|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **개 정 주 기**  *Cycle of Revision* | | | | | 차기 개정시 | | | **제 / 개정, 폐지권자**  *Enactment, Revision & Abolition* | | | | | | IRIS 총괄 | | | | | |
| **기 안 부 서**  *Drafting* | | | | | | 사업관리팀 | | | | | |
| **보 관 기 한**  *Store Due Date* | | | | | 차기 개정시 | | | **관 리 부 서**  *Control* | | | | | | 품질팀 | | | | | |
|  | | | | | | | | | | | | | | | | | |  | |
|  | | | | | | | | | | | | | | | | | |  | |
|  | | | | | | | | | | | | | | | | | |  | |
| **관**  **련**  **표**  **준** |  | | | | | | **승 인**  *Approved By* | | | *20/08/17* | | | 윤예선 | | |  | | | |
|  | | | | | |
|  | | | | | | **검 토**  *Reviewed By* | | | *20/08/17* | | | 박명규 | | | 테이블이(가) 표시된 사진  자동 생성된 설명 | | | |
|  | | | | | |
|  | | | | | | **기 안**  *Drafting By* | | | *20/08/17* | | | 백성규 | | |  | | | |
|  | | | | | |
|  | | | | | |  | | | **Date** | | | **Name** | | | **Signature** | | | |
|  | | | | | |
| **검토부서** | **사업관리팀**  *Management of*  *Business Team* | | **개발팀**  *Technical & Development Team* | | | **구매자재팀**  *Purchasing/*  *Management of Material Team* | | | **인사총무팀**  *HR/Administration Department* | | | **생산**  **(기술)팀**  *Production Technology Team* | | | **품질팀**  *Quality Team* | | | **영업팀**  *Ssles Team* | |
| 백성규 | | 윤교열 | | | 노영주 | | | 김민성 | | | 이하영 | | | 유시문 | | | 박성우 | |
|  | | *비행, 비행기, 그룹, 조류이(가) 표시된 사진  자동 생성된 설명* | | |  | | | *스키타기, 언덕, 그리기, 남자이(가) 표시된 사진  자동 생성된 설명* | | |  | | |  | | |  | |
| **개정번호**  *Rev. No.* | | | **페이지**  *Page* | | **제 / 개정 내역**  *Enactment / Revision* | | | | | | | | | | | | | **일 자**  *Date* | |
| **~에서**  *Before* | | | | | | | **~으로**  *After* | | | | | |
| *0* | | | - | | 최 초 제 정 | | | | | | | | | | | | | 18.04.18 | |
| *1* | | | 전체 | | 유지보수 프로세스 개정됨 / 3.15항 형상변경 요청서 내용수정 / 3.16 형상변경 요청 결과서 추가 / 3.17 PR 삭제 / 3.18 -> 3.17 Software Revision History / 4.2 7) 수정 / 4.4 1) 수정 / 5.1, 5.3 1) 선행조건 삭제 / 5.3 3) 수정 / 5.8 SW 이력 리스트 문서번호 수정 및 SW 버전 기술서 삭제 / 6항 PR->이슈관리대장으로 변경 | | | | | | | | | | | | | 18.07.13 | |
| *2* | | | 1 | | 검토부서 생산기술팀 추가 | | | | | | | | | | | | | 19.01.31 | |
| *3* | | | 1 | | 대표자 변경 및 검토부서 추가(인사총무팀,영업팀),  생산-생산기술 통합 | | | | | | | | | | | | | 20.08.17 | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |
|  | | |  | |  | | | | | | |  | | | | | |  | |

**Content**

[**1.** **서론** 4](#_Toc3232323)

[1.1 목적 4](#_Toc3232324)

[1.2 범위 4](#_Toc3232325)

[1.3 참고 문서 4](#_Toc3232326)

[**2.** **코딩스타일 가이드라인** 5](#_Toc3232327)

[2.1 Types 5](#_Toc3232328)

[2.2 Names 5](#_Toc3232329)

[2.3 Indentation 5](#_Toc3232330)

[2.4 Definitions and macros 6](#_Toc3232331)

[2.5 Expressions 7](#_Toc3232332)

[2.6 Operators 7](#_Toc3232333)

[2.7 Functions 8](#_Toc3232334)

[2.8 Comments 8](#_Toc3232335)

[**3.** **Coding Standard** 9](#_Toc3232336)

[3.1 Company Rule 9](#_Toc3232337)

[3.2 MISRA Rule (Subset of MISRA 2012) 10](#_Toc3232338)

[3.3 Code Sonar 89](#_Toc3232339)

[**Annex A. Template** 101](#_Toc3232340)

[Head file template 101](#_Toc3232341)

[Source file template 103](#_Toc3232342)

[**Annex B. MISRA 2012 Rule set (Entire)** 105](#_Toc3232343)

# **서론**

## 목적

본 문서는 ADS Rail 에서 관리하는 협력업체에서 작성되는 DCU 소프트웨어의 코딩 스타일과 코딩 표준을 정의하기 위한 문서이다.

## 범위

본 문서의 코딩 표준은 차량 제작사인 현대 로템에서 요구하는 코딩 표준 52개를 포함한 ADS Rail 코딩 표준 기준이며, MISRA 2012에서 제시하는 모든 사항과 Code sonar에서 검출하는 모든 위배 사항을 포함하고 있지는 않다.

## 참고 문서

|  |  |  |
| --- | --- | --- |
| Publisher | Document Number | Title |
| Rotem | REDA202085 Rev.1 | 현대로템 C 코딩표준 (MISRA-C 2012) |
| Rotem | REDA200300 Rev.0 | 철도 차량 소프트웨어 CodeSonar 분석표준 |

# **코딩스타일 가이드라인**

## Types

기본 변수 타입을 그대로 사용하지 않고 크기와 부호를 나타내는 기본 변수 타입을 typedef을 통해 정의하여 다음과 같이 사용한다.

typedef char char\_t;

typedef signed char int8\_t;

typedef signed short int16\_t;

typedef signed int int32\_t;

typedef signed long int64\_t;

typedef unsigned char uint8\_t;

typedef unsigned short uint16\_t;

typedef unsigned int uint32\_t;

typedef unsigned long uint64\_t;

typedef float float32\_t;

typedef double float64\_t;

typedef long double float128\_t;

## Names

1. 변수 이름과 함수 명의 단어 구분은 대문자로 구분하고, 필요하다면 언더바(‘\_’)로 구분한다.

Ex)

uint8\_t SwitchOnFlag (O)

uint8\_t switch\_on\_flag (O)

uint8\_t switchonflag (X)

1. 단순 루프 또는 임시 저장 변수에만 짧은 이름을 쓸 수 있다. (아래 예와 같이 루프 인덱스에는 i,j,k 등도 사용 가능)

Ex)

void loop (void)

{

uint8\_t i;

uint8\_t sum;

for (i = 0; i > 5; i++)

{

sum += i;

}

}

1. 변수 및 함수명은 첫 번째 단어가 큰 의미를 부여하고 나중 단어가 구체적 의미를 갖도록 한다.
2. 포인터 선언에 사용하는 asterisk (‘\*’)는 변수명 또는 파라메터명에 붙여 사용한다.

Ex)

uint8\_t \*charpointer;

## Indentation

1. 중괄호 쌍의 여는 중괄호는 조건문 또는 함수 선언문 다음 줄부터 시작한다. 중괄호 사이의 명령문과 선언은 중괄호를 기준으로 들여쓰기를 한다.
2. 명령문과 선언은 오른쪽 중괄호를 기준으로 4칸 들여쓰기를 하며, 탭 대신 space를 사용한다.
3. 닫는 중괄호는 내용 코드와는 별개의 행에 있어야 하며, 여는 중괄호와 일치하는 열에 있도록 들여쓰기를 한다.

Ex)

void Function\_test (void)

{

uint8\_t level;

uint8\_t remainder;

uint8\_t i;

for (i = 0u; i < 100u; i ++)

{

remainder = i % 2;

if (0u == remainder)

{

level = i;

}

}

}

1. switch() 문안의 case는 switch 행보다 한 레벨 더 들여쓰기를 한다.

case 의 실행문들은 case 키워드보다 한 레벨 더 들여쓰기를 한다.

Ex)

switch (temp\_name)

{

case 1 :

break;

case 2 :

break;

default :

break;

}

## Definitions and macros

1. 전처리 정의는 항상 대문자로 한다. 가독성을 위해 언더바를 사용하여 단어를 구분한다.

Ex)

#define CLOSE\_SWITCH (O)

1. 전처리 정의가 숫자 또는 단일 이름 이상인 경우 전처리 과정에서 다른 의미로 해석되지 않도록 괄호를 추가한다.

Ex)

#define A

#define SUM\_DEFINE (A + 3)

## Expressions

1. 괄호 앞에는 항상 공백이 있거나 다른 괄호가 있어야 한다.

Ex)

if ((OPEN\_BRACKET && CLOSE\_BRACKET) || SPACE) (O)

if((OPEN\_BRACKET && CLOSE\_BRACKET) || SPACE) (X)

if( (OPEN\_BRACKET && CLOSE\_BRACKET) || SPACE) (X)

1. 닫는 괄호 앞에는 공백이 없어야 한다.

Ex)

if ((OPEN\_BRACKET && CLOSE\_BRACKET) || SPACE) (O)

if ((OPEN\_BRACKET && CLOSE\_BRACKET ) || SPACE ) (X)

1. 표현이 너무 복잡해지지 않도록 가능한 여러 하위 표현식으로 나누는 것이 바람직하다. 하나의 표현식이 한 행에 들어가지 않아 행을 분리하여 다음 행으로 이어지는 경우 표현식이 시작된 행과 동일한 열에서 시작되게 한다.
2. 비교문에서 0과 비교하는 경우는 0을 명시적으로 표시한다.

Ex)

if (a == 0) (O)

if (a) (X)

1. 배열 변수의 이름과 첫 번째 ‘[’의 사이에는 공백을 사용하지 않는다.

Ex)

Variable\_name[5]; (O)

Variable\_name [5]; (X)

1. ‘[’ 과 ‘]’안의 값과 공백을 사용하지 않고 사용한다.

Ex)

uint32\_t aaa[15] (O)

uint32\_t aaa[ 15 ] (X)

1. 조건문은 다음과 같이 사용한다 : if (6 == errorNum)

## Operators

1. 이진 연산자는 앞 뒤로는 공백이 있어야 한다.

Ex)

if (sum || count)

temp = a\_count & b\_count;

1. 단항 연산자는 변수 또는 식에 항상 붙어있어야 한다. ‘~’와 같은 단항 연산자는 char 또는 unsigned char에 적용하더라도 기본 C타입인 int가 반환되니 주의 하여야 한다.

Ex)

~status; (O)

~ status; (X)

1. 삼항 연산자(?:)는 사용하지 않는다.
2. 형 변환은 변환이 수행되는 객체 앞에 공백 없이 사용한다.

Ex)

b = (uint8\_t)a;

## Functions

1. 함수 호출을 할 때 함수명과 매개 변수 목록의 괄호 사이에는 한 개의 공백이 있어야 한다.

Ex)

void main (void)

{

uint8\_t a, b, sum;

a = 5;

b = 5;

sum = a + b;

sub\_function (sum);

}

1. 전달하는 매개 변수가 없는 경우는 여는 괄호와 닫는 괄호 사이에 공백이 없어야 한다.
2. 함수명 뒤의 괄호는 매개변수 형과 여는 괄호 사이에 공백이 없도록 붙여서 사용해야 하고 닫는 괄호 앞에도 공백이 없어야 한다.

Ex)

uint8\_t Open\_control (uint8\_t current, uint8\_t position) (O)

uint8\_t Open\_control ( uint8\_t current, uint8\_t position ) (X)

1. 매개변수를 구분하는 매개변수 사이의 콤마 뒤에는 공백이 있어야 한다.

Ex)

uint8\_t Function\_name (uint8\_t sum, uint8\_t count)

Function\_name (sum, count);

1. 함수 매개 변수 목록이 최대 줄 길이를 초과하면 다음 줄을 사용할 수 있다.

## Comments

1. 주석이 있는 행에 이어서 코드를 작성하지 않는다. 주석을 사용할 때에는 닫는 주석이 항상 해당 줄의 마지막에 있어야 한다.

Ex)

/\* comments \*/ a = 3; (X)

1. 주석 다음에 여러 행의 실행문이 오는 경우 주석은 실행문과 같은 열에서 시작하는 것이 바람직하다.
2. 한 줄의 주석의 경우 C++스타일의 주석(//)을 사용할 수 없다.

# **Coding Standard**

## Company Rule

1. 컴포넌트의 제약

컴포넌트의 크기와 복잡도는 하기의 기준을 권장한다.

|  |  |  |
| --- | --- | --- |
| **Item** | **Criteria** | **Remark** |
| LOC (Module Size) | <= 1000 |  |
| Cyclomatic complexity | <25 |  |
| Comment Writing Rate | > 35% |  |

1. 코딩 규칙 (CORULE\_01\_01)

If문 뒤에 else 문이 오도록 권장한다. 마지막 else문의 요구는 방어적 프로그래밍을 위함이다. else 문에서 적절한 행동을 취하거나 아무 행동을 취하지 않으면 적절한 주석을 달아야 한다.

Ex)

if (u32a > 5u)

{

....

}

/\* Not compliant. no else clause \*/

if (u32a < 5u)

{

....

}

else

{

/\* Intentional empty else \*/

}

/\* Compliant \*/

## MISRA Rule (Subset of MISRA 2012)

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_04\_08 - 구조체 또는 공용체에 대한 포인터가 변환 단위 내에서 절대 참조 해제되지 않으면 객체의 구현은 숨겨져야만 한다. | |
| **카테고리** | Advisory | |
| **설명** | 구조체 또는 공용체에 대한 포인터가 절대 참조 해제되지 않으면 객체는 필요 없으며 그 내용은 의도하지 않은 변경으로부터 보호 되어야한다. 구현 세부 사항을 숨기면 포인터를 통해 참조 될 수 있지만 내용에 액세스 할 수 없는 불투명 한 유형이 작성된다. | |
| **예제코드- 별도의 헤더 파일에 정의 되어있는지 검사** | | |
| **BAD** | | **GOOD** |
| string\_m.c:  struct string\_mx {  size\_t size;  size\_t maxsize;  unsigned char strtype;  char \*cstr;  };  typedef struct string\_mx string\_mx;  /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/ | | string\_m.c:  struct string\_mx;  typedef struct string\_mx string\_mx;  /\* Function declarations \*/  extern errno\_t strcpy\_m(string\_mx \*s1, const string\_mx \*s2);  extern errno\_t strcat\_m(string\_mx \*s1, const string\_mx \*s2);  /\* ... \*/  string\_m.h:  struct string\_mx {  size\_t size;  size\_t maxsize;  unsigned char strtype;  char \*cstr;  }; |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_04\_12 – 동적메모리 할당은 사용되지 않아야 한다. | |
| **카테고리** | Advisory | |
| **설명** | 표준 라이브러리의 동적 메모리 할당 및 할당 취소 루틴은 정의되지 않은 행동은 규칙 21.3에 설명 된 바와 같다. 다른 모든 동적 메모리 할당 시스템이 나타날 수 있다. 표준 라이브러리와 유사하게 정의되지 않은 행동을합니다. 제 3 자 루틴의 세부 사항은 동적 메모리 할당  부주의로 사용되지 않습니다. 동적 메모리를 사용하기로 결정한 경우 소프트웨어가 올바르게 작동하는지 주의해야한다.  예측 가능한 방식으로 예를 들어 다음과 같은 위험이 있다.   * 요구를 충족시키기 위해 충분한 메모리가 필요할 수 있다. * 할당 실패에 대해 안전하고 적절한 대응이 있다는 것을 의미한다. * 할당 또는 할당 취소를 수행하는 데 필요한 실행 시간에 큰 차이가 있습니다.   사용 패턴 및 결과적인 조각화 정도에 따라 다릅니다. | |
| **예제코드- 특정 함수 사용 금지(malloc, calloc, realloc)** | | |
| **BAD** | | **GOOD** |
|  | |  |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_01\_03 - 정의되지 않거나 명시되지 않은 행동이 발생하면 안 됨 | |
| **카테고리** | Required | |
| **설명** | 프로그램이 예상한 방식으로 동작하지 않으면 정의되지 않은 행동(Undefined behavior)이거나 명시되지 않은 행동(Unspecified behavior)이다. 단순히 이식성의 문제일 수도 있으나, 더 심각한 문제가 발생할 수도 있다. 예를 들면, 정의되지 않은 행동은 계산 결과를 다르게 할 수 있다. 소프트웨어의 동작이 이 계산에 달려있다면 시스템의 안전도 보장할 수 없다. 문제는 이런 정의되지 않은 행동이 아주 드문 상황에만 나타나기 때문에 찾기가 매우 어렵다는 것이다.  MISRA C:2012의 많은 규칙들이 이 문제를 피하기 위해 설계되었다. 예를 들어 Rule 11.4, Rule 11.8, Rule 19.2는 C90의 [Undefined 39], C99의 [Undefined 61]을 피하기 위해 정의되었다. 그 외 항목들에 대해서는 특정 규칙으로 정의하지 않았는데 그 이유는 거의 일어날 가능성이 없거나, 아니면 현실적인 가이드라인을 정의할 수 없기 때문이다. 그러나 구체적인 가이드라인을 제공하지 못하더라도 이 항목들을 매우 중요하게 취급되도록 하기 위해 별도의 규칙으로 정의한다.  기존 MISRA C:2004의 Rule 1.2, Rule 18.1, Rule 19.8, Rule 19.14, Rule 20.6이 이 규칙으로 병합되었다(MISRA C:2012 Addendum 1 — Rule Mappings 참조).  See also Dir 4.1 | |
| **예제코드- #if defined 이후에 identifier, (identifier)의 형태만 사용 가능** | | |
| **BAD** | | **GOOD** |
| #if defined (MIN > 0) /\* not compliant ; invalid defined directive form, may not compile \*/  #define MIN 0  #endif  #if defined MIN > 0 /\* not compliant ; invalid defined directive form \*/  #define MAX 100  #endif | | #if defined MIN /\* Compliant \*/  #define MIN 0  #endif  #if defined MAX /\* Compliant \*/  #define MAX 100  #endif |
| **예제코드- 모든 구조체와 union타입이 complete 한지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  extern struct st \*pst1;  extern struct st \*pst2;  struct {  int8\_t a;  //int8\_t b [ ]; /\* not compliant ; incompleted struct \*/  } st\_ = { 1, { 2, 3, 4 } };  void bad(void)  {  struct st1 s; /\* not compliant ; incompleted struct \*/  \*pst2 = \*pst1; /\* not compliant ; incompleted struct \*/  } | | #include "../basic\_types.h"  extern struct st \*pst1;  extern struct st \*pst2;  struct {  int8\_t a;  int8\_t b [ 3 ]; /\* Compliant \*/  } st\_ = { 1, { 2, 3, 4 } };  void bad(void)  {  struct st1  {  int8\_t x;  }s; /\* Compliant \*/  pst2 = pst1; /\* Compliant \*/  } |
| **예제코드- 함수 매크로 사용시 파라미터와 인자의 개수가 일치하는지 여부 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  #define SUM(x,y) ((x) + (y))  void bad\_2(int32\_t num)  {  int32\_t a = SUM(num); /\* not compliant ; lack of function macro argument \*/ /\*  } | | #include "../basic\_types.h"  #define SUM(x,y) ((x) + (y))  void bad\_2(int32\_t num)  {  int32\_t a = SUM(num, num); /\* Compliant \*/ |
| **예제코드- #if, #elif 이후에 매크로 확장에 의한 defined가 사용금지** | | |
| **BAD** | | **GOOD** |
| #define DEFINED defined  #if DEFINED(CS\_1914\_H) /\* Not compliant \*/ /\* expanded defined \*/  #define CS\_1914\_H4\_ 0  #endif | | #if defined(CS\_1914\_H) /\* Compliant \*/  #define CS\_1914\_H4\_ 0  #endif |
| **예제코드- 특정 매크로 사용 금지(offsetof)** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  #include <stdio.h>  #include <stddef.h>  void bad\_2(int32\_t num)  {  printf ("%d",(int)offsetof(struct timer,COUNT));; /\* not complaint; offsetof is prohibited macro \*/  } | |  |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_02\_01 - 도달할 수 없는 코드(unreachable code)가 있으면 안 됨 | |
| **카테고리** | Required | |
| **설명** | 프로그램에서 undefined behaviour가 발생하지 않는 한, 도달할 수 없는 코드는 실행될 수 없고, 프로그램의 결과에 어떤 영향도 줄 수 없다. 따라서 도달할 수 없는 코드가 존재한다는 것은 프로그램 로직에 오류가 존재할 가능성이 있음을 의미하므로 삭제하는 것을 권장한다.  컴파일러들은 도달할 수 없는 코드들을 삭제할 수 있다. 단, 다음과 같이 컴파일러가 바이너리를 생성하는 동작에 영향을 주는 코드는 삭제하지 않는다. 이 규칙에서도 이런 의도가 포함된 코드는 위배로 보지 않는다.  - 타겟 머신의 메모리를 차지한다.  - 컴파일러가 도달할 수 없는 코드 주위의 소스코드를 번역할 때, 더 길고 느린 점프 명령어를 선택하도록 한다.  - 루프 내에서, 도달할 수 없는 코드는 전체 루프가 캐쉬 내에 상주하는 것을 방해한다.  또한 예외 상황을 처리하기 위한 방어 코드가 도달할 수 없는 코드가 되는 경우가 있다. 예를 들어, Rule 16.4를 준수하기 위해 switch에는 항상 default가 존재해야 한다. 모든 도달 가능한 값들을 처리할 수 있는 case가 존재하더라도 default를 넣는 이유는, 프로그램 내의 undefined behaviour 혹은 하드웨어 고장으로 인해 발생할 수 있는 예외적인 값들을 처리하기 위해서이다. 이런 방어 코드는 이 규칙에서 위배로 검출하지 않는다.  어떤 컴파일러는 default가 도달 불가능한지를 분석하여 삭제하기도 한다. 그러나 반드시 default 방어 코드가 필요한 경우, volatile을 이용하여 컴파일러가 삭제하지 않도록 할 수 있다. switch (\*(volatile uint16\_t\*)&x)와 같이 작성하면 컴파일러는 이 조건식에는 어떤 값도 들어올 수 있다고 가정하므로, default가 삭제되지 않는다.  See also Rule 14.3, Rule 16.4 | |
| **예제코드- 함수 내 도달 불가능한 코드 사용 금지** | | |
| **BAD** | | **GOOD** |
| void error\_handler() {  // do something  return;  }  enum light { red, amber, red\_amber, green };  enum light next\_light ( enum light c )  {  enum light res;  switch ( c )  {  case red:  res = red\_amber;  break;  case red\_amber:  res = green;  break;  case green:  res = amber;  break;  case amber:  res = red;  break;  default:  {  /\*  \* This default will only be reachable if the parameter c  \* holds a value that is not a member of enum light.  \*/  error\_handler();  break;  }  }    return res;  res = c; /\* Non-compliant - this statement is certainly unreachable \*/  } | | void error\_handler() {  // do something  return;  }  enum light { red, amber, red\_amber, green };  enum light next\_light ( enum light c )  {  enum light res;  switch ( c )  {  case red:  res = red\_amber;  break;  case red\_amber:  res = green;  break;  case green:  res = amber;  break;  case amber:  res = red;  break;  default:  {  /\*  \* This default will only be reachable if the parameter c  \* holds a value that is not a member of enum light.  \*/  error\_handler();  break;  }  }    res = c;  return res;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_02\_02 - 죽은 코드(dead code)는 없어야 함 | |
| **카테고리** | Required | |
| **설명** | 실행 가능하지만 제거하더라도 프로그램의 동작에 영향을 주지 않는 모든 연산을 죽은 코드(dead code)라고 한다. 언어 확장에 의한 연산은 항상 프로그램 동작에 영향이 있다고 가정해서 죽은 코드로 보지 않는다.  참고로 Rule 2.1에서 언급한 도달할 수 없는 코드는 실행될 수 없기 때문에 죽은 코드는 아니다.  죽은 코드가 있다는 것은 프로그램 로직에 오류가 존재할 가능성을 암시한다. 이런 코드들은 컴파일러에 의해 삭제될 수 있으므로, 남겨 두면 혼란을 야기할 수 있으므로 삭제를 권장한다.  단, void로의 변환은 의도적으로 사용되지 않는 값을 가리킨다. 그러므로 void로의 변환 자체는 죽은 코드가 아니며, 그 피연산자를 사용하도록 하는 것 또한 죽은 코드가 아니다.  See also Rule 17.7 | |
| **예제코드- 계산 결과가 사용되지 않는 문장(dead code) 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  uint16\_t v;  char\_t \*p;  void f(void)  {  (void) v; /\* compliant - v is accessed for its side effect and the cast to void is permitted by exception \*/  (int32\_t) v; /\* Non-compliant - the cast operator is dead \*/  v >> 3; /\* Non-compliant - the >> operator is dead \*/  \*p++; /\* Non-compliant - result of \* operator is not used \*/  ( \*p ) ++; /\* Compliant - \*p is incremented \*/  }  void f2(void)  {  \*p = 1;  (\*p)++;  } | | #include "../basic\_types.h"  uint16\_t v;  char\_t \*p;  int32\_t f(void)  {  uint16\_t x;  int32\_t z;  (void) v; /\* Compliant - v is accessed for its side effect and the cast to void is permitted by exception \*/  z = (int32\_t) v; /\* Compliant \*/  v = v >> 3; /\* Compliant \*/  x = 3; /\* Compliant \*/  \*p++ = 2; /\* Compliant \*/  ( \*p ) ++; /\* Compliant - \*p is incremented \*/  return z + x + v;  }  void f2(void)  {  \*p = 1;  (\*p)++;  } |
| **예제코드- 타입이 void 이고 함수 내부(body)가 비어있다면, 이 함수를 호출하는 것은 죽은 코드이므로 호출을 제한함** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void g(void);  void h(void);  void g(void)  {  /\* compliant - there are no operations in this function \*/  }  void h(void)  {  g(); /\* Non-compliant - the call could be removed \*/  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  class A  {  public:  A() { }  A(int x) : mem(x) { }  A(const A& rhs) { }  ~A() { }  void setMem(int x) { mem = x; display(); } /\* Non-compliant - display has empty body \*/  void display(void) const { }  int getMem(void) const { return mem; }  private:  int mem;  };  void fA(int x)  {  A obj; /\* compliant - empty constructor is valid \*/  obj.setMem(x);  obj.display(); /\* Non-compliant - display has empty body \*/  obj.~A(); /\* compliant - empty destructor is valid \*/  A obj2(obj); /\* compliant - empty copy constructor is valid \*/    A\* pObj = new A(x); /\* compliant - empty constructor is valid \*/  pObj->getMem(); /\* compliant - all functions are assumed to have side effects. \*/  pObj->display(); /\* Non-compliant - display has empty body \*/  delete pObj; /\* compliant - empty destructor is valid \*/  } | | void g(void);  void h(void);  void g(void)  {  /\* compliant - there are no operations in this function \*/  }  void h(void)  {  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_05\_01 – 외부 식별자는 구별되어야한다. | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 컴파일러 자체 번역 제한(translation limits) 내에서 외부 식별자가 구별되어야 함을 말한다. "구별됨"의 정의는 컴파일러마다, 그리고 언어 표준 버전에 따라 다를 수 있다. 대부분의 컴파일러들은 외부 식별자를 최소 31자이상, 대소문자 구분하여 비교했을 때 구별되어야 한다고 정의한다. 만약 두 식별자가 중요 문자 안에서 차이가 없으면, 정의하지 않은 행동을 일으킬 수 있다. 이식성이 중요하다면, 현재 사용 중인 컴파일러보다 표준에서 정의한 최소한의 제한을 따르는 것이 안전하다. 너무 긴 식별자는 가독성을 저하시킨다. 자동으로 생성한 코드는 긴 식별자를 가지기 쉬운데, 가능하면 식별자 길이를 표준에서 정의하는 제한하는 것이 좋다.  See also  Dir 1.1, Rule 5.2, Rule 5.5 | |
| **예제코드- 시스템에서 external identifier가 식별 가능한지 검사 (31 글자까지 식별가능한지 검사)** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_01.cpp \*/  #include "../../basic\_types.h"  /\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/  int32\_t engine\_exhaust\_gas\_temperature\_raw; /\* :([MISRA\_C\_2012\_05\_01,707,engine\_exhaust\_gas\_temperature\_raw,2] \*/  int32\_t engine\_exhaust\_gas\_temperature\_scaled; /\* Non-compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_01\_02.cpp \*/  #include "../../basic\_types.h"  /\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\*\* \*/  int32\_t example\_identifier\_for\_this\_rule; /\* not compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_01.cpp \*/  #include "../../basic\_types.h"  /\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/  int32\_t engine\_exhaust\_gas\_temp\_raw;  int32\_t engine\_exhaust\_gas\_temp\_scaled; /\* Compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_01\_02.cpp \*/  #include "../../basic\_types.h"  /\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\*\* \*/  int32\_t ex\_identifier\_for\_this\_rule; /\* compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_05\_03 - 안쪽 scope의 식별자를 바깥 scope의 식별자가 가리면 안 됨 | |
| **카테고리** | Required | |
| **설명** | 안쪽 scope의 식별자는 어떤 바깥 scope의 식별자와도 구별할 수 있어야 한다.  "구별됨"의 정의는 컴파일러마다, 그리고 언어 표준 버전에 따라 다를 수 있다. 대부분의 컴파일러들은 외부 식별자를 최소 31자이상, 대소문자 구분하여 비교했을 때 구별되어야 한다고 정의한다.  안쪽 scope에 선언된 식별자가 바깥 scope에 선언된 식별자와 구별되지 않으면, 가장 안쪽의 식별자 선언은 가려지게(hide) 된다. 이는 개발자에게 혼란을 준다.  참고록 서로 다른 name space 에 있는 식별자 끼리는 서로 가리지 않는다.  안쪽(inner)와 바깥(outer) scope는 다음과 같이 설명할 수 있다.  - file scope 는 가장 바깥이다.  - 블록 scope가 더 안쪽이다.  - 연속, 중첩된 블록들은 더 안쪽이다.  See also Rule 5.2, Rule 5.8 | |
| **예제코드- scope 상에서 동일한 identifier 가리는지 검사** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "../basic\_types.h"  void func(int32\_t i)  {  int32\_t y = 3;  if(i > 10){  const int32\_t i = y; /\* Non-compliant \*/  const int32\_t y = 3; /\* Non-compliant \*/  {  const int32\_t y = 0; /\* Non-compliant \*/  }  }  }  int16\_t xyz = 0;  void func2(int32\_t xyz){ /\* Non-compliant \*/  int32\_t z = 10;  if(xyz == 10){  int32\_t z = 3; /\* Non-compliant \*/  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void f()  {  int i;  {  int i; /\* Non-compliant \*/  i = 3;  }  }  struct abc  {  int m;  };  int xyz = 0;  void ff()  {  struct abc xyz; /\* Non-compliant \*/  }  int s;  void fff()  {  typedef float s; /\* Non-compliant \*/  } | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  int variable;  void func (void) {  int local\_variable; /\* Compliant \*/  local\_variable =10;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void f()  {  int i;  {  int ii;  i = 3;  }  }  struct abc  {  int m;  };  int xyz = 0;  void ff()  {  struct abc xyzw;  }  int s;  void fff()  {  typedef float ss;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_05\_08 – 외부 연결을 가진 대상이나 기능을 정의하는 식별자는 고유해야한다. | |
| **카테고리** | Required | |
| **설명** | 외부 연결을 갖는 식별자는 어떤 name space나 translation unit에서도 다른 목적으로 사용되면 안 된다. 이런 식으로 식별자 이름의 유일성을 강조하면 개발자의 혼란을 막을 수 있다. 연결(linkage) 정보가 없는 지역 변수 식별자는 이런 혼란의 위험 요소가 적으므로 유일하지 않아도 된다.  See also  Rule 5.3 | |
| **예제코드- 시스템에서 external linkage 함수의 identifier가 유일한지 검사** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_01.c \*/  void foo(void) /\* not compliant \*/ /\* :([MISRA\_C\_2012\_05\_08,723,foo,2] \*/  {  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_02.c \*/  static void foo(void)  {  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_01.c \*/  void foo(void) /\* compliant \*/  {  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_02.c \*/  static void foo2(void)  {  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |
| **예제코드- 시스템에서 external linkage 변수의 identifier가 유일한지 검사** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_01.c \*/  extern int count; /\* not compliant \*/ /\* :([MISRA\_C\_2012\_05\_08,724,count,2] \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_02.c \*/  int count;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_01.c \*/  extern int count; /\* compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* File: 05\_08\_02.c \*/  int count2;  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_06\_01 - bit-field는 올바른 타입으로 선언되어야 함 | |
| **카테고리** | Required | |
| **설명** | 올바른 bit-field 타입은 다음과 같다:  - C90 : unsigned int 또는 singed int  - C99 : 다음 중 하나:  \* unsinged int 또는 singed int  \* 컴파일러가 제공하는 다른 명시적인 부호가 있는 integer 타입  \* \_Bool  int는 컴파일러에 따라 signed일 수도 있고 unsigned일 수도 있기 때문에 int를 사용하는 것은 컴파일러에 따라 다르다.  enum, short, char 나 다른 타입의 bit-field 는 C90에서 정의하지 않은 행동이므로 허용되지 않는다.  C99에서는 bit-field 선언에서 다른 integer 타입을 사용하는 것을 허용한다. | |
| **예제코드- bit 필드타입의 base 타입은 signed int 혹은 unsigned int 여야 함** | | |
| **BAD** | | **GOOD** |
| typedef unsigned int UINT\_16;  struct s{  unsigned int b1:2; /\* Compliant \*/  int b2:2; /\* Not compliant \*/  UINT\_16 b3:2; /\* Compliant \*/  signed long b4:2; /\* Not compliant \*/  }; | | typedef unsigned int UINT\_16;  struct st{  unsigned int b1:2; /\* Compliant \*/  unsigned int b2:2; /\* Compliant \*/  UINT\_16 b3:2; /\* Compliant \*/  signed int b4:2; /\* Compliant \*/  }; |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_07\_04 – 객체의 타입이 "const-qualified char에 대한 포인터"가 아닌 한 문자열 리터럴은 객체에 할당되어서는 안된다. | |
| **카테고리** | Required | |
| **설명** | 문자열 상수를 변경하는 것은 정의되지 않은 동작을 일으킬 수 있다. 예를 들어 어떤 컴파일러는 문자열 상수를 read-only 메모리에 저장하는데, 이런 경우 런타임에 예외나 crash가 발생할 수 있다. 따라서 const char \*로 선언한 변수에 할당해서, 값을 변경하지 못하도록 하는 것이 안전하다. 또한 & 연산의 결과는 const char(\*)[] 타입 변수에 할당해야 안전하다. wide 문자열도 마찬가지이다. wide 문자열을 const wchar\_t \* 타입이 아닌 변수에 할당하면 안 된다. C99 표준에 문자열 저장 위치에 대해 명시되어 있지 않으므로 문자열을 성공적으로 바꾼 것 같아도 다른 문자열이 의도하지 않게 수정될 수도 있다.  See also  Rule 11.4, Rule 11.8 | |
| **예제코드- 문자열의 직접적인 변경 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void func(void)  {  "0123456789"[0] = '\*'; /\* Not compliant \*/  char \*s = "string"; /\* Not compliant \*/  const volatile char \*p = "string"; /\* Compliant \*/  char \*ex;  ex = "string"; /\* Not compliant \*/  }  extern void func1(char \*s1);  extern void func2(const char \*s2);  void bad(void)  {  func1("string"); /\* Not compliant \*/  func2("string"); /\* Compliant \*/  }  char \*badReturn(void)  {  return ("string"); /\* Not compliant \*/  }  const char \*goodReturn(void)  {  return ("string"); /\* Compliant \*/  } | | #include "../basic\_types.h"  void func(void)  {  const char \*s = "string"; /\* Compliant \*/  const volatile char \*p = "string"; /\* Compliant \*/  const char \*ex;  ex = "string"; /\* Compliant \*/  }  extern void func1(const char \*s1);  extern void func2(const char \*s2);  void bad(void)  {  func1("string"); /\* Compliant \*/  func2("string"); /\* Compliant \*/  }  const char \*badReturn(void)  {  return ("string"); /\* Compliant \*/  }  const char \*goodReturn(void)  {  return ("string"); /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_01 - 타입은 명시적으로 입력해야 함 | |
| **카테고리** | Required | |
| **설명** | C90 표준은 상황에 따라 타입을 명시하지 않아도 int 타입으로 인식한다. 이러한 상황의 예로는 다음이 있다.  - 객체 선언  - 파라미터 선언  - 멤버 선언  - typedef 선언  - 함수 반환 타입  명시적으로 타입을 입력하지 않으면 혼란을 줄 수 있다. 예를 들어, 다음의 선언  extern void g ( char c, const k );  에서 k의 타입은 const char를 기대하는 것과는 달리 const int 이다.  See also Rule 8.2 | |
| **예제코드- 타입이 명시적으로 입력되었는지 검사** | | |
| **BAD** | | **GOOD** |
| - | | - |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_02 - 함수는 이름 있는 파라미터로 구성된 프로토타입 형태여야 함 | |
| **카테고리** | Required | |
| **설명** | 인자(argument)와 매개변수(parameter) 간의 타입, 그리고 함수의 리턴 타입과 실제 리턴 타입의 불일치는 정의되지 않은 결과를 발생시킨다.  이 규칙과 [MISRA\_C\_2012\_08\_01], [MISRA\_C\_2012\_08\_04] 의 목적은 매개변수 타입과 함수 리턴 타입을 명시적으로 입력해서 이러한 미정의된 특성을 피하는 것이다.  또한, 이 규칙은 선언 시에 모든 매개변수의 이름을 명시하도록 요구한다. 매개변수의 이름은 함수 인터페이스에 관한 정보를 제공할 수 있고, 컴파일러가 선언과 정의간의 불일치와 관련된 경고를 출력할때, 더 자세한 정보를 제공할 수 있게 한다.  참고: 함수의 파라메터가 없는 경우에 void 를 명시해야한다. C90/99 에서 int f(); 의 의미는 함수 인자의 개수와 타입을 모르는 경우에 사용하고, int f(void); 는 인자가 없음을 의미한다. C++ 에서는 int f(); 와 int f(void); 동일한 의미를 갖는다.  See also Rule 8.1, Rule 8.4, Rule 17.3 | |
| **예제코드- 함수의 정의 또는 호출 이전에 선언이 존재하는지 검사(definition). 단, 함수 호출은 호출 이전에 정의가 있다면 선언이 없어도 검출하지 않음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int16\_t noFuncDecl ( int16\_t n ) /\* Non-compliant \*/  {  return n;  } | | #include "../basic\_types.h"  int16\_t noFuncDecl ( int16\_t n );  int16\_t noFuncDecl ( int16\_t n ) /\* Compliant \*/  {  return n;  } |
| **예제코드- 함수 호출 시 인자의 개수와 파라미터의 개수가 일치하는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void func(void);  void func2(int i, int j);  void func(void) {  func2(1, 2, 3); /\* Non-compliant \*/  }  void func2(int i, int j) {} | | #include "../basic\_types.h"  void func(void);  void func2(int i, int j);  void func(void) {  func2(1, 2); /\* Compliant \*/  }  void func2(int i, int j) {} |
| **예제코드- 함수의 프로토타입 선언 시 파라미터의 이름이 존재하는지 검사** | | |
| **BAD** | | **GOOD** |
| - | | - |
| **예제코드- 함수의 선언 또는 정의에서 파라미터가 없을 경우 명시적인 void 타입 파라미터를 선언했는지 검사** | | |
| **BAD** | | **GOOD** |
| - | | - |
| **예제코드- 함수 정의 시 old style parameter 선언 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int16\_t func3(int16\_t\* vec, int16\_t n);  static int16\_t func3 ( vec, n ) /\* Non-compliant - old style identifier and declaration list \*/  int16\_t\* vec;  int16\_t n;  {  return vec[ n - 1 ];  } | | #include "../basic\_types.h"  static int16\_t func3 ( int16\_t\* vec, int16\_t n );  static int16\_t func3 ( int16\_t\* vec, int16\_t n ) /\* Compliant \*/  {  return vec[ n - 1 ];  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_03 - 동일한 객체 또는 함수의 모든 선언은 같은 이름과 타입 한정자를 사용해야 함 | |
| **카테고리** | Required | |
| **설명** | 객체 또는 함수를 선언하거나 정의할 때, 타입과 한정자(const, volatile등)는 모든 선언과 정의에서 같아야 한다.  함수 선언시에 파라미터 이름을 명시하면, 컴파일러에 의해 함수 정의와 함수 선언간의 인터페이스의 동일성을 검사할 수 있는 기회를 줄 수 있다.  예외 : 같은 기본 타입이 호환 가능하면 아래 처럼 사용할 수 있다.  extern void f(signed int);  void f( int); /\* 타입 호환되므로 예외 \*/  extern void g(int \* const);  void g(int \* ); /\* 위배: 한정자 불일치 \*/  See also Rule 8.4 | |
| **예제코드- 함수의 선언과 정의에서 반환 타입과 파라미터의 타입이 일치하는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  typedef uint16\_t width\_t;  typedef uint16\_t height\_t;  typedef uint16\_t area\_t;  extern area\_t area ( width\_t w, height\_t h );  area\_t area ( width\_t w, width\_t h ) /\* Non-compliant: different parameter type name. \*/  {  return ( area\_t ) w \* h;  } | | #include "MISRA\_C\_2012\_08\_03.h"  extern int16\_t func ( int16\_t num, int16\_t den );  int16\_t func ( int16\_t den, int16\_t num ) /\* Non-compliant \*/  {  return num / den;  } |
| **예제코드- 함수의 선언마다 반환 타입이 호환되는지(compatible) 검사 (alias 허용: ON)** | | |
| **BAD** | | **GOOD** |
| #include "MISRA\_C\_2012\_08\_03.h"  extern short int g ( int32\_t \* n ); /\* Non-compliant \*/ | | #include "MISRA\_C\_2012\_08\_03.h"  extern int16\_t g ( int32\_t \* const n ); /\* Non-compliant \*/ |
| **예제코드- 함수의 선언과 정의에 사용된 파라미터의 이름이 같은지 검사** | | |
| **BAD** | | **GOOD** |
| #include "MISRA\_C\_2012\_08\_03.h"  extern int16\_t func ( int16\_t num, int16\_t den );  int16\_t func ( int16\_t den, int16\_t num ) /\* Non-compliant \*/  {  return num / den;  } | | #include "../basic\_types.h"  typedef uint16\_t width\_t;  typedef uint16\_t height\_t;  typedef uint16\_t area\_t;  extern area\_t area ( width\_t w, height\_t h );  area\_t area ( width\_t w, width\_t h ) /\* Non-compliant: 비록 같은 타입이라고 하더라도 다른 타입 이름을 사용했으므로 허용하지 않는다. \*/  {  return ( area\_t ) w \* h;  } |
| **예제코드- 변수의 선언마다 타입이 호환되는지(compatible) 검사 (alias 허용: ON)** | | |
| **BAD** | | **GOOD** |
| bad\_case\_1.c:  uint16\_t u16a = '0';  int32\_t s32a = '0';  bad\_case\_2.c:  extern int16\_t u16a; /\* Not compliant \*/  extern int16\_t s32a; /\* Not compliant \*/ | | bad\_case\_1.c:  uint16\_t u16a = '0';  int32\_t s32a = '0';  bad\_case\_2.c:  extern uint16\_t u16a; /\* Compliant \*/  extern int s32a; /\* Compliant \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_04 - 객체나 함수의 정의 또는 호출 이전에 호환가능한 선언이 존재해야함 | |
| **카테고리** | Required | |
| **설명** | 함수나 객체가 정의되었을 때 그 객체나 함수에 대한 선언이 있다면 컴파일러는 반드시 그 선언과 정의가 호환가능한지 확인한다.  함수의 프로토타입은 [MISRA\_C\_2012\_08\_02] 의 요구에 따라 존재해야 하며, 컴파일러는 함수 파라메터의 타입과 파라메터의 개수까지 일치하는지 확대 적용된다.  추천하는 방법은 external 연결을 갖는 객체나 함수선언들은 헤더파일 내에 위치 시킨 후, 해당 함수나 객체가 필요한 모든 코드파일(.c) 내에 해당 헤더파일(.h)을 포함(include)시켜 사용하도록 한다.  See also Rule 8.2, Rule 8.3, Rule 8.5, Rule 17.3 | |
| **예제코드- 함수의 정의 또는 호출 이전에 선언이 존재하는지 검사(all). 단, 함수 호출은 호출 이전에 정의가 있다면 선언이 없어도 검출하지 않음** | | |
| **BAD** | | **GOOD** |
| #include "header.h" /\* in header.h, func() prototype has not been declared \*/  int main(void) {  func(0); /\* Not compliant. \*/  return 0;  } | | /\* header.h \*/  struct simple {  int var1;  float var2;  };  int func(struct simple arg); /\* function prototype declaration \*/  /\* main.c \*/  #include "header.h"  int main(void) {  struct simple st;  func(st); /\* Compliant \*/  return 0;  } |
| **예제코드- 함수의 선언과 정의에서 반환 타입과 파라미터의 타입이 일치하는지 검사** | | |
| **BAD** | | **GOOD** |
| int func1(int a);  int func2(const int a);  int func3(int a, char b);  int func1(char a) { /\* Not compliant \*/  ...  }  int func2(int a) { /\* Not compliant \*/  ...  }  int func3(char a, double b) { /\* Not compliant \*/  ...  } | | int func1(int a);  int func2(const int a);  int func3(int a, char b);  int func1(int a) { /\* Compliant \*/  ...  }  int func2(const int a) { /\* Compliant \*/  ...  }  int func3(int a, char b) { /\* Compliant \*/  ...  } |
| **예제코드- 변수의 정의 또는 호출 이전에 선언이 존재하는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  extern int32\_t speed = 6000u; /\* Non-compliant - no declaration prior to this definition \*/  int32\_t pressure = 101u; /\* Non-compliant - no declaration prior to this definition \*/ | | extern int16\_t count;  int16\_t count = 0; /\* Compliant \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_05 - 외부연결(external linkage)를 갖는 객체나 함수는 오직 하나의 파일에서만 선언되어야함 | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 정의가 아닌 선언에만 적용된다.  일반적으로 선언은 헤더파일에 있고, 이 헤더 파일을 각 코드파일(.c) 에서 포함하여 사용할 것이다. 이것은 다음의 일관성을 보장한다:  - 선언과 정의  - 다른 번역단위(translation unit)들 안에서의 선언  참고: 프로젝트 내에 많은 헤더파일이 있지만 각각의 external 객체나 함수는 하나의 헤더파일에만 선언되어야 한다.  See also Rule 8.4 | |
| **예제코드- extern 변수를 여러 파일에 중복 선언 금지(한 파일 내의 중복 선언은 허용함)** | | |
| **BAD** | | **GOOD** |
| /\* header.h \*/  extern int ex\_var; /\* Not compliant \*/  /\* related.c \*/  #include "header.h"  extern int ex\_var; /\* Not compliant \*/ | | /\* header.h \*/  extern int ex\_var; /\* Compliant \*/  /\* related.c \*/  #include "header.h"  void func(void) {  int a = ex\_var;  } |
| **예제코드- extern 함수를 여러 파일에 중복 선언 금지(한 파일 내의 중복 선언은 허용함)** | | |
| **BAD** | | **GOOD** |
| /\* header.h \*/  extern int ex\_func(); /\* Not compliant \*/  /\* related.c \*/  #include "header.h"  extern int ex\_func(); /\* Not compliant \*/ | | /\* header.h \*/  extern int ex\_func(); /\* Compliant \*/  /\* related.c \*/  #include "header.h"  void func(void) {  int a = ex\_func();  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_06 - 외부 연계를 가진 식별자는 정확히 하나의 외부 정의를 가져야한다. | |
| **카테고리** | Required | |
| **설명** | 여러 개의 정의를 가졌거나(각각 다른 파일 내에서) 정의가 존재하지 않으면 일반적으로 링킹시에 오류가 발생하며, 오류가 발생하지 않더라도 정의되지 않은 결과가 발생한다. 각각 다른 파일 내에서의 동일한 식별자로 정의된 객체나 함수는 내용이 똑같다고 하더라도 허용되지 않는다(one definition rule 위반). 선언이 다르거나 초기화가 다르더라도 이는 정의되지 않은 결과를 발생시킨다. 주의: 이 결함은 하나의 인스펙터 프로젝트에 서로 다른 링크단위의 프로젝트를 같은 모듈 구성에 포함하는 경우에 발생할 수 있으며, 이 경우 잘못된 탐지가 발생 될 수 있다. | |
| **예제코드- external linkage를 가지는 변수의 중복 정의 금지** | | |
| **BAD** | | **GOOD** |
| header.h :  extern int ex\_var;  related1.c :  #include "header.h"  int ex\_var = 1; /\* Not compliant \*/  related2.c :  #include "header.h"  int ex\_var = 2; /\* Not compliant \*/ | | header.h :  extern int ex\_var1;  extern int ex\_var2;  related1.c :  #include "header.h"  int ex\_var1 = 1; /\* Compliant \*/  related2.c :  #include "header.h"  int ex\_var2 = 2; /\* Compliant \*/ |
| **예제코드- external linkage를 가지는 함수의 중복 정의 금지** | | |
| **BAD** | | **GOOD** |
| header.h :  extern void ex\_func(void);  related1.c :  #include "header.h"  void ex\_func(void) { /\* Not compliant \*/  char c = 'c';  }  related2.c :  #include "header.h"  void ex\_func(void) { /\* Not compliant \*/  char d = 'd';  } | | header.h :  extern void ex\_func1(void);  extern void ex\_func2(void);  related1.c :  #include "header.h"  void ex\_func1(void) { /\* Compliant \*/  char c = 'c';  }  related2.c :  #include "header.h"  void ex\_func2(void) { /\* Compliant \*/  char d = 'd';  } |
| **예제코드- external linkage를 가지는 함수의 정의 존재 검사** | | |
| **BAD** | | **GOOD** |
| header.h :  extern void ex\_func(void); /\* has no definition \*/ /\* Not compliant \*/ | | header.h :  extern void ex\_func(void); /\* Compliant \*/  related.c :  #include "header.h"  void ex\_func(void) {  char c = 'c';  } |
| **예제코드- external linkage를 가지는 변수의 정의 존재 검사** | | |
| **BAD** | | **GOOD** |
| extern int32\_t x8; /\* Not Compliant \*/ | | extern int32\_t x8;  int32\_t x8 = 5; |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_09 - 단일 함수에만 쓰이는 객체는 해당 블록범위에서 정의되어야 한다.[ | |
| **카테고리** | Advisory | |
| **설명** | 함수 내에서 객체가 블록 밖에 정의되느냐 내부에 정의되느냐 하는 것은 코딩 스타일의 문제로, 블록 범위에 객체를 정의하는 것은 실수로 접근되는 것은 줄여주고 그 객체가 다른 곳에서 접근되지 말아야 한다는 의도를 명백히 한다.  하지만, 이 규칙을 준수하는 것이 불가능한 상황이 있을수 있다. 예를들어, 블록 내부에 static 으로 선언된 객체는 외부에서 직접 접근할 수 없다. 이 경우 객체를 포인터를 이용하여 접근하는 방법을 사용할 수 있을텐데. 효과적이지 못하다. 이러한 상황에서는 이 규칙을 적용하지 않는 것을 더 좋을 수 있다. | |
| **예제코드- 하나의 함수에서만 사용되는 변수가 함수 scope에 정의되었는지 검사** | | |
| **BAD** | | **GOOD** |
| int local\_var; /\* Not compliant \*/  void func(void) {  local\_var = 5; /\* variable local\_var is used in func() functions only. \*/  } | | void func(void) {  int local\_var; /\* Compliant \*/  local\_var = 5;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_11 - 배열에 대한 외부연결(external linkage) 선언 시, 그 크기를 명시적으로 해야 함 | |
| **카테고리** | Advisory | |
| **설명** | 비록 불완전한 타입으로 배열을 선언하고 요소들에 접근하는 것이 가능하다고 하더라도, 배열의 크기가 명시적으로 기술되어 있을때 더 안전하다. 각 선언마다 크기를 명시적으로 지정하는 것은 그 배열들이 바르게 사용되고 있는지를 일관성있게 확인할 수 있도록 해준다. 또한 정적분석기가 한 개 이상의 컴파일 단위(translation unit)을 분석할 필요 없이 배열범위 분석을 할수 있게 해준다.  이 규칙은 전역변수 배열 선언에만 적용된다. | |
| **예제코드- external linkage 배열의 사이즈는 명시적으로 표시되었는지 검사** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* variable must write before read\*/  void fcn\_bad\_1()  {  int var1;  int var2;  var1 = 10;  var2 = 20; /\* Not compliant \*/  printf("%d",var1);  }  void fcn\_bad\_2()  {  int var1;  int var2;  int var3;  var1 = 10; /\* Not compliant - this line is meaningless \*/  var1 = 20;  printf("%d",var1);  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  extern int array1[ ]; /\* Not Compliant \*/  extern int array2[] = {1,2,3}; /\* Not Compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Data flow analysis \*/  void fcn\_good\_1()  {  int var1;  int var2;  var1 = 10; /\* Compliant \*/  printf("%d",var1);  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  extern int array1[3] = {1,2,3}; /\* Compliant \*/  extern int array2[5]; /\* Compliant \*/  void testFunc(){  int arr[] = {0}; /\* Compliant \*/  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_08\_12 - 열거자 리스트 내에서 묵시적으로 지정된 열거형 상수의 값은 유일해야 함 | |
| **카테고리** | Required | |
| **설명** | 묵시적으로 지정된 열거형 상수는 바로 직전 열거형 상수 보다 1 큰 값을 가진다. 만약 첫번째 열거형 상수의 값이 묵시적으로 지정되었다면 그 값은 0이다.  명시적으로 지정된 열거형 상수는 지정된 값을 그대로 가진다. 만약, 묵시적인 열겨형 상수와 명시적인 열거형 상수가 섞여있다면 그 값이 중복되는 것이 가능하다. 보통의 경우 이러한 중복은 코드 수정에 따른 의도하지 않은 것일 수 있고, 이로 인해 프로그램이 기대하지 않은 행동을 발생시킬수 있다.  이 규칙은 열거형 상수값의 복제가 명시적으로 이루어 지는 것을 요구하며 이로 인해 그 의도를 명백히 하려 함에 목적이 있다. | |
| **예제코드- 열거형 리스트 내에서 암시적 요소가 유일한 값을 가지는지 검사** | | |
| **BAD** | | **GOOD** |
| /\* Non-compliant - yellow replicates implicit green \*/  enum colour { red = 3, blue, green, yellow = 5 }; | | /\* Compliant \*/  enum colour { red = 3, blue, green = 5, yellow = 5 }; |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_09\_01 - 모든 변수는 읽기 전에 할당되어야 함 | |
| **카테고리** | Mandatory | |
| **설명** | 표준에 따르면, static 변수의 저장 공간은 초기화 해주지 않으면 자동으로 0으로 초기화가 된다. 하지만 일반 변수는 자동으로 초기화 되지 않는다.  참고: 변수에 대한 명시적 초기화가 무시되는 경우가 있을 수 있다. goto문이나 switch문의 사용으로 객체의 선언문을 통과하는 경우에 발생할 것이다.  이 규칙의 목적을 위해서, 배열이나 구조체의 멤버 또한 하나의 분리된 객체로 고려해야 한다.  See also Rule 15.1, Rule 15.3 | |
| **예제코드- 변수 사용 전에 값 할당이 이루어져야 함** | | |
| **BAD** | | **GOOD** |
| int32\_t bad\_0901\_5;  int32\_t bad\_0901\_x;  struct bad\_0901\_str\_type bad\_0901\_str\_auto;  int32\_t bad\_0901\_auto;  ...  bad\_0901\_1 = bad\_0901\_5; /\* Not compliant \*/ | | int32\_t good\_0901\_1 = get\_int32 ( );  int32\_t good\_0901\_2;  ...  good\_0901\_2 = get\_int32 ( ); /\* Initialisation before usage. \*/  ...  use\_int32 ( good\_0901\_1 ); /\* Compliant \*/  use\_int32 ( good\_0901\_2 ); /\* Compliant \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_09\_02 - 배열, 구조체, 유니온 타입의 초기화는 큰괄호('{ }') 로 둘러싸여야 함 | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 객체와 하위객체에 모두 적용된다. { 0 } 형태의 초기화는 대상의 모든 객체를 0으로 설정하는데 사용되며, 이 경우 하위 객체를 초기화 하기 위해 큰괄호를 사용하지 않아도 된다.  참고: 이 규칙은 객체나 하위 객체의 명시적 초기화를 요구하지 않으며, 초기화가 아예 없는 코드에는 적용되지 않는다.  하위 객체의 초기화를 위해 큰괄호를 사용하는 것은 코드의 명확성을 향상시키고 개발자가 다차원 배열이나 구조체의 배열과 같은 복잡한 자료구조에서의 초기화를 고려하도록 해준다.  예외사항  1. 문자열 상수을 사용하여 배열을 초기화 하는 경우  2. 호환되는 구조체나 유니온 표현식을 이용해서 구조체나 유니온을 초기화하는 경우  3. 지정 초기화(designated initializer) 를 이용하여 객체의 일부를 초기화하는 경우  예제  int16\_t y[3][2] = { 1,2,0,0,5,6 }; /\* 위배 \*/  int16\_t y[3][2] = { {1,2},{0,0},{5,6} }; /\* 옳음 \*/  int16\_t y[3][2] = { {1,2,0},{0,5,6} }; /\* 옳음 \*/  struct s1 {  uint16\_t len;  char buf[8];  } s[3] = {  { 5u, {'a','b','c','d','e','\0','\0','\0'} },  { 2u, {0} }, /\* 옳음 \*/  { .len = 0u } /\* 옳음 - 예외 3\*/  }; | |
| **예제코드- 배열, 구조체, 유니온 타입의 초기화가 큰 괄호('{ }')로 둘러싸여있지 않음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int16\_t y[ 3 ][ 2 ] = { 1, 2, 0, 0, 5, 6 }; /\* Non-compliant \*/  int16\_t z3[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 0 ] = 0, 1 }; /\* Non-compliant \*/ | | #include "../basic\_types.h"  int16\_t y1[ 3 ][ 2 ] = { { 1, 2 }, { 0 }, { 5, 6 } }; /\* Compliant \*/  int16\_t y2[ 3 ][ 2 ] = { { 1, 2 }, { 0, 0 }, { 5, 6 } }; /\* Compliant \*/  int16\_t z1[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 1 ] = 1 }; /\* Compliant \*/  int16\_t z2[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 1 ] = 1, [ 1 ][ 0 ] = 0}; /\* Compliant \*/  int16\_t z34[ 2 ][ 2 ] = { [ 0 ][ 1 ] = 0, { 0, 1 } }; /\* Compliant \*/  float32\_t a1[ 3 ][ 2 ] = { 0 }; /\* Compliant \*/  float32\_t a2[ 3 ][ 2 ] = { { 0 }, { 0 }, { 0 } }; /\* Compliant \*/  float32\_t a3[ 3 ][ 2 ] = { { 0.0f, 0.0f },  { 0.0f, 0.0f },  { 0.0f, 0.0f }  }; /\* Compliant \*/  union u1 {  int16\_t i;  float32\_t f;  } u = { 0 }; /\* Compliant \*/  struct s1 {  uint16\_t len;  char buf[ 8 ];  } s[ 3 ] = {  { 5u, { 'a', 'b', 'c', 'd', 'e', '\0', '\0', '\0' } },  { 2u, { 0 } },  { .len = 0u } /\* Compliant - buf initialized implicitly \*/  }; /\* Compliant - s[] fully initialized \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_09\_03 - 배열은 일부분만 초기화 되면 안됨 | |
| **카테고리** | Required | |
| **설명** | 배열이나 객체의 일부 요소가 명시적으로 초기화 되면 안된다. 반드시 전체 요소가 명시적으로 초기화 되어야 한다.  배열의 모든 요소를 명시적 초기화 하는 것은 모든 요소가 고려되었다는 것을 명확히 할 수 있다.  예외사항  1. 배열의 모든 하위 요소를 명시적으로 초기화 하기위해 사용된 { 0 } 형태의 초기화문  2. 지정 초기화(designated initalizer)로만 배열의 일부가 초기화된 경우  3. 문자열 리터럴을 사용하여 초기화 된 배열  예제  int32\_t x[3] = {0, 1}; /\* 위배 \*/  float32\_t z[50] = { [1] = 1.0f, [25] = 2.0f } /\* 옳음 - 예외 2 \*/  float32\_t z[50] = { [1] = 1.0f, 2.0f } /\* 위배 - 예외 2 에 맞지 않음 \*/ | |
| **예제코드- 배열이 부분적으로 초기화되었는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int32\_t y[ 3 ] = { 0, 1 }; /\* Non-compliant - y[ 2 ] is implicitly initialized \*/  float32\_t t[ 4 ] = { [ 1 ] = 1.0f, 2.0f }; /\* Non-compliant - t[ 0 ] and t[ 3 ] are implicitly initialized \*/ | | #include "../basic\_types.h"  /\* Compliant \*/  int32\_t x[ 3 ] = { 0, 1, 2 };  /\* Compliant - designated initializers for sparse matrix \*/  float32\_t z[ 50 ] = { [ 1 ] = 1.0f, [ 25 ] = 2.0f };  float32\_t arr[ 3 ][ 2 ] =  {  { 0.0f, 0.0f },  { 3.14 / 4.0f, -3.14 / 4.0f },  { 0 } /\* initializes all elements of array subobject arr[ 2 ] \*/  };  char h[ 10 ] = "Hello"; /\* Compliant by Exception 3 \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_01 - 부적절한 essential 타입의 피연산자를 사용하지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | 아래 표는 연산자와 피연산자의 essential 타입에 대한 위배 내용을 나타낸다.   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Operator | Operand | Essential type category of arithmentic operand | | | | | | | Boolean | character | enum | signed | unsigned | floating | | [ ] | integer | 3 | 4 |  |  |  | 1 | | + (unary) |  | 3 | 4 | 5 |  |  |  | | - (unary) |  | 3 | 4 | 5 |  | 8 |  | | + - | either | 3 |  | 5 |  |  |  | | \* / | either | 3 | 4 | 5 |  |  |  | | % | either | 3 | 4 | 5 |  |  | 1 | | < > <= >= | either | 3 |  |  |  |  |  | | == != | either |  |  |  |  |  |  | | ! && || | any |  | 2 | 2 | 2 | 2 | 2 | | << >> | left | 3 | 4 | 5,6 | 6 |  | 1 | | << >> | right | 3 | 4 | 7 | 7 |  | 1 | | ~ & | ^ | any | 3 | 4 | 5,6 | 6 |  | 1 | | ?: | 1st |  | 2 | 2 | 2 | 2 | 2 | | ?: | 2nd and 3rd |  |  |  |  |  |  |   각 숫자의 위배 내용  1. essential 타입이 floating인 표현을 []의 내부, %, <<, >>에 사용하는 것은 constraint 위반이다  2. !, &&, ||, 그리고 ?: 연산자의 첫번 째 피연산자는 피연산자가 Boolean value로 처리되기 때문에 essential 타입이 Boolean인 표현만 사용되어야 한다.  3. essential 타입이 Boolean인 표현은 산술 값으로 변환되는 연산자인 [], unary +, unary -, +, -, \*, /, %, <, >, <=, >=, <<, >>, ~, &, |, ^에 사용되지 않아야 한다.  4. 어떤 피연산자가 산술 값으로 변환된다면 essential 타입이 character 인 피연산자를 사용하면 안된다. 문자열 데이터를 저장하는 방법은 구현체마다 다르기 때문이다.  5. enum 객체는 implementation-defined 정수 타입으로 저장되기 때문에 essential 타입이 enum인 피연산자는 산술 연산에 사용되지 않아야 한다. enum객체를 사용한 연산은 예상치 못한 타입의 결과값을 생성할 수 있다.  6. Shift 및 bitwise 연산들은 essential 타입이 unsigned인 객체에 대해서만 수행되어야 한다. Essential 타입이 signed인 객체들을 대상으로 한 해당 동작은 구현체마다 다르다.  7. Shift연산의 우측 피연산자는 essential 타입이 unsigned이어야만 음수 값으로 shifting을 시도함으로 발생하는 undefined behavior를 예방할 수 있다.  8. 단항 minus연산의 피연산자로 essential 타입이 unsigned를 사용하지 말아야 한다. 각 구현체마다 int 크기가 다르기 때문에 해당 연산 결과의 부호 여부를 정확히 알 수 없기 때문이다.  See also Rule 10.2 | |
| **예제코드- 부적절한 essential type을 갖는 피연산자를 사용하였음** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  int main(){  int8\_t s8a;  int16\_t s16a;  uint8\_t u8a;  uint16\_t u16a;  uint32\_t u32b;  float32\_t f32a;  char\_t cha, chb;  bool\_t bla, blb;  enum enuma { a1, a2, a3 } ena, enb; /\* Essentially enum<enuma> \*/  enum { K1 = 1, K2 = 2 }; /\* Essentially signed \*/  f32a & 2U; /\* Rationale 1 - constraint violation \*/  f32a << 2; /\* Rationale 1 - constraint violation \*/  cha && bla; /\* Rationale 2 - char type used as a Boolean value \*/  ena ? a1 : a2; /\* Rationale 2 - enum type used as a Boolean value \*/  s8a && bla; /\* Rationale 2 - signed type used as a Boolean value \*/  u8a ? a1 : a2; /\* Rationale 2 - unsigned type used as a Boolean value \*/  f32a && bla; /\* Rationale 2 - floating type used as a Boolean value \*/  bla \* blb; /\* Rationale 3 - Boolean used as a numeric value \*/  bla > blb;/\* Rationale 3 - Boolean used as a numeric value \*/  cha & chb; /\* Rationale 4 - char type used as a numeric value \*/  cha << 1; /\* Rationale 4 - char type used as a numeric value \*/  ena--; /\* Rationale 5 - enum type used in arithmetic operation \*/  ena \* a1; /\* Rationale 5 - enum type used in arithmetic operation \*/  s8a & 2; /\* Rationale 6 - bitwise operation on signed type \*/  50 << 3U; /\* Rationale 6 - shift operation on signed type \*/  u8a << s8a; /\* Rationale 7 - shift magnitude uses signed type \*/  u8a << -1; /\* Rationale 7 - shift magnitude uses signed type \*/  -u8a; /\* Rationale 8 - unary minus on unsigned type \*/  } | | #include "../basiccase/basic\_types.h"  int main(){  int8\_t s8a;  int16\_t s16a;  uint8\_t u8a;  uint16\_t u16a;  uint32\_t u32b;  float32\_t f32a;  char\_t cha, chb;  bool\_t bla, blb;  enum enuma { a1, a2, a3 } ena, enb; /\* Essentially enum<enuma> \*/  enum { K1 = 1, K2 = 2 }; /\* Essentially signed \*/  f32a & 2U; /\* Rationale 1 - constraint violation \*/  f32a << 2; /\* Rationale 1 - constraint violation \*/  cha && bla; /\* Rationale 2 - char type used as a Boolean value \*/  ena ? a1 : a2; /\* Rationale 2 - enum type used as a Boolean value \*/  s8a && bla; /\* Rationale 2 - signed type used as a Boolean value \*/  u8a ? a1 : a2; /\* Rationale 2 - unsigned type used as a Boolean value \*/  f32a && bla; /\* Rationale 2 - floating type used as a Boolean value \*/  bla \* blb; /\* Rationale 3 - Boolean used as a numeric value \*/  bla > blb;/\* Rationale 3 - Boolean used as a numeric value \*/  cha & chb; /\* Rationale 4 - char type used as a numeric value \*/  cha << 1; /\* Rationale 4 - char type used as a numeric value \*/  ena--; /\* Rationale 5 - enum type used in arithmetic operation \*/  ena \* a1; /\* Rationale 5 - enum type used in arithmetic operation \*/  s8a & 2; /\* Rationale 6 - bitwise operation on signed type \*/  50 << 3U; /\* Rationale 6 - shift operation on signed type \*/  u8a << s8a; /\* Rationale 7 - shift magnitude uses signed type \*/  u8a << -1; /\* Rationale 7 - shift magnitude uses signed type \*/  -u8a; /\* Rationale 8 - unary minus on unsigned type \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_02 - Essential 타입이 character인 표현식은 가감연산자에 부적합하게 사용되지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | 적합한 사용의 예는 다음과 같다.  1. +연산자의 경우 피연산자 중 하나의 essential 타입이 character라면 다른 하나는 essential 타입이 signed거나 unsigned여야 한다. 이 연산 결과의 essential 타입은 character가 된다.  2. -연산자의 경우 첫 번째 피연산자의 essential 타입이 character이라면 두 번째 연산자의 essential 타입이 signed거나 unsigned어야 한다. 만약 두 연산자의 essential 타입이 모두 character이라면 이 연산의 결과는 기본 타입(standard type; 보통 int)이고, 그렇지 않을 경우 character이다.  essential 타입이 character인 수식은 데이터가 산술 값을 표현하지 않으므로 산술적으로 사용되지 않아야 한다. 위에 제시된 사용은 character 데이터의 수정을 허용한다.  예를 들어:  - essential 타입이 character인 두 피 연산자의 차는 '0'에서 '9'사이 값이 할당되어 있는 객체의 값을 변경할 수 있다.  - essential 타입이 character인 피 연산자와 essential 타입이 unsigned인 피연산자의 합은 '0'에서 '9'사이 값이 할당되어 있는 객체의 값을 변경할 수 있다.  - essential 타입이 character인 피연산자로부터 essential 타입이 unsigned인 피연산자를 빼는 연산은 소문자를 대문자로 변경할 수 있다.  See also Rule 10.1 | |
| **예제코드- essential type이 charactor인 수식이 가감연산자에 부적합하게 사용되었음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int main(){  enum enm { ena, enb };  int16\_t s16a = 5;  float32\_t f32a = 1.0f;  char\_t cha;  s16a - 'a';  '0' + f32a;  cha + ':';  cha - ena;  } | | #include "../basic\_types.h"  int main(){  int8\_t s8a;  uint8\_t u8a;  char\_t cha;  '0' + u8a; /\* Convert u8a to digit \*/  s8a + '0'; /\* Convert s8a to digit \*/  cha - '0'; /\* Convert cha to ordinal \*/  '0' - s8a; /\* Convert -s8a to digit \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_03 - 표현식의 값은 더 작은 essential 타입이나 다른 essential 타입 분류에 타입를 갖는 객체에 할당되지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | C언어는 프로그래머에게 상당한 자유를 주고, 다른 산술 타입간의 할당이 자동적으로 처리되도록 한다. 그러나 이러한 묵시적 변환을 사용하는 것은 값, 부호, 정확도의 손실과 같은 의도치 않은 결과를 발생시킬 수 있다. Misra의 essential 타입 모델에서 제안하는 더 강한 타입 제한을 사용하는 것은 이러한 문제 발생 가능성을 줄여준다.  See also Rule 10.4, Rule 10.5, Rule 10.6 | |
| **예제코드- 더 작은 essential type이나 다른 essential type의 객체로 변환되었음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  uint8\_t foo1(uint16\_t x)  {  return x; /\* uint16\_t to uint8\_t \*/  }  void use\_uint16(uint16\_t x)  {  }  void use\_uint32(uint32\_t x){  }  int main(){  enum { K1 = 1, K2 = 128 };  uint8\_t u8b,u8c;  uint32\_t u32a ;  int32\_t s32a;  uint16\_t u16a;  char\_t cha;  int8\_t a1, a2, s8a;  uint8\_t u8a = 1.0f; /\* unsigned and floating \*/  bool\_t bla = 0; /\* boolean and signed \*/  cha = 7; /\* character and signed \*/  u8a = 'a'; /\* unsigned and character \*/  u8b = 1 - 2; /\* unsigned and signed \*/  u8c += 'a'; /\* u8c = u8c + 'a' assigns character to unsigned \*/  use\_uint32(s32a); /\* signed and unsigned \*/  s8a = K2; /\* Constant value does not fit \*/  u16a = u32a; /\* uint32\_t to uint16\_t \*/  use\_uint16(u32a); /\* uint32\_t to uint16\_t \*/  } | | #include "../basic\_types.h"  void use\_uint16(uint16\_t x)  {  }  int main(){  enum enuma { A1, A2, A3 } ena;  enum enumb { B1, B2, B3 } enb;  enum { K1 = 1, K2 = 128 };  uint8\_t u8b,u8c,u8d;  uint32\_t u32a;  uint16\_t u16a,u16b;  char\_t cha;  int8\_t a1, a2, s8a;  uint8\_t\* pu8a, pu8b;  uint8\_t u8a = 0; /\* By exception \*/  bool\_t flag = ( bool\_t ) 0;  bool\_t set = TRUE; /\* true is essentially Boolean \*/  bool\_t get = ( u8b > u8c );  ena = A1;  s8a = K1; /\* Constant value fits \*/  u8a = 2; /\* By exception \*/  u8a = 2 \* 24; /\* By exception \*/  cha += 1; /\* cha = cha + 1 assigns character to character \*/  pu8a = pu8b; /\* Same essential type \*/  u8a = u8b + u8c + u8d; /\* Same essential type \*/  u8a = ( uint8\_t ) s8a; /\* Cast gives same essential type \*/  u32a = u16a; /\* Assignment to a wider essential type \*/  u32a = 2U + 125U; /\* Assignment to a wider essential type \*/  use\_uint16 ( u8a ); /\* Assignment to a wider essential type \*/  use\_uint16 ( u8a + u16b ); /\* Assignment to same essential type \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_04 - 일반 산술 변환이 수행되는 연산자의 두 피연산자들은 필히 같은 essential 타입 분류에 속하는 타입이어야 함 | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 일반 산술 변환에 기술되어 있는 연산자들에 적용된다.  이 연산자들은 shift, 논리 &&, ++, 콤마 연산자를 제외한 이항 연산자들이다. 추가적으로 삼항 연산자의 둘째와 셋째 피연산자도 이 규칙에 의해 처리된다. 증감연산자는 이 규칙의 검사 대상이 아닌다.  C언어는 프로그래머에게 상당한 자유를 주고 또한 다른 산술 타입간의 형 변환이 자동으로 수행되도록 한다. 그러나 이러한 묵시적 변환의 사용은 값, 부호, 정확도의 손실과 같은 의도치 않은 결과를 발생시킬 수 있다. MISRA essential 타입 모델에서 강제하는것과 같은 더 강한 타입 제약을 사용하는 것은 개발자가 정확히 의도한 답을 생산하도록 묵시적 변환을 제한한다.  See also Rule 10.3, Rule 10.7 | |
| **예제코드- 일반산술변환의 두 피연산자가 같은 essential type이 아님** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int main(){  enum enuma { A1, A2, A3 } ena;  enum enumb { B1, B2, B3 } enb;  uint8\_t u8a,u8b;  int8\_t s8a;  char\_t cha;    s8a += u8a; /\* signed and unsigned \*/  u8b + 2; /\* unsigned and signed \*/  enb > A1; /\* enum<enumb> and enum<enuma> \*/  ena == enb; /\* enum<enuma> and enum<enumb> \*/  u8a += cha; /\* unsigned and char \*/  } | | #include "../basic\_types.h"  int main(){  enum enuma { A1, A2, A3 } ena;  uint8\_t u8a;  char\_t cha;  ena > A1;  u8a + u16b;  //The following is compliant by exception 1:  cha += u8a;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_06 - 복합 표현식의 값은 더 큰 essential 타입의 객체에 할당되지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | C언어는 프로그래머에게 상당한 자유를 주고, 다른 산술 타입간의 할당이 자동적으로 처리되도록 한다. 그러나 이러한 묵시적 변환을 사용하는 것은 값, 부호, 정확도의 손실과 같은 의도치 않은 결과를 발생시킬 수 있다. Misra의 essential 타입 모델에서 제안하는 더 강한 타입 제한을 사용하는 것은 이러한 문제 발생의 가능성을 줄여준다. | |
| **예제코드- 복합 수식문이 더 큰 essential type의 객체에 변환되었음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void use\_uint32( uint32\_t x )  {  }  int main(){  enum enuma { A1, A2, A3 } ena;  enum enumb { B1, B2, B3 } enb;  uint32\_t u32a;  uint16\_t u16a, u16b;    u32a = u16a + u16b; /\* Implicit conversion on assignment \*/  use\_uint32 ( u16a + u16b ); /\* Implicit conversion of fn argument \*/  } | | #include "../basic\_types.h"  int main(){  uint32\_t u32a;  uint16\_t u16a,u16b,u16c;    u16c = u16a + u16b; /\* Same essential type \*/  u32a = ( uint32\_t ) u16a + u16b; /\* Cast causes addition in uint32\_t \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_07 - 복합 수식이 기본 산술 변환을 수행하는 연산자의 피연산자로 사용된 경우, 다른 피연산자는 해당 수식의 타입보다 큰 essential 타입을 가지지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | 복합 수식에 대한 묵시적 변환을 금지하는 것은 한 수식 내에 존재하는 산술 연산자들이 모두 같은 essential 타입으로 처리되어야 한다는 것을 의미한다. 이는 개발자의 혼동을 줄여준다.  참고: 이것이 한 수식의 모든 피연산자가 같은 essential 타입이어야 된다는 것을 의미하지 않는다는 점을 주의하라.  수식 u32a + u16a + u16c 은 두 덧셈이 모두 uin32\_t로 수행될 것이기 때문에 문제가 없다. 이 경우 비-복합 수식은 묵시적으로 형변환되었다.  수식 (u16a + u16b) + u32c 의 좌측 덧셈은 uin16\_t에서 수행되지만 우측 덧셈은 좌측 연산의 결과값을 uint32\_t로 변환한 후 uint32\_t에서 수행되기 때문에 문제가 있다.  See also Rule 10.4, Rule 10.6, Section 8.10.3 | |
| **예제코드- 일반산술변환의 한 연산자가 복합수식일때 다른 피연산자가 더 큰 essential type을 가짐** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int main(){  enum enuma { A1, A2, A3 } ena;  enum enumb { B1, B2, B3 } enb;  uint32\_t u32a;  uint16\_t u16a, u16b;    u32a \* ( u16a + u16b ); /\* Implicit conversion of ( u16a + u16b ) \*/  u32a + ( u16a + u16b ); /\* Implicit conversion of ( u16a + u16b ) \*/  } | | #include "../basic\_types.h"  int main(){  uint32\_t u32a, u32b;  uint16\_t u16a, u16b;    u32a \* u16a + u16b; /\* No comp osite conversion \*/  ( u32a \* u16a ) + u16b; /\* No composite conversion \*/  u32a \* ( ( uint32\_t ) u16a + u16b ); /\* Both operands of \* have same essential type \*/  u32a += ( u32b + u16b ); /\* No composite conversion \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_10\_08 - 복합 수식의 값은 다른 essential 타입 분류에 속하는 타입이나 더 큰 essential 타입으로 변환되지 않아야 함 | |
| **카테고리** | Required | |
| **설명** | 더 큰 타입으로의 변환은 각 구현체마다 결과가 상이하므로 허용되지 않는다. 아래의 경우를 고려해보자:  (uint32\_t) ( u16a + u16b );  16비트 머신에서는 덧셈이 16비트 상에서 결과를 modulo-2로 처리하며 수행된 후 32비트로 변환된다. 그러나 32비트 머신에서는 덧셈 연산이 32비트상에서 수행되고 16비트 머신에서는 버려졌을 높은 자리수의 비트값들이 보존된다.  같은 essential 타입 분류의 더 작은 타입으로의 변환은 명시적으로 결과값을 잘라내는 것은 항상 같은 정도(머신에 상관 없이)의 정보 손실을 일으키기 때문에 허용한다.  See also Rule 10.5, Section 8.10.3 | |
| **예제코드- 복합수식의 값이 다른 essential type이나 더 큰 essential type으로 명시적변환이 있음** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int main(){  int32\_t s32a,s32b;  uint16\_t u16a,u16b;  ( uint16\_t ) ( s32a + s32b ); /\* Non-compliant - different essential \*/  ( uint32\_t ) ( u16a + u16b ); /\* Non-compliant - cast to wider \* essential type \*/  } | | #include "../basic\_types.h"  int main(){  int32\_t s32a;  int32\_t s32b;  uint32\_t u32a;  uint32\_t u32b;  ( uint16\_t ) ( u32a + u32b ); /\* Compliant \*/  ( uint16\_t ) s32a; /\* Compliant - s32a is not composite \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_11\_09 - 매크로 NULL은 정수 널 포인터 상수의 유일하게 허용 된 형식이어야한다. | |
| **카테고리** | Required | |
| **설명** | 0 대신 NULL을 씀으로써 null 포인터 상수가 사용되었음을 더 명확히 하여라.  만약 아래와 같은 경우에서 0의 값을 갖는 정수 상수식이 나왔다면 그것은 매크로 NULL을 통해 사용된 것이어야 한다: - 해당 값이 포인터에 할당되는 경우 - 연산자 ==나 !=의 다른 피연산자가 포인터인 경우 - ?:연산자의 세 번째 피연산자가 포인터인 경우 두 번째 피연산자에 쓰일 때 - ?:연산자의 두 번째 피연산자가 포인터인 경우 세 번째 피연산자에 쓰일 때   공백과 괄호들 무시하고 난 후 이러한 정수 상수식들은 확장된 매크로 NULL을 표현할 수 있어야 한다. (void\*) 0의 꼴로 만들어진 null 포인터 상수는 매크로 NULL이 확장되었든 그렇지 않든 허용한다.  See also Rule 11.4 | |
| **예제코드- null 포인터 상수는 매크로 NULL 또는 (void\*)0이어야 한다.** | | |
| **BAD** | | **GOOD** |
| #include <stdlib.h>  #include "../basic\_types.h"  #define MY\_NULL\_1 0  #define MY\_NULL\_2 ( void \* ) 0  void testFunc(void)  {  int32\_t \*p1 = 0; /\* Non-compliant \*/  if ( p1 == MY\_NULL\_1 ) /\* Non-compliant \*/  {  }  } | | #include <stdlib.h>  #include "../basic\_types.h"  #define MY\_NULL\_1 0  #define MY\_NULL\_2 ( void \* ) 0  extern void f ( uint8\_t \*p );  void testFunc(void)  {  int32\_t \*p2 = ( void \* ) 0; /\* Compliant \*/  if ( p2 == MY\_NULL\_2 ) /\* Compliant \*/  {  }    /\* Compliant for any conforming definition of NULL, such as:  \* 0  \* ( void \* ) 0  \* ( ( ( 0 ) ) )  \* ( ( ( 1 - 1 ) ) )  \*/  f ( NULL );  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_12\_01 - 수식 내부에 사용된 연산자의 우선순위가 명확한지 검사 | |
| **카테고리** | Required | |
| **설명** | 아래 표는 이 규칙의 정의를 위한 것이다.   |  |  |  | | --- | --- | --- | | Description | Operator or Operand | Precedence | | Primary | identifier, constant, string literal, (expression) | 16(high) | | Postfix | [] () (function call) , -> ++(post-increment) --(post-decrement) () {} (C99: compound literal) | 14 | | Unary | ++(pre-increment) --(pre-decrement) & \* + - ~ ! sizeof defined (preprocessor) | 14 | | Cast | () | 13 | | Multiplicative | \* / % | 12 | | Additive | + - | 11 | | Bitwise shift | << >> | 10 | | Relational | < > <= >= | 9 | | Equality | == != | 8 | | Bitwise And | & | 7 | | Bitwise XOR | ^ | 6 | | Bitwise OR | | | 5 | | Logical AND | && | 4 | | Logical OR | || | 3 | | Conditional | ?: | 2 |   이 테이블에 사용된 우선순위는 규칙설명을 위한 것이다. 다른 연산자 우선순위 관련 테이블과 다를 수 있다. 표현식의 우선순위는 그 표현식의 파싱 트리 루트의 요소의 우선순위이다. 예를 들어 a << b + c 의 파싱 트리는 다음과 같이 표현된다. 파싱트리 루트의 요소는 "<<"이고 이 표현식은 우선순위 10 이다.  <<  / \  a \  / \  b c  \* sizeof의 피연산자는 괄호로 감싸져야 한다.  \* 우선순위가 2-12의 표현식은 다음의 경우 괄호를 가져야 한다.  - 우선순위 13이하  - 표현식의 우선순위보다 클 때  C언어의 상대적으로 많은 연산자와 그 우선순위 관계는 직관적이지 않다. 이 규칙을 통해 개발자의 실수를 줄일 수 있다. 연산자의 우선순위를 명백히 하기 위해 괄호를 사용하는것은 개발자의 실수 가능성을 제거한다. 또한 코드의 유지보수 및 검토 시 원작자의 의도를 명확히 전달할 수 있다.  괄호의 과도한 사용은 코드를 어지럽히고 가독성을 감소시킬 수 있다. 이 규칙은 괄호가 너무 적은 것과 괄호가 너무많아서 이해하기 힘든 코드 사이의 타협점을 찾는 것을 목표로 한다. | |
| **예제코드- 연산자의 우선순위가 명시되어있는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  typedef int size\_t;  #if defined ( X ) && ( ( X + Y ) > Z )  #endif  #if !defined ( X ) && defined ( Y )  #endif  typedef struct {  int n;  }st;  void main(){}  void testfunc(){  uint16\_t a,b,c,d,x,y,i;  size\_t sz;  uint16\_t\* p;  sz = sizeof x + y; /\* Non-compliant - write sizeof (x)+y or sizeof (x+y) \*/  x = a == b ? a : a - b; /\* Non-compliant \*/  a \* b + c;  x = a << b + c ;  } | | #include "../basiccase/basic\_types.h"  typedef int size\_t;  #if defined ( X ) && ( ( X + Y ) > Z )  #endif  #if !defined ( X ) && defined ( Y )  #endif  typedef struct {  int n;  }st;  void main(){}  void testfunc(){  uint16\_t a,b,c,d,x,y,i;  size\_t sz;  uint16\_t\* p;  st\* st1[3];  /\*The following example shows expressions containing operators at the same precedence level. All of  these are compliant but, depending on the types of a, b and c, any expression with more than one  operator may violate other rules. \*/  a + b;  a + b + c;  ( a + b ) + c;  a + ( b + c );  a + b - c + d;  ( a + b ) - ( c + d );  /\* Compliant \*/  x = ( a == b ) ? a : ( a - b );  /\* Compliant \*/  x = a << ( b + c );  /\* Compliant \*/  if ( a && b && c )  {  }  st1[1]->n; /\* Compliant - no need to write ( a[ i ] )->n \*/  \*p++; /\* Compliant - no need to write \*( p++ ) \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_12\_02 - shift 연산자의 우측 피연산자가 좌측 피연산자의 essential 타입의 범위 내의 정수여야 함 | |
| **카테고리** | Advisory | |
| **설명** | 우측 피연산자가 음수이거나 좌측 피연산자의 사이즈보다 크거나 같으면 정의되지 않은 행동이다.  시프트 연산자의 좌측 피연산자가 16비트 integer이면, 이것이 0 - 15의 범위 내에서만 시프트 되는것을 보장하는 것은 매우 중요하다. essential type의 shift연산자에서의 제한과 설명에 대해서는 [Rule 8.10] 을 참고하라.  이 규칙을 보장하는 방법은 다양하다. 가장 단순한 방법은 우측 피연산자로 상수를 사용하는 것이다.( 값이 정적으로 확인될 수 있다. )  피연산자가 unsigned integer 타입이면 음수가 아니므로 상한선만 확인하면 된다. 그렇지 않으면 상, 하 제한 모두 확인해야 한다. | |
| **예제코드- Shift 연산자의 우측 피연산자는 좌측 피연산자의 essential type의 범위 내의 수여야 함.** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void testfunc(){  uint8\_t u8a;  u8a = u8a << 8; /\* Non-compliant \*/  1u << 10u; /\* Non-compliant \*/  } | | #include "../basiccase/basic\_types.h"  void testfunc(){  uint8\_t u8a;  u8a = u8a << 7; /\* Compliant \*/  u16a = ( uint16\_t ) u8a << 9; /\* Compliant \*/  ( uint16\_t ) 1u << 10u; /\* Compliant \*/  1UL << 10u; /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | 2012\_12\_04 - unsigned integer wrap-around를 발생시키는 상수 수식 금지 | |
| **카테고리** | Advisory | |
| **설명** | unsigned integer 표현식은 overfolw에 엄격하지 않은 대신 wrap-around가 발생한다. 비록 런타임에서 modulo 연산을 사용하기 위해 쓰는 것과 같은 좋은 목적일지라도, 컴파일 타임에서는 의도한 것이 아닐 수 있다. | |
| **예제코드- unsigned integer 타입의 상수에 대한 wrap around가 일어나면 안됨** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  #if 1u + ( 0u - 10u ) /\* Non-compliant as ( 0u - 10u ) wraps \*/  #endif  #define BASE 65024u  #define DELAY 10000u  #define WIDTH 60000u  void g2 ( uint16\_t a ){  }  uint16\_t g3(){  return BASE + 513u;  }  void testFunc(){  uint16\_t x;  uint16\_t y = BASE + 513u;  switch ( x )  {  case BASE + 0u:  f ( );  break;  case BASE + 1u:  g ( );  break;  case BASE + 512u: /\* Non-compliant - wraps to 0 \*/  h ( );  break;  }  g2 ( BASE + 513u) ;  }  void fixed\_pulse ( void )  {  uint16\_t off\_time16 = DELAY + WIDTH; /\* Non-compliant \*/  }  void g ( void )  {  bool\_t b;  uint16\_t y = b ? 0u : ( 0u - 1u ); /\* Non-compliant \*/  } | | #include "../basiccase/basic\_types.h"  const uint16\_t c = 0xffffu;  void f ( void )  {  uint16\_t y = c + 1u; /\* Compliant \*/  }  void g ( void )  {  bool\_t b;  uint16\_t x = ( 0u == 0u ) ? 0u : ( 0u - 1u ); /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_13\_02 - 수식의 값과 영구적인 side effect의 평가결과가 평가 순서에 상관없이 같아야 함 | |
| **카테고리** | Required | |
| **설명** | 두 인접한 시퀀스 사이나 전체 표현식 내에서 :  1. 객체는 한번 이상 변형될 수 없다.  2. 어떠한 객체의 값을 읽는 것이 그 객체로 저장될 값의 계산에 영향을 미치지 않는다면 그 객체는 변형되거나 읽을 수 없다.  3. 한번 이상 volatile-qualified 타입을 변형할 수 없다.  4. 한번 이상 volatile-qualified 타입을 읽을 수 없다.  Note : 객체는 표현식에 의해 직접적으로 접근되는 만큼 포인터나 함수 호출에 의해 간접적으로 접근될 수 있다.  Note : 이는 규칙의 제목보다 더욱 엄격하다. 그래서 다음과 같은 표현식  x = x = 0;  은 값 x가 volatile이 아니라면, 값이 영구적인 side effects가 평가 순서에 독립적이라고 하더라도 허용되지 않는다.  표현식을 평가할 때 표준에서는 컴파일러들에게 상당한 유연성을 부여한다. 대부분의 연산자는 어떤 순서에서든지 평가된 피연산자를 가지고 있다. 평가되지 않는 피연산자를 가지는 예외 상황은 다음과 같다 :  - AND 연산자의 두번째 피연산자는 첫번째 피연산자가 0이 아닌경우에만 평가된다.  - OR 연산자의 두번째 피연산자는 첫번째 피연산자가 0일 경우에만 평가된다.  - :? 연산자의 첫번째 피연산자는 항상 평가된다. 그리고나서 두번째 또는 세번째 피연산자가 평가된다.  - 콤마 연산자의 첫번째 피연산자는 평가되고 나서 두번째 피연산자가 평가된다.  참고 : 괄호의 우선순위는 연산자에 적용된 우선순위를 변형시킬 수 있다. 그러나 이는 순서에 평가된 가장 낮은 레벨의 피연산자의 평가 순서에는 영향을 미치지 않는다.  표현식의 평가와 관련된 예측할수 없는 행동의 많은 경우는 [Rule 13.3]과 [Rule 13.4]를 준수함으로써 피할 수 있다  See also Dir 4.9, Rule 13.0, Rule 13.3, Rule 13.4 | |
| **예제코드- sequence point 내에서 동일한 변수에 대해 side effect가 두 번 이상 발생하거나 side effect가 발생하는 행위와 접근이 함께 존재하면 안됨(함수 호출을 side effect로 판단: FALSE)** | | |
| **BAD** | | **GOOD** |
| b = (a = 1, 2 \* a) + 3 \* a; /\* Not compliant \*/ | | a = 1;  b = (2 \* a) + 3 \* a; /\* Compliant \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_13\_04 - 할당 연산자의 결과를 사용 금지 | |
| **카테고리** | Advisory | |
| **설명** | 다른 산술 연산자와 같이 할당 연산자를 사용하는 것은 다음의 이유로 추천하지 않는다:  - 코드의 가독성을 해친다.  - 구문의 추가적인 side effect는 [Rule 13.2]에서 커버된 정의되지 않은 행동을 일으킬 수 있다.  할당문이 포함된 표현식이 평가되지 않는다고 하더라도 이 규칙을 적용한다.  See also Rule 13.2 | |
| **예제코드- Assign 연산자의 결과 사용 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  #include <stdlib.h>  uint16\_t f ( )  {  uint16\_t temp;  return temp;  }  void g ( )  {  bool\_t bool\_var;  uint16\_t x,y,b,c,d;  uint16\_t a[10];  a[ x ] = a[ x = y ]; /\* Non-compliant - the value of x = y  \* is used \*/  /\*  \* Non-compliant - value of bool\_var = false is used but  \* bool\_var == false was probably intended  \*/  if ( bool\_var = false )  {  }  /\* Non-compliant even though bool\_var = true isn't evaluated \*/  if ( ( 0u == 0u ) || ( bool\_var = true ) )  {  }  /\* Non-compliant - value of x = f() is used \*/  if ( ( x = f ( ) ) != 0 )  {  }  /\* Non-compliant - value of b += c is used \*/  a[ b += c ] = a[ b ];  /\* Non-compliant - values of c = 0 and b = c = 0 are used \*/  b = c = d = 0;  /\* Non-compliant \*/  while ( x = b ){  break;  }  /\* Non-compliant \*/  for ( a = 0 ; a = b ; a++){  break;  }  } | | #include "../basiccase/basic\_types.h"  #include <stdlib.h>  uint16\_t f ( )  {  uint16\_t temp;  return temp;  }  void g ( )  {  bool\_t bool\_var;  uint16\_t x,y,b,c,d;  uint16\_t a[10];  x = y; /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_13\_05 - 논리적 &&나 || 연산자의 우측 항에 영구적인 side effect 가 있으면 안 됨 | |
| **카테고리** | Required | |
| **설명** | &&와 || 연산자의 우측 피연산자의 평가 여부는 좌측 피연산자의 값에 의존적이다. 만약 우측 피연산자가 side effect를 포함한다면 이러한 side effect는 프로그래머의 기대와는 다르게 발생하지 않을 수 있다.  우측 피연산자의 평가가 프로그램 상에서 그 표현식이 발생하는 곳에서 영구적이지 않은 side effect를 일으킨다면 우측 피연산자가 평가되느냐는 중요하지 않다. | |
| **예제코드- 논리연산자 &&과 ||의 오른쪽 피연산자에 side effect 포함 금지 (함수call을 side effect로 판단 : FALSE)** | | |
| **BAD** | | **GOOD** |
| if( var && (x == i++) ) { /\* Not compliant \*/  .....  } | | if( var && (x == i) ) { /\* Compliant \*/  .....  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_13\_06 - sizeof의 피연산자에 side effect를 발생시킬 수 있는 수식 포함 금지 | |
| **카테고리** | Mandatory | |
| **설명** | sizeof의 피연산자 내의 모든 표현식은 보통 평가되지 않는다. 이 규칙은 실제로 평가되는지와는 상관없이 어떤 표현식의 평가도 side effect를 포함하지 않을 것은 요구한다.  sizeof의 피연산자는 표현식이거나 타입일 수도 있다. 피연산자가 표현식을 포함한다면, 가능한 프로그래밍 오류는 표현식이 대부분 상황에서 실제로 평가되지 않을 때 평가 받기를 기대하는 것이다.  C90 표준에서는 피연산자의 표현식이 런타임에서 평가되지 않는다고 서술한다.  C99 표준에서는 피연산자의 표현식은 보통 런타임에서 평가되지 않는다. 하지만 피연산자가 가변 길이의 배열타입을 포함하면, 크기 표현식이 평가되어야 한다. 만약 배열크기 표현식의 평가없이 결과가 결정된다면 명시되지 않은 상황이다.  이 규칙에서 함수의 호출은 side effect로 간주한다.  See also Rule 18.8 | |
| **예제코드- sizeof 연산자에 side effect expression 사용 금지 (함수call을 side effect로 판단 : FALSE)** | | |
| **BAD** | | **GOOD** |
| int i;  int i\_size;  i\_size = sizeof(i = 3000); /\* Not compliant : the intention of assigning a value is not applied. \*/ | | int i;  int i\_size;  i = 3000;  i\_size = sizeof(i); /\* Compliant \*/ |
| **예제코드- sizeof 연산자에 side effect expression 사용 금지 ( 피연산자가 가변 길이 배열인 경우 가변길이의 표현식에 대한 검사 ) (함수call을 side effect로 판단 : FALSE)** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void testFunc ( )  {  volatile uint32\_t v;  int32\_t n;  size\_t s;  s = sizeof ( int32\_t[ n++ ] ); /\* Non-compliant \*/  s = sizeof ( int32\_t[ v ] ); /\* Non-compliant \*/  } | | #include "../basiccase/basic\_types.h"  void testFunc ( )  {  int32\_t n;  size\_t s;  s = sizeof ( int32\_t[ n+ 1 ] ); /\* compliant \*/  s = sizeof ( int32\_t[ n ] ); /\* compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_14\_01 - loop counter 의 essential 타입이 실수형이면 안 됨 | |
| **카테고리** | Required | |
| **설명** | 실수형 loop counter 를 사용하면, 정수형으로 반올림할 때 오류가 발생하여, 실제로 반복하는 횟수가 개발자의 기대와 달라질 수 있다. 반복 과정에서 실수형의 소수점 이하을 반올림 할 수 없을 때 발생한다.  어떤 코드에서 실수형 loop counter 가 올바른 동작을 하더라도, 다른 코드에서는 다른 숫자 값을 나타낼 수 도 있다.  See also Rule 14.2 | |
| **예제코드- for, while, do에 대해서 floating point 수식을 조건식 및 증감식에서 사용금지** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void fcn()  {  for (float x = 0.1f;  x <= 1.0f; /\* Not compliant \*/  x += 0.1f) /\* Not compliant \*/  {  /\* loop may iterate 9 or 10 times \*/  }  for (float x = 100000001.0f;  x <= 100000010.0f; /\* Not compliant \*/  x += 1.0f) /\* Not compliant \*/  {  /\* loop may not terminate \*/  }    }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void fcn()  {  for (float x = 0.1f;  x <= 1.0f; /\* Not compliant \*/  x += 0.1f) /\* Not compliant \*/  {  /\* loop may iterate 9 or 10 times \*/  }  for (float x = 100000001.0f;  x <= 100000010.0f; /\* Not compliant \*/  x += 1.0f) /\* Not compliant \*/  {  /\* loop may not terminate \*/  }  } | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void fcn()  {  for (size\_t count = 1;  count <= 10; /\* Compliant \*/  count += 1) /\* Compliant \*/  {  float x = count/10.0f;  /\* loop iterates exactly 10 times \*/  }  for (size\_t count = 1;  count <= 10; /\* Compliant \*/  count += 1) /\* Compliant \*/  {  double x = 100000000.0 + count;  /\* loop iterates exactly 10 times \*/  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  void fcn()  {  for (size\_t count = 1;  count <= 10; /\* Compliant \*/  count += 1) /\* Compliant \*/  {  float x = count/10.0f;  /\* loop iterates exactly 10 times \*/  }  for (size\_t count = 1;  count <= 10; /\* Compliant \*/  count += 1) /\* Compliant \*/  {  double x = 100000000.0 + count;  /\* loop iterates exactly 10 times \*/  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_14\_02 - for 문이 잘 짜여야 함 | |
| **카테고리** | Required | |
| **설명** | for 문의 구조는 다음과 같아야 한다:  첫 번째 절  - 비었거나  - loop counter 에 값을 할당하거나  - loop counter 를 정의하고 초기화 해야 함(C99)  두 번째 절  - 표현식은 side effect 가 없어야 함  - loop counter 와 loop control flag 를 사용할 수 있음  - for 문의 body 에서 변경하는 다른 객체는 사용하면 안됨  세 번째 절  - 표현식의 side effect 로 오직 loop counter 에만 값을 변경해야 함  - loop body 에서 변하는 다른 객체는 사용하면 안됨  for 문의 body 에서 호출하는 어떤 함수에도 마찬가지이다.  loop control flag 는 for 의 두 번째 절에서 사용하는 Boolean 식에서 사용하는 객체이다.  for 문은 다용도의 반복에 사용할 수 있다. 한정된 형태의 반복문을 사용하여 코드를 작성해야 검토하고 분석하기 쉽다.  예외사항  - for 문의 모든 절이 빌 수도 있다. ( ; ; ) 와 같은 무한 루프는 허용한다.  See also Rule 14.1, Rule 14.3, Rule 14.4 | |
| **예제코드- for문의 초기화, 제어, 증감 expression은 모두 loop 제어와 관련이 있어야 함** | | |
| **BAD** | | **GOOD** |
| uint16\_t bad\_1305\_var1;  uint16\_t bad\_1305\_var2;  for ( bad\_1305\_var2 = get\_uint16 ( );  bad\_1305\_var1 < 22U;  bad\_1305\_var1++ )  {  ...  }  use\_uint16 ( bad\_1305\_var2 );  for ( bad\_1305\_var2 = get\_uint16 ( );  bad\_1305\_var2 <= get\_uint16 ( );  bad\_1305\_var1++ ) /\* Not compliant \*/  {  ...  }  for ( bad\_1305\_var2 = get\_uint16 ( );  bad\_1305\_var1 < 48U;  bad\_1305\_var2++ ) /\* Not compliant \*/  {  ...  } | | uint16\_t good\_1305\_var1;  uint16\_t good\_1305\_var2;  good\_1305\_var2 = get\_uint16 ( );  for ( ;  good\_1305\_var2 < 24U;  good\_1305\_var2++ ) /\* Compliant \*/  {  ...  }  ...  uint16\_t good\_1305\_var0;  for ( good\_1305\_var0;  good\_1305\_var0 < 24U;  good\_1305\_var0++ ) /\* Compliant \*/  {  ...  } |
| **예제코드- loop counter는 하나로 제한** | | |
| **BAD** | | **GOOD** |
| void func(void){  int y = 0;  for( int x = 0; x < y ; x = y++ ){ /\* not compliant \*/  /\* do something \*/  }  for( float i = 0.0; i < 10.0f ; ++i){ /\* not compliant \*/  }  } | | void func(void){  int y = 0;  for( int x = 0; x < y ; x++ ){ /\* compliant \*/  /\* do something \*/  }  for( float i = 0; i < 10 ; ++i){ /\* compliant \*/  }  } |
| **예제코드- condition이나 statement에서 loop counter의 변경 제한** | | |
| **BAD** | | **GOOD** |
| bool foo(int x)  {  ++x;  return (x < 10);  }  int bar(int x)  {  return --x;  }  void func(void){  int num[250];  bool bool\_a = true;  for( int x = 0; foo(x); ){ /\* not compliant \*/  /\* do something \*/  }  for( int i = 0 ; i < 10 && bool\_a; ){ /\* not compliant \*/  i = i \* 2;  }  for( int k = 0; k < 10; ++k ){ /\* not compliant \*/  num[k] = bar(k);  }  } | | bool foo(int x)  {  ++x;  return (x < 10);  }  int bar(int x)  {  return --x;  }  void func(void){  int num[250];  bool bool\_a = true;  for( int i = 0 ; i < 10 && bool\_a; i = i \* 2){ /\* compliant \*/    }  for( int j = 0; j < 10; ++j ){ /\* compliant \*/  int a = 0;  a = num[j];  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_14\_03 - 결과가 항상 같은 제어식 사용 금지 | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 다음에 적용된다:  - if, while, for, do ... while 과 switch 문  - ?: 연산자의 첫 번째 피연산자  제어식의 결과 값이 항상 같으면, 프로그램의 오류가 있을 수 있다. 컴파일러가 항상 결과가 같은 표현식 때문에 도달할 수 없는 코드를 지워버릴 수 있다. 이 때문에 실행될 수도 있는 방어 코드도 삭제 될 수 있다  예외사항  1. 무한 루프를 만들기 위한 표현식은 허용한다.  2. do ... while 문의 essentially Boolean 인 제어식이 0 으로 계산되면 허용한다.  See also Rule 2.1, Rule 14.2 | |
| **예제코드- 비교 조건식의 연산 결과가 항상 같게 나오는 수식 사용 금지(변수와 변수타입의 최대/최소값을 넘는 상수 비교, 변수와 경계 값 상수 비교의 결과가 항상 참이거나 거짓인 비교)** | | |
| **BAD** | | **GOOD** |
| uint16\_t bad\_1307\_var1;  int8\_t bad\_1307\_var2;  /\* Always false \*/  if ( bad\_1307\_var1 < 0U ) /\* Not compliant \*/  ...  /\* Always true \*/  if ( bad\_1307\_var1 <= 0xffffU ) /\* Not compliant \*/  ...    /\* Always true \*/  if ( bad\_1307\_var2 < 130 ) /\* Not compliant \*/ | | uint16\_t good\_1307\_var1;  int8\_t good\_1307\_var2;  if ( good\_1307\_var1 > 0U ) /\* Compliant \*/  ...  if ( good\_1307\_var1 < 0xffffU ) /\* Compliant \*/  ...    if ( good\_1307\_var2 < 120 ) /\* Compliant \*/  ... |
| **예제코드- 비교 조건식의 연산 결과가 항상 같게 나오는 관계연산자(논리곱/논리합) 조합 사용 금지** | | |
| **BAD** | | **GOOD** |
| uint16\_t bad\_1307\_var1;  int8\_t bad\_1307\_var2;  /\* Always false \*/  if ( ( bad\_1307\_var2 < 10 ) &&  ( bad\_1307\_var2 > 20 ) ) /\* Not compliant \*/  ...  /\* Always true \*/  if ( ( bad\_1307\_var2 < 10 ) ||  ( bad\_1307\_var2 > 5 ) ) /\* Not compliant \*/  ... | | uint16\_t good\_1307\_var1;  int8\_t good\_1307\_var2;  if ( ( good\_1307\_var2 > 10 ) &&  ( good\_1307\_var2 < 20 ) ) /\* Compliant \*/  ...  if ( ( good\_1307\_var2 > 10 ) ||  ( good\_1307\_var2 < 5 ) ) /\* Compliant \*/  ... |
| **예제코드- 비교 조건식의 연산 결과가 항상 같게 나오는 상수 사용 금지** | | |
| **BAD** | | **GOOD** |
| void fcn()  {  /\* ... \*/  /\* Always false \*/  if(0)  {  /\* ... \*/  }  /\* Always true \*/  if(1)  {  /\* ... \*/  }  if(2)  {  /\* ... \*/  }  /\* ... \*/  } | | void fcn()  {  int flag;  int var1;  int var2;  /\* ... \*/  /\* Always false \*/  if(flag)  {  /\* ... \*/  }  /\* Always true \*/  if(var1)  {  /\* ... \*/  }  if(var2)  {  /\* ... \*/  }  /\* ... \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_14\_04 - 조건문이나 반복문의 제어식이 essentially Boolean type 인지 검사 | |
| **카테고리** | Required | |
| **설명** | 조건문이나 반복문은 반드시 essentially Boolean type 이어야 한다. for 문의 제어식은 없을 수도 있다. 이 규칙에서도 for 문에 제어식을 사용하지 않는 경우는 검출하지 않는다. 하지만 사용한다면 essentially Boolean type 이어야 한다.  See also Rule 14.2, Rule 20.8 | |
| **예제코드- C 에서 조건문과 반복문 등에 사용된 제어식의 최종 결과타입은 bool 타입인지 검사** | | |
| **BAD** | | **GOOD** |
| void func()  {  int i = 0;  if(i) /\* not compliant \*/  {  }  } | | void func()  {  int i = 0;  if(i != 0) /\* compliant \*/  {  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_01 – goto 문을 사용하면 안된다. | |
| **카테고리** | Advisory | |
| **설명** | goto 를 무분별하게 사용하면 프로그램이 체계가 없고 이해하기가 너무 어렵다. goto를 사용하지 않으면 올바른 제어 흐름을 위해 flag를 도입해야 하고, flag를 사용하면 goto를 사용한 것보다 명료하지 못한 경우가 있다.  따라서 이 규칙을 따르지 않는다면, [Rule 15.2] 와 [Rule 15.3] 의 가이드라인을 따라 goto 를 제한적으로 사용해야 한다.  See also Rule 9.1, Rule 15.2, Rule 15.3, Rule 15.4 | |
| **예제코드- goto 문장 사용 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void func(void)  {  int32\_t i = 0;  L1:  i++;  if(i < 10)  {  goto L1; /\* Not complaint \*/  }  } | | #include "../basic\_types.h"  void func(void)  {  int32\_t i = 0;  while(i < 10)  {  i++;  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_02 – goto 문은 같은 함수에서 나중에 선언 된 레이블로 점프해야한다. | |
| **카테고리** | Required | |
| **설명** | goto 를 무분별하게 사용하면 프로그램이 체계가 없고 이해하기가 너무 어렵다.  goto 문으로 뒤로 돌아가는 것을 금지해야 한다. 같은 코드를 재수행 해야 할 때는, 언어에서 제공하는 반복문을 사용하여 코드가 복잡해지는 것을 막을 수 있다.  See also Rule 15.1, Rule 15.3, Rule 15.4 | |
| **예제코드-** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void myclassFunc(void){  int32\_t i = 0;  L1:  ++i;  if (i == 10)  {  goto L2; /\* Compliant \*/  }  goto L1; /\* Not compliant \*/ /\* :([MISRA\_C\_2012\_15\_02,532,7] \*/  L2:  ++i;  } | | #include "../basic\_types.h"  void myclassFunc(void){  int32\_t i = 0;  if (i == 10)  {  goto L2; /\* Compliant \*/  }  goto L1; /\* Compliant \*/ /\* :([MISRA\_C\_2012\_15\_02,532,7] \*/  L1:  ++i;  L2:  ++i;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_03 – goto 문이 참조하는 레이블은 같은 블록 또는 goto 문을 포함하는 블록 | |
| **카테고리** | Required | |
| **설명** | goto 를 무분별하게 사용하면 프로그램이 체계가 없고 이해하기가 너무 어렵다. 블록 사이나 내부 블록으로 뛰지(jump) 않도록 하면, 코드가 복잡해지는 것을 막을 수 있다.  참조: C99 에서는 변하기 쉬운 타입을 사용할 때 더 많은 제약이 있다. 변하기 쉬운 타입인 식별자가 있는 내부 scope 로 뛰려고 시도하는 것은 제약을 위배하는 것이다.  이 규칙에서 복합문으로 이루어지지 않은 switch 절은 블록으로 취급한다.  See also Rule 9.1, Rule 15.1, Rule 15.2, Rule 15.4, Rule 16.1 | |
| **예제코드-** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void func(int a){  if (a <= 0)  {  goto L2; /\* Non-compliant \*/ /\* :([MISRA\_C\_2012\_15\_03,433] \*/  }  goto L2; /\* Non-compliant \*/ /\* :([MISRA\_C\_2012\_15\_03,433] \*/  {  if (a > 0)  {  L2:  a++;  }  }  switch(a)  {  case 0:  if ( a == 0)  {  goto L1; /\* Non-complant \*/ /\* :([MISRA\_C\_2012\_15\_03,433] \*/  }  break;  case 1:  a = 0;  L1:  ++a;  break;  default:  break;  }  } | | #include "../basic\_types.h"  void func(int a){  if (a <= 0)  {  goto L1; /\* Compliant \*/  }  goto L1; /\* Compliant \*/  L1:  a++;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_04 – break문이나 어떤 반복문을 종료하기위해 사용하는 goto문은 하나이상 있으면 안된다. | |
| **카테고리** | Advisory | |
| **설명** | 반복문의 출구를 제한하면 코드가 복잡하지 않다. 하나의 break 나 goto 문을 허용하면 반복문이 조건식 보다 더 일찍 종료되는 경우가 하나가 된다  See also Rule 15.1, Rule 15.2, Rule 15.3 | |
| **예제코드-** | | |
| **BAD** | | **GOOD** |
| void func(void){  int x = 0;  while(x < 10){ /\* not compliant \*/ /\* :([MISRA\_C\_2012\_15\_04,536,2] \*/  if(x%3 == 0)  {  break;  }  if(x%2 == 0)  {  break;  }  ++x;  }  for(int j = 0; j < 10 ; j++){ /\* not compliant \*/ /\* :([MISRA\_C\_2012\_15\_04,536,2] \*/  if(j%2 == 0)  {  break;  }  else if(j%3 == 0)  {  break;  }  else{  /\* do something \*/  }  }  } | | void func(void){  int x = 0;  while(x < 10){ /\* Compliant \*/  if(x%3 == 0 || x%2 == 0)  {  break;  }  ++x;  }  for(int j = 0; j < 10 ; j++){ /\* Compliant \*/  if(j%2 == 0 || j%3 == 0)  {  break;  }  else{  /\* do something \*/  }  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_05 – 함수는 끝에 단일 종료점을 가져야합니다. | |
| **카테고리** | Advisory | |
| **설명** | IEC 61508 과 ISO 61508 에서는 모듈식 접근을 위해 하나의 출구만 요구하고 있다.  함수 중간의 return 은 의도하지 않게 함수 종료 코드의 누락을 발생시킨다. 함수의 출구가 계속해서 side effect를 발생하는 문장들과 배치되어 있다면, 함수가 실행될 때 어떤 side effect 가 발생할지 알기가 힘들다.  함수는 하나의 return 문만 가져야 한다.  return 문은 함수의 몸체의 마지막 문장이어야 한다.  See also Rule 17.4 | |
| **예제코드-** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  bool\_t func(uint16\_t n, char \*p) /\* Non-compliant \*/ /\* :([MISRA\_C\_2012\_15\_05,84,func,3] \*/  {  if(n > 10)  {  return true;  }  if(p == 0)  {  return true;  }  return false;  } | | #include "../basic\_types.h"  bool\_t func(uint16\_t n, char \*p)  {  return n > 10 || p == 0;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_15\_06 - 반복문이나 선택문이 복합문인지 검사 | |
| **카테고리** | Required | |
| **설명** | 개발자가 반복문이나 선택문의 몸체를 괄호없이 문장만을 나열하는 것이 들여쓰기에 장점이 있다고 믿는 경우가 있다. 실수로 제어 표현식 다음에 ";" 이 포함되면 null 문장으로 연결되므로 위험하다. 복합문을 사용해야 반복문이나 선택문의 몸체를 분명하게 정의할 수 있다.  게다가 개발자가 엉뚱한 if 문과 else 문을 연결해서 읽을 수도 있다.  예외사항  else if 는 else 의 몸체를 복합문으로 해서 if 문을 쓰는 형태가 아니어도 된다. | |
| **예제코드- while or do-while 문의 body는 복합문이어야 함** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void foo(void){  /\* do something \*/  }  void func(void){  int32\_t x = 0;  int32\_t y = 1;  for(int32\_t i = 0; i < y; ++i); /\* Not compliant \*/  {  /\* do something \*/  }  while (y < 10) /\* Not compliant \*/  foo();  ++y;  switch(x); /\* Not compliant \*/  {  /\* do something \*/  }  } | | #include "../basic\_types.h"  void foo(void){  /\* do something \*/  }  void func(void){  int32\_t x = 0;  int32\_t y = 1;  for(int32\_t i = 0; i < y; ++i) /\* Compliant \*/  {  /\* do something \*/  }  while (y < 10) /\* Compliant \*/  {  foo();  ++y;  }  switch(x) /\* Compliant \*/  {  case 0:  /\* do something \*/  break;  default:  break;  }  } |
| **예제코드- for 문의 body는 복합문이어야 함** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void foo(void){  /\* do something \*/  }  void func(void){  int32\_t x = 0;  int32\_t y = 1;  for(int32\_t i = 0; i < y; ++i); /\* Not compliant \*/  {  /\* do something \*/  }  while (y < 10) /\* Not compliant \*/  foo();  ++y;  switch(x); /\* Not compliant \*/  {  /\* do something \*/  }  } | | #include "../basic\_types.h"  void foo(void){  /\* do something \*/  }  void func(void){  int32\_t x = 0;  int32\_t y = 1;  for(int32\_t i = 0; i < y; ++i) /\* Compliant \*/  {  /\* do something \*/  }  while (y < 10) /\* Compliant \*/  {  foo();  ++y;  }  switch(x) /\* Compliant \*/  {  case 0:  /\* do something \*/  break;  default:  break;  }  } |
| **예제코드- if 문의 then, else절이 복합문인지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  void func(void){  int32\_t x = 0;  int32\_t y = 1;  if(x > 0); /\* Not compliant \*/  if(y < 3) ++y; /\* Not compliant \*/  if(x == 1){  /\* do something \*/  }  else if(x == 2); /\* Not compliant \*/  {  /\* do something \*/  }  if(y == 0){  /\* do something \*/  }  else /\* Not compliant \*/  x = 2;  y = 0;  } | | #include "../basic\_types.h"  void func(void){  int32\_t x = 0;  int32\_t y = 1;  if(x > 0)  {  /\* do something ; Compliant \*/  }  if(y < 3)  {  ++y; /\* Compliant \*/  }  if(x == 1){  /\* do something \*/  }  else if(x == 2)  {  /\* do something; Compliant \*/  }  if(y == 0){  /\* do something \*/  }  else  { /\* Compliant \*/  x = 2;  y = 0;  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_16\_01 - 모든 switch 문은 잘 짜여야 함 | |
| **카테고리** | Required | |
| **설명** | C의 switch 문의 구문은 특별히 엄격하지 않고 복잡한 구조를 허용하는데, 구조화되지 못한 행동이다 . 이 규칙과 다른 규칙들은 switch 문이 단순하고 일관된 구조가 되도록 보장한다.  See also Rule 15.3, Rule 16.2, Rule 16.3, Rule 16.4, Rule 16.5, Rule 16.6 | |
| **예제코드- switch문 안에 case, default 절이 아닌 문장이 존재하면 안됨** | | |
| **BAD** | | **GOOD** |
| switch ( sw )  {  uint8\_t decl\_b4\_1st\_case; /\* Not compliant \*/  case 0U:  {  var += 1U;  break;  }  default:  {  var += 2U;  break;  }  } | | /\* Compliant \*/  switch ( sw )  {  case 0U:  {  var += 1U;  break;  }  default:  {  var += 2U;  break;  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_16\_02 - switch label 을 포함하는 가장 가까운 문장은 switch 문이어야 함 | |
| **카테고리** | Required | |
| **설명** | 표준에서 switch label 인 case 나 default 가 switch 문의 어떤 문장의 몸체에나 포함될 수 있도록 허용하기 때문에, 구조화되지 못한 코드가 될 수 있다. 이를 방지하기 위해 switch label 은 반드시 switch 문의 가장 바깥쪽 몸체에 있어야 한다.  See also Rule 16.1 | |
| **예제코드- switch label(case, default)을 포함하는 가장 가까운 문장은 switch 문인지 검사** | | |
| **BAD** | | **GOOD** |
| switch ( nest\_switchable )  {  case 1U:  {  use\_uint16 ( nest\_switchable1 );  case 2U: /\* Not compliant \*/  {  use\_uint16 ( nest\_switchable2 );  break;  }  }  ... | | switch ( nest\_switchable )  {  case 1U: /\* Compliant \*/  {  use\_uint16 ( nest\_switchable1 );  use\_uint16 ( nest\_switchable2 );  break;  }  case 2U: /\* Compliant \*/  {  use\_uint16 ( nest\_switchable2 );  break;  }  ... |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_16\_03 - 모든 switch 절은 break 로 끝나야 함 | |
| **카테고리** | Required | |
| **설명** | 개발자가 실수로 switch 절을 break 로 끝마치지 않으면, 다음 switch 절로 빠져나가거나, 다음 switch 절이 없으면, switch 문의 다음 문장으로 빠져나온다. 다음 switch 절로 빠져나가게 의도했다 하더라도 때로는 오류이다. 제어흐름이 switch 문의 마지막에 있는 break로 끝나지 않는 switch 절에서 나중에 추가되는 switch 절로 빠져나갈 수 있다.  이런 오류를 반드시 발견할 수 있도록, 모든 switch 절의 마지막 문장은 break 이어야 한다.  참고: switch 절은 적어도 하나의 문장은 있어야 정의된다. 따라서 두 연속한 label에 어떤 문장도 없는 경우는 이 규칙에서 허용한다.  See also Rule 16.1 | |
| **예제코드- 문장이 있는 모든 switch 절은 throw나 break문으로 끝나는지 검사** | | |
| **BAD** | | **GOOD** |
| #include "../basic\_types.h"  int16\_t func(int16\_t x)  {  switch(x)  {  case 1:  case 2:  x++;  break;  case 3: /\* not compliant \*/  x--;  default:  }  } | | #include "../basic\_types.h"  int16\_t func(int16\_t x)  {  switch(x)  {  case 1:  case 2:  x++;  break;  case 3: /\* Compliant \*/  x--;  break;  default:  break;  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_16\_07 - switch-expression 은 Boolean 타입이면 안 됨 | |
| **카테고리** | Required | |
| **설명** | 표준에 따라 switch 문의 조건식은 정수 타입이어야 한다. Boolean 의 값의 타입은 정수이기 때문에, switch 문을 Boolean 표현식으로 제어하는 것이 가능하다. 이런 경우는 if-else 구조를 사용하는 것이 적절하다. | |
| **예제코드- switch문의 조건식에 Boolean 타입 사용 금지** | | |
| **BAD** | | **GOOD** |
| switch ( bad\_1504\_a == 0 ) /\* Not compliant \*/  {  case 0:  {  use\_int16 ( bad\_1504\_a );  break;  }  default:  {  use\_int16 ( bad\_1504\_a );  break;  }  } | | switch ( good\_1504\_a) /\* Compliant \*/  {  case 0:  {  use\_int16 ( good\_1504\_a );  break;  }  default:  {  use\_int16 ( good\_1504\_a );  break;  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_17\_01 - stdarg.h에 정의된 요소들은 사용 금지 | |
| **카테고리** | Required | |
| **설명** | va\_list, va\_arg, va\_start, va\_end, va\_copy 는 사용하면 안 된다.  표준에는 "stdarg.h" 와 관련된 많은 정의하지 않은 행동을 나열하고 있다:  - va\_start 를 사용한 함수에서 va\_end 를 사용하지 않는 경우  - 같은 va\_list 의 va\_arg 를 서로 다른 함수에서 사용하는 경우  - 함수의 인자의 타입과 va\_arg 에 명시한 타입이 맞지 않는 경우 | |
| **예제코드- 함수에 가변인자 사용 금지** | | |
| **BAD** | | **GOOD** |
| void var\_func(int var, ...);  void func(void) {  var\_func(1,2,3);  }  void var\_func(int var, ...) { /\* Not compliant \*/  .....  } | | void var\_func(int var1, int var2, int var3);  void func(void) {  var\_func(1, 2, 3);  }  void var\_func(int var1, int var2, int var3) { /\* Compliant \*/  int a, b, c;  ...  } |
| **예제코드- 특정 파일 include 금지 (stdarg.h)** | | |
| **BAD** | | **GOOD** |
| /\* Assuming sample.h is a prohibited header file \*/  #include <sample.h> /\* Not compliant \*/ | |  |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_17\_02 | |
| **카테고리** | Required | |
| **설명** | 재귀는 스택 공간을 넘치게 하여, 심각한 오류가 발생할 수 있다. 재귀를 아주 엄격하게 관리해도, 실행 전에 스택 공간에서 오류가 발생 할 수 있는 상태를 측정할 수는 없다. | |
| **예제코드-** | | |
| **BAD** | | **GOOD** |
| void func1(int a) {  if(a == 1) {  return;  }  func1(a - 1); /\* Not compliant \*/  }  void func2(int a) { /\* Not compliant \*/  if(a > 0) {  --a;  ....  func3(a); /\* recursion \*/  }  }  void func3(int a) {  if(a > 0) {  --a;  ....  func2(a); /\* recursion \*/  }  } | | <none> |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_17\_03 - 함수를 묵시적으로 선언 금지 | |
| **카테고리** | Mandatory | |
| **설명** | 함수 호출은 프로토 타입에 따라야 한다. 함수 호출의 인자 수와 타입이 일치해야 한다.  만약 함수를 묵시적으로 선언하면, C90 컴파일러는 함수의 반환 타입을 int 로 간주한다. 묵시적인 함수는 프로토 타입을 제공하지 않기 때문에, 컴파일러는 함수의 인자나 타입의 정보를 알 수 없다. 인자와 반환 값의 타입이 부적절하게 변환되고, 다른 정의하지 않은 행동도 발생할 수 있다. | |
| **예제코드- 함수의 정의 또는 호출 이전에 선언이 존재하는지 검사(call). 단, 함수 호출은 호출 이전에 정의가 있다면 선언이 없어도 검출하지 않음** | | |
| **BAD** | | **GOOD** |
| #include "header.h" /\* in header.h, func() prototype has not been declared \*/  int main(void) {  func(0); /\* Not compliant. \*/  return 0;  } | | /\* header.h \*/  struct simple {  int var1;  float var2;  };  int func(struct simple arg); /\* function prototype declaration \*/  /\* main.c \*/  #include "header.h"  int main(void) {  struct simple st;  func(st); /\* Compliant \*/  return 0;  } |
| **예제코드- 함수 호출 시 인자의 개수와 파라미터의 개수가 일치하는지 검사** | | |
| **BAD** | | **GOOD** |
| int var\_func1(int var1, int var2);  int var\_func2(int var1, int var2);  int main(void) {  .....  var\_func1(var1, var2, var3); /\* Not compliant \*/  var\_func2(var1); /\* Not compliant \*/  return 0;  } | | int var\_func1(int var1, int var2);  int var\_func2(int var1, int var2);  int main(void) {  .....  var\_func1(var1, var2); /\* Compliant \*/  var\_func2(var1, var2); /\* Compliant \*/  return 0;  } |
| **예제코드- 프로토타입이 정의되지 않은 함수 호출 시 인자가 사용되었는지 검사** | | |
| **BAD** | | **GOOD** |
| void func3(int a);  // void func2(int a){  // }  void func3(int a){  }  int main(int num){  func(num); /\* Not compliant \*/  func2(num); /\* Not compliant \*/  return 0;  } | | void func(int a);  void func2(int a);  void func3(int a);  void func(int a){  }  void func2(int a){  }  void func3(int a){  }  int main(int num){  func(num); /\* Compliant \*/  func2(num); /\* Compliant \*/  func3(num); /\* Compliant \*/  return 0;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_17\_04 - 리턴 타입이 void가 아닌 함수의 모든 경로의 마지막은 수식을 포함한 return 문이어야 함 | |
| **카테고리** | Mandatory | |
| **설명** | return 문의 표현식은 함수가 반환하는 값을 나타낸다. void 가 아닌 함수가 반환 값이 없으면, 정의하지 않은 행동이다.  - 모든 return 문에 표현식이 있다.  - 함수의 제어 흐름이 return 문으로 끝난다.  참고: C99 에서 void 가 아닌 함수의 모든 return 문은 값을 반환해야 한다. | |
| **예제코드- non-void 반환 타입의 함수에서 명시적인 값을 가진 return문이 존재해야 함** | | |
| **BAD** | | **GOOD** |
| int func1(void) { /\* Not compliant \*/  }  int func2(int var) { /\* Not compliant \*/  int res;  res += var;  } | | int func1(void) { /\* Compliant \*/  return 2;  }  int func2(int var) { /\* Compliant \*/  int res;  res += var;  return res;  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_17\_06 - 배열 타입의 매개 변수의 선언은 [] 사이에 static keyword를 포함하지 않아야 함 | |
| **카테고리** | Mandatory | |
| **설명** | C99 표준에서는 프로그래머가 컴파일러에게 배열 타입의 파라미터의 원소 개수를 특정 할 수 있는 장치를 제공한다. 어떤 컴파일러는 이 정보를 이용하여 효과적으로 코드를 생성할 수 있다.  프로그래머가 허용 값을 명시하는 것이 허용되지 않고, 원소의 수가 명시한 것보다 적으면, 정의하지 않은 행동이다.  임베디드 프로그램에서 사용하는 프로세서는 프로그래머가 제공하는 정보를 이해하지 못 할 수 있다. 성능의 향상을 위해 사용한 한정 값 때문에 프로그램이 실패할 수 있다. | |
| **예제코드- 배열 타입의 매개 변수의 선언의 [] 사이에 static keyword가 있는지 검사** | | |
| **BAD** | | **GOOD** |
| typedef unsigned short uint16\_t;  uint16\_t total(uint16\_t n, uint16\_t a[static 20]) /\* not compliant and complie error in C90 \*/  {  } | | typedef unsigned short uint16\_t;  uint16\_t total(uint16\_t n, uint16\_t a[20]) /\* compliant \*/  {  } |
| **규칙이름** | MISRA\_C\_2012\_18\_01 – | |
| **카테고리** | Required | |
| **설명** | 몇몇 컴파일러는 컴파일 타임에 배열의 범위가 초과되었는지 알 수 있지만, 일반적으로 런타임 시점에서 유효하지 않은 배열에 대해서 확인하지 않는다. 유효하지 않은 배열을 사용하면 프로그램에서 오류가 발생할 수 있다.  런타임에 결정되는 배열 값들은 정적 분석이나 수작업 검토에 의해 쉽게 확인할 수 없기 때문에 문제가 발생하기 쉽다. 어떤 값이 유효한지 확인하고 필요하다면 적절한 동작을 추가하기 위해서 가능한한 방어적인 프로그래밍 코드가 제공되어야 한다.  상위 표현식의 하나로 부터 얻어진 결과가 pointer\_expression에 의해 가리키는 배열의 요소가 아니거나, 배열의 끝에서 1이상 넘어간 요소라면, 이것은 정의되지 않은 행동이다. C90의 6.3.6과 C99의 6.5.6을 참고하라.  다차원 배열은 "배열의 배열" 이다. 이 규칙은 다른 하위 배열의 주소를 가리키는 포인터를 만들 수 있는 포인터 연산을 금지한다. 내부 범위를 벗어나는 배열 또한 사용하면 안된다.  See also Dir 4.1, Rule 18.4 | |
| **예제코드- 포인터의 연산 결과는 해당 포인터가 가리키는 배열의 요소여야 함** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void f2 ( void )  {  int32\_t data = 0;  int32\_t b = 0;  int32\_t c[ 10 ] = { 0 };  int32\_t d[ 5 ][ 2 ] = { 0 };/\* 5-element array of 2-element arrays  /\* of int32\_t \*/  int32\_t \*p3 = &c[ 11 ]; /\* Non-compliant - undefined, points to  /\* two beyond \*/  c[ -1 ] = 0; /\* Non-compliant - undefined, array  /\* bounds exceeded \*/  data = c[ 10 ]; /\* Non-compliant - undefined, dereference  /\* of address one beyond \*/  data = d[ 2 ][ 3 ]; /\* Non-compliant - undefined, internal  /\* boundary exceeded \*/  } | | #include "../basiccase/basic\_types.h"  void f2 ( void )  {  int32\_t data = 0;  int32\_t b = 0;  int32\_t c[ 10 ] = { 0 };  int32\_t d[ 5 ][ 2 ] = { 0 };/\* 5-element array of 2-element arrays  /\* of int32\_t \*/  int32\_t \*p1 = &c[ 0 ]; /\* Compliant \*/  int32\_t \*p2 = &c[ 10 ]; /\* Compliant - points to one beyond \*/  p1++; /\* Compliant \*/  data = \*( &data + 0 ); /\* Compliant - C treats data as an  /\* array of size 1 \*/  d[ 3 ][ 1 ] = 0; /\* Compliant \*/  data = \*( \*( d + 3 ) + 1 ); /\* Compliant \*/  p1 = d[ 1 ]; /\* Compliant \*/  data = p1[ 1 ]; /\* Compliant - p1 addresses an array of size 2 \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_18\_02 – | |
| **카테고리** | Required | |
| **설명** | 이 규칙은 표현식의 형태에 적용한다:  pointer\_expression\_1 - pointer\_expression\_2  위의 표현식에서 pointer\_expression\_1과 pointer\_expression\_2가 같은 배열의 요소를 가리키지 않거나, 배열의 끝보다 1 뒤에 위치한 요소를 가리키는 경우 정의되지 않은 행동을 할 수 있다.  See also Dir 4.1, Rule 18.4 | |
| **예제코드- 포인터 간의 뺄셈은 같은 배열의 요소들을 가리키고 있는 포인터에만 적용되어야 함** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void f1 ( int32\_t \*ptr )  {  int32\_t a1[ 10 ];  int32\_t a2[ 10 ];  int32\_t \*p1 = &a1[ 1 ];  int32\_t \*p2 = &a2[ 10 ];  int32\_t diff;  diff = p1 - p2; /\* Non-compliant \*/  diff = ptr - p1; /\* Non-compliant \*/  } | | #include "../basiccase/basic\_types.h"  void f1 ( int32\_t \*ptr )  {  int32\_t a1[ 10 ];  int32\_t a2[ 10 ];  int32\_t \*p1 = &a1[ 1 ];  int32\_t \*p2 = &a2[ 10 ];  int32\_t diff;  diff = p1 - a1; /\* Compliant \*/  diff = p2 - a2; /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_18\_03 – | |
| **카테고리** | Required | |
| **설명** | 포인터 간 비교를 시도하는 것은 두 포인터가 같은 객체를 가리키지 않는다면 정의되지 않은 행동을 발생시킨다.  참조 : 배열의 끝보다 1 뒤에 위치한 요소의 주소를 가리키는 것은 허용하지만, 이 요소를 접근하는 것은 허용하지 않는다.  See also Dir 4.1 | |
| **예제코드- 관계 연산자 >, >=, <, <=는 같은 객체를 가리키고 있는 포인터 외의 포인터에 사용 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void f1 ( void )  {  int32\_t a1[ 10 ];  int32\_t a2[ 10 ];  int32\_t \*p1 = a1;  if ( p1 < a2 ) /\* Non-compliant \*/  {  }  }  struct limits  {  int32\_t lwb;  int32\_t upb;  };  void f2 ( void )  {  struct limits limits\_1 = { 2, 5 };  struct limits limits\_2 = { 10, 5 };  if ( &limits\_1.lwb > &limits\_2.upb ) /\* Non-Compliant \*/  {  }  } | | #include "../basiccase/basic\_types.h"  void f1 ( void )  {  int32\_t a1[ 10 ];  int32\_t a2[ 10 ];  int32\_t \*p1 = a1;  if ( p1 < a1 ) /\* Compliant \*/  {  }  }  struct limits  {  int32\_t lwb;  int32\_t upb;  };  void f2 ( void )  {  struct limits limits\_1 = { 2, 5 };  struct limits limits\_2 = { 10, 5 };  if ( &limits\_1.lwb <= &limits\_1.upb ) /\* Compliant \*/  {  }  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_18\_04 - +, -, +=, -= 연산자는 포인터 타입의 표현식에 적용 금지 | |
| **카테고리** | Advisory | |
| **설명** | ptr[expr]과 같이 배열에 사용하는 인덱싱 형태는 포인터 산술연산의 선호되는 형태이다. 이는 명확하고 이로 인해 포인터 조작보다 에러가 적기 때문이다. 포인터 값을 명시적으로 계산하는 것은 의도하지 않거나 잘못된 메모리에 접근할 수 있어서 잠재적인 위험을 가지고 있다. 이러한 행동은 배열 인덱싱을 통해서도 가능하고, 또한 배열 인덱싱을 사용할 경우 리뷰 작업 또한 쉽게 할 수 있을 것이다.  C 언어에서의 포인터 산술연산은 초심자에게 혼란을 줄 수 있다. 표현식 'ptr + 1'은 ptr의 주소에 1을 더하는 것으로 잘못 해석될 수 있다. 사실 새 메모리 주소는 포인터가 가리키는 대상의 크기에 의존한다. 이러한 오해로 인해 sizeof 연산자가 잘못 적용되었을 때 기대하지 않은 행동을 할 수 있다.  하지만 주의해서 사용했을 때 ++를 사용한 포인터 조작은 몇몇 경우에서 더 자연스러울 수 있다. 예를 들면, 메모리 테스트를 하는 동안 연속적으로 어떤 위치에 접근해야할 때, 즉, 연속적인 위치의 집합으로 이루어진 메모리 공간을 다룰 때, 컴파일 타임에 결정되는 주소 범위를 다룰 때 더욱 편리하다.  예외사항: 규칙 18.2의 두 포인터 간의 차감 연산은 허용한다.  See also Rule 18.1, Rule 18.2 | |
| **예제코드- 포인터 타입의 수식에 +, -, +=, -= 연산자 적용금지 (두 포인터 간의 차감연산 허용)** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void fn1 ( void )  {  uint8\_t a[ 10 ];  uint8\_t \*ptr;  uint8\_t index = 0U;  \*( ptr + 5 ) = 0U; /\* Non-compliant \*/  }  void fn3 ( uint8\_t \*p1, uint8\_t p2[ ] )  {  p1 = p1 + 5; /\* Non-compliant \*/  p2 = p2 + 3; /\* Non-compliant \*/  } | | #include "../basiccase/basic\_types.h"  void fn1 ( void )  {  uint8\_t a[ 10 ];  uint8\_t \*ptr;  uint8\_t index = 0U;  index = index + 1U; /\* Compliant - rule only applies to pointers \*/  a[ index ] = 0U; /\* Compliant \*/  ptr = &a[ 5 ]; /\* Compliant \*/  ptr = a;  ptr++; /\* Compliant - increment operator not + \*/  ptr[ 5 ] = 0U; /\* Compliant \*/  }  void fn2 ( void )  {  uint8\_t array\_2\_2[ 2 ][ 2 ] = { { 1U, 2U }, { 4U, 5U } };  uint8\_t i = 0U;  uint8\_t j = 0U;  uint8\_t sum = 0U;  for ( i = 0U; i < 2U; i++ )  {  uint8\_t \*row = array\_2\_2[ i ];  for ( j = 0U; j < 2U; j++ )  {  sum += row[ j ]; /\* Compliant \*/  }  }  }  void fn3 ( uint8\_t \*p1, uint8\_t p2[ ] )  {  p1++; /\* Compliant \*/  p1[ 5 ] = 0U; /\* Compliant \*/  p2++; /\* Compliant \*/  p2[ 3 ] = 0U; /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_18\_06 – | |
| **카테고리** | Required | |
| **설명** | 객체의 주소는 다음과 같은 상황에서 복사된다:  - 할당  - 메모리 이동 또는 복사 함수  객체의 수명이 끝났을 때 그 객체의 주소는 불확실하다. 불확실한 주소를 사용하면 정의되지 않은 행동을 할 수 있다. | |
| **예제코드- 지역변수의 주소가 자신의 scope을 넘어서는 변수에 할당금지** | | |
| **BAD** | | **GOOD** |
| static int8\_t \* bad\_1706 ( int16\_t \*\* bad\_1706ppi )  {  static int8\_t \* bad\_1706\_ptr1 = NULL;  int8\_t \* bad\_1706\_ptr2;  int8\_t local\_auto;  int16\_t local\_auto2;  bad\_1706\_ptr1 = &local\_auto; /\* Not compliant. bad\_1706\_ptr1 is static \*/  {  int8\_t bad\_1706\_i; /\* Nested block scope declaration \*/  bad\_1706\_ptr2 = & bad\_1706\_i; /\* Not compliant \*/  }  \*bad\_1706ppi = & local\_auto2; /\* Not compliant. parameter dereference \*/  ...  } | | static int8\_t \* good\_1706 ( int16\_t \*\* good\_1706ppi )  {  static int8\_t \* good\_1706\_ptr1 = NULL;  int8\_t \* good\_1706\_ptr2;  int8\_t local\_auto;  int8\_t good\_1706\_i;  good\_1706\_ptr1 = &local\_auto; /\* Compliant \*/  {  good\_1706\_ptr2 = & good\_1706\_i; /\* Compliant \*/  }    } |
| **예제코드- 지역변수의 주소가 return 문에 사용금지** | | |
| **BAD** | | **GOOD** |
| static int8\_t \* bad\_1706\_1 ( int16\_t \*\* bad\_1706ppi )  {  int8\_t local\_auto;    return & local\_auto; /\* Not compliant \*/  } | | void good\_1706\_1 ( int16\_t \*\* good\_1706ppi, int8\_t \*result )  {  int8\_t local\_auto;  ...  \*result = local\_auto; /\* Compliant. use out parameter \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_18\_07 – | |
| **카테고리** | Required | |
| **설명** | 가변길이의 배열 멤버는 Dir 4.12와 Rule 21.3에서 금지된 동적 메모리 할당과 함께 사용될 수 있다.  가변길이 배열은 sizeof 연산자의 행동을 개발자가 기대하지 않는 방향으로 변형시킨다. 가변길이 배열을 포함하는 구조체의 같은 타입의 다른 구조체로의 할당은 예상과는 다르게, 가변 길이 배열 멤버의 시작주소를 제외하고 다른 요소들만 복사할 것이다. 가변 길이 배열을 포함하는 구조체에서, 같은 타입의 다른 구조체로의 할당은 기대한 대로 행동하지 않는다. 왜냐하면 복사가 일어날 때 다른 요소들은 복사되지만, 가변 길이 배열의 주소는 복사되지 않기 때문이다.  See also Dir 4.12, Rule 21.3 | |
| **예제코드-**  **구조체의 멤버로 가변 배열 선언 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  #include <stdlib.h>  struct s  {  uint16\_t len;  uint32\_t data[ ]; /\* Non-compliant - flexible array member \*/  } str; | | No Example |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_19\_01 - 오버랩되는 객체에 할당 또는 복사 금지 | |
| **카테고리** | Mandatory | |
| **설명** | 두 객체가 메모리 영역 내에서 overlap 되어 생성되고 한 객체가 다른 객체에 복사되거나 할당되는 경우, 이것은 정의되지 않은 행동이다.  다음의 경우는 예외이다.  1. 정확히 overlap 되었고, 호환되는 타입의 두 객체의 할당(타입한정자 무시)  2. 표준 라이브러리 함수 memmove를 사용하여 부분적 또는 완전하게 overlap된 객체 간의 복사  See also Rule 19.2 | |
| **예제코드-**  **같은 union 변수 내 다른 타입의 멤버 간의 할당 금지** | | |
| **BAD** | | **GOOD** |
| #include "../basiccase/basic\_types.h"  void fn ( void )  {  union  {  int16\_t i;  int16\_t j;  int32\_t k;  } a = { 0 };  a.j = a.k; /\* Not compliant \*/  } | | #include "../basiccase/basic\_types.h"  void fn ( void )  {  union  {  int16\_t i;  int16\_t j;  int32\_t k;  } a = { 0 };  a.i = a.j; /\* compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_20\_04 - 키워드와 같은 이름으로 매크로 정의 금지 | |
| **카테고리** | Required | |
| **설명** | 키워드의 의미를 바꾸기 위해 매크로를 사용하는 것은 혼란을 줄 수 있다. 키워드와 같은 이름인 매크로가 정의되어 있는 동안 해당 표준헤더가 include 되어 있다면 정의되지 않은 행동을 할 수 있다.  See also Rule 21.1 | |
| **예제코드- 특정 매크로 정의 금지 (auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while, \_Bool, \_Complex, \_Imaginary)** | | |
| **BAD** | | **GOOD** |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #define \_\_DATE\_\_ 3 /\* Not compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include <stdlib.h>  #define int some\_other\_type /\* Non-compliant - redefined int \*/  #define while( E ) for ( ; ( E ) ; ) /\* Non-compliant - redefined while \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #define USER\_DATE 3 /\* Compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #define unless( E ) if ( ! ( E ) ) /\* Compliant \*/  #define seq( S1, S2 ) do { \  S1; S2; } while ( false ) /\* Compliant \*/  #define compound( S ) { S; } /\* Compliant \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_20\_07 - 매크로 인자는 괄호로 감싸야 함 | |
| **카테고리** | Required | |
| **설명** | 매크로 인자에 괄호가 사용되지 않는다면, 매크로로 치환할 때 연산자 우선순위가 원하는 결과로 나오지 않을 수 있다. 매크로 인자가 표현식으로 사용되지 않는다면, 어떠한 연산자도 포함되지 않기 때문에 괄호는 필요하지 않다.  See also Dir 4.9 | |
| **예제코드- 함수 매크로의 인자가 괄호로 감싸졌는지 검사** | | |
| **BAD** | | **GOOD** |
| #define M1( x, y ) ( x \* y )  #define M2( x, y ) ( ( x ) \* ( y ) )  #define M3( x ) a ## x = ( x ) /\* x without ( ) is not used as an expression, also breaks R.20.10 \*/  #define GET\_MEMBER( S, M ) ( S ).M /\* M is a structure member name, not an expression \*/  #define F( X ) G( X )  #define G( Y ) ( ( Y ) + 1 )  void R\_20\_7 ( void )  {  int8\_t r;  r = M1 ( 1 + 2, 3 + 4 ); /\* Non-compliant - result is 11 \*/  use\_int8 ( r );  } | | #define M1( x, y ) ( x \* y )  #define M2( x, y ) ( ( x ) \* ( y ) )  #define M3( x ) a ## x = ( x ) /\* x without ( ) is not used as an expression, also breaks R.20.10 \*/  #define GET\_MEMBER( S, M ) ( S ).M /\* M is a structure member name, not an expression \*/  #define F( X ) G( X )  #define G( Y ) ( ( Y ) + 1 )  void R\_20\_7 ( void )  {  int8\_t r;  r = M1 ( ( 1 + 2 ), ( 3 + 4 ) ); /\* Compliant - result is 21 \*/  use\_int8 ( r );  r = M2 ( 1 + 2, 3 + 4 ); /\* Compliant - result is 21 \*/  use\_int8 ( r );  int16\_t M3 ( 0 ); /\* Compliant \*/  use\_int16 ( a0 );  int8\_t v;  struct { int8\_t minval; } s1 = { 2 };  v = GET\_MEMBER ( s1, minval ); /\* Compliant \*/  use\_int8 ( v );  int16\_t x = F ( 2 ); /\* Compliant \*/  use\_int16 ( x );  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_20\_14 - 모든 #else, #elif, #endif 전처리 지시자는 같은 파일 내에 관련된 #if, #ifdef, #ifndef 가 존재해야 한다. | |
| **카테고리** | Required | |
| **설명** | 조건부 컴파일 지시자가 여러 파일에 걸쳐서 사용됨으로써 코드 블럭이 포함되거나 제외된다면 이는 혼란을 불러일으킨다. #if 가 한 파일 내에서 종료되도록 요구하는 것은 코드의 시각적인 복잡도를 줄이고, 유지보수시 오류의 발생을 줄인다. | |
| **예제코드- 파일의 Preprocessor directive 가 같은 파일 내에서 짝이 맞는지 검사** | | |
| **BAD** | | **GOOD** |
| /\* start of file \*/  #ifdef  ...  #endif  void bad\_1917 ( void )  {  ...  }  ...  #endif /\* Not compliant \*/  /\* end of file \*/ | | /\* start of file \*/  #ifdef  ...  #endif  void good\_1917 ( void )  {  ...  }  /\* end of file \*/ |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_21\_03 – | |
| **카테고리** | Required | |
| **설명** | calloc, malloc, realloc과 free는 사용하면 안되고, 매크로는 이들과 같은 이름으로 재정의하면 안된다.  표준에서 제공하는 동적 메모리 할당과 해제를 사용하면 정의되지 않은 행동을 할 수 있다. 예를 들어:  - 동적으로 할당되지 않은 메모리의 해제  - 해제된 메모리의 사용  - 할당된 메모리에 값이 저장되기 전에 사용  See also Dir 4.12, Rule 18.7, Rule 22.1, Rule 22.2 | |
| **예제코드- 특정 함수 사용 금지(calloc, malloc, realloc, free)** | | |
| **BAD** | | **GOOD** |
| /\* Assuming prohibited\_function() is a prohibited function \*/  var = prohibited\_function(); /\* Not compliant \*/ | | <none> |
| **예제코드- 특정 매크로 확장 금지 (malloc(\([^ \t\r\n\v\f]+|), calloc(\([^ \t\r\n\v\f]+|), realloc(\([^ \t\r\n\v\f]+|), free(\([^ \t\r\n\v\f]+|))** | | |
| **BAD** | | **GOOD** |
| #define calloc malloc /\* Not compliant \*/  #define malloc(x) calloc(1, x) /\* Not compliant \*/ | | <none> |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_22\_02 - 메모리는 시스템 라이브러리 함수를 통해 할당되었을 때만 해제되어야 함 | |
| **카테고리** | Mandatory | |
| **설명** | 메모리를 할당하는 표준 라이브러리 함수는 malloc, calloc, realloc이다.  할당되지 않은 메모리를 해제하거나 같은 곳에 할당된 메모리를 한번 이상 해제하는 것은 정의되지 않은 행동을 유발한다.  See also Dir 4.12, Dir 4.13, Rule 21.3 | |
| **예제코드- 중복 free(delete) 금지** | | |
| **BAD** | | **BAD** |
| int f(size\_t n) {  int error\_condition = 0;  int \*x = (int\*)malloc(n \* sizeof(int));  if (x == NULL)  return -1;  if (error\_condition == 1) {  /\* ... \*/  free(x);  }  /\* ... \*/  free(x); /\* Not compliant \*/  return error\_condition;  } | | int f(size\_t n) {  int error\_condition = 0;  if (n > SIZE\_MAX / sizeof(int)) {  errno = EOVERFLOW;  return -1;  }  int \*x = (int\*)malloc(n \* sizeof(int));  if (x == NULL) {  /\* ... \*/  return -1;  }  if (error\_condition != 0) {  /\* ... \*/  }  free(x); /\* compliant \*/  return error\_condition;  } |
| **예제코드- 포인터가 아닌 객체에 free 금지** | | |
| **BAD** | | **BAD** |
| void func( void ){  int32\_t a;  /\* Non-compliant - a does not point to allocated storage \*/  free ( &a );  } | | void func( void ){  int32\_t \*a = (int \* )malloc( sizeof(int32\_t) );    free ( a );  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_22\_04 - 읽기 전용으로 열린 스트림에 쓰기를 하면 안 됨 | |
| **카테고리** | Mandatory | |
| **설명** | 표준에서는 읽기 전용 스트림에 쓰기를 시도하는 경우, 그에 따른 행동을 명시하지 않고 있으므로, 읽기 전용 스트림에 쓰기를 시도하는 것은 안전하지 않다.  See also Rule 21.6 | |
| **예제코드- 읽기 전용으로 열린 스트림에 쓰기를 하면 안 됨** | | |
| **BAD** | | **GOOD** |
| #include <stdio.h>  void fn ( void )  {  FILE \*fp = fopen ( "tmp", "r" );  (void) fprintf ( fp, "What happens now?" ); /\* Non-compliant \*/  (void) fclose ( fp );  } | | #include <stdio.h>  void fn ( void )  {  FILE \*fp = fopen ( "tmp", "r" );  /\* ... \*/  (void) fclose ( fp );  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_22\_05 - FILE 객체를 가리키는 포인터는 dereference되면 안 됨 | |
| **카테고리** | Mandatory | |
| **설명** | FILE 객체에 대한 포인터는 직, 간접적(memcpy나 memcmp 호출 시 인자로 사용)으로 dereference되면 안 된다.  표준(C90 Section 7.9.3(6), C99 Section 7.19.3(6))에서는 FILE 객체의 주소는 스트림을 제어하는데 매우 중요하며, 객체의 사본으로는 동일한 동작을 보장할 수 없다고 말한다. 따라서 이런 사본이 생성되지 않도록 막기 위해 이 규칙이 필요하다.  FILE 객체에 대한 직접적인 조작은 금지된다. 이렇게 조작된 객체를 스트림 지정자로 사용 시 호환되지 않을 가능성이 있기 때문이다.  See also Rule 21.6 | |
| **예제코드- FILE 객체를 가리키는 포인터는 dereference되면 안 됨** | | |
| **BAD** | | **GOOD** |
| #include <stdio.h>  void func(void)  {  FILE \*pf1;  FILE \*pf2;  FILE f3;  pf2 = pf1; /\* Compliant \*/  f3 = \*pf2; /\* Non-compliant \*/  pf1->pos = 0; /\* Non-compliant \*/  } | | #include <stdio.h>  void func(void)  {  FILE \*pf1;  FILE \*pf2;  pf2 = pf1; /\* Compliant \*/  } |

|  |  |  |
| --- | --- | --- |
| **규칙이름** | MISRA\_C\_2012\_22\_06 - FILE 객체를 가리키는 포인터를 해당 스트림이 닫힌 후에 사용하면 안 됨 | |
| **카테고리** | Mandatory | |
| **설명** | 표준에서는 FILE 객체의 해당 스트림이 닫힌 후에는 FILE 포인터의 값은 불확실하다고 기술되어 있다. 따라서 이 포인터의 값을 사용하는 것은 안전하지 않다.  See also Dir 4.13, Rule 21.6 | |
| **예제코드- FILE 객체를 가리키는 포인터를 해당 스트림이 닫힌 후에 사용하면 안 됨** | | |
| **BAD** | | **GOOD** |
| #include <stdio.h>  void error\_action(void);  void fn ( void )  {  FILE \*fp;  void \*p;  fp = fopen ( "tm p", "w" );  if ( fp == NULL )  {  error\_action ( );  }  fclose ( fp );  fprintf ( fp, "?" ); /\* Non-compliant \*/  p = fp; /\* Non-compliant \*/  } | | #include <stdio.h>  void error\_action(void);  void fn ( void )  {  FILE \*fp;  void \*p;  fp = fopen ( "tm p", "w" );  if ( fp == NULL )  {  error\_action ( );  }  fclose ( fp );  } |

## Code Sonar

* 표준 규칙 요약

표준 규칙은 결함 등급에 따라 4가지로 분류 되며, 분류 기준은 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| **순** | **결함 등급** | **결함 설명** | **치명도** |
| 1 | **Fatal** | 고장을 반드시 유발하는 치명적인 결함 | 상 |
| 2 | **Critical** | 고장을 유발할 수도 있는 결함 | 중 |
| 3 | **General** | 기능에 영향을 미칠 수 있는 결함 | 하 |
| 4 | **Innocuous** | 기능에 직접적인 영향을 미치지 않는 결함 | - |

각 등급 별 결함 종류는 아래와 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| **순** | **결함명** | **결함 설명** | **위험 등급** |
| 1 | Division By Zero | Zero값으로 나누기 연산을 수행 | Fatal |
| 2 | Null Pointer Dereference | 할당되지 않은 Resource 참조 | Fatal |
| 3 | Double Free | 이미 Free된 Heap 영역 메모리를 해제 | Fatal |
| 4 | Free Null Pointer | 할당되지 않은 포인터 값을 해제 | Fatal |
| 5 | Unreasonable Size Argument | 복사, 할당 관련 함수의 인자 중 길이 인자가 마이너스 값이거나 너무 큰 경우 | Fatal |
| 6 | Integer Overflow of Allocation Size | 메모리 할당시 크기가 너무 크다는 문제 | Fatal |
| 7 | Use After Free | 이미 Free된 Heap 영역 메모리를 참조 | Fatal |
| 8 | Free Non-Heap Variable | Heap 영역이 아닌 메모리를 해제 | Fatal |
| 9 | Type Overrun | 할당된 버퍼보다 많이 Resource 사용(구조체 내부 배열에서 발생) | Critical |
| 10 | Buffer Overrun | 할당된 버퍼보다 많이 Resource 사용(일반 배열 변수에서 발생) | Critical |
| 11 | Buffer Underrun | 할당된 버퍼보다 이전 Resource 사용(일반 배열 변수에서 발생) | Critical |
| 12 | Type Underrun | 할당된 버퍼보다 이전 Resource 사용(구조체 내부 배열에서 발생) | Critical |
| 13 | Dangerous Function Cast | 함수포인터에서 잘못된 함수 인자 값이나 Return Type으로 인해 원하지 않는 함수를 호출 | Critical |
| 14 | Varargs Function Cast | 함수 호출 시 잘못된 인자 개수로 인한 잘못된 함수 호출 | Critical |
| 15 | Format String | Printf-style Function에서 Format을 지정하지 않음 | Critical |
| 16 | Leak | 메모리 할당 후 해제 하지 않음 | Critical |
| 17 | File System Race Condition | 파일 및 리소스 사용 전 다른곳에서 사용하고 있는지 확인하지 않고 사용 시도 | Critical |
| 18 | Redundant Condition | 조건 문이 항상 True 혹은 False | Critical |
| 19 | Uninitialized Variable | 초기화 되지 않은 변수를 사용 | Critical |
| 20 | No Space For Null Terminator | String Data가 Null값으로 종료되지 않음 | Critical |
| 21 | Cast Alters Value | Casting으로 인해 값이 변경되는 문제 | Critical |
| 22 | Type Mismatch | Heap 영역 메모리 할당과 해제 시 함수가 서로 맞지 않는 경우 | Critical |
| 23 | Use After Close | 이미 Close 된 파일 및 소켓을 사용 | Critical |
| 24 | Overlapping Memory Regions | 중첩된 메모리에 스트링 및 메모리 복사 시도 | Critical |
| 25 | Misaligned Obvject | 할당된 Pointer 변수의 시작 주소를 가리키지 않음 | Critical |
| 26 | Potential Unbounded Loop | 반복문이 종료되지 않음 | Critical |
| 27 | Return Pointer to Freed | Free 된 Pointer를 Return 값으로 사용 | Critical |
| 28 | Double Close | 이미 Close된 File 혹은 Socket 핸들을 다시 Close 시도 | Critical |
| 29 | Double Lock | 뮤텍스를 unlock 없이 두 번 lock | General |
| 30 | Deadlock | 둘 이상의 프로세스나 스레드가 서로 남이 가진 자원을 요구하면서 양쪽 모두 작업 수행을 할 수 없이 대기 상태로 놓여지는 상태 | General |
| 31 | Try-lock that will never succeed | 뮤텍스가 이미 lock된 상태에서 pthread\_mutex\_trylock()를 사용 | General |
| 32 | Negative Shift Amount | 음수의 Shift 연산자를 사용하면 발생 | General |
| 33 | Function Call Has No Effect | 의미 없는 함수 호출(메모리 복사 함수 호출시의 length 인자값이 0으로 실행 될 때) | General |
| 34 | Shift Amount Exceeds Bit Width | 범위를 넘어선 Shift 연산을 수행 시 발생 | General |

* 규칙 상세

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Division By Zero | 위험도 | Fatal |
| 설명 | | 0으로 나누기 연산을 실행 | | |
| 발생가능 문제 | | System Reset을 유발 | | |
| 예제  코드 | Warning Example | int a = 10;  int b = 0;  int c;  c = a / b; | | |
| Fixed Warning Example | int a = 10;  int b = 0;  int c;  if (b == 0)  {  printf (“Division By Zero Error”);  }  else  {  c = a / b;  } | | |
| 해결 방법 | | 0으로 나누기 연산을 하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Null Pointer Dereference | 위험도 | Fatal |
| 설명 | | 할당되지 않은 Resource 참조 | | |
| 발생가능 문제 | | System Reset을 유발 | | |
| 예제  코드 | Warning Example | char \*p = NULL;  char c;  c = p[0]; | | |
| Fixed Warning Example | char \*p = (char \*)malloc (SIZE);  char c;  if (p != NULL)  {  c = p[0];  }  else  {  printf (“malloc Error”);  } | | |
| 해결 방법 | | NULL Pointer를 참조하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Double Free | 위험도 | Fatal |
| 설명 | | 이미 해제된 동적 메모리 영역을 다시 해제 하려 시도 | | |
| 발생가능 문제 | | 프로그램 메모리 데이터 구조를 파괴시켜 System Reset유발 | | |
| 예제  코드 | Warning Example | char\* ptr = (char\*)malloc (SIZE);  if (err)  {  printf (“malloc Error”);  free (ptr);  }  free (ptr); | | |
| Fixed Warning Example | char\* ptr = (char\*)malloc (SIZE);  if (err)  {  printf (“malloc Error”);  return ERROR;  }  free (ptr); | | |
| 해결 방법 | | 동적 메모리 영역은 한번만 해제해 준다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Free Null Pointer | 위험도 | Fatal |
| 설명 | | 할당되지 않은 포인터 값을 해제 | | |
| 발생가능 문제 | | 프로그램 메모리 데이터 구조를 파괴시켜 System Reset유발 | | |
| 예제  코드 | Warning Example | char\* p = (char\*)malloc (SIZE);  if (p == NULL)  {  printf (“malloc Error”);  free (p);  }  free (p); | | |
| Fixed Warning Example | char \*p;  p = (char \*)malloc (SIZE));  free (p); | | |
| 해결 방법 | | 메모리가 할당된 Pointer 변수만 해제한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Unreasonable Size Argument | 위험도 | Fatal |
| 설명 | | 복사, 할당 관련 함수의 인자 중 길이 인자가 마이너스 값이거나 너무 큰 경우 | | |
| 발생가능 문제 | | Buffer Overrun 및 Underrun 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | char dest[4];  char src[5] = “abcd”;  …  int i = sizeof (src);  strncpy (dest, src, sizeof (src));  … | | |
| Fixed Warning Example | char dest[4];  char src[5] = “abcd”;  …  int i = sizeof (src);  if (i > sizeof (dest))  {  …  }  else  {  strncpy (dest, src, sizeof (src));  …  } | | |
| 해결 방법 | | Size에 대한 check를 반드시 한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Integer Overflow of Allocation Size | 위험도 | Fatal |
| 설명 | | 할당되는 메모리의 사이즈를 명시하는 값이 확인되지 않은 곱셈으로 계산되거나 부정확 함 | | |
| 발생가능 문제 | | Memory Overrun으로 인한 Reset 유발 | | |
| 예제  코드 | Warning Example | int numitems, s;  int \*p1, \*p2, \*p3;  ...  /\* warning issued here \*/  p1 = (int \*)malloc (numitems \* sizeof (int));  s = numitems \* sizeof (int)  /\* warning issued here \*/  p2 = (int \*)malloc (s);  /\* warning issued here \*/  p3 = (int \*) new char[numitems \* sizeof (int)]; | | |
| Fixed Warning Example | int numitems = 10, int s;  int \*p1, \*p2, \*p3;  ...  p1 = (int \*)malloc (numitems\*sizeof (int));  s = numitems \* sizeof (int)  p2 = (int \*)malloc (s);  p3 = (int \*) new char[numitems \* sizeof (int)]; | | |
| 해결 방법 | | 할당되는 메모리의 사이즈를 정확하게 명시하고, 곱셈 계산에 의해 선정된다면 확인을 한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Use After Free | 위험도 | Fatal |
| 설명 | | 이미 Free된 Heap 영역 메모리를 참조 | | |
| 발생가능 문제 | | 잘못된 값 참조 혹은 잘못된 Code 실행으로 인해 System Reset 및 오동작 유발 | | |
| 예제  코드 | Warning Example | …  char \*p = malloc (10);  char \*q = p;  free (p);  q[0] = ‘a’;  … | | |
| Fixed Warning Example | …  char \*p = malloc (10);  char \*q = p;  q[0] = ‘a’;  free (p);  … | | |
| 해결 방법 | | Free 되기 전 참조하거나, Free 후에는 참조하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Free Non-Heap Variable | 위험도 | Fatal |
| 설명 | | Heap 영역이 아닌 메모리를 해제ex) Stack에 할당된 변수 | | |
| 발생가능 문제 | | 프로그램 메모리 데이터 구조를 파괴시켜 System Reset유발 | | |
| 예제  코드 | Warning Example | char buf[10];  char \*q = &buf[0];  /\* warning issued here \*/  Free (q); | | |
| Fixed Warning Example | char \*q;  q = (char \*)malloc (SIZE);  free (q); | | |
| 해결 방법 | | Stack에 할당된 것은 free()로 해제를 하지 않고, 동적으로 할당된 메모리만 free()로 해제한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Type Overrun | 위험도 | Critical |
| 설명 | | 할당된 버퍼보다 많이 Resource 사용(구조체 변수, 다중 배열에서 발생) | | |
| 발생가능 문제 | | 다른 변수 영역의 값을 변경시켜 오동작 및 Reset 유발 | | |
| 예제  코드 | Warning Example | struct  {  char name[20];  int ssn;  int grades[50];  } record;  scanf ("%s", record.name); | | |
| Fixed Warning Example | struct  {  char name[20];  int ssn;  int grades[50];  } record;  scanf ("%19s", record.name); | | |
| 해결 방법 | | 해당 구조체 변수의 크기를 확인하거나, 제한을 걸어 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Buffer Overrun | 위험도 | Critical |
| 설명 | | 할당된 버퍼보다 많이 Resource 사용(일반 배열 변수에서 발생) | | |
| 발생가능 문제 | | 다른 변수 영역의 값을 변경시켜 오동작 및 Reset 유발 | | |
| 예제  코드 | Warning Example | char buf[10];  ….  buf[10] = ‘a’; | | |
| Fixed Warning Example | char buf[10]  …  Buf[9] = ‘a’; | | |
| 해결 방법 | | 할당된 메모리의 마지막 번지를 넘지 않도록 Read/Write한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Buffer Underrun | 위험도 | Critical |
| 설명 | | 할당된 버퍼보다 이전 Resource 사용(일반 배열 변수에서 발생) | | |
| 발생가능 문제 | | 다른 변수 영역의 값을 변경시켜 오동작 및 Reset 유발 | | |
| 예제  코드 | Warning Example | char buf[10];  int i;  …  for (i = 9; i >= 0; i --)  {  buf[i] += buf[i - 1];  } | | |
| Fixed Warning Example | char buf[10];  int i;  …  for (i = 9; i >= 0; i --)  {  buf[i] += buf[i];  } | | |
| 해결 방법 | | 할당된 메모리의 처음 번지 이전 주소를 Read/Write 하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Type Underrun | 위험도 | Critical |
| 설명 | | 할당된 버퍼보다 이전 Resource 사용 (구조체 변수, 다중 배열에서 발생) | | |
| 발생가능 문제 | | 다른 변수 영역의 값을 변경시켜 오동작 및 Reset 유발 | | |
| 예제  코드 | Warning Example | struct  {  char name[20];  int ssn;  int grades[50];  } record;  int i,  num\_grades;  ...  for (i = num\_grades; i >= 0; i --)  {  record.grades[i - 1] += record.grades[i];  } | | |
| Fixed Warning Example | struct  {  char name[20];  int ssn;  int grades[50];  } record;  int i,  num\_grades;  ...  for (i = num\_grades; i >= 0; i --)  {  record.grades[i] += record.grades[i];  } | | |
| 해결 방법 | | 할당된 메모리의 처음 번지 이전 주소를 Read/Write 하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Dangerous Function Cast | 위험도 | Critical |
| 설명 | | 함수 포인터를 사용 시 타입이 다른 파라미터나 리턴 타입을 갖는 함수 포인터로 casting할 때 발생 | | |
| 발생가능 문제 | | 오동작 및 Reset 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | void func1 (int x);  void func2 (char z);  void main (void)  {  int a = 0;  char b = ‘a’;  void (\*fptr1)(int) = func2;  void (\*fptr2)(char) = func1;  fptr1 (a);  fptr2 (b);  } | | |
| Fixed Warning Example | void func1 (int x);  void func2 (char z);  void main (void)  {  int a = 0;  char b = ‘a’;  void (\*fptr1)(int) = func1;  void (\*fptr2)(char) = func2;  fptr1 (a);  fptr2 (b);  } | | |
| 해결 방법 | | 함수포인터의 타입을 확인 후 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Varargs Function Cast | 위험도 | Critical |
| 설명 | | 함수 호출 시 잘못된 인자 개수로 인한 잘못된 함수 호출 | | |
| 발생가능 문제 | | 오동작 발생 및 System Reset 가능성 존재 | | |
| 예제  코드 | Warning Example | typedef void (\*va\_t) (int, ...);  void E (int i, int j) { }  int main (void)  {  va\_t va = (va\_t)E;  va (1, 2, 3);  return 42;  } | | |
| Fixed Warning Example | typedef void (\*va\_t) (int, int);  void E (int i, int j) { }  int main (void)  {  va\_t va = (va\_t)E;  va (1, 2);  return 42;  } | | |
| 해결 방법 | | 함수 인자 개수를 명확히 정의, 호출한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Format String | 위험도 | Critical |
| 설명 | | 함수의 특정 인자 위치에 포맷 스트링이 있는 함수는 포맷 스트링이 아니거나 신뢰할 수 없는 소스의 문자열도 통과 | | |
| 발생가능 문제 | | Buffer Overrun이나 값의 표현이 잘못 될 수 있음 | | |
| 예제  코드 | Warning Example | char greeting[18] = "hello";  char buf[12];  printf ("what is your name?\n");  scanf ("%s", buf);  /\* Format String warning issued here \*/  printf (strcat (greeting, buf)); | | |
| Fixed Warning Example | char greeting[18] = "hello";  char buf[12];  printf ("what is your name?\n");  scanf ("%s", buf);  printf ("%s", strcat (greeting, buf)); | | |
| 해결 방법 | | 포맷문자(%n)의 개수와 함수 인자의 개수가 일치해야 한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Leak | 위험도 | Critical |
| 설명 | | 동적 할당된 메모리가 해제되지 않음 | | |
| 발생가능 문제 | | 자원이 고갈되어 System Reset 및 Slow Down유발 | | |
| 예제  코드 | Warning Example | char \*p= (char \*)malloc (SIZE);  return 0; | | |
| Fixed Warning Example | char \*p= (char \*)malloc (SIZE);  …  free (p);  return 0; | | |
| 해결 방법 | | 동적 할당으로 사용한 메모리는 반드시 해제한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | File System Race Condition | 위험도 | Critical |
| 설명 | | 파일 및 리소스 사용 전 다른곳에서 사용하고 있는지 확인하지 않고 사용 시도 | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | fname = "myfile.txt";  unlink (fname);  fdesc = open (fname, (\_CREAT | O\_RDWR));  write (fdesc, secret, secret\_size); | | |
| Fixed Warning Example | fname = "myfile.txt";  fdesc = open (fname, (\_CREAT | O\_RDWR));  write (fdesc, secret, secret\_size); | | |
| 해결 방법 | | 동시에 같은 파일에 접근하지 않도록 주의한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Redundant Condition | 위험도 | Critical |
| 설명 | | 어떤 조건들은 항상 실행되거나 절대 만족될 수 없음 | | |
| 발생가능 문제 | | 예외처리의 실패 혹은 의도하지 않은 동작 유발 가능성 존재 | | |
| 예제  코드 | Warning Example | int i = 0;  if (i == 0)  {  /\* always taken \*/  ...  } | | |
| Fixed Warning Example | int i = 0;  int i = rand ();  if (i == VALUE)  {  ...  } | | |
| 해결 방법 | | 분기문을 항상 만족하거나 절대 만족 할 수 없는 조건을 만들지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Uninitialized Variable | 위험도 | Critical |
| 설명 | | 초기화 되지 않은 변수를 사용 | | |
| 발생가능 문제 | | 예상하지 못하는 값이 입력되어 오동작 및 System Reset 유발 | | |
| 예제  코드 | Warning Example | int func (int condition)  {  int i;  int j = 0;  if (condition)  {  i = 1;  }  j = i + 1;  return j;  } | | |
| Fixed Warning Example | int func (int condition)  {  int i = 0;  int j = 0;  if (condition)  {  i = 1;  }  j = i + 1;  return j;  } | | |
| 해결 방법 | | 변수를 반드시 초기화 하여 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | No Space For Null Terminator | 위험도 | Critical |
| 설명 | | 문자열 복사 함수 사용시 Null 종료문자를 위한 공간이 없음 | | |
| 발생가능 문제 | | String관련 함수의 오동작 발생으로 System 오동작 및 Reset 유발 | | |
| 예제  코드 | Warning Example | #define MAXLEN 1024  ...  char \*pathbuf[MAXLEN];  ...  /\*does not null terminate\*/  read (cfgfile, inputbuf, MAXLEN);  /\*requires null terminated input\*/  strcpy (pathbuf, input\_buf);  ... | | |
| Fixed Warning Example | #define MAXLEN 1024  ...  char \*pathbuf[MAXLEN];  ...  read (cfgfile, inputbuf, MAXLEN-1);  strcpy (pathbuf, input\_buf); | | |
| 해결 방법 | | 문자열을 복사하는 함수를 사용할 때 Null termination을 위한 공간을 확보한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Cast Alters Value | 위험도 | Critical |
| 설명 | | Casting은 예기치 않은 변환을 야기 | | |
| 발생가능 문제 | | Buffer Overflow 및 예상치 않은 동작 유발 | | |
| 예제  코드 | Warning Example | …  int x;  unsigned int y;  …  x = -10;  y = (unsigned int)(x + 5);  … | | |
| Fixed Warning Example | …  int x;  int y;  …  x = -10;  y = x + 5;  … | | |
| 해결 방법 | | 크기가 큰 자료형의 값을 크기가 작은 자료형의 값으로 Casting하거나 signed/unsigned의 부호가 다른 자료형으로 Casting 하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Type Mismatch | 위험도 | Critical |
| 설명 | | Heap 영역 메모리 할당과 해제 시 함수가 서로 맞지 않는 경우 | | |
| 발생가능 문제 | | 잘못된 Heap 영역 메모리 해제로 인한 Reset 유발 | | |
| 예제  코드 | Warning Example | …  (char \*) p = malloc (SIZE);  /\* Type Mismatch warning issued here \*/  delete[] p; | | |
| Fixed Warning Example | …  (char \*) p = malloc (SIZE);  free (p); | | |
| 해결 방법 | | 같이 Pair를 이루는 함수를 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Use After Close | 위험도 | Critical |
| 설명 | | 이미 Close된 socket이나 file을 사용 하려고 할 때 발생 | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | …  FILE \*f = fopen (“a.txt”, “r”)  char buf[MAX];  int readLen;  readLen = fread(buf, 1, MAX, f);  fclose (f);  …  readLen = fread (buf, 1, MAX, f);  … | | |
| Fixed Warning Example | …  FILE \*f = fopen (“a.txt”, “r”)  char buf[MAX];  int readLen;  readLen = fread (buf, 1, MAX, f);  fclose (f);  … | | |
| 해결 방법 | | Close 되기 전 사용하거나, Close 되고 난 후는 사용하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Overlapping Memory Regions | 위험도 | Critical |
| 설명 | | 메모리에 쓰고, 읽는 작업을 수행하는 함수들이 각각 메모리 영역을 침범하여 사용할 때 발생한다. | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | char \* \_mbscpy (char \*s1, const char \*s2)  void \* \_mbsnbcpy (void \*s1, const void \*s2, size\_t n)  char \* mbscpy (char \*s1, const char \*s2)  void \* memccpy (void \*to, const void \*from, int ch, size\_t nbytes)  void \* memcpy (void \*dest, const void \*src, int num)  char \* strcpy (char \*s1, const char \*s2)  char \* strncpy (char \*s1, const char \*s2, size\_t max)  위와 같은 함수를 사용할 때 주의 하도록 한다. | | |
| 해결 방법 | | 메모리 영역 침범이 발생하지 않도록 방어적인 코딩을 한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Misaligned Object | 위험도 | Critical |
| 설명 | | 동적 할당된 Resource를 사용하거나 풀어주는 함수가 할당된 Object의 시작 주소가 아닌 다른 주소로 호출됨 | | |
| 발생가능 문제 | | Heap 영역 구조 파괴로 인한 Reset 및 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | (char \*) p = (char \*)malloc (SIZE);  /\* warning issued here \*/  free (p + 1); | | |
| Fixed Warning Example | (char \*) p = (char \*)malloc (SIZE);  /\* warning issued here \*/  free (p); | | |
| 해결 방법 | | 동적 할당된 메모리의 시작 주소를 해제한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Potential Unbounded Loop | 위험도 | Critical |
| 설명 | | 반복문이 종료되지 않음 | | |
| 발생가능 문제 | | 무한 반복으로 인해 System Reset 발생 | | |
| 예제  코드 | Warning Example | int main (void)  {  int i = 0;  while (i >= 0)  {  i++;  }  return 0;  } | | |
| Fixed Warning Example | int main (void)  {  int i = 0;  while (1)  {  if (10 == i ++)  {  break;  }  }  return 0;  } | | |
| 해결 방법 | | 반드시 종료 조건을 포함한 반복문을 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Return Pointer to Freed | 위험도 | Critical |
| 설명 | | 이미 해제된 메모리의 Pointer를 리턴 | | |
| 발생가능 문제 | | Null Pointer 참조 발생 | | |
| 예제  코드 | Warning Example | int \* returnptr ()  {  int \* px = (int\*)malloc (sizeof (int));  /\* Return Pointer To Freed warning issued \*/  free (px);  return (px);  }  void main (void)  {  int \* ptr;  ptr = returnptr ();  free (ptr);  } | | |
| Fixed Warning Example | int \* returnptr()  {  int \* px = malloc (sizeof (int));  return (px);  }  void main (void)  {  int \* ptr;  ptr = returnptr ();  free (ptr);  } | | |
| 해결 방법 | | 해제된 메모리의 Pointer를 리턴하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Double Close | 위험도 | Critical |
| 설명 | | 이미 Close된 File 혹은 Socket 핸들을 다시 Close 시도 | | |
| 발생가능 문제 | | 두번 Close 시도된 헨들 접근시 오동작 가능성 존재 | | |
| 예제  코드 | Warning Example | aDesc = open (“a.txt", (O\_CREAT | O\_RDWR));  close (aDesc);  bDesc = open (“b.txt", O\_CREAT);  close (aDesc);  write (bDesc, databuf, datasize); | | |
| Fixed Warning Example | aDesc = open (“a.txt", (O\_CREAT | O\_RDWR));  close (aDesc);  bDesc = open (“b.txt", O\_CREAT);  close (bDesc);  write (bDesc, databuf, datasize); | | |
| 해결 방법 | | 파일이나 소켓을 두 번 close()하지 않는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Double Lock | 위험도 | General |
| 설명 | | 뮤텍스를 unlock 없이 두 번 lock | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | pthread\_mutex\_lock (&LockA);  pthread\_mutex\_lock (&LockB);  /\* Locked Twice warning issued here \*/  pthread\_mutex\_lock (&LockA); | | |
| Fixed Warning Example | pthread\_mutex\_lock (&LockA);  pthread\_mutex\_lock (&LockB); | | |
| 해결 방법 | | Deadlock이나 예기치 못한 상황이 발생할 수 있으므로 mutex 상태를 확인하여 int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);와 int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);을 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Deadlock | 위험도 | General |
| 설명 | | 둘 이상의 프로세스나 스레드가 서로 남이 가진 자원을 요구하면서 양쪽 모두 작업 수행을 할 수 없이 대기 상태로 놓이는 상태 | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | int main (void)  {  pthread\_create (&(thread\_id[0]), NULL, update\_firstname, NULL);  pthread\_create (&(thread\_id[1]), NULL, update\_surname, NULL);  }  Thread 1:  void update\_firstname (NameStr newfirst)  {  pthread\_mutex\_lock (&Green);  pthread\_mutex\_lock (&Blue);  }  Thread 2:  void update\_surname (NameStr newsurname)  {  pthread\_mutex\_lock (&Blue);  while (1)  {  ...  }  } | | |
| Fixed Warning Example | int main (void)  {  pthread\_create (&(thread\_id[0]), NULL, update\_firstname, NULL);  pthread\_create (&(thread\_id[1]), NULL, update\_surname, NULL);  }  Thread 1:  void update\_firstname (NameStr newfirst)  {  pthread\_mutex\_lock (&Green);  usleep (1);  pthread\_mutex\_unlock (&Green);  usleep (1);  }  Thread 2:  void update\_surname (NameStr newsurname)  {  usleep (1);  pthread\_mutex\_lock (&Green);  } | | |
| 해결 방법 | | 동기화 기법을 적절히 사용하여 Deadlock을 방지한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Try-lock that will never succeed | 위험도 | General |
| 설명 | | 뮤텍스가 이미 lock된 상태에서 pthread\_mutex\_trylock()를 사용 | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | pthread\_mutex\_lock (&LockA);  pthread\_mutex\_lock (&LockB);  /\* Try-lock warning issued here \*/  pthread\_mutex\_trylock (&LockA); | | |
| Fixed Warning Example | pthread\_mutex\_lock (&LockA);  pthread\_mutex\_lock (&LockB);  pthread\_mutex\_trylock (&LockC); | | |
| 해결 방법 | | 이미 Lock된 뮤텍스에 pthread\_mutex\_trylock() 를 시도했을 경우 Try-lock 함수는 Redundant하게 되므로 Unlock된 뮤텍스에 사용한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Negative Shift Amount | 위험도 | General |
| 설명 | | 음수의 Shift 연산자를 사용하면 발생 | | |
| 발생가능 문제 | | 예상하지 못하는 값이 입력되어 오동작 유발 | | |
| 예제  코드 | Warning Example | ...  int a = 50;  /\* Negative Shift Amount warning issued here \*/  int b = a << -2;  ... | | |
| Fixed Warning Example | ...  int a = 50;  /\* Negative Shift Amount warning issued here \*/  int b = a << 2;  ... | | |
| 해결 방법 | | Shift 연산자 사용시 입력되는 Shift\_amount는 양수를 사용해야 한다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Function Call Has No Effect | 위험도 | General |
| 설명 | | 의미 없는 함수 호출(메모리 복사 함수 호출시의 length 인자값이 0으로 실행 될 때) | | |
| 발생가능 문제 | | 오동작 발생 가능성 존재 | | |
| 예제  코드 | Warning Example | /\*we might see if the programmer intended to set all the entries in buf to '\0', but forgot the correct argument order.\*/  char buf[10];  /\* Function Call Has No Effect  \* warning issued here \*/  memset (buf, 10, '\0'); | | |
| Fixed Warning Example | char buf[10];  memset (buf, 0, (sizeof (char) \* 10)) ; | | |
| 해결 방법 | | 함수 호출 시 각각의 인자값으로 0이 아닌 올바른 값을 넣는다. | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 결함이름 | | Shift Amount Exceeds Bit Width | 위험도 | General |
| 설명 | | 범위를 넘어선 Shift 연산을 수행 시 발생 | | |
| 발생가능 문제 | | 예상하지 못하는 값이 입력되어 오동작 유발 | | |
| 예제  코드 | Warning Example | ...  int a = 50;  int b = a << 200;  … | | |
| Fixed Warning Example | ...  int a = 50;  int b = a << 1;  … | | |
| 해결 방법 | | 해당 연산을 수행하는 변수의 범위를 체크하거나, 범위 내 연산을 수행한다. | | |

# **Annex A. Template**

## Head file template

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C HEADER FILE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* \*/

/\* Project : Example DCU \*/

/\* Filename : TPL.h \*/

/\* Base Project : xxxx DCU Ver 1.0 \*/

/\* Description : Template for C header file \*/

/\* Copyright 2017, KLST All rights reserved. \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef \_TPL\_INCLUDED

#define \_TPL\_INCLUDED

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* MODULES USED \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* DEFINITIONS AND MACROS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* TYPEDEFS AND STRUCTURES \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EXPORTED VARIABLES \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EXPORTED FUNCTIONS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EOF \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## 

## Source file template

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* C SOURCE FILE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* \*/

/\* Project : Example DCU \*/

/\* Filename : TPL.c \*/

/\* Base Project : xxxx DCU Ver 1.0 \*/

/\* Description : Template for C source file \*/

/\* Copyright 2017, KLST All rights reserved. \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* MODULES USED \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* DEFINITIONS AND MACROS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* TYPEDEFS AND STRUCTURES \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* PROTOTYPES OF LOCAL FUNCTIONS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EXPORTED VARIABLES \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* GLOBAL VARIABLES \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EXPORTED FUNCTIONS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* FUNCTION : \*/

/\* DESCRIPTION : \*/

/\* INPUT : \*/

/\* OUTPUT : \*/

/\* GLOBAL INPUT : \*/

/\* GLOBAL OUTPUT: \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Function\_Name (void)

{

Function\_Body ();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* LOCAL FUNCTIONS \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*/

/\* EOF \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# **Annex B. MISRA 2012 Rule set (Entire)**

| Group | Category | No. | Rule | Appliance |
| --- | --- | --- | --- | --- |
| The implementation | Required | Dir 1.1 | Any implementation-defined behavior on which  the output of the program depends shall be documented and understood | Not applied |
| Compilation and build | Required | Dir 2.1 | All source ﬁles shall compile without any compilation errors | Not applied |
| Requirements traceability | Required | Dir 3.1 | All code shall be traceable to documented requirements | Not applied |
| Code design | Required | Dir 4.1 | Run-time failures shall be minimized | Not applied |
|  | Advisory | Dir 4.2 | All usage of assembly language should be documented | Not applied |
|  | Required | Dir 4.3 | Assembly language shall be encapsulated and isolated | Not applied |
|  | Advisory | Dir 4.4 | Sections of code should not be “commented out” | Not applied |
|  | Advisory | Dir 4.5 | Identiﬁers in the same name space with overlapping visibility should be typographically unambiguous | Not applied |
|  | Advisory | Dir 4.6 | typedefs that indicate size and signedness should be used in place of the basic numerical types | Not applied |
|  | Required | Dir 4.7 | If a function returns error information, then that error information shall be tested | Not applied |
|  | Advisory | Dir 4.8 | If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden | Applied  (EN50128) |
|  | Advisory | Dir 4.9 | A function should be used in preference to a function-like macro where they are interchangeable | Not applied |
|  | Required | Dir 4.10 | Precautions shall be taken in order to prevent the contents of a header ﬁle being included more than once | Not applied |
|  | Required | Dir 4.11 | The validity of values passed to library functions shall be checked | Not applied |
|  | Required | Dir 4.12 | Dynamic memory allocation shall not be used | Applied  (EN50128) |
|  | Advisory | Dir 4.13 | Functions which are designed to provide operations on a resource should be called in an appropriate sequence | Not applied |
|  | Required | Rule 1.1 | The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits | Not applied |
|  | Advisory | Rule 1.2 | Language extensions should not be used | Not applied |
|  | Required | Rule 1.3 | There shall be no occurrence of undeﬁned or critical unspeciﬁed behaviour | Customer required |
| Unused code | Required | Rule 2.1 | A project shall not contain unreachable code | Customer required |
|  | Required | Rule 2.2 | There shall be no dead code | Customer required |
|  | Advisory | Rule 2.3 | A project should not contain unused type declarations | Not applied |
|  | Advisory | Rule 2.4 | A project should not contain unused tag declarations | Not applied |
|  | Advisory | Rule 2.5 | A project should not contain unused macro declarations | Not applied |
|  | Advisory | Rule 2.6 | A function should not contain unused label declarations | Not applied |
|  | Advisory | Rule 2.7 | There should be no unused parameters in functions | Not applied |
| Comments | Required | Rule 3.1 | The character sequences /\* and // shall not be used within a comment | Not applied |
|  | Required | Rule 3.2 | Line-splicing shall not be used in // comments | Not applied |
| Character sets and lexical conventions | Required | Rule 4.1 | Octal and hexadecimal escape sequences shall be terminated | Not applied |
|  | Advisory | Rule 4.2 | Trigraphs should not be used | Not applied |
| Identiﬁers | Required | Rule 5.1 | External identiﬁers shall be distinct | Applied  (EN50128) |
|  | Required | Rule 5.2 | Identiﬁers declared in the same scope and name space shall be distinct | Not applied |
|  | Required | Rule 5.3 | An identiﬁer declared in an inner scope shall not hide an identiﬁer declared in an outer scope | Customer required, Applied  (EN50128) |
|  | Required | Rule 5.4 | Macro identiﬁers shall be distinct | Not applied |
|  | Required | Rule 5.5 | Identiﬁers shall be distinct from macro names | Not applied |
|  | Required | Rule 5.6 | A typedef name shall be a unique identiﬁer | Not applied |
|  | Required | Rule 5.7 | A tag name shall be a unique identiﬁer | Not applied |
|  | Required | Rule 5.8 | Identiﬁers that deﬁne objects or functions with  external linkage shall be unique | Applied  (EN50128) |
|  | Advisory | Rule 5.9 | Identiﬁers that deﬁne objects or functions with internal linkage should be unique | Not applied |
| Types | Required | Rule 6.1 | Bit-ﬁelds shall only be declared with an appropriate type | Customer required |
|  | Required | Rule 6.2 | Single-bit named bit ﬁelds shall not be of a signed type | Not applied |
| Literals and constants | Required | Rule 7.1 | Octal constants shall not be used | Not applied |
|  | Required | Rule 7.2 | A “u” or “U” suﬃx shall be applied to all integer constants that are represented in an unsigned type | Not applied |
|  | Required | Rule 7.3 | The lowercase character “l” shall not be used in a literal suﬃx | Not applied |
|  | Required | Rule 7.4 | A string literal shall not be assigned to an object unless the object’s type is “pointer to const-qualiﬁed char ” | Applied  (EN50128) |
| Declarations and deﬁnitions | Required | Rule 8.1 | Types shall be explicitly speciﬁed | Customer required |
|  | Required | Rule 8.2 | Function types shall be in prototype form with named parameters | Customer required |
|  | Required | Rule 8.3 | All declarations of an object or function shall use the same names and type qualiﬁers | Customer required |
|  | Required | Rule 8.4 | A compatible declaration shall be visible when an object or function with external linkage is deﬁned | Customer required, Applied (EN50128) |
|  | Required | Rule 8.5 | An external object or function shall be declared once in one and only one ﬁle | Customer required, Applied  (EN50128) |
|  | Required | Rule 8.6 | An identiﬁer with external linkage shall have exactly one external deﬁnition | Applied  (EN50128) |
|  | Advisory | Rule 8.7 | Functions and objects should not be deﬁned with external linkage if they are referenced in only one translation unit | Not applied |
|  | Required | Rule 8.8 | The static storage class speciﬁer shall e used in all declarations of objects and functions that have internal linkage | Not applied |
|  | Advisory | Rule 8.9 | An object should be deﬁned at block scope if its identiﬁer only appears in a single function | Customer required |
|  | Required | Rule 8.10 | An inline function shall be declared with the static storage class | Not applied |
|  | Advisory | Rule 8.11 | When an array with external linkage is declared, its size should be explicitly speciﬁed | Customer required |
|  | Required | Rule 8.12 | Within an enumerator list, the value of an implicitly-speciﬁed enumeration  constant shall be unique | Customer required |
|  | Advisory | Rule 8.13 | A pointer should point to a const-qualiﬁed type whenever possible | Not applied |
|  | Required | Rule 8.14 | The restrict type qualiﬁer shall not be used | Not applied |
| Initialization | Mandatory | Rule 9.1 | The value of an object with automatic storage duration shall not be read before it has been set | Customer required |
|  | Required | Rule 9.2 | The initializer for an aggregate or union shall be enclosed in braces | Customer required |
|  | Required | Rule 9.3 | Arrays shall not be partially initialized | Customer required |
|  | Required | Rule 9.4 | An element of an object shall not be initialized more than once | Not applied |
|  | Required | Rule 9.5 | Where designated initializers are used to initialize an array object the size of the array shall be speciﬁed explicitly | Not applied |
| The essential type model | Required | Rule 10.1 | Operands shall not be of an inappropriate essential type | Customer required |
|  | Required | Rule 10.2 | Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations | Customer required |
|  | Required | Rule 10.3 | The value of an expression shall not be assigned to an object with a narrower essential type or of a diﬀerent essential type category | Customer required |
|  | Required | Rule 10.4 | Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential  type category | Customer required |
|  | Advisory | Rule 10.5 | The value of an expression should not be cast to an inappropriate essential type | Not applied |
|  | Required | Rule 10.6 | The value of a composite expression shall not be assigned to an object with wider essential type | Customer required |
|  | Required | 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 | Customer required |
|  | Required | Rule 10.8 | The value of a composite expression shall not be cast to a diﬀerent essential type category or a wider essential type | Customer required |
| Pointer type conversions | Required | Rule 11.1 | Conversions shall not be performed between a pointer to a function and any other type | Not applied |
|  | Required | Rule 11.2 | Conversions shall not be performed between a pointer to an incomplete type and any other type | Not applied |
|  | Required | Rule 11.3 | A cast shall not be performed between a pointer to object type and a pointer to a diﬀerent object type | Not applied |
|  | Advisory | Rule 11.4 | A conversion should not be performed between a pointer to object and an integer type | Not applied |
|  | Advisory | Rule 11.5 | A conversion should not be performed from pointer to void into pointer to object | Not applied |
|  | Required | Rule 11.6 | A cast shall not be performed between pointer to void and an arithmetic type | Not applied |
|  | Required | Rule 11.7 | A cast shall not be performed between pointer to object and a non-integer arithmetic type | Not applied |
|  | Required | Rule 11.8 | A cast shall not remove any const or volatile qualiﬁcation from the type pointed to by a pointer | Not applied |
|  | Required | Rule 11.9 | The macro NULL shall be the only permitted form of integer null pointer constant | Applied  (EN50128) |
| Expressions | Advisory | Rule 12.1 | The precedence of operators within expressions should be made explicit | Customer required |
|  | Required | Rule 12.2 | The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand | Customer required |
|  | Advisory | Rule 12.3 | The comma operator should not be used | Not applied |
|  | Advisory | Rule 12.4 | Evaluation of constant expressions should not lead  to unsigned integer wrap-around | Customer required |
| Side eﬀects | Required | Rule 13.1 | Initializer lists shall not contain persistent side eﬀects | Not applied |
|  | Required | Rule 13.2 | The value of an expression and its persistent side eﬀects shall be the same under all permitted evaluation orders | Customer required |
|  | Advisory | 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 | Not applied |
|  | Advisory | Rule 13.4 | The result of an assignment operator should not be used | Customer required |
|  | Required | Rule 13.5 | The right hand operand of a logical && or || operator shall not contain persistent side eﬀects | Customer required |
|  | Mandatory | Rule 13.6 | The operand of the sizeof operator shall not contain any expression which has potential side eﬀects | Customer required |
| Control statement expressions | Required | Rule 14.1 | A loop counter shall not have essentially ﬂoating type | Customer required |
|  | Required | Rule 14.2 | A for loop shall be well-formed | Customer required |
|  | Required | Rule 14.3 | Controlling expressions shall not be invariant | Customer required |
|  | Required | Rule 14.4 | The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type | Customer required |
| Control ﬂow | Advisory | Rule 15.1 | The goto statement should not be used | Applied  (EN50128) |
|  | Required | Rule 15.2 | The goto statement shall jump to a label declared later in the same function | Applied  (EN50128) |
|  | Required | Rule 15.3 | Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement | Applied  (EN50128) |
|  | Advisory | Rule 15.4 | There should be no more than one break or goto statement used to terminate any iteration statement | Applied  (EN50128) |
|  | Advisory | Rule 15.5 | A function should have a single point of exit at the end | Applied  (EN50128) |
|  | Required | Rule 15.6 | The body of an iteration-statement or a selection-statement shall be a compound-statement | Customer required |
|  | Required | Rule 15.7 | All if … else if constructs shall be terminated with an else statement | Not applied |
| Switch statements | Required | Rule 16.1 | All switch statements shall be well-formed | Customer required |
|  | Required | Rule 16.2 | A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement | Customer required |
|  | Required | Rule 16.3 | An unconditional break statement shall terminate every switch-clause | Customer required |
|  | Required | Rule 16.4 | Every switch statement shall have a default label | Not applied |
|  | Required | Rule 16.5 | A default label shall appear as either the ﬁrst or the last switch label of a switch statement | Not applied |
|  | Required | Rule 16.6 | Every switch statement shall have at least two switch-clauses | Not applied |
|  | Required | Rule 16.7 | A switch-expression shall not have essentially Boolean type | Customer required |
| Functions | Required | Rule 17.1 | The features of <stdarg.h> shall not be used | Customer required |
|  | Required | Rule 17.2 | Functions shall not call themselves, either directly or indirectly | Applied  (EN50128) |
|  | Mandatory | Rule 17.3 | A function shall not be declared implicitly | Customer required |
|  | Mandatory | Rule 17.4 | All exit paths from a function with non-void return type shall have an explicit return statement with an expression | Customer required, Applied  (EN50128) |
|  | Advisory | Rule 17.5 | The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements | Not applied |
|  | Mandatory | Rule 17.6 | The declaration of an array parameter shall not contain the static keyword between the [ ] | Customer required |
|  | Required | Rule 17.7 | The value returned by a function having non-void return type shall be used | Not applied |
|  | Advisory | Rule 17.8 | A function parameter should not be modiﬁed | Not applied |
| Pointers and arrays | Required | Rule 18.1 | A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand | Applied  (EN50128) |
|  | Required | Rule 18.2 | Subtraction between pointers shall only be applied to pointers that address elements of the same array | Applied  (EN50128) |
|  | Required | Rule 18.3 | The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object | Applied  (EN50128) |
|  | Advisory | Rule 18.4 | The +, -, += and -= operators should not be applied to an expression of pointer type | Customer required, Applied  (EN50128) |
|  | Advisory | Rule 18.5 | Declarations should contain no more than two levels of pointer nesting | Not applied |
|  | Required | Rule 18.6 | The address of an object with automatic storage shall not be copied to another object that persists after the ﬁrst object has ceased to exist | Applied  (EN50128) |
|  | Required | Rule 18.7 | Flexible array members shall not be declared | Applied  (EN50128) |
|  | Required | Rule 18.8 | Variable-length array types shall not be used | Not applied |
| Overlapping storage | Mandatory | Rule 19.1 | An object shall not be assigned or copied to an overlapping object | Customer required |
|  | Advisory | Rule 19.2 | The union keyword should not be used | Not applied |
| Preprocessing directives | Advisory | Rule 20.1 | #include directives should only be preceded by preprocessor directives or comments | Not applied |
|  | Required | Rule 20.2 | The ', " or \ characters and the /\* or // character sequences shall not occur in a header ﬁle name | Not applied |
|  | Required | Rule 20.3 | The #include directive shall be followed by either a <filename> or "filename" sequence | Not applied |
|  | Required | Rule 20.4 | A macro shall not be deﬁned with the same name as a keyword | Customer required |
|  | Advisory | Rule 20.5 | #undef should not be used | Not applied |
|  | Required | Rule 20.6 | Tokens that look like a preprocessing directive shall not occur within a macro argument | Not applied |
|  | Required | Rule 20.7 | Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses | Customer required |
|  | Required | Rule 20.8 | The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1 | Not applied |
|  | Required | Rule 20.9 | All identiﬁers used in the controlling expression of #if or #elif preprocessing directives shall be #deﬁne ’d before evaluation | Not applied |
|  | Advisory | Rule 20.10 | The # and ## preprocessor operators should not be used | Not applied |
|  | Required | Rule 20.11 | A macro parameter immediately following a # operator shall not immediately be followed by a ## operator | Not applied |
|  | Required | Rule 20.12 | A macro parameter 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 | Not applied |
|  | Required | Rule 20.13 | A line whose ﬁrst token is # shall be a valid reprocessing directive | Not applied |
|  | Required | 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 | Customer required |
| Standard libraries | Required | Rule 21.1 | #deﬁne and #undef shall not be used on a reserved identiﬁer or reserved macro name | Not applied |
|  | Required | Rule 21.2 | A reserved identiﬁer or macro name shall not be declared | Not applied |
|  | Required | Rule 21.3 | The memory allocation and deallocation functions of <stdlib.h> shall not be used | Applied  (EN50128) |
|  | Required | Rule 21.4 | The standard header ﬁle <setjmp.h> shall not be used | Not applied |
|  | Required | Rule 21.5 | The standard header ﬁle <signal.h> shall not be used | Not applied |
|  | Required | Rule 21.6 | The Standard Library input/output functions shall not be used | Not applied |
|  | Required | Rule 21.7 | The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used | Not applied |
|  | Required | Rule 21.8 | The library functions abort, exit, getenv and system of <stdlib.h> shall not be used | Not applied |
|  | Required | Rule 21.9 | The library functions bsearch and qsort of <stdlib.h> shall not be used | Not applied |
|  | Required | Rule 21.10 | The Standard Library time and date functions shall not be used | Not applied |
|  | Required | Rule 21.11 | The standard header ﬁle <tgmath.h> shall not be used | Not applied |
|  | Advisory | Rule 21.12 | The exception handling features of <fenv.h> should not be used | Not applied |
| Resources | Required | Rule 22.1 | All resources obtained dynamically by means of Standard Library functions shall be explicitly released | Not applied |
|  | Mandatory | Rule 22.2 | A block of memory shall only be freed if it was allocated by means of a Standard Library function | Customer required, Applied  (EN50128) |
|  | Required | Rule 22.3 | The same ﬁle shall not be open for read and write access at the same time on diﬀerent streams | Not applied |
|  | Mandatory | Rule 22.4 | There shall be no attempt to write to a stream which has been opened as read-only | Customer required |
|  | Mandatory | Rule 22.5 | A pointer to a FILE object shall not be dereferenced | Customer required |
|  | Mandatory | Rule 22.6 | The value of a pointer to a FILE shall  not be used after the associated  stream has been closed | Customer required |