Verifikation von C-Programmen

Vorlesung 4 vom 13.11.2014: MISRA-C: 2004
Guidelines for the use of the C language in critical systems

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

MISRA-Standard

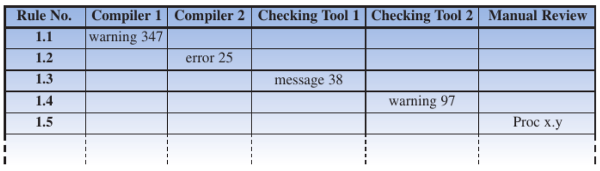
- Beispiel für eine Codierrichtlinie
- Erste Version 1998, letzte Auflage 2004
- Kostenpflichtig (£40,-/£10,-)
- Kein offener Standard
- Regeln: 121 verbindlich (required), 20 empfohlen (advisory)

Gliederung

Anwendung von MISRA-C (§4)

- §4.2: Training, Tool Selection, Style Guide
- §4.3: Adopting the subset
  - Produce a compliance matrix which states how each rule is enforced
  - Produce a deviation procedure
  - Formalise the working practices within the quality management system

MISRA Compliance Matrix



Rule No.	Compiler 1	Compiler 2	Checking Tool 1	Checking Tool 2	Manual Review
1.1	warning 347				
1.2		error 25			
1.3			message 38		
1.4				warning 97	
1.5					Proc x.y

Table 1: Example compliance matrix

Die Regeln (§5)

- Classification of rules:
  - Required (§5.1.1): "C code which is claimed to conform to this document shall comply with every required rule"
  - Advisory (§5.1.2):"should normally be followed", but not mandatory. "Does not mean that these items can be ignored, but that they should be followed as far as is reasonably practical."
- Organisation of rules (§5.4)
- Terminology (§5.5) — from C standard
- Scope(§5.6) : most can be checked for single translation unit

Environment

1.1 (req)	All code shall conform to ISO 9899:1990 "Programming languages — C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996 .	
1.2 (req)	No reliance shall be placed on undefined or unspecified behaviour .	2
1.3 (req)	Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.	1
1.4 (req)	The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.	1
1.5 (adv)	Floating-point implementations should comply with a defined floating-point standard .	1

Language extensions

2.1 (req)	Assembly language shall be encapsulated and isolated.	1
2.2 (req)	Source code shall only use /* ... */ style comments.	2
2.3 (req)	The character sequence /* shall not be used within a comment.	2
2.4 (adv)	Sections of code should not be "commented out".	2

Documentation

3.1 (req)	All usage of implementation-defined behaviour shall be documented.	3
3.2 (req)	The character set and the corresponding encoding shall be documented	1
3.3 (adv)	The implementation of integer division in the chosen compiler should be determined, documented and taken into account.	1
3.4 (req)	All uses of the #pragma directive shall be documented and explained.	1
3.5 (req)	The implementation-defined behaviour and packing of bitfields shall be documented if being relied upon.	1
3.6 (req)	All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation .	1

Character sets

4.1 (req)	Only those escape sequences that are defined in the ISO C standard shall be used.	1
4.2 (req)	Trigraphs shall not be used.	1

Identifiers

5.1 (req)	Identifiers (internal and external) shall not rely on the significance of more than 31 characters.	1
5.2 (req)	Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	1
5.3 (req)	A typedef name shall be a unique identifier.	2
5.4 (req)	A tag name shall be a unique identifier.	2
5.5 (adv)	No object or function identifier with static storage duration should be reused.	2
5.6 (adv)	No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.	2
5.7 (adv)	No identifier name should be reused.	2

Types

6.1 (req)	The plain char type shall be used only for storage and use of character values.	2
6.2 (req)	signed and unsigned char type shall be used only for the storage and use of numeric values.	2
6.3 (adv)	typedefs that indicate size and signedness should be used in place of the basic numerical types.	2
6.4 (req)	Bit fields shall only be defined to be of type unsigned int or signed int.	1
6.5 (req)	Bit fields of signed type shall be at least 2 bits long.	1

Constants

7.1 (req)	Octal constants (other than zero) and octal escape sequences shall not be used.	2

Declarations and definitions (I)

8.1 (req)	Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.	1
8.2 (req)	Whenever an object or function is declared or defined, its type shall be explicitly stated.	1
8.3 (req)	For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.	2
8.4 (req)	If objects or functions are declared more than once their types shall be compatible.	2
8.5 (req)	There shall be no definitions of objects or functions in a header file.	2

Declarations and definitions (II)

8.6 (req)	Functions shall be declared at file scope.	1
8.7 (req)	Objects shall be defined at block scope if they are only accessed from within a single function.	2
8.8 (req)	An external object or function shall be declared in one and only one file.	2
8.9 (req)	An identifier with external linkage shall have exactly one external definition.	2
8.10 (req)	All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.	3
8.11 (req)	The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.	3
8.12 (req)	When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialisation.	2

Initialisation

9.1 (req)	All automatic variables shall have been assigned a value before being used.	3
9.2 (req)	Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures.	1
9.3 (req)	In an enumerator list, the “=” construct shall not be used to explicitly initialise members other than the first, unless all items are explicitly initialised.	1

Arithmetic type conversions (I)

10.1 (req)	The value of an expression of integer type shall not be implicitly converted to a different underlying type if: a) it is not a conversion to a wider integer type of the same signedness, or b) the expression is complex, or c) the expression is not constant and is a function argument, or d) the expression is not constant and is a return expression.	2

Arithmetic type conversions (II)

10.2 (req)	The value of an expression of floating type shall not be implicitly converted to a different type if: a) it is not a conversion to a wider floating type, or b) the expression is complex, or c) the expression is a function argument, or d) the expression is a return expression.	1

Arithmetic type conversions (II)

10.3 (req)	The value of a complex expression of integer type shall only be cast to a type of the same signedness that is no wider than the underlying type of the expression.	2
10.4 (req)	The value of a complex expression of floating type shall only be cast to a floating type which is narrower or of the same size.	1
10.5 (req)	If the bitwise operators ˜ and $<<$ are applied to an operand of underlying type unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	2
10.6 (req)	A “U” suffix shall be applied to all constants of unsigned type.	2

Pointer type conversions

11.1 (req)	Conversions shall not be performed between a pointer to a function and any type other than an integral type.	1
11.2 (req)	Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.	1
11.3 (adv)	A cast should not be performed between a pointer type and an integral type.	1
11.4 (adv)	A cast should not be performed between a pointer to object type and a different pointer to object type.	1
11.5 (req)	A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.	1

Expressions (I)

12.1 (adv)	Limited dependence should be placed on C’s operator precedence rules in expressions.	3
12.2 (req)	The value of an expression shall be the same under any order of evaluation that the standard permits.	3
12.3 (req)	The sizeof operator shall not be used on expressions that contain side effects.	3
12.4 (req)	The right-hand operand of a logical && or || operator shall not contain side effects.	3
12.5 (req)	The operands of a logical && or || shall be primary-expressions.	3
12.6 (adv)	The operands of logical operators (&&, || and !) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, ||, !, =, ==, !=, and ?:).	3

Expressions (II)

12.7 (req)	Bitwise operators shall not be applied to operands whose underlying type is signed.	2
12.8 (req)	The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand.	3
12.9 (req)	The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	2
12.10 (req)	The comma operator shall not be used.	1
12.11 (adv)	Evaluation of constant unsigned integer expressions should not lead to wrap-around.	3
12.12 (req)	The underlying bit representations of floating-point values shall not be used.	3
12.13 (adv)	The increment (++) and decrement (–) operators should not be mixed with other operators in an expression.	1

Control statement expressions

13.1 (req)	Assignment operators shall not be used in expressions that yield a Boolean value.	1
13.2 (adv)	Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.	3
13.3 (req)	Floating-point expressions shall not be tested for equality or inequality.	1
13.4 (req)	The controlling expression of a for statement shall not contain any objects of floating type.	1
13.5 (req)	The three expressions of a for statement shall be concerned only with loop control.	1
13.6 (req)	Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	3
13.7 (req)	Boolean operations whose results are invariant shall not be permitted.	3

Control flow (I)

14.1 (req)	There shall be no unreachable code.	3
14.2 (req)	All non-null statements shall either: a) have at least one side effect however executed, or b) cause control flow to change.	3
14.3 (req)	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	3
14.4 (req)	The goto statement shall not be used.	1
14.5 (req)	The continue statement shall not be used.	1
14.6 (req)	For any iteration statement there shall be at most one break statement used for loop termination.	2

Control flow (I)

14.7 (req)	A function shall have a single point of exit at the end of the function.	1
14.8 (req)	The statement forming the body of a switch, while, do … while or for statement be a compound statement.	1
14.9 (req)	An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	1
14.10 (req)	All if … else if constructs shall be terminated with an else clause.	1

Switch statements

15.1 (req)	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	1
15.2 (req)	An unconditional break statement shall terminate every non-empty switch clause.	1
15.3 (req)	The final clause of a switch statement shall be the default clause.	1
15.4 (req)	A switch expression shall not represent a value that is effectively Boolean.	1
15.5 (req)	Every switch statement shall have at least one case clause.	1

Functions (I)

16.1 (req)	Functions shall not be defined with variable numbers of arguments.	1
16.2 (req)	Functions shall not call themselves, either directly or indirectly.	3
16.3 (req)	Identifiers shall be given for all of the parameters in a function prototype declaration.	1
16.4 (req)	The identifiers used in the declaration and definition of a function shall be identical.	1
16.5 (req)	Functions with no parameters shall be declared and defined with the parameter list void.	1
16.6 (req)	The number of arguments passed to a function shall match the number of parameters.	2
16.7 (adv)	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	3

Functions (I)

16.8 (req)	All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	3
16.9 (req)	A function identifier shall only be used with either a preceding &, or with a parenthesised parameter list, which may be empty.	1
16.10 (req)	If a function returns error information, then that error information shall be tested.	3

Pointers and arrays

17.1 (req)	Pointer arithmetic shall only be applied to pointers that address an array or array element.	3
17.2 (req)	Pointer subtraction shall only be applied to pointers that address elements of the same array.	3
17.3 (req)	$>$, $>=$, $<$, $<=$ shall not be applied to pointer types except where they point to the same array.	3
17.4 (req)	Array indexing shall be the only allowed form of pointer arithmetic.	3
17.5 (adv)	The declaration of objects should contain no more than 2 levels of pointer indirection.	1
17.6 (req)	The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	3

Structures and unions

18.1 (req)	All structure or union types shall be complete at the end of a translation unit.	3
18.2 (req)	An object shall not be assigned to an overlapping object.	3
18.3 (req)	An area of memory shall not be reused for unrelated purposes.	x
18.4 (req)	Unions shall not be used.	1

Preprocessing directives (I)

19.1 (adv)	`#include` statements in a file should only be preceded by other preprocessor directives or comments.	3
19.2 (adv)	Non-standard characters should not occur in header file names in `#include` directives.	3
19.3 (req)	The #include directive shall be followed by either a `<filename>` or `"filename"` sequence.	3
19.4 (req)	C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.	3
19.5 (req)	Macros shall not be `#define`'d or `#undef`'d within a block.	x
19.6 (req)	#undef shall not be used.	2
19.7 (adv)	A function should be used in preference to a function-like macro.	3
19.8 (req)	A function-like macro shall not be invoked without all of its arguments.	3

Preprocessing directives (II)

19.9 (req)	Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	3
19.10 (req)	In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.	3
19.11 (req)	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	3
19.12 (req)	There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.	3
19.13 (adv)	The # and ## preprocessor operators should not be used.	3

Preprocessing directives (III)

19.14 (req)	The defined preprocessor operator shall only be used in one of the two standard forms.	3
19.15 (req)	Precautions shall be taken in order to prevent the contents of a header file being included twice.	3
19.16 (req)	Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.	3
19.17 (req)	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.	3

Standard libraries (I)

20.1 (req)	Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.	3
20.2 (req)	The names of standard library macros, objects and functions shall not be reused.	3
20.3 (req)	The validity of values passed to library functions shall be checked.	3
20.4 (req)	Dynamic heap memory allocation shall not be used.	2
20.5 (req)	The error indicator errno shall not be used.	2
20.6 (req)	The macro offsetof, in library <stddef.h>, shall not be used.	2
20.7 (req)	The setjmp macro and the longjmp function shall not be used.	2

Standard libraries (II)

20.8 (req)	The signal handling facilities of <signal.h> shall not be used.	2
20.9 (req)	The input/output library <stdio.h> shall not be used in production code.	2
20.10 (req)	The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	2
20.11 (req)	The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	2
20.12 (req)	The time handling functions of library <time.h> shall not be used.	2

Run-time failures

21.1 (req)	Minimisation of run-time failures shall be ensured by the use of at least one of:
	a) static analysis tools/techniques;
	b) dynamic analysis tools/techniques;
	c) explicit coding of checks to handle run-time faults.
		3

MISRA-C in der Praxis

- Meiste Werkzeuge kommerziell
- Entwicklung eines MISRA-Prüfwerkzeugs im Rahmen des SAMS-Projektes
  - Diplomarbeit Hennes Maertins (Juni 2010)
- Herausforderungen:
  - Parser und erweiterte Typprüfung für C
  - Re-Implementierung des Präprozessors
  - Einige Regeln sind unentscheidbar
  - Dateiübergreifende Regeln
- Implementierung:
  - 20 KLoc Haskell, im Rahmen des SAMS-Werkzeugs (SVT)