

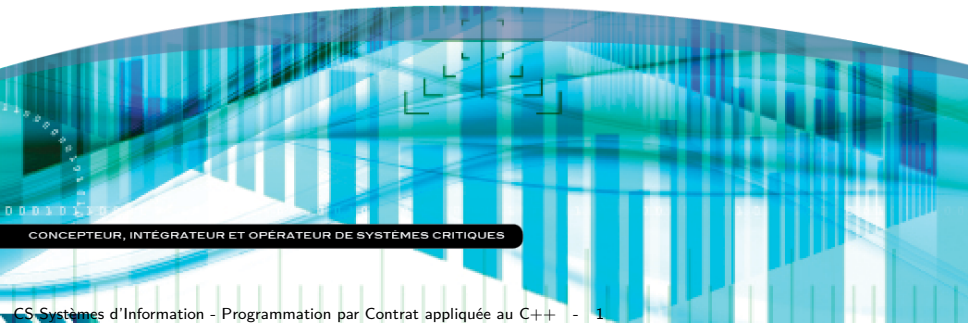
# Programmation par Contrat appliquée au C++

## Capitole du Libre

Hermitte Luc

CS Systèmes d'Information

17 novembre 2017



## Programmation par Contrat appliquée au C++

Programmation par Contrat appliquée au C++  
Capitole du Libre

Hermitte Luc  
CS Systèmes d'Information  
17 novembre 2017



2018-11-26

# Sommaire

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références



Sommaire

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références

— Sommaire

# Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références



Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 3

## Programmation par Contrat appliquée au C++

└ Avant-propos

└ Plan partiel

2018-11-26

- David Côme, Philippe Lacour
- Julien Blanc, Guilhem Bonnefille, alex\_deba, Sébastien Dinot, Iradrille, Cédric Poncet-Montange
- Style ©CS Système d'Informations
- Contenu sous licence Creative Commons CC-BY-NC-SA



Avant-propos  
Crédits et remerciements

- David Côme, Philippe Lacour
- Julien Blanc, Guilhem Bonnefille, alex\_deba, Sébastien Dinot, Iradrille, Cédric Poncet-Montange
- Style ©CS Système d'Informations
- Contenu sous licence Creative Commons CC-BY-NC-SA

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 4

## Programmation par Contrat appliquée au C++

└ Avant-propos

└ Avant-propos

2018-11-26

- Mes billets de blog :  
<http://luchermittle.github.io/blog/2014/05/24/programmation-par-contrat-un-peu-de-theorie/>
- Discussions intéressantes sur dvpz :  
<https://www.developpez.net/forums/d1581316/c-cpp/cpp/apprendre-programmation-contrat-ppc-cpp/>



Avant-propos  
Plus d'infos

- Mes billets de blog :  
<http://luchermittle.github.io/blog/2014/05/24/programmation-par-contrat-un-peu-de-theorie/>
- Discussions intéressantes sur dvpz :  
<https://www.developpez.net/forums/d1581316/c-cpp/cpp/apprendre-programmation-contrat-ppc-cpp/>



# Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références



Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 6

Programmation par Contrat appliquée au C++

└ Besoin de gestion des erreurs

└ Plan partiel

2018-11-26

# Besoin de gestion des erreurs

Types d'erreurs	Moyens et méthodes	Périmètre
Liées au langage	Compilateur	dev
De logique	TU, tests, PpC, analyse statique, preuve formelle...	archi/dev
D'environnement	détection dynamique à l'exécution	dev/user



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 7

## Programmation par Contrat appliquée au C++

### └ Besoin de gestion des erreurs

### └ Besoin de gestion des erreurs

Besoin de gestion des erreurs

Types d'erreurs	Moyens et méthodes	Périmètre
Liées au langage	Compilateur	dev
De logique	TU, tests, PpC, analyse statique, preuve formelle...	archi/dev
D'environnement	détection dynamique à l'exécution	dev/user

1. le programme n'est pas conforme à la syntaxe ou grammaire du langage
2. le résultat n'est pas celui attendu
3. quelque chose échoue lors de l'exécution du programme **sans que cela lui soit imputable**

2018-11-26

# Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
  - Quid ?
  - Que faire ?
  - Quels outils ?
  - La programmation par Contrat

Programmation Défensive  
En résumé  
C++20 ?  
Principe de Substitution de  
Liskov (LSP)  
NVI

- 4 Questions ?
- 5 Références



Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
  - Quid ?
  - Que faire ?
  - Quels outils ?
  - La programmation par Contrat
- 4 Questions ?
- 5 Références

Programmation Défensive  
En résumé  
C++20 ?  
Principe de Substitution de  
Liskov (LSP)  
NVI

## Programmation par Contrat appliquée au C++

### └ Erreurs de programmation

### └ Plan partiel



# 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...



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 9

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...

2018-11-26

- └ Programmation par Contrat appliquée au C++
  - └ Erreurs de programmation
    - └ Quid ?
      - └ C'est quoi une erreur de programmation ?

# 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*.)



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 10

## 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*.)

2018-11-26

## Programmation par Contrat appliquée au C++

### Erreurs de programmation

#### Que faire ?

#### Que faire en cas d'erreur de programmation ?

1. Par «contexte», comprendre l'état exact de toutes les entités du programme au moment de la détection : variables, mémoire, threads, ...
2. (assert) Ainsi les développeurs peuvent investiguer dans de bonnes conditions.

assert permet en phase de développement et de tests (lorsque le programme n'est pas compilé en mode *Release*, ou plus exactement avec l'option `-DNDEBUG`), de disposer d'un fichier core analysable dans le debuggueur. On peut ainsi demander la valeur de chaque variable au moment du plantage.

En phase de *Release*, le code de cette macro est remplacée par ... rien.

# 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)

## Programmation par Contrat appliquée au C++

### Erreurs de programmation

#### Quels outils ?

#### Quels outils pour s'en protéger ?

- *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.



- *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 => on aura un comportement prévisible et valide. Mais si on ne respecte pas le contrat, tout peut arriver. C'est le pays des *Undefined behaviours*.



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 13

2018-11-26

### Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ La programmation par Contrat
  - └ Programmation par Contrat

Programmation par Contrat  
Principes

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

- 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.



- 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 appliquée au C++

- └ Erreurs de programmation
  - └ La programmation par Contrat
    - └ Programmation par Contrat

2018-11-26

1. Le cas «*j'ai fait tous mes calculs, ils sont faux, et je ne sais pas pourquoi*» ne justifie pas une exception. Il s'agit d'une erreur de programmation ou de logique.
2. Vil Coyote a un plan splendide pour attraper Bip Bip (sa post-condition). Il détourne une route pour la faire arriver au pied d'une falaise, et il peint un tunnel sur le rocher. C'est un algo simple et efficace, Bip Bip devrait s'écraser sur la roche, et Vil aura son repas. Sauf que. Il y a un bug avec la peinture qu'il a intégrée (ou avec Bip Bip) : le volatile emprunte le tunnel. Vous connaissez tous la suite, Vil se lance à sa poursuite et boum. La post-condition n'est pas respectée car il y a un bug totalement inattendu dans les pièces que Vil a intégrées. Il n'y avait ici pas de raison de lancer une exception. La seule exception plausible c'est si Bip Bip venait à ne pas vouloir emprunter cette route.

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 appliquée au C++

- └ Erreurs de programmation
  - └ La programmation par Contrat
    - └ Programmation par Contrat

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).



```
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 aberrant, 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



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 17

### Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ La programmation par Contrat
  - └ Programmation par Contrat

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 aberrant, NaN, ...)
➤ Si i est positif => Mathématicien
➤ Si i est négatif => pas le Mathématicien mais
  • Si interrogeES() a pour post-cond positif => Responsable UI
  • Sinon => Intégrateur
```

Dans les faits, nous sommes en tant qu'intégrateurs, tenus de faire en sorte que ça marche. Quitte à tester la sortie du responsable UI, ou quitte à trouver le domaine de définition exact pour lequel `sqrt` va marcher. Après tout dépendra de comment client+sous-contractant+intégrateur sont liés, des gestes de bonne volonté attendus, etc.

2018-11-26

- 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](https://en.wikipedia.org/wiki/Defensive_programming)) qui mélange vite : l'offensif, le sécurisé, la vérification des entrées, et l'excessivement défensif.



- 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](https://en.wikipedia.org/wiki/Defensive_programming)) qui mélange vite : l'offensif, la sécurité, la vérification des entrées, et l'excessivement défensif.

## Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ Programmation Défensive
    - └ Programmation (excessivement) Défensive

2018-11-26

1. Avec la PpC, on s'intéresse à l'écriture de code correct

# 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 : 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'
```

2018-11-26

Programmation par Contrat appliquée au C++  
└─ Erreurs de programmation  
└─ Programmation Défensive  
└─ PpC offensive ou exceptions ?

# PpC offensive ou exceptions ?

Illustration : défensif baclé

```
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 baclé

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

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'
```

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 20

## Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ Programmation Défensive
    - └ PpC offensive ou exceptions ?

2018-11-26

# 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!");
        return EXIT_SUCCESS;
    } catch (std::exception const& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}
```



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!");
        return EXIT_SUCCESS;
    } catch (std::exception const& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}
```

## Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ Programmation Défensive
    - └ PpC offensive ou exceptions ?

2018-11-26

# 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::cout << "Programme en erreur : " << e.what() << "\n";
    }
}
```



CS Systèmes d'Information - Programmation en C++ appliquée au C++ - 22

2018-11-26

- └ Programmation par Contrat appliquée au C++
  - └ Erreurs de programmation
  - └ Programmation Défensive
  - └ PpC offensive ou exceptions ?

PpC offensive ou exceptions ?  
Illustration : offensif avec validation des entrées

```
namespace my {
// ...
double sqrt(double n) {
    assert(n >= 0 && "sqrt can't process negative numbers");
    return std::sqrt(n);
}
}

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::cout << "Programme en erreur : " << e.what() << "\n";
    }
}
```

# 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 (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 une 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++20).
- Ex : `std::vector::operator[]()`, `std::stack::pop()`



En résumé : comparaison  
Cas de la Programmation par Contrat (offensive)

- On veut traquer et éliminer les erreurs de logique au plus tôt.
- On laisse une 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++20).
- Ex : `std::vector::operator[]()`, `std::stack::pop()`



# C++20 ?

## Du côté des prochains standards

- Sujet apprécié et en discussion depuis un moment *cf.* [Lak14a, Lak14b]
- et en bonne voie *cf.* [DRGL<sup>+</sup>17]



C++20 ?  
Du côté des prochains standards

- Sujet apprécié et en discussion depuis un moment *cf.* [Lak14a, Lak14b]
- et en bonne voie *cf.* [DRGL<sup>+</sup>17]

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 25

2018-11-26

Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ C++20 ?
    - └ C++20 ?

La syntaxe des attributs du C++11 a été étendue

**précondition** avec expects :

**postcondition** avec ensures :

**assertions** avec assert :

```
double sqrt(double x)
[[expects: x >= 0]]
[[ensures ret: abs(ret*ret - x) < epsilon_constant]]
;
```



C++20 ?  
[DRGL<sup>+</sup>17] – syntaxe

La syntaxe des attributs du C++11 a été étendue  
précondition avec expects :  
postcondition avec ensures :  
assertions avec assert :  
double sqrt(double x)  
[[expects: x >= 0]]  
[[ensures ret: abs(ret\*ret - x) < epsilon\_constant]]  
;

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 26

## Programmation par Contrat appliquée au C++

### Erreurs de programmation

#### C++20 ?

#### C++20 ?

2018-11-26

1. Ce qui implique que l'on ne peut pas commencer à saisir des contrats qui sont pour l'instant ignorés par nos compilateurs.
2. Et non, il n'y a rien pour les invariants

## ➤ Modes de vérification

- `default` – coût faible
- `audit` – coût élevé
- `axiom` – pour humains et outils d'analyse statiques

## ➤ Modes de compilation

- `off` – pour les *releases*
- `default` – vérification dynamique pour contrats simples
- `audit` – vérifie tout dynamiquement hors axiomes



C++20 ?  
[DRGL<sup>+</sup>17] – modes

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 27

- Modes de vérification
  - `default` – coût faible
  - `audit` – coût élevé
  - `axiom` – pour humains et outils d'analyse statiques
- Modes de compilation
  - `off` – pour les *releases*
  - `default` – vérification dynamique pour contrats simples
  - `audit` – vérifie tout dynamiquement hors axiomes

## Programmation par Contrat appliquée au C++

### └ Erreurs de programmation

#### └ C++20 ?

#### └ C++20 ?

2018-11-26

# C++20 ?

## [DRGL<sup>+</sup>17] – offensif ou excessif ?

### ➤ *violation handler*

- réglé en offensif à `std::abort` par défaut

### ➤ *continuation option*

- pour reprendre après contrat en échec, ou pour avorter.



C++20 ?  
[DRGL<sup>+</sup>17] – offensif ou excessif ?

- *violation handler*
  - réglé en offensif à `std::abort` par défaut
- *continuation option*
  - pour reprendre après contrat en échec, ou pour avorter.

- ABI stable malgré les modes de compilation
- noexcept quand le *violation handler* est réglé en «programmation excessivement défensive».
- Pas d'invariants
- Pas de relaxation ou de renforcement (LSP)
- Postconditions sur les paramètres sortants

```
void incr(int & r)
[[expects: 0 < r]]
{
    int old = r;
    ++r;
    [[assert: r = old + 1]]; // faking a post-condition
}
```



C++20 ?  
[DRGL<sup>+</sup>17] – difficultés

- ABI stable malgré les modes de compilation
- noexcept quand le violation handler est réglé en «programmation excessivement défensive».
- Pas d'invariants
- Pas de relaxation ou de renforcement (LSP)
- Postconditions sur les paramètres sortants

```
void incr(int & r)
[[expects: 0 < r]]
{
    int old = r;
    ++r;
    [[assert: r = old + 1]]; // faking a post-condition
}
```

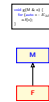
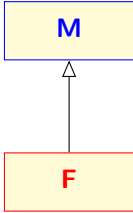
## Programmation par Contrat appliquée au C++

### Erreurs de programmation

#### C++20 ?

#### C++20 ?

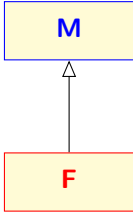
```
void g(M & o) {  
  for (auto x : E_M)  
    o.f(x);  
}
```



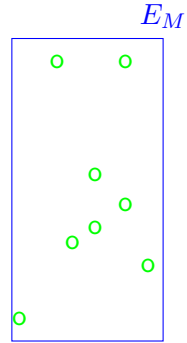
# PpC $\rightarrow$ LSP

## Héritage et préconditions

```
void g(M & o) {
  for (auto x : E_M)
    o.f(x);
}
```



```
void M::f(E x)
[[expects: x ∈ E_M]];
...
g(M{});
```



PpC  $\rightarrow$  LSP  
Héritage et préconditions



Programmation par Contrat appliquée au C++

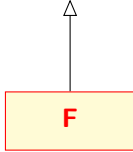
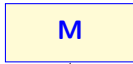
- └ Erreurs de programmation
- └ Principe de Substitution de Liskov (LSP)
  - └ PpC  $\rightarrow$  LSP

1.  $M::f()$  est définie sur  $E_M$
2. Exécuter  $g()$  sur toute instance de M fonctionnera correctement

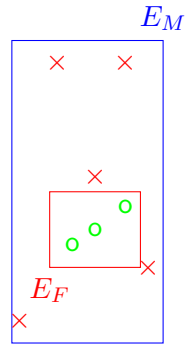
# PpC $\rightarrow$ LSP

## Héritage et préconditions

```
void g(M & o) {
  for (auto x : E_M)
    o.f(x);
}
```



```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```



PpC  $\rightarrow$  LSP  
Héritage et préconditions



## Programmation par Contrat appliquée au C++

- Erreurs de programmation
- Principe de Substitution de Liskov (LSP)
- PpC  $\rightarrow$  LSP

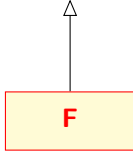
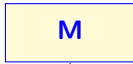
1.  $F::f()$  est définie sur  $E_F$ , plus contrainte
2. Exécuter  $g()$  sur toute instance de  $F$  posera des soucis



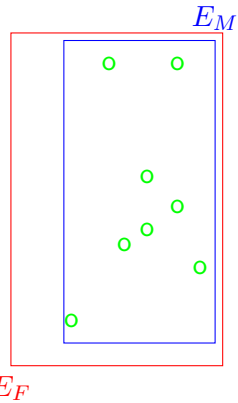
# PpC $\rightarrow$ LSP

## Héritage et préconditions

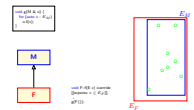
```
void g(M & o) {
  for (auto x : E_M)
    o.f(x);
}
```



```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```



PpC  $\rightarrow$  LSP  
Héritage et préconditions



2018-11-26

Programmation par Contrat appliquée au C++

- Erreurs de programmation
- Principe de Substitution de Liskov (LSP)
- PpC  $\rightarrow$  LSP

1.  $F::f()$  est définie sur  $E_F$ , relaxée
2. Exécuter  $g()$  sur toute instance de F fonctionnera correctement

# PpC $\rightarrow$ LSP

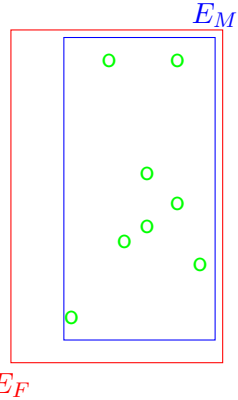
## Héritage et préconditions

```
void g(M & o) {
  for (auto x : E_M)
    o.f(x);
}
```

**M**

**F**

```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```



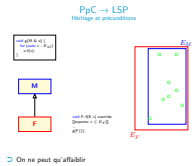
➡ On ne peut qu'affaiblir



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 30

2018-11-26

- Programmation par Contrat appliquée au C++
  - Erreurs de programmation
  - Principe de Substitution de Liskov (LSP)
    - PpC  $\rightarrow$  LSP

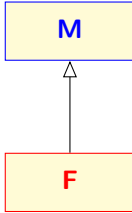


1.  $F::f()$  est définie sur  $E_F$ , relaxée
2. Exécuter  $g()$  sur toute instance de  $F$  fonctionnera correctement

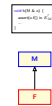
# PpC $\rightarrow$ LSP

## Héritage et postconditions

```
void h(M & o) {
    assert(o.f() in  $E'_M$ );
    ...
}
```



PpC  $\rightarrow$  LSP  
Héritage et postconditions

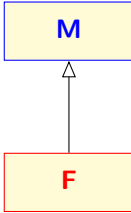


Programmation par Contrat appliquée au C++  
 └ Erreurs de programmation  
 └ Principe de Substitution de Liskov (LSP)  
 └ PpC  $\rightarrow$  LSP

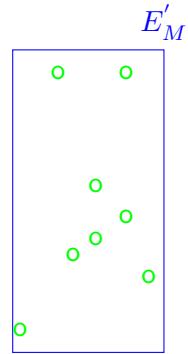
# PpC $\rightarrow$ LSP

## Héritage et postconditions

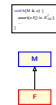
```
void h(M & o) {
  assert(o.f() in  $E'_M$ );
  ...
}
```



$E \ M::f()$   
 $[[\text{ensure } r: r \in E'_M]];$   
 $\dots$   
 $h(M\{\});$



PpC  $\rightarrow$  LSP  
Héritage et postconditions



$E \ M.f()$   
 $[[\text{ensure } r: r \in E'_M]];$   
 $h(M\{\});$



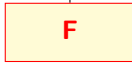
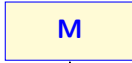
Programmation par Contrat appliquée au C++  
 └ Erreurs de programmation  
 └ Principe de Substitution de Liskov (LSP)  
 └ PpC  $\rightarrow$  LSP

1.  $M::f()$  renvoie un résultat dans  $E'_M$
2. Exécuter  $h()$  sur toute instance de  $M$  fonctionnera correctement

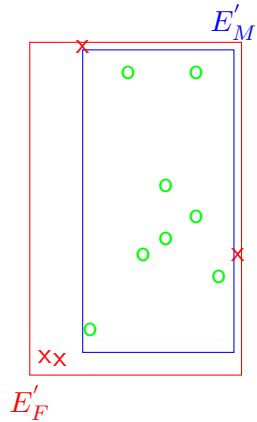
# PpC $\rightarrow$ LSP

## Héritage et postconditions

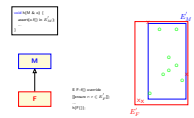
```
void h(M & o) {
  assert(o.f() in  $E'_M$ );
  ...
}
```



E F::f() override  
[[ensure r:  $r \in E'_F$ ]];  
...  
h(F{});



PpC  $\rightarrow$  LSP  
Héritage et postconditions



2018-11-26

Programmation par Contrat appliquée au C++

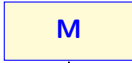
- Erreurs de programmation
- Principe de Substitution de Liskov (LSP)
- PpC  $\rightarrow$  LSP

1. F::f() renvoie un résultat dans  $E'_F$ , plus relaxé
2. Exécuter h() sur toute instance de F posera des soucis

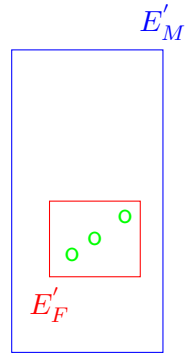
# PpC $\rightarrow$ LSP

## Héritage et postconditions

```
void h(M & o) {
  assert(o.f() in  $E'_M$ );
  ...
}
```



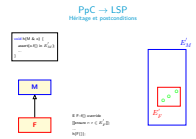
$E \text{ } F::f() \text{ override}$   
 $[[\text{ensure } r: r \in E'_F]];$   
 $\dots$   
 $h(F\{\});$



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 31

2018-11-26

- Programation par Contrat appliquée au C++
  - Erreurs de programmation
  - Principe de Substitution de Liskov (LSP)
    - PpC  $\rightarrow$  LSP

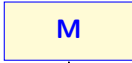


1.  $F::f()$  renvoie un résultat dans  $E'_F$ , plus contrainte
2. Exécuter  $h()$  sur toute instance de  $F$  fonctionnera correctement

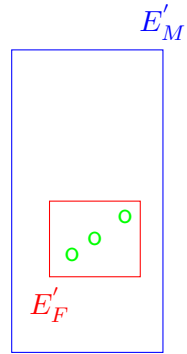
# PpC $\rightarrow$ LSP

## Héritage et postconditions

```
void h(M & o) {
  assert(o.f() in  $E'_M$ );
  ...
}
```



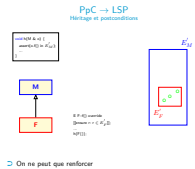
$E \text{ F}::f()$  override  
[[ensure  $r: r \in E'_F$ ];  
...  
 $h(F\{\});$



➔ On ne peut que renforcer

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 31

Programmation par Contrat appliquée au C++  
└ Erreurs de programmation  
└ Principe de Substitution de Liskov (LSP)  
└ PpC  $\rightarrow$  LSP



1.  $F::f()$  renvoie un résultat dans  $E'_F$ , plus contrainte
2. Exécuter  $h()$  sur toute instance de  $F$  fonctionnera correctement

- pré-conditions : elles ne peuvent être qu'affaiblies par les classes dérivées.
- post-condition : elles ne peuvent être que renforcées par les classes dérivées.
- une classe dérivée ne peut qu'ajouter des invariants.



PpC → LSP  
Évolution dans une hiérarchie de classe

- pré-conditions : elles ne peuvent être qu'affaiblies par les classes dérivées.
- post-condition : elles ne peuvent être que renforcées par les classes dérivées.
- une classe dérivée ne peut qu'ajouter des invariants.

### Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ Principe de Substitution de Liskov (LSP)
    - └ PpC → LSP

2018-11-26

1. Une compagnie aérienne a des prérequis sur les bagages en cabine acceptés sans sur-coûts. Pour toutes, on a un prérequis de 50x40x20 cm. Un bagage de 35x20x20 cm sera accepté chez Ryanair. Tandis que les compagnies traditionnelles auront une limite plus lâche sur le poids.
2. À la sortie d'un avion, nous n'avons pas de garantie de ventre rempli gratuitement. Sauf chez les compagnies traditionnelles. Tout nettoyeur de sol nous garantit de retirer la poussière au sol. Pas d'en ajouter.
3. Un rectangle (immuable) a des côtés perpendiculaires. Le carré (immuable) a tous ces côtés de taille identique.



Partout où on attend un objet de type A, on peut passer un objet de type B, si et seulement si B dérive publiquement de A

- EST-SUBSTITUABLE-A : Règle fondamentale pour la construction des hiérarchies de classes (remplace le EST-UN)
- Une ListeTriée N'EST PAS SUBSTITUABLE A une Liste.  
Un Point3d n'est pas un Point2d.  
Un PointColoré n'est pas un Point [Blo08]



Partout où on attend un objet de type A, on peut passer un objet de type B, si et seulement si B dérive publiquement de A

- EST-SUBSTITUABLE-A : Règle fondamentale pour la construction des hiérarchies de classes (remplace le EST-UN)
- Une ListeTriée N'EST PAS SUBSTITUABLE A une Liste.  
Un Point3d n'est pas un Point2d.  
Un PointColoré n'est pas un Point [Blo08]

### Programmation par Contrat appliquée au C++

- └ Erreurs de programmation
  - └ Principe de Substitution de Liskov (LSP)
    - └ PpC → LSP

1. Une liste triée  $sl$  a pour invariant :  $\forall i < j, sl_i < sl_j$   
La fonction `List::push_back(Element e)` a pour post-condition  
`l.last() == e`  
`SortedList sl;`  
`sl.push_back(1);`  
`sl.push_back(10);`  
`sl.push_back(5);` // rompt l'invariant des listes triées
2. `void f(Rectangle & r)`  
`auto a = r.aire();` // supposons le calcul cher  
// utilisation de `a` ....  
`r.largeur *= 2;`  
`a *= 2;`  
// utilisation de `a` ....
3. Je ne vous dis pas la quantité de cours sur l'OO qui montrent la syntaxe de l'héritage en dérivant `PointColoré` de `Point`.

# NVI

## Assurance de la conservation des pré et post conditions

- Fonction virtuelle = point de variation dans l'interface.
- Impossibilité de contrôler la redéfinition dans les classes dérivées.
- Solution : fournir une interface publique non virtuelle et des fonction virtuelles protégées ou privées
  - Améliore robustesse
  - Diminue couplage
- Voir point qualité 3.7.11 classe.NVI



### NVI

Assurance de la conservation des pré et post conditions

- Fonction virtuelle = point de variation dans l'interface.
- Impossibilité de contrôler la redéfinition dans les classes dérivées.
- Solution : fournir une interface publique non virtuelle et des fonction virtuelles protégées ou privées
  - Améliore robustesse
  - Diminue couplage
- Voir point qualité 3.7.11 classe.NVI

## Programmation par Contrat appliquée au C++

### Erreurs de programmation

#### NVI

#### NVI

# NVI Exemple

```
#include <iostream>
#include <cassert>
#include <cmath>
struct Trigo {
    virtual ~Trigo(){}
    /**@pre i < 360
     * @post resultat dans [-1,1]
     */
    double compute(int i) {
        // pre-condition
        assert(i < 360 && "angle trop grand");
        // Point de Variation à spécialiser
        const double res = doCompute(i);
        // post-condition
        assert(res <= 1 && res >= -1.0 && "résultat fct trigo invalide");
        return res;
    }
private :
    virtual double doCompute(int i) =0;
};

class Sin : public Trigo {
    virtual double doCompute(int i)
    { return std::sin(i / 360. * 3.141592653589739); }
};

int main() {
    Sin s;
    std::cout << s.compute(90); // appelle Sin::compute
}
```



CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 35

## Programmation par Contrat appliquée au C++

### Erreurs de programmation

NVI

NVI

2018-11-26

NVI  
Exemple

```
#include <iostream>
#include <cassert>
#include <cmath>
struct Trigo {
    virtual ~Trigo(){}
    /**@pre i < 360
     * @post resultat dans [-1,1]
     */
    double compute(int i) {
        // pre-condition
        assert(i < 360 && "angle trop grand");
        // Point de Variation à spécialiser
        const double res = doCompute(i);
        // post-condition
        assert(res <= 1 && res >= -1.0 && "résultat fct trigo invalide");
        return res;
    }
private :
    virtual double doCompute(int i) =0;
};

class Sin : public Trigo {
    virtual double doCompute(int i)
    { return std::sin(i / 360. * 3.141592653589739); }
};

int main() {
    Sin s;
    std::cout << s.compute(90); // appelle Sin::compute
}
```

# Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références



Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 Références

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 36

Programmation par Contrat appliquée au C++

└ Questions ?

└ Plan partiel

2018-11-26

# Questions ?



Questions ?

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 37

2018-11-26

Programmation par Contrat appliquée au C++

└─ Questions ?

└─ Questions ?

# Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 **Références**



Plan partiel

- 1 Avant-propos
- 2 Besoin de gestion des erreurs
- 3 Erreurs de programmation
- 4 Questions ?
- 5 **Références**

CS Systèmes d'Information - Programmation par Contrat appliquée au C++ - 38





2018-11-26

Programmation par Contrat appliquée au C++

└─Références

└─Plan partiel

# Références I

-  Julien Blanc, *Programmation par contrat, application en c++*, [http://julien-blanc.developpez.com/articles/cpp/Programmation\\_par\\_contrat\\_cplusplus/](http://julien-blanc.developpez.com/articles/cpp/Programmation_par_contrat_cplusplus/), dec 2009.
-  Joshua Bloch, *Effective java*, may 2008.
-  Philippe Dunski and Luc Hermitte, *Coder efficacement - bonnes pratiques et erreurs à éviter (en c++)*, [www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html](http://www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html), fev 2014.
-  Gabriel Dos Reis, J. D. Garcia, John Lakos, Alistair Meredith, Nathan Myers, and Bjarne Stroustrup, <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html>, *SupportforcontractbasedprogramminginC++*, 16 jun 2017.







## Programmation par Contrat appliquée au C++

### └ Références

### └ Références

#### Références I

-  Julien Blanc, *Programmation par contrat, application en c++*, [http://julien-blanc.developpez.com/articles/cpp/Programmation\\_par\\_contrat\\_cplusplus/](http://julien-blanc.developpez.com/articles/cpp/Programmation_par_contrat_cplusplus/), dec 2009.
-  Joshua Bloch, *Effective java*, may 2008.
-  Philippe Dunski and Luc Hermitte, *Coder efficacement - bonnes pratiques et erreurs à éviter (en c++)*, [www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html](http://www.d-booker.fr/programmation-et-langage/157-coder-efficacement.html), fev 2014.
-  Gabriel Dos Reis, J. D. Garcia, John Lakos, Alistair Meredith, Nathan Myers, and Bjarne Stroustrup, <http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html>, *SupportforcontractbasedprogramminginC++*, 16 jun 2017.

## Références II

-  Andrzej Krzemiński, *Préconditions en c++ - partie 1*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-1/>, jan 2013, Traduction.
-  ———, *Préconditions en c++ - partie 2*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-2/>, fev 2013, Traduction.
-  ———, *Préconditions en c++ - partie 3*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-3/>, mar 2013, Traduction.
-  ———, *Préconditions en c++ - partie 4*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-4/>, avr 2013, Traduction.



### Programmation par Contrat appliquée au C++

#### └─Références



#### └─Références

#### Références II

-  Andrzej Krzemiński, *Préconditions en c++ - partie 1*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-1/>, jan 2013, Traduction.
-  ———, *Préconditions en c++ - partie 2*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-2/>, fev 2013, Traduction.
-  ———, *Préconditions en c++ - partie 3*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-3/>, mar 2013, Traduction.
-  ———, *Préconditions en c++ - partie 4*,  
<http://akrzemi1.developpez.com/tutoriels/c++/preconditions/partie-4/>, avr 2013, Traduction.



## Références III

-  John Lakos, *Defensive programming done right - part 1*, [https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube\\_gdata](https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube_gdata), 8 sep 2014.
-  ———, *Defensive programming done right - part 2*, [https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube\\_gdata](https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube_gdata), 8 sep 2014.
-  Bertrand Meyer, *Conception et programmation orientées objet*, <http://www.editions-eyrolles.com/Livre/9782212122701/conception-et-programmation-orientees-objet>, 3 jan 2008.
-  Gregory Pakosz, *Assertions or exceptions ?*, <https://pempek.net/articles/2013/11/16/assertions-or-exceptions/>, nov 2013.



### Références III






-  John Lakos, *Defensive programming done right - part 1*, [https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube\\_gdata](https://www.youtube.com/watch?v=1QhtXRMp3Hg&feature=youtube_gdata), 8 sep 2014.
-  ———, *Defensive programming done right - part 2*, [https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube\\_gdata](https://www.youtube.com/watch?v=tz2khnjnUx8&feature=youtube_gdata), 8 sep 2014.
-  Bertrand Meyer, *Conception et programmation orientées objet*, <http://www.editions-eyrolles.com/Livre/9782212122701/conception-et-programmation-orientees-objet>, 3 jan 2008.
-  Gregory Pakosz, *Assertions or exceptions ?*, <https://pempek.net/articles/2013/11/16/assertions-or-exceptions/>, nov 2013.

## Programmation par Contrat appliquée au C++

### └─Références

### └─Références

## Références IV

-  —, *Cross platform c++ assertion library*,  
<https://pempek.net/articles/2013/11/17/cross-platform-cpp-assertion-library/>, nov 2013.
-  John Regehr, *Use of assertions*,  
<https://blog.regehr.org/archives/1091>, fev 2014.
-  Herb Sutter, *When and how to use exception*,  
<http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836>, jan 2004.
-  Matthew Wilson, *Imperfect c++*, 2004.
-  —, *Contract programming 101*,  
<http://www.artima.com/cppsource/deepspace3.html>, 2006.



### Programmation par Contrat appliquée au C++

#### └ Références

#### └ Références

#### Références IV

-  —, *Cross platform c++ assertion library*,  
<https://pempek.net/articles/2013/11/17/cross-platform-cpp-assertion-library/>, nov 2013.
-  John Regehr, *Use of assertions*,  
<https://blog.regehr.org/archives/1091>, fev 2014.
-  Herb Sutter, *When and how to use exception*,  
<http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836>, jan 2004.
-  Matthew Wilson, *Imperfect c++*, 2004.
-  —, *Contract programming 101*,  
<http://www.artima.com/cppsource/deepspace3.html>, 2006.