



UN PETIT POINT SUR LA PROGRAMMATION PAR CONTRAT EN C++

15 NOVEMBRE 2021, LUC HERMITTE



Sommaire

- 1 Avant-propos
- 2 PpC : la théorie
- 3 Techniques
- 4 Conclusion
- 5 Références



Plan partiel

① Avant-propos

② PpC : la théorie

③ Techniques

④ Conclusion

⑤ Références



Avant-propos

- La programmation par contrat, définition et principes
- Techniques d'exploitations
 - approche offensive VS défensive
 - autres astuces



Plan partiel

① Avant-propos

② PpC : la théorie

Erreurs de programmation

La Programmation par Contrat

③ Techniques

④ Conclusion

⑤ Références



C'est quoi une erreur de programmation ?

- Erreur dans des algorithmes/calculs ;
 - p.ex. `sin()` qui renvoie des valeurs supérieures à 1, mélange entre des pieds et des mètres, ...
- Erreur dans des suppositions
 - p.ex. pointeurs non nuls, indice hors bornes. . .



Que faire en cas d'erreur de programmation ?

➡ Rien, pas même chercher à la détecter



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion
 - On est prévenus, en phase de développement et de tests, que quelque chose ne va pas, et on dispose d'un contexte exact du soucis au moment où il est détecté ;
 - on fait comme si tout allait bien en phase de prod (sauf si on décide de doubler par une exception si le projet exige de la *programmation défensive*.)



Que faire en cas d'erreur de programmation ?

- Rien, pas même chercher à la détecter
 - Programme qui crashe plus loin, sans contexte ;
 - ou programme qui donne des résultats aberrants, qui seront détectés, un jour, ou peut-être jamais.
- La détecter pour lancer une exception
 - On est prévenus que quelque chose ne va pas,
 - on ne plante pas (ni en prod, ni en dev & tests), mais ...
 - perte du contexte pour investigation par l'équipe de dev
- La détecter pour claquer une assertion
 - On est prévenus, en phase de développement et de tests, que quelque chose ne va pas, et on dispose d'un contexte exact du soucis au moment où il est détecté ;
 - on fait comme si tout allait bien en phase de prod (sauf si on décide de doubler par une exception si le projet exige de la *programmation défensive*.)
- Attendre le C++23



Quels outils pour s'en protéger ?

➤ Invariants statiques

- p.ex. constructeur vs initialisation différée, référence vs pointeur, `signed` vs `unsigned`, ...

➤ Typage renforcé

- p.ex. cf. `Boost.unit`, ou les *User-Defined literals* du C++11.

➤ Assertions statiques

- p.ex. taille tableau statique == nombre d'énumérés, type reçu supporte au moins 42000 valeurs (prog. générique), ...

➤ Outils d'analyse statique du code source (Frama-C, Polyspace, autre ?)

➤ TU, TV et assertions

➤ Formaliser les contrats (PpC)



Quels outils pour s'en protéger ?

➤ Invariants statiques

- p.ex. constructeur vs initialisation différée, référence vs pointeur, signed vs unsigned, ...

➤ Typage renforcé

- p.ex. cf. Boost.unit, ou les *User-Defined literals* du C++11.

➤ Assertions statiques

- p.ex. taille tableau statique == nombre d'énumérés, type reçu supporte au moins 42000 valeurs (prog. générique), ...

➤ Outils d'analyse statique du code source (Frama-C, Polyspace, autre ?)

➤ TU, TV et assertions

➤ Formaliser les contrats (PpC)



La Programmation par Contrat

Ils sont partout !

➤ Qu'on le veuille ou non



La Programmation par Contrat

Ils sont partout !

- Qu'on le veuille ou non
- Toute fonction a un contrat



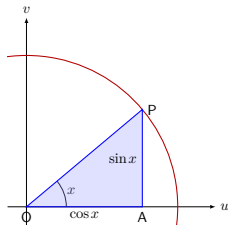
La Programmation par Contrat

Ils sont partout - Illustration `sin`

Exemple

`double sin(double x)`

- x est en nombre flottant (double précision) exprimant un angle en radian
- le résultat est l'ordonnée du point P sur le cercle unitaire dont l'angle \widehat{AOP} vaut x
- incidemment compris entre -1 et 1 .



La Programmation par Contrat

Ils sont partout - Illustration `is_in_range` [KRZEMIENSKI, P2466 : *The notes on contract annotations*]

```
// Check val belongs to [low, high]  
bool is_in_range(int val, int low, int high);
```

Contrat exprimé au travers

- des noms et types des paramètres
- du nom de la fonction
- du commentaire (intervalle fermé `[low, high]`)



- *Pré-condition* : conditions que doit remplir l'appelant d'une fonction pour que cette dernière ait une chance de bien se dérouler. En cas de non respect du contrat par l'appelant, l'appelé ne garantit rien et tout peut arriver.
- *Post-condition* : conditions vérifiées par l'appelé, et la valeur retournée, après appel d'une fonction.
- *Invariant* : ensemble de propriétés qu'une classe doit respecter avant et après chaque appel de fonction de l'interface.



- La PpC, c'est avant tout des garanties si tout va bien et c'est tout.
- On respecte \Rightarrow on aura un comportement prévisible et valide. Mais si on ne respecte pas le contrat, tout peut arriver.
C'est (quasi) le pays des *Undefined behaviours*.



Programmation par Contrat

Pré-conditions

- À rapprocher des domaines de définition
- Si respect des pré-conditions avant appel, alors l'appel doit réussir et produire les résultats attendus dans les post-conditions.
- Le responsable est l'appelant.
- Les assertions sont nos amies – mode *fail fast*.



- Garanties sur résultats d'une fonction si pré-conditions remplies, et aucune erreur de *runtime*.
- Si la fonction sait qu'elle ne peut pas remplir ses post-conditions, alors elle doit échouer.
- Il ne s'agit pas de détecter les erreurs de programmation, mais de contexte.
- Le responsable est l'appelé.
- Relève plus du test unitaire que de l'assertion.



Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)  
[[post: i<j: res[i] <= res[j]]]
```



Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ?? // unique sort
```



Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ?? // unique sort
```

```
sort({1, 3, 2, 2}) -> {0, 1, 2, 3} ?? // autant d'éléments sans rapport
```



Programmation par Contrat

Post-conditions

```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ?? // unique sort
```

```
sort({1, 3, 2, 2}) -> {0, 1, 2, 3} ?? // autant d'éléments sans rapport
```

```
sort({1, 2, 3, 2}) -> {1, 2, 3, 3} ?? // les mêmes en quantités différentes
```



```
std::vector<int> sort(std::vector<int> s)
```

```
[[post: i<j: res[i] <= res[j]]]
```

```
sort({2, 1, 3, 2}) -> {1, 2, 3} ?? // unique sort
```

```
sort({1, 3, 2, 2}) -> {0, 1, 2, 3} ?? // autant d'éléments sans rapport
```

```
sort({1, 2, 3, 2}) -> {1, 2, 3, 3} ?? // les mêmes en quantités différentes
```

La post-condition,

- ce n'est pas seulement trié, mais exactement les mêmes éléments,
- ce qui coûte aussi cher ici.



S'applique à des zones durant lesquelles une propriété restera vraie.

- Invariant de boucle (voire, variant de boucle)
- Référence (pointeur jamais nul)
- Variable (devrait toujours être «*Est utilisable, et est dans un état cohérent et pertinent*», positionné après construction)
- Invariant de classe (propriété toujours observable depuis du code extérieur aux instances de la classe).



Programmation par Contrat

Acteurs et Responsabilités

```
double metier() { // écrit par l'Intégrateur  
    const double i = interrogeES(); // écrit par le responsable UI  
    return sqrt(i); // écrit par le Mathématicien  
}
```

sqrt échoue (assertion, résultat abérrant, NaN, ...)

- Si i est positif \Rightarrow Mathématicien
- Si i est négatif \Rightarrow pas le Mathématicien mais
 - Si `interrogeES()` a pour post-cond positif \Rightarrow Responsable UI
 - Sinon \Rightarrow Intégrateur



Plan partiel

① Avant-propos

② PpC : la théorie

③ Techniques

Exploitation des types

Approche offensive – *narrow contracts*

Approche défensive – *wide contracts*

Illustration : offensif VS défensif

Critique de l'approche fortement défensive

Tests unitaires

④ Conclusion

⑤ Références



Dans un monde idéal on est capable de passer la vérification des contrats

- ① au compilateur
 - → typage...
- ② à des outils d'analyse statique du code
 - ex. FRAMA-C qui analysera des annotations dédiées, C++23 ?
- ③ via des vérifications réalisées en phase de tests à l'exécution
- ④ Mais parfois, souvent, on ne peut pas vérifier des contrats
 - *"Pointe sur une zone mémoire allouée de N éléments"*



Exploitation des types

Le C++ vérifie la cohérence des types à la compilation (avec des faiblesses, certes)

- On ne pourra pas confondre (facilement) un `bool` avec un `complex`,
- On pourra exprimer une différence entier VS booléen,
- On pourra renforcer cohérence autour des tableaux contigus (`std::span`)
- On pourrait avoir des *strong types* pour distinguer `line_t` et `column_t`
- On pourra avoir des indirections non nulles
- On peut se définir des types avec des invariants... [SUTTER, *GotW #101 Solution : Preconditions, Part 2*]



Exploitation des types

not_null – <https://gcc.gnu.org/z/hxMTGsvE5>

```
#include <cassert>
#include <type_traits>

// Simplification de gsl::not_null<>
// https://github.com/microsoft/GSL/blob/main/include/gsl/pointers
template <typename T> struct not_null
{
    constexpr explicit not_null(T v) : h{v} { assert(v); }
    constexpr not_null(std::nullptr_t) = delete;
    constexpr not_null& operator=(std::nullptr_t) = delete;

    constexpr
    std::conditional_t<std::is_copy_constructible_v<T>, T, const T&>
    get() const
    {
        assert(h);
        return h;
    }
    constexpr operator T() const { return get(); }
    constexpr decltype(auto) operator->() const { return get(); }
    constexpr decltype(auto) operator*() const { return *get(); }

private:
    T h;
};
```

```
int f(not_null<int*> p) {
    return *p; // toujours OK!
}

int g(not_null<int*> p) {
    return 2 * f(p); // enchainement garanti
}

int* prehistoric_factory();

int main(int argc, char**argv)
{
    // Il manquerait des utilitaires ici
    // g(&argc); // error: could not convert '& argc' from 'int*' to 'not_null<int*>'
    g(not_null{&argc}); // OK

    // Parfait, on n'en veut pas!
    // g(prehistoric_factory()); // error: could not convert 'prehistoric_factory()'

    auto p = prehistoric_factory();
    g(not_null{p}); // bouhhh! On n'a pas vérifié

    if (p) {
        g(not_null{p}); // parfait
    }
}
```

De manière générale on peut lier des données entre elles en un tout cohérent lié par des invariants. [SUTTER, *GotW #101 Solution : Preconditions, Part 2*]

Exemples

➤ `std::string` VS `struct {char *s; size_t len;};`

➤ `std::span` VS `struct {T *p; size_t len;};`



De manière générale on peut lier des données entre elles en un tout cohérent lié par des invariants. [SUTTER, *GotW #101 Solution : Preconditions, Part 2*]

Exemples

➤ `std::string` VS `struct {char *s; size_t len;};`

➤ `std::span` VS `struct {T *p; size_t len;};`

➤ Avant

```
// Check val belongs to [low, high]
bool is_in_range(int val, int low, int high);
```

Après

```
struct interval {
    interval(int low, int high) [[pre: low <= high]];
    ...
};

bool is_in_range(int val, interval const& rng);
```

Approche offensive – *narrow contracts*

En gros : tout écart à un contrat est UB ou *unspecified* [SUTTER, *GotW* #102
Solution : Assertions and "UB"]

➤ *undefined behaviour*

```
void f(int *p) {  
    stuff(*p); // what if nullptr?  
}
```

➤ *unspecified behaviour*

```
bool is_in_range(int val, int low, int high);  
...  
auto in = is_in_range(lo, 42, hi); // unspecified  
// exploitation de "in" potentiellement undefined
```

Approche offensive – *narrow contracts*

En gros : tout écart à un contrat est UB ou *unspecified* [SUTTER, *GotW* #102

Solution : Assertions and "UB"]

➤ *undefined behaviour*

```
void f(int *p) {  
    stuff(*p); // what if nullptr?  
}
```

➤ *unspecified behaviour*

```
bool is_in_range(int val, int low, int high);  
...  
auto in = is_in_range(lo, 42, hi); // unspecified  
// exploitation de "in" potentiellement undefined
```

Approche typique : *fail-fast* ou équivalent

➤ avec assert

➤ avec le C++23 ?



Approche offensive – *narrow contracts*

assert

Selon le mode, assert va :

Release ne rien faire

- Release == -DNDEBUG

Debug tester la condition, et arrêter le programme avec `std::abort`

- Cela nous laisse avec un programme dans son état exact au moment de la détection
 - ➞ Exploitable en interactif ou dans un *core dumped*
- Une exception aurait provoqué un *stack unwinding* (sauf si breakpoint sur exception...)



Programmation (excessivement) Défensive

Principes

- Objectif : Un programme ne doit jamais s'arrêter afin de toujours pouvoir continuer.
- On s'intéresse à la robustesse d'un programme malgré ses erreurs que l'on laisse de côté.
- Définition assez floue (https://en.wikipedia.org/wiki/Defensive_programming) qui mélange vite : l'offensif, le sécurisé, la vérification des entrées, et l'excessivement défensif.



PpC offensive ou exceptions ?

Illustration : code cavalier

```
int main()
{
    try {
        std::ifstream f("distances.txt");
        if (!f) throw std::runtime_error("Cannot open distances.txt");

        double d;
        while (f >> d) {
            const auto sq = std::sqrt(d);
            treat(sq);
        }
        return EXIT_SUCCESS;
    } catch (std::exception const& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    return EXIT_FAILURE;
}
// Vim: let $CXXFLAGS='-std=c++14 -g'
```



PpC offensive ou exceptions ?

Illustration : défensif bâclé

```
namespace my {
double sqrt(double n) {
    if (n<0) throw std::domain_error("Negative number sent to sqrt");
    return std::sqrt(n);
}
} // my namespace

int main()
{
    try {
        std::ifstream f("distances.txt");
        if (!f) throw std::runtime_error("Cannot open distances.txt");

        double d;
        while (f >> d) {
            const auto sq = my::sqrt(d);
            treat(sq);
        }
        return EXIT_SUCCESS;
    } catch (std::exception const& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    return EXIT_FAILURE;
}
// Vim: let $CXXFLAGS='-std=c++14 -g'
```



PpC offensive ou exceptions ?

Illustration : défensif corrigé

```
namespace my {
double sqrt(double n) {
    if (n<0) throw std::domain_error("Negative number sent to sqrt");
    return std::sqrt(n);
}
} // my namespace

int main()
{
    const auto file = "distances.txt";
    try {
        std::ifstream f(file);
        if (!f) throw std::runtime_error("Cannot open distances.txt");

        double d;
        for (std::size_t l = 1 ; f >> d ; ++l) {
            double sq ;
            try {
                sq = my::sqrt(d);
            }
            catch (std::logic_error const&) {
                throw std::runtime_error(
                    "Invalid negative distance " + std::to_string(d)
                    + " at the " + std::to_string(l)
                    + "th line in distances file " + file);
            }
            treat(sq);
        }
        if (!f.eof()) throw std::runtime_error("oops!");
    }
    catch (std::exception const& e) {
        intusure::Program::SUCCESS
    }
}
```

PpC offensive ou exceptions ?

Illustration : offensif avec validation des entrées

```
namespace my {  
    /// @pre n >= 0  
    double sqrt(double n) {  
        assert(n >= 0 && "sqrt can't process negative numbers");  
        return std::sqrt(n);  
    }  
} // my namespace  
  
int main() {  
    const auto file = "distances.txt";  
    try {  
        std::ifstream f(file);  
        if (!f) throw std::runtime_error("Cannot open distances.txt");  
  
        double d;  
        for (std::size_t l = 1; f >> d; ++l) {  
            if (d <= 0)  
                throw std::runtime_error(  
                    "Invalid negative distance " + std::to_string(d)  
                    + " at the " + std::to_string(l) + "th line in distances file " + file);  
            const auto sq = my::sqrt(d);  
            treat(sq);  
        }  
        if (!f.eof()) throw std::runtime_error("Des symboles incorrects dans le fichier");  
        return EXIT_SUCCESS;  
    } catch (std::exception const& e) {  
        std::cerr << "Error: " << e.what() << "\n";  
    }  
    return EXIT_FAILURE;  
}
```

En résumé : comparaison

Cas de la Programmation (excessivement) Défensive

- On veut résister aux erreurs de logique.
- On ne plante jamais à quitte à :
 - renvoyer des valeurs numériques fausses
 - renvoyer des valeurs par défaut (`nullptr`, 0, -1, 42, NaN, ...), voire des valeurs sentinelles (`nullptr`, -1)
 - remonter l'erreur de logique (codes de retour, ou exception (`std::logic_error`))
- On paie toujours sur `sqrt(1-sin(x))`
- On a un contrat élargi (*wide contract*) qui résiste aux ruptures de contrats.
- Ex : `std::sqrt()`, `std::vector::at()`



En résumé : comparaison

Cas de la Programmation par Contrat (offensive)

- On veut traquer et éliminer les erreurs de logique au plus tôt.
- On laisse un ~UB se produire sur un non respect de contrat
- On ne paie jamais sur `sqrt(1-sin(x))`
- On a un contrat restrictif (*narrow contract*)
- On peut exploiter des assertions, ou des outils de preuve formelle (direction prise par le C++23).
- Ex : `std::vector::operator[]()`, `std::stack::pop()`



Critique de l'approche fortement défensive

➤ Approche particulièrement séduisante pour :

- les débutants
- la résilience aux situations impossibles
 - surtout sur les systèmes critiques !



Critique de l'approche fortement défensive

➤ Approche particulièrement séduisante pour :

- les débutants
- la résilience aux situations impossibles
 - surtout sur les systèmes critiques !
- Mais... vraiment ?



Critique de l'approche fortement défensive

Recoverable errors

[DUFFY], [SUTTER, *P0709 : Zero-overhead deterministic exceptions : Throwing values*] estiment qu'on ne peut pas se rétablir d'une erreur de programmation.

- A *recoverable error* is usually the result of programmatic data validation [...] The response might be to communicate the situation to an end-user, retry, or abandon the operation entirely, however it is a predictable and, frequently, planned situation, despite being called an "error."
- A bug is a kind of error the programmer didn't expect. Inputs weren't validated correctly, logic was written wrong, or any host of problems have arisen. Such problems often aren't even detected promptly; it takes a while until "secondary effects" are observed indirectly, at which point significant damage to the program's state might have occurred. Because the developer didn't expect this to happen, all bets are off. All data structures reachable by this code are now suspect. And because these problems aren't necessarily detected promptly, in fact, a whole lot more is suspect. Depending on the isolation guarantees of your language, perhaps the entire process is tainted.



Vive les corruptions !

Types de comportements inattendus

➤ Erreurs pas très graves (...!)

```
for (std::size_t i=0 ; i < size(v) ; ++i)  
    stuff(v.at(i+1));
```

➤ Corruptions : <https://godbolt.org/z/PTe6YYaxM>

```
char s[4];  
int a = 25;  
  
std::cin >> s;
```



Vive les corruptions !

Types de comportements inattendus

➤ Erreurs pas très graves (...!)

```
for (std::size_t i=0 ; i < size(v) ; ++i)  
    stuff(v.at(i+1));
```

➤ Corruptions : <https://godbolt.org/z/PTe6YYaxM>

```
char s[4];  
int a = 25;  
  
std::cin >> s;
```

Vous voyez le soucis ?



Taux de couverture

A supposer que l'on puisse prévoir toutes les situations d'erreurs de programmation et que l'on puisse les remonter pour se remettre dans un état neutre...

- Le programme est plus complexe : plus de chemins à gérer
- Il y a plus de lignes à couvrir
- Voire il y a des lignes qui feront chuter le taux de couverture

```
for (std::size_t i=0 ; i < size(v) ; ++i)  
    stuff(v.at(i));
```



Les TU sont particulièrement adaptés aux post-conditions.

- ➞ Ils permettent des validations complexes et couteuses d'état en sortie de fonctions.



Les TU sont particulièrement adaptés aux post-conditions.

- Ils permettent des validations complexes et coûteuses d'état en sortie de fonctions.
- cf. coûts pour *sinus(x)*, `sort(collection)`...



- De base ils permettent d'exécuter des scénarios de test et vérifier qu'aucun contrat n'est rompu.



Tests unitaires

et pré-conditions

- De base ils permettent d'exécuter des scénarios de test et vérifier qu'aucun contrat n'est rompu.
- Certains testent aussi les comportements de fonctions quand l'appelant ne respecte pas les pré-conditions...
 - Conséquences des *death tests* :
 - Règle de Lakos : pas de `noexcept` sur les fonctions avec *narrow-contract*
 - Invalidée avec [BERGÉ, P1656 : "*Throws : Nothing*" should be *noexcept*]
 - La tendance, pas d'exception pour des contrats, continue avec les *proposals* courants pour supporter les contrats en C++.



Plan partiel

- ① Avant-propos
- ② PpC : la théorie
- ③ Techniques

- ④ Conclusion
 - Perspectives
 - Questions ?

- ⑤ Références



Je n'ai pas traité

- Discussions autour du support éventuel de la PpC en C++23
- Rapport aux concepts
- L'analyse statique de code



Questions ?







Plan partiel

- ① Avant-propos
- ② PpC : la théorie

- ③ Techniques
- ④ Conclusion
- ⑤ **Références**







Références I

-  Agustín BERGÉ. *P1656 : "Throws : Nothing" should be noexcept.*
<https://wg21.link/p1656>. 2020.
-  Julien BLANC. *Programmation par contrat, application en C++.*
http://julien-blanc.developpez.com/articles/cpp/Programmation_par_contrat_cplusplus/. Déc. 2009.
-  Bjarne STROUSTRUP et Herb SUTTER, éd. *C++ Core Guidelines.*
<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. C++CG, 2015.
-  Gabriel DOS REIS et al. *P0542 : Support for contract based programming in C++.* <http://wg21.link/p0542>. 2018.



Références II

-  Joe DUFFY. *The Error Model : Bugs Aren't Recoverable Errors!*
<http://joeduffyblog.com/2016/02/07/the-error-model/#bugs-arent-recoverable-errors>. Fév. 2016.
-  Philippe DUNSKI et Luc HERMITTE. *Coder Efficacement - Bonnes pratiques et erreurs à éviter (en C++)*. www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html. Fév. 2014.
-  Andrzej KRZEMIENSKI. *P2466 : The notes on contract annotations*.
<http://wg21.link/p2466>. 2021.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 1*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-1/>. Traduction. Jan. 2013.






Références III

-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 2*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-2/>. Traduction. Fév. 2013.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 3*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-3/>. Traduction. Mar. 2013.
-  Andrzej KRZEMIENSKI. *Préconditions en C++ - partie 4*. <http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-4/>. Traduction. Avr. 2013.
-  John LAKOS. *Defensive Programming Done Right - part 1*. https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube_gdata. Sept. 2014.







Références IV

-  John LAKOS. *Defensive Programming Done Right - part 2*. https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube_gdata. Sept. 2014.
-  Bertrand MEYER. *Conception et programmation orientées objet*. <http://www.editions-eyrolles.com/Livre/9782212122701/conception-et-programmation-orientees-objet>. Jan. 2008.
-  Gregory PAKOSZ. *Assertions Or Exceptions ?* <https://pempek.net/articles/2013/11/16/assertions-or-exceptions/>. Nov. 2013.
-  Gregory PAKOSZ. *Cross Platform C++ Assertion Library*. <https://pempek.net/articles/2013/11/17/cross-platform-cpp-assertion-library/>. Nov. 2013.







Références V

-  John REGEHR. *Use of assertions*.
<https://blog.regehr.org/archives/1091>. Fév. 2014.
-  Herb SUTTER. *GotW #100 Solution : Preconditions, Part 1*.
<https://herbsutter.com/2021/02/25/gotw-100-solution-preconditions-part-1-difficulty-8-10/>. Fév. 2021.
-  Herb SUTTER. *GotW #101 Solution : Preconditions, Part 2*.
<https://herbsutter.com/2021/03/25/gotw-101-solution-preconditions-part-2-difficulty-7-10/>. Mar. 2021.
-  Herb SUTTER. *GotW #102 Solution : Assertions and "UB"*.
<https://herbsutter.com/2021/06/03/gotw-102-solution-assertions-and-ub-difficulty-7-10/>. Juin 2021.



Références VI

-  Herb SUTTER. *P0709 : Zero-overhead deterministic exceptions : Throwing values*. <https://wg21.link/p0709>. 2019.
-  Herb SUTTER. *When and How to Use Exception*. <http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836>. Jan. 2004.
-  Matthew WILSON. *Contract Programming 101*. <http://www.artima.com/cppsource/deepspace3.html>. 2006.
-  Matthew WILSON. *Imperfect C++*. 2004.

