramètre de l'API pour répondre au besoin d'un nouveau logiciel. Par exemple, considérons une structure Person :

```
struct Person
{
    std::string firstName;
    std::string lastName;
    bool isSingle; // nouvelle donnée
    size_t nbChildren; // nouvelle donnée
};
```

A priori, la nouvelle librairie doit pouvoir être utilisée avec les anciens logiciels sans nécessité de les modifier (pour profiter de la correction des bugs). Or, cela nécessite de recompiler tous les anciens logiciels car la taille de la structure Person en mémoire n'est plus identique.

Cette problématique est résolue grâce à l'idiome pimpl (qui est une implémentation du design pattern Bridge). L'implémentation classique de l'idiome pimpl est la suivante :

- Person.h

```
class Person
{
public:
    LIBRARY_API Person();
    LIBRARY_API ~Person();

    LIBRARY_API Person(const Person&);
    LIBRARY_API Person(Person&&);
    LIBRARY_API Person& operator=(const Person&);
    LIBRARY_API Person& operator=(Person&&);

    LIBRARY_API const std::string& getFirstName() const;
    LIBRARY_API const std::string& getLastName() const;
    LIBRARY_API bool isSingle() const;
    LIBRARY_API size_t getNbChildren() const;

    LIBRARY_API void setFirstName(const std::string& firstName);
    LIBRARY_API void setLastName(const std::string& lastName);
```

```

    LIBRARY_API void setSingle(const bool single);
    LIBRARY_API void setNbChildren(const size_t children);

private:
    class impl;
    const std::unique_ptr<impl> pimpl;
};

```

- Person.cpp

```

class Person::impl
{
public:
    const std::string& getFirstName() const {
        return firstName_;
    }
    const std::string& getLastName() const {
        return lastName_;
    }
    bool isSingle() const {
        return isSingle_;
    }
    size_t getNbChildren() const {
        return nbChildren_;
    }

    void setFirstName(const std::string& firstName) {
        firstName_ = firstName;
    }
    void setLastName(const std::string& lastName) {
        lastName_ = lastName;
    }
    void setSingle(const bool single) {
        isSingle_ = single;
    }
    void setNbChildren(const size_t children) {
        nbChildren_ = children;
    }

private:
    std::string firstName_;
    std::string lastName_;
    bool isSingle_;
    size_t nbChildren_;
};

Person::Person()
    : pimpl{std::make_unique<impl>()}
{}

Person::~~Person() = default;

```

```

Person::Person(const Person& other)
    : pimpl{std::make_unique<impl>(*other.pimpl)}
{}

Person::Person(Person&& other)
    : pimpl{std::make_unique<impl>(std::move(*other.pimpl))}
{}

Person& Person::operator=(const Person& other)
{
    *pimpl = *other.pimpl;
    return *this;
}

Person& Person::operator=(Person&& other)
{
    *pimpl = std::move(*other.pimpl);
    return *this;
}

const std::string& Person::getFirstName() const
{
    return pimpl->getFirstName();
}

const std::string& Person::getLastName() const
{
    return pimpl->getLastName();
}

bool Person::isSingle() const {
    return pimpl->isSingle();
}

size_t Person::getNbChildren() const {
    return pimpl->getNbChildren();
}

void Person::setFirstName(const std::string& firstName) {
    pimpl->setFirstName(firstName);
}

void Person::setLastName(const std::string& lastName) {
    pimpl->setLastName(lastName);
}

void Person::setSingle(const bool single) {
    pimpl->setSingle(single);
}

```

```
void Person::setNbChildren(const size_t nbChildren) {
    pimpl->setNbChildren(nbChildren);
}
```

Le fichier Person.h partagé avec l'application client n'expose finalement que des déclarations de fonctions et un pointeur const. Toute l'implémentation (encapsulation et traitement des données) se situe dans le fichier Person.cpp. Ainsi, la modification d'un traitement ou l'ajout d'une nouvelle donnée n'affecte pas les applications clientes. Toutefois, remarquez que pour les méthodes spéciales (destructeur, constructeurs de copie/déplacement et opérateurs de copie/déplacement), celles-ci doivent être déclarées dans le fichier Person.h car le pointeur est défini sur un type pré-déclaré (incomplet). Dans le cas contraire, une erreur de compilation avertira que l'opération est impossible. Par conséquent, Il faudra, à l'implémentation, définir le destructeur avec le mot-clé « default » pour obtenir le comportement par défaut, et implémenter la copie et le déplacement de l'objet encapsulé.

---

## DUAL INHERITANCE

Un autre cas d'utilisation nécessitant une ABI stable est la possibilité d'ajouter/modifier des fonctionnalités par l'ajout de plug-in. Prenons l'exemple d'un logiciel qui communique avec un récepteur GNSS (global navigation satellite systems). Il est probable que l'utilisateur du logiciel souhaite utiliser un récepteur spécifique comme récepteur GNSS (Par exemple, pour des raisons de sécurité, un utilisateur américain choisira plus probablement un modèle américain qu'un modèle chinois, et vice-versa. Ou, pour des raisons de performances, un utilisateur indien choisira plus probablement un modèle multi-GNSS pour profiter de sa constellation de satellites à couverture régionale). Ainsi, l'implémentation de gestion du récepteur GNSS pour un modèle particulier pourrait être encapsulée dans un plug-in, ce qui permet de changer de gestionnaire de récepteur GNSS sans modifier ou recompiler l'application hôte.

On peut l'implémenter de la manière suivante :

- GlobalNavigation.h

```
class GlobalNavigation
{
public:
    GlobalNavigation() = default;
    virtual ~GlobalNavigation() = default;

    GlobalNavigation(const GlobalNavigation&) = delete;
    GlobalNavigation(GlobalNavigation&&) = delete;

    GlobalNavigation& operator=(const GlobalNavigation&) = delete;
    GlobalNavigation& operator=(GlobalNavigation&&) = delete;

    using FixObserver = std::function<void(size_t)>;
    using PositionObserver = std::function<void(size_t,double,double,double)>;

    void start(FixObserver fixFct, PositionObserver posFct) {
        startImpl(fixFct, posFct);
    }

    void stop() {
        stopImpl();
    }
}
```

```

    }

private:
    virtual void startImpl(FixObserver fixFct, PositionObserver posFct) = 0;
    virtual void stopImpl() = 0;
};

```

Pour l'exemple, l'API se limite à deux méthodes, « start » et « stop ». La méthode « start » prend en paramètre deux fonctions « observer » : une première pour notifier de la fin de la recherche du nombre de signaux satellites et une seconde pour notifier de la position géographique avec la qualité du signal cycliquement. Une seconde interface est implémentée pour les récepteurs multi-GNSS :

- GlobalNavigationMultiGNSS.h

```

using ConstellationFlags = uint8_t;

namespace Constellation
{
    constexpr ConstellationFlags GPS      {0x1};
    constexpr ConstellationFlags GLONASS {0x1 << 1};
    constexpr ConstellationFlags BDS      {0x1 << 2};
    constexpr ConstellationFlags GALILEO  {0x1 << 3};
    constexpr ConstellationFlags QZSS     {0x1 << 4};
    constexpr ConstellationFlags NAVIC    {0x1 << 5};
};

class GlobalNavigationMultiGNSS : public virtual GlobalNavigation
{
public:
    void setConstellation(const ConstellationFlags flags) {
        setConstellationImpl(flags);
    }

private:
    virtual void setConstellationImpl(const ConstellationFlags flags) = 0;
};

```

Par la suite, si une autre fonctionnalité est ajoutée au récepteur GNSS, on ajoutera cette fonctionnalité avec une nouvelle interface héritant de « GlobalNavigationMultiGNSS ». Remarquez l'utilisation du mot clé « virtual » dans la déclaration de l'héritage, il est nécessaire car les classes d'implémentation hériteront deux fois de l'interface.

- GlobalNavigationAssistedGNSS.h

```

class GlobalNavigationAssistedGNSS : public virtual GlobalNavigationMultiGNSS
{
public:
    void loadEphemerisFile(const std::string& path) {
        loadEphemerisFileImpl(path);
    }

private:

```

```
virtual void loadEphemerisFileImpl(const std::string& path) = 0;
};
```

J'ai fini de déclarer mes interfaces API entre les plug-ins et le logiciel.

Mon premier plug-in gère un module GPS conçu par STMICROELECTRONICS. La déclaration de la classe implémentant l'interface est la suivante :

- STMicroElectronicsGPS.h

```
class STMicroElectronicsGPS : public virtual GlobalNavigation
{
private:
    void startImpl(FixObserver fixFct, PositionObserver posFct) override;
    void stopImpl() override;
};
```

Notez que ce fichier est interne au plug-in. L'application hôte appelle les services du GPS via l'interface « GlobalNavigation ».

Je peux définir un second plug-in pour un modèle, par exemple Broadcom multi-GNSS :

- BroadcomGNSS.h

```
class BroadcomGNSS : public virtual GlobalNavigation
{
private:
    void startImpl(FixObserver fixFct, PositionObserver posFct) override;
    void stopImpl() override;
};
```

- BroadcomMultiGNSS.h

```
class BroadcomMultiGNSS : public virtual GlobalNavigationMultiGNSS, public
BroadcomGNSS
{
private:
    // ...
    void setConstellationImpl(const ConstellationFlags flags) override;
};
```

L'implémentation est séparée en deux fichiers header internes au plug-in afin de permettre aux applications hôte ne gérant pas le multi-GNSS d'utiliser le plug-in avec un paramétrage de constellation par défaut. Remarquez également l'utilisation du mot clé « virtual », ici, la classe « BroadcomMultiGNSS » hérite deux fois de « GlobalNavigation » (une fois via l'interface « GlobalNavigationMultiGNSS » et une autre fois via la classe « BroadcomGNSS »). Notez que la conception se rapproche du patron de conception « Adapter » (La classe hérite à la fois d'une nouvelle interface « GlobalNavigationMultiGNSS » et de l'ancienne interface « BroadcomGNSS »)

Ok, je sens la question qui vous brûle les lèvres : « si les fichiers header implémentant les interfaces sont internes au plug-in, comment fait-on pour les instancier dans l'application hôte ? ».

Effectivement, il nous faut une fonction API renvoyant un pointeur sur les interfaces.

- LibraryGlobalNavigation.h

```
using GlobalNavigationPtr = std::unique_ptr<GlobalNavigation>;  
  
LIBRARY_API GlobalNavigationPtr createGlobalNavigationPtr();
```

- LibraryGlobalNavigationMultiGNSS.h

```
using GlobalNavigationMultiGNSSPtr = std::unique_ptr<GlobalNavigationMultiGNSS>;  
  
LIBRARY_API GlobalNavigationMultiGNSSPtr createGlobalNavigationMultiGNSSPtr();
```

L'implémentation est définie en interne des plug-ins :

- LibraryStMicroElectronics.cpp

```
GlobalNavigationPtr createGlobalNavigationPtr() {  
    return std::make_unique<STMicroElectronicsGPS>();  
}
```

- LibraryBroadcom.cpp

```
GlobalNavigationPtr createGlobalNavigationPtr() {  
    return std::make_unique<BroadcomGNSS>();  
}  
  
GlobalNavigationMultiGNSSPtr createGlobalNavigationMultiGNSSPtr() {  
    return std::make_unique<BroadcomMultiGNSS>();  
}
```

C'est la version simple. Ensuite, en fonction de comment on souhaite que l'application hôte gère l'instanciation des plug-ins, on pourra se tourner vers un autre patron de conception : Abstract Factory (les méthodes « create » seront remplacées par des méthodes « register » avec en paramètre un pointeur vers l'interface AbstractFactory. L'interface AbstractFactory faisant partie de l'API, elle suivra également la conception Dual Inheritance).

Pour être complet sur l'exemple, il y a aussi la question de comment charge-t-on un plug-in dans l'application hôte ? Je vois deux solutions : soit par création d'un lien symbolique (l'application hôte pointe vers le fichier GlobalNavigation.so qui est un lien symbolique vers BroadcomNavigation.so.1.0 par exemple) ou soit par chargement dynamique de la librairie par l'application hôte.

Vous l'aurez compris, la stabilité ABI est aussi obtenue parce que l'API n'expose que des pointeurs sur des interfaces.

---

## AUTRES CONSIDERATIONS

Cette partie a fait l'objet de règles sur la conception d'interface. Je souhaiterais indiquer quelques compléments avant la partie Architecture :

- Premièrement, aucune conception n'est parfaite. Il s'agit toujours de faire des compromis. Par exemple, ici, dans les deux conceptions Pimpl et Dual inheritance, l'ajout d'une indirection, les diverses allocations et la fragmentation mémoire en résultant, impacte les performances (\*).

- Deuxièmement, le choix d'une conception nécessite une expertise métier et une vision de comment va évoluer le logiciel. Il y a donc une règle à suivre si ces prérequis ne sont pas remplis : **éviter de concevoir prématurément**.

Je vois que vous êtes dubitatif et que vous souhaitez faire part d'un commentaire, d'ordre plus technique : « L'API des modules doit être « extern C », les solutions proposées, ici, sont des solutions C++ donc on ne peut pas les appliquer ». En réalité, on peut les appliquer en wrappant les fonctions API membres de classe dans des fonctions libres et en exposant les classes à l'aide de typedef sur une structure (plus d'informations sont disponibles dans la FAQ C/C++ [9]).

```
#ifdef __cplusplus
class Person
{
public:
    LIBRARY_API Person();
    LIBRARY_API ~Person();
    LIBRARY_API const std::string& getName() const;
    // ...
};
#else
typedef struct Person Person;
#endif

#ifdef __cplusplus
extern "C" {
#endif
LIBRARY_API Person* Person_new();
LIBRARY_API void Person_delete(Person*);
LIBRARY_API char* Person_getName(Person*);
#ifdef __cplusplus
}
#endif
```

*(\* Si la compatibilité ABI est trop impactante au niveau des performances et si la recompilation n'est pas impactante, la distribution du module sous forme d'une librairie statique est une solution. Les performances sont améliorées par suppression de l'indirection et seule la compatibilité API est à respecter. Exemple : module de trace, module de données métiers commun, ...)*



## C++ CORE GUIDELINES

- [I.1: Make interfaces explicit](#)
- [I.2: Avoid non-const global variables](#)
- [I.3: Avoid singletons](#)
- [I.4: Make interfaces precisely and strongly typed](#)
- [I.5: State preconditions \(if any\)](#)
- [I.6: Prefer `Expects\(\)` for expressing preconditions](#)
- [I.7: State postconditions](#)
- [I.8: Prefer `Ensures\(\)` for expressing postconditions](#)
- [I.9: If an interface is a template, document its parameters using concepts](#)
- [I.10: Use exceptions to signal a failure to perform a required task](#)



- [I.11: Never transfer ownership by a raw pointer \( \$T^\*\$ \) or reference \( \$T\&\$ \)](#)
- [I.12: Declare a pointer that must not be null as `not\_null`](#)
- [I.13: Do not pass an array as a single pointer](#)
- [I.22: Avoid complex initialization of global objects](#)
- [I.23: Keep the number of function arguments low](#)
- [I.24: Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning](#)
- [I.25: Prefer empty abstract classes as interfaces to class hierarchies](#)
- [I.26: If you want a cross-compiler ABI, use a C-style subset](#)
- [I.27: For stable library ABI, consider the Pimpl idiom](#)
- [I.30: Encapsulate rule violations](#)

## FONCTIONS

J'ai déjà beaucoup évoqué les bonnes pratiques sur les fonctions dans les parties précédentes « Style de codage » et « Interfaces ». Je me concentrerai donc que sur la définition d'une fonction et le passage d'arguments.

### DEFINITION D'UNE FONCTION

Les C++ Core Guidelines (F.1, F.2, F.3 et F.10) insistent sur le fait qu'une fonction doit rester courte et simple. Elle ne doit réaliser qu'une seule opération. En respectant ces principes, il sera plus facile de :

- trouver un nom significatif pour la fonction.
- réutiliser la fonction (principe « DRY »).

Lors de la définition d'une fonction, une bonne pratique est d'ajouter les spécificateurs « const » ou « constexpr » ou « noexcept » lorsqu'ils sont possibles.

Le mot-clé « const » ajouté à une méthode de classe permet à la méthode de classe de s'exécuter sur un objet déclaré « const ». Un exemple bien connu est les méthodes d'accès aux données membres « get ».

Si la fonction a le potentiel de s'exécuter au moment de la compilation, « constexpr » permet de bénéficier de deux avantages :

- Performance : le résultat de la fonction, évaluée au moment de la compilation, est directement stocké dans la mémoire en lecture seule.
- Thread-safe : une fonction « constexpr » est une fonction pure et si elle est évaluée pendant la compilation, le résultat est une valeur constante.

L'utilisation du mot-clé « noexcept » a aussi un double objectif :

- Performance : Cela permet au compilateur d'optimiser le code en ne gérant pas de chemin alternatif pour les exceptions.
- Documentaire : Cela indique à l'utilisateur de la fonction de ne pas se préoccuper d'éventuels exceptions provenant de la fonction. (en effet, déclarer une fonction « noexcept » ne signifie pas que l'on force la fonction à ne pas émettre d'exception. « noexcept » signifie qu'aucune exception ne devrait être émise par la fonction. Si une exception est émise, le programme se termine directement, sans possibilité de récupérer l'exception par une fonction plus haute dans la pile d'appel.)

Si je reprends mon exemple de la partie « style de codage » pour le nommage des fonctions, j'aurais dû écrire :

```
constexpr Percent calcPercentage(const Quantity part, const Quantity total) noexcept
{
    return (part / total) * 100.0;
}
```

*Note* : Si une fonction n'utilise que des fonctions C ou ne réalise que du calcul algébrique ou logique, elle devrait être systématiquement déclarée « noexcept ». Par exemple, la STL a déclaré toutes les fonctions « héritées » de la librairie standard C « noexcept ».

### PASSAGE D'ARGUMENTS

Le passage d'argument en C++ est extrêmement conventionnel. Il existe des exceptions mais il faut pouvoir démontrer leurs utilités. La manière conventionnelle est la plus lisible, la plus sûre, et la plus optimisée dans la plupart des cas. Voici un tableau qui récapitule les différents passages d'arguments :

	Cheap or impossible to copy (e.g., int, unique_ptr)	Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)	Expensive to move (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(const X&)		
In & retain "copy"	f(X)		

*"Cheap" ≈ a handful of hot int copies*

*"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation*

*\* or return unique\_ptr<X>/make\_shared\_<X> at the cost of a dynamic allocation*

On peut résumer le tableau de la manière suivante :

- Pour les variables de sortie, utilisez le type de retour sauf pour les tableaux d'objets de grande taille qui sont passés en paramètre par référence.
- Pour les variables d'entrée et de sortie, utilisez le passage en paramètre par référence.
- Pour les variables d'entrée, utilisez le passage en paramètre par référence constante sauf pour les types de petite taille (types primitifs) ou les types impossible à copier (unique\_ptr, thread) qui sont passés par valeur (et même par valeur constante pour les types de petite taille si on est plus rigoureux : f(const X)).

Il manque dans le tableau, un cas d'utilisation : le transfert de paramètre de type template. Dans ce cas, utilisez une référence universel : f(T&&) et transférez le paramètre à l'aide de la fonction std::forward<T>() à la fonction sous-jacente.

## SEMANTIQUE DE POSSESSION

Afin de compléter le tableau précédent, concentrons-nous sur les différents types de passage de paramètres : par valeur, par référence, par pointeur ou par pointeur intelligent. Selon la syntaxe que l'on va utiliser, une sémantique de possession de la ressource y est associée soit par le langage ou soit, par convention.

Exemple	Possession (celui qui possède la ressource, a le devoir de la libérer à la fin)
f(X)	L'appelant et f possèdent chacun une copie de la ressource. A la fin de f, la copie de f est automatiquement détruite.
f(X*)	L'appelant possède la ressource et prête à f un accès à la ressource. La ressource est optionnelle (si la ressource n'est pas présente, le pointeur vaut null_ptr). f doit vérifier si la ressource est présente et il n'est pas autorisé à libérer la ressource.
f(X&)	L'appelant possède la ressource et prête à f un accès à la ressource. La ressource n'est pas optionnelle (par conséquent, on préférera ce type de passage, si l'appelant souhaite seulement prêter une ressource). f n'est pas autorisé à libérer la ressource.

<code>f(std::unique_ptr)</code>	L'appelant donne explicitement la ressource à f. La ressource est automatiquement libérée à la fin de f sauf si f a déplacé la ressource via <code>std::move</code> à un <code>unique_ptr</code> d'une portée plus grande.
<code>f(std::shared_ptr)</code>	L'appelant et f partagent la ressource. A la fin de f, son partage de la ressource est supprimé. f peut agrandir le partage de la ressource en l'affectant à un <code>shared_ptr</code> d'une portée plus grande. Si à la fin de f, il est le dernier à avoir un partage de la ressource (l'appelant ne partage plus la ressource et f n'a pas affecté le <code>shared_ptr</code> ), la ressource est automatiquement libérée.



## C++ CORE GUIDELINES

- [F.1: "Package" meaningful operations as carefully named functions](#)
- [F.2: A function should perform a single logical operation](#)
- [F.3: Keep functions short and simple](#)
- [F.4: If a function might have to be evaluated at compile time, declare it `constexpr`](#)
- [F.5: If a function is very small and time-critical, declare it `inline`](#)
- [F.6: If your function must not throw, declare it `noexcept`](#)
- [F.7: For general use, take `T\*` or `T&` arguments rather than smart pointers](#)
- [F.8: Prefer pure functions](#)
- [F.9: Unused parameters should be unnamed](#)
- [F.10: If an operation can be reused, give it a name](#)
- [F.11: Use an unnamed lambda if you need a simple function object in one place only](#)
- [F.15: Prefer simple and conventional ways of passing information](#)
- [F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to `const`](#)
- [F.17: For "in-out" parameters, pass by reference to non-`const`](#)
- [F.18: For "will-move-from" parameters, pass by `X&&` and only `std::move` the parameter](#)
- [F.19: For "forward" parameters, pass by `TP&&` and only `std::forward` the parameter](#)
- [F.20: For "out" output values, prefer return values to output parameters](#)
- [F.21: To return multiple "out" values, prefer returning a struct or tuple](#)
- [F.60: Prefer `T\*` over `T&` when "no argument" is a valid option](#)
- [F.22: Use `T\*` or `owner<T\*>` to designate a single object](#)
- [F.23: Use a `not\_null<T>` to indicate that "null" is not a valid value](#)
- [F.24: Use a `span<T>` or a `span\_p<T>` to designate a half-open sequence](#)
- [F.25: Use a `zstring` or a `not\_null<zstring>` to designate a C-style string](#)
- [F.26: Use a `unique\_ptr<T>` to transfer ownership where a pointer is needed](#)
- [F.27: Use a `shared\_ptr<T>` to share ownership](#)
- [F.42: Return a `T\*` to indicate a position \(only\)](#)
- [F.43: Never \(directly or indirectly\) return a pointer or a reference to a local object](#)
- [F.44: Return a `T&` when copy is undesirable and "returning no object" isn't needed](#)
- [F.45: Don't return a `T&&`](#)
- [F.46: `int` is the return type for `main\(\)`](#)
- [F.47: Return `T&` from assignment operators](#)
- [F.48: Don't return `std::move\(local\)`](#)
- [F.49: Don't return `const T`](#)
- [F.50: Use a lambda when a function won't do \(to capture local variables, or to write a local function\)](#)
- [F.51: Where there is a choice, prefer default arguments over overloading](#)
- [F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms](#)
- [F.53: Avoid capturing by reference in lambdas that will be used non-locally, including returned, stored on the heap, or passed to another thread](#)
- [F.54: When writing a lambda that captures `this` or any class data member, don't use `\[=\]` default capture](#)

[F.55: Don't use `va\_arg` arguments](#)

[F.56: Avoid unnecessary condition nesting](#)

## CLASSES

Concernant les classes, il y a énormément de règles définies dans les C++ Core Guidelines. Nous verrons que les plus importantes. Une classe permet d'organiser les données au sein d'une unité.

### TYPE REGULIER

La première bonne pratique consiste à appliquer le principe KISS (« *keep it simple, stupid* »). Si le but de la classe est d'organiser des données, il n'y pas besoin de se préoccuper de virtualisation, d'héritage, de gestion de la mémoire. La classe doit se comporter comme un type régulier, c'est-à-dire qu'elle doit supporter les opérations suivantes :

- Construction par défaut
- Construction par copie
- Construction par déplacement
- Recopie par affectation
- Déplacement par affectation
- Destruction
- Egalité

Créer une classe se comportant comme un type régulier, à partir de types réguliers, est relativement simple. Par exemple, une classe représentant une position géographique peut être déclarée de la manière suivante :

```
class GeoPosition
{
public:
    GeoPosition() = default; // ❶
    constexpr GeoPosition(Degree latitude,
                          Degree longitude,
                          Meter altitude) noexcept; // ❷
    constexpr Degree latitude() const noexcept;
    constexpr Degree longitude() const noexcept;
    constexpr Meter altitude() const noexcept;
    constexpr bool ok() const noexcept;
    friend constexpr bool operator==(const GeoPosition& p1,
                                     const GeoPosition& p2) noexcept = default; // ❸,
C++20
private:
    Degree latitude_{0.0};
    Degree longitude_{0.0};
    Meter altitude_{0.0};
};
```

Vous êtes sûrement étonné que sur les six opérations obligatoires pour obtenir un type régulier, je n'ai déclaré que :

- le constructeur par défaut ❶.
- l'opérateur d'égalité ❸.

De plus, les implémentations des deux opérations sont générées par le compilateur grâce à la déclaration « default ». Cependant, les opérateurs « == » et « != » ne sont générés qu'à partir du C++20. Pour les versions C++ inférieures, la déclaration des opérateurs « == » et « != » et leurs implémentations sont nécessaires.

Les autres opérations de constructions, recopie, déplacement et destruction sont générées automatiquement, sans être déclarées.

Si dans l'exemple, on supprime le constructeur avec argument ❷, le constructeur par défaut ❶ n'a même plus besoin d'être déclaré. Si par contre, on ajoute un constructeur de recopie, les opérations de déplacement (construction et affectation) doivent être déclarées. On peut continuer comme ça les expérimentations et s'apercevoir qu'en fonction de certaines déclarations, d'autres déclarations sont nécessaires pour obtenir un type régulier. On ne va pas poursuivre nos investigations et on va faire confiance à une personne qui a fait le job (Howard Hinnant, pour ne pas la citer, dans sa présentation à la conférence ACCU 2014).

#### Déclarer implicitement par le compilateur

	Constructeur par défaut	Destructeur	Constructeur de recopie	Recopie par affectation	Constructeur de déplacement	Déplacement par affectation
Déclarer par l'utilisateur	Rien	Généré	Généré	Généré	Généré	Généré
	N'importe quel constructeur	Non déclaré	Généré	Généré	Généré	Généré
	Constructeur par défaut	Déclaration utilisateur	Généré	Généré	Généré	Généré
	Destructeur	Généré	Déclaration utilisateur	Généré	Généré	Non déclaré
	Constructeur de recopie	Non déclaré	Généré	Déclaration utilisateur	Généré	Non déclaré
	Recopie par affectation	Généré	Généré	Généré	Déclaration utilisateur	Non déclaré
	Constructeur de déplacement	Non déclaré	Généré	Supprimé	Supprimé	Déclaration utilisateur
	Déplacement par affectation	Généré	Généré	Supprimé	Supprimé	Non déclaré

Et là, je pense que vous vous dites : « Quel mal de tête, s'il faut retenir toutes les combinaisons de déclarations pour créer un type cohérent ». Non, on ne va pas retenir les combinaisons et on va se contenter de deux cas symbolisés par les règles :

- rule of zero
- rule of six

#### RULE OF ZERO

L'exemple, cité au-dessus, est une application de la « rule of zero ». Si les types sous-jacent sont simples (types qui ne nécessitent pas des appels spécifiques d'allocation et de libération de ressource), ne vous préoccupez pas de déclarer une ou des fonctions membres spéciales. Si besoin, vous pouvez déclarer le constructeur par défaut « = default » et les opérateurs d'égalité et d'inégalité qui sont à part, mais pas plus. Dans le cas contraire, en plus de faire du travail en plus, vous pouvez aussi nuire aux performances de l'application.

En reprenant mon exemple de la classe « GeoPosition », et en ajoutant la déclaration suivante :

```

class GeoPosition
{
public:
    // ...
    constexpr GeoPosition(const GeoPosition&) = default; // Ajout
    // ...
};

```

Cela supprime la génération des opérations de déplacement. Par conséquent, dans tous les cas où les opérations de déplacement étaient possibles et s'appliquaient, celles-ci sont remplacées par les opérations de recopie. Ici, le changement n'est peut-être pas si grave (3 doubles en données membres) mais imaginez si la classe contenait un « vector » avec des objets de grande taille à la place. Le déplacement consistait simplement à intervertir les deux pointeurs encapsulés dans les objets « vector » alors que la recopie consiste à allouer la mémoire suffisante dans le premier objet « vector », puis parcourir tout le second objet « vector » et recopier, une par une, chaque donnée contenue dans le second objet « vector » dans le premier. Quelle belle perte de performance !

Ensuite, vous vous posez peut-être la question : « si j'ai besoin d'ajouter un constructeur par défaut, pourquoi le déclarer « = default » ? Cela ne correspond pas à mon besoin d'initialisation. ». La raison est que l'initialisation des données membres doit être faite directement avec leur déclaration, dans la classe. Il est donc inutile de répéter leurs initialisations dans le constructeur par défaut. Cela simplifie aussi la maintenance dans le cas où la classe a plusieurs constructeurs (car l'initialisation par défaut d'une donnée n'est présente qu'à un seul endroit, les constructeurs ne modifient que les données membres impactés par ses arguments). Si dans l'exemple ci-dessus, j'ajoute un nouveau constructeur :

```

    constexpr GeoPosition(Degree latitude,
                          Degree longitude)
        : latitude_{latitude}, longitude_{longitude}
    {}

```

La donnée « altitude\_ » conserve sa valeur par défaut, indiquée à sa déclaration.

Donc, reprenez ces deux bonnes pratiques :

- Ne déclarez pas des fonctions membres spéciales s'il n'y a pas besoin.
- N'implémentez pas le constructeur par défaut. Ajoutez seulement « = default » à la déclaration et faites l'initialisation des données membres dans la classe.

#### RULE OF SIX (OU FIVE)

« Attends, tu nous as dit qu'il y a deux règles à savoir pour la déclaration des fonctions membres spéciales d'une classe. Au final, la « rule of zero » nous recommande de ne déclarer aucune fonction membre spéciale. Alors, c'est quoi la deuxième règle ? » Effectivement, la « rule of zero » recommande de ne pas déclarer de fonction membre spéciale **seulement si** on n'a pas besoin d'en déclarer une. Parfois, il est nécessaire de déclarer un destructeur pour libérer une ressource, effectuer une action lors de l'arrêt. Idem, la recopie peut nécessiter des traitements particuliers ou, elle n'a simplement pas de sens pour la classe. Dès lors, la « rule of six » nous indique qu'il faut définir ou supprimer **toutes** les fonctions membres spéciales. Comme on l'a vu, le constructeur par défaut est un peu à part donc on va plutôt dire « rule of five ». En effet, si l'on se trouve dans la nécessité de définir un destructeur, un constructeur de recopie, un constructeur de déplacement, un opérateur de recopie ou un opérateur de déplacement, c'est que toutes ces opérations ne sont pas triviales (car elles sont étroitement liées). Pour illustrer cette règle, imaginons la classe suivante :



```

class ItineraryFinder
{
public:
    ItineraryFinder(const GeoPosition& from,
                    const GeoPosition& to) noexcept;
    ~ItineraryFinder();
    ItineraryFinder(const ItineraryFinder&) = delete;
    ItineraryFinder(ItineraryFinder&& rhs);
    ItineraryFinder& operator=(const ItineraryFinder&) = delete;
    ItineraryFinder& operator=(ItineraryFinder&& rhs);
private:
    GeoPosition from_;
    GeoPosition to_;
    std::thread worker_;
    // ...
};

```

La classe encapsule un thread réalisant un traitement en asynchrone. Typiquement, la destruction de l'objet nécessite l'annulation du traitement et l'attente de la fin du thread pour éviter une fuite de ressource système. Quid des autres opérations ?

- Le constructeur de recopie : que signifie-la recopie d'un traitement asynchrone ? Un nouveau thread doit-il être créé avec les mêmes positions de départ et d'arrivée ? Cela pourrait être perturbant pour l'utilisateur de la classe si la classe est passée par valeur à une fonction, donc pour éviter toute création de thread implicite, la fonction membre spéciale est supprimée.
- Le constructeur de déplacement : on peut raisonnablement déplacer la ressource système représentant le thread vers le nouvel objet « ItineraryFinder » : Le traitement asynchrone continue, la propriété de la ressource système est simplement attribuée au nouvel objet. Le constructeur peut être utile si la classe est instanciée dans une fonction et transférée par retour de fonction.
- L'opérateur de recopie par affectation : idem que le constructeur de recopie, la fonction membre spéciale est supprimée.
- L'opérateur de déplacement par affectation : on a vu que le déplacement ne posait pas de problème. Il faudra, cependant, s'assurer d'annuler le traitement asynchrone et attendre la fin du thread de l'objet représenté par « this » au préalable.

« On est obligé de faire tout ça dans ton exemple. Je pensais que les classes de la STL implémentaient des mécanismes de RAI et donc, que l'on avait simplement à suivre la « rule of zero ». Effectivement, « std::thread » a déjà supprimé les opérateurs de recopie et fait l'implémentation du destructeur et des opérations de déplacement. Cependant, l'implémentation effectuée par « std::thread » ne peut pas annuler un traitement asynchrone car cela dépend de l'utilisateur de la classe (un « mutex » peut être pris dans le traitement asynchrone). Donc, « std::thread » appelle « std::terminate » au moment de la destruction ou d'un déplacement si le thread est en cours d'exécution. (S'il n'y a pas d'annulation du traitement asynchrone et seulement l'attente de la fin du thread est requis, on pourrait très bien appliquer la « rule of zero » en remplaçant « std::thread » par « std::jthread » ou utiliser une tâche et « std::future »).

On peut résumer la règle par deux bonnes pratiques :

- Déclarez toutes les fonctions membres spéciales si vous en définissez une.
- Soyez cohérent dans l'implémentation des fonctions membres spéciales.

## HIERARCHIES DE CLASSES

Le mécanisme d'héritage est fondamental à la programmation objet et le polymorphisme est essentiel pour concevoir une abstraction. En polymorphisme dynamique, les classes Interfaces et leurs fonctions virtuelles permettent de définir un contrat entre l'utilisation et l'implémentation. Cependant, l'héritage et le polymorphisme dynamique ajoute de la complexité.

Les problématiques sont les suivantes :

Problématiques	Bonnes pratiques
L'instanciation de classes interfaces pose un problème de slicing [10].	Les classes interfaces doivent supprimer les opérations de copie et de déplacement.
La destruction d'un type pointeur sur une classe interface ou une classe de base (*), alloué avec un type d'une classe dérivée provoque une fuite mémoire.	<p>Les classes interfaces doivent déclarer le destructeur « public » et « virtual ». Une classe interface est utilisée par l'appelant pour faire du polymorphisme dynamique et par conséquent, la manipulation via un pointeur sur la classe interface est autorisée et le destructeur de la classe dérivée est appelé.</p> <p>Les classes de base doivent déclarer le destructeur « protected » et non « virtual ». Une classe de base est seulement utilisée pour être étendue et par conséquent, la manipulation via un pointeur sur la classe de base est interdite.</p>
Il n'est pas possible de virtualiser des surcharges d'opérateurs, pour faire une copie ou vérifier l'égalité.	Si les opérations de copie et/ou d'égalité doivent être possibles de manière polymorphique depuis la classe d'interface, celle-ci doit proposer une méthode virtuelle pure « clone » pour la copie et/ou une méthode virtuelle pure « compare » pour l'égalité.
L'utilisation de la classe interface provoque l'ajout des headers liés à l'implémentation comme dépendance (include) de l'appelant.	Les classes interfaces doivent consister seulement à des méthodes « virtual » pures, un destructeur « virtual » par défaut et ne pas avoir de données membres.
La déclaration virtuelle dans les classes dérivées d'une méthode non virtuelle de la classe de base provoque un polymorphisme incohérent.	<p>Les déclarations des méthodes virtuelles ne doivent utiliser qu'un seul des mots clés suivants :</p> <ul style="list-style-type: none"> <li>- « virtual » : déclare une nouvelle méthode virtuelle qui peut être surchargée dans les classes dérivées.</li> <li>- « override » : vérifie que la méthode de la classe de base/interface est virtuelle et la surcharge.</li> <li>- « final » : vérifie que la méthode de la classe de base/interface est virtuelle, la surcharge et interdit la surcharge dans les classes dérivées.</li> </ul>
Les données membre « protected » de la classe de base permettent leurs modifications dans les classes dérivées sans réalisation de contrôle. Les méthodes de la classe de base ont un comportement indéfini.	Eviter les données membre « protected » dans la classe de base.
Les appels aux fonctions virtuelles dérivées dans les constructeurs et les destructeurs ont un comportement indéfini.	Les constructeurs et les destructeurs ne doivent pas appeler de fonctions virtuelles.

(\* La classe ancêtre est nommée différemment selon l'usage de l'héritage :

- La classe est nommée « classe de base » pour un usage lié à l'extension d'une classe.
- La classe est nommée « classe interface » pour un usage lié à du polymorphisme dynamique.)

Un exemple de classe interface avec l'application des bonnes pratiques, est le suivant :

```
class GlobalNavigation
{
public:
    GlobalNavigation() = default; // ❶
    virtual ~GlobalNavigation() = default;

    GlobalNavigation(const GlobalNavigation&) = delete;
    GlobalNavigation(GlobalNavigation&&) = delete;

    GlobalNavigation& operator=(const GlobalNavigation&) = delete;
    GlobalNavigation& operator=(GlobalNavigation&&) = delete;
    // ...
};
```

❶ Il est nécessaire de déclarer un constructeur par défaut car en supprimant, les constructeurs de déplacement et de copie, le constructeur par défaut n'est pas généré par le compilateur et la classe se retrouve sans constructeur.



#### C++ CORE GUIDELINES

- [C.1: Organize related data into structures \(structs or classes\)](#)
- [C.2: Use class if the class has an invariant; use struct if the data members can vary independently](#)
- [C.3: Represent the distinction between an interface and an implementation using a class](#)
- [C.4: Make a function a member only if it needs direct access to the representation of a class](#)
- [C.5: Place helper functions in the same namespace as the class they support](#)
- [C.7: Don't define a class or enum and declare a variable of its type in the same statement](#)
- [C.8: Use class rather than struct if any member is non-public](#)
- [C.9: Minimize exposure of members](#)
- [C.10: Prefer concrete types over class hierarchies](#)
- [C.11: Make concrete types regular](#)
- [C.12: Don't make data members const or references in a copyable or movable type](#)
- [C.20: If you can avoid defining any default operations, do](#)
- [C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all](#)
- [C.22: Make default operations consistent](#)
- [C.30: Define a destructor if a class needs an explicit action at object destruction](#)
- [C.31: All resources acquired by a class must be released by the class's destructor](#)
- [C.32: If a class has a raw pointer \(T\\*\) or reference \(T&\), consider whether it might be owning](#)
- [C.33: If a class has an owning pointer member, define a destructor](#)
- [C.35: A base class destructor should be either public and virtual, or protected and non-virtual](#)
- [C.36: A destructor must not fail](#)
- [C.37: Make destructors noexcept](#)
- [C.40: Define a constructor if a class has an invariant](#)
- [C.41: A constructor should create a fully initialized object](#)
- [C.42: If a constructor cannot construct a valid object, throw an exception](#)
- [C.43: Ensure that a copyable class has a default constructor](#)
- [C.44: Prefer default constructors to be simple and non-throwing](#)
- [C.45: Don't define a default constructor that only initializes data members; use member initializers instead](#)
- [C.46: By default, declare single-argument constructors explicit](#)

[C.47: Define and initialize member variables in the order of member declaration](#)  
[C.48: Prefer in-class initializers to member initializers in constructors for constant initializers](#)  
[C.49: Prefer initialization to assignment in constructors](#)  
[C.50: Use a factory function if you need “virtual behavior” during initialization](#)  
[C.51: Use delegating constructors to represent common actions for all constructors of a class](#)  
[C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization](#)  
[C.60: Make copy assignment non-`virtual`, take the parameter by `const&`, and return by non-`const&`](#)  
[C.61: A copy operation should copy](#)  
[C.62: Make copy assignment safe for self-assignment](#)  
[C.63: Make move assignment non-`virtual`, take the parameter by `&&`, and return by non-`const&`](#)  
[C.64: A move operation should move and leave its source in a valid state](#)  
[C.65: Make move assignment safe for self-assignment](#)  
[C.66: Make move operations `noexcept`](#)  
[C.67: A polymorphic class should suppress public copy/move](#)  
[C.80: Use `=default` if you have to be explicit about using the default semantics](#)  
[C.81: Use `=delete` when you want to disable default behavior \(without wanting an alternative\)](#)  
[C.82: Don’t call virtual functions in constructors and destructors](#)  
[C.83: For value-like types, consider providing a `noexcept` swap function](#)  
[C.84: A `swap` must not fail](#)  
[C.85: Make `swap` `noexcept`](#)  
[C.86: Make `==` symmetric with respect of operand types and `noexcept`](#)  
[C.87: Beware of `==` on base classes](#)  
[C.89: Make a `hash` `noexcept`](#)  
[C.90: Rely on constructors and assignment operators, not `memset` and `memcpy`](#)  
[C.100: Follow the STL when defining a container](#)  
[C.101: Give a container value semantics](#)  
[C.102: Give a container move operations](#)  
[C.103: Give a container an initializer list constructor](#)  
[C.104: Give a container a default constructor that sets it to empty](#)  
[C.109: If a resource handle has pointer semantics, provide `\*` and `->`](#)  
[C.120: Use class hierarchies to represent concepts with inherent hierarchical structure \(only\)](#)  
[C.121: If a base class is used as an interface, make it a pure abstract class](#)  
[C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed](#)  
[C.126: An abstract class typically doesn’t need a user-written constructor](#)  
[C.127: A class with a virtual function should have a virtual or protected destructor](#)  
[C.128: Virtual functions should specify exactly one of `virtual`, `override`, or `final`](#)  
[C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance](#)  
[C.130: For making deep copies of polymorphic classes prefer a virtual `clone` function instead of public copy construction/assignment](#)  
[C.131: Avoid trivial getters and setters](#)  
[C.132: Don’t make a function `virtual` without reason](#)  
[C.133: Avoid `protected` data](#)  
[C.134: Ensure all non-`const` data members have the same access level](#)  
[C.135: Use multiple inheritance to represent multiple distinct interfaces](#)  
[C.136: Use multiple inheritance to represent the union of implementation attributes](#)  
[C.137: Use `virtual` bases to avoid overly general base classes](#)  
[C.138: Create an overload set for a derived class and its bases with `using`](#)  
[C.139: Use `final` on classes sparingly](#)  
[C.140: Do not provide different default arguments for a virtual function and an overrider](#)  
[C.145: Access polymorphic objects through pointers and references](#)  
[C.146: Use `dynamic\_cast` where class hierarchy navigation is unavoidable](#)  
[C.147: Use `dynamic\_cast` to a reference type when failure to find the required class is considered an error](#)  
[C.148: Use `dynamic\_cast` to a pointer type when failure to find the required class is considered a valid alternative](#)

[C.149: Use `unique\_ptr` or `shared\_ptr` to avoid forgetting to `delete` objects created using `new`](#)  
[C.150: Use `make\_unique\(\)` to construct objects owned by `unique\_ptr`s](#)  
[C.151: Use `make\_shared\(\)` to construct objects owned by `shared\_ptr`s](#)  
[C.152: Never assign a pointer to an array of derived class objects to a pointer to its base](#)  
[C.153: Prefer virtual function to casting](#)  
[C.160: Define operators primarily to mimic conventional usage](#)  
[C.161: Use non-member functions for symmetric operators](#)  
[C.162: Overload operations that are roughly equivalent](#)  
[C.163: Overload only for operations that are roughly equivalent](#)  
[C.164: Avoid implicit conversion operators](#)  
[C.165: Use `using` for customization points](#)  
[C.166: Overload unary `&` only as part of a system of smart pointers and references](#)  
[C.167: Use an operator for an operation with its conventional meaning](#)  
[C.168: Define overloaded operators in the namespace of their operands](#)  
[C.170: If you feel like overloading a lambda, use a generic lambda](#)  
[C.180: Use `unions` to save Memory](#)  
[C.181: Avoid “naked” `unions`](#)  
[C.182: Use anonymous `unions` to implement tagged unions](#)  
[C.183: Don’t use a `union` for type punning](#)

## INSTRUCTIONS

Je ne vais pas m'étendre sur cette partie car ce sont des règles qui ont été soit déjà précédemment évoquées dans les parties §Divergence des langages C et C++ et §Principes génériques ou soit ce sont des règles de codage qui sont généralement automatisées.



### C++ CORE GUIDELINES

- [ES.1: Prefer the standard library to other libraries and to “handcrafted code”](#)
- [ES.2: Prefer suitable abstractions to direct use of language features](#)
- [ES.3: Don't repeat yourself, avoid redundant code](#)
- [ES.5: Keep scopes small](#)
- [ES.6: Declare names in for-statement initializers and conditions to limit scope](#)
- [ES.7: Keep common and local names short, and keep uncommon and non-local names longer](#)
- [ES.8: Avoid similar-looking names](#)
- [ES.9: Avoid ALL CAPS names](#)
- [ES.10: Declare one name \(only\) per declaration](#)
- [ES.11: Use `auto` to avoid redundant repetition of type names](#)
- [ES.12: Do not reuse names in nested scopes](#)
- [ES.20: Always initialize an object](#)
- [ES.21: Don't introduce a variable \(or constant\) before you need to use it](#)
- [ES.22: Don't declare a variable until you have a value to initialize it with](#)
- [ES.23: Prefer the `{ }`-initializer syntax](#)
- [ES.24: Use a `unique\_ptr<T>` to hold pointers](#)
- [ES.25: Declare an object `const` or `constexpr` unless you want to modify its value later on](#)
- [ES.26: Don't use a variable for two unrelated purposes](#)
- [ES.27: Use `std::array` or `stack\_array` for arrays on the stack](#)
- [ES.28: Use lambdas for complex initialization, especially of `const` variables](#)
- [ES.30: Don't use macros for program text manipulation](#)
- [ES.31: Don't use macros for constants or “functions”](#)
- [ES.32: Use ALL CAPS for all macro names](#)
- [ES.33: If you must use macros, give them unique names](#)
- [ES.34: Don't define a \(C-style\) variadic function](#)
- [ES.40: Avoid complicated expressions](#)
- [ES.41: If in doubt about operator precedence, parenthesize](#)
- [ES.42: Keep use of pointers simple and straightforward](#)
- [ES.43: Avoid expressions with undefined order of evaluation](#)
- [ES.44: Don't depend on order of evaluation of function arguments](#)
- [ES.45: Avoid “magic constants”; use symbolic constants](#)
- [ES.46: Avoid narrowing conversions](#)
- [ES.47: Use `nullptr` rather than `0` or `NULL`](#)
- [ES.48: Avoid casts](#)
- [ES.49: If you must use a cast, use a named cast](#)
- [ES.50: Don't cast away `const`](#)
- [ES.55: Avoid the need for range checking](#)
- [ES.56: Write `std::move\(\)` only when you need to explicitly move an object to another scope](#)
- [ES.60: Avoid `new` and `delete` outside resource management functions](#)
- [ES.61: Delete arrays using `delete\[\]` and non-arrays using `delete`](#)
- [ES.62: Don't compare pointers into different arrays](#)
- [ES.63: Don't slice](#)
- [ES.64: Use the `T{e}` notation for construction](#)
- [ES.65: Don't dereference an invalid pointer](#)

[ES.70: Prefer a `switch`-statement to an `if`-statement when there is a choice](#)  
[ES.71: Prefer a `range-for`-statement to a `for`-statement when there is a choice](#)  
[ES.72: Prefer a `for`-statement to a `while`-statement when there is an obvious loop variable](#)  
[ES.73: Prefer a `while`-statement to a `for`-statement when there is no obvious loop variable](#)  
[ES.74: Prefer to declare a loop variable in the initializer part of a `for`-statement](#)  
[ES.75: Avoid `do`-statements](#)  
[ES.76: Avoid `goto`](#)  
[ES.77: Minimize the use of `break` and `continue` in loops](#)  
[ES.78: Don't rely on implicit fallthrough in `switch` statements](#)  
[ES.79: Use `default` to handle common cases \(only\)](#)  
[ES.84: Don't try to declare a local variable with no name](#)  
[ES.85: Make empty statements visible](#)  
[ES.86: Avoid modifying loop control variables inside the body of raw `for`-loops](#)  
[ES.87: Don't add redundant `==` or `!=` to conditions](#)  
[ES.100: Don't mix signed and unsigned arithmetic](#)  
[ES.101: Use unsigned types for bit manipulation](#)  
[ES.102: Use signed types for arithmetic](#)  
[ES.103: Don't overflow](#)  
[ES.104: Don't underflow](#)  
[ES.105: Don't divide by integer zero](#)  
[ES.106: Don't try to avoid negative values by using `unsigned`](#)  
[ES.107: Don't use `unsigned` for subscripts, prefer `gsl::index`](#)

## GESTION DES RESSOURCES

Qu'est-ce qu'une ressource ? Une ressource est un composant qui est limitée en quantité ou qui doit être protégée. Généralement, les ressources sont fournies par le système d'exploitation. Il propose une quantité limitée de :

- Mémoire
- Sockets
- Processus
- Threads

Il convient de les libérer après leur acquisition et leur utilisation, pour le bon fonctionnement du système d'exploitation. Le système d'exploitation propose, aussi, la possibilité de partager les données : fichiers ou variables en mémoire, avec plusieurs processus ou threads. Les accès à ces ressources partagées doivent être protégés afin de préserver leur cohérence et par conséquent, le bon fonctionnement des processus ou des threads.

### RAII

Le principe RAII (*Resource Acquisition Is Initialization*) est un concept de base du langage C++. Il consiste à acquérir la ressource dans le constructeur et à libérer la ressource dans le destructeur. Le modèle d'une classe RAII est le suivant (\*) :

```
class ResourceGuard
{
public:
    explicit ResourceGuard(const RESOURCE res)
        : resource(res)
    {}
    ~ResourceGuard()
    {
        RELEASE(resource);
    }

private:
    RESOURCE resource;
};
```

*(\* J'entends votre exclamation : « Mais ça ne respecte pas la « rule of six/five » ! ». Vous avez tout à fait raison, il faut définir les constructeurs de déplacement et de copie et les opérateurs de déplacement et de copie. Cependant, je présente ici, un modèle de classe et donc, je ne peux pas décider comment définir ces fonctions membres spéciales. Cela dépend de la ressource et de son utilisation. Par exemple, la copie peut s'implémenter soit par l'acquisition d'une nouvelle ressource, soit par l'incrément d'un compteur de référence ou soit par la suppression de la fonction membre spéciale. Aussi, le déplacement nécessite un mécanisme pour transférer la ressource à une nouvelle ressource sans la libérer : soit par la copie de la ressource (pointeur ou « handle ») avec la possibilité d'invalider la ressource courante (nullptr ou « magic number » ou booléen) ou soit par la suppression de la fonction membre spéciale.)*

RAII est largement utilisé dans l'écosystème C++. La STL propose plusieurs exemples de RAII :

- Containers



- Smart Pointers
- Locks

La force du principe RAII repose sur le mécanisme de destruction qui est appelé automatiquement lorsque l'instance d'une classe sort de la portée de sa déclaration. La libération de la ressource devient ainsi déterministe. La première bonne pratique pour la gestion des ressources est donc, d'encapsuler dans une classe tout objet nécessitant des appels de fonctions pour son acquisition et sa libération.

Une autre bonne pratique liée au principe de RAII est de garder les portées de code petites. En effet, les portées conditionnent la durée de vie des objets déclarés à l'intérieur. Cette préoccupation est même indispensable concernant les objets encapsulant les locks :

```
volatile int g_process_counter = 0;
std::mutex g_process_counter_mutex; // protège g_process_counter

void incrementProcessCounter1()
{
    const std::lock_guard<std::mutex> lock(g_process_counter_mutex); // mauvais
    std::cout << "thread #" << std::this_thread::get_id() << " : call process()";
    process(++g_process_counter);
}

void incrementProcessCounter2()
{
    std::cout << "thread #" << std::this_thread::get_id() << " : call process()";
    int counter;
    {
        const std::lock_guard<std::mutex> lock(g_process_counter_mutex); // mieux
        counter = ++g_process_counter;
    }
    process(counter);
}
```

Dans le premier exemple de code, le mutex est pris tout au long de l'exécution de la fonction « incrementProcessCounter1() » (et de sa sous-fonction « process() »).

Dans le second exemple de code, le mutex est pris seulement pendant le temps d'exécution de l'instruction modifiant la donnée partagée.

## SMART POINTERS

J'aimerais faire un petit point d'attention sur les smart pointers car l'une des ressources communément utilisée dans la programmation est la mémoire. Par exemple, l'allocation de mémoire est utilisée pour la lecture de données dans un fichier, la réception de données sur un socket ou l'affichage dynamique de widget graphique. Néanmoins, l'utilisation des mots clés « new » et « delete » en C++ moderne est suspicieux car la STL vous propose des conteneurs ou des smart pointers qui vont gérer la mémoire de manière efficace et sûre. Toutefois, il y a quelques bonnes pratiques à connaître pour l'utilisation des smart pointers :

- Préférez « unique\_ptr » à « shared\_ptr ». « unique\_ptr » est le smart pointer pour remplacer le pointeur brute traditionnel (T\*). Il est aussi efficace en temps et en mémoire que son équivalent T\* alloué manuellement. Il est, cependant, boudé par les débutants qui utilisent à tort son confrère

« shared\_ptr » car « unique\_ptr » ne peut pas être copié. Pourtant, il peut être déplacé ou prêté (voir §Sémantique de possession). L'exemple suivant montre l'utilisation du « unique\_ptr » pour une utilisation efficace du temps CPU et de la mémoire, en remplacement du « shared\_ptr » :

```
// Mauvais
std::vector<std::shared_ptr<Element>> readElements(istream& in)
{
    std::vector<std::shared_ptr<Element>> elements;
    std::string s;
    while (in >> s)
    {
        if (s == "registry")
        {
            auto reg = std::make_shared<Registry>();
            reg->load();
            elements.push_back(reg); // ③ : Incrémentation-décrémentation du
compteur de référence de la ressource
        }
        else if (s == "command")
            elements.push_back(std::make_shared<Command>());
    }
    return elements;
}

void operate(std::shared_ptr<Element> ptr) // ④ : Incrémentation-
décrémentation du compteur de référence de la ressource
{
    if (ptr != nullptr)
    {
        std::cout << ptr->name() << std::endl;
    }
}

int main()
{
    std::vector<std::shared_ptr<Element>> elements = readElements(cin); // ①
    for (auto ptr : elements)
    {
        operate(ptr); // ②
    }
    return 0;
}
```

Dans l'exemple ci-dessus, les éléments sont alloués en début de la fonction « main() » dans la fonction de lecture « readElements() » ①. Les objets alloués sont passés à une sous-fonction « operate() » de la fonction « main() » ②. Les pointeurs alloués ne sont pas partagés entre deux objets ayant des durées de vie différentes. Les pointeurs ont la durée de vie de la fonction « main ». Donc, l'utilisation du « shared\_ptr » est abusive car elle consomme sans nécessité, de la mémoire et du temps CPU pour la gestion du compteur de référence (③ et ④).

```
// Mieux
```

```

std::vector<std::unique_ptr<Element>> readElements(istream& in)
{
    std::vector<std::unique_ptr<Element>> elements;
    std::string s;
    while (in >> s)
    {
        if (s == "registry")
        {
            auto reg = std::make_unique<Registry>();
            reg->load();
            elements.push_back(std::move(reg)); // ❸ : Déplacement de la
ressource
        }
        else if (s == "command")
            elements.push_back(std::make_unique<Command>());
    }
    return elements;
}

void operate(Element& ptr) // ❹ : Prêt de la ressource
{
    std::cout << ptr.name() << std::endl;
}

int main()
{
    std::vector<std::unique_ptr<Element>> elements = readElements(cin); // ❺
    for (const auto& ptr : elements)
    {
        if (ptr != nullptr)
        {
            operate(*ptr); // ❻
        }
    }
    return 0;
}

```

Après le remplacement avec « unique\_ptr », les pointeurs alloués sont bien déplacés de la sous-fonction « readElements() » au « vector » de la fonction « main() » (❺ et ❸). Les objets alloués sont ensuite passés par référence à la fonction « operate() » qui manipule l'instance d'un élément et qui n'a finalement pas besoin du pointeur (❻ et ❹).

- Utilisez les fonctions libres « make\_unique » ou « make\_shared » à la place des constructeurs. L'utilisation de ces fonctions permet de se prémunir de fuite mémoire si une exception est lancée après l'allocation de la mémoire (voir C++ Core Guidelines : R.13 [1]).



C++ CORE GUIDELINES

[R.1: Manage resources automatically using resource handles and RAII \(Resource Acquisition Is Initialization\)](#)

[R.2: In interfaces, use raw pointers to denote individual objects \(only\)](#)  
[R.3: A raw pointer \(a `T\*`\) is non-owning](#)  
[R.4: A raw reference \(a `T&`\) is non-owning](#)  
[R.5: Prefer scoped objects, don't heap-allocate unnecessarily](#)  
[R.6: Avoid non-`const` global variables](#)  
[R.10: Avoid `malloc\(\)` and `free\(\)`](#)  
[R.11: Avoid calling `new` and `delete` explicitly](#)  
[R.12: Immediately give the result of an explicit resource allocation to a manager object](#)  
[R.13: Perform at most one explicit resource allocation in a single expression statement](#)  
[R.14: Avoid `\[\]` parameters, prefer `span`](#)  
[R.15: Always overload matched allocation/deallocation pairs](#)  
[R.20: Use `unique\_ptr` or `shared\_ptr` to represent ownership](#)  
[R.21: Prefer `unique\_ptr` over `shared\_ptr` unless you need to share ownership](#)  
[R.22: Use `make\_shared\(\)` to make `shared\_ptr`s](#)  
[R.23: Use `make\_unique\(\)` to make `unique\_ptr`s](#)  
[R.24: Use `std::weak\_ptr` to break cycles of `shared\_ptr`s](#)  
[R.30: Take smart pointers as parameters only to explicitly express lifetime semantics](#)  
[R.31: If you have non-`std` smart pointers, follow the basic pattern from `std`](#)  
[R.32: Take a `unique\_ptr<widget>` parameter to express that a function assumes ownership of a `widget`](#)  
[R.33: Take a `unique\_ptr<widget>&` parameter to express that a function reseats the `widget`](#)  
[R.34: Take a `shared\_ptr<widget>` parameter to express shared ownership](#)  
[R.35: Take a `shared\_ptr<widget>&` parameter to express that a function might reseat the shared pointer](#)  
[R.36: Take a `const shared\_ptr<widget>&` parameter to express that it might retain a reference count to the object ???](#)  
[R.37: Do not pass a pointer or reference obtained from an aliased smart pointer](#)

## GESTION D'ERREUR

La gestion des erreurs se compose des activités suivantes :

- Détection d'une erreur
- Transmission de l'erreur au code de gestion
- Conservation de l'état de fonctionnement du programme
- Evitement des fuites de ressources

## CONCEPTION

On peut commencer par distinguer deux types d'erreurs :

- Les erreurs de programmations. Ces sont les erreurs communes à la programmation et évoquées dans la partie §Programmation défensive. On retrouve les erreurs :
  - o Violation de type (overflow, cast, ...)
  - o Fuite de ressource (fuite mémoire)
  - o Erreur de limite (accès d'un élément à l'extérieur du tableau)
  - o Erreur de cycle de vie (accès à un élément détruit)
  - o Erreur logique (effet de bord)
  - o Erreur d'interface (erreur de domaine)
- Les erreurs fonctionnelles. Ce sont les erreurs liées aux codes métier (cas d'utilisation non défini).

Nous nous intéressons ici, principalement qu'aux erreurs de programmation, puisque que ce sont des erreurs connues. La gestion des erreurs doit être prise en compte dès la conception du logiciel car l'interface de l'unité logicielle doit prendre en compte deux cas d'utilisation : le cas de fonctionnement nominal et le cas de fonctionnement non nominal. De plus, le langage C++ propose deux flux de communication pour transmettre une erreur : le retour de fonction et les exceptions.

Afin de simplifier l'interface logicielle, il est recommandé de gérer les erreurs le plus tôt possible. L'objectif est de définir une erreur qui soit compréhensible par l'appelant et de masquer la complexité de l'implémentation. La gestion de l'erreur peut s'accompagner d'une assertion ou d'un log pour détailler la raison de l'erreur. Dans le code ci-dessous, la fonction est interne à l'implémentation :

```
namespace Impl
{
    std::pair<bool, std::vector<std::string>>
    searchLine(const std::string& content, const std::string& pattern)
    {
        bool success{false};
        std::vector<std::string> res;
        if (!content.empty() && !pattern.empty())
        {
            try
            {
                const std::regex re(pattern);
                std::istringstream stream(content);
                std::string line;
                while (std::getline(stream, line))
                {
                    if (std::regex_search(line, re))
                    {

```

```

        res.push_back(line);
    }
}
success = true;
}
catch(const std::regex_error&)
{
    success = false; // ❶
}
}
return {success, std::move(res)};
}
}

```

L'exemple décrit une fonction permettant de rechercher toutes les lignes contenant un motif de chaîne de caractères dans un texte. L'erreur provenant de l'objet `std::regex` est attrapée en interne de la fonction ❶. L'appelant de la fonction n'a ainsi pas besoin de connaître le détail de l'implémentation (en particulier, l'utilisation de la librairie `std::regex`).

Egalement, il faut essayer de rester simple dans les vérifications, c'est-à-dire, de déterminer la logique qui minimise le nombre de vérifications. Hélas, il n'y a pas de recette miracle pour faire du code simple. Dans certains cas, la construction d'une table de vérité ou la technique des partitions d'équivalence permettra d'obtenir une vision claire de la répartition des cas d'erreur. De plus, une mauvaise conception de la gestion d'erreur peut rendre le code particulièrement compliqué à analyser et à maintenir. Par conséquent, elle ajoute un risque d'erreur supplémentaire, comme des vérifications manquantes ou des vérifications trop restrictives. Comme d'habitude, un petit exemple permet de mieux illustrer la problématique :

```

template<class T>
concept arithmetic = std::is_arithmetic_v<T>; // (*)

template<std::integral T> // ❷
T convertParameter(const std::string& parameter)
{
    return static_cast<T>(std::atoll(parameter.c_str())); // ❸
}

template<std::floating_point T> // ❹
T convertParameter(const std::string& parameter)
{
    return static_cast<T>(std::atof(parameter.c_str())); // ❺
}

template<std::integral T> // ❻
bool checkParameter(const T value, const T min, const T max)
{
    const bool isTooLow = value < min;
    const bool isTooHigh = value > max;
    return !(isTooLow || isTooHigh);
}

template<std::floating_point T> // ❼

```

```

bool checkParameter(const T value, const T min, const T max)
{
    const bool isNaN = std::isnan(value); // ⑧
    const bool isInf = std::isinf(value); // ⑨
    const bool isTooLow = value < min;
    const bool isTooHigh = value > max;
    return !(isNaN || isInf || isTooLow || isTooHigh);
}

template<arithmetic T> // ①
std::pair<bool, T> parseParameter(const std::string& parameter, const T min, const
T max)
{
    const T value = convertParameter<T>(parameter);
    return { checkParameter(value, min, max), value };
}

void print(const std::pair<bool,double>& pair)
{
    std::cout << std::boolalpha << pair.first << " " << pair.second << '\n';
}

int main()
{
    print(parseParameter("0.6", 0.6, 1.0)); // true 0.6
    print(parseParameter("INVALID", -100., 100.)); // true 0
    std::cout << checkParameter(std::numeric_limits<double>::infinity(),
                                -std::numeric_limits<double>::infinity(),
                                std::numeric_limits<double>::infinity())
              << '\n'; // false
    std::cout << checkParameter(std::numeric_limits<double>::quiet_NaN(),
                                -std::numeric_limits<double>::infinity(),
                                std::numeric_limits<double>::infinity())
              << '\n'; // false
    return 0;
}

```

L'exemple ci-dessus réalise la conversion et la vérification de paramètres numériques depuis un type « chaîne de caractères ».

D'un point de vue conceptuel, l'analyse des paramètres numériques est identique quelque-soit le type de destination ①. Cependant, la gestion d'erreur va contraindre l'implémentation et faire perdre la capacité de généralisation (②, ③, ④, ⑤). L'implémentation nécessite d'écrire des spécialisations pour les types flottants et entiers pour chaque sous-fonction. Les dépendances (voir §Architecture) sont inversés et l'utilisation de « static\_cast » est suspicieux (⑥, ⑦). L'erreur n'est pas gérée au niveau de la conversion. Par conséquent, la valeur « INVALID » est considérée correcte. De plus, la vérification des bornes est implémentée par vérification des cas d'erreurs : valeur hors borne, valeur différente de la valeur spéciale « NaN » (Not a Number) et valeur différente de la valeur spéciale « infinity » (⑧, ⑨). Dans le détail, l'implémentation vérifie que la valeur est hors borne. Or, toutes comparaisons avec la valeur « NaN » renvoie faux. Donc, la valeur « NaN » doit être retirée des cas valides. Cependant, ce n'est pas le cas avec la valeur « infinity ». En conséquence, la valeur

« infinity » est considérée incorrecte dans les bornes [« -infinity », « +infinity »]. La gestion de l'erreur peut être simplifiée en conservant la généralisation :

```
template<arithmetic T> // ③
std::pair<bool,T> convertParameter(const std::string& parameter)
{
    T value{};
    std::stringstream str(parameter);
    str >> value;
    return { (str.eof() && !str.fail()), value }; // ④
}

template<arithmetic T> // ②
bool checkParameter(const T value, const T min, const T max)
{
    return (min <= value) && (value <= max); // ⑤
}

template<arithmetic T> // ①
std::pair<bool, T> parseParameter(const std::string& parameter, const T min, const
T max)
{
    const auto [validity, value] = convertParameter<T>(parameter);
    return { (validity && checkParameter(value, min, max)), value };
}
```

L'analyse des paramètres numérique est identique quelque-soit le type de destination ①. La généralisation est conservée pour chaque sous-fonction (②, ③). La fonction de conversion renvoie une erreur et par conséquent, la valeur « INVALID » et considérée incorrecte ④. La vérification des bornes est implémentée selon si la valeur est à l'intérieur de son domaine de définition ⑤. Ainsi, il n'y a plus besoin de vérifier les valeurs spéciales pour les types flottant. Donc, la valeur « infinity » est correcte dans les bornes [« -infinity », « +infinity »] et la valeur « NaN » est incorrecte quelque-soit les bornes.

(\* Les exemples sont écrits en C++20 afin d'être plus lisibles. L'équivalent C++11 de la déclaration des template sans la notion de concept est la suivante : `template<typename T, std::enable_if<std::is_arithmetic<T>::value, bool>::type = true>`)

Enfin, l'utilisation des exceptions doit être strictement réservée à la gestion d'erreur et plus particulièrement à la violation d'invariant. Un invariant est une propriété qui est constamment vraie pendant l'exécution du programme [14]. Un exemple classique de lancement d'une exception est lorsque le constructeur d'une classe ne peut initialiser complètement un objet.

```
class SerialCom
{
public:
    SerialCom(const std::string& port,
              const BaudRate baudrate = BaudRate::B9600,
              const ByteSize bytesize = ByteSize::EightBits,
              const Parity parity = Parity::None,
              const StopBits stopbits = StopBits::One)
    {
```



```

    const int serial_port = open(port.c_str(), O_RDWR);
    if (serial_port < 0)
    {
        throw SerialComException(std::string{"Unable to open port "} + port);
    }
    // ...

```

Dans l'exemple ci-dessus, l'ouverture du port échoue et par conséquent, l'objet de communication série n'est pas initialisé. Aucun traitement relatif à l'objet n'est possible. L'erreur est propagée par une exception au code client jusqu'à avoir suffisamment de contexte pour la gérer.

## EXCEPTION

Les exceptions permettent un retour rapide en cas d'erreur. Cependant, il convient de les utiliser correctement pour ne pas ajouter des erreurs de programmation à la gestion d'erreur (et avoir une mise en abyme de la gestion d'erreur !)

Problématiques	Bonnes pratiques
Définir la donnée d'une exception.	Les exceptions sont un genre d'instruction « goto ». Les exceptions doivent être utilisées <b>seulement</b> pour la gestion d'erreur. Une ou plusieurs données peuvent être associées à l'exception mais celles-ci doivent être des données de contexte, pour permettre de gérer l'erreur. De plus, vous ne devez pas utiliser les types primitifs comme exception ou les exceptions de la STL pour vos propres exceptions. En effet, le but est d'ajouter du contexte pour la gestion de l'exception. Les exceptions avec les types primitifs sont trop obscures et les exceptions de la STL sont trop génériques. Vous devez donc définir vos propres classes d'exception.
Attraper les exceptions.	Les bonnes pratiques sur les classes et les hiérarchies de classes restent d'actualité pour les classes d'exceptions (Voir §Classes). Le passage d'une exception dans le « catch » implique le passage par référence constante (car une classe d'interface n'est pas instanciable). Si l'exception est modifiée alors le passage est fait seulement par référence et utilisez seulement « throw; » pour la renvoyer (et non pas « throw e; »).
Lancer une exception.	Là aussi, les bonnes pratiques sur la gestion de ressource restent d'actualité. Si une ressource est prise et qu'une exception est lancée avant la libération de la ressource alors il y a une fuite de ressource. Utilisez toujours un mécanisme RAII (conteneur STL, smart pointer, ...) pour éviter les fuites de ressources.
Spécifier les exceptions d'une fonction.	Il n'y a qu'une <b>seule</b> spécification d'exception pour les fonctions : <ul style="list-style-type: none"> <li>« noexcept » : la fonction n'envoie pas d'exception.</li> </ul> Ne <b>jamais</b> utiliser les spécifications d'exception dynamique throw() (void function() throw();). Par ailleurs, celles-ci ont été dépréciées en C++11 et supprimées dans les normes C++17 et C++20.

## CODES D'ERREURS

Dans la partie précédente, nous avons vu l'utilisation des exceptions pour gérer les erreurs d'invariants. Cependant, parfois, la gestion d'erreur par exception est un tantinet trop excessif ou interdite pour du code critique. Par exemple, la gestion d'erreur par exception n'est pas adaptée pour la classe « GeoPosition » décrite dans le §Type régulier. Néanmoins, la classe devrait vérifier la validité de la position, en contrôlant que les paramètres sont définis selon que :

- la latitude est comprise entre [-90. ; 90.] degrés,
- la longitude est comprise entre [-180. ; 180.] degrés
- l'altitude est comprise entre [-11000. ; 9000.] mètres

Il est certainement probable que lors du traitement des positions géographiques fournies par une source extérieure, des positions soient invalides et que le traitement souhaité consiste simplement à ignorer ces positions. Donc, dans ce genre de contexte, il est préférable d'encapsuler les données et de proposer une méthode « ok() » retournant un booléen et indiquant la validité de l'objet (Les C++ Core Guidelines désigne cette gestion d'erreur comme une façon de simuler une RAI). L'implémentation du constructeur « GeoPosition » avec paramètres et celle de la méthode « ok() » sont les suivantes :

```
constexpr GeoPosition::GeoPosition(Degree latitude,
                                   Degree longitude,
                                   Meter altitude) noexcept
: latitude_{latitude}, longitude_{longitude}, altitude_{altitude}
{}

constexpr bool GeoPosition::ok() const noexcept
{
    return (-90.0 <= latitude_) && (latitude_ <= 90.0)
        && (-180.0 <= longitude_) && (longitude_ <= 180.0)
        && (-11000.0 <= altitude_) && (altitude_ <= 9000.0);
}
```

La vérification de la validité est, donc, de la responsabilité de l'appelant. Par contre, la vérification n'est implémentée qu'à un seul endroit (principe « DRY »).

« D'accord, cela fonctionne pour gérer des erreurs sur des données mais comment fait-on pour gérer des erreurs dans des fonctions ? Je suppose, comme le chapitre s'intitule codes d'erreurs, qu'il faut utiliser des codes d'erreurs ».

Exact. Si on ne peut pas utiliser des exceptions, il faut utiliser des codes d'erreurs pour remonter le contexte de l'erreur à l'appelant. Ainsi, nous pouvons poursuivre notre exemple avec la classe « GeoPosition » et l'implémentation d'une fonction « ComputeDistance() » :

```
// ... Code à l'intérieur d'un namespace regroupant la classe GeoPosition et la
// fonction ComputeDistance()

constexpr Radian PI = std::numbers::pi; // ou std::acos(-1) avant C++20
constexpr Meter EARTH_RADIUS{6'378'137.0};

constexpr Radian toRadian(Degree angle) noexcept
{
    return angle * PI / 180.0;
```

```

}

enum class ErrorCode // ❶
{
    Unknown,
    Success,
    InvalidParameter,
    ComputationError
};

constexpr std::pair<ErrorCode, Meter> ComputeDistance(const GeoPosition& p1, const
GeoPosition& p2) noexcept
{
    ErrorCode err{ErrorCode::Unknown};
    Meter eDistance{0.0};
    if (!p1.ok() || !p2.ok())
    {
        err = ErrorCode::InvalidParameter; // ❷
    }
    else
    {
        const Radian dLongitude =
            toRadian(p2.longitude() - p1.longitude());
        const Radian latitude1 = toRadian(p1.latitude());
        const Radian latitude2 = toRadian(p2.latitude());
        const Radian aDistance =
            std::acos(std::sin(latitude1) * std::sin(latitude2) +
std::cos(latitude1) * std::cos(latitude2) * std::cos(dLongitude));
        eDistance = EARTH_RADIUS * aDistance;
        if (std::isnan(eDistance))
        {
            err = ErrorCode::ComputationError; // ❸
        }
        else
        {
            err = ErrorCode::Success; // ❹
        }
    }
    return { err, eDistance }; // ❺
}

std::ostream& operator<<(std::ostream& out, const GeoPosition& p)
{
    return out << "[" << p.latitude() << ", " << p.longitude()<< ", " << p.altitude()
<< "]";
}

// ... Fin du namespace

int main()
{

```

```

// (*)
constexpr GeoPosition home{47.08235792701292, 2.399191729224431, 0.0};
constexpr GeoPosition work{47.09682137236728, 2.3933981577909282, 0.0};
if (constexpr auto distance = ComputeDistance(home, work); distance.first ==
    ErrorCode::Success)
{
    std::cout << std::setprecision(16) << "Distance between Home : " << home
        << " and Work : " << work << " = "
        << distance.second << " Meters\n";
}
return 0;
}

```

Les bonnes pratiques sont toujours valables. Je ne l'ai pas indiqué systématiquement dans les exemples mais les types, les classes, les fonctions doivent être encapsulés dans un « namespace » (c'est d'autant plus important quand le nom d'un type est un peu générique comme « ErrorCode » ou quand on implémente une surcharge d'opérateur). Les codes d'erreurs sont des constantes ❶ qui font partie de l'interface logicielle mais aussi de l'implémentation et donc, introduisent un lien de dépendance entre l'implémentation et le code appelant. Une attention particulière doit, donc, être portée quant au niveau d'abstraction (voir §Architecture), à l'emplacement des déclarations et à la portée d'utilisation des codes d'erreurs. Il faut minimiser l'impact lorsque l'on ajoute ou modifie un code d'erreur pour ne pas recompiler l'intégralité de la solution logicielle. De plus, évitez de réutiliser des codes d'erreurs. Comme pour les exceptions, différenciez les contextes d'erreurs pour aider à la gestion (❷, ❸). Par contre, ne définissez qu'une seule valeur de code d'erreur pour le cas sans erreur ❹. L'utilisation des codes d'erreurs implique l'utilisation des retours multiples ; std::pair et std::tuple sont des bons candidats ❺.

*(\* Remarquez que la variable distance est constexpr. Donc, la fonction ComputeDistance() est exécutée lors de la compilation. Dans le cas d'utilisation des exceptions avec une fonction constexpr, il est impossible d'attraper une exception lancée par une fonction constexpr si celle-ci est exécutée à la compilation. Le compilateur arrête la compilation. Evidemment, dans notre cas où l'on souhaite ignorer les erreurs, notre solution d'utilisation des codes d'erreurs est donc totalement adaptée.)*

« Le code exemple est relativement simple mais comment fait-on si le code se complexifie avec de nombreux codes d'erreurs en retour de fonctions ? »

Vous avez raison. L'utilisation des codes d'erreurs a pour conséquence d'ajouter des conditions if-else et de rendre le code moins lisible. De plus, en cas d'erreur, il vaut mieux échouer vite que d'ajouter des erreurs à l'erreur précédente. Dans le code de l'exemple précédent, j'ai inconsciemment (vraiment ?) appliqué une non règle de codage des C++ Core Guidelines. La non règle est celle-ci « N'ayez qu'une seule instruction return à la fin de la fonction ». En appliquant les bonnes pratiques des C++ Core Guidelines, j'obtiens le code suivant :

```

constexpr std::pair<ErrorCode,Meter> ComputeDistance(const GeoPosition& p1, const
    GeoPosition& p2) noexcept
{
    if (!p1.ok() || !p2.ok())
    {
        return { ErrorCode::InvalidParameter, 0.0 };
    }

    const Radian dLongitude =
        toRadian(p2.longitude() - p1.longitude());
}

```

```

const Radian latitude1 = toRadian(p1.latitude());
const Radian latitude2 = toRadian(p2.latitude());
const Radian aDistance =
    std::acos(std::sin(latitude1) * std::sin(latitude2) +
std::cos(latitude1) * std::cos(latitude2) * std::cos(dLongitude));
    eDistance = EARTH_RADIUS * aDistance;

    if (std::isnan(eDistance))
    {
        return { ErrorCode::ComputationError, 0.0 };
    }

    return { ErrorCode::Success, eDistance };
}

```

Le code est plus lisible et en cas d'erreur, la fonction retourne immédiatement. Cependant, je ne respecte plus la règle MISRA C++:2008, « 6-6-5 - A function shall have a single point of exit at the end of the function » et les outils d'analyse statique continuent de détecter cette règle de codage. Alors que faire ?

En réalité, la règle des C++ Core Guidelines est « N'insistez pas pour n'avoir qu'une seule instruction return dans une fonction ». La nuance est là. Personnellement, j'ai relu du code où l'instruction « return » était placée dans des conditions if-else imbriqués ou en plein milieu d'une boucle. Donc, je ne jetterai pas le bébé avec l'eau du bain et la règle reste valide. Mais, dans le cas spécifique où justement l'instruction « return » permet d'éviter des imbrications de condition if-else et d'améliorer la lisibilité, alors la dérogation est justifiée.



#### C++ CORE GUIDELINES

- [E.1: Develop an error-handling strategy early in a design](#)
- [E.2: Throw an exception to signal that a function can't perform its assigned task](#)
- [E.3: Use exceptions for error handling only](#)
- [E.4: Design your error-handling strategy around invariants](#)
- [E.5: Let a constructor establish an invariant, and throw if it cannot](#)
- [E.6: Use RAII to prevent leaks](#)
- [E.7: State your preconditions](#)
- [E.8: State your postconditions](#)
- [E.12: Use `noexcept` when exiting a function because of a `throw` is impossible or unacceptable](#)
- [E.13: Never throw while being the direct owner of an object](#)
- [E.14: Use purpose-designed user-defined types as exceptions \(not built-in types\)](#)
- [E.15: Throw by value, catch exceptions from a hierarchy by reference](#)
- [E.16: Destructors, deallocation, `swap`, and exception type copy/move construction must never fail](#)
- [E.17: Don't try to catch every exception in every function](#)
- [E.18: Minimize the use of explicit `try/catch`](#)
- [E.19: Use a `final\_action` object to express cleanup if no suitable resource handle is available](#)
- [E.25: If you can't throw exceptions, simulate RAII for resource management](#)
- [E.26: If you can't throw exceptions, consider failing fast](#)
- [E.27: If you can't throw exceptions, use error codes systematically](#)
- [E.28: Avoid error handling based on global state \(e.g. `errno`\)](#)
- [E.30: Don't use exception specifications](#)
- [E.31: Properly order your `catch`-clauses](#)

## CONCURRENCE

Ce chapitre couvre les bonnes pratiques pour l'exécution de code dans un environnement multi-thread. Je ne ferai qu'effleurer la surface de ce champ de programmation. La programmation multi-thread est un domaine complexe et vaste. Les C++ Core Guidelines ciblent d'abord les non experts du domaine.

### GENERALITES

La première bonne pratique est de concevoir son code pour un environnement multi-thread. Je conçois que cette règle soit un peu vague et s'applique surtout à du code avec un potentiel de réutilisation. Par exemple, imaginons une classe `Frame` utilisée pour encapsuler les données provenant d'un moyen de communication. On souhaite détecter des possibles surcharges de la communication. Pour cela, chaque trame reçue dans un intervalle inférieur à 1 seconde est numérotée :

```
class Frame
{
public:
    explicit Frame(const std::vector<std::byte>& data)
        : data_(data)
    {
        auto frameTime = std::chrono::system_clock::now();
        if ((frameTime - lastFrameTime_) < std::chrono::seconds(1))
        {
            ++counter_;
            number_ = counter_;
        }
        else
        {
            counter_ = 0U;
        }
        lastFrameTime_ = frameTime;
    }
    // ...
private:
    uint32_t number_{0U};
    std::vector<std::byte> data_;
    static uint32_t counter_;
    static std::chrono::time_point<std::chrono::system_clock> lastFrameTime_;
};
```

Le code ci-dessus ne fonctionne pas dans un environnement multi-thread. Les variables « static » peuvent être modifiées simultanément par plusieurs threads. La lecture et l'écriture non synchronisées des variables partagées, provoquent la perte de la cohérence des données (data race). Le code a un comportement indéfini.

Plusieurs choix s'offrent à nous :

- 1 : Ajouter un `std::mutex` pour protéger la lecture et l'écriture des variables « static » :

```
explicit Frame(const std::vector<std::byte>& data)
    : data_(data)
{
```

```

        auto frameTime = std::chrono::system_clock::now();
        std::lock_guard<std::mutex> lock(mutex_)
        if ((frameTime - lastFrameTime_) < std::chrono::seconds(1))
        {
            ++counter_;
            number_ = counter_;
        }
        else
        {
            counter_ = 0U;
        }
        lastFrameTime_ = frameTime;
    }
    // ...
private:
    static std::mutex mutex_;
    // ...

```

La solution fonctionne et peut être ok avec le comportement désiré. Cependant, l'utilisation des attributs « static » laisse une mauvaise impression.

- 2 : Déclarer les attributs « static » avec le mot clé « thread\_local ». « thread\_local » associe la variable au thread et non plus, à tout le processus (chaque thread possède une copie de la variable et qui a la même durée de vie) :

```

explicit Frame(const std::vector<std::byte>& data)
: data_(data)
{
    auto frameTime = std::chrono::system_clock::now();
    if ((frameTime - lastFrameTime_) < std::chrono::seconds(1))
    {
        ++counter_;
        number_ = counter_;
    }
    else
    {
        counter_ = 0U;
    }
    lastFrameTime_ = frameTime;
}
// ...
private:
    // ...
    static thread_local uint32_t counter_;
    static thread_local std::chrono::time_point<std::chrono::system_clock>
lastFrameTime_;

```

La solution fonctionne et peut être ok avec le comportement désiré. Cependant, l'utilisation des attributs « thread\_local » semble limité à des besoins spécifiques (variables non partagées entre les threads mais ayant une portée globale à l'intérieur du thread).

- 3 : Modifier les types des attributs « static » en type « atomic » :

```

explicit Frame(const std::vector<std::byte>& data)
: data_(data)
{
    auto frameTime = std::chrono::system_clock::now();
    if ((frameTime - lastFrameTime_.load()) < std::chrono::seconds(1))
    {
        ++counter_;
        number_ = counter_;
    }
    else
    {
        counter_ = 0U;
    }
    lastFrameTime_ = frameTime;
}
// ...
private:
    // ...
    static std::atomic_uint32_t counter_;
    static std::atomic<std::chrono::time_point<std::chrono::system_clock>>
lastFrameTime_;

```

La solution ne fonctionne pas. Les attributs sont corrélés avec d'autres variables, les opérations d'échange de valeurs ne sont pas protégées. La programmation multi-thread avec des opérations atomiques est aussi appelée « lock-free programming ». Elle nécessite une expertise élevée de l'ordonnancement des instructions par le CPU.

Maintenant, revenons sur les deux solutions qui fonctionnent (1 et 2). Pourquoi laissent-elles une mauvaise impression ? Parce qu'elles ne respectent pas les principes « SOLID ». Les deux attributs « static » sont présents pour ajouter une logique de comportement à une classe qui n'a pour but que d'encapsuler des données ensembles. La logique de comportement devrait être extraite et portée par du code de plus bas niveau d'abstraction (voir §Architecture). Ainsi, on peut développer l'exemple en imaginant une utilisation possible de la classe Frame dans une architecture multi-thread. Chaque trame reçue est encapsulée et traitée par un service asynchrone :

```

class Frame
{
public:
    explicit Frame(const std::vector<std::byte>& data)
    : data_(data)
    {}
    // ...
private:
    std::vector<std::byte> data_;
};

namespace Impl::Server
{
    using time_point_t = std::chrono::time_point<std::chrono::system_clock>;
    bool isSupDuration(const time_point_t firstTime,
                       const time_point_t lastTime,

```



```

        const std::chrono::milliseconds duration)
    {
        return ((lastTime - firstTime) > duration);
    }
}

class Server
{
public:
    using serviceFct = std::function<void(const Frame&)>;
    explicit Server(const serviceFct& fct)
        : decoder(fct)
    {
        std::generate(
            poolThreads.begin(),
            poolThreads.end(),
            [this]() {
                return std::jthread(&Server::executeService, this); // 10
            });
    }

    ~Server()
    {
        {
            std::lock_guard<std::mutex> lock(mtx);
            end_program = true;
        }
        cv.notify_all();
    }

    Server(const Server&) = delete;
    Server& operator=(const Server&) = delete;

    Server(Server&&) = delete;
    Server& operator=(Server&&) = delete;

    void serve(ByteReader& reader)
    {
        uint32_t frameCounter{0U};
        Impl::Server::time_point_t lastFrameTime;
        std::vector<std::byte> data;
        while (reader.read(data))
        {
            ++frameCounter;
            auto frameTime = std::chrono::system_clock::now();
            if (Impl::Server::isSupDuration(lastFrameTime, frameTime,
std::chrono::seconds(1))) // 2
            {
                frameCounter = 0U;
            }
            lastFrameTime = frameTime;
        }
    }
}

```

```

        // if (frameCounter ...
        {
            std::lock_guard<std::mutex> lock(mtx); // ⑤
            buffers.push(std::move(data));
        }
        cv.notify_one();
    }
}

private:
void executeService()
{
    std::unique_lock<std::mutex> lock(mtx); // ⑥
    while (!end_program) {
        cv.wait(lock, [this]{ return end_program || !buffers.empty(); }); // ⑦
        if (!buffers.empty())
        {
            auto data = buffers.front();
            buffers.pop();
            lock.unlock();
            Frame frame(data); // ①
            decoder(frame);
            lock.lock();
        }
    }
}

static constexpr uint32_t MAX_THREAD{10U};
std::array<std::jthread, MAX_THREAD> poolThreads;
std::queue<std::vector<std::byte>> buffers; // ④
std::mutex mtx;
std::condition_variable cv;
bool end_program{false};
const serviceFct decoder; // ③
};

int main()
{
    FrameDecoderStrategy decoder;
    ByteReaderAdapter reader;
    Server s(decoder);
    s.serve(reader);
    return 0;
}

```

Dans l'exemple, la vérification de la surcharge de la communication est placée au niveau de la lecture ② et le décodage d'une trame est réalisé dans un service ①. La lecture est réalisée dans le thread « server ». Le traitement de chaque trame est isolé dans un thread « service ». Il n'y a donc plus nécessité de partager les variables « counter » et « lastFrameTime ». La synchronisation est ainsi simplifiée en appliquant les principes d'isolation du code. L'exemple est assez complexe. On verra le détail du code dans la suite du chapitre.

Je vois que vous fronchez les sourcils et que l'exemple ne vous convient pas pour illustrer la bonne pratique énoncée au début de la partie §Généralités. « Si le code doit être conçu pour un environnement multi-thread, j'aurais écrit ma classe Frame de la manière suivante, pour être garanti qu'elle fonctionne dans tous les cas ».

```
class Frame
{
public:
    explicit Frame(const std::vector<std::byte>& data)
        : data_(data)
    {}
    // ...
private:
    std::mutex mtx_; // protège les accès concurrentiels à data_
    std::vector<std::byte> data_;
};
```

Il y a deux problèmes avec ce code :

- Le premier problème est d'ordre technique. Les « `std::mutex` » ne sont pas des objets copiables ou déplaçables. Donc, vous limitez fortement l'utilisation de la classe Frame. Elle ne peut plus être utilisée comme une classe encapsulant une trame d'octets. Les objets en résultant, ne peuvent plus être manipulés facilement par divers algorithmes. C'est particulièrement embêtant car l'objet de sa conception était justement l'abstraction d'une trame d'octets.
- Le second problème est d'ordre conceptuel. Le code pose le même problème conceptuel que les « `static` » mais à l'extrême opposé. L'ajout du « `std::mutex` » impose une logique de comportement : la classe sert à partager une donnée entre deux threads. Si l'objet de la conception de la classe Frame est une abstraction d'une trame d'octets, le « `std::mutex` » doit être porté par du code de plus bas niveau d'abstraction.

La première bonne pratique est donc à prendre avec vigilance. Elle constitue, plus un point d'attention au moment de la conception qu'une bonne pratique au niveau de l'implémentation.

## SECTION CRITIQUE

Comme on l'a vu, le plus grand défi de la programmation multi-thread est le partage de données entre les threads. Plusieurs bonnes pratiques visent à encadrer cette situation :

Problématiques	Bonnes pratiques
Partager des données initialisées entre différents threads.	Si vous partagez des données en lecture seule, vous n'avez pas de situation de concurrence. Pas besoin de verrouillage. Si vous pouvez initialiser les variables avant le lancement des threads asynchrones et si vos données ne sont pas modifiables, partagez des variables const. (Voir exemple §Généralités, classe Server, ③) Exemple d'une capture par valeur, §Tâche, ⑧)
Partager des données entre différents threads.	Si vous devez partager des données entre thread, partagez les données par valeur (pour les données dont la recopie est économe). Il n'y a pas de situation de concurrence sur une donnée recopiée et la recopie de la donnée est plus économe que l'utilisation d'un mécanisme de verrouillage. Les techniques permettant

	<p>la transmission de copie de données sont la file de message ou les promesses.</p> <p>(Voir exemple d'une file de message, §Généralités, classe Server, ④)</p> <p>Exemple d'une promesse, §Tâche, ⑨)</p>
Partager une donnée entre différents thread avec un verrou.	<p>Si la donnée est utilisée par plusieurs threads (et la recopie n'est pas économe ou l'écriture est effectuée par plusieurs threads), la donnée doit être protégée par un verrou. L'utilisation du verrou est soumise à quelques règles :</p> <ul style="list-style-type: none"> <li>- Manipulez le verrou avec un mécanisme RAI : <code>std::lock_guard</code> (prise d'un mutex) ou <code>std::scoped_lock</code> (prise de plusieurs mutex) ou <code>std::unique_lock</code> (prise optionnelle d'un mutex).</li> <li>- Ne pas oublier de nommer <code>std::lock_guard</code> ou <code>std::scoped_lock</code>. Une variable sans nom est temporaire et est immédiatement détruite après la création.</li> <li>- Minimisez le nombre d'instructions encapsulés par le verrou.</li> <li>- Ne jamais appelez un code inconnu pendant la prise du verrou (callback).</li> </ul> <p>(Voir exemple §Généralités, classe Server, ⑤ et ⑥)</p>
Notifier la modification d'une donnée partagée avec une condition variable.	<p>L'utilisation des conditions variables n'est pas évidente et est sujette aux problèmes de Lost Wakeup et de Spurious Wakeup. Pour éviter ces problèmes, la condition variable doit être utilisée en combinaison avec une condition renvoyant un booléen protégé par un mutex.</p> <p>(Voir exemple §Généralités, classe Server, ⑦)</p>
Gérer le cycle de vie des données.	<p>Considérez le thread comme un conteneur :</p> <ul style="list-style-type: none"> <li>- Appelez la méthode <code>join()</code> avant la destruction d'un <code>std::thread</code> ou utilisez <code>std::jthread</code> (C++ 20)</li> <li>- Ne jamais appeler la méthode <code>detach()</code>.</li> </ul> <p>(Voir exemple §Généralités, classe Server, ⑩)</p>

## TACHE

Le C++ 11 a introduit la notion de tâche asynchrone. Cette technique est très utile lorsque que l'on souhaite réaliser un traitement asynchrone sans réutiliser le thread (contrairement à l'exemple §Généralités) comme pour du parallélisme ou un timer. La communication entre les tâches est simplifiée par l'utilisation de futures et de promesses. Une promesse est un objet qui ouvre un canal de communication entre deux threads. Le thread qui détient le future issu de la promesse, est notifié dès que l'autre thread définit la promesse.

Reprenons l'exemple de la partie §Généralités et détaillons la fonction `main()` avec une tâche qui réalise l'instanciation de la classe `Server` et l'appel à la méthode `serve()` en parallèle du thread principal. Afin d'arrêter proprement l'exécution, une promesse est déclarée et est définie dans le gestionnaire de signal `SIGINT` (\*). Ici, le thread principal réalise une attente de la réception du signal mais on pourrait implémenter un traitement parallèle.

```

std::promise<bool> promSignal; // 9

void signal_handler(int signal)
{
    promSignal.set_value(true);
}

int main()
{
    std::future<bool> futSignal = promSignal.get_future();
    std::signal(SIGINT, signal_handler);
    ByteReaderAdapter reader;
    const FrameConfig config;
    auto decoder = [config](const Frame& frame) { // 8
        uint32_t crcFrame = frame.computeCrc32(config.getCrcPolynomial());
        // ...
    };
    auto serverTask = std::async([decoder, &reader]() {
        Server s(decoder);
        s.serve(reader);
    });
    std::future_status status;
    do
    {
        status = futSignal.wait_for(std::chrono::seconds(1));
    }
    while (status != std::future_status::ready);
    reader.close();
    // RAII : serverTask.wait(); est appelé implicitement à la destruction.
    return 0;
}

```

La tâche est démarrée avec l'appel de la fonction « `std::async()` ». Cette fonction renvoie un future correspondant au retour de la fonction démarrée en asynchrone. De cette manière, on pourrait utiliser une tâche pour lancer une demande de service asynchrone (requête BDD, HTTP REST, ...) et obtenir la réponse via le future.

*(\* L'utilisation des signaux UNIX n'est pas recommandée pour de la communication. Ils sont cependant acceptables pour gérer l'arrêt d'un logiciel.)*

## LOCK-FREE PROGRAMMING

Les techniques de « lock-free programming » nécessitent une expertise élevée et l'étude approfondie de la littérature associée [13]. Elles s'appuient sur les opérations atomiques read-modify-write (RMW) implémentées par les processeurs. Elle nécessite souvent de mettre en place des solutions sur mesure (Assembleur) pour pallier aux problèmes suivants :

- Barrière de synchronisation
- Cohérence séquentielle
- Problème ABA
- Ordonnancement de la mémoire

Ces techniques dépendent du type de processeur. Il est donc déconseillé de les utiliser sauf en cas de nécessité (Voir exemple §Généralités, solution 3).



## C++ CORE GUIDELINES

- [CP.1: Assume that your code will run as part of a multi-threaded program](#)
- [CP.2: Avoid data races](#)
- [CP.3: Minimize explicit sharing of writable data](#)
- [CP.4: Think in terms of tasks, rather than threads](#)
- [CP.8: Don't try to use `volatile` for synchronization](#)
- [CP.9: Whenever feasible use tools to validate your concurrent code](#)
- [CP.20: Use RAII, never plain `lock\(\)`/`unlock\(\)`](#)
- [CP.21: Use `std::lock\(\)` or `std::scoped\_lock` to acquire multiple mutexes](#)
- [CP.22: Never call unknown code while holding a lock \(e.g., a callback\)](#)
- [CP.23: Think of a joining thread as a scoped container](#)
- [CP.24: Think of a thread as a global container](#)
- [CP.25: Prefer `gsl::joining\_thread` over `std::thread`](#)
- [CP.26: Don't `detach\(\)` a thread](#)
- [CP.31: Pass small amounts of data between threads by value, rather than by reference or pointer](#)
- [CP.32: To share ownership between unrelated threads use `shared\_ptr`](#)
- [CP.40: Minimize context switching](#)
- [CP.41: Minimize thread creation and destruction](#)
- [CP.42: Don't `wait` without a condition](#)
- [CP.43: Minimize time spent in a critical section](#)
- [CP.44: Remember to name your `lock\_guards` and `unique\_locks`](#)
- [CP.50: Define a `mutex` together with the data it guards. Use `synchronized\_value<T>` where possible](#)
- [CP.51: Do not use capturing lambdas that are coroutines](#)
- [CP.52: Do not hold locks or other synchronization primitives across suspension points](#)
- [CP.53: Parameters to coroutines should not be passed by reference](#)
- [CP.60: Use a `future` to return a value from a concurrent task](#)
- [CP.61: Use `async\(\)` to spawn concurrent tasks](#)
- [CP.100: Don't use lock-free programming unless you absolutely have to](#)
- [CP.101: Distrust your hardware/compiler combination](#)
- [CP.102: Carefully study the literature](#)
- [CP.110: Do not write your own double-checked locking for initialization](#)
- [CP.111: Use a conventional pattern if you really need double-checked locking](#)
- [CP.200: Use `volatile` only to talk to non-C++ memory](#)

## PERFORMANCE

Bizarrement, dans cette partie, je vais plus insister sur les avertissements que sur les règles permettant d'obtenir des optimisations. D'ailleurs, les C++ Core Guidelines mettent en garde le lecteur à suivre naïvement les règles de performance qui y sont détaillées. On peut résumer les avertissements par cette célèbre citation :

« *The real problem is that programmers have spent far too **much time worrying about efficiency in the wrong places and at the wrong times**; premature optimization is the root of all evil (or at least most of it) in programming.* »

Donald Knuth, *The Art of Computer Programming* (1974)

## MAUVAISES OPTIMISATIONS

Le C++ est un langage très souple, certainement, par le fait qu'il hérite d'un ancêtre commun du langage C. il est ainsi facile d'écrire un service de différentes manières. En partant de ce postulat, certains pourront avancer comme hypothèse que le fait d'utiliser les fonctions bas-niveau permet d'avoir de meilleures performances.

Pour illustrer que ces idées reçues en termes de performance se révèlent inexactes, considérons l'exemple suivant :

```
constexpr uint64_t N {200'000'000ULL}; // ①
constexpr uint64_t NLoop {1ULL};
using iteration_t = std::array<uint64_t, NLoop>;

uint64_t testDynamicArray(const uint64_t N, const iteration_t& request)
{
    // Allocation et initialisation
    uint64_t* golomb = new uint64_t[N]; // ④
    golomb[0] = 0; // Indéfini
    golomb[1] = 1;
    for (uint64_t i = 2; i < N; i++)
    {
        golomb[i] = 1 + golomb[i - golomb[golomb[i - 1]]]; // ②
    }

    uint64_t res = 0U;
    // Iterations de la recherche
    for (uint64_t bound : request)
    {
        // Recherche de l'index de la limite inférieure
        bool found = false;
        uint64_t i = 0U;
        while (!found && (i < N))
        {
            if (golomb[i] >= bound) // ③
            {
                res = i;
                found = true;
            }
            i++;
        }
    }
}
```

```

    }
}
delete [] golomb;
return res;
}

int main()
{
    iteration_t requests;
    std::random_device rd;
    std::mt19937 mte(rd());
    std::uniform_int_distribution<uint64_t> dist(0, N / 2);
    std::generate(requests.begin(), requests.end(), [&] () { return dist(mte); });

    auto begin = std::chrono::system_clock::now();
    uint64_t res = testDynamicArray(N, requests);
    auto durationMs =
std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now() - begin);
    std::cout << " time : " << durationMs.count() << std::endl;

    return 0;
}

```

Le programme ci-dessus alloue 200 000 000 d'entier 64 bits ❶. Les valeurs sont initialisées selon une suite de Golomb ❷ [11] (suite croissante d'entier naturel, le  $n^{\text{e}}$  terme de la suite de Golomb est le nombre d'occurrences de l'entier  $n$  dans cette suite). Ensuite, le programme fait une recherche d'un nombre généré aléatoirement dans la suite (quel est le premier entier  $n$  qui a  $x$  occurrences dans la suite ?) ❸. Vous l'avez remarqué : la fonction « testDynamicArray() » qui réalise le traitement, est implémentée avec un « c-array » ❹. Car je prends l'hypothèse que « c-array » sera plus rapide qu'un « std::vector ».

Après exécution du code, le temps d'exécution de la fonction « testDynamicArray() » est de **1 667 millisecondes**. (Ne faites pas attention au temps car cela dépend du matériel, du compilateur et de sa version, des paramètres de compilations, de la version du C++, ... Seule la comparaison des temps d'exécution après les modifications du programme est importante).

Je modifie le programme pour remplacer le « c-array » par un « std::vector ». Le code devient :

```

uint64_t testVector(const uint64_t N, const iteration_t& request)
{
    // Allocation et initialisation
    std::vector<uint64_t> golomb(N);
    golomb[0] = 0; // Indéfini
    golomb[1] = 1;
    std::generate(
        std::next(golomb.begin(), 2U),
        golomb.end(),
        [i = 1, &golomb]() mutable {
            ++i;
            return 1 + golomb[i - golomb[golomb[i - 1]]];
        });
}

```



```

// Iterations de la recherche
uint64_t res = 0U;
for (uint64_t bound : request)
{
    // Recherche de l'index de la limite inférieure
    auto ite = golomb.begin();
    if (auto ite = std::find(golomb.begin(), golomb.end(), bound); ite !=
golomb.end())
    {
        res = std::distance(golomb.begin(), ite);
    }
}
return res;
}

```

Après exécution du code, le temps d'exécution de la fonction « **testVector()** » est de **2 058 millisecondes**. Vous vous dites que mon exemple tombe à plat car effectivement le « c-array » est plus rapide que « std::vector » et que je vais essayer de me justifier en trouvant des excuses :

- « std::vector » initialise les valeurs à 0 à la construction. « c-array » n'initialise pas les données.
- « std::vector » fournit des méthodes pour modifier/ajouter/supprimer facilement des données. Toute manipulation du « c-array » est à implémenter soi-même.
- Les données du « std::vector » sont encapsulées. Par exemple, un overflow dans la modification d'une donnée ne va pas modifier une donnée voisine.
- 391 millisecondes de temps d'initialisation supplémentaire, ce n'est pas non plus une durée excessive pour, je le rappelle, 200 000 000 valeurs.

Vous balayez mes justifications d'un revers de main : c'est les performances qui comptent !

De toute façon, je n'avais pas terminé ma démonstration. Je n'ai pas mis 200 000 000 valeurs en mémoire pour faire seulement une recherche ! Dans le code de l'exemple, je modifie la variable « NLoop ».

```
constexpr uint64_t NLoop {200ULL};
```

Les temps d'exécution sont les suivants :

- « **testDynamicArray()** » : **27 206 millisecondes**
- « **testVector()** » : **25 076 millisecondes**

Surprise ! Le parcours des valeurs dans « std::vector » est plus rapide d'au moins 2 secondes, qu'un parcours dans « c-array ». Comparativement, à l'initialisation où l'écart de temps entre « std::vector » et « c-array » était de l'ordre de la centaine de milliseconde, l'écart est maintenant de l'ordre de la seconde dans l'autre sens.

Si on synthétise les résultats démontrés avec l'exemple :

- Un code bas niveau n'est pas forcément plus rapide qu'un code de plus haut niveau.
- Cela ne sert à rien de faire du code compliqué pour essayer d'optimiser.
- Les hypothèses d'optimisation ne valent rien si celles-ci ne sont pas mesurées car il n'est pas possible de déduire le gain à l'avance.

Vous m'avez sans doute vu venir avec mes gros sabots. On cherche à optimiser en discutant sur du détail d'implémentation mais l'algorithme de recherche dans la suite de Golomb n'est pas du tout adapté. La suite est croissante (donc les valeurs sont triées). Une recherche par dichotomie serait beaucoup plus adaptée. Je ne vous

refait pas la démonstration de l'optimisation de différentes implémentations car l'implémentation la plus simple (avec l'utilisation de la fonction « `std::lower_bound()` ») sera certainement suffisamment optimisée. Le code est le suivant :

```
uint64_t testVector2(const uint64_t N, const iteration_t& request)
{
    // Allocation et initialisation
    std::vector<uint64_t> golomb(N);
    golomb[0] = 0; // Indéfini
    golomb[1] = 1;
    std::generate(
        std::next(golomb.begin(), 2U),
        golomb.end(),
        [i = 1, &golomb]() mutable {
            ++i;
            return 1 + golomb[i - golomb[golomb[i - 1]]];
        });

    // Iterations de la recherche
    uint64_t res = 0U;
    for (uint64_t bound : request)
    {
        // Recherche de l'index de la limite inférieure
        auto ite = golomb.begin();
        uint64_t res = 0U;
        if (auto ite = std::lower_bound(golomb.begin(), golomb.end(), bound); ite
        != golomb.end())
        {
            res = std::distance(golomb.begin(), ite);
        }
    }
    return res;
}
```

Le temps d'exécution avec la variable « `NLoop = 200` » est de **1 901 millisecondes**. Il n'y a pas de contestation possible, en changeant d'algorithme, le gain est de 23 secondes.

Par conséquent, on peut ajouter que :

- Cela ne sert à rien de dépenser du temps et des efforts pour gagner quelques microsecondes sur l'implémentation qui sont imperceptibles pour l'utilisateur final. (Mon exemple est délibérément exagéré, pour faire apparaître des temps de traitement de l'ordre de la seconde, voire de la dizaine de seconde. Dans la plupart des cas, les choix d'implémentation liés au langage vont faire apparaître des écarts de l'ordre de la nanoseconde ou de la microseconde)
- Les gains d'optimisation perceptibles se feront toujours, d'abord, sur la conception et les algorithmes.

## APPLICATION DES BONNES PRATIQUES

En réalité, j'ai déjà évoqué beaucoup d'optimisations dans les parties précédentes :


- Fonctions :

- Utiliser le mot-clé « constexpr » pour réaliser des traitements à la compilation et non plus à l'exécution.
- Utiliser le mot-clé « noexcept » pour aider le compilateur à générer un code plus optimisé.
- Respecter le passage conventionnel d'arguments en C++ pour éviter des recopies d'objet et pour effectuer des déplacements.
- Classes :
  - Respecter la « Rule of zero » et la « Rule of six » pour éviter le remplacement des opérations de déplacement par des opérations de copie.
  - Utiliser le mot-clé « final » pour aider le compilateur à générer un code plus optimisé.
- Gestion des ressources :
  - Utiliser des mécanismes de RAII pour gérer la mémoire de manière déterministe.
  - Utiliser « unique\_ptr » à la place de « shared\_ptr »

On peut, maintenant, revenir sur l'exemple cité dans la partie §Mauvaises optimisations. Il y a sans doute une question que vous vous posez : « La STL propose différents types de conteneur, on a mesuré le temps d'exécution d'une recherche d'une valeur dans « std::vector ». Est-ce qu'il y a un conteneur plus optimisé pour ce type de traitement ? Et selon le traitement que l'on fait, y-a-t-il des conteneurs préférentiels à utiliser ? ».

Les architectures modernes de CPU sont optimisées pour la lecture contiguë de la mémoire ([12] , C++Core Guidelines : SL.con [1]). Par conséquent, les conteneurs préférentiels à utiliser pour du traitement générique (remplissage et parcours) sont :

- std::array si vous connaissez la taille exacte du tableau à la compilation (allocation statique)
- std::vector pour de l'allocation dynamique d'un tableau. Utilisez la méthode « reserve() » si le tableau est rempli au fur et à mesure et si vous connaissez la taille maximum, pour pré-allouer la mémoire et éviter les réallocations qui sont coûteuses en temps.
- std::string pour une chaîne de caractères. Utilisez la méthode « reserve() » si la chaîne de caractères est construite au fur et à mesure et si vous connaissez la taille maximum, pour pré-allouer la mémoire et éviter les réallocations.



## C++ CORE GUIDELINES

[Per.1: Don't optimize without reason](#)

[Per.2: Don't optimize prematurely](#)

[Per.3: Don't optimize something that's not performance critical](#)

[Per.4: Don't assume that complicated code is necessarily faster than simple code](#)

[Per.5: Don't assume that low-level code is necessarily faster than high-level code](#)

[Per.6: Don't make claims about performance without measurements](#)

[Per.7: Design to enable optimization](#)

[Per.10: Rely on the static type system](#)

[Per.11: Move computation from run time to compile time](#)

[Per.12: Eliminate redundant aliases](#)

[Per.13: Eliminate redundant indirections](#)

[Per.14: Minimize the number of allocations and deallocations](#)

[Per.15: Do not allocate on a critical branch](#)

[Per.16: Use compact data structures](#)

[Per.17: Declare the most used member of a time-critical struct first](#)

[Per.18: Space is time](#)

[Per.19: Access memory predictably](#)

[Per.30: Avoid context switches on the critical path](#)

## TEMPLATE

Les C++ Core Guidelines possèdent beaucoup de règles concernant les Template. Je vais me focaliser sur les règles de bons usages, d'interfaces et de définitions des Template.

Aussi, avant de créer ses propres fonctions template, il est important de vérifier dans la STL si la fonction n'existe pas. Effectivement, la STL propose déjà une profusion d'algorithmes, de types traits, de fonctions, de conteneurs, etc.

## USAGE

De mon point de vue, il y a deux écueils à éviter pour un débutant :

- Utiliser les template sans faire de généralisation.
- Vouloir tout généraliser.

La définition de la programmation générique indiquée dans Wikipedia [15] éclaire bien l'objectif d'utilisation des templates en C++. Voici la définition :

*« En programmation, la généricité (ou programmation générique), consiste à définir des algorithmes identiques opérant sur des données de types différents. On définit de cette façon des procédures ou des types entiers génériques. On pourrait ainsi programmer une pile, ou une procédure qui prend l'élément supérieur de la pile, indépendamment du type de données contenues. »*

*C'est donc une forme de polymorphisme, le « polymorphisme de type » dit aussi « paramétrage de type » : en effet, le type de donnée général (abstrait) apparaît comme un paramètre des algorithmes définis, avec la particularité que ce paramètre-là est un type. C'est un concept important pour un langage de haut niveau car il permet d'écrire des algorithmes généraux opérant sur toute une série de types : la généricité augmente donc le niveau d'abstraction des programmes écrits dans un langage qui possède cette fonctionnalité. »*

Les bonnes pratiques concernant l'usage des templates reprennent cette définition et sont les suivantes :

- Utilisez les templates pour élever le niveau d'abstraction du code. J'ai déjà évoqué un exemple dans le chapitre Gestion d'erreur, partie §Conception. L'exemple portait sur l'utilisation de types que l'on pouvait généraliser (Integral, Float => Arithmetic). L'exemple qui suit, porte sur les opérations qui peuvent aussi, parfois, être généralisées :

```
template<Incrementable T>
T sum1(const std::vector<T>& v, T s) {
    for (auto x : v) s += x;
    return s;
}

template<Addable T>
T sum2(const std::vector<T>& v, T s) {
    for (auto x : v) s = s + x;
    return s;
}
```

- Les concepts Incrementable et Addable sont trop spécifiques. Les deux concepts s'appuient sur les opérations spécifiques += et +, alors que sémantiquement les deux opérations sont des additions. D'ailleurs, on peut aller plus loin, les opérations d'additions sont généralisables à tous les types arithmétiques.

- Aussi, l'opération d'itération des éléments d'un conteneur pour constituer la somme des éléments n'est pas spécifique à `std::vector`.

Si on généralise l'algorithme `sum`, on obtient :

```
template<Container Cont, Arithmetic T>
T sum(const Cont& c, T s) {
    for (auto x : c) s += x;
    return s;
}
```

L'exemple peut paraître évident. Mais, en pratique, il arrive parfois qu'un premier développeur écrive une fonction template 1 et qu'ensuite un second développeur écrive une fonction template 2 parce qu'au moins un des deux template est trop spécifique (oui, comme on pourra le voir avec les autres règles, ce n'est pas forcément le premier développeur qui écrit le code trop spécifique).

- Utilisez les templates pour généraliser des algorithmes avec plusieurs types de données. Les algorithmes sur les conteneurs sont des bons exemples, les algorithmes de somme, de recherche d'une valeur précise, de recherche de la valeur max, etc. sont généralisables quelque-soit le type de données dans le conteneur (ne les redéveloppez pas ! tous les algorithmes sur les conteneurs existent déjà dans la STL). Par contre, ne faites pas de la sur-abstraction :

```
template<Floating_point T>
T DegToRadian(T angle)
{
    return angle * std::numbers::pi / 180.0;
}
```

Ici, les constantes sont des « doubles ». L'opération est faite quoiqu'il arrive avec un type « double ».

- Utilisez les templates en complément des patrons de conception POO. Deux design patterns illustrent cette règle : External Polymorphism et Type Erasure (voir §External Polymorphism et Type Erasure).

## INTERFACES

La programmation générique s'appuie sur des principes de conception (voir §Architecture). Les bonnes pratiques suivantes résument ces principes appliqués aux templates :

- Passer des objets fonctions pour généraliser les algorithmes. La STL fournit de nombreux exemples d'adaptation du comportement de ses algorithmes par ajout d'un paramètre « callable ». Un paramètre « callable » peut être une fonction, un foncteur ou un lambda. Je vous laisse naviguer dans la documentation de l'include « `algorithm` » [19] afin d'étudier les interfaces des fonctions templates. (*Le passage d'un paramètre « callable » correspond à l'implémentation du design pattern Strategy pour les fonctions template*)
- Imposer des paramètres de type régulier ou semi-régulier. Nous avons vu précédemment la définition du type régulier (voir §Type régulier). Un type semi-régulier est un type régulier sans les opérateurs d'égalité (`==` et `!=`). En effet, les algorithmes doivent reposer sur des opérations génériques. Si la classe

ne présente pas ces opérations, l'implémenteur de la classe prive l'usage de sa classe avec les algorithmes génériques.

```
struct MyInt
{
    MyInt() = default;
    explicit MyInt(const MyInt&) = default;
    MyInt(MyInt&&) = default;
    MyInt(const int v)
        : value{v}
    {}
    int value{0};
};

int main()
{
    std::vector<MyInt> a{ 0, 1, 2, 3 };
    std::vector<MyInt> b(4U);
    std::copy(a.cbegin(), a.cend(), b.begin());
    return 0;
}
```

Le programme ci-dessus, ne compile pas car MyInt n'est pas un type régulier.

L'implémentation ou l'utilisation de templates nécessite, aussi, quelques bonnes pratiques afin de faciliter la lisibilité :

Problématiques	Bonnes pratiques
L'écriture d'un type issu d'une classe template est verbeuse.	Utilisez des alias (définis avec le mot clé « using »).
La déclaration d'une classe template est verbeuse.	Implémentez ou utilisez des fonctions template pour déduire automatiquement les types. Par exemple, les fonctions « std::make_tuple » et « std::make_pair » permettent de construire un type « std::tuple » et un type « std::pair » respectivement sans avoir besoin de spécifier les types sous-jacent.

## DEFINITIONS

L'implémentation d'une classe ou d'une fonction template s'accompagne des bonnes pratiques suivantes :

- Utilisez les concepts ou « std::enable\_if » pour définir sémantiquement la généralisation. Dans la partie §Usage, nous avons vu un exemple de généralisation d'un algorithme. La généralisation était décrite par des concepts, en C++17. L'équivalent C++14, avec « std::enable\_if », du même exemple est le suivant :

```
template<typename Cont, typename T,
        std::enable_if_t<is_container<Cont>::value
                        && std::is_arithmetic<T>::value, bool> = true>
T sum(const Cont& c, T s) {
    for (auto x : c) s += x;
```

```
    return s;
}
```

- Utilisez correctement le principe de programmation SFINAE. L'acronyme SFINAE signifie « Substitution Failure Is Not An Error ». D'accord, cela ne nous avance pas beaucoup, la définition est assez obscure. En réalité, cela signifie que le compilateur déduit la classe ou la fonction template à instancier à partir des paramètres template. Si une spécialisation de la classe ou de la fonction template existe avec le type instancié, elle sera choisie prioritairement. Dans le cas contraire, la classe ou la fonction template non spécialisée sera choisie. J'avoue avoir eu un peu de difficulté à trouver un exemple parlant. En effet, on entre dans le domaine de la métaprogrammation. L'exemple suivant décrit la fonction « container\_size() » qui renvoie la taille du conteneur passé en paramètre (\*).

```
template <class T>
concept Sizeable = requires(T container)
{
    container.size();
};

template <class T>
concept IsArray = std::is_array_v<std::remove_reference_t<T>>;

template<typename T> // ❶
constexpr size_t container_size(T&& x) noexcept
{
    return 0U;
}

template<Sizeable T> // ❷
constexpr size_t container_size(T&& x) noexcept
{
    return x.size();
}

template<IsArray T> // ❸
constexpr size_t container_size(T&& x) noexcept
{
    return sizeof(std::forward<T>(x)) /
sizeof(std::remove_pointer_t<std::decay_t<T>>);
}

int main()
{
    std::array<int,10> a;
    std::vector<int> v(10);
    int ca[10];
    int i;
    const char* str = "abcd";
    std::cout << "size: " << container_size(a)
    << ", " << container_size(v)
```

```

    << ", " << container_size(ca)
    << ", " << container_size(i)
    << ", " << container_size(str)
    << ", " << container_size(42)
    << ", " << container_size(nullptr)
    ;
    // size: 10, 10, 10, 0, 0, 0, 0
    return 0;
}

```

La fonction template « `container_size()` » est d'abord définie pour tous les types ❶. Ensuite, on spécialise la fonction template « `container_size()` » pour un type ou un groupe de types (❷, ❸). Une erreur répandue est de ne pas définir la fonction template pour tous les types et de définir, à la place, une fonction template avec les concepts contraires aux spécialisations :

```

// Mauvais
template <class T>
concept NotArrayOrSize = !(isArray<T> || Sizeable<T>);

template<NotArrayOrSize T>
constexpr size_t container_size(T&& x) noexcept
{
    return 0U;
}

```

C'est une erreur car le nombre de combinaisons des concepts pour définir les différents cas peut devenir grand ( $2^n$ ). Aussi, à chaque nouvelle spécialisation, les anciens concepts sont à modifier pour tenir compte de la nouvelle spécialisation.

*(\* La fonction `std::size()` implémente un comportement similaire)*

Vous semblez être mitigé par rapport à mon exemple. Je comprends : la métaprogrammation est une activité complexe qui demande un peu d'expérience. Cependant, le principe SFINAE est très important car le trait « `std::enable_if` » et les concepts reposent sur ce principe. Egalement, on retrouve ce principe pour étendre une fonctionnalité à des types personnalisés. Par exemple, si je souhaite étendre la classe « `std::hash` » avec ma classe « `GeoPosition` », le code est le suivant :

```

template<>
struct std::hash<GeoPosition>
{
    std::size_t operator()(const GeoPosition & g) const noexcept
    {
        const std::size_t h1 = std::hash<Degree>{}(g.latitude());
        const std::size_t h2 = std::hash<Degree>{}(g.longitude());
        const std::size_t h3 = std::hash<Meter>{}(g.altitude());
        return h1 ^ (h2 << 1) ^ (h3 << 2);
    }
};

```

La spécialisation de la classe « `std::hash` » pour le type « `GeoPosition` » permet à un appelant d'obtenir un hash de la même manière qu'avec les types natifs du C++ ou de la STL. Cela facilite aussi l'utilisation des conteneurs non ordonnés : « `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, `std::unordered_multimap` ».



- Organisez vos classes template en séparant les paramètres template et les paramètres non template. Sans plus attendre, voici un exemple présentant le défaut :

```
// Mauvais
template <class T>
class NamedValue
{
public:
    NamedValue(const std::string& name, T&& value)
        : name_{name}, value_{std::forward<T>(value)}
    {}

    constexpr const std::string& name() const noexcept
    {
        return name_;
    }

    constexpr T value() const noexcept
    {
        return value_;
    }

private:
    std::string name_;
    T value_;
};
```

Le code de l'exemple peut paraître innocent. Mais, chaque instantiation d'une classe template duplique le code non relatif à un paramètre template. Dans l'exemple, c'est le code de gestion de la variable membre « name » qui est dupliqué à chaque nouvelle instantiation de la classe template « NamedValue ». Cela peut s'avérer surprenamment coûteux dans des cas plus complexes. Pour résoudre ce problème, le code non relatif au paramètre template est défini dans une classe de base :

```
// Mieux
class NamedBase
{
public:
    explicit NamedBase(const std::string& name)
        : name_{name}
    {}

    constexpr const std::string& name() const noexcept
    {
        return name_;
    }

protected:
    ~NamedBase() = default;

private:
    std::string name_;
};
```

```
};

template <class T>
class NamedValue : public NamedBase
{
public:
    NamedValue(const std::string& name, T&& value)
        : NamedBase{name}, value_{std::forward<T>(value)}
    {}

    constexpr T value() const noexcept
    {
        return value_;
    }

private:
    T value_;
};
```

La classe « NamedValue » ne gère plus que le code relatif au paramètre template.

- Utilisez les fonctions « constexpr » pour définir des expressions évaluées à la compilation. Pour illustrer cette bonne pratique, je vous propose de regarder un peu d'assembleur. Le programme calcule des ratios (possiblement utilisés par des filtres successifs). Les ratios sont définis dans le code et le ratio cumulé est affiché à la fin du programme.

The screenshot displays the Compiler Explorer interface. On the left, the C++ source code is shown, featuring concepts for arithmetic and arithmetic containers, and the use of constexpr functions to calculate ratios. On the right, the assembly output for x86-64 clang 17.0.1 is displayed, showing the compiled code for the program. The assembly output includes instructions for stack frame setup, memory allocation, and the final output of the program.

Comme vous l'avez sans doute remarqué, les fonctions « constexpr » ont été évaluées à la compilation et la variable « globalRatio » contient directement la valeur calculée (partie jaune).



## C++ CORE GUIDELINES

- [T.1: Use templates to raise the level of abstraction of code](#)
- [T.2: Use templates to express algorithms that apply to many argument types](#)
- [T.3: Use templates to express containers and ranges](#)
- [T.4: Use templates to express syntax tree manipulation](#)
- [T.5: Combine generic and OO techniques to amplify their strengths, not their costs](#)
- [T.10: Specify concepts for all template arguments](#)
- [T.11: Whenever possible use standard concepts](#)
- [T.12: Prefer concept names over `auto` for local variables](#)
- [T.13: Prefer the shorthand notation for simple, single-type argument concepts](#)
- [T.20: Avoid “concepts” without meaningful semantics](#)
- [T.21: Require a complete set of operations for a concept](#)
- [T.22: Specify axioms for concepts](#)
- [T.23: Differentiate a refined concept from its more general case by adding new use patterns](#)
- [T.24: Use tag classes or traits to differentiate concepts that differ only in semantics](#)
- [T.25: Avoid complementary constraints](#)
- [T.26: Prefer to define concepts in terms of use-patterns rather than simple syntax](#)
- [T.30: Use concept negation \(`!C<T>`\) sparingly to express a minor difference](#)
- [T.31: Use concept disjunction \(`C1<T> || C2<T>`\) sparingly to express alternatives](#)
- [T.40: Use function objects to pass operations to algorithms](#)
- [T.41: Require only essential properties in a template’s concepts](#)
- [T.42: Use template aliases to simplify notation and hide implementation details](#)
- [T.43: Prefer `using` over `typedef` for defining aliases](#)
- [T.44: Use function templates to deduce class template argument types \(where feasible\)](#)
- [T.46: Require template arguments to be at least semiregular](#)
- [T.47: Avoid highly visible unconstrained templates with common names](#)
- [T.48: If your compiler does not support concepts, fake them with `enable\_if`](#)
- [T.49: Where possible, avoid type-erasure](#)
- [T.60: Minimize a template’s context dependencies](#)
- [T.61: Do not over-parameterize members \(SCARY\)](#)
- [T.62: Place non-dependent class template members in a non-templated base class](#)
- [T.64: Use specialization to provide alternative implementations of class templates](#)
- [T.65: Use tag dispatch to provide alternative implementations of functions](#)
- [T.67: Use specialization to provide alternative implementations for irregular types](#)
- [T.68: Use `{ }` rather than `\( \)` within templates to avoid ambiguities](#)
- [T.69: Inside a template, don’t make an unqualified non-member function call unless you intend it to be a customization point](#)
- [T.80: Do not naively templatize a class hierarchy](#)
- [T.81: Do not mix hierarchies and arrays](#)
- [T.82: Linearize a hierarchy when virtual functions are undesirable](#)
- [T.83: Do not declare a member function template virtual](#)
- [T.84: Use a non-template core implementation to provide an ABI-stable interface](#)
- [T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types](#)
- [T.101: ??? How to pass arguments to a variadic template ???](#)
- [T.102: ??? How to process arguments to a variadic template ???](#)
- [T.103: Don’t use variadic templates for homogeneous argument lists](#)
- [T.120: Use template metaprogramming only when you really need to](#)

[T.121: Use template metaprogramming primarily to emulate concepts](#)  
[T.122: Use templates \(usually template aliases\) to compute types at compile time](#)  
[T.123: Use `constexpr` functions to compute values at compile time](#)  
[T.124: Prefer to use standard-library TMP facilities](#)  
[T.125: If you need to go beyond the standard-library TMP facilities, use an existing library](#)  
[T.140: If an operation can be reused, give it a name](#)  
[T.141: Use an unnamed lambda if you need a simple function object in one place only](#)  
[T.142: Use template variables to simplify notation](#)  
[T.143: Don't write unintentionally non-generic code](#)  
[T.144: Don't specialize function templates](#)  
[T.150: Check that a class matches a concept using `static\_assert`](#)

## TESTS

Dans ce chapitre, je souhaite vous faire part de quelques recommandations ou indications dans l'écriture de tests. Les tests constituent une tâche difficile car elle dépend de plusieurs paramètres. Ces paramètres doivent être décidés en amont du projet car elles impactent le délai de livraison, la conception et le développement. Ces paramètres sont :

- Quels sont les outils et les moyens que l'on dispose pour faire des tests ?
- Quel niveau de granularité souhaite-t-on pour les tests ?
- Les tests doivent-ils être manuels ou automatisés ?
- Quels types de test (fonctionnel/robustesse/performance) sont à prévoir ?

Il n'y a pas de réponse unique à chacune des questions car on peut définir une stratégie de test différente selon la phase du cycle de vie du logiciel. De plus, il est important de faire évoluer les décisions prises pour les tests en adéquation de l'évolution du besoin et des choix de conception (Une décision malavisée « du lundi matin » pour gagner du temps peut s'avérer pénalisante au moment des tests).

Aussi, si du point de vue « contractuelle », aucun test n'est attendu ou seul des tests logiciels de haut niveau sont attendus lors de la livraison, il est fortement recommandé de prévoir, tout de même, des tests selon les différentes phases du cycle de développement. Je vous rassure « Tester, ce n'est pas douter », mais « Tester, c'est coder collectivement et maintenir la qualité du code ». Pour cela, il faut pouvoir faire des tests dès le début du développement et donc, avoir l'ensemble des moyens et des outils disponibles, ainsi que les projets de tests et de bouchonnages amorcés.

Enfin, la granularité des tests impacte directement la conception (et vice-versa). Les unités décidées pour les tests doivent être délimitées par des interfaces logicielles. Chacune des unités doivent être construite en isolation et, en fonction des moyens de test, être extensible (voir §Interfaces et §Architecture).

L'implémentation des tests doivent se faire à partir de la spécification ou des cas d'utilisation. L'étude de la validité des valeurs d'entrées et des valeurs de sorties permettent de déterminer les cas de test :

- par partition d'équivalence [21]
- par l'analyse des valeurs aux limites [22]

Les valeurs de test doivent être choisies afin de limiter le nombre de combinaisons. Aussi, elles doivent être différentes des valeurs par défaut pour les cas de tests valides, dans le but d'éviter de remonter des faux positifs.

Au niveau de la mise en pratique, n'hésitez pas à créer un projet de test par module. Dans le cas contraire, la création de bouchon se révélera compliquée. En complément, créez des projets outils, bouchons, bibliothèques de fonctions que vous puissiez réutiliser facilement pour vos projets de test.

Pour finir, n'oubliez pas de documenter les tests avec :

- la référence de l'exigence fonctionnelle et/ou du cas d'utilisation
- les conditions préalables à l'exécution du test
- la description des étapes de tests détaillant l'action effectuée et le résultat attendu

## ARCHITECTURE

La conception d'une application est une tâche à part entière dans la création d'une application. Cet aspect est souvent mis de côté à cause de l'incompréhension de l'activité à réaliser. La conception n'est pas :

- de la macro-architecture : Architecture de microservices, modules, ...
- des idiomes d'implémentation : RAI, Meyer's Singleton, ...
- des fonctionnalités : lambda, concept, `std::chrono`, `std::thread`, ...

Alors qu'est-ce que la conception logicielle et pourquoi est-ce important ?

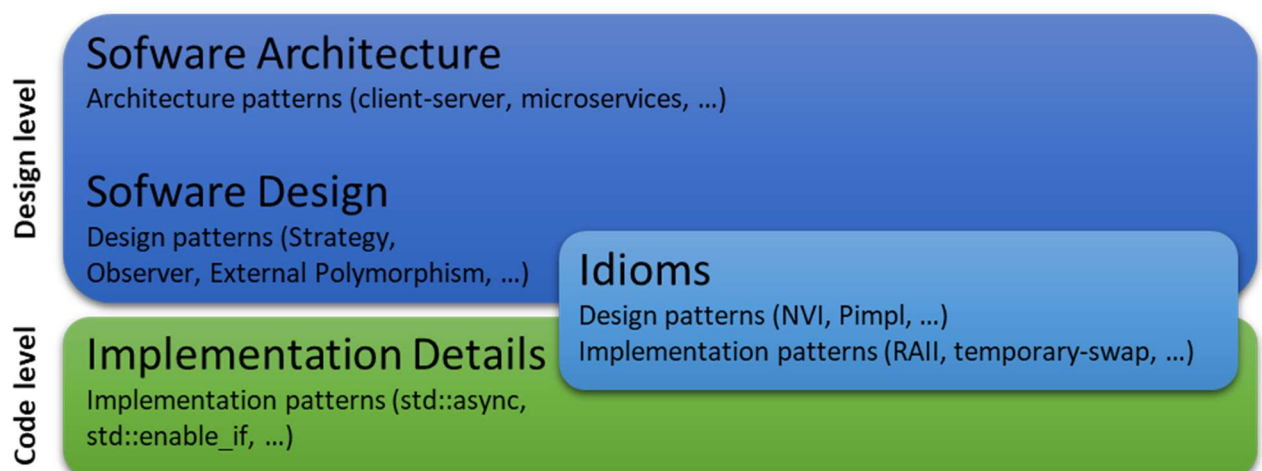
## DEFINITIONS DE LA CONCEPTION

Il n'y a pas de définition « officielle » de la conception de logiciels. Alors je vais tenter une traduction de celle proposée par Klaus Iglberger dans le livre « C++ Software Design » :

*« La conception de logiciels est l'art de gérer les interdépendances entre les composants d'un logiciel. Elle vise à minimiser les dépendances artificielles (techniques) et introduit les abstractions et les compromis nécessaires. ».*

La conception est importante car c'est de sa réalisation que le logiciel tirera sa capacité à être modifié. (Le « **software** », contrairement au « **hardware** », doit pouvoir être modifiable). De plus, bien qu'elle soit essentielle, l'activité de conception est l'une des plus difficiles car il n'y a pas de réponse prédéfinie (« cela dépend »).

Aussi, pour mieux cerner où se positionne le niveau de granularité logiciel attendu pour la conception de logiciels, voici un petit schéma détaillant les trois niveaux du développement logiciel.



Comme vous pouvez le voir, les frontières sont malléables entre les patrons d'architecture, les patrons de conception et les idiomes de conception.

Avant d'aller plus loin, mettons-nous d'accord sur la terminologie. Comme on l'a vu plus haut, un des critères pour la qualité d'un logiciel est sa capacité aux changements. Pour cela, on différencie plusieurs niveaux :

- Haut niveau : Fonctionnalité considérée comme stable, ayant peu de dépendances.
- Bas niveau : Fonctionnalité considérée comme flexible, ayant beaucoup de dépendances.

La terminologie provient de l'UML (*Unified Modeling Language*) où les éléments les plus stables apparaissent au-dessus et les éléments modifiables apparaissent en bas du diagramme. Le but des niveaux d'abstractions est de modéliser les interdépendances. Aussi, on veillera :

- à n'avoir les dépendances que dans un seul sens (le bas niveau dépend du haut niveau).
- à minimiser les dépendances au sein d'un même niveau.

Pour atteindre ces objectifs, il existe des principes de conception, regroupés sous l'acronyme « SOLID ». Les cinq principes sont les suivants :

- Single responsibility principle (Responsabilité unique)
- Open/close (Ouvert/fermé)
- Liskov substitution (Substitution de Liskov)
- Interface segregation (Ségrégation des interfaces)
- Dependency Inversion (Inversion des dépendances)

Tout cela peut paraître nébuleux ou complexe et, à la place de longues phrases d'explication, je vous propose de mettre en œuvre les principes « SOLID » au travers d'un cas pratique.

#### MODIFICATION/EXTENSION – CAS PRATIQUE

Imaginons une interface « File » qui possède deux classes dérivées :

- « IniFile » réalisant la lecture d'un fichier au format « ini »
- « XmlFile » réalisant la lecture d'un fichier au format « xml »

On souhaite ajouter une méthode de vérification du fichier avant sa lecture. Les fichiers au format « ini » sont vérifiés à l'aide d'un calcul de « CRC » et les fichiers « xml » sont vérifiés par l'un des deux moyens : un calcul de « CRC » ou une validation par schéma XML.

Sans plus attendre, passons à la pratique et comme dirait Linus Torvalds, « Talk is cheap. Show me the code »:

```
class File
{
    // ...
    virtual void read() = 0;
};
class IniFile : public File
{
    // ...
    void read() override
    {
        // lecture du fichier ini
    }
};
class XmlFile : public File
{
    // ...
    void read() override
    {
        // lecture du fichier xml
    }
};
```

Le diagramme UML correspondant est le suivant :

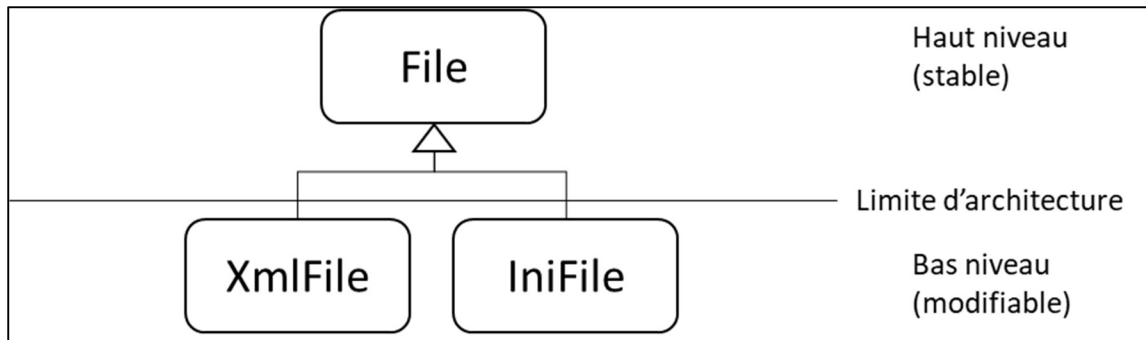


Diagramme 1. Situation initiale

La première proposition correspond simplement à ajouter une méthode « validate » dans l'interface « File ». Les deux classes dérivées implémentent la nouvelle méthode.

```

class File
{
// ...
    virtual void read() = 0;
    virtual bool validate() const = 0;
};
class IniFile : public File
{
// ...
    void read() override
    {
        // lecture du fichier ini
    }
    bool validate() const override
    {
        // vérification du checksum du contenu binaire
    }
};
class XmlWithoutXsdFile : public File
{
public:
// ...
    void read() override
    {
        // lecture du fichier xml
    }
    bool validate() const override
    {
        // vérification du checksum du contenu binaire
    }
};
class XmlWithXsdFile : public File
{
// ...
    void read() override
    {

```



```

        // lecture du fichier xml
    }
    bool validate() const override
    {
        // validation par schéma Xml
    }
};

```

Le diagramme UML correspondant est le suivant :

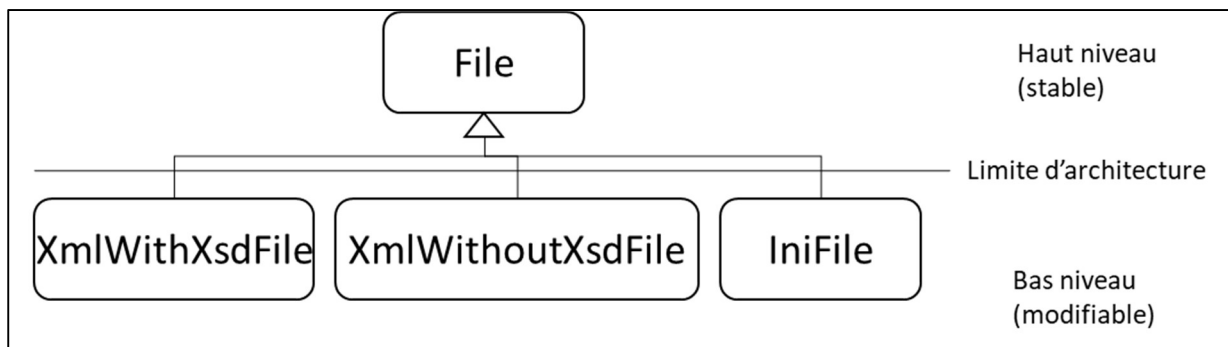


Diagramme 2. Première proposition

Ça ne s'est pas vraiment passé comme prévu. La classe « XmlFile » a dû être dupliquée en deux classes « XmlWithXsdFile » et « XmlWithoutXsdFile » pour implémenter la nouvelle méthode selon les deux façons de valider les fichiers XML. En plus, le code de vérification du checksum se retrouve dupliqué dans les classes « IniFile » et « XmlWithoutXsdFile ». Cette solution n'est pas du tout satisfaisante : La transformation de la classe « XmlFile » en deux nouvelles classes a un impact sur tout le code client et la duplication de code n'aidera pas à maintenir facilement les classes. On pourrait limiter la duplication en proposant une implémentation par défaut dans les classes ancêtres.

```

class File
{
    // ...
    virtual void read() = 0;
    virtual bool validate() const
    {
        // vérification du checksum du contenu binaire
    }
};

class IniFile : public File
{
    // ...
    void read() override
    {
        // lecture du fichier ini
    }
};

class XmlWithoutXsdFile : public File
{
public:

```

```

// ...
void read() override
{
    // lecture du fichier xml
}
};
class XmlWithXsdFile : public XmlWithoutXsdFile
{
// ...
bool validate() const override
{
    // validation par schéma Xml
}
};

```

Le diagramme UML correspondant est le suivant :

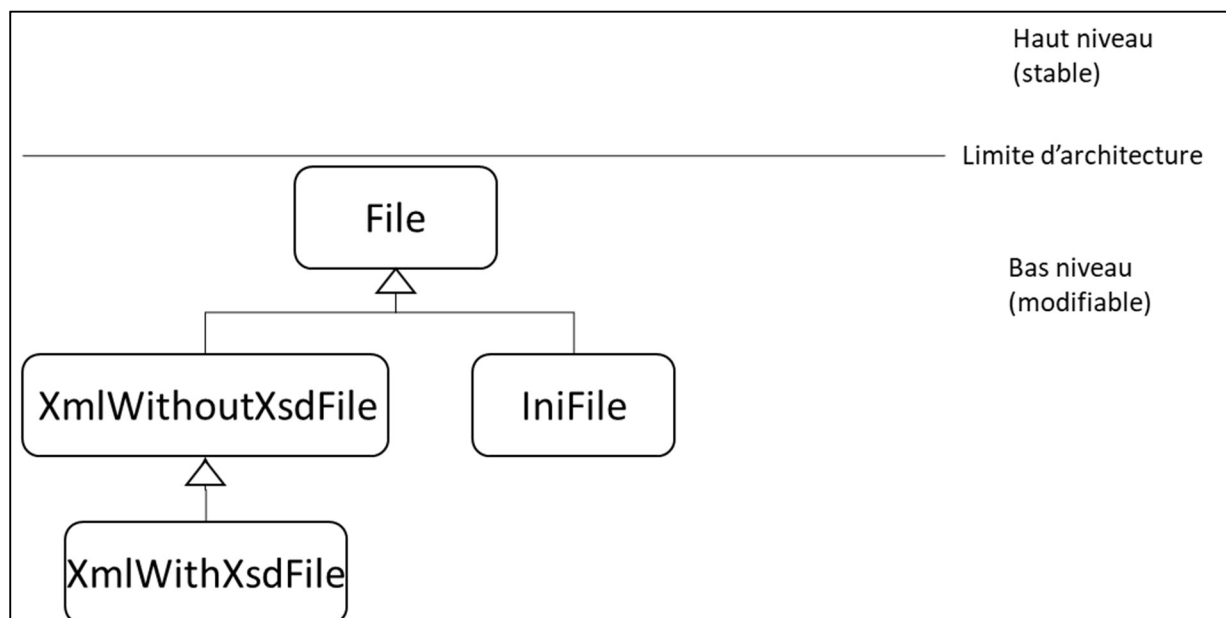


Diagramme 3. Seconde proposition

On s'enfonce dans les problèmes. « File » n'est plus une interface. Cela a possiblement les conséquences suivantes :

- Chaque modification nécessite la recompilation du code client.
- Le code client a besoin d'accéder aux fichiers « headers » nécessaires à l'implémentation du calcul de CRC.
- Les classes ancêtre possèdent des membres (attributs et fonctions) non utilisés par les classes dérivées.

Si l'algorithme de calcul de « CRC » est déprécié et que l'on doit maintenir une compatibilité pour les anciens fichiers, la maintenance va devenir très compliquée (constitution de code mort au fur et à mesure, ajout de variables de contexte, ...). Idem, si on doit rajouter une nouvelle opération sur les fichiers, la hiérarchie de classes comportera des classes avec des noms à rallonge (Fichier Avec Fonctionnalité 1 et Fonctionnalité 2, Fichier Avec Fonctionnalité 1 Sans Fonctionnalité 2, ...) avec des possibles duplications de code. On retombera inévitablement aux problèmes de la proposition 1, en plus des nouveaux problèmes.

Bon, on reprend depuis le début et appliquons le principe de ségrégation des interfaces (le « I » de « SOLID »).

```
class File
{
// ...
    virtual void read() = 0;
};
class Validable
{
// ...
    virtual bool validate() const = 0;
};
class ChecksumValidation : public Validable
{
// ...
    bool validate() const override
    {
        // vérification du checksum du contenu binaire
    }
};
class XsdValidation : public Validable
{
// ...
    bool validate() const override
    {
        // validation par schéma Xml
    }
};
class IniFile : public File, public ChecksumValidation
{
// ...
    void read() override
    {
        // lecture du fichier ini
    }
};
class XmlWithoutXsdFile : public File, public ChecksumValidation
{
public:
// ...
    void read() override
    {
        // lecture du fichier xml
    }
};
class XmlWithXsdFile : public File, public XsdValidation
{
// ...
    void read() override
    {
        // lecture du fichier xml
    }
};
```

```

    }
};

```

Le diagramme UML correspondant est le suivant :

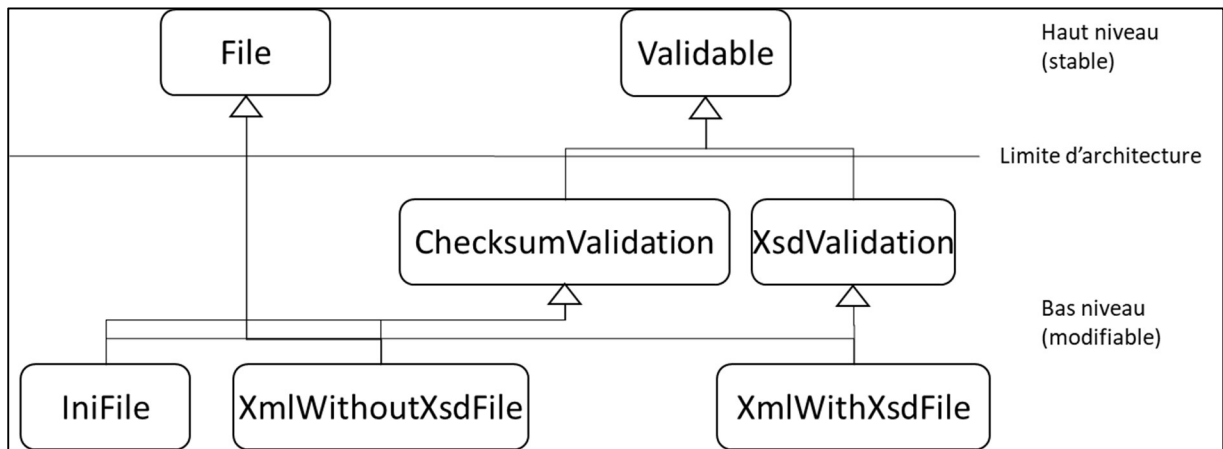


Diagramme 4. Troisième proposition

C'est mieux que les propositions 1 et 2. L'utilisation d'interfaces logicielles permet d'isoler les dépendances. Dans l'exemple, on conserve l'interface « File » d'origine. Le code de vérification du checksum n'est plus dupliqué. Il reste le problème de la duplication de la classe « XmlFile » en deux classes. Appliquons le principe de responsabilité unique (le « S » de « SOLID »).

```

class File
{
    // ...
    virtual void read() = 0;
};
class Validable
{
    // ...
    virtual bool validate() const = 0;
};
template <class FileT>
class Validator
{
    // ...
    virtual bool validate(const FileT&) const = 0;
};
class ChecksumValidation : public Validator<File>, public Validator<XmlFile>
{
    // ...
    bool validate(const File&) const override
    {
        // vérification du checksum du contenu binaire
    }
    bool validate(const XmlFile& f) const override {
        validate(static_cast<const File&>(f));
    }
}

```

```

}
class XsdValidation : public Validator<XmlFile>
{
// ...
    bool validate(const XmlFile& f) const override
    {
        // validation par schéma Xml
    }
};
class IniFile : public Validable, public File
{
// ...
    explicit IniFile(std::unique_ptr<Validator<File>> validator)
        : validator_{std::move(validator)}
    {}
    bool validate() const override
    {
        validator_->validate(*this);
    }
private:
    std::unique_ptr<Validator<File>> validator_;
};
class XmlFile : public Validable, public File
{
// ...
    explicit XmlFile(std::unique_ptr<Validator<XmlFile>> validator)
        : validator_{std::move(validator)}
    {}
    bool validate() const override
    {
        validator_->validate(*this);
    }
private:
    std::unique_ptr<Validator<XmlFile>> validator_;
};

```

Le diagramme UML correspondant est le suivant :

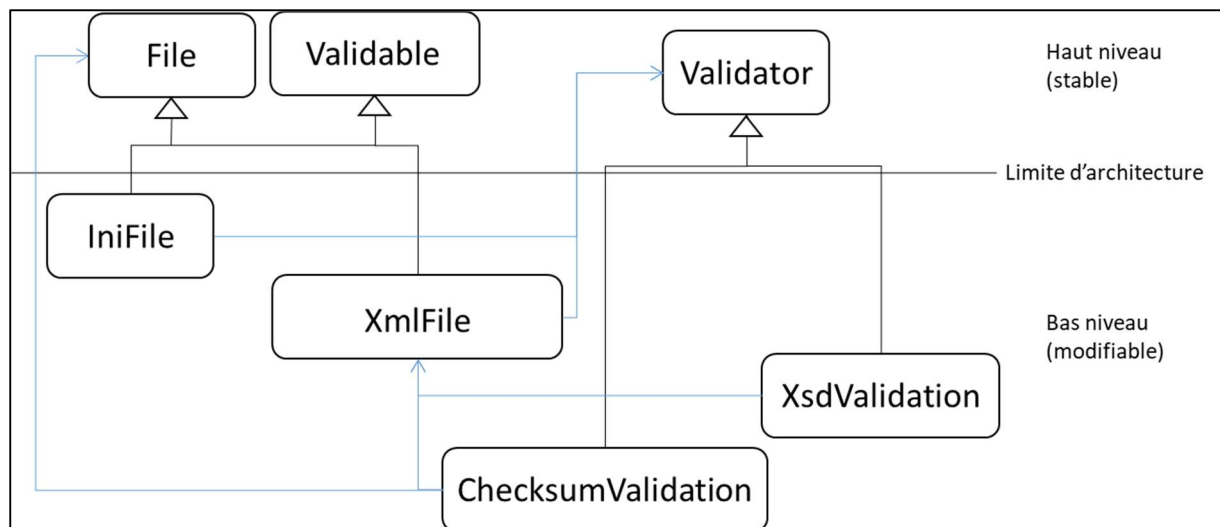


Diagramme 5. Quatrième proposition

Voilà. Ainsi, l'héritage est rarement une réponse au principe de responsabilité unique. Le principe se base sur l'injection de dépendance (patron de conception Strategy) ou sur les arguments template (Policy-based design). L'héritage maintient un couplage compact alors que la composition (par injection de dépendance ou par arguments template) assure un couplage lâche. Au niveau de la conception, la solution semble convenir. Par contre, au niveau de l'implémentation, l'utilisation d'une interface intermédiaire « Validator » n'est pas nécessaire en C++ moderne. Aussi, l'utilisation des pointeurs complexifie l'implémentation. Si vous pouvez limiter le nombre de lignes de code, faites-le.

```

class File
{
// ...
    virtual void read() = 0;
};
class Validable
{
// ...
    virtual bool validate() const = 0;
};
class IniFile : public Validable, public File
{
// ...
    using Validator = std::function<bool(const File&)>;
    explicit IniFile(Validator validator)
        : validator_{std::move(validator)}
    {}
    bool validate() const override
    {
        validator_(*this);
    }
private:
    Validator validator_;
};
class XmlFile : public Validable, public File
  
```

```

{
// ...
    using Validator = std::function<bool(const XmlFile&)>;
    explicit XmlFile(Validator validator)
        : validator_{std::move(validator)}
    {}
    bool validate() const override
    {
        validator_(*this);
    }
private:
    Validator validator_;
};
bool validFileChecksum(const File& f)
{
    // vérification du checksum du contenu binaire
}
bool validFileChecksum(const XmlFile& f)
{
    return validFileChecksum(static_cast<const File&>(f));
}
bool validXmlSchema(const XmlFile& f)
{
    // validation par schéma Xml
}

```

Le diagramme UML correspondant est le suivant :

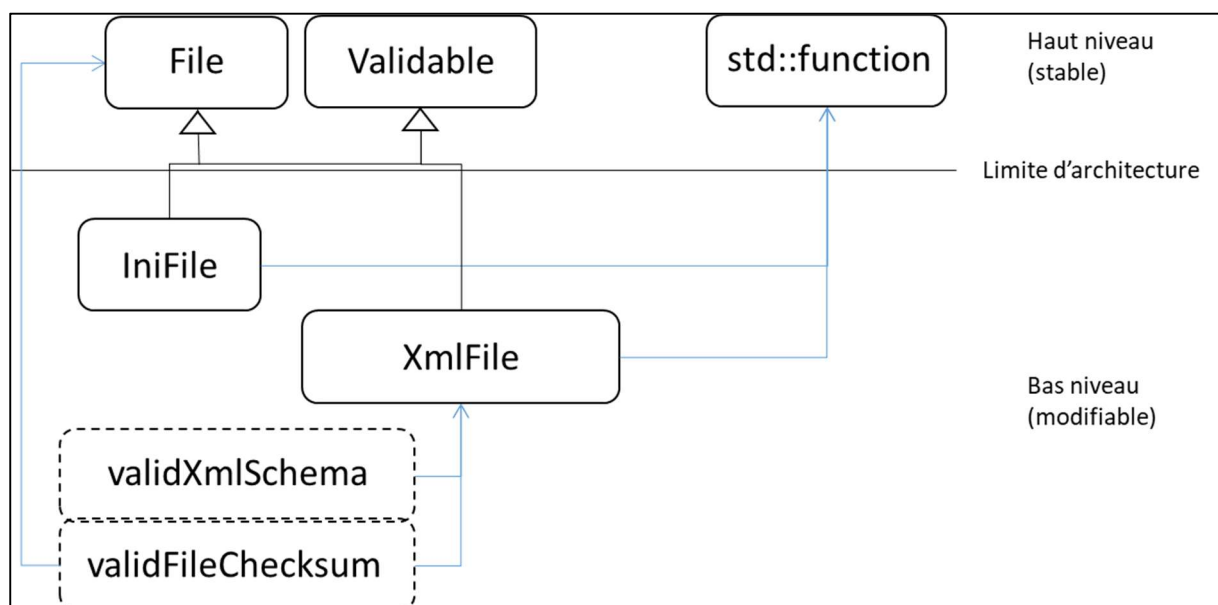


Diagramme 6. Cinquième proposition

Conception : OK, Implémentation C++ : OK. En regardant, de plus près, vous trouvez aussi que les surcharges compliquent un peu la syntaxe. Je suis d'accord, on peut factoriser le code des surcharges pour simplifier l'ajout de formats de fichier. Avant de voir le code, parlons un peu du principe Ouvert/fermé (le « O » de « SOLID »), ce principe signifie : Ouvert à l'extension et fermé à la modification. Dans la pratique, sa mise en œuvre est similaire

au principe de responsabilité unique. Dans notre exemple, la mise en place du patron de conception « Strategy » remplit cette définition. D'un autre côté, l'organisation architecturale du code doit être aussi pensée pour remplir complètement cet objectif. En effet, si les fonctions de vérification à l'aide d'un calcul de « CRC » se trouvent dans un autre module que les classes réalisant la lecture d'un format de fichier, l'ajout du format « JSON » ayant deux méthodes de validation (« JSON Schema » et « CRC » avec les surcharges similaires au XML) nous force soit à modifier les deux modules ou soit à réaliser un troisième module spécifique au format JSON ayant comme dépendances les deux premiers modules. Finalement, le principe Ouvert/fermé demande de prendre en compte des considérations architecturales sur qu'est-ce qu'on étend et comment on l'étend. La factorisation du code va nous permettre de nous soustraire de cette problématique.

```
class File
{
// ...
    virtual void read() = 0;
};
class Validable
{
// ...
    virtual bool validate() const = 0;
};
template<class DerivedT>
class ValidableFile : public Validable, public File
{
// ...
    bool validate() const override
    {
        return fct_();
    }
protected:
    template<class CallableT>
    explicit ValidableFile(CallableT&& fct)
        : fct_{std::bind(
            std::forward<CallableT>(fct),
            std::ref(static_cast<DerivedT>(*this)))}
    {
    }
private:
    std::function<bool()> fct_;
};
class IniFile : public ValidableFile<IniFile>
{
// ...
public:
    template<class CallableT>
    explicit IniFile(CallableT&& fct)
        : ValidableFile{std::forward<CallableT>(fct)}
    {
    }
};
class XmlFile : public ValidableFile<XmlFile>
{
```



```

// ...
public:
    template<class CallableT>
    explicit XmlFile(CallableT&& fct)
        : ValidableFile{std::forward<CallableT>(fct)}
    {
    }
};
bool validFileChecksum(const File& f)
{
    // vérification du checksum du contenu binaire
}
bool validXmlSchema(const XmlFile& f)
{
    // validation par schéma Xml
}

```

Le diagramme UML correspondant est le suivant :

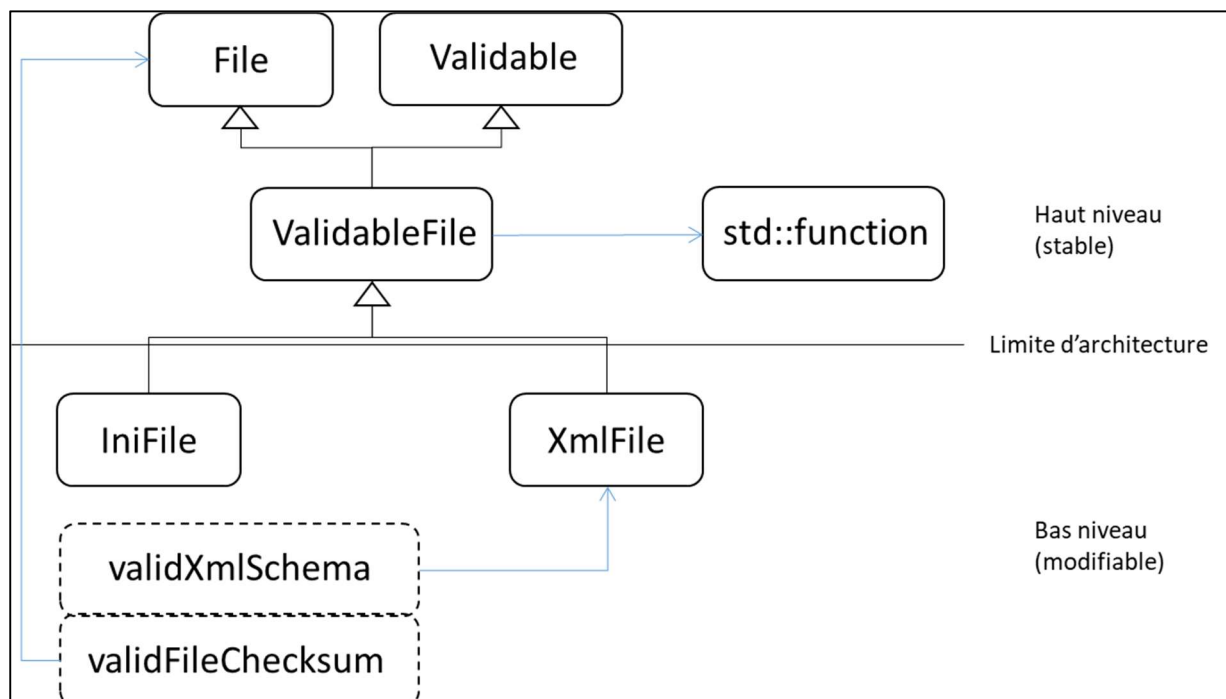


Diagramme 7. Sixième proposition

On répond à tous nos critères. C'est fini ? Peut-être que vous vous dites comme moi, que la hiérarchie de classes « ValidableFile » → « Validable » ressemble un peu au patron de conception « External Polymorphism » [17]. Ne pourrait-on pas implémenter complètement ce patron de conception ? (Et donc supprimer le lien d'héritage entre les classes bas niveau « IniFile », « XmlFile » et la classe « ValidableFile »). Non, principalement parce que ce lien (« IniFile », « XmlFile » → « File ») est dans l'architecture d'origine et que l'on ne souhaite pas modifier le code client. Deuxièmement, appliquons le principe « YAGNI » : « *you ain't gonna need it* ». Nous n'avons aucune raison à faire ce travail. La contrainte liée à ce lien compact est intégrée au logiciel et ne pose pas de problème sur l'existant, il n'y a donc aucun bénéfice à vouloir le supprimer.

## ABSTRACTION

Je sais : je n'ai pas parlé de la substitution de Liskov (le « L » de « SOLID ») et de l'inversion des dépendances (le « D » de « SOLID »). Avant d'aborder ces sujets, faisons un petit détour sur les différents types de polymorphismes en C++ :

- Polymorphisme dynamique : elle se base sur les notions de méthodes virtuelles et d'héritages. Le cas pratique décrit au-dessus est un exemple de polymorphisme dynamique.
- Polymorphisme statique : on distingue deux sous-types : le polymorphisme via template et le polymorphisme via surcharge de fonction.

Les principes « SOLID » s'applique à ces trois types de polymorphismes. Pour varier les exemples, illustrons les deux derniers principes avec les deux types de polymorphismes statiques. La substitution de Liskov (LSP) s'appuie sur la conception par contrat [14]. Elle indique que :

- Les préconditions ne peuvent pas être renforcées dans une sous-classe. Cela signifie que vous ne pouvez pas avoir une sous-classe avec des préconditions plus fortes que celles de sa superclasse ;
- Les postconditions ne peuvent pas être affaiblies dans une sous-classe. Cela signifie que vous ne pouvez pas avoir une sous-classe avec des postconditions plus faibles que celles de sa superclasse.

En polymorphisme statique, le rôle de définitions de l'interface de la superclasse est accompli par la notion de concept.

Imaginons l'exemple suivant pour expliquer le principe « LSP » : Nous avons un équipement qui propose une librairie avec deux interfaces de communications : Communication série ou Ethernet TCP/IP. Le choix du type de communication est défini au moment de la compilation du module. L'implémentation de la classe de réception des trames prend en argument template, la classe établissant la connexion :

```
template<typename T>
concept Receiver = requires(T r, std::string dest) {
    r.connect(dest);
    { r.receive() } -> std::convertible_to<std::string>;
};

struct Socket {
    void connect(const std::string& destIp) {
        // ...
    }
    std::string receive() {
        return "...";
    }
};

struct SerialCom {
    void connect(const std::string& dest) {
        // ouverture du port ?
    }
    std::string receive() {
        return "...";
    }
};

template<Receiver T>
class FrameReader {
```

```

public:
    explicit FrameReader(T receiver, const std::string& dest)
        : receiver_{std::move(receiver)} {
        receiver_.connect(dest);
    }
private:
    T receiver_;
};

```

La classe « SerialCom » viole le principe « LSP » car la communication série n'est pas multi-destinataire. La fonction « connect » n'existe pas pour ce type de communication. On pourrait envisager d'implémenter l'ouverture du port de communication à la place mais cela pourrait introduire des effets non désirés dans l'utilisation de la classe. L'implémentation de l'ouverture du port entraîne les contraintes suivantes, supplémentaires par rapport à une connexion de type socket :

- L'ouverture du port nécessite que le port soit fermé. La communication ne peut pas être établie plusieurs fois sur le même port de communication (précondition renforcée).
- L'ouverture du port ne garantit pas que la communication est effective. L'ouverture du port signifie seulement que la liaison est ouverte localement (postcondition affaiblie).

Finissons ce chapitre avec le principe le plus simple à comprendre : l'inversion des dépendances. Pour cela, reprenons mon exemple de la classe « GeoPosition ». Je souhaite lui ajouter une fonction « similar() », qui est aussi applicable à d'autres classes, et qui consiste à retourner « true » si les objets sont à peu près équivalents (ils ne sont pas égaux mais assez proches pour les considérer comme identiques dans un certain contexte). Evidemment, si une classe n'implémente pas spécifiquement une surcharge de la fonction « similar() », la fonction renverra « true » seulement si les objets sont égaux. Voici ma première version du code :

```

namespace nav
{
    class GeoPosition // ...
}

namespace util
{
    // Fonction similar() par défaut : appel à l'opérateur d'égalité
    template<class T>
    constexpr bool similar(const T& a, const T& b)
    {
        return a == b;
    }

    // Fonction similar() pour la classe GeoPosition
    constexpr Degree degree_precision = 0.000001;
    constexpr Meter meter_precision = 0.001;
    constexpr bool similar(const nav::GeoPosition& a, const nav::GeoPosition& b)
noexcept
    {
        return (std::fabs(a.latitude() - b.latitude()) < degree_precision)
            && (std::fabs(a.longitude() - b.longitude()) < degree_precision)
            && (std::fabs(a.altitude() - b.altitude()) < meter_precision);
    }
}

```

```

}

namespace util::range
{
    template <class T>
    concept Iterable = requires(const T& a)
    {
        std::cbegin(a);
        std::cend(a);
        std::size(a);
    };

    // Fonction similar() pour tous les types de conteneur
    template<Iterable T>
    constexpr bool similar(const T& a, const T& b)
    {
        auto first1 = std::cbegin(a);
        auto first2 = std::cbegin(b);
        auto last1 = std::cend(a);
        auto last2 = std::cend(b);
        bool isSimilar = (std::size(a) == std::size(b));
        while(isSimilar && (first1 != last1) && (first2 != last2))
        {
            isSimilar = util::similar(*first1, *first2);
            ++first1;
            ++first2;
        }
        return isSimilar;
    }
}

namespace nav
{
    struct Point
    {
        int x;
        int y;
        friend constexpr bool operator==(const Point& p1, const Point& p2) noexcept
        = default;
    };

    void testRange()
    {
        Point p1[2] = {Point{1, 1}, Point{2, 2}};
        Point p2[2] = {Point{1, 1}, Point{2, 2}};

        std::vector<GeoPosition> g1 = {
            GeoPosition{1.0, 1.0, 1.0},
            GeoPosition{2.0, 2.0, 2.0}};
        std::vector<GeoPosition> g2 = {
            GeoPosition{1.0000005, 1.0000005, 1.0005},

```

```

        GeoPosition{2.0000002, 2.0000002, 2.0002}};
        std::cout<< std::boolalpha << util::range::similar(p1, p2) << ", " <<
util::range::similar(g1, g2); // true, true
    }
}

```

Le diagramme UML correspondant est le suivant :

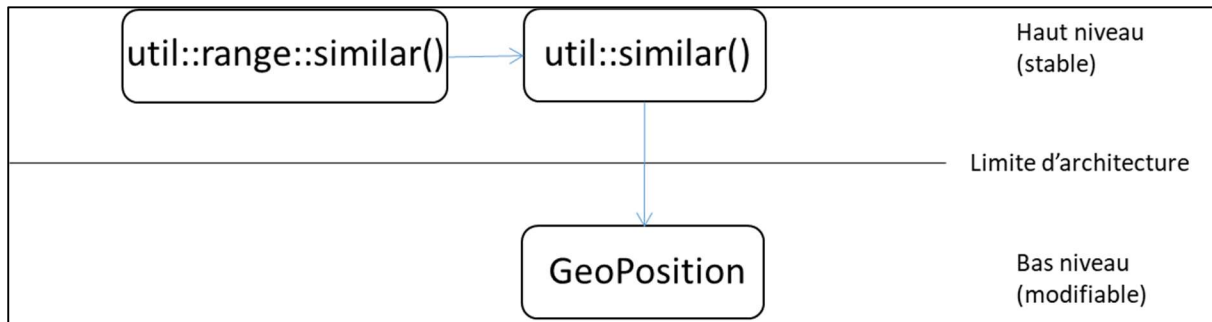


Diagramme 8. Mauvais sens de la dépendance

Dans la première version du code, la seule façon pour que la fonction « `util::range::similar()` » prenant en paramètres des types conteneurs, compile, nécessite de déclarer la fonction « `util::similar()` » avec comme paramètres le type « `GeoPosition` » dans le namespace « `util` », avant le namespace « `util::range` ». On comprend assez bien que cela viole le principe Ouvert/fermé, car à chaque modification de `GeoPosition` ou à l'ajout d'une nouvelle fonction qui surcharge « `similar()` », il faudra modifier le namespace « `util` ».

Pour nous sortir de ce mauvais pas, recourons aux règles « Argument-Dependent Lookup » (ADL) (recherche dépendante de l'argument) [23] :

```

namespace util
{
    // Fonction similar() par défaut : appel à l'opérateur d'égalité
    template<class T>
    constexpr bool similar(const T& a, const T& b)
    {
        return a == b;
    }
}

namespace util::range
{
    template <class T>
    concept Iterable = requires(const T& a)
    {
        std::cbegin(a);
        std::cend(a);
        std::size(a);
    };

    // Fonction similar() pour tous les types de conteneur
    template<Iterable T>
    constexpr bool similar(const T& a, const T& b)

```

```

{
    using util::similar; // ❶
    auto first1 = std::cbegin(a);
    auto first2 = std::cbegin(b);
    auto last1 = std::cend(a);
    auto last2 = std::cend(b);
    bool isSimilar = (std::size(a) == std::size(b));
    while(isSimilar && (first1 != last1) && (first2 != last2))
    {
        isSimilar = similar(*first1, *first2); // ❷
        ++first1;
        ++first2;
    }
    return isSimilar;
}
}

namespace nav
{
    struct Point
    {
        int x;
        int y;
        friend constexpr bool operator==(const Point& p1, const Point& p2) noexcept
= default;
    };

    class GeoPosition // ...

    // Fonction similar() pour la classe GeoPosition
    constexpr Degree degree_precision = 0.000001;
    constexpr Meter meter_precision = 0.001;
    constexpr bool similar(const GeoPosition& a, const GeoPosition& b) noexcept
    {
        return (std::fabs(a.latitude() - b.latitude()) < degree_precision)
            && (std::fabs(a.longitude() - b.longitude()) < degree_precision)
            && (std::fabs(a.altitude() - b.altitude()) < meter_precision);
    }

    void testRange()
    {
        Point p1[2] = {Point{1, 1}, Point{2, 2}};
        Point p2[2] = {Point{1, 1}, Point{2, 2}};

        std::vector<GeoPosition> g1 = {
            GeoPosition{1.0, 1.0, 1.0},
            GeoPosition{2.0, 2.0, 2.0}};
        std::vector<GeoPosition> g2 = {
            GeoPosition{1.0000005, 1.0000005, 1.0005},
            GeoPosition{2.0000002, 2.0000002, 2.0002}};
    }
}

```

```

std::cout<< std::boolalpha << util::range::similar(p1, p2) << ", " <<
util::range::similar(g1, g2); // true, true
    }
}

```

Le diagramme UML correspondant est le suivant :

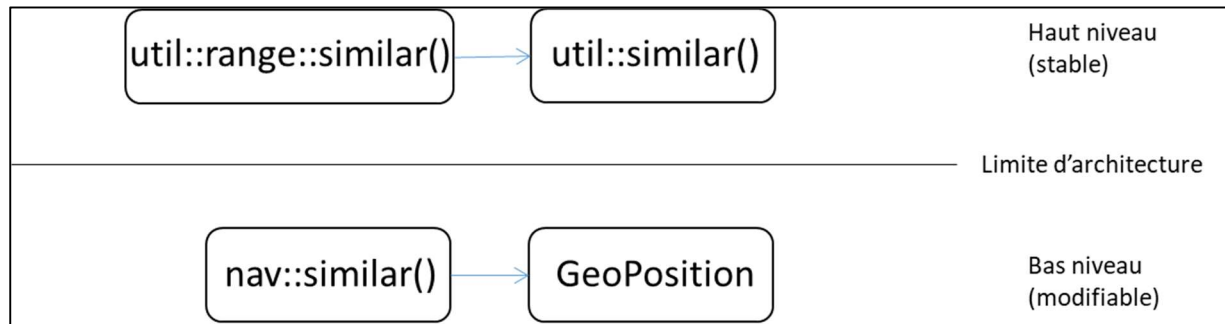


Diagramme 9. Bon sens de la dépendance

Comme vous l’avez tout de suite vu, peu de modifications ont été nécessaires pour remettre les dépendances dans le bon sens. Nous avons ajouté une directive d’utilisation locale de namespace ❶ et supprimé la spécification du namespace dans l’appel de la fonction « similar() » ❷. Ainsi, lors de l’instantiation du template « util::range::similar() », le compilateur recherchera la fonction « similar() » dans tous les namespaces courants (namespaces « nav », « util::range » et « util ») et choisira la fonction « similar() » dont les arguments correspondent aux paramètres.

Un système temps réel est un système informatique dans lequel les contraintes temporelles ont été prises en compte. La programmation temps réel est un vaste sujet. Elle nécessite une bonne connaissance du fonctionnement du système d'exploitation (ordonnancement des tâches, mémoire virtuelle, IPC, ...). Pratiquement, la programmation temps réel nécessite un système d'exploitation temps réel et un framework de développement. Je ne rentrerai donc pas dans le détail. Je vous conseille fortement de lire le livre « Solutions temps réel sous Linux » de Christophe Blaess pour avoir un aperçu de la programmation temps réel sous Linux :

<https://www.eyrolles.com/Informatique/Livre/solutions-temps-reel-sous-linux-9782212677119/>

Cependant, on peut énoncer quelques grands principes :

- Ne faites pas d'allocation dynamique. En effet, l'allocation dynamique fait sortir la tâche de son exécution prioritaire et elle ne sera réintégrée qu'après un nouvel ordonnancement des tâches. Au début du programme, allouez la mémoire de manière statique avec la taille maximum. Utilisez « `std::array` » comme conteneur d'objets. Si besoin, utilisez les conteneurs « `std::pmr` » pour faciliter l'usage du pool mémoire :

```
std::array<std::byte, total_nodes * 32> buffer; // Taille max
std::pmr::monotonic_buffer_resource mbr{buffer.data(), buffer.size()};
std::pmr::polymorphic_allocator<int> pa{&mbr};
std::pmr::queue<int> queue{pa};
for (int i{}; i != total_nodes; ++i)
    queue.push(i);
```

- Utilisez le polymorphisme statique pour concevoir l'application. L'idée du polymorphisme statique est de faciliter la maintenance, l'extensibilité, la testabilité et la réutilisabilité et cela sans perte de performance. Le désavantage est la perte de flexibilité à l'exécution. Aussi, certains patrons de conception comportementaux et structurels existent en versions template. Par exemple, le patron de conception « Strategy » en version template est le suivant :

```
Using ByteArray = std::array<std::byte, 50>;

template< typename DecoderStrategy >
class FiringStatus : public Frame
{
public:
    FiringStatus(const ByteArray& frame, DecoderStrategy&& decoder)
        : Frame{frame}
        , decoder_{std::forward<DecoderStrategy>(decoder)}
    {}

    void decode() const
    {
        decoder_(*this, /*autres arguments*/);
    }
private:
    DecoderStrategy decoder_;
};
```

Cependant, le polymorphisme dynamique n'est pas exclu s'il n'y a pas d'allocation dynamique. Mais, il faut être extrêmement vigilant au surcoût engendré par les indirections (*Le patron de conception*

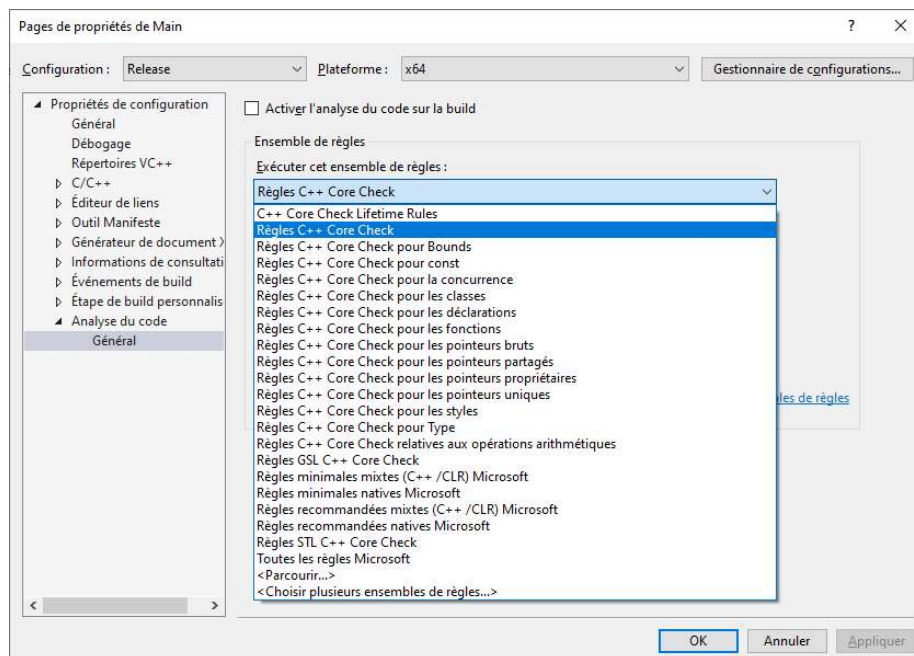


*Acyclic Visitor est l'exemple qui me vient en tête avec sa double indirection et surtout, son crosscast particulièrement coûteux [20]).*

- Utilisez les fonctions du framework temps réel pour toutes les interactions avec le système d'exploitation (processus, objets de synchronisation, ...)
- Séparez les interactions non temps réelles dans des modules non temps réel (opérations sur le système de fichiers, IHM, réseaux, ...)
- Évaluez les performances temps-réel de votre application à l'aide d'outils spécifiques.

La vérification des C++ Core Guidelines est implémentée par les outils suivants :

- Visual Studio : Code analysis (<https://learn.microsoft.com/en-us/cpp/code-quality/using-the-cpp-core-guidelines-checkers?view=msvc-170>)



- Clang-tidy (<https://clang.llvm.org/extra/clang-tidy>)

```
teddy@VULCAIN: ~
Fichier Édition Affichage Recherche Terminal Aide
teddy@VULCAIN:~$ clang-tidy --list-checks
Enabled checks:
clang-analyzer-apiModeling.StdLibraryFunctions
clang-analyzer-apiModeling.TrustNonnull
clang-analyzer-apiModeling.google.GTest
clang-analyzer-apiModeling.llvm.CastValue
clang-analyzer-apiModeling.llvm.ReturnValue
clang-analyzer-core.CallAndMessage
clang-analyzer-core.CallAndMessageModeling
clang-analyzer-core.DivideZero
clang-analyzer-core.DynamicTypePropagation
clang-analyzer-core.NonNullParamChecker
clang-analyzer-core.NonnullStringConstants
clang-analyzer-core.NullDereference
clang-analyzer-core.StackAddrEscapeBase
clang-analyzer-core.StackAddressEscape
clang-analyzer-core.UndefinedBinaryOperatorResult
clang-analyzer-core.VLASize
clang-analyzer-core.builtin.BuiltinFunctions
clang-analyzer-core.builtin.NoReturnFunctions
clang-analyzer-core.uninitialized.ArraySubscript
clang-analyzer-core.uninitialized.Assign
clang-analyzer-core.uninitialized.Branch
clang-analyzer-core.uninitialized.CapturedBlockVariable
clang-analyzer-core.uninitialized.UndefReturn
clang-analyzer-cplusplus.InnerPointer
clang-analyzer-cplusplus.Move
clang-analyzer-cplusplus.NewDelete
clang-analyzer-cplusplus.NewDeleteLeaks
clang-analyzer-cplusplus.PlacementNew
clang-analyzer-cplusplus.PureVirtualCall
clang-analyzer-cplusplus.SelfAssignment
clang-analyzer-cplusplus.SmartPtrModeling
clang-analyzer-cplusplus.StringChecker
clang-analyzer-cplusplus.VirtualCallModeling
clang-analyzer-deadcode.DeadStores
clang-analyzer-fuchsia.HandChecker
clang-analyzer-nullability.NullPassedToNonnull
clang-analyzer-nullability.NullReturnedFromNonnull
clang-analyzer-nullability.NullabilityBase
clang-analyzer-nullability.NullableDereferenced
clang-analyzer-nullability.NullablePassedToNonnull
clang-analyzer-nullability.NullableReturnedFromNonnull
clang-analyzer-optin.cplusplus.UninitializedObject
clang-analyzer-optin.cplusplus.VirtualCall
clang-analyzer-optin.mpi.MPI-Checker
clang-analyzer-optin.osx.OSObjectCStyleCast
clang-analyzer-optin.osx.cocoa.localizability.EmptyLocalizationContextChecker
clang-analyzer-optin.osx.cocoa.localizability.NonLocalizedStringChecker
clang-analyzer-optin.performance.GCDAntipattern
clang-analyzer-optin.performance.Padding
clang-analyzer-optin.portability.UnixAPI
clang-analyzer-osx.API
clang-analyzer-osx.MIG
clang-analyzer-osx.NSObjectErrorDerefChecker
```

## ANNEXES

Je mets ici quelques compléments au guide pour éclaircir quelques points qui pourraient apparaître obscurs dans les exemples, réservé aux plus expérimentés d'entre vous en C++ et en design patterns. Cependant, je pense que ces détails vont plus vous compliquer la compréhension des bonnes pratiques que les faciliter. D'ailleurs, l'explication des patterns présentés dans l'annexe n'est pas le but du guide (il faudrait encore une centaine de pages supplémentaires). Donc, je laisserais quelques liens vers de la documentation pour ceux que ça intéresse.

### TYPES FORTS

Dans le guide, j'ai souvent utilisé des exemples avec des types Radian, Degree, Meter. L'implémentation la plus simple serait de définir ces types par un « using ».

```
using Degree = double;
using Meter = double;
using Radian = double;
```

Cela est tout à fait acceptable. Cependant, dans un contexte où la sécurité des données est importante, on aurait plutôt tendance à les écrire sous forme de type régulier. On évite ainsi des erreurs de programmation en interdisant les opérations sur des types différents.

```
class Degree
{
public:
    Degree() = default;
    explicit constexpr Degree(const double val) noexcept;
    constexpr bool ok() const noexcept;
    constexpr double get() const noexcept;
    constexpr Degree operator+() const noexcept;
    constexpr Degree operator-() const noexcept;
    constexpr Degree& operator+=(const Degree& r) noexcept;
    constexpr Degree& operator-=(const Degree& r) noexcept;
    constexpr Degree& operator*=(const Degree& r) noexcept;
    constexpr Degree& operator/=(const Degree& r) noexcept;
    constexpr auto operator<=>(const Degree& r) const noexcept = default; // C++20
    friend constexpr Degree& operator+(const Degree& lhs, const Degree& rhs) noexcept;
    friend constexpr Degree& operator-(const Degree& lhs, const Degree& rhs) noexcept;
    friend constexpr Degree& operator*(const Degree& lhs, const Degree& rhs) noexcept;
    friend constexpr Degree& operator/(const Degree& lhs, const Degree& rhs) noexcept;
    friend std::ostream& operator<<(std::ostream& os, const Degree& rhs);
private:
    double valeur_{0.0};
};
```

Vous avez remarqué la surcharge de tous les opérateurs peut être long à écrire (et encore, j'ai perdu patience, j'ai écrit l'operator<=> (three way) = default qui laisse le soin au compilateur de générer les opérateurs de comparaison (uniquement en C++20)).

L'écriture de ces classes peut être fastidieuse. De plus, ces classes possèdent toutes le même « modèle » : une valeur de type natif auquel on associe des opérations. Il existe un pattern template permettant de résoudre ce problème : le CRTP (Curiously Recurring Template Pattern). Le pattern CRTP [16] possède deux usages :

- La création d'interface statique. Cependant, on préférera la notion de concept en C++17 pour ce type d'usage.
- L'ajout de fonctionnalité à une classe.

Le détail de l'implémentation du pattern CRTP pour créer des types forts « strong type » est décrit dans le blog de Jonathan Boccara :

<https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/>

Vous pouvez également retrouver une description du pattern CRTP et un exemple d'implémentation de types forts dans le livre C++ Software Design de Klaus Iglberger :

<https://www.oreilly.com/library/view/c-software-design/9781098113155/>

Enfin, vous pouvez trouver différentes librairies implémentant les types forts :

<https://github.com/joboccara/NamedType>

[https://github.com/foonathan/type\\_safe](https://github.com/foonathan/type_safe)

[https://github.com/rollbear/strong\\_type/tree/main](https://github.com/rollbear/strong_type/tree/main)

[https://www.justsoftwaresolutions.co.uk/cplusplus/strong\\_typedef.html](https://www.justsoftwaresolutions.co.uk/cplusplus/strong_typedef.html)

Voyons rapidement l'équivalent de notre classe « Degree » avec la librairie NamedType. Avant de définir la classe « Degree », je souhaite déclarer une fonctionnalité de vérification de bornes, non présente dans la librairie NamedType :

```
template <auto min, auto max>
struct DomainValidable
{
    template <typename T>
    struct templ : fluent::crtp<T, templ> // ②
    {
        constexpr bool ok() const noexcept // ①
        {
            return (min <= this->underlying().get())
                && (this->underlying().get() < max);
        }
    };
};
```

Cette fonctionnalité me permettra d'ajouter la méthode « bool ok() » à mes types ①. Pour expliquer très rapidement le pattern CRTP ② :

- T est le type de la classe dérivée
- Fluent::crtp<T, templ> encapsule la méthode « T underlying() » me permettant d'accéder au méthode de la classe dérivée.

Il me suffit maintenant de déclarer la classe fluent::NamedType avec :

- Le type sous-jacent : « double »
- Un tag identifiant : « AngleDegTag »

- Les fonctionnalités : « Arithmetic » pour définir toutes les surcharges des opérateurs arithmétiques et « DegreeValidable » pour définir ma méthode « bool ok() »

La classe « Degree » est un alias de la précédente déclaration :

```
using DegreeValidable = DomainValidable<0.0, 360.0>;
using Degree = fluent::NamedType<double, struct AngleDegTag, fluent::Arithmetic,
DegreeValidable::templ>;
Degree operator"" _degree(long double value) { return Degree(value); }
```

Je vous propose un exemple complet pour montrer l'utilisation de la classe :

```
#include "named_type.hpp"
#include <numbers>
#include <ratio>
#include <cmath>

// -- Fonctionnalités personnalisées (CustomsSkills.h)

template <auto min, decltype(min) max>
struct DomainValidable
{
    template <typename T>
    struct templ : fluent::crtp<T, templ>
    {
        constexpr bool ok() const noexcept
        {
            return (min <= this->underlying().get())
                && (this->underlying().get() < max);
        }
    };
};

template <typename Origine, typename Converter>
struct ConvertibleFrom
{
    template <typename T>
    struct templ : fluent::crtp<T, templ>
    {
        static constexpr T from(const Origine o) noexcept
        {
            return T{ Converter{}(o.get()) };
        }
    };
};

template<typename T, auto Num, auto Den>
struct ratioFunctor
{
    constexpr T operator()(T t) const noexcept { return t * Num / Den; }
};

template <typename Origine, auto Num, auto Den>
```

```

using RatioConvertibleFrom = ConvertibleFrom<Origine, ratioFunctor<typename
Origine::UnderlyingType,Num,Den>>;

template <typename Origine, typename Ratio>
using StdRatioConvertibleFrom = RatioConvertibleFrom<Origine, Ratio::num,
Ratio::den>;

template <typename Destination, typename Converter>
struct ConvertibleTo
{
    template <typename T>
    struct templ : fluent::crtp<T, templ>
    {
        template <typename Type>
        constexpr std::enable_if_t<std::is_same_v<Destination,Type>, Destination>
to() const noexcept
        {
            return Destination{ Converter{}(this->underlying().get()) };
        }
    };
};

template <typename Destination, auto Num, auto Den>
using RatioConvertibleTo = ConvertibleTo<Destination, ratioFunctor<typename
Destination::UnderlyingType,Num,Den>>;

template <typename Destination, typename Ratio>
using StdRatioConvertibleTo = RatioConvertibleTo<Destination, Ratio::num,
Ratio::den>;

// -- Types forts (CoreTypes.h)

using MeterValidable = DomainValidable<-11000.0, 9000.0>;
using Meter = fluent::NamedType<double, struct DistanceMTag, fluent::Arithmetic,
MeterValidable::templ>;
Meter operator"" _meter(long double value) { return Meter(value); }

using DegreeValidable = DomainValidable<0.0, 360.0>;
using Degree = fluent::NamedType<double, struct AngleDegTag, fluent::Arithmetic,
DegreeValidable::templ>;
Degree operator"" _degree(long double value) { return Degree(value); }

using TenthDegree = fluent::NamedType<double, struct AngleTenthDegTag,
StdRatioConvertibleTo<Degree,std::deci>::templ>;
using Radian = fluent::NamedType<double, struct AngleRadTag,
RatioConvertibleFrom<Degree,std::numbers::pi,180.0>::templ,
RatioConvertibleTo<Degree,180.0,std::numbers::pi>::templ>;

// -- Exemple (main.cpp)

int main()

```

```

{
    auto distance = 75.5_meter;
    const auto latitude = 45.7_degree;
    if (distance.ok() && latitude.ok())
    {
        distance += 4.5_meter;
        const Degree calc = [](const Degree d)
        {
            // Bibliothèques externes : angle en radian
            const auto r = Radian::from(d);
            const double sinus = std::sin(r.get());
            const Radian res{std::asin(sinus)};
            return res.to<Degree>();
        }(latitude);

        std::cout << distance << "m, " << latitude << "° == " << calc << "° : "
            << std::boolalpha << (latitude == calc) << '\n';
        // 80m, 45.7° == 45.7° : true

        const TenthDegree angle{457.0}; // entrée en dixième de degree
        [](const Degree d)
        {
            std::cout << d << "°\n";
            // 45.7°
        }(angle.to<Degree>());
    }
    return 0;
}

```

Je vous laisse décortiquer l'exemple. Il est écrit en C++20 (donc n'oubliez pas de mettre l'option de compilation qui va bien dans votre compilateur favori si vous souhaitez le compiler). Pour finir, parlons un peu des inconvénients de l'implémentation de types forts avec le pattern CRTP. Il demande beaucoup de rigueur et de discipline de l'équipe de développement. Deux dérives peuvent se produire :

- les développeurs cassent l'encapsulation et utilisent la méthode « get() » sans utiliser les surcharges.
- les développeurs font porter du code métier dans les types. Par exemple, il serait tentant d'ajouter une fonctionnalité de sérialisation des types. Cependant, les types perdraient leurs principes génériques. (Le pattern CRTP utilisé dans ce cadre n'est pas un design pattern, c'est un pattern d'implémentation permettant de faciliter l'écriture des types régulier).

Cependant, l'utilisation d'une bibliothèque plus complète comme Type\_Safe permettrait peut-être de limiter les mauvais usages.

## EXTERNAL POLYMORPHISM ET TYPE ERASURE

Pour présenter les patterns External Polymorphism et Type Erasure, prenons l'exemple d'un programme qui réalise le décodage des trames à partir d'un fichier de spécification de trames. L'exemple est simplifié :

- La trame ne possède que deux formats de données (Chaîne de caractères ASCII et Valeur binaire sur 4 octets)
- Les données décodées n'ont que trois types (Chaîne de caractère, entier, double)

- Trois filtres de décodage :
  - Hexadécimal : converti une chaîne de caractères représentant une valeur hexadécimale en entier.
  - Ratio : applique un ratio à une valeur arithmétique.
  - Degree : normalise une valeur arithmétique représentant un angle entre [0 ; 360[.
- La lecture du fichier de spécification, la réception d'une trame et la construction dynamique de l'analyseur ne sont pas implémentées.

```
#include <ostream>
#include <variant>
#include <memory>

// -- Types (FrameTypes.h)

// Types de données analysés
using Data = std::variant<std::string,int,double>;

template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

std::ostream& operator<< ( std::ostream& os, Data data )
{
    std::visit([&os](auto&& arg){ os << arg; }, data);
    return os;
}

// Représentation d'un champ de donnée
class Field
{
public:
    template< typename T > // ② Type Erasure
    explicit Field( T field )
        : pimpl_( std::make_unique<Model<T>>( std::move(field) ) )
    {}

    Field( Field const& field ) : pimpl_( field.pimpl_->clone() ) {}

    Field& operator=( Field const& field )
    {
        pimpl_ = field.pimpl_->clone();
        return *this;
    }

    ~Field() = default;
    Field( Field&& ) = default;
    Field& operator=( Field&& field ) = default;

    Data decode() const { return pimpl_->decode(); }
```



```

private:
    struct Concept // ❶ External Polymorphism
    {
        virtual ~Concept() = default;
        virtual Data decode() const = 0;
        virtual std::unique_ptr<Concept> clone() const = 0;
    };

    template< typename T >
    struct Model : public Concept
    {
        explicit Model( T const& field ) : field_( field ) {}
        explicit Model( T&& field ) : field_( std::move(field) ) {}

        Data decode() const override
        {
            return field_.decode();
        }

        std::unique_ptr<Concept> clone() const override
        {
            return std::make_unique<Model<T>>(*this);
        }

        T field_;
    };

    std::unique_ptr<Concept> pimpl_;
};

struct NamedField
{
    std::string name;
    Field field;
};

// -- Format de trame (FrameFormats.h)

#include <string>
#include <cmath>
#include <sstream>

// Format de données

class Text
{
public:
    explicit Text( const std::string& data )
        : data_{ data }
    {}

```

```

    Data decode() const { return data_; }

private:
    std::string data_;
};

class Int32
{
public:
    explicit Int32( const int data )
        : data_{ data }
    {}

    Data decode() const { return data_; }

private:
    int data_;
};

// Decorators

class Ratio
{
public:
    Ratio( const double factor, Field field )
        : field_( std::move(field) )
        , factor_( factor )
    {}

    Data decode() const
    {
        const double factor = factor_;
        return std::visit(overloaded{
            [factor](auto arg) -> Data { return factor * arg; },
            [](const std::string& arg) -> Data { return arg; }
        }, field_.decode());
    }

private:
    Field field_;
    double factor_;
};

using IOSManipulator = std::ios_base&(std::ios_base&);
template <IOSManipulator Manipulator>
class Base32
{
public:
    explicit Base32( Field field )
        : field_( std::move(field) )
    {}

```

```

    Data decode() const
    {
        return std::visit(overloaded{
            [](auto arg) -> Data { return arg; },
            [](const std::string& arg) -> Data {
                int x;
                std::stringstream ss;
                ss << Manipulator << arg;
                ss >> x;
                return x;
            }
        }, field_.decode());
    }

private:
    Field field_;
};
using Hex32 = Base32<std::hex>;

class Degree
{
public:
    explicit Degree( Field field )
        : field_( std::move(field) )
    {}

    Data decode() const
    {
        return std::visit(overloaded{
            [](auto arg) -> Data { return std::fmod(arg, 360.0); },
            [](const std::string& arg) -> Data { return arg; }
        }, field_.decode());
    }

private:
    Field field_;
};

// -- Example (main.cpp)
#include <cstdlib>
#include <vector>
#include <iostream>

int main()
{
    // Lecture du fichier de spécification d'une trame
    // Construction et lecture d'une trame
    std::vector<NamedField> frame
    {
        { "msgName", Text{"Position"} },

```

```

        { "latitude", Degree{ Ratio{ 1. / 1000000., Hex32{ Text{"18450209"} } } } },
    },
    { "error", Int32{0} },
};

for (const auto item : frame)
{
    std::cout << item.name << " : " << item.field.decode() << '\n';
}
return EXIT_SUCCESS;
}

```

❶ Le design pattern External Polymorphism [17] permet de rendre une classe non polymorphique dynamiquement utilisable avec du polymorphisme dynamique. La classe template a pour but de simplifier l'écriture des opérateurs obligatoires en polymorphisme dynamique (voir §Hierarchies de classes). Elle se base sur deux classes :

- Une classe Concept qui est la classe interface
- Une classe template Model qui représente la hiérarchie des classes dérivées

❷ Le design pattern Type Erasure [18] permet d'encapsuler un type issu d'un argument template. La manipulation du type est opacifiée par une classe Interface.

Comme pour l'autre exemple, je vous laisse le soin d'étudier le programme. Il est aussi écrit en C++20. Le code présente plusieurs design patterns. Pouvez-vous m'indiquer lesquelles ?

*(J'espère que vous avez gardé le petit miroir des Mystères de Pékin pour visualiser la réponse ou cliquer sur le cadre sinon)*

```

Decorator (classes Ratio, Base32 et Degree)' Strategy (paramètre template IOManipulator)
Type Erasure (classe Field)' External Polymorphism (classes Concept et Model)' Visitor (type Data)

```

- [1] C++ Core Guideline: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- [2] Modern C: <https://gustedt.gitlabpages.inria.fr/modern-c/>
- [3] Documentation C++, std::condition\_variable: [https://en.cppreference.com/w/cpp/thread/condition\\_variable](https://en.cppreference.com/w/cpp/thread/condition_variable)
- [4] style de codage C++: <https://www.stroustrup.com/Programming/PPP-style.pdf>
- [5] Return Value Optimization: [https://en.cppreference.com/w/cpp/language/copy\\_elision](https://en.cppreference.com/w/cpp/language/copy_elision)
- [6] défense en profondeur: <https://www.it-connect.fr/cybersecurite-defense-en-profondeur/>
- [7] RAII : <https://en.cppreference.com/w/cpp/language/raii>
- [8] C++ Core Guidelines Explained: Best Practices for Modern C++, Rainer Grimm
- [9] API C : <https://isocpp.org/wiki/faq/mixing-c-and-cpp>
- [10] Slicing : <https://www.geeksforgeeks.org/object-slicing-in-c>
- [11] Suite de Golomb: [https://fr.wikipedia.org/wiki/Suite\\_de\\_Golomb](https://fr.wikipedia.org/wiki/Suite_de_Golomb)
- [12] Benchmark conteneur STL : <https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html>
- [13] Lock-free programming : <https://preshing.com/20120612/an-introduction-to-lock-free-programming/>
- [14] Invariant : [https://fr.wikipedia.org/wiki/Programmation\\_par\\_contrat](https://fr.wikipedia.org/wiki/Programmation_par_contrat)
- [15] Généricité : <https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9ricit%C3%A9>
- [16] CRTP : <https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/>
- [17] External Polymorphism : <https://www.dre.vanderbilt.edu/~schmidt/PDF/External-Polymorphism.pdf>
- [18] Type Erasure : <https://www.modernescpp.com/index.php/type-erasure/>
- [19] Algorithm : <https://en.cppreference.com/w/cpp/algorithm>
- [20] Acyclic Visitor : <https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/acv.pdf>
- [21] Partition d'équivalence : <https://latavernedutesteur.fr/2018/09/06/techniques-basees-sur-les-specifications-1-7-les-partitions-dequivalence/>
- [22] Analyse des valeurs limites : <https://latavernedutesteur.fr/2018/09/13/techniques-basees-sur-les-specifications-2-7-analyse-des-valeurs-limites/>
- [23] ADL : <https://en.cppreference.com/w/cpp/language/adl>