la force de l'innovation

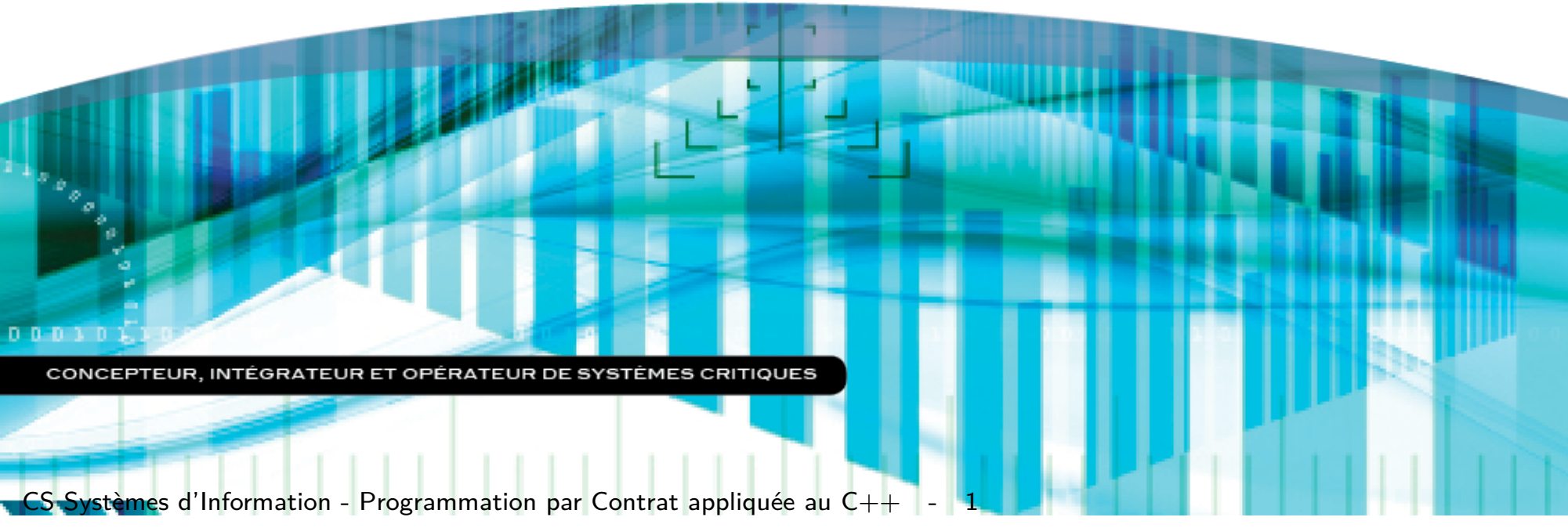
Programmation par Contrat appliquée au C++

Capitole du Libre

Hermitte Luc

CS Systèmes d'Information

17 novembre 2017



Sommaire

Plan partiel

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
-

Avant-propos
Plus d'infos

- Mes billets de blog :
  http ://luchermitte.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

Besoin de gestion des erreurs

Types d'erreurs	Moyens et méthodes	Périmètre
Liées au langage	Compilateur	dev

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

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

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

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...

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 abérrants, 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 abérrants, 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 abérrants, 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 dév & tests), mais ...
  - perte du contexte pour investigation par l'équipe de dév

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 abérrants, 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 dév & tests), mais ...
  - perte du contexte pour investigation par l'équipe de dév
- 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 abérrants, 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 dév & tests), mais ...
  - perte du contexte pour investigation par l'équipe de dév
- 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*.)

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
Définition

- *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.

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*.

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*.

Programmation par Contrat
Post-conditions

- 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
Invariants

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

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

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 symbles incorrects dans le fichier");
        return EXIT_SUCCESS;
    } catch (std::exception const& e) {
        std::cerr << "Error: " << 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 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+17]

C++20 ?
[DRGL+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]]
;
```

C++20 ?
[DRGL+17] – modes

- 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+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.

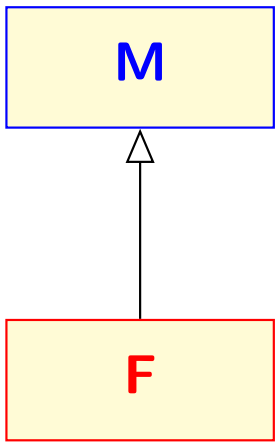
C++20 ?
[DRGL+17] – difficultés

- 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
}
```

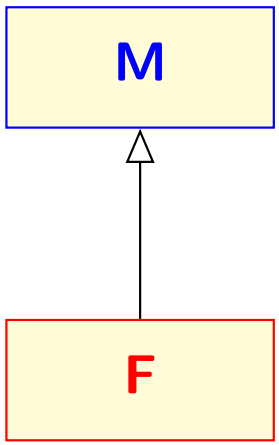
PpC → LSP
Héritage et préconditions

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

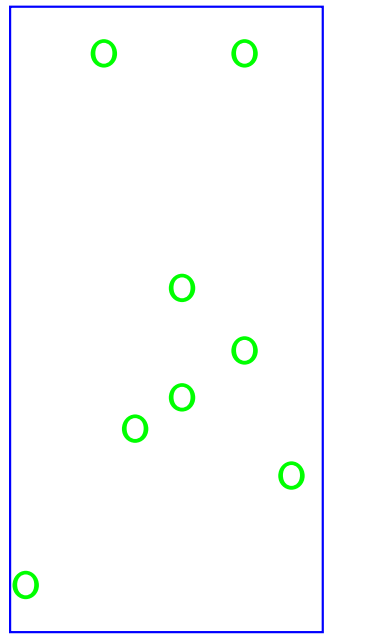


PpC → LSP
Héritage et préconditions

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

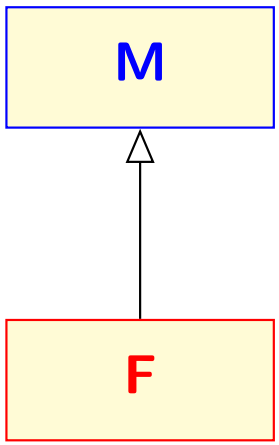



```
void M::f(E x)
[[expects: x ∈ E_M]];
...
g(M{});
```

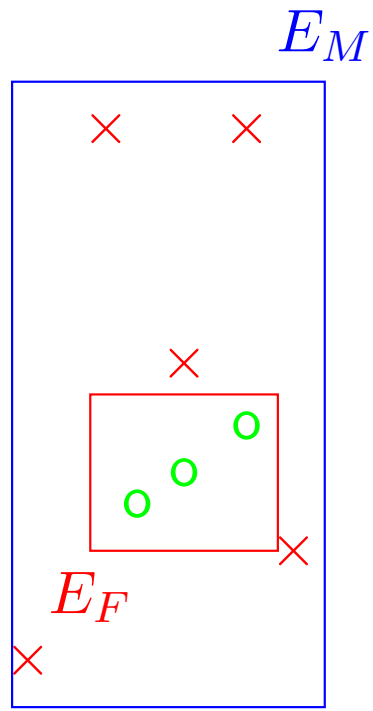


PpC → LSP
Héritage et préconditions

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

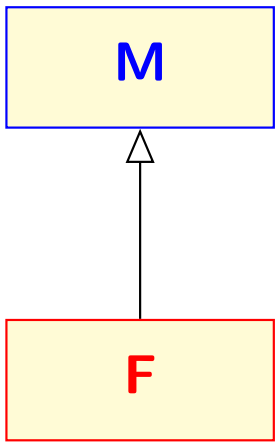



```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```

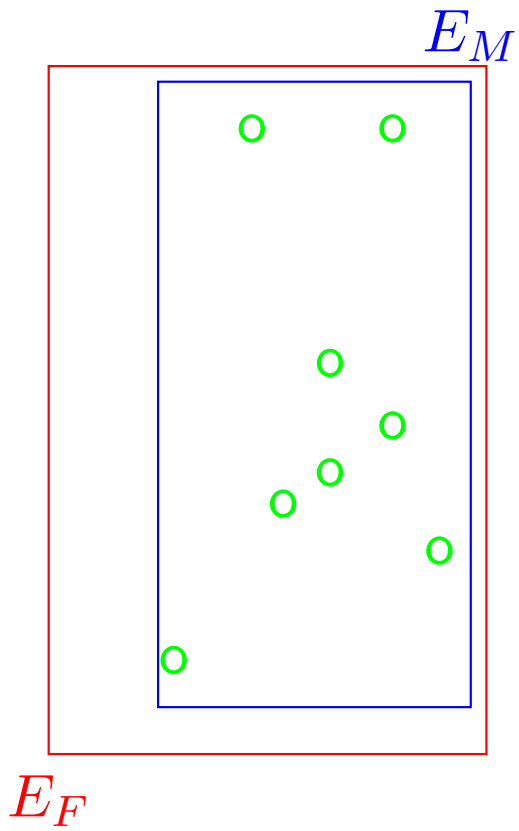


PpC → LSP
Héritage et préconditions

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

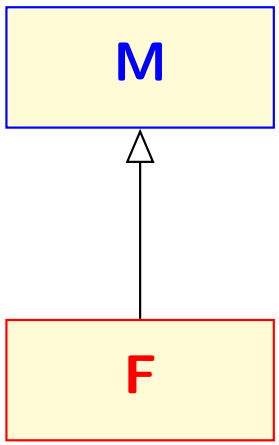



```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```

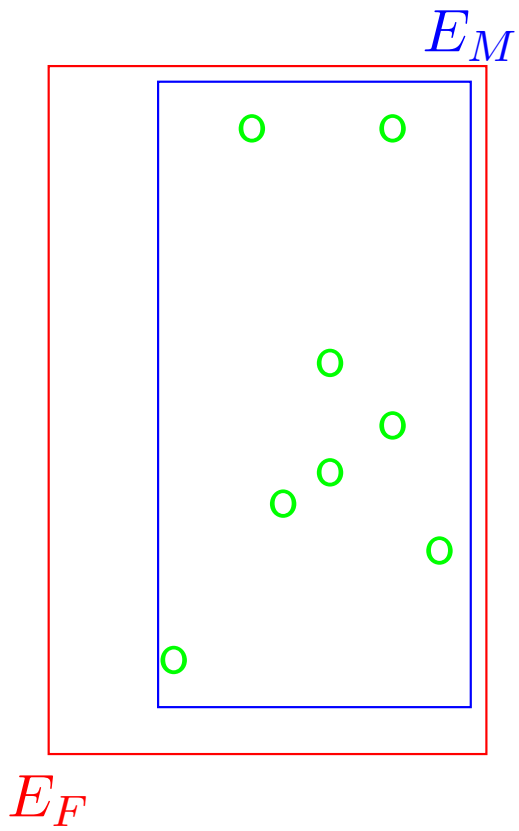


PpC → LSP
Héritage et préconditions

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



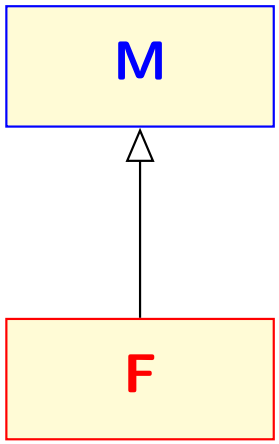

```
void F::f(E x) override
[[expects: x ∈ E_F]];
...
g(F{});
```



- On ne peut qu'affaiblir

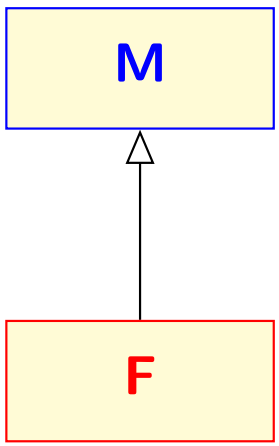
PpC → LSP
Héritage et postconditions

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

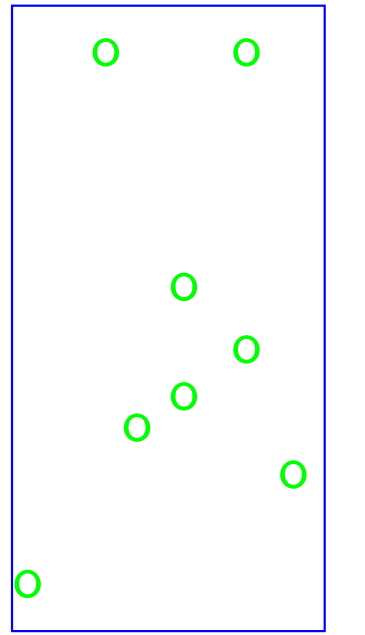


PpC → LSP
Héritage et postconditions

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

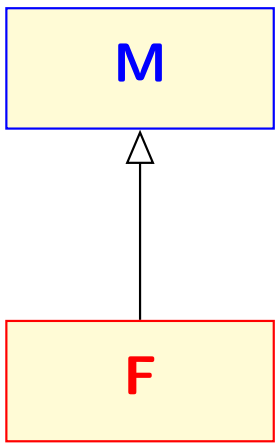


E M::f()
[[ensure r: r ∈ E'_M]];
...
h(M{});

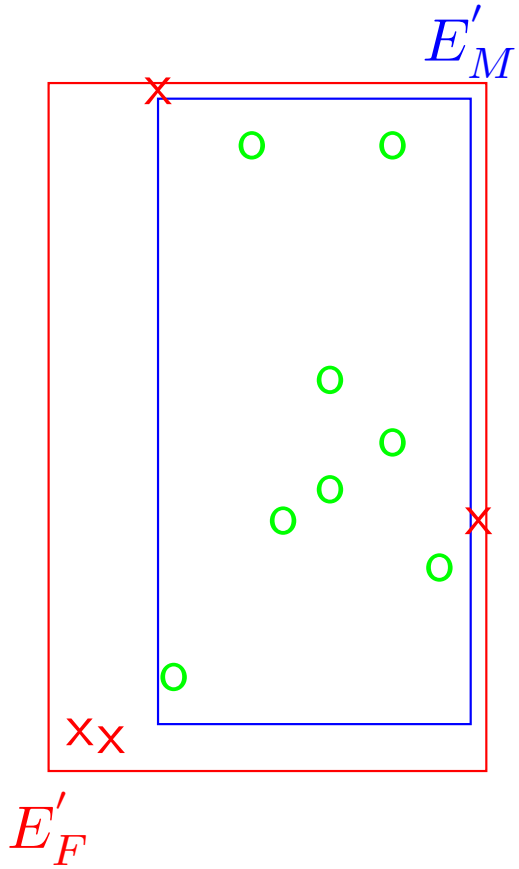


PpC → LSP
Héritage et postconditions

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

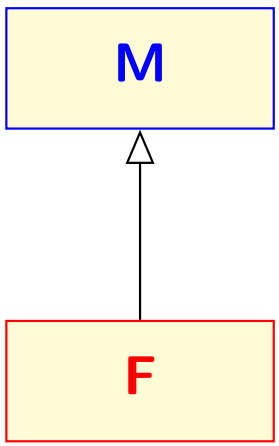


E F::f() override
[[ensure r: r ∈ E'_F]];
...
h(F{});



PpC → LSP
Héritage et postconditions

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

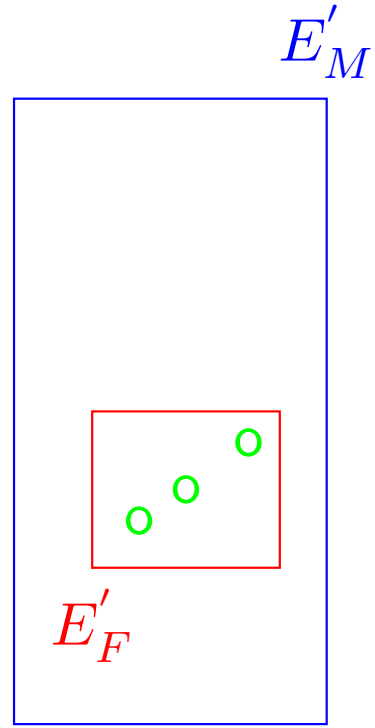


E F::f() override

[[ensure r: r ∈ E'_F]];

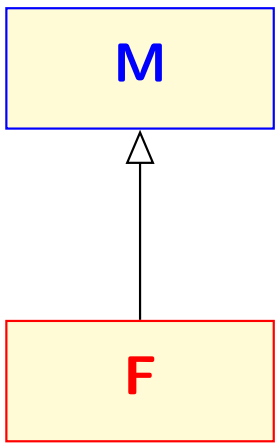
...

h(F{});

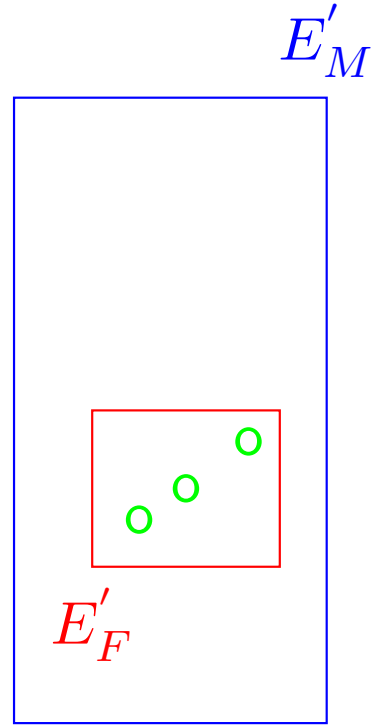


PpC → LSP
Héritage et postconditions

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



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



- On ne peut que renforcer

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.

PpC → LSP

Le Principe de Substitution de Liskov : une règle fondamentale de conception

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 `ListeTriee` N'EST PAS SUBSTITUABLE A une `Liste`.
  Un `Point3d` n'est pas un `Point2d`.
  Un `PointColoré` n'est pas un `Point` [Blo08]

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
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

Questions ?

Plan partiel

Références I


Julien Blanc, *Programmation par contrat, application en c++*, 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, fev 2014.


Gabriel Dos Reis, J. D. Garcia, John Lakos, Alistair Meredith, Nathan Myers, and Bjarne Stroustrup, *http ://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0542r1.html*, SupportforcontractbasedprogramminginC++, 16 jun 2017.

Références II


Andrzej Krzemienski, *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, 8 sep 2014.


_____, *Defensive programming done right - part 2*, 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 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.