RÈGLES DE PROGRAMMATION POUR LE DÉVELOPPEMENT SÉCURISÉ DE LOGICIELS EN LANGAGE C

GUIDE ANSSI

ANSSI-PA-073
16/11/2021

PUBLIC VISÉ :

Développeur Administrateur RSSI DSI Utilisateur

Informations

Attention

Ce document rédigé par l'ANSSI présente les « **Règles de programmation pour le développement sécurisé de logiciels en langage C** ». Il est téléchargeable sur le site `www.ssi.gouv.fr`. Il constitue une production originale de l'ANSSI. Il est à ce titre placé sous le régime de la « Licence ouverte » publiée par la mission Etalab (`www.etalab.gouv.fr`). Il est par conséquent diffusable sans restriction.

Ces recommandations n'ont pas de caractère normatif, elles sont livrées en l'état et adaptées aux menaces au jour de leur publication. Au regard de la diversité des systèmes d'information, l'ANSSI ne peut garantir que ces informations puissent être reprises sans adaptation sur les systèmes d'information cibles. Dans tous les cas, la pertinence de l'implémentation des éléments proposés par l'ANSSI doit être soumise, au préalable, à la validation de l'administrateur du système et/ou des personnes en charge de la sécurité des systèmes d'information.

Évolutions du document :

VERSION	DATE	NATURE DES MODIFICATIONS
1.0	25/05/2020	Version initiale
1.1	29/05/2020	Corrections mineures
1.2	21/07/2020	Précisions sur certaines règles
1.3	16/11/2021	Corrections mineures et ajout d'un chapitre consacré à la compilation

Table des matières

1
Introduction

Le langage C offre une grande liberté aux développeurs. Cependant, il comporte des constructions ambiguës ou risquées qui favorisent l'introduction d'erreurs lors du développement. Le standard du langage C ne spécifie pas l'ensemble des comportements souhaités, et donc certains restent indéfinis ou non spécifiés [1]*. Libre alors aux développeurs de compilateurs, de bibliothèques ou de systèmes d'exploitation de faire leurs propres choix.*

Il est ainsi nécessaire de définir des restrictions quant à l'utilisation du langage C afin d'identifier les différentes constructions risquées ou non portables et d'en limiter voire interdire l'utilisation. Les restrictions définies dans ce guide ont pour but de favoriser la production de logiciels plus sécurisés, plus sûrs, d'une plus grande robustesse et également de favoriser leur portabilité d'un système à un autre, qu'il soit de type PC ou embarqué.

Le présent guide définit un ensemble de règles, de recommandations et de bonnes pratiques dédiées aux développements sécurisés en langage C. *Dans ce présent document, nous nous limitons, à ce jour, aux 2 standards C90* [2] *et C99 qui restent les plus utilisés.*

Quand une règle est directement associée à un standard précis, celui-ci est indiqué clairement pour éviter toute confusion. Sans précision, les deux standards sont concernés.

Règle

Une *règle* doit *toujours* être respectée ; aucune exception n'est tolérée.

Recommandation

Une *recommandation* doit être respectée sauf dans certains cas exceptionnels, ce qui implique une justification claire et précise du développeur. En abrégé, une recommandation est notée « RECO. ».

Ce guide contient également des bonnes pratiques. Il s'agit souvent de points un peu plus subjectifs comme des conventions de codage telles que l'indentation du code par exemple.

1. Ces notions sont définies en page 9.
2. N.B. le standard C90 est également désigné comme C89 par la communauté C.

Bonne pratique

Les *bonnes pratiques* définies dans ce guide sont vivement recommandées mais elles peuvent être remplacées par celles déjà en place dans l'organisation du développeur ou au sein de son équipe de développement si des règles équivalentes existent.

Ce guide a différents objectifs :

- augmenter la sécurité, la qualité et la fiabilité du code source produit, en identifiant les mauvaises pratiques ou les pratiques dangereuses de programmation ;
- faciliter l'analyse du code source lors d'une relecture par un pair ou par des outils d'analyse statique ;
- établir un niveau de confiance dans la sécurité, la fiabilité et la robustesse d'un développement ;
- favoriser la maintenabilité du logiciel mais également l'ajout de fonctionnalités.

L'idée de ce guide n'est pas de réinventer la roue mais de partir de documents existants (guides méthodologiques, références du standard du langage, ...) pour en extraire, modifier et préciser un ensemble de recommandations quant au développement sécurisé pour le langage C. Les documents de référence utilisés sont les suivants :

- MISRA-C : 2012 Guidelines for the use of the C language in critical systems[Misra2012],
- Norme C ANSI 90[AnsiC90],
- Norme C ANSI 99[AnsiC99],
- GCC : Reference Documentation[GccRef],
- CLANG'S Documentation[ClangRef],
- SEI CERT C Coding Standard[Cert],
- ISO 17961 C Secure Coding Rules[IsoSecu],
- CWE MITRE Common Weakness Enumeration[Cwe].

Ce guide ne s'inscrit pas dans un domaine d'application particulier et ne veut pas remplacer les contraintes de développement imposées par tout contexte normatif (domaine automobile, aéronautique, systèmes critiques, etc.). Son but est d'adresser justement les développements en C sécurisés non couverts par ces contraintes normatives.

2
Convention de codage

Avant toute chose, tout projet de développement quel qu'il soit doit suivre une convention de développement claire, précise et documentée. Cette convention de développement doit absolument être connue de tous les développeurs et appliquée de façon systématique.

Chaque développeur a ses habitudes de programmation, de mise en page du code et de nommage des variables. Cependant, lors de la production d'un logiciel, ces différentes habitudes de programmation entre développeurs aboutissent à un ensemble hétérogène de fichiers sources, dont la vérification et la maintenance sont plus difficiles.

RÈGLE 1

RÈGLE — Application de conventions de codage claires et explicites

Des conventions de codage doivent être définies et documentées pour le logiciel à produire. Ces conventions doivent définir au minimum les points suivants : l'encodage des fichiers sources, la mise en page du code et l'indentation, les types standards à utiliser, le nommage (bibliothèques, fichiers, fonctions, types, variables, …), le format de la documentation.
Ces conventions doivent être appliquées par chaque développeur.

Cette règle autour d'une convention de développement est certes évidente et le but ici n'est pas d'imposer, par exemple, un choix de nommage de variable (tel que snake-case versus camel-case) mais de s'assurer qu'un choix a bien été fait au début du projet de développement et que celui-ci est clairement explicité.

L'annexe E donne des exemples de convention de codage qu'il est possible de reprendre ou d'adapter selon les besoins.

[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[IsoSecu] Using Identifiers that are reserved for the implementation [resident].

3
Comportements non définis et non spécifiés

La suite de ce guide faisant fréquemment appel aux notions de comportements non définis (*undefined behaviors*) ou non spécifiés (*unspecified behaviors*), nous les rappelons ci-dessous.

Comportement non défini/Undefined behavior

Un *comportement non défini* est un comportement pour lequel le standard C n'impose rien et qui fait suite à une erreur de construction du programme, à une construction non portable ou à une utilisation de donnée erronée. On peut citer par exemple un dépassement d'entier signé (*signed integer overflow*).

Comportement non spécifié/Unspecified behavior

Un *comportement non spécifié* est un comportement pour lequel le standard C fournit au moins deux comportements alternatifs acceptés mais dont aucun n'est imposé. On peut citer par exemple l'ordre d'évaluation des opérations # et ## pendant la substitution d'une macro.

Information

La liste exhaustive de tous les comportements non spécifiés et de tous les comportements non définis sont disponibles dans les annexes G et J des standards C90[AnsiC90] et C99[AnsiC99].

Ce guide ne considère qu'un environnement de codage C conforme aux standards C90 ou C99.

Information

La programmation concurrente n'est pas traitée dans cette version du guide mais le sera dans une version ultérieure.

RÈGLE 2

RÈGLE – Seul le codage C conforme au standard est autorisé

Aucune violation des contraintes et de la syntaxe C telles que définies dans les standards C90 ou C99 n'est autorisée.

3.1 Références

[Misra2012] Rule 1.1 The program shall contain no violations of the standard C syntax and *constraints* and shall not exceed the implementation translation limits.

[Misra2012] Rule 1.2 Language extensions should not be used.
[Misra2012] Rule 1.3 There shall be no occurrence of undefined or critical unspecified behaviour.
[Cwe] CWE-710 Improper Adherence to coding standard.
[Cwe] CWE-758 Reliance on undefined, unspecified or implementation-defined behavior.

4
Préprocesseur et macros

4.1 Inclusion des fichiers d'en-tête nécessaires

Seuls les fichiers d'en-tête nécessaires doivent être inclus mais certaines règles complémentaires sont à observer. Lorsqu'un fichier d'en-tête inclut lui-même d'autres fichiers d'en-tête, ces déclarations vont être propagées dans tous les fichiers sources ou d'en-tête qui incluent ce premier fichier. Ce mécanisme du langage C aboutit à l'inclusion en cascade de fichiers d'en-tête et de déclarations.

Si l'inclusion des fichiers d'en-tête n'est pas réduite au minimum nécessaire, cela génère des dépendances inutiles, augmente le temps de compilation, et rend l'analyse du code plus complexe par la suite (qu'elle soit manuelle ou outillée). Afin de réduire les dépendances et une propagation inutile de déclarations, les inclusions de fichiers d'en-tête doivent être réalisées dans un fichier « .c » et non pas dans un fichier d'en-tête « .h ». Cependant, dans certains cas, comme typiquement la définition de types, l'inclusion de fichiers d'en-tête de la librairie standard (comme `stddef.h` et `stdint.h`) dans un autre fichier d'en-tête est justifiable.

RECO 3

RECOMMANDATION – Limiter et justifier les inclusions de fichier d'en-tête dans un autre fichier d'en-tête

Les fichiers d'en-tête doivent être inclus au fur et à mesure des besoins présents dans le développement et non « par automatisme » du développeur.

RÈGLE 4

RÈGLE – Seuls les fichiers d'en-tête nécessaires doivent être inclus

De plus, le mécanisme d'inclusion de fichiers d'en-tête peut aboutir à l'inclusion multiple d'un même fichier d'en-tête entraînant au mieux une relecture du code difficile. La définition d'un symbole propre à chaque fichier d'en-tête à l'aide de la directive préprocesseur (`#define`), et la vérification que ce symbole n'a pas déjà été défini (`#ifndef`) permet d'éviter l'inclusion multiple d'un fichier d'en-tête. On parle alors de *macro de garde d'inclusion multiple*. Il faut veiller à définir un symbole unique pour chaque fichier. Le nom de ce symbole peut être construit en reprenant le nom du fichier et en substituant le « . » par « _ ».

RÈGLE 5

RÈGLE — Utiliser des macros de garde d'inclusion multiple d'un fichier

Une macro de garde contre l'inclusion multiple d'un fichier doit être utilisée afin d'empêcher que le contenu d'un fichier d'en-tête soit inclus plus d'une fois :

```
// début du fichier d'en-tête
#ifndef HEADER_H
#define HEADER_H
/* contenu du fichier */
#endif
//fin du fichier d'en-tête
```

Attention

L'utilisation de la directive `#pragma once` est largement répandue mais elle n'appartient pas au standard. Cette solution n'est donc pas reconnue dans ce guide bien que supportée par la plupart des compilateurs. Son utilisation peut poser des problèmes car cette directive est spécifique à chaque compilateur (en particulier en ce qui concerne la gestion de fichiers d'en-tête dupliqués en plusieurs sources physiques ou points de montage).

Enfin, par soucis de lisibilité, la localisation des inclusions de fichiers d'en-tête doit respecter certaines règles précises.

RÈGLE 6

RÈGLE — Les inclusions de fichiers d'en-tête sont groupées en début de fichier

Toutes les inclusions de fichiers d'en-tête doivent être regroupées au début du fichier ou juste après des commentaires ou les directives de préprocesseur, mais systématiquement avant la définition de variables globales ou de fonctions.

RECO 7

RECOMMANDATION — Les inclusions de fichiers d'en-tête systèmes sont effectuées avant les inclusions des fichiers d'en-tête utilisateur

BONNE PRATIQUE 8

BONNE PRATIQUE — Utiliser l'ordre alphabétique dans l'inclusion de chaque type de fichiers d'en-tête

Pour éviter les redondances dans les inclusions de fichiers d'en-tête systèmes ou utilisateur, le développeur peut les ordonner par ordre alphabétique ce qui permet d'avoir un ordre d'inclusion déterministe et de faciliter la revue de code.

Information

Ces trois dernières règle, recommandation et bonne pratique abordent un problème de lisibilité et de maintenabilité et pas directement un problème de sécurité au sens strict du terme mais ces deux premiers aspects restent essentiels pour tout type de développement.

Lorsque l'inclusion d'un fichier d'en-tête est omise, le compilateur peut fournir un avertissement concernant l'utilisation d'une fonction déclarée implicitement.

Information

Les déclarations implicites de fonction sont détectées avec GCC et CLANG via l'option `-Wimplicit-function-declaration`.

Mauvais exemple

Dans le code ci-dessous, l'inclusion de `string.h` est inutile dans `fichier.h` car la seule déclaration utilisée de `string.h` par le `fichier.c` est la fonction `memcpy`. L'inclusion de `string.h` doit donc être déplacée dans `fichier.c` avec une garde d'inclusion multiple dans `fichier.h`.

```
/* header.h */
#include <string.h> /* Ne devrait être inclus que dans fichier.c */

void foo(uint8_t* val, uint32_t length);

/* file.c */
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include "header.h"

#define BUFFER_LEN 8U

void foo(uint8_t *val, uint32_t length) {
  uint8_t buffer[BUFFER_LEN];
  if (NULL != val) {
    memcpy(buffer, val, min(BUFFER_LEN, length));
    ...
  }
}
```

Bon exemple

L'exemple ci-dessous n'inclut dans le fichier d'en-tête que les définitions nécessaires et une garde d'inclusion est présente. Attention cependant lors de l'utilisation de la garde d'inclusion à ne pas utiliser comme nom dans la macro un identifiant déjà réservé ce qui est une erreur classique lors de l'utilisation de la garde d'inclusion.

```
/* header.h */
#ifndef HEADER_H /* garde d'inclusion pour éviter l'inclusion multiple */
#define HEADER_H

void foo(uint8_t *val, uint32_t length);

#endif /*HEADER_H*/

/* file.c */
#include <string.h>
#include <stdint.h>
#include <stdlib.h>
#include "header.h"

#define BUFFER_LEN 8U

void foo(uint8_t *val, uint32_t length) {
  uint8_t buffer[BUFFER_LEN];

  if (NULL != val) {
```

```
    memcpy(buffer, val, min(BUFFER_LEN, length));
    ...
  }
}
```

4.1.1 Références

[Misra2012] Dir. 4.10. Precautions shall be taken in order to prevent the contents of a header file being included more than once.
[Misra2012] Rule 20.1. `#include` directives should only be preceded by preprocessor directives or comments.
[Cert] Rec. PRE06-C Enclose header files in an include guard.

4.2 Non inclusion de fichiers sources

L'inclusion d'un fichier source dans un autre fichier source peut générer des problèmes d'éditions de liens (définitions multiples de variables globales ou de fonctions identiques) ou de duplication de code binaire (dans le cas où les éléments inclus ont été déclarés avec le mot clé `static`). Si un fichier source requiert l'usage de fonctions provenant d'un autre fichier source, il faut déclarer un fichier d'en-tête correspondant et l'inclure dans le fichier source qui en a besoin. Le code doit être séparé dans des modules (fichiers « .c ») indépendants.

Si l'objectif de l'inclusion d'un module au sein d'un autre est de profiter des optimisations inter-procédurales du compilateur (*inlining*, propagation de constantes, etc.), il est possible de faire appel aux fonctionnalités d'optimisation à l'édition des liens (*Link Time Optimization* ou LTO), par exemple avec GCC en utilisant l'option `-flto` lors de la compilation **et** de l'édition des liens : `gcc -o binary -flto file1.c file2.c`.

RÈGLE 9

RÈGLE — Ne pas inclure un fichier source dans un autre fichier source

Seule l'inclusion de fichiers d'en-tête est autorisée dans un fichier source.

Mauvais exemple

Dans l'exemple suivant, une inclusion de fichier source est effectuée, ce qui est justement prohibé.

```
/* file1.c */
#include <stdint.h>

void foo(uint16_t val) {
  ...
}

/* file2.c */
#include "file1.c" /* interdit */

void bar() {
  foo(MAGIC_VALUE);
}
```

Bon exemple

L'exemple ci-dessous décompose correctement le code en différents modules.

```
/* header.h */
#ifndef HEADER_H
#define HEADER_H

void foo(uint16_t val);

#endif

/* file1.c */
void foo(uint16_t val) {
  ...
}

/* file2.c */
#include <stdint.h>
#include "header.h"

#define MAGIC_VALUE 42U

void bar() {
  foo(MAGIC_VALUE);
}
```

4.3 Format d'une directive d'inclusion d'un fichier

Les systèmes de fichiers n'ont pas des comportements identiques : certains systèmes de fichiers sont sensibles à la casse et le séparateur des constituants d'un chemin peut varier.

Lorsqu'un séparateur de chemin spécifique à un système d'exploitation est utilisé, cela empêche la portabilité du code source. Lorsqu'une instruction `#include` comporte un chemin vers le fichier d'en-tête à inclure, le caractère de séparation des constituants du chemin doit être le slash « / », et non l'antislash « \ », afin d'assurer la portabilité des sources. De plus, le caractère « \ » comme les caractères ou séquences de caractères suivants : « ', ", /* et // » situés entre les chevrons ouvrant et fermant (« < » et « > ») ou entre des guillemets (ou *double quotes i.e.* « " ») entraînent un comportement non défini.

Par ailleurs, la casse des noms de répertoire et des noms de fichier doit être respectée.

RÈGLE 10

RÈGLE — Les chemins des fichiers doivent être portables et la casse doit être respectée

Les chemins de fichiers, que ce soit pour une directive d'inclusion `#include` ou non, doivent être portables tout en respectant la casse des répertoires.

Mauvais exemple

Dans l'exemple suivant, la portabilité n'est pas assurée et entraîne un comportement non défini :

```
#include <sys\stat.h>
#include "Module_A\Sub_Module_A\Header.h"
```

Bon exemple

L'exemple ci-dessous utilise un format correct pour inclure les fichiers d'en-tête.

```
#include <sys/stat.h>
#include "module_a/sub_module_a/header.h"
```

De plus, certaines règles précises doivent être respectées dans le nom du fichier d'en-tête pour éviter des comportements non définis.

RÈGLE 11

RÈGLE — Le nom d'un fichier d'en-tête ne doit pas contenir certains caractères ou séquences de caractères

Le nom d'un fichier d'en-tête doit être exempt des caractères et des séquences de caractères suivants : « `'`, `"`, `\`, `/*` et `//` ».

4.3.1 Références

[Misra2012] Rule 20.2 : The ”,” or ”\” characters and the ”/*” and ”//” character sequences shall not occur in a header file name.
[Misra2012] Rule 20.3 : The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.
[AnsiC99] Sections 6.4.7 et 6.10.2.

4.4 Commentaire et définition des blocs préprocesseurs

Les directives de compilation `#if`, `#ifdef`,`#ifndef`, `#else`, `#elif` et `#endif` forment des blocs. Ceux-ci peuvent être longs et ne pas pouvoir être affichés sur un seul écran. Ils peuvent également contenir des blocs imbriqués. Il peut alors être très difficile de déterminer les interdépendances entre les différentes directives. Ces directives doivent donc être commentées avec soin pour expliquer les cas traités et l'enchaînement des différentes directives.

RECO 12

RECOMMANDATION — Les blocs préprocesseurs doivent être commentés

Les directives des blocs préprocesseurs doivent être commentées afin d'expliciter le cas traité et pour les directives « intermédiaires » et « fermantes », celles-ci doivent aussi être associées à la directive « ouvrante » correspondante par le biais d'un commentaire.

Par soucis de lisibilité, la double négation dans les expressions des directives préprocesseurs est à éviter typiquement via l'utilisation de `#ifndef` et un « non mode » (i.e. un mode défini via la négation d'un autre mode comme `NDEBUG`).

BONNE PRATIQUE 13

BONNE PRATIQUE — La double négation dans l'expression des conditions des blocs préprocesseurs doit être évitée

Enfin, il est primordial que toutes les directives associées à un bloc de préprocesseur (*i.e.* les directives « ouvrantes », « intermédiaires » et « fermantes ») soient présentes dans un même fichier. De plus les conditions de contrôle utilisées dans ces directives ne doivent pouvoir être évaluées qu'à 1 ou 0.

RÈGLE 14

RÈGLE — Définition d'un bloc préprocesseur dans un seul et même fichier

Pour un bloc préprocesseur, toutes les directives associées doivent se trouver dans le même fichier.

RECO 15

RECOMMANDATION — Les expressions de contrôle des directives de préprocesseur doivent être bien formées.

Les expressions de contrôle doivent être évaluées uniquement à 0 ou 1 et doivent utiliser uniquement des identifiants définis (via #define).

Mauvais exemple

Le code ci-dessous devrait être modifié afin d'ajouter des commentaires et de ne pas avoir recours à la double négation pour les directives `#else` et `#endif`.

```
/* file1.c */
#ifdef A /*pas de #endif */
#include "log.h"

/* log.h */

#endif /* #endif de file1.c */

#ifndef LOG_H
#define LOG_H

typedef enum {
  DEBUG = 0,
  WARN,
  INFO,
  ERROR,
  FATAL
} LogLevel_T;

void log_msg(LogLevel_T level, const unsigned char* sLogMessage);

#ifndef NDEBUG  /* double négation */
#define LOG_DEBUG(msg)  log_msg(DEBUG, (msg))
#define LOG_WARN(msg)   log_msg(WARN, (msg))
#define LOG_INFO(msg)   log_msg(INFO, (msg))
#define LOG_ERROR(msg)  log_msg(ERROR, (msg))
#define LOG_FATAL(msg)  log_msg(FATAL, (msg))
#else
#define LOG_DEBUG(msg)
#define LOG_WARN(msg)
#define LOG_INFO(msg)
#define LOG_ERROR(msg)
#define LOG_FATAL(msg)
#endif
#endif
```

Bon exemple

Dans l'exemple suivant, les directives sont bien commentées et les directives associées sont dans le même fichier :

```
/* file1.c */
#ifdef A
#include "log.h"
#endif

/* log.h */
#ifndef LOG_H
#define LOG_H

typedef enum {
  DEBUG = 0,
  WARN,
  INFO,
  ERROR,
  FATAL
} LogLevel_T;

void log_msg(LogLevel_T level, const unsigned char* sLogMessage);

#ifdef NDEBUG /* double négation supprimée */
/* Not debug mode */
#define LOG_DEBUG(msg)
#define LOG_WARN(msg)
#define LOG_INFO(msg)
#define LOG_ERROR(msg)
#define LOG_FATAL(msg)
#else /* #ifdef NDEBUG */
/* Debug mode */
#define LOG_DEBUG(msg)  log_msg(DEBUG, (msg))
#define LOG_WARN(msg)   log_msg(WARN, (msg))
#define LOG_INFO(msg)   log_msg(INFO, (msg))
#define LOG_ERROR(msg)  log_msg(ERROR, (msg))
#define LOG_FATAL(msg)  log_msg(FATAL, (msg))
#endif /* #ifdef NDEBUG */
#endif /* #ifndef LOG_H */
```

4.4.1 Références

[Misra2012] Rule 20.8 The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.
[Misra2012] Rule 20.9 All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #defined before evaluation.
[Misra2012] Rule 20.14 All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

4.5 Utilisation des opérateurs de préprocesseur # et

L'ordre d'évaluation associé à plusieurs # (opérateur de *stringification* ou de conversion en chaîne de caractères) ou ## (opérateur de concaténation) ou le mélange de ces deux opérateurs n'est pas spécifié.

RÈGLE 16

RÈGLE — Ne pas utiliser dans une même expression plus d'un des opérateurs de préprocesseur # et

De plus, il est important, pour ces deux opérateurs, de bien comprendre le fonctionnement, *i.e.* les étapes suivies lors de l'expansion d'une macro.

RÈGLE 17

RÈGLE — Utiliser les opérateurs de préprocesseur # et ## en maîtrisant leur expansion

Mauvais exemple

```
#include <stdio.h>
...
#define MYPRINT(s) printf(#s)
#define TWO 2

int main(void)
{
  MYPRINT(TWO); /* impression de « TWO » */
  return 1;
}
```

Bon exemple

```
#include <stdio.h>
...
#define MYPRINT2(s) PRINT(s) /* indirection supplémentaire pour expanser
<<TWO>> */
#define PRINT(s) printf(#s)
#define TWO 2

int main(void)
{
  MYPRINT2(TWO); /* impression de « 2 » */
  return 1;
}
```

4.5.1 Références

[Misra2012] Rule 20.10 The # and ## preprocessor operators should not be used.
[Misra2012] Rule 20.11 A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.
[Misra2012] Rule 20.12 A macro operator used as an operand to the # or ## operators which is itself subject to further macro replacement, shall only be used as an operand to these operators.
[Cert] PRE05-C Understand macro replacement when concatenating tokens or performing stringification.
[AnsiC90] Section 6.8.3.
[AnsiC99] Section 6.10.3.

4.6 Nommage de façon spécifique des macros

Il n'est pas toujours aisé de distinguer dans le code source l'utilisation d'une macro préprocesseur. En effet, l'utilisation de certaines macros peut ressembler à des appels de fonctions.

Par ailleurs, lorsqu'une macro n'est pas nommée en capitales, il y a un risque que le nom corresponde à un véritable nom de fonction voire à un mot réservé du langage C. Cela peut donc aboutir à la substitution d'un appel de fonction par le code remplacé par le préprocesseur voire à un comportement non défini.

RÈGLE 18

RÈGLE — Les macros doivent être nommées de façon spécifique

Pour différencier aisément les macros des fonctions et ne pas utiliser un nom réservé d'une autre macro C, les macros préprocesseurs doivent être en capitales et les mots composant le nom séparés par le caractère souligné « _ » mais sans les faire débuter par le caractère souligné car il s'agit d'une convention pour les noms réservés du langage C.

Mauvais exemple

Dans l'exemple suivant, la règle pour le nommage des macros n'est pas respectée :

```
#define cte 0x7EU /* minuscules */
#define _my_squared(a) ((a)*(a)) /* minuscules et début avec _ */
```

Bon exemple

L'exemple ci-dessous présente un nommage adéquat pour des macros préprocesseurs :

```
#define CTE 0x7EU
#define MY_SQUARED(a) ((a)*(a))
```

4.6.1 Références

[Misra2012] Rule 5.4 Macro identifiers shall be distinct.
[Misra2012] Rule 5.5 Identifiers shall be distinct from macro names.
[Misra2012] Rule 21.1 : A #define or #undef shall not be used on a reserved identifier or reserved macro name.
[Misra2012] Rule 21.2 : A reserved identifier or macro name shall not be declared.
[Misra2012] Rule 20.4 : A macro shall not be defined with the same name as a keyword.
[IsoSecu] Using Identifiers that are reserved for the implementation [resident].
[Cert] Rule DCL37-C Do not declare or define a reserved identifier.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.

4.7 Une macro ne doit pas se terminer par un point-virgule

Les macros sont utilisées pour faciliter la lecture du code et éviter de répéter plusieurs fois le même motif de code. Lors de l'expansion d'une macro, si la définition de celle-ci contient un point-virgule, celui-ci aussi est expansé ce qui peut provoquer un changement complet et inattendu du flot de contrôle.

RÈGLE 19

RÈGLE — Ne pas terminer une macro par un point-virgule

Le point-virgule final doit être omis à la fin de la définition d'une macro.

Mauvais exemple

La macro n'est pas définie en appliquant la règle et se termine par un point-virgule :

```
#define SQUARED(n) (n) = (n) * (n); /* pas de point-virgule à la fin d'une macro */
...
if (x >= 0)
  SQUARED(x); /* conditionnelle sans accolade */
else
  x = -x;
...
```

à l'expansion, on obtient :

```
#define SQUARED(n) (n) = (n) * (n);
...
if (x >= 0)
  x= x * x;
; /* instruction vide */
else  /* erreur de parsing avant le else */
  x = -x;
...
```

Bon exemple

La macro est corrigée :

```
#define SQUARED(n) (n) = (n) * (n)
/* ... */
if (x >= 0)
{
  SQUARED(x);
}
else
{
  x = -x;
}
```

à l'expansion, on obtient :

```
 #define SQUARED(n) (n) = (n) * (n)
/* ... */
if (x >= 0)
{
    x = x * x;
  }
else
{
  x = -x;
}
```

4.7.1 Références

[Cert] Rec. PRE11-C Do not conclude macro definitions with a semicolon.

4.8 Préférer les fonctions `static inline` aux macros de « type fonction »

Les fonctions *inline* sont disponibles depuis la version C99 du langage C.

Information

Comme indiqué dans la section 13.5, les fonctions *inline* doivent également être déclarées comme `static`.

L'utilisation d'une fonction *static inline* pour remplacer ces macros de « type fonction » permet d'éviter les erreurs dans l'ordre d'évaluation des opérateurs lors de l'*inlining* des macros et de faciliter la lecture du code.

Information

Il est important de signaler que les fonctions *static inline* non utilisées peuvent entraîner l'émission d'avertissements à la compilation pour certaines versions de compilateurs. En particulier, certaines versions du compilateur CLANG émettent des warnings dans ce cas, au contraire du compilateur GCC. Ce comportement différent selon les compilateurs survient dès que l'option `-Wunused-function` est activée explicitement à la compilation ou via d'autres options comme `-Wall`. Quand le code associé ne peut vraiment pas être supprimé (bibliothèque par exemple), le développeur pourra être amené à utiliser des extensions du compilateur pour taire ces avertissements mais il faudra commenter et justifier clairement ces ajouts.

RECO 20

RECOMMANDATION – Utiliser des fonctions `static inline` plutôt que des macros à plusieurs instructions

En plus des recommandations et règles précédentes, il est important d'ajouter qu'il ne faut pas utiliser des macros dont l'expansion définit des fonctions dans le code. Le risque d'erreur associé est trop grand et la lisibilité du code ne peut que pâtir de ce genre de pratique.

RÈGLE 21

RÈGLE – L'expansion d'une macro définie par le développeur ne doit pas créer de fonction

4.8.1 Références

[Misra2012] Dir. 4.9 : A function should be used in preference to a *function-like macro* where they are interchangeable.
[Cert] Rec. PRE00-C Prefer inline or static functions to function-like macros.

4.9 Macros multi-instructions

L'utilisation de macros comportant plusieurs instructions peut aboutir à des comportements non souhaités. En effet, lors de la définition d'une macro avec plusieurs instructions, il faut utiliser le caractère « \ » afin d'indiquer au préprocesseur la continuation de la ligne. Cela rend peu lisible la macro définie et peut aussi être source d'erreurs. Le regroupement des instructions au sein d'une boucle `do { ... } while(0)` permet de limiter les comportements non attendus. La boucle `do { ... } while(0)` est toujours exécutée exactement une fois et permet d'éviter de modifier le flot de contrôle de la fonction appelant la macro en regroupant toutes ses instructions au sein d'une boucle.

RÈGLE 22

RÈGLE – Les macros contenant plusieurs instructions doivent utiliser une boucle `do { ... } while(0)` pour leur définition

Mauvais exemple

La macro n'est pas définie en appliquant la règle :

```
#define HALF_SUM(a,b,c,d) \
  (a) = ((c) + (d)) / 2; \
  (b) = ((c) - (d)) / 2
/* conduit à un autre comportement que celui souhaité avec un appel dans une
instruction conditionnelle type if sans accolades  */

  if(c > d)
    HALF_SUM(a, b, c, d);
  else
    /* ... */
```

Même si la macro était définie entre accolades :

```
#define HALF_SUM(a,b,c,d) { (a) = ((c) + (d)) / 2; (b) = ((c) - (d)) / 2}
```

l'expansion de la macro dans la même conditionnelle pose toujours problème à cause de l'absence d'accolades dans la conditionnelle et au « ; » suivant l'appel de la macro :

```
if(c>d)
    { (a)=((c) + (d)) / 2; (b) = ((c) - (d)) / 2};
else
   /* ... */
```

Bon exemple

Dans l'exemple suivant la macro est bien définie à l'aide d'une structure répétitive *do-while(0)* :

```
#define HALF_SUM(a, b, c, d) \
```

```
do {                          \
    (a) = ((c) + (d)) / 2; \
    (b) = ((c) - (d)) / 2; \
} while(0)
```

4.9.1 Références

[Cert] Rec. PRE10-C Wrap multistatement macros in a do-while loop.

4.10 Arguments et paramètres d'une macro

Lors de l'expansion d'une macro par le préprocesseur, des effets de bord inattendus par le développeur peuvent se produire en l'absence de la protection des paramètres de la macro. L'ajout de parenthèses autour des paramètres dans la définition d'une macro doit être systématique.

RÈGLE 23

RÈGLE — Parenthèses obligatoires autour des paramètres utilisés dans le corps d'une macro

Les paramètres d'une macro doivent systématiquement être entourés de parenthèses lors de leur utilisation, afin de préserver l'ordre souhaité d'évaluation des expressions.

De façon générale, il vaut mieux éviter les arguments d'une macro entraînant une opération au sens large. En dehors des effets de bords, même si l'opération effectuée en argument est constante pour une entrée donnée, la performance du code n'est pas optimale.

RECO 24

RECOMMANDATION — Il faut éviter les arguments d'une macro réalisant une opération

Si, de plus, l'opération effectuée par les arguments d'une macro entraîne un effet de bord au sens compilation du terme, cela peut engendrer un comportement inattendu comme des évaluations multiples des arguments de la macro voire aucune évaluation.

RÈGLE 25

RÈGLE — Les arguments d'une macro ne doivent pas contenir d'effets de bord.

Des arguments de macro avec des effets de bord peuvent entraîner des évaluations multiples non désirées.

Enfin, l'utilisation de directives de préprocesseur (`#define`, `#ifdef`...) en arguments d'une macro entraîne un comportement non défini et est donc à proscrire.

RÈGLE 26

RÈGLE — Ne pas utiliser de directives de préprocesseur en arguments d'une macro

Mauvais exemple

Dans l'exemple suivant, le résultat ne sera pas celui attendu lors de l'exécution :

```
#define ABS(x) (x >= 0 ? x  :-x)

a = c + ABS(a - b) + d;
/*  résultat lors de l expansion a = c + (a - b >= 0 ? a - b  :-a -b) + d */

m=ABS(n++);
/* Incrément supplémentaire de n à l'expansion  m=((n++ < 0) ? - n++  :n++) */
```

Bon exemple

Le code ci-dessous définit une macro avec des parenthèses bien placées autour de son argument :

```
#define ABS(x) (((x) >= 0) ? (x)  :-(x))

a = c + ABS(a - b) + d;
/* résultat correct à l'expansion a = c + (((a - b) >= 0) ? (a - b)  :-(a - b)) + d
     */

p=n++;
m=ABS(p);

/* 1 seul incrément  de n,  m=(((p) < 0) ? - (p)  :(p)); */
```

4.10.1 Références

[Misra2012] Rule 20.7 : Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
[Cert] Rec. PRE01-C Use parenthesis within macros around parameters names.
[Cert] Rec. PRE02-C Macro replacement lists should be parenthesized.
[Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.
[Cert] Rule. PRE31-C Avoid side effects in arguments to unsafe macros.
[Cert] Rule. PRE32-C Do not use preprocessor directives in invocations of function-like macros.
[Cert] Rec. PRE12-C Do not define unsafe macros.

4.11 Utilisation de la directive `#undef`

L'utilisation de la directive `#undef` donne fréquemment lieu à des confusions. Par inadvertance, son utilisation peut aboutir à la suppression partielle de code si l'inclusion de ce code est justement contrôlée par le symbole dont la définition est supprimée. Il ne doit jamais être nécessaire de supprimer la définition d'un symbole préprocesseur. Si la suppression du symbole a pour but d'en limiter sa portée, il est préférable de vérifier pourquoi la portée du symbole doit être limitée.

L'utilisation de `#undef` peut provenir du risque de collision du nom choisi pour un symbole préprocesseur. Le nom du symbole doit alors être modifié pour prévenir cette collision.

RÈGLE 27

RÈGLE — La directive `#undef` ne doit pas être utilisée

4.11.1 Références

[Misra2012] Rule 20.5 #undef should not be used.

4.12 Trigraphe et double point d'interrogation

Deux points d'interrogation successifs marquent en C le début d'une séquence associée à un trigraphe. Par exemple, le trigraphe suivant : « `??-` » représente le caractère « ~ ». Tous les trigraphes seront remplacés avant les directives de préprocesseur, et ce quelle que soit la position du trigraphe. Il ne faut donc pas les utiliser.

RÈGLE 28

RÈGLE — Ne pas utiliser de trigraphes

De plus, pour éviter toute confusion avec un trigraphe, tous les commentaires, chaînes de caractères et autres littéraux ne doivent pas contenir deux points d'interrogation successifs.

RECO 29

RECOMMANDATION — Les points d'interrogation ne doivent pas être utilisés de façon successive

Cette recommandation s'applique pour toute partie du code mais aussi pour les commentaires.

Information

L'option `-Wtrigraphs` émet une alerte dès qu'un trigraphe est détecté.

Information

Par défaut, les trigraphes sont désactivés dans GCC.

4.12.1 Références

[Misra2012] Rule 4.2 Trigraphs should not be used.
[Cert] Rec. PRE07-C Avoid using repeated question marks.

5
Compilation

La compilation est une étape importante du développement d'un logiciel puisqu'elle fait le lien entre le code écrit par le développeur et le code qui s'exécutera réellement sur la machine de l'utilisateur final. Il est donc primordial de la maîtriser. De plus, le compilateur est un allié précieux dans la détection d'erreurs de programmation ou d'utilisation dangereuse du langage, et il dispose généralement aussi de fonctionnalités de durcissement capables d'améliorer sensiblement la sécurité du logiciel produit.

Attention

Dans ce guide et *a fortiori* ce chapitre, deux compilateurs sont souvent utilisés à titre d'illustration : GCC [GccRef] et CLANG [ClangRef]. Ce choix s'explique essentiellement par la popularité de ces compilateurs auprès des développeurs et par leur caractère *open-source*. Cela ne signifie en rien que ce guide recommande uniquement l'utilisation de l'un de ces deux compilateurs. Toute alternative peut être proposée mais le développeur devra lui-même transposer les différentes options présentées dans ce guide.

5.1 Maîtrise de l'étape de compilation

Les compilateurs proposent différents niveaux d'avertissements destinés à informer le développeur de l'utilisation de constructions à risques ou de la présence d'erreurs de programmation. Le niveau activé par défaut est en général un niveau peu élevé, qui signale peu de mauvaises pratiques. Il est donc insuffisant et doit être augmenté, ce qui nécessite d'expliciter les options de compilation utilisées. D'autre part, pour une même version du standard C, certains comportements par défaut peuvent varier d'un compilateur à un autre. Même les avertissements émis lors de la compilation sont directement liés à la version du compilateur. Il est donc primordial de connaître exactement le compilateur utilisé, sa version mais aussi toutes les options activées, avec dans l'idéal une justification pour chacune d'elles.

RÈGLE 30

RÈGLE — Définir précisément les options de compilation

Les options utilisées pour la compilation doivent être précisément définies pour l'ensemble des sources d'un logiciel. Ces options doivent notamment établir de façon précise :

- la version du standard C utilisée (par exemple C99 ou C90) ;
- le nom et la version du compilateur utilisé ;
- le niveau d'avertissements (par exemple `-Wextra` pour GCC) ;
- les définitions de symboles préprocesseurs (par exemple définir `NDEBUG` pour une compilation en mode *release*).

De plus, tout développeur activant des options de compilation ou d'édition des liens doit être pleinement conscient des conséquences en matière de sécurité sur l'exécutable ou la bibliothèque générés.

RECO 31

RECOMMANDATION — Maîtriser les actions opérées à la compilation et à l'édition des liens

Le développeur doit connaître et documenter les actions associées aux options de compilation et d'édition des liens activées, y compris lorsqu'elles ont trait à des optimisations de code.

Attention

En particulier, l'utilisation d'options de compilation telles que `-fno-strict-overflow`, `-fwrapv`, `-fwrapv-pointer`, `-fno-delete-null-pointer-checks` ou `-fno-strict-aliasing` est la plupart du temps symptomatique d'une utilisation risquée du langage C.

L'utilisation d'un générateur de projets, tel que *make*, *CMake* ou *Meson* facilite la gestion des options de compilation. Celles-ci peuvent être définies de façon globale et appliquées à tous les fichiers sources à compiler.

BONNE PRATIQUE 32

BONNE PRATIQUE — Utiliser des générateurs de projets pour la compilation.

5.1.1 Références

[Misra2012] Sous-section 4.2. Understanding the compiler.
[Cert] MSC06-C Beware of compiler optimizations.
[Cert] PRE13-C Use the standard predefined macros to test for versions and features.
[Cwe] CWE-14 Compiler Removal of Code to Clear Buffers.

5.2 Compilation sans erreur ni avertissement

S'assurer que le code source compile sans erreur ni avertissement est un excellent moyen de réduire le risque que des erreurs de programmation ou des constructions à risque subsistent dans celui-ci. Évidemment, il ne s'agit pas de baisser le niveau d'exigence des options de compilation pour atteindre cet objectif mais bien de résoudre tous les problèmes remontés par le compilateur. Par défaut, il est attendu d'avoir des options de compilation les plus strictes possibles afin d'augmenter au maximum le niveau d'exigence du compilateur.

RÈGLE 33

RÈGLE — Compiler le code sans erreur ni avertissement en activant des options de compilation exigeantes

Les niveaux élevés des avertissements et erreurs du compilateur et de l'éditeur de liens doivent être activés afin de s'assurer autant que possible de l'absence de problèmes potentiels liés à l'utilisation incorrecte du langage de programmation.
Tous les avertissements et toutes les erreurs signalés par le compilateur et l'éditeur de liens doivent être traités. Il est d'ailleurs fortement conseillé, en cas d'utilisation de GCC ou CLANG, d'utiliser l'option `-Werror` afin de transformer tout avertissement en erreur de compilation et ainsi ne pas risquer de l'ignorer.

La pertinence et la justesse des avertissements émis par le compilateur dépend directement de la précision des analyses menées par celui-ci, lesquelles sont tributaires des optimisations que le compilateur est capable de mener. Il est donc avantageux de spécifier un niveau d'optimisation raisonnablement élevé.

RÈGLE 34

RÈGLE — Activer un niveau d'optimisation raisonnablement élevé

Pour GCC et CLANG, le niveau d'optimisation doit être au moins `-O1`, et idéalement `-O2` ou `-Os`.

Attention

Le développeur doit s'assurer qu'un haut niveau d'optimisation n'élimine pas du code défensif ou des contre-mesures logicielles manuellement ajoutées.

Information

Par exemple, la ligne de commande minimale pour une compilation avec GCC ou CLANG est : `gcc/clang -O1 -Wall`[a] `-Wextra`[b] `-Wpedantic`[c] `-Werror -std=c99/c90`[d] `file.c -o file.exe`

RECO 35

RECOMMANDATION — Utiliser les options de compilation les plus exigeantes

Si une option de compilation apparaît trop stricte pour un développement donné et que le choix est donc fait de la désactiver, une justification devra être fournie pour l'expliquer.

Information

L'annexe B.2 dresse une liste non exhaustive d'avertissements supplémentaires pour GCC et CLANG, qui peut servir de point de départ au développeur.

a. Active tous les avertissements portant sur des constructions à risques du langage et qui sont faciles à éviter. Voir les manuels des compilateurs GCC et CLANG pour la liste complète.
b. Active d'autres avertissements non inclus dans `-Wall`. Voir les manuels des compilateurs GCC et CLANG pour la liste complète.
c. Active tous les avertissements requis par le standard C ; désactive les extensions du compilateur, y compris celles qui ne contredisent pas le standard.
d. Spécifie le standard C utilisé par le compilateur. Voir l'annexe B.1.

Afin de supprimer les erreurs et les avertissements, la première chose à faire est évidemment de corriger le code source, en commentant obligatoirement les modifications effectuées. S'il semble en revanche s'agir d'un faux positif, plusieurs méthodes existent. Tout d'abord, la complexité d'un morceau de code peut parfois suffire à induire en erreur l'analyse du compilateur et il est alors en général bénéfique de simplement réécrire ce morceau dans une forme plus intelligible, tout en s'assurant évidemment qu'elle est sémantiquement équivalente. Ensuite, s'il s'avère qu'un avertissement ne peut pas être éliminé en modifiant le code source, et si le compilateur le permet (via une directive `#pragma` par exemple), celui-ci pourra être désactivé localement.

Attention

La suppression d'avertissements via l'utilisation de `#pragma` peut être très dangereuse et doit donc être parfaitement maîtrisée pour ne pas désactiver, par erreur, un ou plusieurs avertissements sur la totalité du code. De plus, l'utilisation de `#pragma` n'étant pas standard, le développeur doit être conscient que cela est spécifique à chaque compilateur (*implementation-defined*) et donc risqué.

Si le développeur opte pour la suppression d'avertissements, une justification claire doit obligatoirement être fournie à l'aide d'un commentaire :

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-variable"
/* code */
/* des variables ne sont plus utilisées dans le nouvel algorithme  mais elles sont conservées
    pour des raisons de compatibilité API */
/* les avertissements sur ces variables non utilisées sont donc désactivés pour le code suivant
    le pragma */
#pragma GCC diagnostic pop
```

5.2.1 Références

[Misra2012] Dir. 2.1 : All sources files shall compile without any compilation errors.
[Misra2012] Dir. 4.1 : Run-time failures shall be minimized.
[Cert] MSC00-C : Compile cleanly at high warning levels.
[Cwe] CWE-563 Unused variable.
[Cwe] CWE-570 Expression is always false.
[Cwe] CWE-571 Expression is always true.

5.3 Utilisation des fonctionnalités de sécurité des compilateurs

Les compilateurs modernes proposent différentes options permettant d'améliorer la robustesse et le caractère défensif de l'exécutable final. Il peut s'agir de prévenir le risque d'apparition d'une vulnérabilité ou de modérer ses conséquences en matière de sécurité, mais également de durcir le programme contre des tentatives d'exploitation de vulnérabilités.

RÈGLE 36

RÈGLE — Utiliser les fonctionnalités de sécurité offertes par les compilateurs

Les développeurs doivent autant que possible tirer avantage des options de compilation qui permettent d'améliorer la sécurité des logiciels produits.

Attention

Dans la suite de cette section, lorsque des options pour GCC ou CLANG sont données à titre d'exemples, il est nécessaire de garder en tête que :

- ces options sont répertoriées respectivement pour GCC 11 et CLANG 13 ;
- certaines d'entre elles sont déjà activées par défaut, parfois partiellement, selon le compilateur et sa version, mais il est préférable de toujours les spécifier ;
- la justesse des avertissements émis par le compilateur peut dépendre du niveau d'optimisation activé ;
- l'impact sur les performances, s'il est mentionné, est donné à titre indicatif et peut grandement varier en fonction des cas d'usage ; il revient donc au développeur de s'assurer de la conformité à ses besoins.

5.3.1 Avertissements orientés vulnérabilités

Certains avertissements proposés par le compilateur se concentrent tout particulièrement sur la détection de problèmes de sécurité potentiels. Les activer et traiter les alertes remontées constituent donc une première étape dans la réduction du risque de présence de vulnérabilités dans un logiciel.

RÈGLE 37

RÈGLE — Activer et traiter les avertissements orientés vulnérabilités

Par exemple, l'option de compilation `-Wformat=2`[a] de GCC et CLANG doit être activée et les éventuels avertissements alors signalés sont à traiter systématiquement.
Les options `-Wformat-overflow=2` et `-Wformat-truncation=2` de GCC peuvent également être utilisées.

Information

Les compilateurs GCC et CLANG intègrent aujourd'hui des analyseurs statiques de code source capables d'analyser de façon plus poussée et plus fine — mais plus coûteuse — un programme afin de détecter davantage d'erreurs de programmation et en particulier de potentielles vulnérabilités.
Bien que ces analyseurs intégrés demeurent plutôt basiques et finalement encore assez expérimentaux par rapport à de véritables outils d'analyse statique de code source, leur utilisation peut s'avérer pratique et intéressante. Le développeur pourra pour cela se référer à la documentation [GccRef] de l'option `-fanalyzer` ainsi qu'à la section *Clang Static Analyzer* de la documentation [ClangRef].

a. Cette option ajoute des vérifications supplémentaires à la compilation qui portent sur les chaînes de format et les appels de fonctions qui les prennent en arguments.

5.3.2 Instrumentation de certaines fonctions particulièrement dangereuses

L'utilisation de fonctions manipulant la mémoire ou des chaînes de caractères est une grande source d'erreurs de programmation conduisant fréquemment à des vulnérabilités telles que des débordements de tampons. En plus du choix des variantes moins dangereuses de telles fonctions lorsque disponibles (cf. règles de la section 17.4), certains compilateurs sont capables de leur ajouter automatiquement des vérifications simples, effectuées à la compilation ou à l'exécution, pour détecter d'éventuels débordements de tampons.

RÈGLE 38

RÈGLE — Activer l'utilisation de variantes durcies des fonctions dangereuses

Par exemple, en cas de compilation avec GCC d'un programme destiné à un système GNU/Linux muni de la glibc, la macro `_FORTIFY_SOURCE` doit être définie. Le niveau d'optimisation doit être supérieur ou égal à `-O1` pour que les vérifications ajoutées soient effectives.
Les contrôles réalisés à l'exécution terminent le programme lorsqu'un débordement est détecté.

Information

L'option `-Wstringop-overflow=n` de GCC permet d'effectuer des vérifications similaires uniquement à la compilation. Leur sévérité dépend de la valeur de `n`.

5.3.3 Initialisation des variables automatiques

L'utilisation de variables automatiques [3] non initialisées est une autre erreur courante et source de vulnérabilités (voir également la section 6.10). Des compilateurs sont toutefois capables de détecter certaines d'entre elles lorsque les avertissements adéquats sont activés.

RÈGLE 39

RÈGLE — Activer les avertissements du compilateur liés à l'utilisation de variables non initialisées

En particulier, les options `-Wuninitialized`[a], `-Winit-self` et `-Wmaybe-uninitialized`[b] doivent être activées lorsque GCC est utilisé.
Concernant CLANG, les options `-Wuninitialized`[c] et `-Wconditional-uninitialized` doivent être activées.

Attention

Il est important de noter que ces avertissements ne détectent pas tous les cas d'utilisation de variables automatiques non initialisées, notamment lorsque celles-ci sont passées par référence à d'autres fonctions.

3. Une variable automatique est une variable définie au sein d'une fonction, sans le spécificateur de classe de stockage `static`. Son espace de stockage est alloué et désalloué automatiquement sur la pile d'exécution.

a. Automatiquement activée par `-Wall`
b. Automatiquement activée par `-Wall`
c. Automatiquement activée par `-Wall`

En outre, certains compilateurs supportent l'initialisation automatique de ces variables. Cette initialisation forcée peut en pratique être effectuée avec la valeur zéro ou avec une autre valeur particulière, appelée *motif*. L'initialisation automatique à zéro est à privilégier pour les compilations en mode *release* (cf. section suivante 5.4) car elle limite en général l'exploitabilité d'un bogue de ce type. En revanche, pendant les phases de développement, de test et de débogage, l'initialisation avec un motif est préférable car plus susceptible de révéler certains bogues. Le choix du motif est alors important et il peut par exemple s'agir, pour les variables de type pointeur, d'une adresse mémoire non canonique afin de faire systématiquement fauter tout accès mémoire utilisant un pointeur non initialisé.

RÈGLE 40

RÈGLE — Activer l'initialisation forcée des variables automatiques par le compilateur

CLANG supporte l'initialisation automatique avec les deux approches susmentionnées :

- `-ftrivial-auto-var-init=pattern` pour le développement, les tests et le débogage ;
- `-ftrivial-auto-var-init=zero` pour la compilation en mode *release*, option à laquelle il est actuellement nécessaire d'adjoindre l'option `-enable-trivial-auto-var-init-zero-knowing-it-will-be-removed-from-clang`.

Attention

Même en cas d'initialisation forcée par le compilateur de toutes les variables automatiques, leur utilisation sans initialisation par le développeur demeure une erreur de programmation qu'il faut absolument corriger.
L'initialisation automatique par le compilateur constitue en cela un durcissement destiné seulement à limiter l'impact en matière de sécurité de ce type de bogues, et le développeur ne doit donc pas se reposer sur ce comportement.

5.3.4 Débordements d'entiers

Les débordements d'entiers signés ne sont pas définis par le standard C et sont donc particulièrement dangereux. Par exemple, selon les architectures matérielles et les compilateurs, une variable de type `int` atteignant la valeur `INT_MAX` peut boucler (*wrap*) après une nouvelle incrémentation, c'est-à-dire passer à la valeur `INT_MIN`, ce qui peut s'avérer très problématique notamment dans le cas d'une variable représentant un compteur de références à une allocation mémoire. Le compilateur peut être capable de détecter certains débordements d'entiers signés.

RECO 41

RECOMMANDATION — Activer les options du compilateur permettant de détecter les débordements d'entiers signés

GCC et CLANG supportent notamment l'option `-ftrapv`, qui conduit le compilateur à instrumenter le code source afin de générer une exception à l'exécution du programme pour tout débordement d'entiers signés lors d'une addition, d'une soustraction ou d'une multiplication.

Attention

Bien que les débordements d'entiers **non signés** constituent eux un comportement bien défini du C, ils n'en représentent pas moins un aspect périlleux et peuvent également conduire à l'introduction de bogues et de vulnérabilités dans un logiciel. Le développeur doit donc rester particulièrement prudent lorsqu'il effectue des opérations susceptibles de déborder, même avec des opérandes non signés.

Information

De nombreuses autres options utiles à la détection de débordements d'entiers sont supportées par GCC et CLANG mais font partie du *sanitizer* UBSan [a]. L'utilisation de ce dernier ainsi que des autres *sanitizers* n'est pas traitée dans ce guide.

5.3.5 Durcissements de la pile d'exécution

Afin de rendre l'exploitation de certaines vulnérabilités plus difficile, la zone mémoire correspondant à la pile d'exécution d'un programme ne doit pas être exécutable. Les chaînes de compilation modernes s'efforcent en général par défaut à faire respecter cette règle.

Toutefois, l'utilisation de fonctions imbriquées, supportées par GCC, impose une pile exécutable [4] et est donc proscrite [5]. De plus, le support de cette fonctionnalité a complexifié la façon dont les éditeurs de liens GNU — c'est-à-dire BFD et gold — choisissent de marquer un exécutable ou une bibliothèque comme ayant besoin ou non d'une pile exécutable ; il faut donc être très prudent lors de leur utilisation.

RÈGLE 42

RÈGLE — Ne pas utiliser de pile exécutable

En particulier, les fonctions imbriquées de GCC ne doivent pas être utilisées. De plus, pour un logiciel destiné à un environnement GNU/Linux ou FreeBSD :

- l'option `-z execstack` des éditeurs de liens BFD, gold et lld ne doit **pas** être utilisée ;
- l'option `-z nognustack` de l'éditeur de liens lld ne doit **pas** être utilisée ;
- l'option `-z noexecstack` des éditeurs de liens BFD et gold **doit** être utilisée.

Les débordements de tampons situés sur la pile d'exécution sont certainement parmi les corruptions de mémoire les plus anciennes et les plus classiques. Le positionnement de variables de garde à valeur aléatoire, communément appelées *canaris* [6], permet entre autres de détecter certaines tentatives de débordements linéaires cherchant à modifier la valeur d'une adresse de retour sauvegardée sur la pile. Le cas échéant, l'exécution du programme est alors automatiquement terminée.

a. UndefinedBehaviorSanitizer

4. Plus précisément, c'est l'utilisation de fermetures (*closures*), implémentées par GCC au moyen de ces fonctions imbriquées et de trampolines positionnés sur la pile d'exécution, qui est problématique.

5. Il s'agit de toute manière d'une extension du langage, proposée par un compilateur, or pour rappel seul le C conforme aux standards C90 ou C99 est autorisé par le présent guide.

6. On trouve parfois aussi le nom *cookie*.

RÈGLE 43

RÈGLE — Activer les canaris

Avec GCC et CLANG, l'option de compilation `-fstack-protector-strong` doit être activée.
La section 19.2 aborde en outre la mise en œuvre manuelle d'un mécanisme de canaris.

Selon les architectures matérielles, les plateformes et les compilateurs, il est également possible d'utiliser une variable de garde dont la valeur est différente pour chaque fil d'exécution au sein d'un même processus. Cela permet de réduire la gravité d'une éventuelle fuite de la valeur d'un canari.

RECO 44

RECOMMANDATION — Utiliser des canaris à valeurs locales

En particulier, pour les architectures x86, GCC et CLANG proposent l'option `-mstack-protector-guard=tls`, en s'appuyant sur le *Thread Local Storage* de la glibc.

5.3.6 Chargement dynamique

La randomisation est une tactique de défense stochastique visant à réduire la fiabilité de l'exploitation d'une vulnérabilité logicielle. Les chaînes de compilation courantes permettent notamment de produire des exécutables pouvant être chargés à une adresse aléatoire, afin d'utiliser au mieux la distribution aléatoire de l'espace d'adressage (ASLR [7]) implémentée par le système d'exploitation.

RÈGLE 45

RÈGLE — Produire des exécutables relocalisables

Avec GCC comme avec CLANG, l'option de compilation `-fPIE` doit être activée, de même que l'option `-pie` des éditeurs de liens BFD, gold et lld.

Pour permettre la relocalisation de bibliothèques partagées et d'exécutables, le chargeur dynamique doit être en mesure de modifier certaines de leurs sections. Les zones mémoire correspondantes, si elles demeurent ensuite réinscriptibles pendant l'exécution du programme, peuvent être utiles à un attaquant tentant d'exploiter une vulnérabilité logicielle. Il est cependant possible au moment de l'édition des liens de marquer les sections en question de façon à ce que le chargeur dynamique passe en lecture seule les zones mémoire associées dès que possible. On parle alors de mode `relro` ou *partial* `relro`.

RÈGLE 46

RÈGLE — Utiliser le mode `relro` de l'éditeur de liens

Par exemple avec les éditeurs de liens BFD et gold, l'option `-z relro` doit être utilisée. lld applique par défaut le mode `relro` et il n'y a donc pas d'option supplémentaire à lui fournir.

Néanmoins, puisque la résolution des symboles de fonctions s'effectue généralement au fur et à mesure de l'exécution d'un programme (*lazy binding*), certaines sections au sein de bibliothèques partagées ou d'exécutables restent réinscriptibles en dépit du mode `relro`. Il est alors possible de

7. Address space layout randomization

forcer le chargeur dynamique à résoudre l'intégralité des symboles au lancement du programme [8] afin qu'il puisse ensuite passer en lecture seule les zones mémoire associées. On parle alors de mode *full* `relro` ou `BIND_NOW`.

RECO 47

RECOMMANDATION — Ne pas utiliser le lazy binding

Par exemple, l'option `-z now` des éditeurs de liens BFD, gold et lld permet de marquer les exécutables et bibliothèques produits de façon à indiquer au chargeur dynamique qu'il doit résoudre tous les symboles au lancement du programme.

Information

Lorsque le mode `relro` est utilisé et que le *lazy binding* est désactivé, certains éditeurs de liens réordonnent les sections des binaires produits afin d'éviter que des débordements de données situées dans une section écrasent le contenu de sections sensibles.
C'est notamment le cas avec BFD, gold et lld.

5.3.7 Reproductibilité des binaires

La reproductibilité des binaires permet entre autres aux utilisateurs d'un logiciel de vérifier de façon indépendante qu'un binaire qui leur est fourni est bien le produit non altéré d'un état précis de ses sources, en recompilant celles-ci et en comparant le résultat obtenu. Cela implique d'avoir un processus de compilation entièrement déterministe. Il s'agit d'un sujet non trivial qui dépasse le cadre de ce guide, mais la prise en compte le plus tôt possible de cette problématique au sein d'un projet peut grandement faciliter la mise en place de cette fonctionnalité le jour où elle devient un objectif.

BONNE PRATIQUE 48

BONNE PRATIQUE — Assurer la reproductibilité des binaires

Par exemple, des options de compilation comme `-Wdate-time` de GCC et CLANG ou `-frandom-seed=` de GCC permettent de limiter l'introduction de non-déterminisme à la compilation.

5.4 Modes debug et release

Les deux modes de compilation *debug* et *release* sont normalement disponibles dans tous les compilateurs et sont très utiles au développement d'un logiciel grâce aux changements significatifs qu'ils peuvent induire sur le résultat de la compilation.

Mode debug et mode release

En mode *debug*, destiné principalement au débogage lors du développement, la plupart des optimisations sont désactivées et les informations symboliques sont conservées, facilitant notamment l'ajout de points d'arrêt. La compilation est plus rapide et utilise moins de mémoire, mais le code généré sera en général plus lent à l'exé-

8. La désactivation du *lazy binding* peut donc ralentir **le démarrage** d'un logiciel de taille importante. À noter également que cela devient en pratique peu gênant dans le cas d'un *daemon*.

cution et de taille plus importante. En mode *release*, correspondant au mode final approprié à la livraison au client ou à la mise en production, les optimisations sont activées et les informations non utiles à l'exécution comme les symboles sont éliminées. La compilation est alors plus longue et plus gourmande en mémoire, mais permet de générer du code machine plus complexe qui sera ainsi plus compact et plus rapide.

Le mode *debug* permet de mieux comprendre le fonctionnement d'un programme et de corriger les erreurs trouvées tandis que le mode *release* est nécessaire en livraison pour des raisons de performances ou de taille de programme. Par exemple, en mode *debug*, certains compilateurs font en sorte que toutes les variables soient automatiquement initialisées à `0`, alors qu'en mode *release* seules les variables globales le sont, conformément au standard.

Information

Si la directive `NDEBUG` est définie à l'inclusion du fichier d'en-tête `assert.h` de la bibliothèque standard, alors la macro `assert` est redéfinie pour être désactivée.

RÈGLE 49

RÈGLE — Tout code mis en production doit être compilé en mode release

La compilation en mode *release* est obligatoire pour une mise en production.

Cela peut paraître redondant avec les règles et recommandations des sections 5.1 et 5.2 mais il s'agit d'une erreur assez courante en génie logiciel. En plus des comportements différents au niveau de la gestion mémoire et des optimisations du code, le mode *debug* peut parfois même augmenter la surface d'attaque d'un logiciel. Il est donc très important que le développeur utilise ces modes en toute connaissance de cause.

RECO 50

RECOMMANDATION — Prêter une attention particulière aux modes debug et release lors de la compilation

L'utilisation des modes *debug* et *release* à la compilation doit se faire en s'assurant que les modifications induites en matière de gestion de la mémoire et d'optimisation sont bien connues. Les différences entre ces deux modes doivent être documentées de façon exhaustive.

6
Déclaration, définition et initialisation

Définition versus utilisation de variable

La définition d'une variable correspond à lui affecter une valeur (*i.e.* écrire à l'adresse mémoire de la variable) et l'utilisation d'une variable correspond à utiliser la valeur de la variable (*i.e.* lire la valeur stockée à l'adresse mémoire associée).

6.1 Déclarations multiples de variables

Le langage C autorise la déclaration de plusieurs variables d'un même type simultanément en séparant chaque variable par une virgule. Cela permet d'associer à un groupe de variables un type donné et de regrouper ensemble des variables dont le rôle est lié. Cependant ce type de déclaration multiple ne doit être utilisé que sur des variables simples (pas de pointeur ou de variable structurée) et de même type.

RECO 51

RECOMMANDATION — Seules les déclarations multiples de variables simples de même type sont autorisées

Pour éviter également toute erreur dans l'initialisation de variables, les initialisations couplées à une déclaration multiple sont à proscrire. En effet, en cas d'initialisation unique à la fin de la déclaration multiple, seule la dernière variable déclarée est effectivement initialisée.

RÈGLE 52

RÈGLE — Ne pas faire de déclaration multiple de variables associée à une initialisation.

Les initialisations associées (*i.e.* consécutives et dans une même instruction) à une déclaration multiple sont interdites.

Mauvais exemple

Dans le code ci-dessous, plusieurs variables sont déclarées dans une même instruction mais seule la dernière variable est initialisée :

```
uint32_t  abs, ord = 0 ; /* attention la variable abs n'est pas initialisée à
zéro ici !*/
uint32_t  a, *b; /* à proscrire  :mélange déclaration variable simple et pointeur
     */
struct blob_t g, h[35]; /* à proscrire  :mélange déclaration variable simple,
    pointeur et tableau */
```

Bon exemple

```
uint32_t  a,
uint32_t *b; /* séparation de la variable simple et du pointeur */
struct blob_t g;
struct blob_t  h[35]; /* idem séparation tableau et variable simple */
uint32_t  abs, ord ; /* déclaration commune de deux variables liées
    fonctionnellement */
abs = 0; /* affectation des deux variables */
ord = 0;
```

6.1.1 Références

[Cert] Rec. DCL04-C Do not declare more than one variable per declaration.

6.2 Déclaration libre de variables

Depuis C99, la déclaration de variables peut se faire partout dans le code. Cette fonctionnalité semble pratique mais un abus peut complexifier de façon notable la lecture du code et peut entraîner de possibles redéfinitions de variables.

RECO 53

RECOMMANDATION — Regrouper les déclarations de variables en début du bloc dans lequel elles sont utilisées

Pour des questions de lisibilité et pour éviter les redéfinitions, les déclarations de variables sont regroupées en début de fichier, fonction ou bloc d'instructions selon leur portée.

Information

Cette recommandation n'est pas directement liée au sens strict à la sécurité, mais elle influence la lisibilité, la portabilité et la maintenabilité du code, concernant ainsi tout type de développement.
L'option de compilation `-Wdeclaration-after-statement` de GCC et CLANG peut aider au respect de cette recommandation.

Une pratique très courante pour les compteurs de boucle est de les déclarer directement dans la boucle associée. Ce cas de déclaration « libre » est accepté mais il faut s'assurer que la variable associée à ce compteur de boucle ne masque pas une des autres variables utilisées dans le corps de la boucle.

Mauvais exemple

Dans le code suivant, les variables sont déclarées « au fil de l'eau » et non de façon groupée et structurée. Ce genre de pratique complexifie l'identification de toutes les variables déclarées augmentant par conséquent le risque de masquage des variables.

```
#include <stdint.h>
uint8_t glob_var; /* variable globale */
uint8_t fonc(void)
```

```
{
    uint8_t var1; /* variable locale */
    if (glob_var >=0)
    {
        /* ... */
    }
    else
        var1=glob_var;
    uint8_t var2; /* autre variable locale déclarée au milieu d'un bloc */
    /* ... */
}
uint8_t glob_var2; /* autre variable globale déclarée entre deux fonctions */
void main(void)
{
    uint8_t x = fonc();
    /* ... */
}
```

Bon exemple

Les variables sont déclarées de façon groupée et structurée en début des blocs ce qui facilite la lecture.

```
#include <stdint.h>

uint8_t glob_var; /* variables globales déclarées ensemble*/
uint8_t glob_var2;

uint8_t fonc(void)
{
    uint8_t var1; /* variables locales déclarées ensemble au début de la
    fonction */
    uint8_t var2;
    if (glob_var >= 0)
    {
        /* ... */
    }
    else
    {
        var1 = glob_var;
    }
    /* ... */
}
void main(void)
{
    uint8_t x = fonc();
    /* ... */
}
```

6.3 Déclaration des constantes

L'utilisation directe de valeurs numériques (ou caractères et chaînes de caractères constantes ou *littéraux*) rend le code source difficile à maintenir. En cas de modification d'une valeur, il faut penser à modifier toutes les instructions où la valeur est utilisée.

RÈGLE 54

RÈGLE — Ne pas utiliser des valeurs en dur

Les valeurs utilisées dans le code doivent être déclarées comme des constantes.

La règle de déclaration des constantes est également à appliquer pour tous types de valeurs apparaissant plusieurs fois dans le code source. Ainsi, si un caractère ou une chaîne de caractères est

répété plusieurs fois, celui-ci doit également être défini à l'aide d'une variable globale `const` ou à défaut, à l'aide d'une macro préprocesseur.

La centralisation de la déclaration des constantes permet de s'assurer que le changement de leur valeur est appliqué sur l'ensemble de l'implémentation.

BONNE PRATIQUE 55

BONNE PRATIQUE — Centraliser la déclaration des constantes en début de fichier

Pour faciliter la lecture, les constantes sont déclarées ensemble en début du fichier.

Pour identifier ces constantes, plusieurs règles sont à respecter.

RÈGLE 56

RÈGLE — Déclarer les constantes en capitales

Les constantes avec un contrôle de type non nécessaire se déclarent avec le mot clé `#define`.

RÈGLE 57

RÈGLE — Les constantes sans contrôle de type sont déclarées avec la directive `#define` du préprocesseur.

RÈGLE 58

RÈGLE — Les constantes avec un contrôle de type explicite doivent être déclarées avec le mot clé `const`

RÈGLE 59

RÈGLE — Les valeurs constantes doivent être associées à un suffixe dépendant du type

Pour éviter toute mauvaise interprétation, les valeurs constantes doivent utiliser un suffixe selon leurs types :

- il faut utiliser le suffixe `U` pour toutes les constantes de type entier non signé ;
- pour indiquer une constante de type `long` (ou `long long` pour C99), il faut utiliser le suffixe `L` (resp. `LL`) et non `l` (resp. `ll`) afin d'éviter toute ambiguïté avec le chiffre `1` ;
- les valeurs flottantes sont par défaut considérées comme `double`, il faut utiliser le suffixe `f` pour le type `float` (resp. `d` pour le type `double`).

Attention

Par défaut, les valeurs entières sont considérées de type `int` et les valeurs flottantes de type `double`.

RÈGLE 60

RÈGLE — La taille du type associé à une expression constante doit être suffisante pour la contenir

Il faut s'assurer que les valeurs ou expressions constantes utilisées ne dépassent pas du type qui leur est associé.

Pour éviter toute confusion, les constantes octales sont à proscrire. Certains cas peuvent être tolérés comme les modes de fichiers UNIX mais ils seront systématiquement identifiés et commentés.

RECO 61

RECOMMANDATION — Proscrire les constantes en octal

Ne pas utiliser de constante ni de séquence d'échappement en octal.

Mauvais exemple

L'exemple suivant ne centralise pas la définition des constantes. Certaines constantes n'ont pas été déclarées. Il y aussi l'absence de nommage spécifique pour les constantes.

```
#define octal_const 075 /* constante numérique en base octale et
nom en minuscule */

const int64_t b = 0l; /* l et non L  et pb de nommage de la constante */
uint8_t buffer[0x82]; /* constante non déclarée et besoin de contrôle de type
pour cette constante  */
int16_t i;

for(i = 0; i < 0x82; i++) { /* valeur utilisée en dur */
  ...
}

printf("Message\012"); /* séquence d'échappement en base octale \012 = \n */
```

Bon exemple

Le code suivant applique les différentes règles et recommandations pour la déclaration de constantes.

```
const uint32_t INIT_VALUE = 0x1294U; /* constante déclarée et avec le contrôle
de type qui est nécessaire */

#define BUFFER_SIZE 0x82U /* constante déclarée avec un  contrôle de type
non nécessaire */

const int64_t B = 0L; /* correction du suffixe et nommage spécifique de la
    constante */

uint8_t buffer[BUFFER_SIZE];
uint16_t i;

for(i = 0; i < BUFFER_SIZE; i++) {
  ...
}
```

6.3.1 Références

[Misra2012] Rule 11.8 A cast shall not remove a const or volatile qualification from the type pointed

to by a pointer.
[Misra2012] Rule 7.1. Octal constants shall not be used.
[Misra2012] Rule 7.2. A `u` or `U` suffix shall be applied to all integer constants that are represented in an unsigned type.
[Misra2012] Rule 7.3. The lowercase character `l` shall not be used as a literal suffix.
[Misra2012] Rule 7.4. A string literal shall not be assigned to an object unless the object's type is "pointer to a const-qualifed char".
[Misra2012] Rule 12.4 Evaluation of constant expressions should not lead to unsigned integer wrap-around.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL16-C Use `L`, not `l`, to indicate a long value.
[Cert] Rec. DECL00-C Const-qualify immutable objects.
[Cert] Rec. STR05-C Use pointers to const when referring to string literals.
[Cert] Rec. DECL18-C Do not begin integer constants with 0 when specifying a decimal value.
[Cert] Rule EXP40-C Do not modify constants objects.
[Cert] Rule STR30-C Do not attempt to modify string literals.
[Cert] Rec. EXP05-C Do not cast away a const qualification.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rec. DCL06-C Use meaningful symbolic constants to represent literal values.
[Cwe] CWE-547 Use of Hard-coded, security relevant constants.
[Cwe] CWE-704 Incorrect type conversion or cast
[IsoSecu] Modifying string literals [strmod].

6.4 Utilisation limitée de variables globales

Lorsque des variables globales sont utilisées, il est difficile de déterminer les fonctions qui modifient ces variables. Par ailleurs, si une variable globale n'est pas nommée selon des règles établies, la lecture du code d'une fonction utilisant cette variable ne permet pas d'identifier immédiatement l'effet de bord de la fonction sur cette variable globale. Ce nommage spécifique doit être clair et reste au choix du développeur (utilisation de capitales, de préfixe `g_`, etc.).

De plus, l'utilisation de variables globales peut rapidement faire survenir des problèmes de concurrence dans le cas d'une application multi-tâches. Pour chacune des variables globales, le développeur doit étudier la possibilité de limiter la portée de la variable systématiquement.

RÈGLE 62

RÈGLE — Limiter les variables globales au strict nécessaire

Limiter l'usage des variables globales et préférer les paramètres de fonctions pour propager une structure de données au travers d'une application.

Mauvais exemple

Le code suivant utilise une variable globale. Cependant, l'utilisation de celle-ci pourrait facilement être évitée :

```
static uint32_t g_state;
```

```
void foo(void) {
  ...
  g_state = 1;
}

void bar(void) {
  ...
  g_state = 2;
}

int main(int argc, char* argv[]) {
  foo();
  bar();
  ...
}
```

Bon exemple

L'exemple ci-dessous ne fait pas appel à une variable globale. La variable `state` est propagée de fonction en fonction en la passant en tant que paramètre :

```
void foo(uint32_t* state) {
  ...
  (*state) = 1;
}

void bar(uint32_t* state) {
  ...
  (*state) = 2;
}
int main(int argc, char* argv[]) {
  uint32_t state = 0;
  foo(&state);
  bar(&state);
  ...
}
```

6.4.1 Références

[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rule DCL30-C Declare objects with appropriate storage durations.
[Cert] Rec. DCL19-C Minimize the scope of variables and functions.
[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Misra2012] Rule 8.9 An object should be defined at block scope if its identifier only appears in a single function.

6.5 Utilisation du mot-clé `static`

Lorsqu'une fonction est déclarée, définie et utilisée uniquement à l'intérieur d'un seul fichier source, il est fréquent que le spécificateur de classe de stockage `static` soit oublié. Il est alors possible que des conflits surviennent à l'édition des liens. Par ailleurs, l'absence du spécificateur `static` rend la relecture du code plus difficile car cela ne permet pas d'identifier rapidement qu'il s'agit d'une « fonction privée/locale ». Le mot-clé `static` signale au compilateur que la variable/-

fonction est bien une variable/fonction globale mais que sa visibilité doit être limitée au fichier source dans lequel elle est déclarée.

Il en est de même pour les variables globales à un fichier et non utilisées en dehors de ce fichier. Les variables globales de ce type devront être déclarées systématiquement comme `static`. Cela permet de limiter la portée de ces variables uniquement aux autres fonctions définies dans le même fichier et donc de limiter l'exposition des dites variables. Ces fonctions et variables globales ne doivent pas être déclarées dans un fichier d'en-tête.

RÈGLE 63

RÈGLE – Utiliser systématiquement le spécificateur `static` pour les déclarations

Le spécificateur de classe de stockage `static` doit être employé pour toutes les fonctions et variables globales qui ne sont pas utilisées à l'extérieur du fichier source dans lequel elles sont définies.

6.5.1 Références

[Cert] Rec. DCL15-C Declare file-scope objects or functions that do not need external linkage as static.
[Cert] Rule MSC40-C Do not violate constraints.
[Misra2012] Rule 8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
[Misra2012] Rule 8.8 The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

6.6 Utilisation du mot-clé `volatile`

Le mot-clé `volatile` doit être utilisé pour qualifier soit une variable correspondant à une zone matérielle représentant en mémoire un port d'entrée/sortie, soit une variable lue ou écrite par une fonction d'interruption asynchrone. Les accès à de telles variables ne doivent en effet pas faire l'objet d'optimisations de la part du compilateur.

RÈGLE 64

RÈGLE – Seules les variables modifiables en dehors de l'implémentation doivent être déclarées `volatile`

Seules les variables associées à des ports entrée/sortie ou des fonctions d'interruption asynchrone doivent être déclarées comme `volatile` pour empêcher toute optimisation ou réorganisation à la compilation.

De plus, pour éviter un comportement indéfini, seul un pointeur lui-même qualifié de `volatile` peut accéder à une variable `volatile`.

RÈGLE 65

RÈGLE — Seuls des pointeurs qualifiés comme `volatile` peuvent accéder à des variables `volatile`

6.6.1 Références

[Cert] Rec. DCL17-C Beware of miscompiled volatile-qualified variables.
[Cert] Rec. DCL22-C Use volatile for data that cannot be cached.
[Cert] Rule EXP32-C Do not access a volatile object through a nonvolatile reference.
[Misra2012] Rule 2.2 There shall be no dead code.
[Misra2012] Rule 11.8 A cast shall not remove any const or volatile qualification from the type pointed to by a pointer.
[Cwe] CWE-704 Incorrect type conversion or cast.
[Cwe] CWE-561 Dead code.

6.7 Déclaration implicite de type interdite

Le C90 autorise la déclaration implicite de variables au sens de l'omission de type dans certaines circonstances comme pour les paramètres d'une fonction, les éléments d'une structure ou la déclaration d'un `typedef`.

Information

En pratique, les compilateurs émettent une alarme (`-Wimplicit-int`) mais font l'hypothèse implicite que le type est `int`.

RÈGLE 66

RÈGLE — Aucune omission de type n'est acceptée lors de la déclaration d'une variable

Toutes les variables utilisées doivent avoir été préalablement déclarées de façon explicite.

Pour aller plus loin, la déclaration de fonction *à la K&R* [9] comme par exemple :

```
int foo(a,p)
   int a;
   char *p;
{ ...
}
```

est également interdite. D'une part, ce type de déclaration est obsolète et de plus cela réduit la lisibilité du code et donc, potentiellement, les vérifications faites au niveau du compilateur.

9. Syntaxe C de Kernighan & Ritchie avant les normes ANSI

Mauvais exemple

Le code suivant (C90) contient plusieurs déclarations implicites de type.

```
...
const ACONST = 42; /* à proscrire  : le type de la constante non explicitement
défini (implicite int) */
unsigned e; /* à proscrire  : le type de e non explicitement défini (implicite
unsigned int) */
signed f; /* à proscrire  : le type de la constante non pas explicitement défini
(implicite signed int) */
...
int foo(char a, const b) /* à proscrire  : le type de b non explicitement défini
(implicite int) */
{
...
}
bar(char c, const int d) /* à proscrire  : le type de retour de la fonction non
explicitement défini (implicite int)  */
{
..
}
```

Bon exemple

Tous les types sont maintenant explicites.

```
...
const int ACONST = 42;
unsigned int e;
signed int f;
...
int foo(unsigned char a, const int b)
{
...
}
int bar(unsigned char c, const int d)
{
...
}
```

6.7.1 Références

[Cert] Rule DCL31-C Declare identifier before using them.
[Misra2012] Rule 8.1 Types shall be explicitly specified.

6.8 Compound literals

Les *compound literals* ont été introduits par le C99 et permettent de créer des objets sans nom à partir d'une liste de valeurs d'initialisation. Cette construction est souvent utilisée avec des structures en particulier passées en paramètres de fonction. La durée de vie d'un `compound literal` est soit statique soit automatique selon que sa déclaration est faite au niveau fichier ou au niveau d'un bloc d'instructions.

Une tentative d'accès à l'objet associé en dehors de sa portée entraînera un comportement non défini. Il est donc primordial de bien comprendre la portée associée à ce type de construction.

RECO 67

RECOMMANDATION – Limiter l'utilisation des compound literals

Du fait du risque de mauvaise manipulation des *compound literals*, leur utilisation doit être limitée, documentée et une attention particulière doit être donnée à leur portée.

Mauvais exemple

```
#include <stdio.h>
#include <stdint.h>
#define MAX 10

 struct point {
   uint8_t x,y;
  } ;

int main(void)
{
  uint8_t i;
  struct point *tab[MAX];
  for (i = 0; i < MAX; i++){
    tab[i] = &(struct point){i,2*i};
  }
  for (i = 0; i < MAX; i++){
    printf("%d\n", tab[i]->x); /* comportement non défini le compound
    literal créé dans la boucle précédente n'existe plus */
  }
  ...
}
```

Bon exemple

```
#include <stdio.h>
#include <stdint.h>

struct point {
  uint8_t x,y;
} ;

#define MAX 10
int main(void)
{
  uint8_t i;
  struct point tab[MAX];
  for (i = 0; i < MAX; i++){
    tab[i] = (struct point){i, 2*i};
  }
  for (i = 0; i < MAX; i++){
    printf("%d\n", tab[i].x);
  }
  ...
}
```

6.8.1 Références

[Cert] Rec. DCL21-C Understand the storage of compound literals.
[Cert] Rule DCL30-C Declare objects with appropriate storage durations.
[IsoSecu] Escaping of the address of an automatic object [addrescape].

6.9 Énumérations

La valeur non explicitée d'une constante dans une énumération est supérieure de 1 par rapport à la valeur de la constante précédente. Si la première valeur de constante n'est pas explicite alors celle-ci vaut 0. Dans le cas où toutes les valeurs de l'énumération sont implicites, aucun problème ne se pose mais si le développeur explicite certaines valeurs, l'erreur est possible. Il vaut donc mieux éviter de mélanger des constantes à valeurs explicites et à valeurs implicites. Dans le cas où des constantes d'une même énumération possèdent la même valeur, cela entraîne un comportement indéfini. Dans le cas où des valeurs sont explicitées, il faut alors expliciter toutes les valeurs des constantes de l'énumération pour s'assurer qu'aucune des valeurs données n'est répétée.

RÈGLE 68

RÈGLE — Ne pas mélanger des constantes explicites et implicites dans une énumération

Il faut soit expliciter toutes les constantes d'une énumération avec une valeur unique soit n'en expliciter aucune.

Les constantes d'une énumération sont également soumises aux règles de la section 6.3 comme l'utilisation de capitales pour la déclaration de constantes.

Une utilisation parfois observée autour des énumérations est la déclaration d'énumérations anonymes pour la déclaration de constantes. Par exemple :

```
enum {
  ZERO,
  UN
};
```

est utilisé en lieu et place de :

```
const int ZERO=0;
const int UN=1;
```

Les énumérations ne sont pas faites pour cet usage, il s'agit d'un détournement qui peut complexifier la compréhension du code.

RÈGLE 69

RÈGLE — Ne pas utiliser des énumérations anonymes

Mauvais exemple

```
enum une_enum{
  enum1=1,
  enum2,
  enum3,
  enum4=3 /*enum4 et enum3 ont la même valeur*/
};
```

Bon exemple

Toutes les constantes ont une valeur unique et sont en capitales :

```
enum une_enum{
  ENUM1=0,
  ENUM2=1,
  ENUM3=2,
  ENUM4=3
};
```

6.9.1 Références

[Misra2012] Rule 8.12 Within an enumerator list, the value of an implicitly-specified enumeration shall be unique.
[Cert] Rec. INT09-C Ensure enumeration constants maps to unique values.

6.10 Initialisation des variables avant utilisation

En l'absence d'initialisation des variables à leur déclaration, il existe un risque d'utiliser la variable alors que celle-ci n'a pas été initialisée. Le comportement est alors indéfini.

Information

Les variables globales et statiques sont automatiquement initialisées à l'endroit de leur définition, mais il s'agit d'une valeur par défaut spécifiée par le standard. Du fait de la possible méconnaissance de ces valeurs par défaut, il est recommandé d'initialiser toutes les variables.

Un moyen aisé de s'assurer de cela est de le faire de façon systématique lors de la déclaration d'une variable quand celle-ci est déclarée seule ou juste après sa déclaration pour les déclarations multiples.

RECO 70

RECOMMANDATION — Les variables doivent être initialisées à la déclaration ou immédiatement après

Toutes les variables doivent être systématiquement initialisées à leur déclaration ou immédiatement après dans le cas de déclarations multiples.

Information

Le compilateur peut détecter certaines absences d'initialisation. GCC propose par exemple l'option `-Wuninitialized`. La sous-section 5.3.3 donne plus de détails et évoque aussi les limites de tels options. En particulier, l'absence d'avertissement remonté par cette option n'est pas suffisante pour garantir que toutes les variables sont bien initialisées.

Mauvais exemple

Dans l'exemple suivant l'initialisation des variables est manquante au moment de leur utilisation :

```
/* déclarations dans le corps d'une fonction */
uint32_t  a;
uint32_t  b;
uint32_t c;

a = b + c; /* variables utilisées mais non initialisées */
```

Bon exemple

Dans le code ci-dessous, les variables sont bien initialisées avant d'être utilisées (dès leur déclaration ici) :

```
/* déclarations dans le corps de fonction */
uint32_t  a = 0;
uint32_t  b = 0;
uint32_t  c = 0;

a = b + c;
```

6.10.1 Références

[Misra2012] Rule 9.1 : The value of an object with automatic storage duration shall not be read before it has been set.
[Cert] Rule Exp33-C Do not read uninitialized memory.
[Cwe] CWE-457 Use of uninitialized variable.
[Cwe] CWE-758 Reliance on undefined, unspecified, or Implementation-defined behavior.
[Cwe] CWE-908 Use of uninitialized Resource.
[IsoSecu] Referencing uninitialized memory [uninitref].

6.11 Initialisation de variables structurées

Le langage C offre de multiples possibilités pour initialiser les tableaux, structures et autres variables structurées. Ces possibilités étant nombreuses, elles peuvent porter à confusion et également être mal interprétées.

RÈGLE 71

RÈGLE — Ne pas mélanger les différents types d'initialisation pour les variables structurées

Pour l'initialisation d'une variable structurée, un seul et unique type d'initialisation doit être choisi et utilisé.

Mauvais exemple

```
int tab[10] = { 0, [4] = 3, 5, 6, [1] = 1, 2 };
struct type_t o = { .a = 10, 0, "bob" };
```

Bon exemple

```
int tab[10] = { 0, 1, 2, 3, 5, 6, 0, 0, 0, 0 };
struct type_t o = { 10, 0, "bob" };
struct type_t p = { .a = 10, .b = 0, .c ="bob" };
```

Une initialisation des variables structurées souvent utilisée et acceptée est :

```
int tab[N]={0};
une_structure st = {0};
```

Cette initialisation garantit que *tous* les éléments/champs de la variable structurée sont initialisés à zéro.

Attention

Il faut cependant ne pas se méprendre sur la sémantique de cette notation :

```
int tab[N] = {1}; /*ne veut pas dire que tous les éléments sont à 1 mais
que tous sont à zéro et seul le premier élément est à 1.*/
```

Cela vient du fait qu'en cas d'initialisation incomplète d'une variable structurée *i.e.* si tous les champs/éléments ne sont pas explicitement initialisés alors les champs/éléments non listés sont initialisés par défaut à 0. Attention cette initialisation ne concerne pas les éléments de bourrage.

De plus, le C99 a introduit la possibilité d'initialiser un (ou des) élément(s) donné(s) d'un tableau, ce qui ajoute encore une source possible d'erreurs et de confusion, voire de multiples initialisations des mêmes éléments avec des valeurs potentiellement différentes.

RÈGLE 72

RÈGLE — Les variables structurées ne doivent pas être initialisées sans expliciter la valeur d'initialisation et chacun des champs/éléments de la variable structurée doit être initialisé

Les variables non scalaires doivent être initialisées explicitement : chaque élément doit être initialisé en étant clairement identifié sans valeur d'initialisation superflue, ou l'initialiseur `{0}` peut être utilisé à la déclaration. Enfin, les tailles des tableaux doivent être explicitées à l'initialisation.

Mauvais exemple

Les initialisations ne sont pas précises dans l'exemple suivant : initialisations non explicites des éléments des variables structurées et valeurs d'initialisations superflues.

```
int32_t y[5] = {1, 2, 3}; /* l'initialisation n'est pas claire ici - en réalité
les deux derniers éléments sont initialisés à zéro */

int32_t z[2] = {1, 2, 3}; /* idem - en réalité la valeur 3 est ignorée */

int16_t vv[5] = { [0] = -2, [1] = -9, [3] = -8, [2] = 18 }; /* source
d'erreur  les index 2 et 3 non ordonnés et 4 oublié */

struct person {
  unsigned char name[20];
  uint16_t roll;
  float marks;
```

```
  int notes[10];
};

struct person p1 = {" ", 0}; /* obscur */
struct person p2 = {"toto",67,78.3,{0},12}; /* tout est bien initialisé y
compris tous les éléments du tableau notes à 0 mais 12 est ignoré */
```

Bon exemple

Les initialisations sont maintenant explicites et concernent tous les éléments des variables structurées sans valeurs d'initialisation superflues.

```
int32_t y[5] = { 1, 2, 3, 4, 5 }; /* initialisation  complète */

int32_t z[2] = { 1, 2 }; /* aucun élément superflu */

int32_t w[3] = { 0 };  /* notation reconnue pour initialiser tous les éléments à la
     valeur 0 */

int16_t vv[5] = { [0] = -2, [1] = -9, [2] = 18, [3] = -8, [4] = 33 }; /* ok */

struct person {
  unsigned char name[20];
  uint16_t roll;
  float marks;
  int notes[10];
};

struct person p1 = { .name = "titi", .roll = 12, .marks = 10.0f, .note = {0}};
/* tous les éléments sont initialisés explicitement */

struct person p2  = { .name="toto", .roll=67, .marks=78.3, .note={0}}; /* autre
exemple d'initialisation reconnue sans élément superflu cette fois */
```

6.11.1 Références

[Misra2012] Rule 9.2 The initializer for an aggregate or union shall be enclosed in braces.
[Misra2012] Rule 9.3 Arrays shall not be partially initialized.
[Misra2012] Rule 9.4 An element of an object shall not be initialized more than once.
[Misra2012] Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.
[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an array initializer.
[Cwe] CWE-665 Incorrect or incomplete initialization.

6.12 Utilisation obligatoire des déclarations

Lorsque des identifiants sont déclarés mais ne sont pas utilisés ensuite, cela peut signifier que le développeur s'est trompé lors de l'écriture du code et qu'un élément a été utilisé à la place d'un autre ou que son utilisation a été supprimée du programme.

Information

Des options de compilation de GCC et CLANG telles que `-Wunused-variable` et `-Wunused-parameter` permettent de détecter ce genre de motifs.

RECO 73

RECOMMANDATION – Chaque déclaration doit être utilisée

Tous les identifiants déclarés doivent être utilisés, qu'il s'agisse de variables, fonctions, labels, paramètres de fonctions ou autres.

Attention

Dans le cadre du développement d'une bibliothèque, tous les identifiants déclarés ne sont pas obligatoirement utilisés : les fonctions et variables déclarées et exportées par la bibliothèque peuvent évidemment ne pas être utilisées par la bibliothèque elle-même.

Mauvais exemple

Dans le code ci-dessous, les variables déclarées mais non utilisées doivent être supprimées :

```
uint32_t init_list(list_t** pp_list) {
    list_t* p_list = NULL;
    list_element_t*  p_element = NULL;
    uint32_t ui32_list_len = 0;

    if (NULL == pp_list) {
        return 0;
    }

    (*pp_list) = (list_t*)malloc(sizeof(list_t));

    if (NULL == (*pp_list)) {
        return 0;
    }

    (*pp_list)->p_head = NULL;
    (*pp_list)->p_tail = NULL;

    return 1;
}
```

Bon exemple

Dans l'exemple suivant, toutes les variables déclarées sont utilisées :

```
uint32_t init_list(list_t** pp_list) {
    if (NULL == pp_list) {
        return 0;
    }

    (*pp_list) = (list_t*)malloc(sizeof(list_t));

    if (NULL == (*pp_list)) {
        return 0;
    }

    (*pp_list)->p_head = NULL;
    (*pp_list)->p_tail = NULL;

    return 1;
}
```

6.12.1 Références

[Misra2012] Rule 2.2 There shall be no dead code.
[Misra2012] Rule 2.3 A project should not contain unused type declarations.
[Misra2012] Rule 2.4 A project should not contain unused tag declarations.
[Misra2012] Rule 2.5 A project should not contain unused macro declarations.
[Misra2012] Rule 2.6 A project should not contain unused label declarations.
[Misra2012] Rule 2.7 There should be no unused parameters in functions.
[Cert] Rec. MSC07-C Detect and remove dead code.
[Cert] Rec. MSC13-C Detect and remove unused values.
[Cert] Rec. MSC12-C Detect and remove code that has no effect or is never executed.

6.13 Nommage des variables pour les données sensibles

Il est impératif d'utiliser des variables distinctes pour stocker des données sensibles et non sensibles. En l'absence d'une convention de nommage bien définie, le développeur risque d'utiliser des variables pour stocker successivement des données sensibles et non sensibles.

RÈGLE 74

RÈGLE — Utiliser des variables pour les données sensibles distinctes des variables pour les données non sensibles

Pour les données sensibles, il faut également utiliser des variables distinctes pour les données sensibles en clair et des données sensibles protégées en confidentialité et/ou intégrité.

RÈGLE 75

RÈGLE — Utiliser des variables pour les données sensibles et protégées en confidentialité et/ou intégrité distinctes des variables pour les données sensibles non protégées

Ces règles sont plus un principe de programmation sécurisée pour éviter de manipuler dans une même variable des données non sensibles, sensibles chiffrées et sensibles en clair.

Cela va de soi mais il est logiquement interdit de coder en dur toute information sensible quelle qu'elle soit (mot de passe, identifiant, clé de chiffrement, etc.).

RÈGLE 76

RÈGLE — Ne jamais coder en dur une donnée sensible.

Mauvais exemple

Le code ci-dessous n'utilise pas de convention de nommage :

```
#define KEY_SIZE 32U
#define BUFFER_SIZE 512U

 size_t   key_len = 0;
 size_t   clear_data1_len = 0;
 size_t   encrypted_data2_len = 0;
 uint8_t  key[KEY_SIZE];
 uint8_t  data1[BUFFER_SIZE];
 uint8_t  data2[BUFFER_SIZE];
 uint32_t error_code = 0;
 error_code = cipher_data(clear_data, clear_data_len, key, key_len,
 encrypted_data, encrypted_data_len);
```

Bon exemple

Dans l'exemple suivant, une convention de nommage est utilisée pour ne pas utiliser les mêmes variables pour des données sensibles chiffrées ou en clair :

```
#define KEY_SIZE 32U
#define BUFFER_SIZE 512U

/* conventions  :
 suffixe s pour les variables de données sensibles
 préfixe clear pour les données en clair
 préfixe encrypted pour les données chiffrées */

 size_t   encrypted_key_len_s = 0;
 size_t   clear_data_len_s = 0;
 size_t   encrypted_data_len_s = 0;
 uint8_t  encrypted_key_s[KEY_SIZE];
 uint8_t  clear_data_s[BUFFER_SIZE];
 uint8_t  encrypted_data_s[BUFFER_SIZE];
 uint32_t encrypted_error_code_s = 0;

 encryptederrorCode = cipher_data(clear_data_s, clear_data_len_s,
 encrypted_key_s, encrypted_key_len_s, encrypted_data_s, encrypted_data_len_s);
```

6.13.1 Références

[Misra2012] Dir. 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous.
[Cert] Rec. DCL02-C Use visually distinct identifiers.
[Cert] Rule MSC41-C Never hard code sensitive information.
[Cwe] CWE-259 Use of Hard-Coded Password.
[Cwe] CWE-798 Use of Hard-Coded Credentials.

7
Types et transtypages

7.1 Taille explicite pour les entiers

Attention

Le standard C ne définit pas une taille explicite pour chaque type d'entiers. En particulier, pour le type `int`, suivant l'architecture, celui-ci peut être sur 16, 32 ou 64 bits.

De ce fait, l'utilisation du type `int` est risquée car il faut être sûr de la taille associée et des valeurs possibles pour éviter tout débordement ou comportement non attendu comme bouclage de valeurs (*wrap* pour des entiers non signés).

Il vaut donc mieux éviter l'utilisation de ce type sauf si le développeur est certain que la plage de valeurs associées est bien contenue dans le type (par exemple dans des compteurs de boucle).

Le nommage du type doit inclure sa taille sur la machine cible de manière explicite comme ceux définis dans le fichier d'en-tête `stdint.h` disponible dans C99. Son utilisation est à privilégier plutôt que le type générique `int`. En C90, des types équivalents doivent être définis et utilisés. La redéfinition de types entiers est possible mais cette redéfinition doit être explicite à la fois sur la taille et le signe associés.

RECO 77

RECOMMANDATION – Seuls des types d'entiers dont la taille et le signe sont explicites doivent être utilisés

De plus, le type `char` pur et simple ne doit pas être utilisé pour des valeurs numériques car son signe n'est pas spécifié par le standard C et dépend des implémentations. Ce type est réservé à la manipulation de caractères.

RÈGLE 78

RÈGLE – Seuls les types `signed char` et `unsigned char` doivent être utilisés pour manipuler des valeurs numériques.

Mauvais exemple

```
#define MAXUINT16 65535U
int value;
char c = 35; /* signe non précisé */

if (value >= MAXUINT16)
{
  /* dépendant de l'architecture */
}
```

Bon exemple

```
#include <stdint.h> /* si C99 */
#define MAXUINT16 65535U
unsigned char c = 35U; /* signe explicité car valeur numérique */
...
uint32_t  value; /* si C99 */

typedef unsigned char uint8_t; /* définition de type en C90 */
if (value >= MAXUINT16)
{
  /* ... */
}
```

7.1.1 Références

[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.
[Misra2012] Rule 10.3 The value of an expression should not be assigned to an object with a narrower essential type or of a different essential type category.
[Misra2012] Rule 10.4 Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
[Misra2012] Rule 8.1 Types shall be explicitly specified.
[Misra2012] Directive 4.6 `typedef` that indicate size and signedness should be used in place of the basic numerical types.
[Cert] Rec. INT00-C Understand the data model used by your implementation(s).
[Cert] Rec. INT07-C Use only explicitly signed or unsigned char type for numeric values.
[Cert] Rule INT35-C Use correct integer precisions.
[Cert] Rec STR00-C Represent characters using an appropriate type.
[Cwe] CWE-682 Incorrect calculation.

7.2 Alias de types

L'opérateur `typedef` autorise la redéfinition d'un type qui a lui-même été redéfini avec `typedef`. Il est alors difficile de suivre le typage effectif d'une variable. Il y a un risque important de confusion et de création de doublons dans les définitions de type. Il ne faut donc pas redéfinir plus d'une fois les types définis par le langage ou par des bibliothèques externes. Si cela est fait de façon intentionnel par la développeur en vue d'un typage fort (*strong type checking*), cela doit être commenté et expliqué mais aussi maitrisé pour éviter toute confusion de type.

RECO 79

RECOMMANDATION — Ne pas redéfinir des alias de types

Mauvais exemple

L'exemple suivant (C90) présente des alias multiples d'un même type :

```
typedef unsigned short  uint16_t; /* définition du type uint16_t */
typedef unsigned short  uint16_type; /* le type uint16_type est un alias du type
    uint16_t */
typedef uint16_t unsigned_short; /* le type unsigned_short est  une redéfinition
du type uint16_t */
```

Bon exemple

Dans l'exemple suivant (en C90 *i.e.* avant l'introduction de `stdint.h`), un seul type est bien défini à partir de la définition d'un type standard du langage :

```
typedef unsigned short  uint16_t; /* définition du type uint16_t */
```

7.2.1 Références

[Cert] Rec. PRE03-C Prefer typedefs to defines for encoding non-pointer types.

7.3 Transtypage

Les compilateurs C effectuent des conversions implicites d'un type à un autre. Cependant, ces promotions de type et conversions implicites peuvent aboutir à des erreurs (perte d'information, erreurs de calcul). En outre, l'absence des conversions explicites ne facilite pas la relecture de code. Il faut donc ajouter l'opérateur de conversion de type de façon systématique et ne pas mélanger dans une même opération arithmétique des types signés et non signés (opérateurs : +, -, *, /, %, ~, >, <, >=, <=, <<, >>...).

De multiples conversions implicites sont faites que ce soit en C90 ou en C99.

D'une part, la *promotion d'entiers* est effectuée sur des valeurs entières dont le type est plus petit que le type `int` et quand ces valeurs entières sont soumises à une opération (opérateurs binaires, unaires, décalages ...). Ces valeurs entières sont alors automatiquement et systématiquement converties en `int` ou `unsigned int`.

D'autre part, la *balance* (ou équilibrage) de types correspond à une conversion classique à un type commun quand des opérandes sont de types différents. Enfin, la dernière conversion implicite correspond à l'affectation d'une valeur dans un type différent.

Information

Le détail de la promotion d'entiers peut être consulté dans les sections 6.2.1.1. et 6.3.1.1. respectivement des normes [AnsiC90] et [AnsiC99]. Pour l'équilibrage des types, les sections concernées sont les sections 6.2.1.5 et 6.3.1.8 respectivement des normes [AnsiC90] et [AnsiC99].

RÈGLE 80

RÈGLE – Compréhension fine et précise des règles de conversions

Le développeur se doit de connaître et comprendre toutes les règles de conversion implicites des types entiers.

Le développeur se doit d'expliciter les conversions implicites dans le code pour éviter toute erreur. Le cas classique souvent source d'erreur est une conversion implicite entre des types signés et non signés.

RÈGLE 81

RÈGLE – Conversions explicites entre des types signés et non signés

Proscrire les conversions implicites de types. Utiliser des conversions explicites notamment entre type signé et type non signé.

Mauvais exemple

```
signed int v1 = -1 ;
unsigned int v2 = 1 ;
if (v1 < v2)
{
  /* v1 converti en unsigned int et valeur -1 devient UINT_MAX  donc
  la condition du if est toujours false */
}
```

Bon exemple

```
signed int v1 = -1;
unsigned int v2 = 1;
if (v1 < (signed int)v2)
{
  /* v2 est converti explicitement en entier signé - la condition est
  vraie  */
}
```

Toujours pour les mêmes raisons, il ne faut pas faire de conversion implicite entre un type entier et un type flottant ou d'un type entier vers un type entier plus petit.

Mauvais exemple

Dans les lignes suivantes, les conversions sont implicites :

```
uint32_t u32;
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;
```

```c
s32 = 42;
u32 = s32; /* conversion implicite */
u16 = u32 + 2 * s32; /* conversion implicite vers un type plus petit */
dbl = u32 / u16; /* le résultat vaut 0 (division entière) */
s32 = dbl; /* conversion implicite flottant -> entier */

/* la boucle suivante est infinie  :idx étant non signé, idx >= 0 est
toujours vrai car une valeur non signée ne peut être négative */
for(idx = 27; idx >= 0; idx--) {
  ...
}
```

Exemple toléré

L'exemple ci-dessous présente un code dans lequel les conversions de type sont explicites. Il s'agit d'un exemple dit toléré car d'autres solutions plus propres comme l'incrément de l'indice de boucle existent.

```c
uint32_t u32;
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;

s32 = 42;
u32 = (uint32_t)s32;
u16 = (uint16_t)((int32_t)u32 + 2 * s32);
dbl = (double)u16 / (double)u32;

/* le cast en entier signé permet d'éviter une boucle  infinie */
for(idx = 27; (int8_t)idx >= 0; idx--) {
  ...
}
```

Bon exemple

```c
int32_t s32;
uint16_t u16;
double dbl;
uint8_t idx;

s32 = 42;
u32 = (uint32_t)s32;
u16 = (uint16_t)((int32_t)u32 + 2 * s32);
dbl = (double)u16 / (double)u32;

/* changement de la boucle */
idx=27;
while (idx>0)
{
  ...
}
```

Information

Les avertissements `-Wconversion` et `-Wsign-conversion` de GCC et CLANG peuvent aider à détecter ce genre de conversions implicites.

7.3.1 Références

[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.

[Misra2012] Rule 10.3 The value of an expression should not be assigned to an object with a narrower essential type or of a different essential type category.
[Misra2012] Rule 10.4 Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
[Misra2012] Rule 10.5 The value of an expression should not be cast to an inappropriate essential type.
[Misra2012] Rule 10.6 The value of a composite expression shall not be assigned to an object with wider essential type.
[Misra2012] Rule 10.7 If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
[Misra2012] Rule 10.8 The value of a composite expression shall not be cast to a different essential type category or a wider essential type.
[Cert] Rec. INT02-C Understand integer conversion rules.
[Cert] Rule INT30-C Ensure that unsigned integer operation do not wrap.
[Cert] Rule INT31-C Ensure that integer conversions do not result in lost or misinterpreted data.
[Cert] Rule INT32-C Ensure that operations on signed integers do not result in overflow.
[Cert] Rec. INT18-C Evaluate integer expressions in a larger size before comparing or assigning to that size.
[Cert] Rec. EXP14-C Beware of integer promotion when performing bitwise operations on integer types smaller than int.
[Cwe] CWE-190 Integer overflow or wraparound.
[Cwe] CWE-192 Integer coercion error.
[Cwe] CWE-197 Numeric Truncation Error.
[Cwe] CWE-681 Incorrect conversion between numerical types.
[Cwe] CWE-704 Incorrect Type Conversion or Cast.
[IsoSecu] Conversion of signed characters to wider integer types before a check for EOF [signconv].
[IsoSecu] Overflowing signed integers [intoflow].

7.4 Transtypage de pointeurs sur des variables structurées de types différents

Le transtypage depuis ou vers des variables structurées via des pointeurs peut aboutir à des débordements lorsque le type cible est d'une taille plus grande que la zone mémoire pointée. En effet, un transtypage d'une structure vers une structure plus grande par exemple, donne un accès non souhaitable à des zones de la mémoire extérieure à la structure initiale.

Par ailleurs, le transtypage depuis/vers des variables structurées rend la relecture de code plus complexe.

RECO 82

RECOMMANDATION – Ne pas utiliser de transtypage de pointeurs sur des types structurés différents

Mauvais exemple

Dans le code suivant, un transtypage de structure va aboutir à un débordement.

```
#define TAB_SIZE 16U

typedef struct {
  int32_t magic;
} s_a;

typedef struct {
  int32_t magic;
  int16_t s;
  uint8_t x[TAB_SIZE];
} s_b;

void foo(s_a* structa) {
  s_b* p = (s_b*)structa; /* transtypage à bannir */

  p->magic = 0xBAADCAFE;
  p->s = 0xDEAD; /* débordement hors de la structure s_a */
  p->x[0] = 4; /* et risque de données écrasées (buffer overflow)*/
}
```

Bon exemple

Dans le code ci-dessous, le transtypage de structure n'est plus effectué :

```
#define TAB_SIZE 16U

typedef struct {
  int32_t magic;
} s_a;

typedef struct {
  s_a h;
  int16_t s;
  uint8_t x[TAB_SIZE];
} s_b;

void foo(s_b* structb) {
  structb->h.magic = 0xCAFEBABE;
  structb->s = 0xBEEF;
  structb->x[0] = 4;
}
```

7.4.1 Références

[Misra2012] Rule 11.2 Conversions shall not be performed between a pointer to an incomplete type of any other type.
[Misra2012] Rule 11.3 A cast shall not be performed between a pointer to object type and a pointer to a different object type.
[Misra2012] Rule 11.8 A cast shall not remove a const or volatile qualification from the type pointed to by a pointer.
[Cert] Rule EXP36-C Do not cast pointer into more strictly aligned pointer types.
[Cwe] CWE-704 Incorrect type conversion or cast.
[IsoSecu] Converting pointer values to more strictly aligned pointer types [alignconv].

8
Pointeurs et tableaux

Nous ne parlons, dans cette section, que de tableaux à simple dimension mais comme tout tableau multidimensionnel peut aussi être représenté via un tableau à simple dimension, toutes les règles et recommandations s'appliquent de fait aussi aux tableaux multidimensionnels.

8.1 Accès normalisé aux éléments d'un tableau

La confusion entre tableaux et pointeurs est courante, et il est souvent admis qu'un tableau se comporte comme un pointeur constant sur son premier élément. Cette affirmation est un raccourci qui s'avère faux dans le cas général.

Ainsi, pour le code suivant :

```
int32_t tab1[6];
int32_t * tab2 = malloc(6 * sizeof(int32_t));
int32_t * tab3 = tab1;
int32_t * tab4 = tab2;

printf("tab1  :%p, %p, %p\n", tab1, &tab1[0], &tab1);
printf("tab2  :%p, %p, %p\n", tab2, &tab2[0], &tab2);
printf("tab3  :%p, %p, %p\n", tab3, &tab3[0], &tab3);
printf("tab4  :%p, %p, %p\n", tab4, &tab4[0], &tab4);
```

le résultat obtenu est le suivant :

```
tab1  :1559248928, 1559248928, 1559248928 /* tab1=&tab1[0]=&tab1 représentent tous l'adresse
                                             du premier élément du tableau */
tab2  :911295072, 911295072, 1559248904   /* &tab2 est l adresse du pointeur retourné par malloc
                                             pointant sur le tableau et tab2 (ou &tab2[0])
                                             l'adresse du premier élément du tableau tab2 */
tab3  :1559248928, 1559248928, 1559248912 /* cas similaire à tab2 */
tab4  :911295072, 911295072, 1559248920   /* cas similaire à tab2 */
```

Les subtilités entre tableaux et pointeurs sont nombreuses et on ne pourra qu'attirer l'attention du lecteur et l'inciter à la plus grande prudence.

Un autre exemple de code prêtant à confusion est :

```
int *var[N];
int (*var2)[N];
```

La première ligne consiste à déclarer `N` pointeurs de type `int` en mémoire soit un tableau de `N` pointeurs de type `int`. La seconde ligne déclare un pointeur sur un tableau de `N` éléments de type `int` en mémoire.

Le standard précise ce point en expliquant que toute *expression* de type tableau est convertie en une expression de type pointeur pointant sur le premier élément du tableau et n'est pas une *lvalue*

sauf quand l'expression initiale est utilisée comme opérande des opérateurs `sizeof`, `_Alignof` ou `&` ou si l'expression est une chaîne littérale utilisée pour initialiser un tableau.

Expression

Une expression n'est pas un objet en mémoire mais un bout de code source comme `a+b` ou `&a` par exemple.

Lvalue

Une *lvalue* (*locator value*) est une expression avec un type même incomplet mais différent de *void* qui est associé à une adresse en mémoire.

Un tableau n'est pas une *lvalue* modifiable ce qui implique qu'il ne peut pas être affecté, incrémenté ou modifié en général.

```
int tab[N];
tab = 0; // erreur
tab--; // erreur
```

Lorsque l'expression de type tableau est convertie en une expression de type pointeur, cette expression produit alors une simple valeur et n'est plus une *lvalue*.

Attention

Pour un tableau `tab`, les notations `tab` et `&tab[0]` représentent l'adresse du premier élément du tableau créé en mémoire. La notation `&tab` va en revanche varier. Quand un tableau est déclaré statiquement, l'adresse du tableau ne peut pas changer et il n'y a pas de création de pointeur à proprement parler sur le tableau : la notation `tab` est assimilable à une étiquette gérée par le compilateur contenant l'adresse du tableau.

- Ainsi, si le tableau est déclaré statiquement (cas `tab1` dans l'exemple précédent), `&tab` représente toujours l'adresse du premier élément du tableau *i.e.* l'adresse du tableau.
- En revanche, si le tableau est déclaré dynamiquement, la notation `tab` représente le pointeur contenant l'adresse du tableau créé en mémoire et donc, `&tab` représente l'adresse du pointeur sur le tableau (cas `tab2` de l'exemple).

La norme C permet de représenter l'accès au i^e élément d'un tableau `tab` de diverses façons qui peuvent être sources d'erreurs ou de confusion.

Attention

Pour un tableau `tab`, l'accès au i^e élément peut s'écrire :

```
*(tab+i); /* notation usuelle 1 */
tab[i]; /* notation usuelle 2  */
*(i+tab); /* tableau et indice interchangeables ! */
i[tab]; /* tableau et indice interchangeables !  */
```

Ces notations sont toutes reconnues par la norme et sont donc correctes mais cela peut rapidement gêner la compréhension du code. Notons que, même avec des options exigeantes de compilation, ni GCC ni CLANG n'émettra d'alertes sur ce type de notations qui peuvent être source d'erreurs.

Pour éviter toute ambiguïté et mauvaise compréhension du code et donc potentiellement des erreurs, les notations tolérées par la norme visant à inverser indice et nom d'un tableau ne seront pas utilisées.

RÈGLE 83

RÈGLE — L'accès aux éléments d'un tableau se fera toujours en désignant en premier attribut le tableau et en second l'indice de l'élement concerné

L'accès au i^{ème} élément d'un tableau s'écrira toujours avec le nom du tableau en premier suivi de l'indice de la case à atteindre.

De plus, toujours par soucis de transparence, la notation typique des tableaux via les crochets `[]` sera préférée.

RECO 84

RECOMMANDATION — L'accès aux éléments d'un tableau doit se faire en utilisant les crochets

Dans le cas d'une variable de type tableau, la notation dédiée (via les crochets) doit être utilisée pour éviter toute ambiguïté.

Mauvais exemple

```
for (i = 0; i< size_tab; i++) {
  *(i+tab) = i; /* les crochets ne sont pas utilisés et l'indice est en première
  position */
  ...
}
```

Bon exemple

```
for (i = 0; i < size_tab; i++) {
  tab[i] = i;
  ...
}
```

8.1.1 Références

[Cert] ARR00-C Understand how arrays work.

8.2 Non utilisation des VLA

Les VLA [10] introduits avec le C99 correspondent à des tableaux dont la taille n'est pas associée à une expression entière constante à la compilation mais une variable entière. Cela correspond donc à implémenter un objet de taille variable sur la pile. Si la taille du tableau n'est pas strictement positive, cela correspond à un comportement indéfini du C. De plus, pour une taille excessive du tableau, le comportement du programme peut être différent de celui attendu. Enfin, si la taille du

10. Variable-Length Array

dit tableau peut être contrôlée par l'utilisateur, il s'agit d'une vulnérabilité. Pour toutes ces raisons, les VLA ne doivent pas être utilisées.

Information

L'option `-Wvla` permet d'alerter sur l'utilisation de VLA dans le code.

RÈGLE 85

RÈGLE — Ne pas utiliser de VLA

8.2.1 Références

[Misra2012] Rule 18.8 Variable-length array types shall not be used.
[Cert] ARR32-C Ensure size arguments for VLA are in a valid range.
[Cert] MEM05-C Avoid large stack allocations.
[Cwe] CWE-758 Reliance on undefined, unspecified, or Implementation-defined behavior.
[IsoSecu] Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink].

8.3 Taille explicite des tableaux

La taille des tableaux doit être explicite afin d'éviter les accès hors bornes. Cette recommandation peut paraître redondante par rapport à une règle précédente imposant des déclarations explicites mais le point d'attention est mis ici sur le cas de la taille des tableaux.

RECO 86

RECOMMANDATION — Ne pas utiliser de taille implicite pour les tableaux

Afin de s'assurer que les accès tableaux sont bien valides, la taille de ceux-ci doit être explicitée.

Mauvais exemple

Dans l'exemple ci-dessous, la taille du tableau est implicite par rapport à son initialisation :

```
int32_t tab [] = { 1, 2, 3 }; /* tableau de 3 éléments, taille implicite */
```

Bon exemple

Cette fois, la taille des tableaux est clairement explicitée :

```
int32_t tab[3] = { 1, 2, 3 }; /* tableau de 3 éléments, taille explicite, avec
initialisation */
int32_t tab2[2] = { 2, 3 };  /* tableau de 2 éléments, taille explicite, avec
initialisation */
```

8.3.1 Références

[Misra2012] Rule 8.11 When an array with external linkage is declared, its size should be explicitly specified.
[Misra2012] Rule 9.5 Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.
[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.
[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an initializer.
[Cwe] CWE-655 Incorrect or incomplete initialization.

8.4 Vérification systématique de non débordement de tableau

L'accès à une case de tableau en dehors de la taille allouée est une erreur classique de développement. Pour chaque accès à l'élément d'un tableau, il faut vérifier si l'indice utilisé est bien strictement positif ou nul et strictement inférieur au nombre d'éléments alloués pour ce tableau.

RÈGLE 87

RÈGLE — Utiliser des entiers non signés pour les tailles de tableaux

RÈGLE 88

RÈGLE — Ne pas accèder à un élément de tableau sans vérifier la validité de l'indice utilisé

La validité des indices de tableau utilisés doit être vérifié de façon systématique : un indice de tableau est valide s'il est supérieur ou égal à zéro et strictement inférieur à la taille déclarée du tableau. Dans le cas d'un tableau de caractères, le caractère de fin de chaîne '\0' doit être pris en compte.

Mauvais exemple

```
i++;
tab[i] = i; /* pas de vérification de non débordement */
```

Bon exemple

```
for (i = 0; i < size_tab; i++){ /* size_tab est le nombre d'éléments du tableau
  */
  tab[i] = i;
  ...
}
```

8.4.1 Références

[Cert] Rec. ARR02-C Explicitly specify array bounds, even if implicitly defined by an initializer.
[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.
[Cert] Rule STR31-C Guarantee that storage for strings has sufficient space for character data and the null terminator.
[IsoSecu] Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink].
[IsoSecu] Forming or using out-of-bound pointers or array subscripts [invptr].
[IsoSecu] Using a tainted value to write to an object using a formatted input or output function [taintformatio].
[IsoSecu] Tainted strings are passed to a string copying function [taintstrcpy].
[Cwe] CWE-119 Improper Restriction of Operations within the bounds of a Memory buffer.
[Cwe] CWE-120 Buffer Copy without Checking Sizeof Input (Classic Buffer Overflow).
[Cwe] CWE-123 Write-what-where Condition.
[Cwe] CWE-125 Out-of-bounds read.
[Cwe] CWE-129 Improper Validation of Array Index.
[Cwe] CWE-170 Improper Null termination.

8.5 Ne pas déréférencer des pointeurs NULL

Déréférencer un pointeur *NULL* entraîne un comportement non défini. Cela peut amener à une terminaison anormale du programme. Il faut donc s'assurer avant de déréférencer un pointeur que celui-ci n'est pas *NULL*.

RÈGLE 89

RÈGLE — Un pointeur NULL ne doit pas être déréférencé

Avant de déréférencer un pointeur, le développeur doit s'assurer que celui-ci n'est pas *NULL*.

Mauvais exemple

Dans la fonction suivante, le pointeur passé en paramètre est utilisé sans être vérifié :

```
void fonction(const unsigned char *input)
{
  size_t size = strlen(input); /* le pointeur peut être NULL */
  ...
}
```

Bon exemple

La gestion de l'erreur liée au pointeur NULL à été ajoutée :

```
void fonction(const unsigned char *input)
{
  if (NULL == input)
  {
    /* gestion cas pointeur NULL */
  }
```

```
    else
    {
    size_t size = strlen(input);
    /* ... */
    }
}
```

8.5.1 Références

[Cert] Rule EXP34-C Do not dereference null pointers.
[IsoSecu] Dereferencing an out-of-domain pointer [nullref].
[Cwe] CWE-476 NULL Pointer Dereference.
[AnsiC99] Section 6.5.3.2..
[AnsiC90] Section 6.3.3.3..

8.6 Affectation à `NULL` des pointeurs désalloués

Suite à la désallocation de la mémoire pointée par un pointeur, la variable pointeur stocke encore son adresse. On parle de *dangling pointer*.

Dangling pointer

Un *dangling pointer* est un pointeur qui contient l'adresse mémoire d'un élément qui a été libéré.

En cas de bogue et d'utilisation erronée du pointeur désalloué, la mémoire risque d'être corrompue. La mémoire une fois libérée peut-être réutilisée (ou non) par le système. Le résultat de l'utilisation de la zone mémoire (via le pointeur) est alors non défini et non forcément visible et peut poser des problèmes de sécurité (*use-after-free*). L'affectation à `NULL` du pointeur, après désallocation, permet de spécifier que le pointeur ne pointe plus sur une zone mémoire valide. Et en cas d'utilisation accidentelle du pointeur, aucune zone mémoire ne va être corrompue puisque le pointeur ne pointe plus sur aucune zone mémoire valide.

RÈGLE 90

RÈGLE – Un pointeur doit être affecté à `NULL` après désallocation

Un pointeur doit être systématiquement affecté à `NULL` suite à la désallocation de la mémoire qu'il pointe.

Mauvais exemple

Dans le code ci-dessous, le pointeur n'est pas mis à `NULL` suite à sa désallocation.

```
list_t *p_list = NULL;
p_list = create_list();
...
if(p_list != NULL) {
   free_list(p_list);
}
/* la mise à NULL de p_list est manquante. */
```

Bon exemple

Dans l'exemple suivant, le pointeur est bien mis à `NULL` suite à la désallocation de la zone pointée :

```
list_t *p_list = NULL;
p_list = create_list();
...
if(p_list != NULL) {
   free_list(p_list);
   p_list = NULL;
}
```

8.6.1 Références

[Cert] Rule MEM30-C Do not access freed memory.
[Cert] Rec MEM01-C Store a new value in pointers immediately after free().
[Misra2012] Rule 18.6. The adresse of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
[Cwe] CWE-415. Double free.
[Cwe] CWE-416. Use after free.
[Cwe] CWE-672 Operation on a resource after expiration or release.
[IsoSecu] Accessing freed memory [accfree].
[IsoSecu] Freeing memory multiple times [dbfree].

8.7 Utilisation du qualificatif de type `restrict`

Le qualificatif `restrict`, introduit dans le C99, est un moyen d'indiquer au compilateur qu'on ne peut pas accéder à la zone pointée sans passer par le pointeur marqué `restrict`. Un pointeur associé au qualificatif `restrict` implique donc que l'objet pointé par celui-ci soit atteint directement ou indirectement uniquement par ce pointeur. Cela demande donc de ne pas avoir d'autre alias sur l'objet pointé.

Alias

Deux alias sont deux variables ou chemins d'accès permettant d'atteindre une même case mémoire.

Attention

Le qualificatif `restrict` est une déclaration d'intention du développeur d'associer un seul et unique pointeur à une zone mémoire et non un état de fait. En effet, en pratique, rien n'empêche d'atteindre la même zone via un pointeur différent.

Le comportement devient indéfini si des objets pointés par des pointeurs de type `restrict` ont des adresses mémoires communes. De plus, cela impose de vérifier l'absence d'adresses mémoires communes à chaque appel de fonctions avec des paramètres de type `restrict` mais aussi pendant l'exécution des dites fonctions.

Attention

Plusieurs fonctions de la bibliothèque standard ont des paramètres de type `restrict` depuis C99 (`memcpy`, `strcat`, `strcpy`, ...).

Il est très facile d'introduire un comportement indéfini via l'utilisation de `restrict` car il faut s'assurer qu'aucun des pointeurs concernés ne partage une zone mémoire tout en prenant en compte les appels de fonctions de la bibliothèque standard qui ont également des paramètres de type `restrict` depuis C99. L'utilisation du qualificatif `restrict` directement par l'utilisateur est donc à proscrire.

RÈGLE 91

RÈGLE — Ne pas utiliser le qualificatif de pointeur `restrict`

Le qualificatif `restrict` ne doit pas être utilisé directement par le développeur. Seule l'utilisation indirecte *i.e.* via l'appel de fonctions de la bibliothèque standard est tolérée mais le développeur devra s'assurer qu'aucun comportement indéfini résultera de l'utilisation de telles fonctions.

Information

L'option `-Wrestrict` de GCC permet d'alerter sur une mauvaise utilisation de pointeurs `restrict`.

Mauvais exemple

Dans l'exemple suivant, les pointeurs de type `restrict` partagent des zones mémoire et donc provoquent un comportement non défini :

```
uint16_t * restrict ptdeb;
uint16_t * restrict ptfin;
uint16_t tab[12];
unsigned char * pt1;
unsigned char * pt2;
unsigned char c_str[] = "blabla";
...
ptdeb = &tab[0];
ptfin = &tab[11];
ptdeb = ptfin; /* comportement non défini */
...
pt1 = pt2 + 2;
memcpy(pt2, pt1, 3); /* comportement non défini - paramètres de memcpy
de type restrict */
```

Bon exemple

Dans l'exemple suivant, les qualificatifs *restrict* ont été supprimés et il n'y a plus de comportement non défini :

```
uint16_t *  ptdeb; /* suppression de restrict */
uint16_t *  ptfin; /* suppression de restrict */
uint16_t tab[12];
unsigned char * pt1;
unsigned char * pt2;
unsigned char c_str[] = "blabla";
...
ptdeb = &tab[0];
ptfin = &tab[11];
ptdeb = ptfin; /* ok */
...
```

```
pt1 = pt2 + 2;
memmove(pt2, pt1, 3); /* changement de fonction  */
```

8.7.1 Références

[Misra2012] Rule 8.14 The restrict type shall not be used.
[IsoSecu] Passing pointers into the same objects as arguments to different restrict-qualified parameters [restrict].
[Cert] Rule EXP43-C Avoid undefined behavior when using restrict-qualified pointers.

8.8 Limitation du nombre d'indirections de pointeur

Lorsqu'un pointeur présente plus de deux niveaux d'indirection (par exemple : pointeur de pointeur de pointeur `int32_t ***pppInt32`), il devient difficile de comprendre les intentions du développeur et le comportement du code.

RECO 92

RECOMMANDATION — Le nombre de niveau d'indirections de pointeur doit être limité à deux

Le nombre d'indirections pour un pointeur ne doit pas dépasser deux niveaux.

Mauvais exemple

Le code ci-dessous présente des niveaux d'indirection trop importants :

```
void fonction(int8_t ***arr_pt) /* 3 niveaux */
{
  int8_t ***pt;
...
}
```

Bon exemple

Dans l'exemple suivant, des pointeurs temporaires sont introduits afin de faciliter l'accès aux données et limiter le nombre d'indirections :

```
typedef int8_t *int8ptr_t;
void fonction(int8ptr_t **arr_pt) /* réduction à deux niveaux */
{
  int8_t *pt_temp; /* pointeur temporaire */
  int8ptr_t **pt;
...
}
```

8.8.1 Références

[Misra2012] Rule 18.5 Declarations should contain no more than two levels of pointer nesting.

8.9 Privilégier l'utilisation de l'opérateur d'indirection ->

Deux écritures sont possibles dans le langage C pour atteindre un champ de structure par l'intermédiaire d'un pointeur : l'opérateur d'indirection `ptr->field` et le déréférencement `(*ptr).field`. Cependant la seconde écriture est souvent source d'erreurs et de problèmes de compréhension. Il vaut donc mieux éviter d'utiliser le déréférencement `(*ptr).field` pour atteindre un champ d'une structure par l'intermédiaire d'un pointeur.

RECO 93

RECOMMANDATION – Préférer l'utilisation de l'opérateur d'indirection ->

L'opérateur d'indirection -> doit être utilisé pour atteindre les champs d'une structure par l'intermédiaire d'un pointeur.

Mauvais exemple

Dans l'exemple ci-dessous, l'accès devrait être réécrit avec l'opérateur d'indirection :

```
(*list.p_head).pNext = NULL;
```

Bon exemple

Dans le code ci-dessous, l'opérateur d'indirection est bien utilisé :

```
list.p_head->pNext = NULL;
```

8.10 Arithmétique des pointeurs

Le langage C permet d'accéder directement à la mémoire en utilisant des pointeurs. Il est possible d'appliquer des opérations arithmétiques sur la valeur d'un pointeur que ce soit pour l'incrémenter ou le décrémenter.

Arithmétique de pointeurs

L'arithmétique de pointeurs correspond à utiliser les valeurs des pointeurs comme valeur entière dans une opération arithmétique élémentaire (soustraction et addition).

L'arithmétique de pointeurs est très souvent utilisée dans le cas de pointeur sur un élément de tableau pour naviguer entre les différents éléments du tableau. En dehors de ce cas, l'arithmétique sur des adresses mémoire est très risquée.

RÈGLE 94

RÈGLE – Seul l'incrément ou le décrément de pointeurs de tableaux est autorisé

L'incrément ou le décrément de pointeurs ne doit être utilisé que sur des pointeurs représentant un tableau ou un élément d'un tableau.

L'arithmétique sur des pointeurs de type `void*` est, par voie de conséquence, interdite. En effet, aucune taille mémoire n'est associée au type `void*` ce qui provoque un comportement non défini, en plus de la violation de la règle précédente.

RÈGLE 95

RÈGLE — Aucune arithmétique sur les pointeurs `void*` n'est autorisée

Il faut proscrire l'utilisation de toute arithmétique sur des pointeurs de type `void*`.

Même dans le cas d'arithmétique de pointeurs sur les éléments d'un tableau, il faut s'assurer que l'arithmétique ne va pas causer de déréférencement hors du tableau.

RECO 96

RECOMMANDATION — Arithmétique des pointeurs sur tableaux contrôlée

L'arithmétique sur des pointeurs représentant un tableau ou un élément d'un tableau doit être faite en s'assurant que le pointeur résultant pointera toujours sur un élément du même tableau.

Par voie de conséquence, les soustractions ou comparaisons entre pointeurs n'auront un sens que pour des pointeurs sur un même tableau.

RÈGLE 97

RÈGLE — Soustraction et comparaison entre pointeurs d'un même tableau uniquement

Seules les soustractions et comparaisons de pointeurs sur un même tableau sont autorisés.

Enfin, l'affectation d'une adresse fixe à un pointeur est vivement déconseillée.

RECO 98

RECOMMANDATION — Il ne faut pas affecter directement une adresse fixe à un pointeur.

Mauvais exemple

```
#include <stddef.h>
#include <stdint.h>
void fonction(int8_t * ptr_param)
{
  int8_t tab1[10];
  int8_t tab2[100];

  int8_t *pt1=&tab1[0];
  int8_t *pt2=&tab2[0];

  ptr_param ++ ; /* on se sait pas si ptr_param  pointe sur un tableau ... */
  pt1++; /* pt1 pointe sur élément suivant de tab1 */
  ptr_param = pt1 + pt2 ; /* accès mémoire illicite */

  if (pt2>=15 ) /* pas de sens  */
  {
    /* ... */
  }
```

```
  uint8_t nb_elem = pt2 - pt1 ; /* les deux pointeurs ne sont pas sur le même
  tableau et le type est non adapté */
 ...
}
```

Bon exemple

```
#include <stddef.h>
#include <stdint.h>
void fonction(int8_t * ptr_param)
{
  int8_t tab1[10];
  int8_t tab2[100];

  int8_t *pt1 = &tab1[0];
  int8_t *pt2 = &tab2[0];

  pt1++; /*pt1 pointe sur élément suivant de tab1 */
  pt2 = pt2 + 3; /* pt2 pointe sur tab2[3] */
  pt1 = pt1 + 8 ; /* pt1 pointe sur le dernier élément de tab1 */

  if (pt1 >= tab1 ) /* même tableau  ok   */
  {
    /* ... */
  }

  ptrdiff_t nb_elem = pt2 - tab2 ; /*les deux pointeurs  sont sur
  le même tableau  et type dédié utilisé (issu de stddef.h) */
 ...
}
```

8.10.1 Références

[Misra2012] Rule 18.1 A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand
[Misra2012] Rule 18.2 Substraction between pointer shall only be applied to pointers that address elements of the same array.
[Misra2012] Rule 18.3 The relational operators shall not be applied to objects of pointer type exception where they point into a same object.
[Misra2012] Rule 18.4 The +, -, += and -= operators shall not be applied to an expression of pointer type
[Cert] Rule ARR36-C Do not substract or compare two pointers that do not refer to the same array.
[Cert] Rule ARR37-C Do not add or subtract an integer to a pointer to a non-array object.
[Cert] Rule ARR39-C Do not add or substract a scaled integer to a pointer.
[Cert] Rec. EXP08-C Ensure pointer arithmetic is used correctly.
[IsoSecu] Subtracting or comparing two pointers that do not refer to the same array [ptrobj].
[IsoSecu] Forming or using out-of-bounds pointers or array subscripts [invptr].
[Cwe] CWE-469 Use of pointer substraction to determine size.
[Cwe] CWE-468 Incorrect pointer scaling
[Cwe] CWE-466 Return of pointer value outside of expected range.
[Cwe] CWE-587 Assignment of a fixed address to a pointer.
[AnsiC99] Sections 6.2.5., 6.3.2.3., 6.5.2.1., 6.5.6.
[AnsiC90] Sections 6.1.2.5., 6.2.2.3., 6.3.6., 6.3.8.

9
Structures et unions

9.1 Déclaration de structures

Afin de modéliser une entité au sein d'un programme, il est souvent nécessaire de vouloir associer plusieurs données de type scalaire (entiers, caractères, …). La définition de variables indépendantes pour représenter cette entité rend difficile la compréhension du code et fastidieux le passage des paramètres à une fonction. Une structure doit être utilisée pour regrouper les données qui représentent une même entité. Et il est nécessaire de définir autant de structures qu'il y a d'entités à modéliser. Il ne faut pas utiliser une seule structure et y regrouper des données relatives à des entités différentes.

RÈGLE 99

RÈGLE — Une structure doit être utilisée pour regrouper les données représentant une même entité

Les données liées doivent être regroupées au sein d'une structure.

Information

Cette règle n'est pas liée à un risque de sécurité immédiat mais est une règle de bon sens à appliquer pour tous les développements.

Mauvais exemple

Dans l'exemple suivant, l'absence de structure aboutit à des prototypes de fonction difficilement compréhensibles :

```
void rectangle (float x0, float y0, float x1, float y1, float x2,
float y2, float x3, float y3);
void pyramide(float* coords);  /* coords est un tableau */
```

Bon exemple

L'exemple ci-dessous utilise bien des structures indépendantes pour représenter différentes formes géométriques :

```
  typedef struct point_s {
    float x;
    float y;
} Point_t;

typedef struct rectangle_s {
    point_t xy0;
    point_t xy1;
    point_t xy2;
    point_t xy3;
} rectangle_t;
```

```c
typedef struct pyramide_s {
    rectangle_t base;
    point_t     top;
} pyramide_t;

void rectangle (rectangle_t* rect);

void pyramide(pyramide_t* pyra);
```

9.2 Taille d'une structure

La taille d'une structure ne doit pas être supposée égale à la somme de la taille de ses éléments. Le bourrage de structure en est la cause. Il correspond à un réarrangement des champs en mémoire pour aligner proprement la structure (on parle de champs de bourrage). Pour cette raison, il ne faut pas calculer la taille d'une structure en additionnant la taille de ses champs car cela ne prend pas en compte la taille des champs de bourrage.

RÈGLE 100

RÈGLE — Ne pas calculer la taille d'une structure comme la somme de la taille de ses champs

Du fait du bourrage, la taille d'une structure ne doit pas être supposée comme la somme de la taille de ses champs.

Mauvais exemple

```c
#define SIZE_TABL 100
...
typedef struct{
  int tabl[SIZE_TABL];
  size_t size;
} ma_struct;
...
size_t sizestruct= sizeof(ma_struct.tabl)+sizeof(ma_struct.size);
/*suppose que taille de la structure est la somme de la taille des
éléments */
...
```

Bon exemple

```c
#define SIZE_TABL 100
...
typedef struct{
  int tabl[SIZE_TABL];
  size_t size;
} ma_struct;
...
size_t sizestruct = sizeof(struct); /*bonne taille  */
...
```

Attention

L'utilisation d'attributs non standard comme *packed* n'est pas considérée dans ce guide.

9.2.1 Références

[Cert] EXP42-C Do not compare padding data.
[Cert] EXP03-C Do not assume the size of a structure is the sum of the sizes of its members.
[Cert] Rule DCL39-C Avoid information leakage when passing a structure across a trust boundary.

9.3 Bitfield

Il est possible en C de spécifier la taille (en bits) des éléments d'une structure ou d'une union pour utiliser plus efficacement la mémoire en particulier. L'utilisation de *bitfield* doit s'accompagner des précautions d'usage. D'une part, un bitfield de type `int` ne sera pas obligatoirement signé. En effet, une variable `int` est bien, par défaut, signée sauf dans le cas de bitfield où le signe devient alors dépendant de l'implémentation du compilateur.

RÈGLE 101

RÈGLE — Tout bitfield doit obligatoirement être déclaré explicitement comme non signé

De plus, la représentation interne des structures avec bitfields est également dépendante de l'implémentation, il ne faut donc en aucun cas présumer de cette représentation.

RÈGLE 102

RÈGLE — Ne pas faire d'hypothèse sur la représentation interne de structures avec des bitfields

Mauvais exemple

```
typedef struct structure{
  int ok :1; /* bitfield  de taille 1 */
  int value  :7; /* bitfield dont le signe est dépendant du compilateur utilisé */
} struct_bitfield;
..
struct_bitfield s;
int *pt_s;
pt_s=(int *) &s;
s.ok=1;
..
if(s.ok==1) /* si compilé avec gcc par exemple, par défaut les bitfields sont
signés donc étant de taille 1, s.ok vaut 0 ou -1 ! */
{
  pt_s++;  /* ?  */
  *pt_s=100; /* ? */
}
```

Bon exemple

```
typedef struct structure{
  unsigned int ok :1; /* bitfield non signé  */
  unsigned int value  :7; /* bitfield non signé */
```

```
} struct_bitfield;
..
struct_bitfield s;
s.ok=1;
..
if(s.ok==1) /* plus de dépendance au compilateur */
{
   s.value=100;
}
```

9.3.1 Références

[Cert] Rule EXP11-C Do not make assumptions regarding the layout of structures with bit-fields.
[Cert] Rec. EXP12-C Do not make assumptions about the type of a plain int bit-field when used in an expression.
[Misra2012] Dir. 1.1. Any implementation-defined behaviour on which the output of the program depends shall be documented and understood.

9.4 Utilisation des FAM

Les FAM[11] ont été introduits via le C99. Cela correspond à déclarer en dernier membre d'une structure un tableau sans dimension et donc de taille flexible par nature. Si la structure associée n'est pas allouée (ou copiée) dynamiquement mais sur la pile, aucun espace est alloué pour ce tableau et y accéder provoque alors un comportement non défini.

De plus cela revient à accepter les tableaux de taille non définie ce qui est en contradiction avec la règle de la section 8.3. Les FAM sont donc prohibés.

RÈGLE 103

RÈGLE — Ne pas utiliser les FAM

9.4.1 Références

[Misra2012] Rule 18.7 Flexible array member shall not be used.
[Cert] MEM33-C Allocate and copy structures containing a flexible array member dynamically.
[Cert] Rule DCL38-C Use the correct syntax when declaring a flexible array member.

9.5 Ne pas utiliser les unions

Le langage C permet via le mécanisme d'union, d'utiliser un même espace mémoire pour stocker des données de natures différentes. Cependant, cela comporte un risque de mauvaise interpréta-

11. Flexible Array Member

tion et utilisation des données. L'utilisation du même espace pour plusieurs types de données est donc à éviter au maximum.

RECO 104

RECOMMANDATION — Ne pas utiliser les unions

L'utilisation du même espace mémoire pour plusieurs données de natures différentes n'est pas autorisée.

L'utilisation des unions doit strictement être limitée à des cas où le type est vérifié par un autre moyen et que si cela est nécessaire (pour le *parsing* de trame réseau par exemple) et cela devra être justifié en commentaire dans le code.

9.5.1 Références

[Misra2012] Rule 19.2 The union keyword should not be used.

10
Expressions

10.1 Expressions entières

Plusieurs précautions de base sont à prendre dès que des expressions manipulent des entiers.

Pour les opérations sur des entiers signés, il faut s'assurer qu'il n'y a pas de débordement de la taille du type associé et pour des opérations d'entiers non signés, qu'il n'y aura pas de bouclage (*wrap*) de valeurs.

RÈGLE 105

RÈGLE — Supprimer tous les débordements de valeurs possibles pour des entiers signés.

RECO 106

RECOMMANDATION — Détecter tous les wraps possibles de valeurs pour les entiers non signés.

Mauvais exemple

Dans la fonction suivante, aucun débordement n'est vérifié :

```
#include <stdint.h>
void f(uint8_t i, int8_t j)
{
  uint8_t ibis = i+ 2;
  int8_t j_bis = j +3;
 /* ... */
}
```

Bon exemple

Dans la fonction suivante, les débordements sont vérifiés :

```
#include <stdint.h>

void f(uint8_t i, int8_t j)
{
  uint8_t ibis ;
  int8_t j_bis ;
  if (i > (UINT8_MAX - 2))
  {
    /* erreur */
  }
  else
  {
    ibis=i+2;
```

```
  }
  if (j > (INT8_MAX - 3))
  {
    /* erreur */
  }
  else
  {
    jbis=j+3;
  }
  /* ... */
}
```

De la même façon, toutes les potentielles erreurs dues à des divisions par zéro doivent être évitées.

RÈGLE 107

RÈGLE — Détecter et supprimer toute potentielle division par zéro

Cette vérification doit être systématique pour tout calcul de division ou de reste de division.

Mauvais exemple

Dans la fonction suivante, aucune vérification sur une possible division par zéro :

```
#include <stdint.h>
void func(int8_t i, int8_t j)
{
  int8_t result ;
  result = i / j;
  ...
}
```

Bon exemple

Dans la fonction suivante, il y a bien une vérification sur une possible division par zéro :

```
#include <stdint.h>
void func(int8_t i, int8_t j)
{
  int8_t result ;
  if (0 == j)
  {
    /* erreur */
  }
  else
  {
  result = i / j;
  }
  ...
}
```

10.1.1 Références

[Cert] Rule INT30-C Ensure that unsigned integer operation do not wrap.
[Cert] Rule INT31-C Ensure that integer conversions do not result in lost or misinterpreted data.
[Cert] Rule INT32-C Ensure that operations on signed integers do not result in overflow.
[Cert] Rule INT33-C Ensure that division and remainder operations do not result in divide-by-zero errors.

[Cert] Rec. INT08-C Verify that all integer values are in range.
[Cert] Rec. INT10-C Do not assume a positive remainder when using % operator.
[Cert] Rec. INT18-C Evaluate integer expressions in a larger size before comparing or assigning to that size.
[Cert] Rec. INT16-C Do not make assumptions about representation of signed integers.
[Cwe] CWE-190 Integer overflow or wraparound.
[Cwe] CWE-682 Incorrect calculation.
[Cwe] CWE-369 Divide by Zero.
[IsoSecu] Integer division errors [diverr].

10.2 Lisibilité des opérations arithmétiques

La compréhension d'un calcul arithmétique peut s'avérer complexe si un effort n'a pas été fait concernant sa lisibilité. Par ailleurs, suivant l'écriture choisie pour le calcul, celui-ci peut s'avérer ambigu.

Une expression complexe devra être simplifiée pour aider à la compréhension. Si la complexité est pertinente (optimisation, ...), un commentaire devra expliquer et accompagner l'expression. Un exemple assez courant est d'utiliser le décalage de n bits à gauche pour une multiplication par 2^n (ou le décalage de bits à droite pour une division). Ainsi, l'expression suivante :

```
a << b ;
```

peut être utilisée afin d'effectuer l'opération $a * 2^b$. Ce genre d'expressions ne favorise pas la compréhension du code. De plus, ces décalages doivent respecter des règles précises prenant en compte le nombre de décalage de bits demandés et la taille du type concerné (cf. section 10.7). Il est recommandé de n'utiliser les décalages de bits que quand le but est justement de manipuler les bits d'un registre par exemple.

RECO 108

RECOMMANDATION – Les opérations arithmétiques doivent être écrites en favorisant leur lisibilité

Il faut utiliser des opérations arithmétiques le plus explicites possibles (naturelles) et dans la logique du programme.

Mauvais exemple

Dans l'exemple suivant, les opérations arithmétiques ne sont pas lisibles. La compréhension des opérations n'est pas immédiate :

```
/* Dans le calcul suivant, on souhaite calculer a² + 4ac + b². */
uint64_t res;
uint32_t a, b, c;
res = a * a + ((a * c) << 2) + b * b; /* une explication serait la  bienvenue */
/* a<<b est équivalent à a* 2^b mais ici le décalage de
bits est utilisé pour une multiplication */
...
/* calcul d'un masque */
uint32_t bitfield = 0xCAFEBABE;
uint32_t n, bitmask;
n = 4;
bitmask = 1;
for(n = n; n > 0; n--) {
```

```
  bitmask = 2 * bitmask;
} /* a contrario ici le décalage de bits aurait été plus logique */
bitfield = bitfield & (~bitmask);
```

Bon exemple

Le code suivant effectue des calculs en utilisant des opérations arithmétiques simples :

```
/*  calcul de  a² + 4ac + b². */
uint64_t res;
uint32_t a;
uint32_t b;
uint32_t c;
res = (a * a) + (4 * (a * c)) + (b * b); /* plus clair */
...
/* calcul d'un masque */
uint32_t bitfield = 0xCAFEBABE;
uint32_t n = 4;
uint32_t bitmask
bitmask = 2 << n;  /* manipulation de bits */
bitfield = bitfield & (~bitmask);
```

10.2.1 Références

[Cert] Rec. INT14-C Avoid performing bitwise and arithmetic operations on the same data.

10.3 Utilisation des parenthèses pour expliciter l'ordre des opérateurs

Le langage C comporte de nombreux opérateurs, avec différents niveaux de priorité quant à leur associativité. Cependant, l'absence de parenthèses dans une expression rend celle-ci difficile à comprendre et à relire.

L'utilisation systématique de parenthèses dans les calculs permet de montrer et de choisir clairement la priorité des opérations ainsi que l'ordre dans lequel le calcul est effectué.

Information

Les opérateurs du langage C et leur priorité sont présentés dans l'annexe D.

RÈGLE 109

RÈGLE – Explicitation de l'ordre d'évaluation des calculs par utilisation de parenthèses

Malgré la priorité des opérateurs, pour éviter toute ambiguïté, les expressions seront entourées de parenthèses pour rendre plus explicite l'ordre d'évaluation d'un calcul.

10.3.1 Références

[Cert] EXP10-C Rec. Do not depend on the order of evaluation of subexpressions or the order in which side effects take place.
[Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.
[Misra2012] Adv. 12.1 The precedence of operators within expressions should be made explicit
[Misra2012] Rule 13.2 The value of an expression and its persistant side effects shall be the same under all permitted evaluation orders
[Misra2012] Rule 13.5 The right hand operand of a logical `&&` or `||` operator shall not contain persistant side effects.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.4 Pas de comparaison multiple de variables sans parenthèses

Il est fréquent de vouloir vérifier la valeur d'une variable par rapport à une borne inférieure et à une borne supérieure et le raccourci de le faire en une seule instruction sans parenthèses est une erreur. Prenons l'exemple de l'expression suivante : `(0<=x<=n)`. La partie gauche, *i.e.* `0<=x`, est évaluée en premier. Le résultat de cette évaluation (0 ou 1) est ensuite comparé à la valeur bornant à droite qui sera toujours vérifié pour toute valeur de `n` supérieure ou égale à 1. L'instruction `(0<=x<=n)` est donc sémantiquement équivalente à `((0<=x)<=n)` et non à `((0<=x)&&(x<=n))`.

Une autre erreur classique est le test d'égalité combiné `if(a==b==c)` dont l'objectif est, a priori, de vérifier l'égalité des trois variables. En pratique, comme pour le cas précédent, ce test ne se comporte pas comme le développeur l'attend. En effet, cette conditionnelle ne sera vraie que si les trois variables valent `1` ou si `c` vaut `0` et que `a` et `b` sont différents.

Expression booléenne

Le langage C ne possède pas de véritable type booléen en C90, le type booléen a été introduit avec le C99. Une bibliothèque y est associée (`stdbool.h`). On parlera cependant d'expression booléenne pour les expressions du langage C, y compris avant le C99, dont le résultat de l'évaluation correspond à une valeur de vérité comme typiquement les expressions de comparaison. Une expression booléenne correspond à la valeur de vérité fausse pour une évaluation retournant la valeur 0, toute autre valeur retournée par une expression booléenne (que ce soit 1 ou une valeur négative, positive, entière ou non) correspond à la valeur de vérité vraie.

Les expressions booléennes contenant au moins deux opérateurs relationnels sont interdites sans parenthèse et doivent être décomposées soit en conditionnelles imbriquées soit en plusieurs expressions relationnelles.

RECO 110

RECOMMANDATION — Eviter les expressions de comparaison ou d'égalité multiple

RÈGLE 111

RÈGLE — Toujours utiliser les parenthèses dans les expressions de comparaison ou d'égalité multiple

Les expressions booléennes de comparaison ou d'égalité contenant au moins deux opérateurs relationnels sont interdites sans parenthèses.

Mauvais exemple

```
#define N 100
...
if (0 <= x <= N) {
  /* instruction 1 TOUJOURS exécutée  */
} else {
  /* instruction 2 JAMAIS exécutée */
}
...
if (30 < x < 40) {
  printf("pb"); /* TOUJOURS exécutée  */
}
...
```

Bon exemple

```
#define N 100
...
if ((0 <= x) && (x <= N)) { /* cas 1  : décomposition en 2 expressions
    relationnelles */
  /* instruction 1 */
} else {
  /* instruction 2 */
}
 ...
if (30 < x) { /* cas 2  : décomposition en 2 instructions conditionnelles imbriquées
     */
  if (x<40) {
    printf("pb");
 ...
}}
```

10.4.1 Références

[Misra2012] Rule 10.1 Req. Operands shann not be of an inappropriate essential type.
[Misra2012] Rule 12.1 The precedence of operators within expressions should be made explicit.
[Cert] EXP00-C Rec. Use parentheses for precedence of operation.
[Cert] EXP13-C Rec. Treat relational and equality operators as if they were nonassociative.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.5 Parenthèses autour des éléments d'une expression booléenne

Lors de l'écriture d'expressions booléennes, l'absence de parenthèses et l'utilisation exclusive de l'associativité rend la compréhension du code difficile. L'utilisation systématique des parenthèses évite les erreurs de programmation en rendant explicite l'ordre d'évaluation des opérations.

RÈGLE 112

RÈGLE – Parenthèses autour des éléments d'une expression booléenne

Il est nécessaire de toujours mettre entre parenthèses les différents éléments d'une expression booléenne, afin qu'il n'y ait aucune ambiguïté dans l'ordre d'évaluation.

Mauvais exemple

Dans l'exemple suivant, il est nécessaire de connaître l'associativité entre les opérateurs et la priorité entre ceux-ci afin de comprendre l'ordre d'évaluation :

```
if (x > 0 && y * z > length) {
    ...
}
u = z > 100 && 100 == x || x + y < z;
```

Bon exemple

Dans le code suivant, l'utilisation de parenthèses permet de connaître explicitement l'ordre d'évaluation :

```
if ((x > 0) && ((y * z) > length)) {
  ...
}
u = (z > 100) && ((100 == x) || ((x + y) < z));
```

10.5.1 Références

[Cert] EXP00-C Rec. Use parentheses for precedence of operation.
[Misra2012] Rule 12.1 The precedence of operators within expressions should be made explicit.
[Cwe] CWE-783 Operator Precedence Logic Error.

10.6 Comparaison implicite avec 0 interdite

En C90, la valeur « vrai » correspond à n'importe quelle valeur différente de 0 (que cette valeur soit négative, positive, entière ou pas) et la valeur « faux » correspond à la valeur 0. De ce fait, il est possible d'écrire des expressions booléennes où une comparaison avec 0 est effectuée implicitement. Les comparaisons implicites rendent difficile la compréhension du code et sa maintenance. Les expressions booléennes doivent utiliser un opérateur explicite de comparaison :

==, !=, <, >, <=, >=

RÈGLE 113

RÈGLE — Comparaison implicite avec 0 interdite

Toutes les expressions booléennes doivent utiliser des opérateurs de comparaison. Aucun test implicite avec une valeur égale à 0 ou différente de 0 ne doit être effectué.

RECO 114

RECOMMANDATION — Utilisation du type bool en C99

En C99, le type `bool` (ou `_Bool`) doit être utilisé pour les variables à valeurs booléennes.

En C99, l'utilisation de variable de type `bool` directement dans une conditionnelle est acceptée.

Mauvais exemple

Les comparaisons implicites dans le code suivant devraient être supprimées au profit de comparaisons explicites :

```
#define MAX 10
uint8 z;
..
while (x) {
  ...
}
if (x < y) {
  ...
} else {
  ...
}
if (!z) {  /* comparaison implicite avec 0 et z devrait être de type bool */
  ...
}
if (ptr) {
  ...
}
for (x = MAX; x; x--) {
  ...
}
```

Bon exemple

Dans l'exemple suivant, aucune comparaison implicite n'est effectuée et le fichier d'en-tête dédié est utilisé :

```
#include <stdbool.h>

bool z; /* utilisation du type bool */
while (x > 0) {
  ...
}
if (x < y) {
  ...
} else {
  ...
}
if (FALSE== z) { /* constante à gauche et comparaison explicite de z;  z étant
  de type bool,  if (! z) peut être utilisé  */
  ...
}
if (NULL != ptr) {
  ...
}
for (x = MAX; x > 0; x--) {
  ...
```

```
}
```

10.6.1 Références

[Cert] Rec. EXP20-C Perform explicit tests to determine success, true and false, and equality.

10.7 Opérateurs bit à bit

De façon générale, les opérateurs bit à bit ne doivent être employés qu'avec des expressions de types non signés. Cela permet d'éviter de nombreux comportements non définis ou *implementation-defined*.

RECO 115

RECOMMANDATION – Les opérateurs bit à bit ne doivent être appliqués qu'à des opérandes non signés

De plus, certains opérateurs bit à bit comme `&`, `|` ou `^` peuvent facilement — en particulier les deux premiers — être utilisés à tort en lieu et place d'opérateurs logiques comme `&&`, `||` et `!`. Afin d'éviter cette confusion, il est important de vérifier que les opérateurs bit à bit utilisés dans les expressions booléennes sont bien les opérateurs désirés. Des opérateurs bit à bit n'ont pas lieu d'être appliqués sur un opérande de type booléen ou assimilé.

RÈGLE 116

RÈGLE – Pas d'opérateur bit à bit sur un opérande de type booléen ou assimilé

Mauvais exemple

```
if ((var >= 0) & (var < 120)) { /* confusion des opérateurs */
 ...
if ((val && FLAG) != 0) /* confusion des opérateurs ? */
 ...
}
```

Bon exemple

```
 if ((var >= 0) && (var < 120)) { /* correction */
 ...
 if ((val & FLAG) != 0) /* utilisation correcte ici de l'opérateur bit à bit
                           dans une expression booléenne */
 ...
}
```

10.7.1 Références

[Cert] Rec. INT13-C Use Bitwise operators only on unsigned operands.
[Cert] Rec. EXP14-C Beware of integer promotion when performing bitwise operations on integer types smaller than int.
[Cert] Rule INT34-C Do not shift an expression by a negative number of bits or by greater or equal the number of bits that exist in the operand.
[Cwe] CWE-682 Incorrect calculation.
[Cert] EXP46-C Do not use a bitwise operator with a Boolean-like operand.
[Cwe] CWE-480 Use of incorrect operator.
[Misra2012] Rule 10.1 Operands shall not be of an inappropriate essential type.

10.8 Affectation et expression booléenne

L'opérateur d'affectation du langage C retourne une valeur. Il est donc possible d'exploiter cette valeur. Cependant, il s'agit souvent d'une affectation non intentionnelle du développeur résultant d'une confusion entre l'opérateur d'affectation = et l'opérateur d'égalité ==.

BONNE PRATIQUE 117

BONNE PRATIQUE — Ne pas utiliser la valeur retournée lors d'une affectation

Information

Lors de la phase de compilation avec les bonnes options (`-Wall`, ...), une alarme sera émise suggérant en particulier d'encadrer l'affectation par des parenthèses dans l'expression booléenne (option `-Wparentheses`).

RÈGLE 118

RÈGLE — Affectation interdite dans une expression booléenne

Une affectation ne doit pas être effectuée dans une expression booléenne quelle qu'elle soit. Une affectation doit être effectuée dans une instruction indépendante.

Afin de limiter les risques d'écriture d'une affectation avec l'opérateur = à la place d'une comparaison avec l'opérateur ==, lorsque la comparaison est faite entre une variable et un opérande constant, il est souhaitable d'écrire l'opérande constant comme opérande gauche de l'opérateur == et la variable comme opérande droit. En effet, le compilateur lève une erreur en cas de tentative d'affectation d'une valeur à un opérande constant.

BONNE PRATIQUE 119

BONNE PRATIQUE — Comparaison avec opérande constant à gauche

Quand une comparaison fait intervenir un opérande constant celui-ci sera de préférence mis comme opérande gauche pour éviter une affectation non intentionnelle.

Information

Cette bonne pratique est discutable d'où le fait qu'elle ne soit pas imposée mais juste conseillée. En effet, la compilation de ce type de code avec des options strictes (en particulier `-Wall`), comme exigé par la section 5.2, est suffisante pour détecter l'utilisation de l'opérateur d'affectation au lieu de l'opérateur de comparaison dans une expression booléenne.

Mauvais exemple

Le code suivant contient des affectations dans des expressions booléennes. Une des affectations dans une expression conditionnelle est une erreur de programmation.

```
if ((x = y + z) > 0) { /* affectation dans une expression booléenne et
  utilisation de la valeur retournée par l'affectation et constante à droite */
    ...
}
if (z = VALUE) { /* constante à droite et = au lieu de == */
    ...
}
```

Bon exemple

Dans l'exemple suivant, toutes les affectations sont effectuées dans des instructions indépendantes et les différents problèmes sont corrigés :

```
x = y + z;
if (0 < x) {
  ...
}
if (VALUE == z) {
  ...
}
```

10.8.1 Références

[Misra2012] Rule 13.4 The result of an assignment operator should not be used.
[Cert] Rule EXP45-C Do not perform assignments in selection statements.
[IsoSecu] No assignment in conditional expressions [boolasign].
[Cwe] CWE-480 Use of incorrect operator.
[Cwe] CWE-481 Assigning instead of comparing.
[Cwe] CWE-482 Comparing instead of assigning.

10.9 Affectation multiple de variables interdite

Le langage C autorise à l'aide d'une seule instruction d'affecter la même valeur à plusieurs variables. Cette affectation multiple est souvent utilisée pour des initialisations de variables.

Cependant, du code contenant des affectations multiples est difficile à lire et également difficile à maintenir. Décomposer l'instruction d'affectation multiple de variables en autant d'instructions d'affectation qu'il y a de variables.

RÈGLE 120

RÈGLE — Affectation multiple de variables interdite

L'affectation multiple de variables n'est pas autorisée.

Mauvais exemple

L'exemple suivant présente une affectation multiple :

```
...
a = b = c = d = 1;  /* affectation multiple à réécrire */
```

Bon exemple

Le code suivant contient autant d'instructions d'affectation qu'il y a de variables :

```
...
a = 1;
b = 1;
c = 1;
d = 1;
```

10.10 Une seule instruction par ligne de code

La fin d'une instruction dans le langage C est marquée par le point-virgule. Le langage C ne contraint pas le développeur à n'écrire qu'une seule instruction par ligne de code.

Cependant, lorsque plusieurs instructions sont présentes sur une même ligne, le code est moins lisible. Le débogage est également plus difficile, puisqu'il n'est pas possible de contrôler finement l'exécution du code instruction par instruction.

La présence de plusieurs instructions par ligne de code fausse également la métrique du nombre de lignes de code.

RÈGLE 121

RÈGLE — Une seule instruction par ligne de code

Mauvais exemple

Dans l'exemple suivant, la compréhension du code est difficile :

```
int32_t a; int64_t b;
a = 4; b = a / 6; printf("a = %d, b = %lld\n", a, b);
```

Bon exemple

```
int32_t a;
int64_t b;
a = 4;
b = a / 6;
printf("a = %d, b = %lld\n", a, b);
```

10.11 Utilisation des nombres flottants

Information

Nous parlons ici de nombres flottants par abus de langage pour représenter les nombres utilisant une représentation à virgule flottante.

La représentativité des nombres flottants en machine est une notion complexe souvent non ou mal maîtrisée et de plus, dépendant de la précision associée au type. Ces nombres flottants sont souvent source d'erreurs.

Toutes les valeurs réelles ne peuvent pas être représentées en flottants et d'autres phénomènes « non naturels » comme l'*absorption* [12] et la *cancellation* [13] peuvent survenir avec l'utilisation de nombres flottants mais ces points ne seront pas détaillés dans ce guide. De plus amples détails sont données dans la norme IEEE754[float] qui permet de garantir un comportement reproductible inter-compilateur et inter-architecture en présence de nombres flottants.

De plus l'erreur associée à l'utilisation de ces nombres flottants peut devenir plus grande que le résultat du calcul qui les utilise.

Enfin, certaines propriétés élémentaires de l'arithmétique réelle sont mises à mal lors de l'utilisation de flottants : commutativité, associativité ...

Pour toutes ces raisons, l'utilisation de nombres flottants est vivement déconseillée. Et si l'utilisation de nombres flottants s'avère nécessaire pour des développements de traitement numérique par exemple, le développeur devra s'assurer de la représentativité des valeurs flottantes constantes et la bonne utilisation de ceux-ci conformément à la précision associée.

BONNE PRATIQUE 122

BONNE PRATIQUE — Éviter les constantes flottantes

Ne pas utiliser de constantes numériques flottantes pour éviter les pertes de précision et autres phénomènes liés aux nombres flottants. Si cela ne peut être évité, la représentativité de la valeur flottante en question doit être vérifiée.

RECO 123

RECOMMANDATION — Limiter l'utilisation des nombres flottants au strict nécessaire

Il faut limiter l'utilisation des nombres flottants.

Les compteurs de boucle de type flottant sont sources d'erreurs du fait de la représentativité restreinte de ce type et de la complexité associée.

12. Phénomène sur les flottants lié à la précision tel que $GrandeValeurFlottante + EpsilonFlottant = GrandeValeurFlottante$, *i.e.* une grande valeur flottante va « absorber » une petite valeur flottante

13. Autre phénomène toujours lié à la précision et la représentativité des flottants tel que pour deux valeurs flottantes proches $ValeurFlottante1 - ValeurFlottante2 = 0$ alors que formellement $ValeurFlottante1 \neq ValeurFlottante2$

RÈGLE 124

RÈGLE — Pas de compteur de boucle de type flottant

Les compteurs de boucle doivent être uniquement de type entier, avec la vérification de non débordement de type des valeurs des compteurs.

La manipulation de valeurs flottantes dans des expressions booléennes est également très risquée toujours en lien avec les problèmes de représentabilité et de précision de ces valeurs. L'utilisation des opérateurs logiques != ou == sur des flottants est erronée dans la majorité des cas. Les résultats pourront en effet à la fois dépendre du niveau d'optimisation, du compilateur lui-même mais aussi de la plate-forme utilisée.

RÈGLE 125

RÈGLE — Ne pas utiliser de nombres flottants pour des comparaisons d'égalité ou d'inégalité

Mauvais exemple

```
float y = 0.1; /* non représentable en simple précision */
 ...
if (y == 0.1) /* comparaison de valeur flottante ET 0.1 valeur double donc
promotion de y */
   printf("egal\n");
else
   if (y == 0.1f) /* 0.1f valeur float - condition vérifiée ici */
      printf("egal2\n");
   else printf("different\n");
...
for (float x = 0.1f; x <= 1.0f; x += 0.1f{ /* la boucle se déroulera 9 ou 10
  fois */
}
```

Bon exemple

```
double y = 0.1; /* correction du type */
 ...
for (uint count = 1; count <= 10; count ++){
  float x = count/10.0f; /* 10 passages exactement dans la boucle */
}
```

10.11.1 Références

[Misra2012] Rule 14.1 A loop counter shall not have essentially floating type
[Cert] Rule FLP30-C Do not use floating-point variables as loop counters.
[Cert] Rule FLP30-C Do not use object representations to compare floating-point values.
[Cert] Rec. FLP00-C Understand the limitations of floating-point numbers.
[Cert] Rec. FLP01-C Take care in rearranging floating-point expressions.
[Cert] Rec. FLP02-C Avoid using floating-point numbers when precise computation is needed.
[Cert] Rec. FLP03-C Detect and handle floating-point errors.
[Cert] Rec. FLP04-C Check floating-point inputs for exceptional values.
[Cert] Rec. FLP05-C Do not use denormalized numbers.
[Cwe] CWE-369 Divide by Zero.

[Cwe] CWE-681 Incorrect conversion between numerical types.
[Cwe] CWE-682 Incorrect calculation.

10.12 Nombres complexes

Depuis C99, le langage C prend en charge le calcul des nombres complexes avec trois nouveaux types intégrés `double_Complex`, `long double_Complex` et `float_Complex`. Avec le fichier en-tête associé `complex.h`, ces types sont aussi accessibles via `double complex`, `long double complex` et `float complex`. De plus, les trois types imaginaires associés sont aussi supportés : `double_Imaginary`, `long double_Imaginary` et `float_Imaginary` (ou avec l'en-tête `double imaginary`, `long double imaginary` et `float imaginary`.

Ces nombres complexes reposant sur une représentation flottante, leur usage est vivement déconseillé.

RECO 126

RECOMMANDATION – Non utilisation des nombres complexes

Les nombres complexes introduits depuis C99 ne doivent pas être utilisés.

11
Structures conditionnelles et itératives

11.1 Utilisation des accolades pour les conditionnelles et les boucles

Le langage C n'impose pas de délimiter les instructions d'une conditionnelle ou d'une boucle par des accolades. Une absence d'accolades rend une conditionnelle ou une boucle moins lisible. Par ailleurs, il y a un risque d'erreur en cas de modification du code : une instruction pourrait être ajoutée en souhaitant qu'elle fasse partie de la conditionnelle, alors qu'elle va se retrouver en dehors.

RÈGLE 127

RÈGLE — Utilisation systématique des accolades pour les conditionnelles et les boucles

Ne jamais omettre les accolades pour délimiter un bloc d'instructions. Les accolades doivent être écrites pour délimiter un bloc d'instructions après les boucles (for, while, do) et les conditionnelles (if, else).

Mauvais exemple

Dans le code ci-dessous, une conditionnelle n'est pas délimitée par des accolades :

```
if (x == 0)       /* Il faut mettre les accolades même pour une seule instruction */
  printf("X = 0\n");
  /* Une instruction indentée sous le printf peut laisser penser visuellement
     que l'instruction est dans le if, alors que non. Pour une instruction de
     saut comme « goto », cela peut aboutir à ne pas exécuter une portion
     importante de code. */
  if (x != 0) {
    if (x < 0)  {
      ...
      while (x < 0) {
        x++;
        ...
      }
    } else {
      while (x > 0) {
        x--;
        ...
      }
    }
  }
  /* exemple du goto fail d'Apple */
if (err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
   goto fail;
   goto fail;
   /* autres vérifications mais non atteignables du fait du doublon goto fail sans
      accolades */
 fail  :
   /* nettoyage et libération de buffers */
```

```
    return err ;
```

Bon exemple

L'exemple ci-dessous délimite toutes les conditionnelles et les boucles avec des accolades :

```
if (0 == x) {
  printf("X = 0\n");
} else {
  if (x < 0) {
    ...
    while (x < 0) {
      x++;
      ...
    }
  } else {
    while (x > 0) {
      x--;
      ...
    }
  }
}
/* exemple du goto fail - Apple */
if (err=SSLhashSHA1.update(&hashCtx,&signedParams))!=)
  {
    goto fail;
  }
 /*  autres vérifications */
fail  :
 /* nettoyage et libération de buffers */
 return err ;
```

11.1.1 Références

[Misra2012] Rule 15.6 The body of an iteration-statement or a selection statement shall be a compound-statement.
[Cert] Rec. EXP19-C Use braces for the body of an if, for, or while statement.

11.2 Bonne construction et utilisation des instructions switch

L'instruction `switch` du langage C permet d'écrire de façon élégante la gestion de différents cas en fonction de la valeur d'une variable ou d'une expression. Cependant, cette instruction est aussi source d'erreurs lors de l'omission de l'instruction `break` : du code non souhaité peut être exécuté. En effet, les conditions successives de l'instruction `switch` n'étant pas exclusives, plusieurs cas peuvent être activés. Il faut également s'assurer qu'un traitement est effectué dans le cas où la valeur de l'expression ne correspond à aucun des cas du `switch` (cas par défaut).

RÈGLE 128

RÈGLE – Définition systématique d'un cas par défaut dans les `switch`

Un `switch-case` doit toujours contenir un cas `default` placé en dernier.

RECO 129

RECOMMANDATION — Utilisation de `break` dans chaque cas des instructions `switch`

Un `switch-case` doit par défaut toujours contenir un `break` pour chaque cas. L'absence de `break` pour éviter de dupliquer du code est tolérée mais doit être explicitée dans un commentaire.

Information

L'option `-Wimplicit-fallthrough` permet de vérifier la bonne application de cette recommandation.

Le code présent dans chaque `case` doit être simple et comporter peu d'instructions. Dans le cas où des traitements complexes doivent être effectués dans un `case`, il faut alors définir une fonction pour ces traitements et appeler cette fonction à partir du `case`.

RECO 130

RECOMMANDATION — Pas d'imbrication de structure de contrôle dans un `switch-case`

Même si le C l'autorise, l'imbrication de structures de contrôle à l'intérieur d'un `switch` est à éviter.

Enfin, il est interdit de déclarer, initialiser des variables ou d'introduire des instructions de code dans une instruction `switch` avant le premier label associé au premier cas du `switch-case`.

RÈGLE 131

RÈGLE — Ne pas introduire d'instructions avant le premier label d'un `switch-case`

Mauvais exemple

Dans l'exemple suivant, l'instruction `default` finale est manquante.

```
switch(var) {
  int i = 0; /* non autorisé */
  case VAL1 :
    ...
    break;
  case VAL2 :
    ...
  case VAL3 :
    ...
    break;
  /* absence de default et  break et absence de  commentaire pour expliquer le cas
       sur la valeur VAL2 */
}
```

Bon exemple

Dans l'exemple suivant, une instruction `break` est bien présente pour chaque cas. Et une instruction default finale permet de s'assurer qu'un traitement est effectué si la valeur ne correspondait à aucun des cas.

```
int i = 0;  /* déclaration et initialisation déplacées */
switch(var) {
```

```
    case VAL1 :
        ...
        break;
    case VAL2 :
        ...
        /* Absence volontaire de break */
    case VAL3 :
        ...
        break;
    default :
        ...
}
```

11.2.1 Références

[Misra2012] Rule 16.1 All switch statements shall be well-formed.
[Misra2012] Rule 16.2 A switch label shall only be used when the most closely-enclosing compound statement is the body of the switch statement.
[Misra2012] Rule 16.3 An unconditional break statement shall terminate every switch-clause
[Misra2012] Rule 16.4 Every switch statement shall have a default label.
[Misra2012] Rule 16.6 Every switch statement shall have at least 2 switch-clauses.
[Misra2012] Rule 16.7 A switch expression shall not have a essentially boolean type.
[Cert] Rule DCL41-C Do not declare variables inside a switch statement before the first case label.
[Cert] Rec. MSC01-C Strive for logical completeness.
[Cert] Rec. MSC17-C Finish every set of statements associated with a case label with a break statement.
[Cwe] CWE-484 : Omitted Break Statement in Switch.
[IsoSecu] Use of an implied default in a `switch` statement [swtchdflt].

11.3 Bonne construction des boucles `for`

Le langage C autorise l'ajout de plusieurs expressions dans le premier et le dernier élément d'une instruction `for`, en les séparant par une virgule. Cela permet par exemple d'initialiser plusieurs variables dans le premier élément du `for`, ou encore d'incrémenter plusieurs variables dans le troisième élément du `for`. Cependant, la présence d'expressions séparées par une virgule dans l'instruction `for` rend difficile la compréhension du code et est source d'erreurs. De plus, le séquencement d'instructions est déjà interdit dans la section 14.1. Dans le cas où d'autres variables sont à initialiser en début de boucle, leur initialisation doit être faite juste avant l'instruction `for`. Si plusieurs variables sont à incrémenter ou décrémenter, leur modification doit être faite en fin de boucle.

De plus, le langage C n'oblige pas que chaque élément (initialisation, condition d'arrêt et incrément/décrément) de l'instruction `for` soit complété. Il est possible de laisser l'initialisation vide par exemple, voire de laisser chaque élément vide ce qui aboutit à une boucle infinie. Dans le cas d'une boucle infinie, il faut préférer la forme `while (1) { ... }` à la forme `for(;;) { ... }`.

RÈGLE 132

RÈGLE — Bonne construction des boucles `for`

Chaque élément d'une boucle `for` doit être complété et contenir exactement une seule instruction. Ainsi une boucle `for` doit contenir une initialisation de son compteur, une condition d'arrêt portant sur son compteur, et une incrémentation ou décrémentation du compteur de boucle.

Mauvais exemple

Dans l'exemple suivant, la virgule est utilisée pour séparer plusieurs initialisations et modifications de variables dans le premier et troisième élément du `for`. De plus, des boucles `for` devraient être remplacées par des boucles `while() { ... }` ou `do { ... } while ();`

```
#define MAX_LOOP    10U
for(;;) {
  data = read();
  if (0 == data) {
    break;
  }
  ...
}
for (i = 0;;) { /* elements absents */
  max = some_function(i);
  i++;
  if (i >= max) {
    break;
  }
}
...
for (i = 0, a = 0; i < MAX_LOOP; i++, a += 2) {
  ...
  some_function(a);
  ...
}
```

Bon exemple

Le code ci-dessous effectue uniquement une initialisation du compteur de boucle dans le premier élément du for, et uniquement une incrémentation du compteur dans le troisième élément. Les boucles `for` contiennent bien tous ses éléments (initialisation, condition d'arrêt, incrémentation du compteur) :

```
#define MAX_LOOP    10U
for (i = 0; i < arraySize; i++) {
  ...
  dataArray[i] = some_function(i);
  ...
}
while (1) {
  data = read();
  if (0 == data) {
    break;
  }
  ...
}
i = 0;
do {
  max = some_function(i);
  i++;
}
while (i < max);
...
```

```
a = 0;
for (i = 0; i < MAX_LOOP; i++) {
  ...
  some_function(a);
  ...
  a += 2;
}
```

11.3.1 Références

[Misra2012] Rule 14.2 A for loop shall be well-formed.
[Misra2012] Rule 15.6 The body of an iteration-statement or a selection statement shall be a compound-statement.
[Cert] Rec. EXP15-C Do not place a semicolon on the same line as an if, for, or while statement.
[Cert] Rec. EXP19-C Use braces for the body of an if, for, or while statement.

11.4 Modification d'un compteur de boucle for interdite dans le corps de la boucle

La modification du compteur d'une boucle `for` à l'intérieur de la boucle rend difficile la compréhension et la maintenance du code. Par ailleurs, cela peut aboutir à une boucle infinie lorsque l'opérateur de comparaison dans la condition de la boucle est une égalité ou une inégalité. Dans le cas où la boucle accède à un tableau, il y a également un risque de débordement.

Le compteur de boucle ne doit être modifié que dans la troisième partie de la boucle `for`.

Il est courant de modifier à l'intérieur du corps de la boucle un *flag* ou une autre variable qui intervient au niveau de l'expression conditionnelle d'arrêt de la boucle for. Ce scénario doit être alors remplacé par l'utilisation d'un `break` permettant la sortie de la boucle.

RÈGLE 133

RÈGLE – Modification d'un compteur d'une boucle `for` interdite dans le corps de la boucle

Le compteur d'une boucle `for` ne doit pas être modifié à l'intérieur du corps de la boucle `for`.

Mauvais exemple

L'exemple ci-dessous présente une boucle `for` avec une modification de son compteur dans son corps. Un risque de boucle infinie est présent :

```
#define MAX_LOOP 10U

for (i = 0; i != MAX_LOOP; i++) {
  ...
  if (...) {
    /* Si la condition est remplie avec i == 9,
    on aboutit à une boucle infinie. */
    i++;
```

```
  }
  ...
}
```

Bon exemple

Dans le code suivant, aucune modification du compteur n'est faite dans le corps de la boucle `for` :

```
#define MAX_LOOP 10U

for (i = 0; i < MAX_LOOP; i++) {
  ...
  if (some_function()) {
      break; /* arrêt de la boucle */
  }
  ...
}
```

11.4.1 Références

[Misra2012] Rule 14.2
[Cert] Rule ARR30-C Do not form or use out-of-bounds pointers or array subscripts.

12
Sauts dans le code

12.1 Non utilisation de goto arrière (backward goto)

L'utilisation d'un goto arrière rend la relecture et la maintenance du code très difficile et est source d'erreurs comme des boucles infinies non voulues. Dans le cas où ce besoin apparaît, c'est que l'algorithme à implémenter comporte en fait une boucle. Utiliser alors les structures de contrôle proposées par le langage pour les boucles, c'est-à-dire `for`, `while` ou `do-while`.

RÈGLE 134

RÈGLE — Non utilisation de goto arrière (backward goto)

Proscrire, au sein d'une fonction, l'utilisation d'instructions goto renvoyant vers un label qui est placé avant cette instruction goto.

Mauvais exemple

L'exemple suivant contient un goto arrière. Ce choix d'implémentation correspond à une approche assembleur, et ne tire pas partie des possibilités de plus haut niveau proposées par le langage C.

```
#define BUFFER_SIZE 100U
void foo() {
  uint8_t s[BUFFER_SIZE];
  uint8_t x = 0;
  my_loop :
    s[x] = x;
  x++;
  if(x < BUFFER_SIZE)
    {
      goto my_loop;        /* backward goto interdit */
    }
}
```

Bon exemple

L'exemple suivant utilise une structure de contrôle de type boucle. Il n'est pas nécessaire de faire un backward goto.

```
#define BUFFER_SIZE 100U
void foo() {
  uint8_t s[BUFFER_SIZE];
  uint8_t x = 0;
  for(x = 0; x < BUFFER_SIZE; x++) {
    s[x] = x;
  }
}
```

12.1.1 Références

[Misra2012] Rule 15.1 The goto statement should not be used.

12.2 Utilisation limitée du goto avant (forward goto)

Le *forward* `goto` peut permettre de simplifier la gestion des erreurs, et diminuer le nombre de points de sortie d'une fonction. Cependant, dans le cas d'un *forward* `goto` situé en dehors d'une instruction conditionnelle, du code peut être exécuté avec des variables qui n'ont pas été initialisées. Une autre conséquence grave possible est par exemple l'oubli de la libération mémoire ou de ressources entre autres. Il faut donc modifier le code afin d'utiliser des structures de contrôles évitant l'utilisation du goto.

Le *forward* `goto` ne doit être utilisé que dans le cadre de la gestion d'erreur, et le nombre de labels doit être limité au strict minimum.

RECO 135

RECOMMANDATION — Utilisation limitée du saut avant (forward goto)

L'utilisation d'un *forward* `goto` est tolérée uniquement dans les cas où elle permet :

- de limiter significativement le nombre de points de sortie de la fonction ;
- de rendre le code beaucoup plus lisible.

Le ou les labels référencés par les instructions `goto` doivent tous être situés en fin de fonction.

Bon exemple

Le code ci-dessous ne fait pas usage du *forward* `goto`, mais utilise les structures de contrôle proposées par le langage C.

```
#define BUFFER_LEN    (128U)
int32_t my_function(int32_t a) {
  FILE* f = NULL;
  uint8_t *buffer = NULL;
  int32_t result = ERR_UNDEFINED;
  f = fopen("/my/path", "r");
  if(NULL == f) {
    result = ERR_FOPEN;
  } else {
    buffer = (uint8_t *) malloc(BUFFER_LEN * sizeof(uint8_t));

    if(NULL == buffer) {
      result = ERR_MALLOC;
    }
    else
    {
      ...
      free(buffer);
      buffer = NULL;
      result = ERR_NOERROR;
    }

    fclose(f);
    f = NULL;
  }
```

```
  return result;
}
```

Exemple toléré

L'exemple suivant utilise le *forward* `goto` pour une gestion d'erreurs. Il s'agit d'un cas toléré.

```
int32_t my_function(int32_t a) {
  FILE* f = NULL;
  uint8_t *buffer = NULL;
  int32_t result = ERR_UNDEFINED;
  f = fopen("/my/path", "r");
  if(NULL == f) {
    result = ERR_FOPEN;
      goto cleanup;
  }
  buffer=(uint8_t *)malloc(BUFFER_LEN * sizeof(uint8_t));
  if(NULL == buffer) {
    result = ERR_MALLOC;
    goto cleanup;
  }
  ...
  result = ERR_NOERROR;
cleanup :
    if(NULL != f) {
        fclose(f);
        f = NULL;
      }
  if(NULL != buffer) {
    free(buffer);
    buffer = NULL;
  }
  return result;
}
```

12.2.1 Références

[Misra2012] Rule 15.1 15.5

13
Fonctions

13.1 Déclaration et définition correctes et cohérentes

Déclaration/Prototype de fonction

La déclaration d'une fonction ou son prototype est une instruction dans laquelle le développeur définit trois éléments : le type de retour de la fonction, son nom et la liste de ses arguments suivis par un point virgule.

Définition de fonction

La définition d'une fonction est le corps de celle-ci *i.e.* l'ensemble des instructions qu'elle exécute. Une définition de fonction contient également un prototype de la fonction.

Le C90 autorise la déclaration implicite de fonctions que ce soit l'absence de type de retour ou l'absence de déclaration de la fonction. Le C99 est plus strict et impose au moins un spécificateur de type.

Le langage C, dans ses versions successives, propose différentes formes pour la déclaration des fonctions. Combiner ces différentes formes de déclaration de fonction est déconseillé car le risque est d'aboutir à une analyse du code beaucoup moins précise et d'engendrer des problèmes lors de l'édition de liens.

Information

Les compilateurs émettent une alarme (`-Wimplicit-function-declaration` ou `-Wimplicit-int` ou `-Wreturn-type`) mais supposent un type implicite `extern int` *i.e.* une fonction non associée à un type de retour est par défaut de type entier.

RÈGLE 136

RÈGLE — Toute fonction (non static) définie doit posséder une déclaration/prototype de fonction

RÈGLE 137

RÈGLE — Le prototype de déclaration d'une fonction doit concorder avec sa définition

Les types des paramètres utilisés pour la définition et la déclaration d'une fonction doivent être les mêmes.

RÈGLE 138

RÈGLE — Toute fonction doit être associée à un type de retour et à une liste de paramètres explicites

Chaque fonction est définie explicitement avec un type de retour. Les fonctions sans valeur de retour doivent être déclarées avec un paramètre du type `void`. De la même façon, une fonction sans paramètre devra être définie et déclarée avec `void` en argument.

L'activation des avertissements du compilateur permet de connaître les fonctions qui ne sont pas correctement déclarées (type de retour absent, incohérence des types entre la déclaration et la définition).

Mauvais exemple

Dans l'exemple ci-dessous, le type de retour est manquant pour une déclaration, une fonction sans paramètre n'est pas correctement déclarée et il existe une incohérence entre la déclaration et la définition.

```
 /* header.h */
foo(uint8_t a);  /* il manque le type de retour */
uint32_t bar(uint16_t b);
void car();    /* il manque void en type de paramètre pour indiquer que la
fonction ne prend pas de paramètres */
/* file.c */
foo(uint8_t a) {
    ...
}
uint32_t bar(int32_t b) { /* après la déclaration, b doit être un  uint16_t */
    ...
}
void car() {
    ...
}
```

Bon exemple

L'exemple suivant effectue une déclaration et une définition correcte des fonctions :

```
/* header.h */
void foo(uint8_t a);
uint32_t bar(uint16_t b);
void car(void);
/* file.c */
void foo(uint8_t a) {
    ...
}
uint32_t bar(uint16_t b) {
    ...
}
void car(void) {
    ...
}
```

13.1.1 Références

[Misra2012] Rule 8.1 Types shall be explicitly specified
[Misra2012] Rule 8.2 Function types shall be in prototype form with named parameters
[Misra2012] Rule 8.3 All declarations of an object or function shall use the same names and type qualifiers
[Misra2012] Rule 17.3 A function shall not be declared implicitly
[Misra2012] Rule 17.5 The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
[Cert] Rec. DCL07-C Include the appropriate type information in function declarators
[Cert] Rec. DCL20-C Explicitly specify void when a function accepts no arguments
[Cert] Rule DCL31-C Declare identifier before using them
[Cwe] CWE-628 Function call with incorrectly specified arguments
[IsoSecu] Using a tainted value as an argument to an unprototyped function pointer [taintnoproto].
[IsoSecu] Calling functions with incorrect arguments [argcomp].

13.2 Documentation des fonctions

La documentation incomplète d'une fonction risque d'engendrer des erreurs de programmation. Cela comprend la fonctionnalité de la fonction, une description précise des paramètres justifiant les passages par valeur ou référence mais aussi les conditions à vérifier pour le bon usage de la fonction.

Passage de paramètre par valeur ou copie

Lors d'un passage de paramètre par valeur (ou par copie), la valeur de l'argument réel à l'appel de la fonction est transmise (copiée) à l'argument formel respectif de la fonction appelée. La conséquence directe est que toute modification apportée à un argument formel n'est pas propagée à l'argument réel.

Passage de paramètre par référence ou par pointeur

Lors d'un passage de paramètre par référence, l'adresse de la valeur de l'argument réel à l'appel de la fonction est transmise à l'argument formel respectif de la fonction appelée. La conséquence directe est que toute modification apportée à un argument formel sera propagée à l'argument réel.

RECO 139

RECOMMANDATION — Documentation des fonctions

Tous les fonctions doivent être documentées. Cela comprend :

- une description de la fonction et du traitement effectué ;
- la documentation de chaque paramètre, le sens du paramètre (en entrée, en sortie, en entrée et en sortie) et les éventuelles conditions existant sur celui-ci ;
- les valeurs de retour possibles doivent être décrites.

Cela comprend également les conditions de bonne utilisation de la fonction à préciser dans son prototype surtout en cas de code portable (Linux, Windows).

RECO 140

RECOMMANDATION — Préciser les conditions d'appel pour chaque fonction

13.3 Validation des paramètres d'entrée

Des erreurs de programmation peuvent engendrer le passage de valeurs non valides en paramètres de fonction. En l'absence de contrôle de la validité des paramètres, le comportement de la fonction est non défini.

Il faut donc vérifier :

- la cohérence des adresses (non nulles, alignement, …) ;
- les valeurs des paramètres (plages de valeurs, …).

Un traitement générique doit être appliqué comme le retour d'un code d'erreur par exemple.

RÈGLE 141

RÈGLE — La validité de tous les paramètres d'une fonction doit systématiquement être remise en cause

Cela inclut :

- la validité des adresses pour les paramètres de type pointeur doit être vérifiée (non nulles, alignement des adresses conforme, etc.) ;
- l'appartenance des paramètres à leur domaine doit être vérifiée.

Cela s'applique aux fonctions définies par le développeur (cf. section 13.2) mais aussi aux fonctions de la bibliothèque standard.

Mauvais exemple

Dans l'exemple suivant, la validité des paramètres n'est pas vérifiée :

```
double division(int32_t n, int32_t d) {
  /* d != 0 non vérifié */
  return ((double)n) / ((double)d);
}
```

Bon exemple

Exemple 1 :

Le code ci-dessous présente un exemple pour lequel la validité des paramètres est vérifiée :

```
double division(int32_t n, int32_t d) {
  double res = 0.0;
  if(0 == d) {
```

```
    /* gestion erreur */
  }
  else {
    res = ((double)n) / ((double)d);
  }
  return res;
}
```

Exemple 2 :
Le code ci-dessous présente un second exemple pour lequel la validité des paramètres est vérifiée :

```
uint8_t encrypt(uint8_t *output, int32_t *output_len,
                const uint8_t *input, const int32_t input_len,
                encrypted_ctx *ctx) {
    uint8_t err = 0;
    if((NULL == output) || (NULL == output_len) || (NULL == encrypted_ctx)) {
        err++; /* gestion erreur */
    }
    if((NULL == input) || (input_len <= 0) || (input_len > MAX_INPUT_LEN)) {
        err++; /* gestion erreur */
    }
    if(0 == err) {
        /* code API */
    }
    return err;
}
```

13.3.1 Références

[Misra2012] The validity of values passed to library shall be checked.
[Misra2012] Dir 4.1 Run-time failures shall be minimized
[Cert] API00-C Functions should validate their parameters
[Cert] Rule ARR38-C Guarantee that library functions do not form invalid pointers.
[Cert] Rec. MEM10-C Define and use a pointer validation function.
[Cwe] CWE-20 Insufficient input validation
[Cwe] CWE-628 Function call with incorrectly specified arguments.
[Cwe] CWE-686 Function call with incorrect argument type.
[Cwe] CWE-687 Function call with incorrectly specified argument value.
[IsoSecu] Calling functions with incorrect arguments [argcomp].

13.4 Utilisation du qualificatif `const` pour les paramètres de fonction de type pointeur

Lors de la lecture d'un prototype de fonction avec des paramètres de type pointeur, l'absence du qualificatif `const` peut laisser penser qu'une modification va être faite sur la zone mémoire pointée. L'absence de ce qualificatif alors qu'il devrait être utilisé rend la définition des interfaces peu claire et ajoute une difficulté à la relecture de code. Lors de la déclaration d'une fonction avec des pointeurs en paramètres, le développeur doit immédiatement s'interroger sur l'usage qui va être fait des pointeurs et utiliser `const` par défaut sauf si la zone mémoire pointée est modifiée lors de l'exécution de la fonction.

RÈGLE 142

RÈGLE – Les paramètres de fonction de type pointeur pour lesquels la zone mémoire pointée n'est pas modifiée doivent être déclarés comme `const`

Marquer `const` tous les paramètres de type pointeur d'une fonction qui pointent vers une zone mémoire qui ne doit pas être modifiée dans le corps de celle-ci. Le qualificatif `const` doit s'appliquer à l'objet pointé.

Mauvais exemple

L'exemple ci-dessous devrait utiliser `const` pour son paramètre :

```
uint32_t foo(uint32_t *val) {
  /* val lue */
  uint32_t ret = 0;
  if(TEST_VALUE > *val) {
      ret = (*val) * 2;
  } else {
      ret = (*val);
  }
  return ret;
}
```

Bon exemple

Dans l'exemple suivant, `const` est bien utilisé pour le paramètre pointeur. En effet la zone mémoire pointée n'est pas modifiée dans le corps de la fonction :

```
uint32_t foo(const uint32_t *val) {
  uint32_t ret = 0;
  if (NULL != val){
    if(TEST_VALUE > *val) {
      ret = (*val) * 2;
    } else {
      ret = (*val);
    }
  }
  return ret;
}
```

13.4.1 Références

[Cert] Rec. DECL00-C Const-qualify immutable objects.
[Cert] Rec DECL13-C Declare function parameters that are pointers to values not changed by the function as const.
[Cert] Rule EXP40-C Do not modify constants objects.
[Misra2012] Rule 8.13 A pointer should point to a const-qualified type whenever possible.
[Cwe] CWE-20 Improper Input Validation.
[Cwe] CWE-369 Divide by Zero.

13.5 Utilisation des fonctions inline

Le C99 a introduit le nouveau mot-clé *inline* comme spécificateur de fonction. Une fonction *inline* déclarée avec des liens externes mais qui n'est pas définie dans le même fichier entraîne un com-

portement non défini. Il faut donc que la déclaration et la définition d'une fonction *inlinée* soient dans la même unité de compilation.

Information

Une fonction *inline* peut être accessible à plusieurs fichiers en étant déclarée dans un fichier d'en-tête.

RÈGLE 143

RÈGLE — Les fonctions inline doivent être déclarées comme `static`

Pour éviter un comportement non défini, une fonction *inline* est systématiquement `static`.

13.5.1 Références

[Misra2012] Rule 8.10 An inline function shall be declared with the static storage class.
[Cert] Rec. DCL15-C Declare file-scope objects or functions that do not need external linkage as static.
[Cert] MSC40-C Do not violate constraints.

13.6 Redéfinition de fonctions

Un nom de fonction peut être déclaré par le programmeur alors qu'il s'agit d'un nom déjà défini dans la bibliothèque standard ou dans une autre bibliothèque. Cette déclaration risque d'engendrer une confusion. Toute fonction doit porter un nom qui lui est propre.

RÈGLE 144

RÈGLE — Interdiction de redéfinir les fonctions ou macros de la bibliothèque standard ou d'une autre bibliothèque

Les identifiants, macros ou noms de fonctions faisant partie de la bibliothèque standard ou d'une autre bibliothèque utilisée ne doivent pas être redéfinis.

Mauvais exemple

Dans l'exemple ci-dessous, une confusion va se produire du fait de l'utilisation d'un nom de fonction déjà existant dans la bibliothèque standard.

```
/* ne pas réutiliser le nom de la bibliothèque standard */
void* malloc (size_t taille);
```

Bon exemple

L'exemple ci-dessous définit un nom qui n'entre pas en collision avec le nom d'une fonction de la bibliothèque standard.

```
/* renommage de la fonction */
void* mymalloc (size_t taille);
```

13.6.1 Références

[Misra2012] Rule 5.8 Identifiers that define objects or functions with external linkage shall be unique.
[Misra2012] Rule 5.9 Identifiers that define objects or functions with internal linkage shall be unique.

13.7 Utilisation obligatoire de la valeur de retour d'une fonction

Une fonction, qui n'est pas de type `void`, retourne une valeur indiquant le succès ou l'échec du traitement ou la valeur calculée par cette fonction. Ces retours de fonctions sont une source très importante d'information et permettent d'identifier au plus tôt des comportements non attendus voire des erreurs. Il faut donc toujours lire et gérer ces retours de fonctions.

La fonction appelante doit tester la valeur retournée par la fonction afin de s'assurer de sa validité par rapport à la documentation de l'interface (valeur retournée dans le domaine de valeurs ou valeur retournée correspondant à un code de succès ou d'erreur).

RÈGLE 145

RÈGLE – La valeur de retour d'une fonction doit toujours être testée

Lorsqu'une fonction retourne une valeur, la valeur retournée doit être systématiquement testée.

Mauvais exemple

Dans le code ci-dessous, la valeur retournée par la fonction n'est pas testée et aucun traitement n'est effectué dans le cas où une erreur s'est produite :

```
struct stat o_stat_buffer;
stat("somefile.txt", &o_stat_buffer);
/* le succès de la fonction stat non testé */
/* suite du programme*/
...
```

Bon exemple

Dans le code suivant, la valeur retournée par la fonction est bien testée :

```
struct stat o_stat_buffer;
uint8_t          i_result = 0;
i_result = stat("somefile.txt", &o_stat_buffer);
if (0 != i_result) {
    /* erreur  */
    return 0;
}
/* suite du programme */
...
```

13.7.1 Références

[Cert] Rec. EXP12-C Do not ignore values returned by functions.
[Misra2012] Dir. 4.7 : If a function returns error information, then that error information shall be tested.
[Misra2012] Rule 17.7 The value returned by a function having non-void return type shall be used.
[Cwe] CWE-252 Unchecked Return Value.
[Cwe] CWE-253 Incorrect Check of Function Return Value.
[Cwe] CWE-754 Improper check for unusual or exceptional conditions.

13.8 Retour implicite interdit pour les fonctions de type non `void`

En l'absence de valeur de retour explicite pour tous les chemins d'une fonction retournant une valeur (fonction non `void`), certains compilateurs ne génèrent pas toujours d'erreur. Le comportement du code est alors indéfini. Certains compilateurs retournent une valeur arbitraire.

RÈGLE 146

RÈGLE — Retour implicite interdit pour les fonctions de type non `void`

Tous les chemins d'une fonction non `void` doivent retourner une valeur explicitement.

Mauvais exemple

Dans l'exemple ci-dessous, il existe des chemins qui ne retournent pas explicitement une valeur :

```
uint32_t encr_data(const uint8_t *p_data, uint32_t ui32_data_len,
                   uint8_t **pp_encrypted_data, uint32_t *ui32_encrypted_data_len)
                       {
  uint8_t *p_encrypted_data = NULL;
  if (NULL != p_data
      && NULL != pp_encrypted_data
      && NULL != ui32_encrypted_data_len) {
      if (ui32_data_len > 0) {
         p_encrypted_data = (uint8_t *)calloc(ui32_data_len, sizeof(uint8_t));
         ...
         return 1;
       }
    }
    /* retour implicite */
  }
```

Bon exemple

Dans le code suivant, le code de la fonction retourne toujours explicitement une valeur :

```
uint32_t encr_data(const uint8_t *p_data, uint32_t ui32_data_len,
                   uint8_t **pp_encrypted_data, uint32_t *ui32_encrypted_data_len)
                       {
  uint32_t ui32_result_code = 0;
```

```
  uint8_t *p_encrypted_data = NULL;
  if (NULL == p_data
  || NULL == pp_encrypted_data
  || NULL == ui32_encrypted_data_len) {
    ui32_result_code = 0;
    goto End;
  }
  if (0 == ui32_data_len) {
    ui32_result_code = 0;
    goto End;
  }
  p_encrypted_data = (uint8_t *)calloc(ui32_data_len, sizeof(uint8_t));
  ...
  (*pp_encrypted_data) = p_encrypted_data;
  ui32_result_code = 1;
End :
  return ui32_result_code;
}
```

13.8.1 Références

[Misra2012] Rule 17.4 All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
[Cert] Rule MSC37-C Ensure that control never reacher the end of a non-void function.

13.9 Pas de passage par copie de structure en paramètre de fonction

Il est possible avec le langage C de passer en paramètre d'une fonction des paramètres de type structure. Ceux-ci sont alors copiés sur la pile. Mais cela nuit aux performances et augmente le risque de débordement de pile voire de fuite d'information pour des données sensibles.

Le paramètre correspondant à une structure doit être passé sous la forme d'un pointeur qualifié par `const`. Seule l'adresse de la structure est alors copiée sur la pile. Et le modificateur `const` évite les modifications de l'objet pointé (ce qui est souhaité lors du passage de la structure par copie).

RÈGLE 147

RÈGLE – Les structures doivent être passées par référence à une fonction

Il ne faut pas passer de paramètres de type structure par copie lors de l'appel d'une fonction.

Mauvais exemple

Dans l'exemple suivant, le paramètre est passé par copie au lieu de le passer par adresse :

```
#define STR_SIZE 20U
typedef struct
{
  unsigned char surname[STR_SIZE];
  unsigned char firstname[STR_SIZE];
} person_t;
```

```
uint32_t add_person(person_t person) {
  size_t   sz_surname_len = 0;
  size_t   sz_firstname_len = 0;
  sz_surname_len = strlen(person.surname);
  sz_firstname_len = strlen(person.firstname);
  if (0 != sz_surname_len && 0 != sz_firstname_len) {
    ...
    ui32_result = 1;
  } else {
    ui32_result = 0;
  }
  return ui32_result;
}
...
void some_function() {
  person_t person;
  ...
  add_person(person);
  ...
}
```

Bon exemple

Le code ci-dessous effectue bien le passage d'un paramètre de type structure à l'aide d'un pointeur :

```
#define STR_SIZE 20U
typedef struct
{
  unsigned char surname[STR_SIZE];
  unsigned char firstname[STR_SIZE];
} person_t;

uint32_t add_person(const person_t *person) {
  uint32_t ui32_result = 0;
  size_t   sz_surname_len = 0;
  size_t   sz_firstname_len = 0;
  if (NULL != person) {
    sz_surname_len = strlen(person->surname);
    sz_firstname_len = strlen(person->firstname);

    if (0 != sz_surname_len && 0 != sz_firstname_len) {
      ...
      ui32_result = 1;
    } else {
      ui32_result = 0;
    }
  }
  else {
      ui32_result = 0;
  }
  return ui32_result;
}

void some_function() {
  person_t person;
  ...
  add_person(&person);
  ...
}
```

13.10 Passage d'un tableau en paramètre d'une fonction

Lorsqu'une fonction prend en paramètre un pointeur, il n'est pas possible de déterminer si le pointeur est l'adresse du premier élément d'un tableau ou si celui-ci pointe sur un unique élément.

Afin de lever l'ambiguïté dans la déclaration d'une fonction, il est préférable d'utiliser la forme avec [] pour un paramètre de type tableau comme indiqué dans la sous section 8.1.

RECO 148

RECOMMANDATION – Passage d'un tableau en paramètre d'une fonction

Il existe plusieurs façons de passer un tableau en paramètre d'une fonction. Lors du passage par pointeur, il faut préciser dans la documentation de la fonction que le paramètre correspond à un tableau et également utiliser la notation dédiée aux tableaux.

Attention

Pour un tableau multidimensionnel, seule la première dimension du tableau peut rester non définie lors du passage en paramètre ce qui implique donc de définir les dimensions suivantes. Par exemple, pour un tableau à deux dimensions, utiliser `tab[][]` en paramètre est une erreur, il faudra au minimum préciser la deuxième dimension.

Mauvais exemple

L'exemple suivant présente un prototype de fonction avec un paramètre de type pointeur. Il s'agit d'un tableau passé en paramètre mais rien ici ne permet de le deviner. Dans cet exemple, `tab` peut :

- soit être un entier passé par adresse
- soit être un tableau d'entiers.

```
void func(int32_t *tab, uint32_t count);
```

Bon exemple

Dans ce second exemple, la notation et les commentaires permettent de déterminer immédiatement que le paramètre est bien un tableau.

```
void func(int32_t tab[], uint32_t count); /* tab est un tableau de count éléments
    */
```

13.11 Utilisation obligatoire dans une fonction de tous ses paramètres

La non utilisation d'un paramètre dans l'implémentation d'une fonction est généralement une erreur du développeur. Par ailleurs, cela consomme inutilement de l'espace sur la pile.

Le prototype de la fonction doit être modifié dans le cas où le paramètre n'est pas utile.

Cependant, dans certains cas, la non utilisation d'un paramètre (ou plus) d'une fonction peut être justifiée :

- la fonction correspond à une fonction de rappel dont le prototype est imposé ;

- pour des raisons de compatibilité avec du code existant, lors de l'évolution d'une bibliothèque. Un paramètre précédemment utilisé ne l'est plus ;
- dans le cas d'une évolution future dans laquelle le paramètre sera utilisé.

Dans tous ces cas, un commentaire doit alors indiquer explicitement pourquoi le paramètre est ignoré.

RECO 149

RECOMMANDATION — Utilisation obligatoire dans une fonction de tous ses paramètres

Tous les paramètres présents dans le prototype de la fonction doivent être utilisés dans son implémentation.

Information

L'option *-Wunused-parameter* permet d'alerter sur ce genre de scénario.

Mauvais exemple

Dans l'exemple suivant, un paramètre de la fonction n'est pas utilisé et devrait donc être supprimé :

```
uint32_t compute_data(uint32_t ui32A, uint32_t ui32B, uint32_t ui32C) {
  uint32_t ui32_result = 0;
  ui32_result = 2 * ui32A + 2 * ui32B;
  return ui32_result;
}
```

Bon exemple

Dans le code ci-dessous, il n'y a pas de paramètre non utilisé dans l'implémentation de la fonction :

```
uint32_t compute_data(uint32_t ui32A, uint32_t ui32B) {
  uint32_t ui32_result = 0;
  ui32_result = 2 * ui32A + 2 * ui32B;
  return ui32_result;
}
```

13.11.1 Références

[Misra2012] Rule 2.7 There should be no unused parameters in functions.
[Cert] Rule EXP37-C Call functions with the correct number and type of arguments.

13.12 Fonctions variadiques

Les fonctions variadiques (*i.e.* à nombre variables d'arguments ou dont les types peuvent varier) peuvent poser plusieurs problèmes. Il est déconseillé de définir des fonctions variadiques mais la bibliothèque standard en contient elle-même plusieurs qui sont souvent utilisées. Le type des

arguments d'une fonction variadique n'est pas vérifié par le compilateur, par défaut, ce qui peut, en cas de mauvaise utilisation de ces fonctions, entraîner quelques surprises comme des terminaisons anormales ou des comportements inattendus.

Information

L'option `-Wformat=2`, imposée par la sous-section 5.3.1, permet d'étendre la vérification du compilateur aux arguments des fonctions variadiques.

Quand `NULL` est passé à une fonction classique, `NULL` est transtypé dans le bon type. Ce transtypage ne fonctionne pas avec les fonctions variadiques puisque le « bon type » n'est pas connu. En particulier, le standard permet que `NULL` soit une constante entière ou une constante pointeur ainsi sur des plateformes où `NULL` est également une constante entière, le passage de `NULL` à des fonctions variadiques peut entraîner un comportement indéfini.

RÈGLE 150

RÈGLE — Ne pas appeler de fonctions variadiques avec `NULL` en argument

Mauvais exemple

```
...
unsigned char *string = NULL;
printf("%s %d\n", string, 1); /* comportement non défini */
...
```

Bon exemple

```
...
unsigned char *string = NULL;
printf("%s %d\n", (string ? string  :"null"), 1); /* pas de passage de NULL  */
...
```

13.12.1 Références

[Misra2012] The features of `<stdarg.h>` shall not be used. [Cert] Rec. DCL10-C Maintain the contract between the writer and the caller of variadic functions.
[Cert] Rec. DCL11-C Understand the type issues associated with variadic functions.
[Cert] Rule EXP47-C Do not call va_arg with an argument of the incorrect type.
[Cert] Rule MSC39-C Do not call va_arg on a va_list that has an indeterminate value.
[Cwe] CWE-628 Function call with incorrectly specified arguments.
[IsoSecu] Calling functions with incorrect arguments [argcomp].

14
Opérateurs sensibles

14.1 Utilisation de la virgule interdite pour le séquencement d'instructions

La virgule doit être utilisée comme séparateur des paramètres d'une fonction ou comme séparateur pour l'initialisation des champs d'une structure ou d'un tableau. L'utilisation de la virgule est également tolérée lors d'une déclaration, sous respect des autres règles autour des déclarations multiples. En revanche, l'utilisation de la virgule pour enchaîner des instructions dans le langage C rend le code peu lisible, et peut aboutir à un résultat différent de celui escompté.

RÈGLE 151

RÈGLE — Usage de la virgule interdit pour le séquencement d'instructions

La virgule n'est pas autorisée dans le cadre du séquencement des instructions de code.

Il faut remplacer la virgule par un point-virgule pour le séquencement des instructions, cela implique que :

- des accolades deviennent nécessaires ;
- les paramètres de boucles `for` doivent être réorganisés.

Mauvais exemple

L'exemple suivant fait usage de la virgule dans des expressions. Il n'est pas possible de savoir, à la lecture du code, le résultat de ces instructions.

```
int32_t i = (j = 2, 1);
y = x ? (a++, a + 4)  :c;
z = 3 * b + 2, 7 * c + 42;
a = (b = 2, c = 3, d = 4);
for(i = 0, j = SZ_MAX; i < SZ_MAX; i++, j--) {
  ...
}
```

Bon exemple

Dans le code ci-dessous, la virgule n'est utilisée que pour la déclaration de variables.

```
int32_t i, j;
i=1;
j=2;
if (0 != x) {
  a++;
```

```
    y = a + 4;
} else {
    y = c;
}
z = 3 * b + 2 ;
j = SZ_MAX;
for(i = 0 ; i < SZ_MAX; i++) {
    ...
    j--;
}
```

14.1.1 Références

[Misra2012] Rule 12.3 : The comma operator shall not be used.

14.2 Utilisation des opérateurs pré/post-fixes ++ et −− et des opérateurs composés d'affectation

Lorsque les opérateurs pré/post-fixes ++ et−− sont utilisés à l'intérieur d'un calcul, il est très difficile d'établir le résultat du calcul lors de l'analyse du code. Par ailleurs, c'est également une source de confusion voire d'erreurs pour le développeur. Ces opérateurs doivent donc être utilisés seuls dans une instruction. Par conséquent, les opérateurs pre/post-fixes étant sémantiquement équivalents quand ils sont utilisés isolément dans une instruction (*i.e.* `i++;` `++j;`), seuls les opérateurs post-fixes sont autorisés pour éviter toute ambiguïté [14]. Les opérateurs pré-fixes ne seront quant à eux pas utilisés.

RECO 152

RECOMMANDATION — Les opérateurs pré-fixes ++ et −− ne doivent pas être utilisés

Les opérateurs de pré-incrémentation et pré-décrémentation ne seront pas utilisés.

Enfin, les instructions complexes seront décomposées en éléments simples.

RECO 153

RECOMMANDATION — Pas d'utilisation combinée des opérateurs post-fixes avec d'autres opérateurs

Les opérateurs de post-incrémentation et de post-décrémentation ne doivent pas être mixés avec d'autres opérateurs.

Pour finir et toujours pour des raisons de lisibilité, il est recommandé de ne pas utiliser d'opérateurs d'affectations combinés (`>>=`, `&=`, `*=`,...).

RECO 154

RECOMMANDATION — Éviter l'utilisation d'opérateurs d'affectation combinés

14. Le choix des opérateurs post-fixes peut être discuté puisque les deux opérateurs utilisés isolément sont équivalents.

Mauvais exemple

Le code ci-dessous utilise des opérateurs post-fixes mêlés à d'autres opérateurs. Le comportement de ce code n'est pas spécifié. Il dépend du compilateur utilisé :

```
#define TAB_SIZE 25U

...

uint32_t x;
uint8_t  b[TAB_SIZE] = { 0 };
uint16_t i = 0;
x = foo(i++, i);  /* non spécifié  :problème avec l'ordre d'évaluation des
    paramètres */

...

uint32_t foo(uint16_t a, uint16_t b) {
  return a * b ;
}
```

Bon exemple

Dans l'exemple suivant, les opérateurs post-fixes sont utilisés dans des instructions isolées :

```
#define TAB_SIZE 25U

...

uint32_t x;
uint8_t  b[TAB_SIZE] = { 0 };
uint16_t i = 0;
i++;  /* N.B. le remplacement de cette instruction par ++i; ne changerait en
rien le comportement du programme */
x = foo(i, i);

...

uint32_t foo(uint16_t a, uint16_t b) {
  return a * b ;
}
```

14.2.1 Références

[Misra2012] Rule 13.3 A full expression containing an increment or decrement operator should have no other potential side effects other than that caused by the increment or decrement operator.
[Cert] Rule EXP30-C Do not depend on the order of evaluation for side effects.

14.3 Non utilisation imbriquée de l'opérateur ternaire « ?: »

L'opérateur ternaire `?:` peut permettre l'écriture d'une façon concise d'une affectation d'une variable en fonction d'une condition.

Cependant, lorsque l'opérateur ternaire est utilisé avec une expression conditionnelle complexe, ou si plusieurs opérateurs ternaires sont imbriqués, la compréhension du code et sa maintenance deviennent difficiles.

Dans le cas d'expressions complexes, une conditionnelle *if- else* doit être utilisée.

RÈGLE 155

RÈGLE – Non utilisation imbriquée de l'opérateur ternaire ? :

L'imbrication d'opérateurs ternaires ? : est interdite.

De plus, si les types des expressions sont différents dans les deux « branches », cela implique un cast implicite selon la valeur de la condition de l'opérateur ternaire.

RÈGLE 156

RÈGLE – Bonne construction des expressions avec l'opérateur ternaire ? :

Les expressions résultantes de l'opérateur ternaire ? : doivent être exactement de même type pour éviter tout transtypage.

Mauvais exemple

Dans l'exemple suivant, l'expression ternaire imbriquée rend la compréhension du code difficile et les expressions des deux branches ne sont pas de même type :

```
y = (x<42) ? 1042  :(t> 0) ? -1042  :0.0;
/*entier et flottant  donc cast implicite */
```

Bon exemple

L'exemple suivant utilise plusieurs conditionnelles *if- else* afin de traiter l'affectation d'une valeur à la variable y qui dépend de plusieurs conditions :

```
if(x < 42) {
  y = 1042;
} else {
  if(t > 0) {
    y = -1042;
  } else {
      y = 0;
  }
}
```

14.3.1 Références

15
Gestion de la mémoire

15.1 Allocation dynamique de mémoire

Pour tous les objets alloués dynamiquement par le développeur, différentes règles sont à respecter. Il faut dans un premier temps que le développeur s'assure d'avoir alloué un espace mémoire suffisant pour l'objet en question. Une erreur courante est d'appliquer l'opérateur *sizeof* sur un pointeur de l'objet à allouer au lieu de l'objet à allouer directement ou de ne pas appliquer cet opérateur sur le bon type.

RÈGLE 157

RÈGLE — Allouer dynamiquement un espace mémoire dont la taille est suffisante pour l'objet alloué

Pour un pointeur `ptr`, on préférera utiliser `ptr=malloc(sizeof(*ptr));` quand cela est possible.

De plus, toute mémoire allouée dynamiquement doit être libérée dès que possible.

RÈGLE 158

RÈGLE — Libérer la mémoire allouée dynamiquement au plus tôt

Tout espace mémoire alloué dynamiquement doit être libéré quand celui-ci n'est plus utile.

Cette règle fait écho avec celle de la section 8.6.

Pour les objets stockant des données sensibles, les zones mémoires doivent être réinitialisées avant d'être libérées.

RÈGLE 159

RÈGLE — Les zones mémoires sensibles doivent être mises à zéro avant d'être libérées.

Attention

Il est crucial de s'assurer que ce code de mise à zéro de la mémoire n'est pas optimisé et est bien conservé à la compilation. La plupart des compilateurs considèrent cette mise à zéro comme du code mort puisque les variables associées ne sont pas utilisées ensuite. De façon générale, il ne faut pas pousser les niveaux d'optimisations à la compilation mais parfois même à un niveau bas d'optimisation, il faut malheureusement recoder son propre `memset` pour éviter ce genre de désagrément.

Il est également important de noter que la libération de mémoire est uniquement autorisée pour des objets alloués dynamiquement.

RÈGLE 160

RÈGLE – Ne pas libérer de mémoire non allouée dynamiquement

Enfin, il ne faut pas utiliser `realloc` pour modifier l'espace alloué dynamiquement. Cette fonction peut en effet modifier l'espace mémoire alloué à un objet en augmentant ou diminuant sa taille mais peut aussi libérer la mémoire de l'objet passé en paramètre. Du fait des risques liés à la manipulation de la mémoire ou la potentielle double libération de mémoire correspondant à un comportement indéfini, l'utilisation de cette fonction est à éviter.

RÈGLE 161

RÈGLE – Ne pas modifier l'allocation dynamique via `realloc`

Attention

En cas d'échec, la fonction `realloc` retourne `NULL` mais l'emplacement mémoire initial est resté intact et est donc toujours accessible.

Mauvais exemple

```
#include <stdlib.h>
void fonc(size_t len){
  long *p;
  p = (long *) malloc(len *sizeof(int)); /* mauvais type */
  ...
  p = (long *) realloc(p,0); /* libération de p via realloc */
  ...
  free(p); /* double libération de p */
}
```

Bon exemple

```
#include <stdlib.h>
void fonc(size_t len){
  long *p;
  p = (long *) malloc(len *sizeof(long)); /* type corrigé */
  ...
  free(p); /* realloc supprimé et remplacé par free  */
}
```

15.1.1 Références

[Cert] Rec. MEM00-C Allocate and free memory in the same module, at the same level of abstraction.
[Cert] Rule MEM31-C Free dynamically allocated memory when no longer needed
[Cert] Rule MEM34-C Only free memory allocated dynamically

[Cert] Rule MEM35-C Allocate sufficient memory for an object
[Cert] Rule MEM36-C Do not modify the alignment of objects by calling realloc
[Cert] Rec. MEM03-C Clear sensitive information stored in reusable resources
[Cert] Rec. MEM04-C Beware of zero-length allocations
[Misra2012] Rule 22.1 All resources obtained dynamically by means of Standard Library functions shall be explicitly released
[Misra2012] Rule 22.2 A block of memory shall only be freed if it was allocated by means of a Standard Library function.
[Cwe] CWE-226 Sensitive information uncleared before release.
[Cwe] CWE-244 Failure to clear heap memory before release (”heap inspection”).
[Cwe] CWE-590 Free of memory not on the heap.
[Cwe] CWE-672 Operation on a ressource after expiration or release.
[Cwe] CWE-131 Incorrect Calculation of Buffer Size.
[Cwe] CWE-680 Integer Overflow to Buffer Overflow.
[Cwe] CWE-789 Uncontrolled Memory Allocation.
[IsoSecu] Accessing freed memory [accfree].
[IsoSecu] Freeing memory multiple times [dblfree].
[IsoSecu] Reallocating or freeing memory that was not dynamically allocated [xfree].
[IsoSecu] Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr].
[IsoSecu] Allocating insufficient memory [insufmem].

15.2 Utilisation de l'opérateur sizeof

L'opérateur `sizeof` est essentiel en C afin de connaître la taille d'un objet en mémoire. Cependant, une utilisation non rigoureuse de cet opérateur peut conduire à des comportements inattendus et aboutir à une taille mémoire incorrecte ou aboutir à une expression non évaluée.

Il faut préférer utiliser en paramètre de l'opérateur `sizeof` le type de l'objet et non l'identifiant d'une variable. Le fait d'utiliser l'identifiant a l'avantage d'être « résistant » au changement de type associé mais il faut alors s'assurer que l'utilisation de l'opérateur `sizeof` est correcte.

Afin d'éviter les problèmes liés à l'alignement des membres d'une structure, il est courant d'utiliser :

- soit la directive de pré-compilation *pack* ;
- soit un champ de bourrage explicite.

Attention

La directive *pack* n'est pas standard.

Si un alignement des membres de la structure est nécessaire, la directive de pré-compilation *pack* ou des champs de bourrage peuvent être utilisés.

De plus l'utilisation de l'expression idiomatique `sizeof(array)/sizeof(array[0])` pour déterminer le nombre d'éléments d'un tableau est assez classique mais il faut être très prudent quant

à son usage. En effet, cette expression est correcte uniquement si l'opérateur `sizeof` est appliqué sur le tableau dans le bloc dans lequel le tableau est déclaré. Le résultat de cette expression sera tout autre si l'opérateur `sizeof` est appliqué sur un tableau passé en paramètre car le dit tableau sera alors de type pointeur et non plus tableau.

RÈGLE 162

RÈGLE – Bonne utilisation de l'opérateur sizeof

Une expression contenue dans un `sizeof` ne doit pas :

- contenir l'opérateur « = » car l'expression ne sera pas évaluée ;
- contenir de déréférencement de pointeur ;
- être appliqué sur un pointeur représentant un tableau.

Attention

L'opérateur `sizeof` ne renvoie pas la taille de l'objet, mais celle utilisée en mémoire.

Mauvais exemple

L'exemple ci-dessous présente des utilisations incorrectes de l'opérateur `sizeof`.

```
uint8_t tab[LEN];
typedef struct s_example{
  uint32_t ui_field1;
  uint8_t ui_field2;
} t_example;

int32_t i, isize;
t_exemple test;
t_exemple* ptr;

i = 5;
ptr = NULL;
isize = sizeof(i = 1234); /* l'expression i= 1234 ne sera pas traitée. */
/* i a pour valeur 5 et non 1234. isize vaut 4 */
isize = sizeof(t_example);/* la valeur retournée par sizeof est  8 pour un
    alignement sur 32 bits */
isize = sizeof(*ptr); /* l'expression sizeof(*ptr) retourne bien la taille de la
structure t_example */
void crawl_tab(uint8_t tab[])
{
  for (size_t i = 0; i < sizeof(tab) / sizeof(tab[0]); i++) /* tab est un paramètre
       donc de type pointeur ! */
  ...
}
```

Exemple toléré

L'exemple suivant fait un bon usage de la directive d'alignement et n'utilise pas d'expression comme paramètre de l'appel à l'opérateur sizeof.

```
#pragma pack(push, 1) /* alignement sur 1 octet  - NON STANDARD - */
uint8_t tab[LEN];
typedef struct s_example
{
  uint32_t  ui_field1;
  uint8_t   ui_field2;
} t_exemple ;
#pragma pack(pop) /* retour alignement par défaut */
```

```c
int32_t i;
size_t isize;

i = 5;
isize = sizeof(int32_t);
i = 1234 ;
isize = sizeof(t_example); /* La valeur retournée par sizeof est 5 car la
structure a été déclarée avec un alignement sur 1 octet */
void crawl_tab(uint8_t tab[], size_t n) /*taille connue du tableau */
{
  for (size_t i = 0; i <n ; i++) /* tab est un paramètre donc de type pointeur ! */
  ...
}
```

15.2.1 Références

[Cert] Rec. EXP09-C Use sizeof to determine the size of a type of a variable.
[Cert] Rule EXP44-C Do not rely on side effects in operand to sizeof, _Alignof, or _Generic
[Cert] Rec. ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array.
[Misra2012] Rule 13.6 The operand of the sizeof operator shall not contain any expression which has potential side effects.
[IsoSecu] Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr].
[Cwe] CWE-131 Incorrect Calculation of Buffer Size.
[Cwe] CWE-467 Use of `sizeof()` on a pointer type.
[Cwe] CWE-805 Buffer access with incorrect length value.

15.3 Vérification obligatoire du succès d'une allocation mémoire

Lors d'une allocation mémoire, il est possible que celle-ci échoue dans le cas où le système ne dispose plus de mémoire libre. L'absence de test du pointeur retourné par la fonction d'allocation va engendrer un crash du programme lors de la première utilisation du pointeur.

Généralement, en cas d'échec d'allocation, la fonction d'allocation mémoire retourne un pointeur `NULL`. Il est donc nécessaire de vérifier que le pointeur retourné par la fonction d'allocation est différent de `NULL`.

Dans le cas où la fonction d'allocation a un comportement différent sur une erreur d'allocation, se reporter à la documentation de la fonction afin de gérer l'erreur de façon appropriée.

RÈGLE 163

RÈGLE — Vérification obligatoire du succès d'une allocation mémoire

Le succès d'une allocation mémoire doit toujours être vérifié.

Cette règle est un cas particulier de la section 13.7 mais avec une attention particulière sur la gestion des erreurs d'allocation mémoire.

Mauvais exemple

Dans l'exemple ci-dessous, il manque la vérification du succès de l'allocation mémoire :

```
point_t *p_point;
p_point = (point_t *)malloc(sizeof(point_t));
/* absence de vérification du retour de fonction */
p_point->x = 0.0f;
p_point->y = 0.0f;
```

Bon exemple

Dans le code suivant, le succès de l'allocation mémoire est vérifié avant l'utilisation du pointeur :

```
point_t  *p_point = NULL;

p_point = (point_T *)malloc(sizeof(point_t));

if (NULL != p_point) {
    p_point->x = 0.0f;
    p_point->y = 0.0f;
} else {
    /* traitement erreur */
}
```

15.4 Isolation des données sensibles

Lorsque des données sensibles sont chargées en mémoire (par exemple des clés de chiffrement), celles-ci demeurent encore en mémoire après que le programme ait fini d'y accéder. Un autre programme peut accéder à la mémoire de notre programme par *des canaux auxiliaires* [15].

Il faut donc associer les zones mémoires à leur usage : les données représentant des valeurs distinctes sont stockées dans des espaces mémoires distincts. En cas de recyclage d'une zone mémoire partagée, il faut s'assurer de l'effacement de cette zone avant sa réutilisation.

Toutes les zones mémoires qui contiennent des données sensibles doivent être effacées explicitement une fois que le programme n'a plus besoin d'accéder à ces données.

Attention

Le fait d'effacer des buffers pour que les données ne restent pas sur la pile via un `memset` par exemple peut être jugé inutile par le compilateur et les appels associés peuvent donc être supprimés en vue d'optimiser le code. Le développeur doit être conscient de ce risque et consulter la documentation du compilateur utilisé afin de s'assurer que les appels en question soient bien conservés.

RÈGLE 164

RÈGLE — L'isolement des données sensibles doit être effectué

Contrôler le bon usage d'une zone mémoire stockant des données sensibles *i.e.* minimiser l'exposition en mémoire, minimiser la copie et effacer la/les zones ayant contenu les données sensibles au plus tôt.

15. Les illustrations sont nombreuses : Meltdown, Spectre, ZombieLoad, etc.

Mauvais exemple

Dans l'exemple ci-dessous, un même buffer est utilisé pour stocker la clé puis le vecteur d'initialisation :

```
#define WORK_SIZE 32U

#include <stdlib.h>

void process(const uint8_t *key, uint16_t key_size, const uint8_t *init,
             uint16_t init_size) {
  uint8_t buffer[WORK_SIZE];

  if ((NULL == key) || (NULL == init)) {
    /* gestion erreur */
    ...
  }

  memcpy(buffer, key, min(key_size, WORK_SIZE));
  print_key(buffer);

  /* buffer contient 4 octets du vecteur d initialisation ainsi que les 12
  derniers octets de cle */
  memcpy(buffer, init, min(init_size, WORK_SIZE));
  printIV(buffer);
  ...
  /* pas d effacement sécurisé */
}
```

Bon exemple

L'exemple suivant présente un cloisonnement entre la clé et le vecteur d'initialisation :

```
#define KEY_SIZE 16U
#define IV_SIZE 4U

#include <stdlib.h>

void process(const uint8_t *key, uint16_t key_size, const uint8_t *init,
             uint16_t init_size) {
  uint8_t my_key[KEY_SIZE];
  uint8_t iv[IV_SIZE];

  if ((NULL == key) || (NULL == init)) {
    /* gérer l'erreur */
    ...
  }

  memcpy(my_key, key, min(key_size, KEY_SIZE));
  print_key(cle);

  memcpy(iv, init, min(init_size, IV_SIZE));
  printIV(iv);
  ...
  /* effacement des buffers, afin que les données ne restent pas sur la
  pile
  ATTENTION  : le compilateur peut optimiser et supprimer ces appels
  qui peuvent être jugés comme inutiles.
  Il faut donc consulter la documentation du compilateur afin que
  les appels soient conservés. */
  memset(cle, 0, KEY_SIZE);
  memset(iv, 0, IV_SIZE);
}
```

15.4.1 Références

[Cert] Rec. MSC18-C Be careful while handling sensitive data, such as passwords, in program code.

16
Gestion des erreurs

16.1 Bonne utilisation de `errno`

La variable `errno`, activée par le fichier d'en-tête `<errno.h>` est de type `int` et différentes fonctions de la bibliothèque standard modifient sa valeur avec une valeur positive en cas d'erreur. Il est donc important d'initialiser `errno` avant tout appel de fonction de la bibliothèque standard qui modifie sa valeur et il faut aussi par conséquent consulter sa valeur à la fin de l'exécution de telles fonctions.

RÈGLE 165

RÈGLE — Initialiser et consulter la valeur de `errno` avant et après toute exécution d'une fonction de la bibliothèque standard qui modifie sa valeur

Mauvais exemple

```
#include <stdlib.h>
void try1 (const unsigned char * len)
{
  unsigned long res;
  res = strtoul(len,NULL,5); /* conversion chaîne de caractères en unsigned long */
  /* la fonction strtoul écrit dans errno */
  if (res == ULONG_MAX)
  {
    /* gestion pb */
  }
...
}
```

Bon exemple

```
#include <stdlib.h>
#include <errno.h>
void try1 (const unsigned char * len)
{
  unsigned long res;
  errno = 0; /* init errno */
  res = strtoul(len,NULL,5); /* conversion chaîne de caractères en unsigned long */
  /* strtoul écrit dans errno */
  if (res == ULONG_MAX && errno != 0) /* lecture errno */
  {
    /* gestion pb */
  }
...
}
```

16.1.1 Références

[Cert] Rule ERR30-C Set errno to zero before calling a library function known to set errno and check errno only after the function returns a value indicated failure.
[Cert] Rule ERR32-C Do not rely on indeterminate values of errno.
[IsoSecu] Incorrectly setting and using errno [inverrno]
[Cwe] CWE-456 Missing Initialisation of a variable.

16.2 Prise en compte systématique des erreurs retournées par les fonctions de la bibliothèque standard

La plupart des fonctions de la bibliothèque standard retournent des valeurs pour indiquer le bon fonctionnement de la fonction mais aussi une erreur à l'exécution de la fonction. L'absence de test de la valeur de retour risque de mener à l'utilisation de données erronées produites par la fonction.

RÈGLE 166

RÈGLE — La gestion des erreurs retournées par une fonction de la bibliothèque standard doit être systématique

Tout retour de fonction doit être lu afin de mettre en place le traitement adapté suite à l'exécution de la fonction.

Cette règle est un cas particulier de la section 13.7 mais avec une attention particulière sur la gestion des erreurs des fonctions de la bibliothèque standard.

Mauvais exemple

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  FILE *fp = fopen("myfile.txt", "w"); /* retour fonction non lu */
  fputs("hello\n", fp);
...
}
```

Bon exemple

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  FILE *fp = fopen("myfile.txt", "w");
  if (fp != NULL) {
    fputs("hello\n", fp);
  ...
  }
}
```

16.2.1 Références

[Misra2012] Dir. 4.7 : If a function returns error information, then that error information shall be tested.
[Cert] EXP12-C Do not ignore values returned by functions.
[Cert] ERR33-C Detect and handle standard library errors.
[Cert] FIO37-C Do not assume that fgets() or fgetsw() returns a nonempty string when successful.
[Cwe] CWE-241 Improper Handling of Unexpected Data Type.
[Cwe] CWE-252 Unchecked Return Value.
[Cwe] CWE-253 Incorrect Check of Function Return Value.
[Cwe] CWE-391 Unchecked Error Condition.
[IsoSecu] Failing to detect and handle library errors [liberr].
[IsoSecu] Forming invalid pointers by library function [libptr].

16.3 Documentation et structuration des codes d'erreur

La documentation incomplète du prototype d'une fonction risque d'engendrer des erreurs de programmation, en particulier au niveau de la gestion d'erreur, si l'ensemble des codes de retour n'est pas indiqué avec leur signification.

Un modèle de documentation pour les codes d'erreur doit être défini. Celui-ci devrait contenir pour chaque code de retour, l'erreur associée et dans le cas où plusieurs codes d'erreur peuvent survenir en même temps, la priorité entre ces codes doit être précisée pour la gestion des erreurs.

RÈGLE 167

RÈGLE — Documentation des codes d'erreur

Tous les codes d'erreur retournés par une fonction doivent être documentés. Dans le cas où plusieurs codes d'erreur peuvent être retournés en même temps par la fonction, la documentation doit définir la priorité de gestion de ces codes.

Les codes d'erreur doivent être porteurs d'informations. Sans structuration, l'information indiquée par le code de retour est souvent insuffisante. La structuration des codes de retour via des masques est une possibilité. Les codes de retour doivent également être structurés de façon à pouvoir déterminer si la valeur provient d'une exécution normale de la fonction ou au contraire si un élément externe est venu la perturber (débordement de buffer, …).

RECO 168

RECOMMANDATION — Structuration des codes de retour

Les codes de retour doivent être structurés de façon à pouvoir obtenir rapidement une information concernant le déroulement de la fonction appelée :

- erreur ;
- type d'erreur ;
- alarme ;
- type d'alarme ;
- ok ;

- ...

16.3.1 Références

[Cert] Rec. ERR00-C Adopt and implement a consistent and comprehensive error-handling policy.

16.4 Code de retour d'un programme C en fonction du succès ou non de son exécution

La gestion du code retour d'un programme n'est pas identique d'un système d'exploitation à un autre ou d'un interpréteur de commandes à un autre. Cela peut donc provoquer des problèmes de portabilité du code. D'un système d'exploitation à un autre, ou d'un interpréteur de commandes à un autre, le domaine de valeurs autorisé n'est pas identique pour le code de retour d'un programme :

- sous Windows, l'interpréteur de commandes cmd.exe autorise des entiers 32 bits signés (valeur accessible dans la variable ERRORLEVEL) ;
- sous Linux, l'interpréteur de commandes autorise une valeur comprise entre 0 et 255 (même si des codes sont réservés pour les signaux ; la valeur est accessible via la variable $?).

L'utilisation d'un code de retour compris entre 0 et 127 permet de se prémunir des risques de modification (par transtypage) ou de mauvaise interprétation du code de retour d'un programme :

- les valeurs comprises entre 0 et 127 peuvent être codées sur 7 bits ;
- et elles ont le même codage que le type d'entier soit signé ou non signé.

RÈGLE 169

RÈGLE — Code de retour d'un programme C en fonction du résultat de son exécution

Le code de retour d'un programme C doit avoir une signification afin d'indiquer le bon déroulement du programme ou la survenue d'une erreur :

- la valeur du code de retour doit être comprise entre 0 et 127 ;
- la valeur 0 indique que le programme s'est exécuté sans erreur ;
- la valeur 2 est généralement utilisée sous Unix pour indiquer une erreur dans les arguments passés en paramètres au programme.

La signification des codes de retour du programme doit être indiquée dans sa documentation.

Mauvais exemple

Le code suivant présente un problème de portabilité entre Windows et Linux. En effet, la valeur -1 est convertie en 255 sous Linux avec l'interpréteur de commandes bash :

```
int main(int argc, char* argv[]) {
```

```
  if (argc != 2) {
    /* nombre arguments incorrect */
    return -1; /* le code de retour ne sera pas interprété correctement sous Linux
        */
  }
  ...
  return 0;
}
```

Bon exemple

Dans l'exemple suivant, les codes de retour utilisés ne posent pas de problème de portabilité :

```
#define RESULT_OK (0U)
#define ARG_ERROR (2U)

int main(int argc, char* argv[]) {
  if (argc != 2) {
    /* nombre arguments incorrect */
    return ARG_ERROR;
  }
  ...
  return RESULT_OK;
}
```

16.5 Terminaison d'un programme C suite à une erreur

Lorsque plusieurs points de sortie sont définis dans un programme C, cela rend difficile la mise en place de tests pour ce programme ou les bibliothèques utilisées par ce programme. La gestion des erreurs doit être effectuée à l'aide de codes d'erreur. Dans le cas où une erreur critique est rencontrée, le programme ne doit pas être terminé par un appel à la fonction `abort()` ou la fonction `_Exit()`(C99) dans le code où l'erreur a été détectée. En effet, ces deux fonctions ne terminent pas *proprement* le programme *i.e.* elles court-circuitent les routines de terminaisons normales (fermeture des fichiers, suppression des fichiers temporaires, écriture des données ...). L'erreur doit être remontée à l'aide d'un code d'erreur jusqu'à la fonction principale `main()`, qui se charge alors de terminer le programme.

RECO 170

RECOMMANDATION — Privilégier les retours d'erreurs via des codes de retour dans la fonction principale

Un programme C doit disposer d'une fonction `main()` minimale. Les retours d'erreurs se font par un retour de code dédié (et donc documenté) de cette fonction.

RÈGLE 171

RÈGLE — Ne pas utiliser les fonctions abort() ou _Exit()

La fonction `exit()` entraîne une terminaison normale du programme et non dépendante de l'implémentation. Cette sortie du programme peut être utilisée mais une utilisation trop fréquente de cette fonction dans le programme peut rendre sa compréhension difficile.

RECO 172

RECOMMANDATION – Limiter les appels à exit()

Les appels à la fonction `exit()` doivent être commentés et non systématiques. Le développeur doit le plus souvent possible les remplacer par un retour de code d'erreur dans la fonction principale.

Enfin les fonctions `setjmp()` et `longjump()` définies dans la bibliothèque `setjmp.h` principalement utilisées pour la gestion des exceptions en C peuvent amener facilement à des comportements indéfinis et ne doivent donc pas être utilisées. En particulier, leurs utilisations posent problème avec la gestion des signaux.

RÈGLE 173

RÈGLE – Ne pas utiliser les fonctions `setjmp()` et `longjump()`

Mauvais exemple

```
#include <stdlib.h>
#include <stdio.h>
int read_file(void)
{
  FILE *f = fopen("C :\\myfile.txt", "w");
  if (NULL == f)
  {
    /* pb ouverture fichier */
    _Exit(12); /* non autorisé  */
  }
  fprintf(f, "%s", "blablabla");
  ...
  abort(); /* non autorisé */
  ...
  return 0;
}
int main(void)
{
  int val = read_file();
  ...
  return 1;
}
```

Bon exemple

```
#include <stdlib.h>
#include <stdio.h>
int read_file(void)
{
  FILE *f = fopen("C :\\myfile.txt", "w");
  if (NULL == f)
  {
    /* pb ouverture fichier */
    return 12; /* code erreur documenté pour pb ouverture de fichier */
  }
  fprintf(f, "%s", "blablabla");
  ...
  return 10; /* autre code erreur documenté */
  ...
  return 0; /* pas de pb */
}
int main(void)
{
```

```
  int val = read_file();
  if (val == 0)
  { /* pas de pb dans la fonction */
    ...
    return 0;
  }
  else
  { /* traitement des erreurs selon code retourné */
    ...
    return 1;
  }
}
```

16.5.1 Références

[Cert] Rule SIG30-C Call only asynchronous functions with signal handlers.
[Cert] ERR00-C Adopt and implement a consistent and comprehensive error-handling policy.
[Cert] ERR04-C Choose an appropriate termination strategy.
[Cert] ERR06-C Understand termination behavior of assert() and abort().
[IsoSecu] Calling functions in the C Standard Library other than `abort`, `_Exit` and `signal` from within a signal handler [asyncsig].
[IsoSecu] Calling `signal` from interruptible signal handlers [sigcall].
[Cwe] CWE-479 Signal Handler Use of a Non-reentrant Function.

17
Bibliothèque standard

17.1 Fichiers d'en-tête de la bibliothèque standard interdits

Plusieurs fichiers d'en-tête de la bibliothèque standard introduisent uniquement des fonctions qui sont en contradiction avec les règles ou recommandations de ce guide :

- `setjmp.h`;
- `stdarg.h`.

Par voie de conséquence, ces fichiers d'en-tête ne doivent pas être utilisés car ils entrent en violation avec plusieurs règles précédentes.

La bibliothèque `<stdarg.h>`, par exemple, introduite dans le standard C90, déclare un type et définit 3 macros : `va_start`, `va_arg`, `va_end`. Une nouvelle macro (`va_copy`) est introduite avec le C99. Cette bibliothèque a pour but de permettre la définition de fonction à nombre et type variables d'arguments. De plus, l'utilisation de ces fonctionnalités peut engendrer, dans plusieurs cas, un comportement non défini.

Une incohérence de typage dans l'appel d'une fonction variadique peut entraîner un arrêt inattendu de la fonction voire un comportement non défini.

RÈGLE 174

RÈGLE — Ne pas utiliser les bibliothèques standards setjmp.h et stdarg.h

17.1.1 Références

[Misra2012] Rule 17.1 The features of <stdarg.h> shall not be used.
[Cert] Rec. DCL10-C Maintain the contract between the writer and caller of variadic functions
[Cert] Rec. DCL11-C Understand the type issues associated with variadic functions
[Cert] Rule MSC39-C Do not call va_arg() on a va_list that has an indeterminate value
[Misra2012] 21.4 The standard header file <setjmp.h> shall not be used.
[Cert] MSC22-C Use the setjmp(), longjmp() facility securely
[Cert] ERR04-C Choose an appropriate termination stategy.
[Cert] ERR05-C Application independent code should provide error detection without dictating error handling.

17.2 Bibliothèques standards déconseillées

Pour les bibliothèques suivantes, leur utilisation doit être limitée et conservée uniquement si cela est nécessaire :

- float.h
- complex.h
- fenv.h
- math.h

RECO 175

RECOMMANDATION — Limiter l'utilisation des bibliothèques standards manipulant des nombres flottants

Les bibliothèques standards float.h, fenv.h, complex.h et math.h ne doivent être utilisées que si cela est vraiment nécessaire comme pour le cas de traitements numériques.

17.2.1 Références

[Misra2012] 21.11 The standard header file <tgmath.h> shall not be used.
[Misra2012] 21.12 The standard header file <fenv.h> shall not be used.
[Cert] FLP32-C Prevent or detect domain and range errors inmath functions.
[Cert] FLP03-C Detect and handle floating point errors.
[Cwe] CWE-682 Incorrect calculation

17.3 Fonctions de bibliothèques standards interdites

D'autres bibliothèques contiennent des fonctions dangereuses comme les fonctions `atoi()`, `atol()`,`atof()` et `atoll()` de `stdlib.h` qui entraînent des comportements indéfinis si la valeur résultante ne peut être représentée. Les fonctions `strto*()` sont à privilégier car elles ont la même action sans le risque de comportement indéfini.

RÈGLE 176

RÈGLE — Ne pas utiliser les fonctions `atoi()` `atol()` `atof()` et `atoll()` de la bibliothèque stdlib.h

Les fonctions équivalentes `strto*()` sont à utiliser en remplacement.

La fonction `rand()` de la bibliothèque standard pour la génération pseudo-aléatoire de nombres ne donne aucune garantie quant à la qualité de l'aléa généré.

RÈGLE 177

RÈGLE — Ne pas utiliser la fonction `rand()` de la bibliothèque standard

17.3.1 Références

[Misra2012] The atof, atoi, atol and atoll functions shall not be used.
[Cert] ERR07-C Prefer functions that support error checking over equivalent functions that don't.
[Cert] Rec MSC25-C Do not use insecure or weak cryptographic algorithms.
[Cert] Rule MSC30-C Do not use the `rand()` function for generating pseudorandom numbers.
[Cwe] CWE-327 Use of a Broken or Risky Cryptographic Algorithm.
[Cwe] CWE-338 Use of Cryptographically Weak Pseudo-random Number Generator (PRNG).
[Cwe] CWE-676 Use of potentially dangerous functions.

17.4 Choix entre les différentes versions de fonctions de la bibliothèque standard

Quand une fonction de la bibliothèque standard possède une version « moins dangereuse » dans le sens où celle-ci rajoute une sécurité supplémentaire, l'utilisation de cette version doit être privilégiée.

Attention

On parle de version « moins dangereuse » car ces fonctions ajoutent par exemple une limite sur la taille d'un paramètre d'entrée, mais elles peuvent toujours entraîner un comportement non défini ou non spécifié.

Information

Dans des versions ultérieures du langage C (en particulier pour le C11), des nouvelles versions réellement plus sécurisées sont proposées comme `strcpy_s()`.

Les fonctions de manipulation de chaînes de caractères de type `strxx` seront remplacées par les fonctions équivalentes `strnxx` quand il est possible de borner le nombre de caractères concernés par la manipulation.

RÈGLE 178

RÈGLE — Utiliser les versions « plus sécurisées » pour les fonctions de la bibliothèque standard

Quand des fonctions de la bibliothèque standard existent en différentes versions, la version « plus sécurisée » doit être utilisée.

De la même façon, toutes les fonctions obsolescentes ou obsolètes ne doivent pas être utilisées. L'exemple le plus connu est celui de la fonction `gets()` rendue obsolète dans le troisième correctif technique du C99[AnsiC99] et qui fut supprimée des standards suivants.

RÈGLE 179

RÈGLE — Ne pas utiliser de fonctions de la bibliothèque obsolescentes ou devenues obsolètes dans des normes suivantes.

RÈGLE 180

RÈGLE — Ne pas utiliser de fonctions de la bibliothèque manipulant des buffers sans prendre la taille du buffer en argument.

17.4.1 Références

[Cert] Rec. PRE09-C Do not replace secure functions with deprecated or obsolescent functions.
[Cert] Rec. MSC24-C Do not use deprecated or obsolescent functions.
[Cwe] CWE-20 Insufficient input validation
[Cwe] CWE-120 Buffer Copy without checking Size of Input('Classic Buffer Overflow')
[Cwe] CWE-676 Use of potentially dangerous function
[Cwe] CWE-684 Failure to provide specified functionality.

18
Analyse, évaluation du code

18.1 Relecture de code

Il est sain pour tout développeur et même si cela n'est pas imposé de refaire lire son code au moins une fois par un relecteur dédié ou un autre développeur pour vérifier la maintenabilité et la compréhension de son code.

BONNE PRATIQUE 181

BONNE PRATIQUE — Tout code doit être soumis à relecture

18.2 Indentation des expressions longues

Lorsqu'une expression est longue, en l'absence d'une indentation adéquate, il est très difficile de comprendre le code et l'intention du développeur. L'utilisation de caractères espaces pour l'indentation des expressions et instructions permet plus de souplesse pour l'indentation que l'utilisation du caractère tabulation.

RECO 182

RECOMMANDATION — Indentation des expressions longues

Lorsqu'une instruction ou une expression s'étale sur plusieurs lignes, il est indispensable de l'indenter afin de faciliter la compréhension du code.

Mauvais exemple

Le code dans l'exemple suivant devrait être ré-indenté afin d'être plus facilement compréhensible :

```
if ((OPT_1 == opt)
|| ((c >= a) && (0xdeafbeef == b)
&& (NULL == p_point)))
{
  /* traitements */
}
if (0 != un_nom_de_fonction_vraiment_a_rallonge_pour_etre_explicite_au_maximum(
    UNE_CONSTANTE_TOUJOURS_AVEC_UN_NOM_TRES_EXPLICITE,
    UNE_SECONDE_CONSTANTE_TOUJOURS_AVEC_UN_NOM_A_RALLONGE,5, 50))
{
  /*  traitements */
}
```

Bon exemple

Le code suivant présente une indentation correcte d'une conditionnelle sur plusieurs lignes :

```
if((OPT_1 == opt)
  || ((c >= a)
  && (0xdeafbeef == b)
  && (NULL == p_point)
  ))
  {
  /* traitements */
  }
if(0 != un_nom_de_fonction_vraiment_a_rallonge_pour_etre_explicite_au_maximum(
      UNE_CONSTANTE,UNE_CONSTANTE_TOUJOURS_AVEC_UN_NOM_TRES_EXPLICITE,
      UNE_SECONDE_CONSTANTE_TOUJOURS_AVEC_UN_NOM_A_RALLONGE,
      5,
      50))
   {
    /*  traitements */
   }
```

18.3 Identifier et supprimer tout code mort ou code inatteignable

La présence de code mort ou de code inatteignable gêne la relecture et la compréhension du code.

Code inatteignable

Du code est considéré comme inatteignable dans le cas où il n'existe aucune entrée qui permet d'atteindre ce point du programme (instructions dans une conditionnelle toujours fausse, instructions situées après une instruction `return`, …)

Code mort

Il est entendu par « code mort » du code dont l'exécution n'a aucun effet (aucune modification de variable, aucun impact sur le flot de contrôle, …).

Par ailleurs, d'un point de vue sécurité, le code mort ou le code inatteignable peut être utilisé au cours d'un détournement du flot d'exécution. Ce code inatteignable peut être un code de mise au point, désactivant des contrôles de sécurité.

RÈGLE 183

RÈGLE — Identifier et supprimer tout code mort

RÈGLE 184

RÈGLE — Le code doit être exempt de code non atteignable en dehors de code défensif et de code d'interface

Il ne doit jamais y avoir de code inatteignable, sauf s'il s'agit de code défensif ou s'il s'agit de code d'une interface et dans ces deux cas, il faut le préciser en commentaire.

18.3.1 Références

[Misra2012] Rule 2.1 A project shall not contain *unreachable* code.
[Misra2012] Rule 2.2 There shall be no dead code.
[Cwe] CWE-561 Dead code.
[Cwe] CWE-563 Assignment to Variable without use.
[Cwe] CWE-570 Expression is always False.
[Cwe] CWE-571 Expression is always True.

18.4 Évaluation outillée du code source pour limiter les risques d'erreurs d'exécution

Malgré l'application de conventions de codage, de bonnes pratiques de programmation et de l'exécution de tests, des erreurs subsistent fréquemment dans un logiciel. Une partie de ces erreurs résiduelles peut être découverte avec des outils d'analyse de code. L'analyseur de code doit être utilisé au fur et à mesure du développement ce qui permet de limiter l'impact des modifications et corrections réalisées sur le code mais aussi la complexité de ces modifications et corrections.

Lors de l'exécution des tests, une analyse dynamique peut être effectuée afin d'identifier les fuites mémoires. Une mesure de la couverture de code doit également être faite afin d'identifier les parties du logiciel qui n'ont pas été testées.

RECO 185

RECOMMANDATION — Evaluation outillée du code source pour limiter les risques d'erreurs d'exécution

Le code source du logiciel doit être analysé via au moins un outil d'analyse de code. Les résultats produits par l'outil d'analyse doivent être étudiés par le développeur et les corrections doivent être effectuées par rapport aux problèmes découverts.

18.4.1 Références

[Misra2012] Rule 1.3 There shall be no occurrence of undefined or critical unspecified behaviour.
[Misra2012] Dir. 4.1 : Run-time failures shall be minimized.

18.5 Limitation de la complexité cyclomatique

Complexité cyclomatique

La complexité cyclomatique est une métrique qui mesure la complexité structurelle d'un programme informatique (module, fonction). Elle correspond au nombre de chemins existants.

Il est souvent constaté que plus la complexité cyclomatique est importante, plus le programme informatique est difficile à tester et à maintenir. Une complexité cyclomatique élevée indique une forte probabilité d'introduire des erreurs lors de l'évolution ou de la maintenance d'un programme.

En cas de complexité cyclomatique importante, il convient de réorganiser le code afin de le simplifier. Cela peut notamment se faire par l'écriture de fonctions supplémentaires.

RECO 186

RECOMMANDATION – Limitation de la complexité cyclomatique

La complexité cyclomatique d'une fonction doit être limitée au maximum.

18.6 Limitation de la longueur des fonctions

Dans la lignée de la section précédente, à chaque fonction d'un programme doit correspondre une action claire. Trop souvent, des fonctions en C sont en réalité destinées à plusieurs actions/traitements ce qui complexifie la lecture du code, son évolution et sa maintenance. Une fonction trop longue, en terme de nombre de lignes de code, est souvent signe d'une fonction trop complexe avec différents traitements et qui pourrait donc être coupée en plusieurs sous-fonctions. Dans de tels cas, il convient de réorganiser le code de la fonction afin de le simplifier et de le réorganiser en différentes fonctions de plus petites tailles et associées à un traitement précis.

RECO 187

RECOMMANDATION – Limitation de la longueur et la complexité d'une fonction

Une fonction doit être associée idéalement à un seul et unique traitement et doit donc correspondre à un nombre de lignes de code raisonnable.

18.7 Ne pas utiliser de mots clés du C++

Les langages C et C++ sont deux langages de programmation **différents**, bien qu'ils comportent de nombreuses similitudes, et que le langage C++ incorpore la plupart des fonctionnalités du langage C.

Un développeur peut utiliser par inadvertance des mots clés du C++ (par exemple : class, new, private, public, delete, …) au sein d'un code C que ce soit pour nommer une fonction, variable ou autre. Cependant, cela gêne la relecture de code, et risque d'entraîner la confusion des outils d'analyse. De plus, cela peut gêner la maintenance et peut poser des soucis à la compilation, si le compilateur comprend également le C++. Une recherche de ces mots clés dans les sources d'un programme C peut être aisément automatisée. Lorsqu'un des mots clés est trouvé, le nom de la variable, du type ou de la fonction doit être modifié.

L'annexe C indique la liste des mots clés du C++.

RÈGLE 188

RÈGLE — Ne pas utiliser de mots clés du C++

Aucun mot clé du C++ ne doit être utilisé dans le code source d'un programme C.

Mauvais exemple

Dans le code ci-dessous, les noms des fonctions new et delete devraient être modifiés en new_point et delete_point par exemple.

```
/* point.h */

typedef struct
{
  float x;
  float y;
} point_t;

point_t *new();

void delete(point_t *p);
```

Bon exemple

Dans l'exemple suivant, aucun mot clé C++ n'est utilisé :

```
/* point.h */
typedef struct
{
  float x;
  float y;
} point_t;
point_t *new_point();
void delete_point(point_t *p);
```

19
Divers

19.1 Format des commentaires

Le format des commentaires accepté selon le C90 est uniquement de la forme :

```
/* commentaires pouvant être sur plusieurs
lignes  */
```

En C99, la notation des commentaires sur une ligne est étendue avec le format suivant :

```
// commentaires sur une seule ligne
```

Les séquences de caractères /* et // sont interdites dans tout commentaire de même que le caractère de continuation de ligne \ est proscrit dans un commentaire introduit par // car cela entraîne un comportement non défini.

RÈGLE 189

RÈGLE — Séquences de caractères interdites dans les commentaires

Les séquences /* et // sont interdites dans tous les commentaires. Et un commentaire sur une ligne introduit par // ne doit pas contenir de caractère de continuation de ligne \.

19.1.1 Références

[Misra2012] Rule 3.1 The characters sequences /* and // shall not be used within a comment
[Misra2012] Rule 3.2 Line-splicing shall not be used in // comments.

19.2 Mise en œuvre manuelle d'un mécanisme de « canaris »

Déjà présentés en sous-section 5.3.5, les canaris offrent une protection contre certaines erreurs de programmation qui pourraient par exemple permettre le détournement du flot d'exécution du programme par écrasement d'une adresse de retour de fonction sauvegardée sur la pile.

Lorsque la chaîne de compilation ne supporte pas l'insertion automatique de canaris, ce mécanisme doit être mis en œuvre manuellement. Il peut s'agir par exemple de passer un paramètre

supplémentaire à chaque fonction critique et de vérifier sa valeur au début et à la fin de cette fonction, comme illustré dans l'exemple ci-dessous.

RÈGLE 190

RÈGLE – Mettre en œuvre manuellement un mécanisme de canaris lorsque ceux-ci ne sont pas déjà supportés par la chaîne de compilation

Ce mécanisme doit au moins être appliqué aux fonctions critiques du programme.

Lorsque cela n'est pas possible, il reste la possibilité de réaliser une analyse approfondie du code source pour, par exemple, s'assurer qu'il n'y a pas d'utilisation de variables locales de type « tableau », afin de s'affranchir du risque de détournement du flot d'exécution consécutivement au débordement d'un tampon local.

Attention

La mise en œuvre « automatique » de canaris via la chaîne de compilation reste à privilégier. En effet, le développement d'un mécanisme de canaris demeure une tâche compliquée et souvent source d'erreurs de programmation voire de vulnérabilités.

Bon exemple

Le mot clé `volatile` est utilisé afin d'empêcher des optimisations éventuelles du compilateur pour l'accès aux valeurs des variables `canari` et `canariRef`. En effet, il est nécessaire d'aller systématiquement lire les valeurs de `canari` et `canariRef` en mémoire.

```
typedef volatile uint32_t fid_t;

#ifdef ACTIVATE_CANARIES
static inline void verifcanari(fid_t canari, fid_t canariRef) {
  uint8_t res = !!(canari != canariRef);

  if (0 != res)
  {
    /* Traitement spécifique au contexte */
  }
}
#else   /* ifdef ACTIVATE_CANARIES */
static inline void verifcanari(fid_t canari, fid_t canariRef) { }
#endif  /* ifdef ACTIVATE_CANARIES */
void foo(fid_t canari) {
  /* Vérification du paramètre canari en début de fonction */
  verifcanari(canari, FID_FOO);

  /* Corps de la fonction... */

  /* Vérification du paramètre canari en fin de fonction */
  verifcanari(canari, FID_FOO);
}
```

19.3 Assertions de mise au point et assertions d'intégrité

Deux types d'assertions peuvent être distingués dans un logiciel :

- des assertions destinées à la mise au point. Celles-ci ont vocation à être supprimées du logiciel une fois sa phase de qualification terminée (il peut s'agir par exemple de vérifier qu'un paramètre pointeur n'est pas nul) ;
- des assertions destinées à contrôler l'intégrité du logiciel au cours de son exécution : elles ont pour but de s'assurer que le logiciel s'exécute normalement et détecter une défaillance matérielle ou une tentative de modification extérieure (par exemple une attaque par faute).

Une assertion de contrôle de l'intégrité du logiciel ne doit pas être écrite à l'aide de la macro `assert()`. En effet, cette macro est supprimée du code généré lors d'une compilation en mode *release*. De plus, ces assertions ne doivent être utilisées que pour du *debug* et sont en particulier déconseillées pour de la vérification en particulier du fait des initialisations activées en mode *debug* qui ne seront plus d'actualité hors mode *debug*.

Il peut arriver qu'un code contrôlant l'intégrité d'un logiciel soit détecté comme étant du code inatteignable par le compilateur ou un outil d'analyse statique (en effet, le code peut vérifier un ensemble de conditions qui ne peuvent pas se produire lors d'une exécution normale du programme). Il est nécessaire de clairement documenter le but de ce code et également de s'assurer que les optimisations du compilateur n'aboutissent pas à une suppression de ce code dans le binaire généré.

RÈGLE 191

RÈGLE — Pas d'assertions de mise au point sur un code mis en production

Les assertions de mise au point ne doivent pas être présentes en production.

RECO 192

RECOMMANDATION — La gestion des assertions d'intégrité doit inclure un effacement des données d'urgence

Les assertions d'intégrité doivent apparaître en production. En cas de déclenchement d'une assertion d'intégrité, le code de traitement doit aboutir à un effacement d'urgence des données sensibles.

19.4 Dernière ligne d'un fichier non vide doit se terminer par un retour à la ligne

L'absence d'un retour à la ligne en fin de fichier non vide entraîne un comportement non défini selon les standards C90 et C99.

Attention

La très grande majorité des éditeurs en particulier en environnement Linux ajoute de façon automatique et invisible ce retour à la ligne à la fermeture des fichiers.

De plus, toutes les directives de préprocesseur et les commentaires doivent être fermés.

RÈGLE 193

RÈGLE – Tout fichier non vide doit se terminer par un retour à la ligne et les directives de préprocesseur et les commentaires doivent être fermés

Un fichier non vide ne doit pas se terminer au milieu d'un commentaire ou d'une directive de préprocesseur.

Annexe A
Acronymes

ANSI American National Standards Institute

API Application Programming Interface

ASLR Address space layout randomization

FAM Flexible Array Member

IDE Integrated Development Environment

ISO International Standards Organization

MISRA Motor Industry Software Reliability Association

VLA Variable Length Array

Annexe B
Compléments sur les options de GCC et Clang

Les informations données dans cette annexe proviennent de [GccRef] et [ClangRef], respectivement pour les versions 11 de GCC et 13 de CLANG.

B.1 Définition de la version du standard C utilisée

L'option `-std` permet de déterminer la version du standard C — ou du dialecte GNU correspondant — utilisée par le compilateur. Sans cette option, le dialecte GNU du C17 est employé par défaut.

Information

L'option `-ansi` est équivalente à l'option `-std=c90`, elle-même équivalente à `-std=c89` et `-std=iso9899:1990`.

Information

L'option `-std=iso9899:199409` représente le C90 tel que modifié dans l'amendement 1 de 1995.

Information

L'option `-std=iso9899:1999` est équivalente à `-std=c99`.

B.2 Avertissements supplémentaires

Les options suivantes ne sont incluses ni dans `-Wall` ni dans `-Wextra` ni dans `-Wpedantic` et n'ont pas été mentionnées dans le chapitre 5, mais peuvent néanmoins se révéler utiles [16] :

- `-Wbad-function-cast`
- `-Wcast-align`
- `-Wcast-qual` (avertit lorsque des conversions explicites de pointeurs suppriment — ou ajoutent de façon dangereuse — un qualificatif de type comme `const`)

16. Seules les options dont le nom n'est pas jugé assez évocateur sont expliquées. Dans tous les cas, le lecteur est invité à consulter les manuels des compilateurs GCC et CLANG pour des explications plus détaillées.

- `-Wconversion` (avertit de conversions implicites susceptibles d'altérer une valeur, y compris en changeant son signe)
- `-Wfloat-equal`
- `-Wnull-dereference`
- `-Wshadow` (avertit lorsqu'une variable locale ou une déclaration de type réutilise un identifiant déjà lié à une autre variable, à un paramètre ou à un type)
- `-Wstack-protector` (avertit lorsqu'une fonction n'est pas instrumentée avec un canari)
- `-Wstrict-prototypes`
- `-Wswitch-enum` (avertit lorsque l'expression de contrôle d'une instruction `switch` est de type énuméré mais qu'il manque un `case` pour l'une ou plusieurs des constantes nommées définies avec ce type)
- `-Wmissing-prototypes`
- `-Wundef`
- `-Wvla`

Les options suivantes sont propres à GCC :

- `-Wduplicated-branches`
- `-Wduplicated-cond`
- `-Wformat-signedness`
- `-Wjump-misses-init`
- `-Wlogical-op` (avertit lors de l'utilisation suspecte d'un opérateur logique)
- `-Wnested-externs` (avertit lorsqu'une déclaration avec le spécificateur de classe de stockage `extern` est effectuée dans le corps d'une fonction)
- `-Wnormalized` (avertit lorsque des identifiants ne sont pas sous forme normalisée)
- `-Wold-style-definition`
- `-Wshift-negative-value`
- `-Wshift-overflow=2`
- `-Wstrict-overflow=3` (avertit de certains cas où le compilateur effectue des optimisations faisant l'hypothèse qu'il n'y a pas de débordement d'entiers signés)
- `-Wsuggest-attribute=format` (avertit des cas où l'ajout de l'attribut GCC `format` pourrait être bénéfique)
- `-Wsuggest-attribute=malloc` (avertit des cas où l'ajout de l'attribut GCC `malloc` pourrait être bénéfique)
- `-Wswitch-default` (avertit lorsqu'une instruction `switch` ne présente pas de cas par défaut)
- `-Wtraditional-conversion` (avertit lorsqu'un prototype de fonction provoque une conversion de type différente de celle qu'aurait subie le même argument en l'absence d'un prototype)
- `-Wtrampolines` (avertit de la génération de trampolines, mentionnés en note de bas de page dans la sous-section 5.3.5)

- `-Wwrite-strings` (ajoute le qualificatif de type `const` aux chaînes de caractères constantes afin de provoquer l'émission d'un avertissement lorsque l'adresse d'une telle chaîne est copiée dans un pointeur dont le type n'est pas qualifié `const` ; cela permet de détecter du code essayant d'écrire dans une chaîne de caractères constante — à condition d'avoir bien utilisé le mot-clé `const` dans les déclarations et prototypes, sans quoi cet avertissement devient inutilement très bruyant)

Les options suivantes sont propres à CLANG :

- `-Warray-bounds-pointer-arithmetic`
- `-Wassign-enum` (avertit lorsqu'une constante entière affectée à une variable de type énuméré n'est pas dans la plage de valeurs définie pour ce type)
- `-Wcast-function-type`
- `-Wcomma`
- `-Wcovered-switch-default`
- `-Wduplicate-enum`
- `-Widiomatic-parentheses` (avertit lorsqu'une affectation est utilisée comme condition sans être entourée de parenthèses)
- `-Wloop-analysis`
- `-Wformat-non-iso`
- `-Wformat-pedantic`
- `-Wformat-type-confusion`
- `-Wfour-char-constants`
- `-Wimplicit-fallthrough`
- `-Wpointer-arith`
- `-Wpragmas`
- `-Wreserved-identifier`
- `-Wshift-sign-overflow`
- `-Wsigned-enum-bitfield`
- `-Wstatic-in-inline`
- `-Wtautological-constant-in-range-compare`
- `-Wthread-safety`
- `-Wunreachable-code`
- `-Wunreachable-code-aggressive`
- `-Wunused-macros`
- `-Wused-but-marked-unused`
- `-Wvariadic-macros`
- `-Wzero-as-null-pointer-constant`

Information

L'option `-Wall` de CLANG active automatiquement l'option `-Wmost`, qui elle-même active de nombreux avertissements supplémentaires. Les options correspondant à ces derniers ne sont donc pas listées ci-dessus.

B.3 Clang et l'option `-Weverything`

CLANG possède une option `-Weverything`[17] qui active *tous* les avertissements supportés par CLANG sans exception.

L'utilisation de `-Weverything` peut être intéressante pour découvrir de nouveaux avertissements proposés par le compilateur ou en cas de niveau d'exigence maximum sur une certaine base de code. Elle ne doit cependant pas être systématique puisqu'elle est par exemple susceptible de poser problème au moment du *build* d'un projet après une mise à jour d'un des outils utilisés.

17. À ne pas confondre avec `-Wall`, `-Wextra` et `-Wmost`.

Annexe C
Mots réservés du C++

La liste suivante contient les mots réservés du C++ et qui n'appartiennent pas au langage C. Les mots suffixés par un astérisque sont des mots réservés ajoutés dans le C++11. Une sémantique supplémentaire a été ajoutée au mot réservé delete dans le C++11 lors de la déclaration d'une classe.

alignas *	const_cast	not_eq	this
alignof *	decltype *	nullptr *	throw
and	delete *	operator	
and_eq	dynamic_cast	or	try
asm	explicit	or_eq	typeid
thread_local *	export	override *	typename
bitand	final*	private	using
bitor	friend	protected	virtual
char16_t *	mutable	public	xor
char32_t *	namespace	reinterpret_cast	xor_eq
catch	new	static_assert *	
class	noexcept *	static_cast	
compl		template	
constexpr *			

Annexe D
Priorité des opérateurs

L'ordre adopté est par priorité décroissante. Les opérateurs présents dans la même cellule du tableau ont le même niveau de priorité, même s'ils sont situés sur une ligne différente de la cellule.

G. à D. signifie « associativité de gauche à droite », et D. à G. indique « associativité de droite à gauche ».

Catégorie	Op.	Nom	Associativité
Référence	() [] -> .	Appel de fonction Accès à un élément d'un tableau Accès à un champ d'une structure d'adresse donnée Accès à un champ d'une structure	G. à D.
Unaire	+ - ++ – ! ~ & (cast) sizeof	Identité Opposé Incrémentation Décrémentation Négation logique Inversion de tous les bits Référencement de pointeur Déréférencement de pointeur Transtypage Taille d'un objet	D. à G.
Arithmétique	* / % + -	Produit Division Modulo Somme de deux nombres ou d'un pointeur et d'un nombre Soustraction de deux nombres ou de deux pointeurs	G. à D.
Décalage	<< >>	Décalage binaire à gauche Décalage binaire à droite	G. à D.
Comparaison	< <= > >= == !=	Inférieur strictement à, inférieur ou égal à Supérieur strictement à, supérieur ou égal à Egal à Différent de	G. à D.
Traitement de bits	& ^ |	Et bit à bit Ou exclusif bit à bit Ou bit à bit	G. à D.
Logique	&& ||	Et logique Ou logique	G. à D.
Conditionnel	?:	Opérateur conditionnel ternaire	D. à G.
Affectation	= += -= *= /= %= &= ^= |= <<= >>=	Affectation Incrémentation, décrémentation, produit, division puis affectation Affectation modulo puis affectation...	D. à G.
Séquence	,	Séparateur d'arguments ou d'expressions	G. à D.

Annexe E
Exemple de conventions de développement

Au début de la réalisation d'un projet informatique, l'équipe de développement devrait toujours s'accorder sur les conventions de codage à appliquer. Le but est de produire un code source cohérent. Par ailleurs, le choix de conventions judicieuses permet de réduire les erreurs de programmation.

Information

Les points suivants proposent un **exemple de conventions de codage. Certains choix sont arbitraires et discutables. Cet exemple de conventions peut être repris ou servir de base, si aucune convention de développement n'a été définie pour le projet à produire. Différents outils ou des éditeurs avancés sont en mesure de mettre en oeuvre de façon automatique certaines de ces conventions de codage.**

Dans le cas où des conventions ont été définies dans le cadre de la réalisation d'un projet, le document spécifiant clairement ces conventions doit accompagner le projet en question.

E.1 Encodage des fichiers

Les fichiers sources sont encodés au format UTF8.

Le caractère de retour à la ligne est le caractère « line feed » \n (retour à la ligne au format Unix).

E.2 Mise en page du code et indentation

E.2.1 Longueurs maximums

Une ligne de code ou de commentaire ne doit pas dépasser 100 caractères.

Une ligne de documentation ne doit pas dépasser 100 caractères.

Un fichier ne doit pas dépasser 4000 lignes (documentation et commentaires compris).

Une fonction ne doit pas dépasser 500 lignes.

E.2.2 Indentation du code

L'indentation du code s'effectue avec des caractères espaces : un niveau d'indentation correspond à 4 caractères espaces. L'utilisation du caractère de tabulation comme caractère d'indentation est interdite.

La déclaration des variables et leur initialisation doivent être alignées à l'aide d'indentations.

Un caractère espace est laissé systématiquement entre un mot clé et la parenthèse ouvrante qui le suit.

L'accolade d'ouverture d'un bloc est placée sur une nouvelle ligne. L'accolade de fermeture de bloc est également placée sur une nouvelle ligne.

Un caractère espace est laissé avant et après chaque opérateur.

Un caractère espace est laissé après une virgule.

Le point-virgule indiquant la fin d'une instruction est collé au dernier opérande de l'instruction.

Dans le cas d'un appel d'une fonction avec de nombreux paramètres, s'il est nécessaire de placer les paramètres sur plusieurs lignes, ces paramètres sont indentés pour être positionnés au niveau de la parenthèse ouvrante de l'appel de la fonction.

Bon exemple

```
...
uint32_t processing_function(linked_list_t *p_param1, uint32_t ui32_param2,
                             const unsigned char *s_param3)
 {
   uint32_t ui32_result  = 0;
   element_t *pp_out_param4 = NULL;

   if ((NULL == p_param1) || (NULL == s_param3))
   {
     ui32_result = 0;
     goto End;
   }

   ui32_result = function_with_many_params(p_param1, ui32_param2, s_param3,
       pp_out_param4);

   if (1 == ui32_result)
   {
     ...
   }

   End :
   return ui32_result;
 }
```

E.3 Types standards

Dans le cas où l'en-tête `stdint.h` est présent, celui-ci doit être inclus afin de bénéficier des types entiers qu'il définit. En son absence, il est nécessaire de définir les types entiers tels qu'ils sont présentés dans la section 7.

Dans le cas où l'en-tête `stdbool.h` est présent, celui-ci doit être inclus afin de bénéficier du type booléen qu'il définit. En son absence, il est nécessaire de définir le type `bool` tel qu'il est présenté sur le code suivant (fichier d'en-tête provenant de la version 4.8.2 de GCC). Le type `_Bool` est défini pour les compilateurs compatibles avec la norme C99 et normes suivantes.

```
/* Copyright (C) 1998-2013 Free Software Foundation, Inc. */

/*
 * ISO C Standard : 7.16  Boolean type and values  <stdbool.h>
 */

#ifndef _STDBOOL_H
#define _STDBOOL_H

#ifndef __cplusplus

#define bool    _Bool
#define true    1
#define false   0

#else // __cplusplus

/* Supporting <stdbool.h> in C++ is a GCC extension.  */
#define _Bool   bool
#define bool    bool
#define false   false
#define true    true

#endif // __cplusplus

/* Signal that all the definitions are present.  */
#define __bool_true_false_are_defined   1

#endif  // stdbool.h
```

E.4 Nommage

E.4.1 Langue pour l'implémentation

La langue utilisée pour le nommage des bibliothèques, des fichiers d'en-tête, des fichiers sources, des macros, des types, des variables, des fonctions doit être l'anglais. Cette utilisation de l'anglais évite le mélange au sein du code de mots en français avec les mots clés du langage C qui sont en anglais. L'ensemble du code source produit est ainsi plus cohérent.

La langue utilisée pour la documentation et les commentaires doit être l'anglais et ce, dès le début du développement et pour l'intégralité de la documentation et des commentaires. [18]

E.4.2 Nommage des répertoires des fichiers sources

Les fichiers sources doivent être organisés en bibliothèques. Dans le cas d'une bibliothèque de taille importante, il est conseillé de créer une arborescence pour organiser les fichiers sources. Le répertoire de plus haut niveau doit être nommé avec le nom de la bibliothèque. Les sous-répertoires doivent être nommés tels qu'ils reflètent le critère de regroupement des fichiers sources.

18. Dans ce document, les commentaires sont des faux commentaires et expliquent les problèmes du code et c'est uniquement pour cette raison que ceux-ci sont rédigés en français et non en anglais.

L'exemple suivant présente l'organisation des répertoires pour une bibliothèque contenant des fonctions utilitaires :

Arborescence	Commentaire
utils	Répertoire de base de la bibliothèque
utils/includes	Répertoire contenant l'ensemble des fichiers d'en-tête de la bibliothèque (API)
utils/collection	Répertoire contenant l'implémentation de toutes les structures de données de type collections (listes, pile, tableau, table de hachage...)
utils/concurrency	Répertoire contenant l'implémentation des mutex, sémaphores, variables conditionnelles
utils/threads	Répertoire contenant l'implémentation des threads
…	…

E.4.3 Nommage des fichiers d'en-tête et des fichiers d'implémentation

Les fichiers d'en-têtes et les fichiers sources doivent être préfixés par le nom de la bibliothèque à laquelle ils appartiennent. Dans le cas où le nom de la bibliothèque est un nom long, il est judicieux d'utiliser une abréviation comme préfixe. Cette abréviation doit être choisie telle qu'elle n'entre pas en conflit avec une bibliothèque déjà existante (bibliothèques standards, bibliothèques tierces, …).

La liste suivante donne des exemples de noms de fichiers d'en-tête et de fichiers sources :

utils_linked_list.h, utils_linked_list.c, utils_mutex.h, utils_mutex.c, utils_thread.h, utils_thread.c, …

E.4.4 Nommage des macros

Les macros préprocesseurs doivent avoir des noms en capitales. Les mots composants le nom de la macro doivent être séparés par le caractère souligné. Le nom de la macro ne doit pas correspondre à un nom déjà existant d'une macro : par exemple une macro appartenant à un fichier d'en-tête d'une bibliothèque standard. Les paramètres d'une macro doivent respecter la convention de nommage des variables.

Bon exemple

```
#define LOG_DEBUG(sMessage) write_log_message(sMessage)
```

Le nom d'une macro, définie afin d'éviter l'inclusion multiple d'un fichier d'en-tête, reprend le nom du fichier d'en-tête en capitales. Le caractère point est substitué par un caractère souligné.

Bon exemple

```
#define UTILS_LINKED_LIST_H
```

E.4.5 Nommage des types

Le nom d'un type défini à l'aide de l'instruction `typedef` doit être écrit en minuscule et suffixé par `_t`. Les mots composant le nom du type doivent être séparés par le caractère souligné.

Lors de la définition d'un type pour une énumération ou une structure, le nom suivant le mot clé enum ou struct doit être suffixé par `_tag`. Le nom du type situé après l'accolade fermante de définition du type doit reprendre le même nom auquel `_tag` est substitué par `_t`.

Bon exemple

```
typedef enum status_tag {
  ...
} status_t;
typedef signed long sint32_t;
typedef struct linked_list_tag
{
  ...
} linked_list_t;
```

E.4.6 Nommage des fonctions

Le nom d'une fonction doit être préfixé par le nom (ou l'abréviation du nom) de la bibliothèque à laquelle elle appartient. Les mots composants le nom de la fonction doivent être séparés par le caractère souligné. Le nom d'une fonction doit être écrit en minuscule.

Bon exemple

```
status_t utils_create_linked_list(linked_list_t **pp_list);
status_t utils_delete_linked_list(linked_list_t *pp_list);
```

E.4.7 Nommage des variables

Les identifiants des variables seront composés de mots séparés par le caractère souligné, sans espace ni capitales. Chaque élément de l'identifiant permet de préciser la variable associée (type, signe, taille, rôle ....).

Le tableau suivant présente les préfixes pour les noms de variables en fonction du type, ainsi qu'un exemple pour chaque type de variable :

Préfixe	Type de variable	Exemple
i8	Entier sur 8 bits signé	int8_t i8_byte = 0;
ui8	Entier sur 8 bits non-signé	uint8_t ui8_byte = 0U;
i16	Entier sur 16 bits signé	int16_t i16_option = 0;
ui16	Entier sur 16 bits non-signé	uint16_t ui16_port = 0U;
i32	Entier sur 32 bits signé	int32_t i32_value = 0L;
ui32	Entier sur 32 bits non-signé	uint32_t ui32_counter = 0UL;
i64	Entier sur 64 bits signé	int64_t i64_big_value = 0LL;
ui64	Entier sur 64 bits non-signé	uint64_t ui64_big_counter = 0ULL;

Préfixe	Type de variable	Exemple
b	Booléen	bool b_is_set = false ;
c	Caractère	char c_letter = '\0' ;
f	Flottant	float f_value = 0.0f ;
d	Double	double d_precised_result = 0.0d ;
sz	Type size_t	size_t sz_string_length = 0U ;
e	Variable de type énuméré	status_t e_status_code = STATUS_ERR ;
st	Variable de type structure	linked_list_t st_list ;
a	Tableau	uint32_t a_values[10] ;
p	Variable de type pointeur	linked_list_t* p_list = NULL ;
pp	Variable de type pointeur de pointeur	linked_list_t** pp_list = NULL ;
s	Variable de type chaîne de caractères	char* s_message = NULL ;
ws	Variable de type chaîne de caractères en unicode	wchar_t* ws_message = NULL ;

E.5 Documentation

E.5.1 Format des balises pour la documentation

La documentation du code source doit être effectuée en utilisant le système de balises de l'outil *Doxygen*. Les balises *Doxygen* doivent toutes débuter par le caractère arobase *@*. L'outil *Doxygen* autorise également le caractère antislash. Cependant afin d'avoir une uniformité pour la documentation du code source, le préfixe arobase pour les commandes *Doxygen* est imposé.

Un commentaire de documentation débute par les caractères **/*!** et se termine par les caractères ***/**

Les points suivants présentent la documentation minimale qui doit être présente dans un fichier d'en-tête.

E.5.2 Cartouche d'en-tête des fichiers

Tous les fichiers d'en-tête et tous les fichiers sources doivent débuter par un cartouche d'en-tête destiné à identifier :

- le logiciel et / ou la bibliothèque auquel appartient le fichier d'en-tête / source ;
- La société (et éventuellement l'auteur) et le copyright associés au fichier ;
- La balise *Doxygen* @file. La balise @file peut être suivie optionnellement du nom du fichier. En l'absence du nom du fichier, le nom de celui-ci est automatiquement déduit à partir du fichier dans lequel la balise @file est située.

  Il est indispensable d'utiliser la balise @file dans les fichiers d'en-tête et les fichiers sources. En effet, en son absence la documentation sur les fonctions, les variables globales, les définitions de types et les énumérations présente dans le fichier n'est pas incluse dans la documentation *Doxygen* produite.

Si le fichier fait partie d'une bibliothèque, la commande @addtogroup <label> [titre] doit être utilisée. Elle permet de grouper la documentation de toutes les fonctions d'une bibliothèque au sein d'un module dans la documentation produite. Le label est le nom du groupe à utiliser dans tous les fichiers appartenant à la bibliothèque. Le titre est optionnel. Il est utilisé pour nommer le groupe dans la documentation.

La commande @addtogroup doit être complétée par le couple de balises @{ et @} afin de délimiter les éléments du fichier appartenant au groupe.

E.5.3 Documentation d'une structure

La définition d'une structure doit être documentée avec un commentaire précédent sa définition. Ce commentaire doit indiquer le rôle de la structure. Chaque champ de la structure doit être documenté.

E.5.4 Documentation d'une énumération

La définition d'une énumération doit être documentée avec un commentaire précédent sa définition. Ce commentaire doit indiquer dans quel cadre l'énumération doit être utilisée. Chaque valeur de l'énumération doit être documentée.

E.5.5 Documentation d'une variable globale

Une variable globale doit être documentée avec un commentaire précédent sa définition. Ce commentaire doit indiquer le rôle de la variable, sa valeur d'initialisation, les éventuels invariants qui doivent être respectés.

E.5.6 Documentation d'une fonction

La documentation d'une fonction doit précéder la définition du prototype de la fonction dans le fichier d'en-tête. La documentation d'une fonction est constituée de :

- un commentaire bref ;
- un commentaire détaillé expliquant la fonctionnalité offerte par la fonction ;
- la présentation de chaque paramètre, avec en précision s'il s'agit d'un paramètre en entrée, en sortie ou à la fois en entrée et en sortie ;
- la valeur retournée par la fonction. Dans le cas où il s'agit d'un code d'erreur, il doit être indiqué le ou les cas de succès, et les différents codes d'erreur pouvant être retournés ainsi que leur priorité ;
- une pré-condition si elle existe sur l'appel de la fonction ;
- une post-condition si existe, suite à l'appel de la fonction ;
- d'éventuelles remarques ou avertissements supplémentaires.

Bon exemple

Les lignes suivantes présentent la documentation minimale pour un fichier d'en-tête.

```
#ifndef UTILS_LINKED_LIST_H
#define UTILS_LINKED_LIST_H

/*!
* @file linked_list.h
* @author DEV 1
*
* @brief Linked List
*
* Function declarations for the manipulation of linked list.
*
* @addtogroup utils Library Utils
* @{
  */

  /*!
  * @brief Enumeration of status codes
  *
  * Status codes to indicate the success or the failure of functions
  */
  typedef enum status_tag {
    STATUS_SUCCESS = 0,      //!< success
    STATUS_GENERIC_ERROR,    //!< generic error
    STATUS_MEMORY_ERROR,     //!< memory allocation error
    STATUS_INVALID_PARAM     //!< invalid parameter
  } status_t;

  /*!
  * @brief Element of the linked list
  */
  typedef struct linked_list_element_tag
  {
    struct linked_list_element_tag*  pNext;     //!< next element
    struct linked_list_element_tag*  pPrevious; //!< previous element
    void*                            pData;     //!< data of the element
  } linked_list_element_t;

  /*!
  * @brief Double linked list
  *
  * Structure to define a double linked list. The type data of the list is void.
  */
  typedef struct linked_list_tag
  {
    linked_list_element_t *pHead;  //!< first element
    linked_list_element_t *pTail;  //!< last element
  } linked_list_t;

  /*!
  * @brief New linked list
  *
  * Creation of a new linked list by allocating the memory for the structure and by
       initializing the list.
  * The new list is empty.
  *
  * @param[out] ppList is the new list
  * @return #STATUS_SUCCESS the creation and the initialization are done with
      success
  * @return #STATUS_INVALID_PARAM if ppList is NULL or
  *                               if (*ppList) != NULL
  * @return #STATUS_MEMORY_ERROR fail of the memory allocation
  * @pre ppList != NULL and (*ppList) == NULL
  * @note the created list has to be deleted
  *       by calling #utils_delete_linked_list
  */
  status_t utils_create_linked_list(linked_list_t **ppList);
```

```
  /*!
  * @brief Deletion of the list
  *
  * All the elements of the list are deleted and the used memory is freed.
  * @warning The memory used by the data in the list is not freed..
  *
  * @param[in, out] ppList the list to delete.
  * @return #STATUS_SUCCESS if the deletion of the list is a success
  * @return #STATUS_INVALID_PARAM if  ppList is NULL
  *                                or if  (*ppList) is NULL
  * @pre ppList != NULL and (*ppList) != NULL
  * @post (*ppList) == NULL
  */
  status_t utils_delete_linked_list(linked_list_t **ppList);

  ...

  /*! @} */

#endif // UTILS_LINKED_LIST_H
```

Liste des recommandations

Index

Bibliographie

[float] *IEEE Standard for Floating-Point Arithmetic*.
Standard, IEEE.

[AnsiC90] *ISO/IEC 9899 :1990, Programming Languages - C*.
Norme, International Organization for standardization.

[AnsiC99] *ISO/IEC 9899 :1999, Programming Languages - C*.
Norme, International Organization for standardization.

[Cert] *SEI CERT C Coding Standard*.
Technical report, Carnegie Mellon University.

[ClangRef] *CLANG'S Documentation*.
Documentation publique, https ://clang.llvm.org/docs/.

[Cwe] *CWE Common Weakness Enumeration*.
Technical report, MITRE.

[GccRef] *GCC : Reference Documentation*.
Documentation publique, http ://www.gnu.org/software/gcc/onlinedocs.

[IsoSecu] *ISO/IEC TS 17961 Information Technology - Programming languages, their envrionments and system software interfaces - C Secure Coding Rules*.
Technical report, Switzerland, Genève.

[Misra2012] *MISRA-C :2012 Guidelines for the use of the C language in critical systems*.
Guide méthodologique, https ://www.misra.org.uk/MISRAHome/MISRAC2012.

ANSSI-PA-073
Version 1.3 - 16/11/2021

AGENCE NATIONALE DE LA SÉCURITÉ DES SYSTÈMES D'INFORMATION

ANSSI - 51, boulevard de La Tour-Maubourg, 75700 PARIS 07 SP
www.ssi.gouv.fr / conseil.technique@ssi.gouv.fr

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
Premier ministre
SGDSN