

# Examen de règles qualité désuètes dans des langages contemporains, dont le C++

## Capitole du Libre

Hermitte Luc

CS Systèmes d'Information

17 novembre 2018

# Sommaire

- 1 Préambule
- 2 De la déclaration des variables
- 3 Des retours anticipés
- 4 Références

# Plan partiel

- ① Préambule
- ② De la déclaration des variables
- ③ Des retours anticipés
- ④ Références

# Préambule

## Sujet de la présentation

➞ Processus d'évolution d'un référentiel qualité

# Préambule

## Sujet de la présentation

➔ ~~Processus d'évolution d'un référentiel qualité~~

# Préambule

## Sujet de la présentation

- ~~Processus d'évolution d'un référentiel qualité~~
- Dissonance cognitive, effet rebond...

*'If you're arguing, you're losing'*

[SAKS]

# Préambule

## Sujet de la présentation

- ~~Processus d'évolution d'un référentiel qualité~~
- ~~Dissonance cognitive, effet rebond...~~

# Préambule

## Sujet de la présentation

- ~~Processus d'évolution d'un référentiel qualité~~
- ~~Dissonance cognitive, effet rebond...~~
- Discussions techniques



# Plan partiel

## 1 Preamble

Analyse de la définition  
retardée

## 2 De la déclaration des variables

Aspects connexes

Constat

Analyse de l'approche  
historique

## 3 Des retours anticipés

## 4 Références

# Constat

Contradictions entre des référentiels (traditionnels) imposés et la littérature

# Constat

## Persistance des traditions

*'Toute variable doit être déclarée en début de fonction.'*      Référentiels traditionnels 1

*'Toute variable doit être déclarée en début de bloc.'*      Référentiels traditionnels 2

*'In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements.'* [K&R]

*'In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements.'* [K&R]

➔ En vrai, on n'a pas le choix en C89.

*'In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements.'* [K&R]

- En vrai, on n'a pas le choix en C89.
- On trouve des recommandations pour retarder en C99 (SO, forums...)

# Constat

## Clean Code, Robert C. Martin

*'Variables and methods should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope.'*

[MARTIN]

# Constat

## C++ Coding Standards

*'§18. Declare variables as locally as possible.'* [SUTTER et ALEXANDRESCU, p35]

*'§15. Use const proactively.'* [SUTTER et ALEXANDRESCU, p30]



# Constat

## Google C++ Coding Style

*'Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.'* [Google C++ Style Guide]

# Constat

## AUTOSAR, héritier de MISRA C++

*'Rule M3-4-1 (required, implementation, automated) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.'*

[AUTOSAR 2017]

*'AV Rule 136 : Declarations should be at the smallest feasible scope*

*This rule attempts to minimize the number of live variables that must be simultaneously considered.*

*Furthermore, variable declarations should be postponed until enough information is available for full initialization'*

*[JSF-AV 2005]*

# Constat

## High Integrity C++ (as created by Perforce)

*'6.4.1 Postpone variable definitions as long as possible'* [HIC++]

*'7.1.2 Use const whenever possible'* [HIC++]

# Constat

## C++ Core Guidelines

*'Con.1 : By default, make objects immutable.'* [C++CG]

*'Con.4 : Use const to define objects with values that do not change after construction.'* [C++CG]

*'ES.20 : Always initialize an object'* [C++CG]

*'ES.21 : Don't introduce a variable (or constant) before you need to use it'* [C++CG]

*'ES.22 : Don't declare a variable until you have a value to initialize it with'* [C++CG]

*'NR.1 : Don't : All declarations should be at the top of a function'* [C++CG]

# Constat

## Mon interrogation

- Pourquoi cette persistance au vue de l'état de l'art dominant ?
- Quels sont donc les arguments de chacun ?

# Analyse de l'approche historique

## Pourquoi ? Mes soupçons

- Mes soupçons premiers
  - Reproduction des pratiques de nos aïeux

# Analyse de l'approche historique

## Pourquoi ? Mes soupçons

### ➤ Mes soupçons premiers

- Reproduction des pratiques de nos aïeux
- Après tout Pascal, Ada, C89... ne laissent pas le choix



# Analyse de l'approche historique

## Pourquoi ? Mes soupçons

### ➤ Mes soupçons premiers

- Reproduction des pratiques de nos aïeux
- Après tout Pascal, Ada, C89... ne laissent pas le choix
- Incidemment nos cours d'algorithmiques sont ainsi

# Analyse de l'approche historique

## Pourquoi ? Mes soupçons

### ➤ Mes soupçons premiers

- Reproduction des pratiques de nos aïeux
- Après tout Pascal, Ada, C89... ne laissent pas le choix
- Incidemment nos cours d'algorithmiques sont ainsi
- ⇒ habitudes fortement ancrées
  - Je le constate régulièrement avec mes apprenants, collègues...

# Analyse de l'approche historique

## Pourquoi ? Les arguments classiques

- Permet de trouver rapidement toutes les variables au début
- Plus simple pour les initialiser au début et les surveiller dans un débogueur
- Si la liste des variables explose, bon indice qu'il faut refactoriser

# Analyse de l'approche historique

## Pourquoi ? Les arguments classiques

- Permet de trouver rapidement toutes les variables au début
  - Si la fonction était courte (SRP) cela ne devrait pas être un problème
  - Pire, on augmente le nombre de lignes nécessaires
  - Même Vim dispose d'un *goto-definition* (gd)...
- Plus simple pour les initialiser au début et les surveiller dans un débogueur
- Si la liste des variables explose, bon indice qu'il faut refactoriser

# Analyse de l'approche historique

## Pourquoi ? Les arguments classiques

- Permet de trouver rapidement toutes les variables au début
- Plus simple pour les initialiser au début et les surveiller dans un débogueur
  - Je les préfère `const`, aucune surveillance nécessaire
  - Les débogueurs savent s'adapter...
- Si la liste des variables explose, bon indice qu'il faut refactoriser

# Analyse de l'approche historique

## Pourquoi ? Les arguments classiques

- Permet de trouver rapidement toutes les variables au début
- Plus simple pour les initialiser au début et les surveiller dans un débogueur
- Si la liste des variables explose, bon indice qu'il faut refactoriser
  - Ne peut-on pas le voir avant ?
  - Faire un *Extract Method* sans assistance, est l'enfer avec toutes les variables au début.

# Analyse de l'approche historique

## Pourquoi ? Quelques arguments cognitifs ?

- Déclarer et initialiser, au fond, ce sont deux opérations (SRP)
- Un fort *tracing* est signe de complexité dans un code
  - Aller chercher des informations (e.g. variables, doc...) loin augmente ce *tracing*. [ZDS]

# Analyse de la définition retardée

## Apports mineurs

Le grand classique : optimisation

- On ne procède pas à des constructions et initialisation inutiles



# Analyse de la définition retardée

## Apports mineurs

### Le grand classique : optimisation

- On ne procède pas à des constructions et initialisation inutiles
  - Chose partiellement neutralisée par l'utilisation de *Static Single Assignment Form*

```
y := 1
y := 2
x := y
```

```
y1 := 1
y2 := 2
x1 := y2
```

# Analyse de la définition retardée

## Apports mineurs

### Le grand classique : optimisation

- On ne procède pas à des constructions et initialisation inutiles
  - Chose partiellement neutralisée par l'utilisation de *Static Single Assignment Form*

```
y := 1
y := 2
x := y
```

```
y1 := 1
y2 := 2
x1 := y2
```

- Mais qui reste valable pour les constructions en deux temps

```
vector<int> v;
v.resize(42);
fill_n(begin(v), size(v), 2);
```

```
std::vector<int> v(42, 2);
```

# Analyse de la définition retardée

## Apports mineurs

➔ Réduit les risques d'*unused variable* après refactorisation

# Analyse de la définition retardée

## Apports mineurs

- Réduit les risques d'*unused variable* après refactorisation
  - -Wunused-variable y remédie...

# Analyse de la définition retardée

## Apports mineurs

- Réduit les risques d'*unused variable* après refactorisation
- Synchronisation nom  $\leftrightarrow$  contenu plus aisée (car *tracing* réduit)

# Analyse de la définition retardée

## Apports mineurs

- Réduit les risques d'*unused variable* après refactorisation
- Synchronisation nom  $\leftrightarrow$  contenu plus aisée (car *tracing* réduit)
- Plus simple pour refactoriser avec un *Extract Method*

# Analyse de la définition retardée

## Apports mineurs

### ↳ Nécessaire avec auto

- auto ne permet pas de déclarer une variable sans l'initialiser

```
auto variable; // KO!!  
variable = expression;
```

# Analyse de la définition retardée

## Apports mineurs

### ↳ Nécessaire avec auto

- auto ne permet pas de déclarer une variable sans l'initialiser

### ↳ Nécessaire avec const

- const ne permet pas de définir une variable sans l'initialiser définitivement

`const` type variable; // KO!!

variable = expression;



# Analyse de la définition retardée

## Apports mineurs

### ➤ Nécessaire avec `auto`

- `auto` ne permet pas de déclarer une variable sans l'initialiser

### ➤ Nécessaire avec `const`

- `const` ne permet pas de définir une variable sans l'initialiser définitivement
- style fonctionnel

#### ➤ plus simple à reprendre,

*'To lower the cognitive burden of future readers : Yes, there are 10 local variables here, but only 2 of them vary.'*

[GREGORY]

#### ➤ ne nécessite pas d'attention particulière en debug [SAJANIEMI et NAVARRO-PRIETO]

#### ➤ plus *concurrent-friendly*,

# Analyse de la définition retardée

## Apports mineurs

### ➤ Nécessaire avec `auto`

- `auto` ne permet pas de déclarer une variable sans l'initialiser

### ➤ Nécessaire avec `const`

- `const` ne permet pas de définir une variable sans l'initialiser définitivement
- style fonctionnel

#### ➤ plus simple à reprendre,

*'To lower the cognitive burden of future readers : Yes, there are 10 local variables here, but only 2 of them vary.'*

[GREGORY]

#### ➤ ne nécessite pas d'attention particulière en debug [SAJANIEMI et NAVARRO-PRIETO]

#### ➤ plus *concurrent-friendly*,

### ➤ Quelques limitations en présence de branches...

# Analyse de la définition retardée

## Apports majeurs

Après un refactoring, on se retrouve vite avec des situations, où la déclaration retardée...

➤ ... réduit, voire élimine avec `const`, le risque d'utiliser une variable non initialisée...

```
int f(void) {
    int i;
    int result;
    if (h(time(nullptr))) result = g(i);
    // ... du code2 ...
    i = 42;
    // ... du code3 ...
    return result;
}
```

```
$ g++ -c uninitialized.cpp -Wall
```

```
uninitialized.cpp: In function 'int f()':
```

```
uninitialized.cpp:10:39: warning: 'i' may be used uninitialized in this function [-Wmaybe-uninitialized]
    result = g(i);
               ~^~
```

Notez le warning !

# Analyse de la définition retardée

## Apports majeurs

Après un refactoring, on se retrouve vite avec des situations, où la déclaration retardée...

- ➔ ... réduit, voire élimine avec `const`, le risque d'utiliser une variable non initialisée...
- ➔ ... pire d'utiliser une variable dans un état non pertinent
  - car «Toute donnée doit être initialisée»...

```
int f(void) {
    int i = 0; // <-- L'initialisation indésirable...
    int result = -1;
    if (h(time(nullptr))) result = g(i);
    // ... du code2 ...
    i = 42;
    // ... du code3 ...
    return result;
}
```

```
$ g++ -c zero-initialized.cpp -Wall
```

Prog (excessivement) défensive, on perd le warning !

# Analyse de la définition retardée

## Apports majeurs

➔ ...En effet, le compilateur va émettre une erreur si on retarde !

```
int f(void) {
    int result = -1;
    if (h(time(nullptr))) {
        result = g(i);
    }
    // ... du code2 ...
    int i = 42;
    // ... du code3 ...
    return result;
}
```

```
$ g++ -c delayed-declared.cpp -Wall
```

```
delayed-declared.cpp: In function 'int f()':
```

```
delayed-declared.cpp:8:20: error: 'i' was not declared in this scope
```

```
    result = g(i);
                ^
```

```
delayed-declared.cpp:11:9: warning: unused variable 'i' [-Wunused-variable]
```

```
    int i = 42;
        ^
```

# Analyse de la définition retardée

## Apports majeurs

- ...En effet, le compilateur va émettre une erreur si on retarde !
- Nouvelle problématique, humaine :
  - Acceptera-t-on de se faire réprimander par une machine capable de détecter nos erreurs de logiques ?

# Aspects connexes

## Initialisation conditionnelle

- Il existe un mouvement pour une programmation plus déclarative [DEANE]
- $\Rightarrow$  moins de statements, plus déclarations, `const` si possible, etc.
- Solution aux initialisations conditionnelles
  - opérateurs ternaires
  - lambdas

*'ES.28 : Use lambdas for complex initialization, especially of const variables'* [C++CG]

```
string var = [&]{
    if (!in) return ""; // default
    string s;
    for (char c : in >> c)
        s += toupper(c);
    return s;
}(); // note ()
```

Mais... Est-ce bien *simple*? Ce serait un nouvel idiome [GREGORY]

# Plan partiel

- ① Préambule
- ② De la déclaration des variables
- ③ Des retours anticipés
  - Constat
  - Aspects cognitifs

Un peu d'histoire  
Problématique des  
ressources  
Les exceptions  
Quid de la preuve formelle ?

- ④ Références



# Constat

- Contradictions entre divers référentiels.
- On observe une évolution ces dernières années vers les retours multiples

*'JSF-AV 2005 : AV Rule 113 Functions will have a single exit point.'* [JSF-AV 2005]

*'6-6-5 (Required) A function shall have a single point of exit at the end of the function.'* [MISRA-C++ 2008]

# Constat

Autant de retours que l'on veut

*'Single point of exit approach does not necessarily improve readability, maintainability and testability. A function can have multiple points of exit.'* [AUTOSAR 2017]

*'MSC52-CPP. Value-returning functions must return a value from all exit paths'* sous-entendu chez [SEI CERT C++]

*'Don't nest deeply – return early'* [GREGORY]

*'NR.2 : Don't : Have only a single return-statement in a function'* [C++CG], Non rules and myths

*'Use Early Exits and continue to Simplify Code'* [LLVM, §SESE]

## Aspects cognitifs humains

*'I often find I use Replace Nested Conditional with Guard Clauses when I'm working with a programmer who has been taught to have only one entry point and one exit point from a method. One entry point is enforced by modern languages, and one exit point is really not a useful rule. : if the method is clearer with one exit point, use one exit **Clarity is the key principle** point ; otherwise don't.'* [FOWLER, Refactoring]

*'Nested conditional code often is written by programmers **who are taught** to have one exit point from a method. I've found that is a too simplistic rule. When I have no further interest in a method, I signal my lack of interest by getting out. Directing the reader to look at an empty else block only gets in the way of **comprehension**.'* [FOWLER, Refactoring]

## Aspects cognitifs humains

*'Some programmers follow Dijkstra's rules of structured prog. D. said that every function, and every block within a function, should have 1 entry and 1 exit. Following these rules means that there should only be 1 return statement in a function, no break or continue statements in a loop, and never, ever, any goto statements. While we are sympathetic to the goals and disciplines of structured programming, those rules serve little benefit when functions are very small. It is only in larger functions that such rules provide significant benefit. So **if you keep your functions small**, then the occasional multiple return, break, or continue statement does no harm and can sometimes even be more expressive than the SESE rule. On the other hand, goto only makes sense in large functions, so it should be avoided.'*

[MARTIN, Clean Code]

# Aspects cognitifs

## À l'inverse

- Et il est tout autant facile de trouver des personnes qui ont un ressenti inverse !
- Essayons de mettre la passion de côté

# Un peu d'histoire

## Le Single Entry

- Il faut remonter avant la programmation structurée
- quand une routine pouvait connaître plusieurs points d'entrée
- Fortran déprécie ENTRY en 2008

```

SUBROUTINE S(X, Y)
  R = SQRT(X*X + Y*Y)
C ALTERNATE ENTRY USED WHEN R IS ALREADY KNOWN
  ENTRY S2(R)
  ...
  RETURN
END

C USAGE
  CALL S(3,4)
C ALTERNATE USAGE
  CALL S2(5)
  
```

<https://softwareengineering.stackexchange.com/a/118793/11576>

# Un peu d'histoire

## Le Single Exit (destination)

- Historiquement, cela a fait aussi référence à une autre notion
  - des points de reprise alternatifs
- [KNUTH] évoque les travaux de [BOCHMANN]
  - sorties de boucles alternatives
  - mais aussi sortie de fonctions
- Présent en Fortran, déprécié en 90

```

CALL MYSUB(args, *123, *456)
C ... code here for normal return
123 CONTINUE
C ... code here for to handle abnormal exit
456 CONTINUE
C ... ditto for some other condition

SUBROUTINE MYSUB(args, *2, *3)
C *N corresponds to a RETURN statement with given integer value
IF(ENDFIL) RETURN 2
IF(ERROR) RETURN 3
    
```

[http://fortranwiki.org/fortran/show/Modernizing+Old+Fortran#alternate\\_return](http://fortranwiki.org/fortran/show/Modernizing+Old+Fortran#alternate_return)



- On en revient sur le débat initié par [DIJKSTRA]
- ~ goto nuit à la prouvabilité et doit être banni
  - (multiple destinations ou *early return*, c'est le même combat)

# Un peu d'histoire

## Structured Programming

- On en revient sur le débat initié par [DIJKSTRA]
  - ~ goto nuit à la prouvabilité et doit être banni
  - (multiple destinations ou *early return*, c'est le même combat)
- *contredit* par [KNUTH]
  - goto est acceptable tant que sur un organigramme
    - on avance sur la gauche,
    - on remonte sur la droite,
    - et on ne croise jamais de fils.
  - La prouvabilité ne devrait pas être impactée ainsi

# Problématique des ressources

## Problème majeur

⇒ On ne doit pas quitter une portée sans libérer

# Problématique des ressources

## Problème majeur

- On ne doit pas quitter une portée sans libérer
- Dans les langages sans exceptions (e.g. C!)
  - soit on imbrique et indente
    - Ce qui complexifie la compréhensibilité d'un code
    - Pas toujours *DRY* et factorisé
  - soit goto cleanup/goto error
    - ⇒ violation locale de la programmation structurée
    - perte de prouvabilité... ?

# Problématique des ressources

## Problème majeur

- On ne doit pas quitter une portée sans libérer
- Dans les langages sans exceptions (e.g. C!)
  - soit on imbrique et indente
    - Ce qui complexifie la compréhensibilité d'un code
    - Pas toujours *DRY* et factorisé
  - soit goto cleanup/goto error
    - ⇒ violation locale de la programmation structurée
    - perte de prouvabilité... ?
- Dans les langages à exceptions
  - finally, *try-with-resources*, with, using
  - ou RAII en C++ ! RAII -> (p.76)

# Les exceptions

## et les langages modernes

- Pas d'exceptions aux balbutiements de [DIJKSTRA]
- Les exceptions
  - sont des *early exits*
  - qui renvoient vers des destinations multiples

# Les exceptions

## et les langages modernes

- Pas d'exceptions aux balbutiements de [DIJKSTRA]
- Les exceptions
  - sont des *early exits*
  - qui renvoient vers des destinations multiples
  - $\Rightarrow$  double violation du *Single Exit*

# Les exceptions

## Que faire ?

[https://en.wikipedia.org/wiki/Structured\\_programming](https://en.wikipedia.org/wiki/Structured_programming)

- On interdit les exceptions pour respecter le *Single Exit* ?
- On les rattrape pour ne sortir que depuis un seul point ?
- On dit, «*Un seul return, plein d'exceptions*» ?
- Ou on range le *Single Exit* dans les *Non Rules* [C++CG] ?



# Quid de la preuve formelle ?

Les exceptions compliquent tout

*'TL ; DR : Les exceptions font exploser le nombre de chemins globaux, les error-codes font exploser le nombre de chemins locaux. Pour les méthodes fortement automatique, l'explosion globale est très coûteuse, pour les méthodes où l'on spécifie manuellement, l'explosion locale rend la preuve fastidieuse.*

*Dans le cas de la preuve manuelle, les exceptions entraînent malgré tout une petite explosion au niveau local sur les fonctions qui envoient des exceptions et sur les fonctions qui traitent. Et ça demande un travail d'annotation plus lourd.'*

[KSASS'PEUK]

# Conclusion

Des questions ?



# Plan partiel

- 1 Préambule
- 2 De la déclaration des variables
- 3 Des retours anticipés
- 4 Références

# Références I



*AUTOSAR : Guidelines for the use of the C++14 language in critical and safety-related systems.*

[https://www.autosar.org/fileadmin/user\\_upload/standards/adaptive/17-03/AUTOSAR\\_RS\\_CPP14Guidelines.pdf](https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CPP14Guidelines.pdf). AUTOSAR, mar. 2017.



G. V. BOCHMANN. 'Multiple Exits from a Loop Without the GOTO'. In : *Commun. ACM* 16.7 (juil. 1973), p. 443–444. ISSN : 0001-0782.



Bjarne STROUSTRUP et Herb SUTTER, éd. *C++ Core Guidelines*.

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. C++CG, 2015.



## Références II



*SEI CERT C++ Coding Standard : Rules for Developing Safe, Reliable, and Secure Systems.*

<https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682>. CERT, 2016.



**Ben DEANE.** *CppCon 2018 : "Declarative Style in C++".*

[https://github.com/CppCon/CppCon2018/blob/master/Presentations/declarative\\_style\\_in\\_cpp/declarative\\_style\\_in\\_cpp\\_\\_ben\\_deane\\_\\_cppcon\\_2018.pdf](https://github.com/CppCon/CppCon2018/blob/master/Presentations/declarative_style_in_cpp/declarative_style_in_cpp__ben_deane__cppcon_2018.pdf). 2018.



**Edsger W. DIJKSTRA.** 'Go To statement considered harmful'. In : *Comm. ACM* 11.3 (1968). letter to the Editor, p. 147–148.



**Philippe DUNSKI et Luc HERMITTE.** *Coder Efficacement - Bonnes pratiques et erreurs à éviter (en C++)*. 1e. D-Booker, fév. 2014. ISBN : 978-2-8227-0166-2.

## Références III



**Martin FOWLER.** *Refactoring : Improving the Design of Existing Code.* Boston, MA, USA : Addison-Wesley, 1999. ISBN : 0-201-48567-2.



**Google C++ Style Guide.** [https://google.github.io/styleguide/cppguide.html#Local\\_Variables](https://google.github.io/styleguide/cppguide.html#Local_Variables). Google, 2018.



**Kate GREGORY.** *CppCon 2018 : "Simplicity not just for beginners".* [https://github.com/CppCon/CppCon2018/blob/master/Presentations/simplicity\\_not\\_just\\_for\\_beginners/simplicity\\_not\\_just\\_for\\_beginners\\_\\_kate\\_gregory\\_\\_cppcon\\_2018.pdf](https://github.com/CppCon/CppCon2018/blob/master/Presentations/simplicity_not_just_for_beginners/simplicity_not_just_for_beginners__kate_gregory__cppcon_2018.pdf). 2018.



**Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.** <http://www.stroustrup.com/JSF-AV-rules.pdf>. Lockheed Martin Corporation, déc. 2005.



## Références IV



Brian W. KERNIGHAN et Dennis M. RITCHIE. *The C Programming Language*. Sous la dir. de Dennis M. RITCHIE. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN : 0131103709.



Donald E. KNUTH. 'Structured programming with go to statements'. In : *Computing Surveys* 6 (1974), p. 261–301.



KSASS'PEUK. *Impact des exceptions sur la preuve formelle, en très gros*. Sept. 2018. URL : <https://openclassrooms.com/forum/sujet/le-systeme-dexceptions-fondamentalement-bugge?page=1#message-92670306>.



Aaron LAHMAN. *Return-code vs. Exception handling*. <https://ra3s.com/wordpress/dysfunctional-programming/2009/07/15/return-code-vs-exception-handling/>. Fév. 2008.

## Références V



*LLVM Coding Standards.*

<http://llvm.org/docs/CodingStandards.html#use-early-exits-and-continue-to-simplify-code>. LLVM, 2018.



Robert C. MARTIN. *Clean Code : A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Upper Saddle River, NJ : Prentice Hall, 2008. ISBN : 978-0-13235-088-4. URL : <https://www.safaribooksonline.com/library/view/clean-code/9780136083238/>.



*MISRA C++ : Guidelines for the Use of the C++ Language in Critical Systems.* MISRA, 2008.



PERFORCE. *High Integrity C++ Coding Standards*. <https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard>. Oct. 2013.



## Références VI



**Jorma SAJANIEMI et Raquel NAVARRO-PRIETO.** 'An investigation into professional programmers' mental representations of variables'. In : *13th International Workshop on Program Comprehension (IWPC'05)* (2005), p. 55–64.



**Dan SAKS.** *CppCon 2016 : "extern c : Talking to C Programmers about C++"*.  
[https://www.youtube.com/watch?v=D7Sd8A6\\_fYU](https://www.youtube.com/watch?v=D7Sd8A6_fYU). 2016.



**Herb SUTTER et Andrei ALEXANDRESCU.** *C++ Coding Standards : 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN : 0321113586.



**ZDS.** *Lisibilité d'un code source*.  
<https://zestedesavoir.com/articles/4/lisibilite-dun-code-source/>. 2014.

# Première partie I

## Annexe

# Plan partiel

## ⑤ Parenthèse RAI

# Illustration du RAI

## Fuyons !

Au pays magique où les erreurs n'existent pas, on trouve ça :

```
// Code d'exemple d'Aaron Lahman
Notifylcon* CreateNotifylcon()
{
    Notifylcon* icon = new Notifylcon();

    icon->set_text("Blah blah blah");
    icon->set_icon(new Icon(...), GetInfo());
    icon->set_visible(true);

    return icon;
}
```

## Conseil

Vous ne voulez pas de telles horreurs dans vos codes.

N.B. : cf. [LAHMAN] pour ce code et les suivants

# Illustration du RAI

## Essayons mieux

Quand on croit pouvoir s'en sortir avec quelques `try ... catch`.

*// Code d'exemple d'Aaron Lahman*

```
NotifyIcon* CreateNotifyIcon()
{
    NotifyIcon* icon = new NotifyIcon();
    try {
        icon->set_text("Blah blah blah");
        icon->set_visible(true);
        Info info = GetInfo();
        icon->set_icon(new Icon(...), info);
    } catch (...) {
        delete icon; throw;
    }
    return icon;
}
```

Sauf que

Si on rajoute une troisième ressource, ou si un `GetInfo()` pouvant échouer était appelé après le `new Icon, ...`, cela deviendrait vite très très compliqué.

# Illustration du RAIL

## S'il n'y avait pas d'exceptions

Au pays pas magique avec des erreurs remontées sans exceptions :

```
// Code d'exemple d'Aaron Lahman
HRESULT
CreateNotifyIcon(NotifyIcon** ppResult)
{
    NotifyIcon* icon = 0;
    Icon* inner = 0;
    const wchar_t* tmp1 = 0;
    HRESULT hr = S_OK;

    if ( SUCCEEDED(hr) ) {
        icon = new (nothrow) NotifyIcon();
        if ( !icon ) hr = E_OUTOFMEM;
    }

    if ( SUCCEEDED(hr) )
        hr = icon->set_text("Blah");

    if ( SUCCEEDED(hr) ) {
        inner = new (nothrow) Icon(...);
        if ( !inner )
            hr = E_OUTOFMEM;
        else {
            Info info;
            hr = GetInfo( &info );
        }
    }
}
```

```
if ( SUCCEEDED(hr) )
    hr = icon->set_icon(inner, info);
if ( SUCCEEDED(hr) )
    inner = NULL;
}
}
if ( SUCCEEDED(hr) )
    hr = icon->set_visible(true);

if ( SUCCEEDED(hr) ) {
    *ppResult = icon;
    icon = NULL;
} else {
    *ppResult = NULL;
}

cleanup:
if ( inner ) delete inner;
if ( icon ) delete icon;
return hr;
}
```

# Illustration du RAI

## Voici le C++ moderne de 2014

Au pays pas magique avec des erreurs remontées avec exceptions :

```
// Inspiré du code d'exemple d'Aaron Lahman
std::unique_ptr<Notifylcon> CreateNotifylcon()
{
    auto icon = std::make_unique<Notifylcon>();
    icon->set_text("Blah blah blah");

    auto inner = std::make_unique<Icon>(...);
    icon->set_icon(move(inner), GetInfo());
    icon->set_visible(true);

    return icon;
}
```

## Conseil

C'est ce type de code que vous voulez maintenir !  
Il va donc falloir commencer par l'écrire...