MISRA C:2012

Guidelines for the use of the C language in critical systems

March 2013

First published March 2013 by MIRA Limited
Watling Street
Nuneaton
Warwickshire
CV10 0TU
UK

www.misra.org.uk

ISBN 978-1-906400-10-1 paperback
ISBN 978-1-906400-11-8 PDF

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

MISRA C:2012

Guidelines for the use of the C language in critical systems

March 2013

MISRA Mission Statement

We provide world-leading, best practice guidelines for the safe application of both embedded control systems and standalone software.

MISRA, The Motor Industry Software Reliability Association, is a collaboration between manufacturers, component suppliers and engineering consultancies which seeks to promote best practice in developing safety-related embedded electronic systems and other software-intensive applications. To this end, MISRA publishes documents that provide accessible information for engineers and management, and holds events to permit the exchange of experiences between practitioners.

www.misra.org.uk

Foreword

At first sight, this third revision of the MISRA C Guidelines may seem somewhat daunting. Since it is roughly twice the size of the previous revision, one might think that it contains twice as many guidelines, and that compliance with those guidelines might take twice as much effort.

In fact, the increase in the number of guidelines is relatively modest at around 10%. The remainder of the increase in size is due to improvements in the guidance given, such as:

- Better rationales for guidelines;
- More precise descriptions;
- Code examples, showing compliance and non-compliance, for most of the guidelines;
- More detailed guidance on compliance checking, and the deviation procedure;
- Checklists that can be used to support a compliance statement.

Finally, I would like to draw attention to the introductory sections of the document. These not only contain practical guidance on how to use MISRA C but, at the same time, have been made more concise than their predecessors. I encourage all users to familiarize themselves with this material.

Steve Montgomery MA (Cantab), PhD
Chairman, MISRA C Working Group

Acknowledgements

The MISRA consortium would like to thank the following individuals for their significant contribution to the writing of this document:

Dave Banham	Rolls-Royce plc (previously of Alstom Grid)
Andrew Banks	Intuitive Consulting
Mark Bradbury	Aero Engine Controls
Paul Burden	Programming Research Ltd
Mark Dawson-Butterworth	Zytek Automotive Ltd
Mike Hennell	LDRA Ltd
Chris Hills	Phaedrus Systems Ltd
Steve Montgomery	Ricardo UK Ltd
Chris Tapp	LDRA Ltd (also Keylevel Consultants Ltd)
Liz Whiting	LDRA Ltd (previously of QinetiQ plc)

The MISRA consortium also wishes to acknowledge contributions from the following individuals during the development and review process:

Roberto Bagnara	William Forbes	Voilmy Laurent	Koki Onoda
John Bailey	Takao Futagami	Fred Long	Paulo Pinheiro
Johan Bezem	Jim Gimpel	Daniel Lundin	Mohanraj Ragupathi
Gunter Blache	Gilles Goulas	Gavin McCall	Paul Rigas
Michael Burke	Wolfgang von Hansen	Douglas Mearns	Andrew Scholan
Andrew Burnard	Takahiro Hashimoto	Svante Möller	Marco Sorich
Paul Butler	Dave Higham	Frederic Mondot	Takuji Takuma
Mirko Conrad	Shinya Ito	Jürgen Mottok	Martin Thompson
David Cozens	David Jennings	Yannick Moy	Takafumi Wakita
David Crocker	Peter Jesty	Alexander Much	David Ward
Greg Davis	Grzegorz Konopko	Robert Mummé	Tetsuhiro Yamamoto
Manoj Dwivedi	Taneli Korhonen	Tadanori Nakagawa	Naoki Yoshikawa
Carl Edmunds	Joel Kuehner	Greg Newman	Achim Olaf Zacher

Particular thanks are due to David Crocker for his significant contribution towards the development of Appendix H.

The descriptions of implementation-defined behaviours in Appendix G have been reproduced from versions of the ISO Standards published by BSI Standards Limited; the text is identical to that in the ISO versions. Permission to reproduce extracts from British Standards is granted by the BSI Standards Limited (BSI) under Licence No. 2013ET0003. No other use of this material is permitted. British Standards can be obtained in PDF or hard copy formats from the BSI online shop: www.bsigroup.com/Shop or by contacting BSI Customer Services for hard copies only: Tel: +44 20 8996 9001, Email: cservices@bsigroup.com.

DokuWiki was used extensively during the drafting of this document. Our thanks go to all those involved in its development.

Contents

1 The vision

The MISRA C Guidelines define a subset of the C language in which the opportunity to make mistakes is either removed or reduced. Many standards for the development of safety-related software require, or recommend, the use of a language subset, and this can also be used to develop any application with high integrity or high reliability requirements.

As well as defining this subset, these MISRA C Guidelines provide:

- Educational material for those developing C programs;
- Reference material for tool developers.

Previous editions of MISRA C have been based on the 1990 ISO definition of C. As the 1999 ISO definition has now been adopted, in varying degrees, by embedded implementations it was considered that the time was right to publish a new edition of MISRA C which recognized the 1999 ISO definition.

Each aspect of the guidance presented in the previous edition has been comprehensively reviewed and improved where appropriate. This third edition also incorporates material created in response to the feedback that has been provided by users of earlier editions of the Guidelines.

A major change in this third edition is the development of the second edition's concept of *underlying type* into the *essential type*. Using the new *essential type* concept, it has been possible to develop a set of guidelines that bring stronger typing to the C language.

The vision for the third edition of MISRA C is therefore to:

- Adopt the 1999 ISO definition of the C language, while retaining support for the older 1990 definition;
- Correct any known issues with the second edition;
- Add new guidelines for which there is a strong rationale;
- Improve the specification and the rationale for existing guidelines;
- Remove any guidelines for which the rationale is insufficient;
- Increase the number of guidelines that can be processed by static analysis tools;
- Provide guidance on the applicability of the guidelines to automatically-generated code.

2 Background to MISRA C

2.1 The popularity of C

The C programming language is popular because:

- C compilers are readily available for many processors;
- C programs can be compiled to efficient machine code;
- It is defined by an international standard;
- It provides mechanisms to access the input/output capabilities of the target processor, whether directly or by means of language extensions;
- There is a considerable body of experience with using C in critical systems;
- It is widely supported by static analysis and test tools.

2.2 Disadvantages of C

While popular, the language has several drawbacks which are discussed in the following sub-sections.

2.2.1 Language definition

The ISO Standard does not specify the language completely but places some aspects under the control of an implementation. This is intentional, partly because of the desire to support many pre-existing implementations for widely different target processors.

As a result there are areas of the language in which:

- The behaviour is undefined;
- The behaviour is unspecified;
- An implementation is free to choose its own behaviour provided that it is documented.

A program that relies on undefined or unspecified behaviour is not necessarily guaranteed to behave in a predictable manner.

A program that places excessive reliance on implementation-defined behaviour may be difficult to port to a different target. The presence of implementation-defined behaviour may also hinder static analysis if it is not possible to configure the analyser to handle it.

2.2.2 Language misuse

While C programs can be laid out in a structured and comprehensible manner, C makes it easy for programmers to write obscure code that is difficult to understand.

The specification of the operators makes it difficult for programming errors to be detected by a compiler. For example, the following two fragments of code are both perfectly legal so it is impossible for a compiler to know whether one has been mistakenly used in place of the other:

```
if ( a == b )    /* tests whether a and b are equal                */
if ( a = b )     /* assigns b to a and tests whether a is non-zero */
```

2.2.3 Language misunderstanding

There are areas of the language that are commonly misunderstood by programmers. For example, C has more operators than some other languages and consequently has a high number of different operator precedence levels, some of which are not intuitive.

The type rules provided by C can also be confusing to programmers who are familiar with strongly-typed languages. For example, operands may be "promoted" to wider types, meaning that the type resulting from an operation is not necessarily the same as that of the operands.

2.2.4 Run-time error checking

C programs can be compiled into small and efficient machine code, but the trade-off is that there is a very limited degree of run-time checking. C programs generally do not provide run-time checking for common problems such as arithmetic exceptions (e.g. divide by zero), overflow, validity of pointers, or array bound errors. The C philosophy is that the programmer is responsible for making such checks explicitly.

3 Tool selection

3.1 The C language and its compiler

When work started on this document, the ISO standard for the C language was ISO/IEC 9899:1999 [8] as corrected by [9], [10] and [11]. This version of the language is hereafter referred to as C99. However, many compilers are still available for its predecessor, defined by ISO/IEC 9899:1990 [2] as amended and corrected by [4], [5] and [6], and hereafter referred to as C90. Some compilers provide an option to select between the two versions of the language.

The current ISO standard for the C language is ISO/IEC 9899:2011 [13] as corrected by [14]. When this standard was published, work on the MISRA C Guidelines had been nearly completed and it has therefore not been possible to incorporate guidance for this later version.

Note: access to a copy of the relevant standard is not necessary for the use of MISRA C, but it may be helpful.

The choice between C90 and C99 might be affected by factors such as the amount of legacy code being reused on the project and the availability of compilers for the target processor.

The compiler selected for the project should meet the requirements of a conforming freestanding implementation for the chosen version of the C language. It may exceed these requirements, for example by providing all the features of the language (a freestanding implementation need only provide a well-defined subset), or it might provide extensions as permitted by the language standard.

Ideally, confirmation that the compiler is indeed conforming should be supplied by the compiler developer, for example by providing details of the conformance tests that were run and the results that were obtained.

Sometimes, there may be a limited choice of compilers whose quality may not be known. If it proves difficult to obtain information from the compiler developers, the following steps could be taken to assist in the selection process:

- Checking that the compiler developer follows a suitable software development process, e.g. one that meets the requirements of ISO 9001:2008 [21] as assessed using ISO 90003:2004 [22];
- Reading reviews of the compiler and user experience reports;
- Reviewing the size of the user base and types of applications developed using the compiler;
- Performing and documenting independent validation testing, e.g. by using a conformance test suite or by compiling existing applications.

Note: some process standards require qualification of compilers in some situations, e.g. IEC 61508:2010 [32], ISO 26262:2011 [23] and DO-178C [24].

3.2 Analysis tools

It is possible to check that C source code complies with MISRA C by means of inspection alone. However, this is likely to be extremely time-consuming and error prone. Any realistic process for checking code against MISRA C will therefore involve the use of at least one static analysis tool.

All of the factors that apply to compiler selection apply also to the selection of static analysis tools, although the validation of analysis tools is a little different from that of compilers. An ideal static analysis tool would:

- Detect all violations of MISRA C guidelines;
- Produce no "false positives", i.e. would only report genuine violations and would not report non-violations or possible violations.

For the reasons explained in Section 6.5, it is not, and never will be, possible to produce a static analysis tool that meets this ideal behaviour. The ability to detect the maximum number of violations possible, while minimizing the number of false positive messages, is therefore an important factor in choosing a tool.

There is a wide range of tools available with execution times ranging from seconds to days. Broadly speaking, tools that consume less time are more likely to produce false positives than those that consume large amounts of time. Consideration should also be given to the balance between analysis time and analysis precision during tool selection.

Analysis tools vary in their emphasis. Some might be general purpose, whereas others might focus on performing a thorough analysis of a subset of potential issues. Thus it might be necessary to use more than one tool in order to maximize the coverage of issues.

Note: some process standards, including IEC 61508:2010 [32], ISO 26262:2011 [23] and DO-178C [24], require the qualification of verification tools such as static analysers in some situations.

4 Prerequisite knowledge

4.1 Training

In order to ensure an appropriate level of skill and competence on the part of those who produce C source code, formal training should be provided for:

- The use of the C programming language for embedded applications;
- The use of the C programming language for high-integrity and safety-related systems.

Since compilers and static analysis tools are complex pieces of software, consideration should also be given to providing training in their use. In the case of static analysis tools, it might be possible to obtain training in their use specifically in relation to MISRA C.

4.2 Understanding the compiler

In this document, the term "compiler", referred to as "the implementation" by the ISO C standards [2], [8], means the compiler itself as well as any associated tools such as a linker, library manager and executable file format conversion tools.

The compiler may provide various options that control its behaviour. It is important to understand the effect of these options as their use, or non-use, might affect:

- The availability of language extensions;
- The conformance of the compiler with the ISO C standards;
- The resources, especially processing time and memory space, required by the program;
- The likelihood that a defect in the compiler will be exposed, such as might occur when complex highly-optimizing code transformations are performed.

It is important to understand the way in which the compiler implements those features of the C language that are termed "implementation-defined" in the ISO C standards. It is also important to understand any language extensions that the compiler may provide.

The compiler developer may maintain a list of defects that are known to affect the compiler along with any workarounds that are available. Knowledge of the contents of this list would clearly be advantageous before starting a project using that compiler. If such a list is not available from the compiler developer then a local list should be maintained whenever a defect, or suspected defect, is discovered and reported to the compiler developer.

4.3 Understanding the static analysis tools

The documentation for each static analysis tool being used on a project should be reviewed in order to understand:

- How to configure the analyser to match the compiler's implementation-defined behaviour, e.g. sizes of the integer types;
- Which MISRA C guidelines the analyser is capable of checking (Section 5.3);
- Whether it is possible to configure the analyser to handle any language extensions that will be used;
- Whether it is possible to adjust the analyser's behaviour in order to achieve a different balance between analysis time and analysis precision.

The tool developer may maintain a list of defects that are known to affect the tool along with any workarounds that are available. Knowledge of the contents of this list would clearly be advantageous before starting a project using that tool. If such a list is not available from the tool developer then a local list should be maintained whenever a defect or suspected defect is discovered and reported to the tool developer.

5 Adopting and using MISRA C

5.1 Adoption

MISRA C should be adopted from the outset of a project.

If a project is building on existing code that has a proven track record then the benefits of compliance with MISRA C may be outweighed by the risks of introducing a defect when making the code compliant. In such cases a judgement on adopting MISRA C should be made based on the net benefit likely to be obtained.

5.2 Software development process

MISRA C is intended to be used within the framework of a documented software development process. While MISRA C can be used in isolation, it is of greater benefit when used within a process to ensure that other activities have been performed correctly so that, for example:

- The software requirements, including any safety requirements, are complete, unambiguous and correct;
- The design specifications reaching the coding phase are correct, consistent with the requirements and do not contain any other functionality;
- The object modules produced by the compiler behave as specified in the corresponding designs;
- The object modules have been tested, individually and together, to identify and eliminate errors.

MISRA C should be used by programmers before code is submitted for review or unit testing. A project that checks for MISRA C compliance late in its lifecycle is likely to spend a considerable amount of time re-coding, re-reviewing and re-testing. It is therefore expected that the software development process will require the early application of MISRA C principles.

A full discussion of the requirements for safety-related software development processes is outside the scope of this document. Examples of development processes may be found in standards and guidelines such as IEC 61508:2010 [32], ISO 26262:2011 [23], DO-178C [24], EN 50128:2011 [33] and IEC 62304:2006 [34]. The remainder of this section deals with the interaction between MISRA C and the software development process.

5.2.1 Process activities required by MISRA C

In order to use MISRA C, it is necessary to develop and document:

- A compliance matrix, showing how compliance with each MISRA C guideline will be checked;
- A deviation process by which justifiable non-compliances can be authorized and recorded.

The software development process should also document the steps that will be taken to avoid run-time errors, and to demonstrate that they have been avoided. For example, it should include descriptions of the processes by which it is demonstrated and recorded that:

- The execution environment provides sufficient resources, especially processing time and stack space, for the program;
- Run-time errors, such as arithmetic overflow, are absent from areas of the program: for example by virtue of code that checks the ranges of inputs to a calculation.

5.2.2 Process activities expected by MISRA C

It is recognized that a consistent style assists programmers in understanding code written by others. However, since style is a matter for individual organizations, MISRA C does not make any recommendations related purely to programming style. It is expected that local style guides will be developed and used as part of the software development process.

The use of metrics is recommended by many software process standards as a means to identify code that may require additional review and testing effort. However, the nature of the metrics being collected, and their corresponding thresholds, will be determined by the industry, organization and/or the nature of the project. This document, therefore, does not offer any guidance on software metrics.

5.3 Compliance

In order to ensure that code complies with all of the MISRA C guidelines, a compliance matrix should be produced. This matrix lists each guideline and indicates how it is to be checked. For most guidelines, the easiest, most reliable and most cost-effective means of checking will be to use a static analysis tool or tools, the compiler, or a combination of these. Where a guideline cannot be completely checked by a tool, then a manual review will be required.

Some automatic code generator developers supply a compliance statement along with their tools. This statement identifies those MISRA C guidelines that will not be violated by the generated code provided the tool has been configured as specified by the tool developer. For example it might:

- List the guidelines that are not violated; or
- State that all mandatory and required guidelines are not violated.

The use of an automatic code generator compliance statement can significantly reduce the number of MISRA C guidelines that need to be checked by other means.

See Table 1 for an example of a compliance matrix, and see Appendix A for a summary of the guidelines, which could be used to assist in generating a full compliance matrix.

Guideline	Compilers		Checking tools		Manual review
	'A'	'B'	'A'	'B'	
Dir 1.1					Procedure x
Dir 2.1	no errors	no errors			
...					
Rule 4.1			message 38		
Rule 4.2				warning 97	
Rule 5.1	warning 347				
...					

Table 1: Example compliance matrix

Once a compliance matrix is in place, the compilers and analysers can be configured and used to generate a list of messages that require investigation.

The following information should be recorded for each tool used in the checking process:

- Version number;
- Options used when invoking the tool;
- Any configuration data that the tool uses.

5.3.1 Compiler configuration

If the compiler provides a choice between C90 and C99, it must be configured for the variant being used on the project. Similarly, if the compiler provides a choice of targets, it must be configured for the correct variant of the selected target.

The compiler's optimization options should be reviewed and selected carefully in order to ensure that an appropriate balance between execution speed and code size has been obtained. Using more aggressive optimizations may increase the risk of exposing defects in the compiler.

Even if the compiler is not being used to ensure compliance, it may produce messages during the translation process that indicate the presence of potential defects. If the compiler provides control over the number and nature of the messages it produces, these should be reviewed and an appropriate level selected.

5.3.2 Static analysis tool configuration

While a compiler is usually specific to a particular processor, or family of processors, static analysis tools tend to be general purpose. It is therefore important to configure each static analysis tool to reflect the implementation decisions made by the compiler. For example, static analysers need to know the sizes of the integer types.

If the static analyser supports both C90 and C99, it will need to be configured to match the variant in use on the project.

If possible, the static analyser should be configured to support language extensions. If this is not possible then an alternative procedure will need to be put in place for checking that the code conforms to the extended language.

It may also be necessary to configure a static analyser in order to allow it to check certain guidelines. For example, unless a project is using *_Bool* to represent Boolean data, an analyser needs to be made aware of the *essentially Boolean* type (Appendix D) in order to check some MISRA C guidelines.

5.3.3 Investigating messages

The messages produced by the compliance checking process, or by the translation process, fall into one of the following categories:

1. Correct diagnosis of a MISRA C guideline violation;
2. Diagnosis of a possible MISRA C guideline violation;
3. False diagnosis of a MISRA C guideline violation;
4. Diagnosis of something that is not a MISRA C guideline violation.

The preferred remedy for any messages in category (1) is to correct the source code in order to make it compliant with the MISRA C guideline. If it is undesirable or impossible to render the code MISRA C compliant then a deviation may need to be raised (see Section 5.4).

Any messages in the other categories should be investigated. Sometimes, the easiest and quickest solution will be to modify the source code to eliminate the message. However, this may not always be possible or desirable, in which case a record of the investigation should be kept. The purpose of the record is to:

- Explain why, despite diagnosis of a possible violation, the code complies with the MISRA C guideline for category (2) messages;
- Explain and, if possible, obtain the tool developer's agreement that the tool has incorrectly diagnosed a violation of the MISRA C guideline for category (3) messages;
- Justify why the message can safely be disregarded for category (4) messages.

All records of such investigations should be reviewed and approved by an appropriately qualified technical authority.

5.4 Deviation procedure

In some instances it may be necessary to deviate from the guidelines given in this document. For example, a common method of accessing memory-mapped I/O ports at fixed addresses does not comply with MISRA C because it involves converting an integer to a pointer:

```
#define PORT (*(volatile unsigned char *)0x0002)

PORT = 0x10u;
```

It is important that such deviations are properly recorded and authorized. Individual programmers can be prevented from deviating guidelines by means of a formal authorization procedure. Formal record-keeping is good practice and supports any safety argument that might be made for the software.

The deviation procedure should be formalized within the software development process. No process is imposed by MISRA C because methods used will vary between organizations, for example:

- The physical methods used to raise, review and approve a deviation;
- The degree of review and evidence required, which may vary according to the guideline in question, before a deviation can be approved.

Each MISRA C guideline is given a category which affects the way in which deviations may be applied (see Section 6.2).

Deviations are divided into two categories: Project Deviations and Specific Deviations.

A Project Deviation is used when a MISRA C guideline is deviated in a particular class of circumstances. For example, if memory-mapped I/O is used in a communications driver then it might be appropriate to raise a Project Deviation to cover use of memory-mapped I/O in all files making up that driver. Project Deviations are typically, though not exclusively, raised early in the project.

A Specific Deviation is used when a MISRA C guideline is deviated for a single instance in a single file.

A deviation record should include:

- The guideline being deviated;
- The circumstances in which the deviation is permitted: for Project Deviations this may be a generic description such as "all code in module X" but for Specific Deviations it will identify the location at which the non-compliant code occurs;
- Justification for the deviation, including an assessment of the risks associating with deviating compared with other possibilities;
- Demonstration of how safety is assured, for example a proof that an unsafe condition cannot occur, together with any additional reviews or tests that are required;
- Potential consequences of the deviation and any mitigating actions that have been taken.

An example deviation record is provided in Appendix I.

5.5 Claiming compliance

Compliance cannot be claimed for an organization, only for a project.

When claiming compliance with MISRA C for a project, a developer is stating that evidence exists to show:

1. A compliance matrix has been completed which shows how compliance has been enforced;
2. All the C code in the project is compliant with the guidelines of this document, or is subject to approved deviations;
3. There is a record of each approved deviation;
4. The guidance given in this section, as well as that given in Section 3 and Section 4, has been adopted;
5. Staff are suitably skilled and have a sufficient range of experience.

Note: when claiming that code complies with the guidelines, the assumption is being made that the tools or reviewers have identified all non-compliances. Since other tools or reviewers might identify different non-compliances, it is important to recognize that compliance claims are not absolute, but depend on the checking process used.

A summary of the guidance referred to in point (4) is provided in the form of a checklist in Appendix F.

6 Introduction to the guidelines

This section explains the presentation of the guidelines in Section 7 and Section 8, the main content of this document, and serves as an introduction to those sections.

6.1 Guideline classification

Every MISRA C guideline is classified as either being a "rule" or a "directive".

A directive is a guideline for which it is not possible to provide the full description necessary to perform a check for compliance. Additional information, such as might be provided in design documents or requirements specifications, is required in order to be able to perform the check. Static analysis tools may be able to assist in checking compliance with directives but different tools may place widely different interpretations on what constitutes a non-compliance.

A rule is a guideline for which a complete description of the requirement has been provided. It should be possible to check that source code complies with a rule without needing any other information. In particular, static analysis tools should be capable of checking compliance with rules subject to the limitations described in Section 6.5.

Section 7 contains all the directives and Section 8 contains all the rules.

6.2 Guideline categories

Every MISRA C guideline is given a single category of "mandatory", "required" or "advisory", whose meanings are described below. Beyond this basic classification the document does not give, nor intend to imply, any grading of importance of each of the guidelines. All required guidelines, whether rules or directives, should be considered to be of equal importance, as should all mandatory and advisory ones.

6.2.1 Mandatory guidelines

C code which is claimed to conform to this document shall comply with every mandatory guideline — deviation from mandatory guidelines is not permitted.

Note: if a checking tool produces a diagnostic message, this does not necessarily mean that a guideline has been violated for the reasons given in Section 6.5.

6.2.2 Required guidelines

C code which is claimed to conform to this document shall comply with every required guideline, with a formal deviation required, as described in Section 5.4, where this is not the case.

An organization or project may choose to treat any required guideline as if it were mandatory.

6.2.3 Advisory guidelines

These are recommendations. However, the status of "advisory" does not mean that these items can be ignored, but rather that they should be followed as far as is reasonably practical. Formal deviation is not necessary for advisory guidelines but, if the formal deviation process is not followed, alternative arrangements should be made for documenting non-compliances.

An organization or project may choose to treat any advisory guideline as if it were mandatory or required.

6.3 Organization of guidelines

The guidelines are organized under different topics within the C language. However there is inevitably overlap, with one guideline possibly being relevant to a number of topics. Where this is the case the guideline has been placed under the most relevant topic.

6.4 Redundancy in the guidelines

There are a few cases within this document where a guideline is given that refers to a language feature that is banned or advised against elsewhere in the document. This is intentional. It may be that the user chooses to use that feature, either by raising a deviation against a required guideline, or by choosing not to follow an advisory guideline. In this case the second guideline, constraining the use of that feature, becomes relevant.

6.5 Decidability of rules

Each mandatory, required and advisory rule is classified as *decidable* or *undecidable*. This classification describes the theoretical ability of a static analyser to answer the question "Does this code comply with this rule?" The directives are **not** classified in this way because it is impossible, given only the source code, to devise an algorithm that could guarantee to check for compliance.

A rule is *decidable* if it is possible for a program to answer the question with a "yes" or a "no" **in every case** and *undecidable* otherwise. A review of the theory of computation, on which this classification is based, is beyond the scope of this document but a rule is likely to be undecidable if detecting violations depends on run-time properties such as:

- The value that an object holds;
- Whether control reaches a particular point in the program.

Decidable rules have useful properties with regard to static analysis. Provided that a defect-free and complete static analyser is configured correctly:

- A reported violation of a decidable rule indicates a real violation;
- No reported violation of a decidable rule indicates there are no violations in the code being analysed.

Some examples of decidable rules are:

- Rule 5.2: depends on the names and scopes of identifiers;
- Rule 11.3: depends on the source pointer and destination pointer types;
- Rule 20.7: depends on the syntactic form of the result of a macro expansion.

Static analysers vary in their ability to detect violations of undecidable rules:

- A reported violation of an undecidable rule may not necessarily indicate a real violation; some analysers take the approach of reporting **possible** violations to remind users of the uncertainty;
- No reported violation of an undecidable rule does not necessarily indicate that there are no violations in the code being analysed.

Some examples of undecidable rules are:

- Rule 12.2: depends on the value of the right-hand operand of a shift operator;
- Rule 2.1: depends on knowing whether execution never reaches a certain point.

As indicated in Section 5.3.3, a process should be developed for analysing the results of static analysis and recording the outcome. Particular attention should be paid to the process for analysing any output that relates to undecidable rules.

6.6 Scope of analysis

Each rule is classified according to the amount of code that needs to be checked in order to detect violations. As for decidability, the concept of analysis scope is not applied to directives.

The analysis scopes that may be applied to rules are "Single Translation Unit" and "System".

If a rule is classified as capable of being checked on a "Single Translation Unit" basis then it is possible to detect all violations within a project by checking each translation unit independently. For example, the presence of *switch* statements that do not contain *default* labels (Rule 16.4) within one translation unit has no effect on whether other translation units contain such *switch* statements.

If a rule is classified as needing to be checked on a "System" basis, then identifying violations of a rule within a translation unit requires checking more than the translation unit in question. Rules that are classified as "System" are best checked by analysing all the source code, although it will be possible to identify some violations when checking a subset of the whole source. For example, if a project has two translation units *A* and *B*, it is possible to check that all declarations and definitions of an object within each translation unit use the same type names and qualifiers (Rule 8.3). However, this does not guarantee that the declarations and definitions in *A* use the same type names and qualifiers as those in *B*. All of the source code that will be compiled and linked into the executable therefore needs to be checked to guarantee compliance with this rule.

All undecidable rules need to be checked on a "System" basis because, in the general case, information about the behaviour of other translation units will be needed. For example, whether or not the value of the automatic object `x` is set before `x` is used (Rule 9.1) in the function `g`, below, will depend on the behaviour of the function `f` which is defined in another translation unit:

```
extern void f ( uint16_t *p );

uint16_t  y;

void g ( void )
{
  uint16_t  x;  /* x is not given a value                                */

  f ( &x );     /* f might modify the object pointed to by its parameter */
  y = x;        /* x may or may not be unset                             */
}
```

6.7 Multi-organization projects

Projects may involve code from various organizations, for example:

- The Standard Library code from the compiler implementer;
- Low-level driver code from a device vendor;
- Operating system and high-level driver code from a specialist supplier;
- Application code which may be shared between collaborating organizations according to their expertise.

The Standard Library code is likely to be concerned with efficiency of execution. It may rely on implementation details or unspecified behaviours such as:

- Casting pointers to object types into pointers to other object types;
- The layout of stack frames, and in particular the locations of function parameters within those frames;
- Embedding assembly language statements in C.

As it is part of the implementation, and its functionality and interface is defined in the ISO C standard, The Standard Library code is not required to comply with MISRA C. Unless otherwise specified in the individual guidelines, the contents of standard *header files*, and any files that are included during processing of a standard *header file*, are not required to comply with MISRA C. However, guidelines that rely on the interface provided by standard header declarations and macros are still applicable. For example, the *essential type* rules apply to the types of arguments passed to functions specified in The Standard Library and to their results.

All other code should comply with the MISRA C guidelines to the greatest extent possible. If all the source code is available then the entire program can be checked for compliance. However, in many projects it is not possible for the developing organization to obtain all the source code. For example, collaborating organizations may share functional specifications, interface specifications and object code but are likely to protect their own intellectual property by not sharing source code. When only part of the source code is available for analysis, other techniques should be applied in order to confirm compliance. For example:

- Organizations that supply commercial code may be willing to issue a statement of MISRA C compliance;
- Organizations that are collaborating might:
  - Agree upon a common procedure and tools that each will apply to their own source code;
  - Supply each other with stub versions of source code to permit cross-organizational checks to be made.

A project should define which of the following methods it will use to deal with code supplied by other organizations:

1. In the same manner as code being developed in-house — this relies on having the source code available;
2. In the same manner as The Standard Library — this relies on the organization supplying a suitable MISRA C compliance statement;
3. Compliance checks to be performed on the contents of *header files* and their interface — this assumes that no compliance statement is available and no source code, other than *header files*, is available.

In case (3), appropriate additional verification and validation techniques should be employed prior to using the code.

Note: some process standards may include wider requirements for managing multi-organization developments, for example ISO 26262 [23] Part 8 Clause 5 “Interfaces in distributed development”.

6.8 Automatically generated code

The MISRA C guidelines are applicable to code that has been generated automatically. Responsibility for compliance lies both with the developer of the automatic code generation tool, and with the developer of the model from which code is being generated. Since there are several modelling packages, each of which may have several automatic code generators, it is not possible to allocate this

responsibility individually for each MISRA C guideline. It is expected that users of modelling packages and code generators will employ relevant guidelines such as MISRA AC GMG [17], MISRA AC SLSF [18] and MISRA AC TL [19].

The category assigned to a MISRA C guideline when applied to automatically generated code is not necessarily the same as that when applied to manually generated code. When making this distinction, it is important to note that some modelling packages permit the injection of manually generated code into a model. Such code is **not** treated as having been generated automatically when determining the applicable category of a guideline.

Appendix E identifies those MISRA C guidelines whose category as applied to automatically generated code is different from that for manually generated code. It also states the documentation requirements that are placed on automatic code generator developers.

6.9 Presentation of guidelines

The individual requirements of this document are presented in the following format:

Ident	Requirement text

[Source ref]

Category	Category
Analysis	Decidability, Scope
Applies to	Cxx

where:

- "Ident" is a unique identifier for the guideline;
- "Requirement text" is the guideline itself;
- "Source ref" indicates the primary source(s) which led to this item or group of items, where applicable. See Section 6.10 for an explanation of the significance of these references, and a key to the source materials;
- "Category" is one of "Mandatory", "Required" or "Advisory", as explained in Section 6.2;
- "Decidability" is one of "Decidable" or "Undecidable", as explained in Section 6.5;
- "Scope" is one of "System" or "Single Translation Unit", as explained in Section 6.6;
- "Cxx" is one or more of "C90" and "C99", separated by a comma, indicating which versions of the C language the guideline applies to.

Note: since directives do not have a decidability or a scope of analysis, the "Analysis" line is omitted for directives.

In addition, supporting text is provided for each item or group of related items. The text gives, where appropriate, some explanation of the underlying issues being addressed by the guideline(s), and examples of how to apply the guideline(s).

Within the supporting text, there may be a heading titled "Amplification", followed by text that provides a more precise description of the guideline. An amplification is normative; if it conflicts with the headline, the amplification takes precedence. This mechanism is convenient as it allows a complicated concept to be conveyed using a short headline.

Within the supporting text, there may be a heading titled "Exception", followed by text that describes situations in which the rule does not apply. The use of exceptions permits the description of some

guidelines to be simplified. It is important to note that an exception is a situation in which the guideline does not apply. Code that complies with a guideline by virtue of an exception does not require a deviation.

The supporting text is not intended as a tutorial in the relevant language feature, as the reader is assumed to have a working knowledge of the language. Further information on the language features can be obtained by consulting the relevant section of the language standard or other C language reference books. Where a source reference is given for one or more of the "Portability Issues" listed in The Standard, then the original issue raised in it may provide additional help in understanding the guideline.

Within the guidelines, and their supporting text, the following font styles are used to represent C keywords and C code:

- C keywords appear in *italic text*;
- Items defined in the Glossary appear in *italic text*;
- C code appears in a `monospaced font`, either within other text;
- Or

```
as separate code fragments.
```

Note: where code is quoted the fragments may be incomplete (for example an *if* statement without its body). This is for the sake of brevity.

In code fragments the following *typedef*'d types have been assumed (this is to comply with Dir 4.6):

```
uint8_t           /* unsigned 8-bit integer  */
uint16_t          /* unsigned 16-bit integer */
uint32_t          /* unsigned 32-bit integer */
int8_t            /* signed 8-bit integer    */
int16_t           /* signed 16-bit integer   */
int32_t           /* signed 32-bit integer   */
float32_t         /* 32-bit floating-point   */
float64_t         /* 64-bit floating-point   */
```

Non-specific object names are constructed to give an indication of the type. For example:

```
uint8_t   u8a;    /* 8-bit unsigned integer  */
int16_t   s16b;   /* 16-bit signed integer   */
int32_t   s32c;   /* 32-bit signed integer   */
bool_t    bla;    /* essentially Boolean     */
enum atag ena;    /* enumerated type         */
char      chb;    /* character               */
float32_t f32a;   /* 32-bit floating-point   */
float64_t f64b;   /* 64-bit floating-point   */
```

6.10 Understanding the source references

Where a guideline originates from one or more published sources these are indicated in square brackets after the guideline. This serves two purposes. Firstly the specific sources may be consulted by a reader wishing to gain a fuller understanding of the rationale behind the guideline (for example when considering a request for a deviation). Secondly, with regard to Portability Issues described in the ISO C standard, the form of the source gives extra information about the nature of the problem.

Rules which do not have a source reference may have originated from a contributing company's in-house standard, or have been suggested by a reviewer, or be widely accepted good practice.

6.10.1 ISO C portability issue references

The ISO C Portability Issues are described in subsections of Annexes in the relevant standard. It is important to note that for both C90 and C99, the numbering of references is derived from the original standard and not from any later version that incorporates corrections and amendments. The reason for this decision is partly historical, in that MISRA C:2004 used this scheme, and partly practical, in that there is no officially-available revision of the C99 Standard that incorporates its Technical Corrigenda although there is a draft [12].

The subsections of the relevant standard corresponding to this kind of reference are:

Reference	Annex G of ISO 9899:1990	Annex J of ISO 9899:1999
Unspecified	G.1	J.1
Undefined	G.2	J.2
Implementation	G.3	J.3
Locale	G.4	J.4

Where text follows the reference, it has the following meanings:

- Annex G of [2] references: the text identifies the number of an item or items in the relevant section of the Annex, numbered from the beginning of that section. So for example [Locale 2] refers to the second item in section G.4 of The Standard and [Undefined 3, 5] refers to the third and fifth items in Section G.2 of The Standard.
- Annex J of [8] references: for sections J.1, J.2 and J.4, the same interpretation as above applies. For section J.3, the text identifies the subsection of J.3, followed by a parenthesized item number or numbers. So for example [Unspecified 6] refers to the sixth item in Section J.1 of The Standard and [Implementation J3.4(2, 5)] refers to the second and fifth items in Section J.3.4 of The Standard.

Where a guideline is based on issues from Annex G (C90) or Annex J (C99) of The Standard, it is helpful for the reader to understand the distinction between the Unspecified, Undefined, Implementation and Locale references. These are explained briefly here, and further information can be found in Hatton [3].

Unspecified

These are language constructs that must compile successfully, but in which the compiler writer has some freedom as to what the construct does. An example of this is the "order of evaluation" described in Rule 13.2.

While the use of some unspecified behaviour is unavoidable, it is unwise to assume that the compiler produces object code that behaves in a particular way; the compiler need not even perform consistently across all possible constructs.

MISRA C identifies specific instances of unspecified behaviour which are liable to result in unsafe behaviour.

Undefined

These are essentially programming errors, but for which the compiler writer is not obliged to provide error messages. Examples are invalid parameters to functions, or functions whose arguments do not match the defined parameters.

These are particularly important from a safety point of view, as they represent programming errors which may not necessarily be trapped by the compiler.

Implementation

These are similar to the "unspecified" issues, the main difference being that the compiler writer must take a consistent approach and document it. In other words the functionality can vary from one compiler to another, making code non-portable, but on any one compiler the behaviour should be well-defined. An example of this is the behaviour of the integer division and remainder operators, `/` and `%`, when applied to one positive and one negative integer.

The implementation-defined behaviours tend to be less critical from a safety point of view, provided the compiler writer has fully documented the intended approach and then implemented it consistently. It is advisable to avoid these issues where possible.

Locale

The locale-specific behaviours form a small set of features which may vary with international requirements. An example of this is the facility to represent a decimal point by the "," character instead of the "." character. No issues arising from this source are addressed in this document.

6.10.2 Other references

References other than the ISO C Portability Issues are taken from the following sources:

Reference	Source
MISRA Guidelines	The MISRA Guidelines [15]
Koenig	"C Traps and Pitfalls", Koenig [31]
IEC 61508	IEC 61508:2010 [32]
ISO 26262	ISO 26262:2011 [23]
DO-178C	DO-178C [24]

Unless stated otherwise, the text following a reference gives the relevant page number.

7 Directives

7.1 The implementation

Dir 1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood

C90 [Annex G.3], C99 [Annex J.3]

Category Required

Applies to C90, C99

Amplification

Appendix G of this document lists, for both C90 and C99, those implementation-defined behaviours that:

- Are considered to have the potential to cause unexpected program operation, and
- May be present in a program even if it complies with all the other MISRA C guidelines.

All of these implementation-defined behaviours on which the program output depends must be:

- Documented, and
- Understood by developers.

Note: a conforming implementation is required to document its treatment of all implementation-defined behaviour. The developer of an implementation should be consulted if any documentation is missing.

Rationale

It is important to know that the output of a program was intentional and was not produced by chance.

Some of the more common implementation-defined behaviours on which safety-related embedded software is likely to depend are described below.

Core behaviour

The fundamental implementation-defined behaviours, which are likely to be required by most programs, include:

- How to identify a diagnostic message produced during translation;
- The type of the function *main*, commonly declared as `void main (void)` in freestanding implementations;
- The number of significant characters in identifiers — needed to configure an analysis tool for Rule 5.2;
- The source and execution character sets;
- The sizes of the integer types;
- How a *#include*'d name is mapped onto a file name and located in the host file system.

Extensions

Extensions are often used in embedded systems to provide access to peripherals and to place objects into regions of memory with special properties such as Flash EEPROM or fast-access RAM. A conforming implementation is permitted to provide extensions provided that they do not alter the meaning of any strictly conforming program.

Some C90 implementations may provide extensions which implement a subset of the C99 features.

Some of the methods by which an implementation can provide extensions are:

- The *#pragma* preprocessing directive or the *_Pragma* operator (C99 only);
- New keywords.

The Standard Library

Some aspects of The Standard Library implementation that may be important are:

- The values assigned to *errno* when certain Standard Library functions are used;
- The implementation of clock and time functions;
- The characteristics of the file system.

The Application Binary Interface

It is sometimes necessary to interface C code with assembly language, for example to improve execution speed in those places where it is critical. It might also be necessary to interface code produced by different compilers, possibly for different languages.

The Application Binary Interface (ABI) for a compiler provides the information necessary to perform this task, including some of the implementation-defined behaviours. It typically specifies:

- How function parameters are passed in registers and on the stack;
- How function values are returned;
- Which registers must be preserved by a function;
- How objects with automatic storage duration are allocated to stack frames;
- Alignment requirements for each data type;
- How structures are laid out and how bit-fields are allocated to storage units.

Some processors have a standard ABI which will be used by all implementations. Where no standard ABI exists, an implementation will provide its own.

Integer division

In C90, signed integer division or remainder operations in which either operand has a negative value may either round downwards or towards zero. In C99, rounding is guaranteed to be towards zero.

Floating-point implementation

The implementation of floating-point types can have significant impact on program behaviour, for example:

- The range of values and the precision with which they are stored;
- The direction of rounding following floating-point operations;

- The direction of rounding when converting to narrower floating-point types or to integer types;
- The behaviour in the presence of underflow, overflow and NaNs;
- The behaviour of library functions in the event of domain and range errors.

See also

Rule 5.1, Rule 5.2

7.2 Compilation and build

Dir 2.1	All source files shall compile without any compilation errors

Category Required

Applies to C90, C99

Rationale

A conforming compiler is permitted to produce an object module despite the presence of compilation errors. However, execution of the resulting program may produce unexpected behaviour.

See Section 5.3.1 for further guidance on controlling and analysing compiler messages.

See also

Rule 1.1

7.3 Requirements traceability

Dir 3.1	All code shall be traceable to documented requirements

[DO-178C Section 6.4.4.3.d]

Category Required

Applies to C90, C99

Rationale

Functionality that is not needed to meet the project requirements gives rise to unnecessary paths. It is possible that the developers of the software are not aware of the wider implications that might arise from this additional functionality. For example, developers might add code that toggles the state of a processor output pin every time a particular point in the program is reached. This might be very useful during development for measuring timing or for triggering an emulator or logic analyser. However, even though the pin in question might appear to be unused because it is not mentioned in the software requirements specification, it might be connected to an actuator in the target controller, resulting in unwanted external effects.

The method by which code is traced back to documented requirements shall be determined by the project. One method of achieving traceability is to review the code against the corresponding design documents which have in turn been reviewed against the requirements.

Note: there should not be any conflict between this guideline and the provision of a protective coding strategy as the latter should be part of the requirements in a critical system.

7.4 Code design

Dir 4.1 Run-time failures shall be minimized

C90 [Undefined 15, 19, 26, 30, 31, 32, 94]
C99 [Undefined 15, 16, 33, 40, 43–45, 48, 49, 113]

Category Required

Applies to C90, C99

Rationale

The C language was designed to provide very limited built-in run-time checking. While this approach allows generation of compact and fast executable code, it places the burden of run-time checking on the programmer. In order to achieve the desired level of robustness, it is therefore important that programmers carefully consider adding dynamic checks wherever there is potential for run-time errors to occur.

It is sometimes possible to demonstrate that the values of operands preclude the possibility of a run-time error during evaluation of an expression. In such cases, a dynamic check is not required provided that the argument supporting its omission is documented. Any such documentation should include the assumptions on which the argument depends. This information can be used during subsequent modifications of the code to ensure that the argument remains valid.

The techniques that will be employed to minimize run-time failures should be planned and documented, for example in design standards, test plans, static analysis configuration files and code review checklists. The nature of these techniques may depend on the integrity requirements of the project. See Section 5.2 for further details.

Note: the presence of a run-time error indicates a violation of Rule 1.3.

The following notes give some guidance on areas where consideration needs to be given to the provision of dynamic checks.

- arithmetic errors This includes errors occurring in the evaluation of expressions, such as overflow, underflow, divide by zero or loss of significant bits through shifting. In considering integer overflow, note that unsigned integer calculations do not strictly overflow but wrap around producing defined, but possibly unexpected, values. Careful consideration should be given to the ranges of values and order of operation in arithmetic expressions, for example:

```
float32_t f1 = 1E38f;
float32_t f2 = 10.0f;
float32_t f3 = 0.1f;
float32_t f4 = ( f1 * f2 ) * f3;   /* (f1 * f2) will overflow        */
float32_t f5 = f1 * ( f2 * f3 );   /* no overflow because (f2 * f3)
                                    * is (approximately) 1          */

if ( ( f3 >= 0.0f ) && ( f3 <= 1.0f ) )
{
  /*
   * no overflow because f3 is known to be in range 0..1 so the
   * result of the multiplication will fit in type float32_t
   */
  f4 = f3 * 100.0f;
}
```

- pointer arithmetic Ensure that when an address is calculated dynamically the computed address is reasonable and points somewhere meaningful. In particular it should be ensured that if a pointer points within an array, then when the pointer has been incremented or otherwise altered it still points within the same array. See restrictions on pointer arithmetic — Rule 18.1, Rule 18.2 and Rule 18.3.

- **array bound errors** Ensure that array indices are within the bounds of the array size before using them to index the array — Rule 18.1.
- **function parameters** The validity of arguments should be checked prior to passing them to library functions — Dir 4.11.
- **pointer dereferencing** Unless a pointer is already known to be non-`NULL`, a run-time check should be made before dereferencing that pointer. Once a check has been made, it is relatively straightforward within a single function to reason about whether the pointer may have changed and whether another check is therefore required. It is much more difficult to reason across function boundaries, especially when calling functions defined in other source files or libraries.

```
/*
 * Given a pointer to a message, check the message header and return
 * a pointer to the body of the message or NULL if the message is
 * invalid.
 */
const char *msg_body ( const char *msg )
{
  const char *body = NULL;

  if ( msg != NULL )
  {
    if ( msg_header_valid ( msg ) )
    {
      body = &msg[ MSG_HEADER_SIZE ];
    }
  }
  return body;
}

      char  msg_buffer[ MAX_MSG_SIZE ];
const char *payload;

payload = msg_body ( msg_buffer );

/* Check if there is a payload */
if ( payload != NULL )
{
  /* Process the payload */
}
```

- **dynamic memory** If dynamic memory allocation is being performed, it is essential to check that each allocation succeeds and that an appropriate degradation or recovery strategy is designed and tested.

See also

Dir 4.11, Dir 4.12, Rule 1.3, Rule 18.1, Rule 18.2, Rule 18.3

Dir 4.2 All usage of assembly language should be documented

Category Advisory

Applies to C90, C99

Amplification

The rationale for the use of the assembly language and the mechanism for interfacing between C and the assembly language should be documented.

Rationale

Assembly language code is implementation-defined and therefore not portable.

Dir 4.3 Assembly language shall be encapsulated and isolated

Category Required

Applies to C90, C99

Amplification

Where assembly language instructions are used they shall be encapsulated and isolated in:

- Assembly language functions;
- C functions (*inline functions* preferred for C99);
- C macros.

Rationale

For reasons of efficiency it is sometimes necessary to embed simple assembly language instructions in-line, for example to enable and disable interrupts. If this is necessary, then it is recommended that it be achieved by using macros, or for C99, *inline functions*.

Encapsulating assembly language is beneficial because:

- It improves readability;
- The name, and documentation, of the encapsulating macro or function makes the intent of the assembly language clear;
- All uses of assembly language for a given purpose can share the same encapsulation which improves maintainability;
- The assembly language can easily be substituted for a different target or for purposes of static analysis.

Note: the use of in-line assembly language is an extension to standard C, and therefore violates Rule 1.2.

Example

```
#define NOP asm("    NOP")
```

Dir 4.4 Sections of code should not be "commented out"

Category Advisory

Applies to C90, C99

Amplification

This rule applies to both `//` and `/* ... */` styles of comment.

Rationale

Where it is required for sections of source code not to be compiled then this should be achieved by use of conditional compilation (e.g. *#if* or *#ifdef* constructs with a comment). Using start and end comment markers for this purpose is dangerous because C does not support nested comments, and any comments already existing in the section of code would change the effect.

See also

Rule 3.1, Rule 3.2

Dir 4.5 Identifiers in the same name space with overlapping visibility should be typographically unambiguous

Category Advisory

Applies to C90, C99

Amplification

The definition of the term "unambiguous" should be determined for a project taking into account the alphabet and language in which the source code is being written.

For the Latin alphabet as used in English words, it is advised as a minimum that identifiers should not differ by any combination of:

- The interchange of a lowercase character with its uppercase equivalent;
- The presence or absence of the underscore character;
- The interchange of the letter "O", and the digit "0";
- The interchange of the letter "I", and the digit "1";
- The interchange of the letter "I", and the letter "l" (el);
- The interchange of the letter "l" (el), and the digit "1";
- The interchange of the letter "S", and the digit "5";
- The interchange of the letter "Z", and the digit "2";
- The interchange of the letter "n", and the letter "h";
- The interchange of the letter "B", and the digit "8";
- The interchange of the letter sequence "rn" ("r" followed by "n"), and the letter "m";

Rationale

Depending upon the font used to display the character set, it is possible for certain glyphs to appear the same, even though the characters are different. This may lead to the developer confusing an identifier with another one.

Example

The following examples assume the interpretation suggested in the Amplification for the Latin alphabet and English language.

```
int32_t   id1_a_b_c;
int32_t   id1_abc;      /* Non-compliant */

int32_t   id2_abc;
int32_t   id2_ABC;      /* Non-compliant */

int32_t   id3_a_bc;
int32_t   id3_ab_c;     /* Non-compliant */

int32_t   id4_I;
int32_t   id4_1;        /* Non-compliant */

int32_t   id5_Z;
int32_t   id5_2;        /* Non-compliant */

int32_t   id6_O;
int32_t   id6_0;        /* Non-compliant */

int32_t   id7_B;
int32_t   id7_8;        /* Non-compliant */

int32_t   id8_rn;
int32_t   id8_m;        /* Non-compliant */

int32_t   id9_rn;
struct
{
  int32_t id9_m;        /* Compliant     */
};
```

Dir 4.6 *typedefs* that indicate size and signedness should be used in place of the basic numerical types

Category Advisory

Applies to C90, C99

Amplification

The basic numerical types of *char*, *short*, *int*, *long*, *long long* (C99), *float*, *double* and *long double* should not be used, but specific-length *typedef*s should be used.

For C99, the types provided by `<stdint.h>` should be used. For C90, equivalent types should be defined and used.

A type must not be defined with a specific length unless the implemented type is actually of that length.

It is not necessary to use *typedef*s in the declaration of bit-fields.

For example, on a 32-bit C90 implementation the following definitions might be suitable:

```
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;
```

Rationale

In situations where the amount of memory being allocated is important, using specific-length types makes it clear how much storage is being reserved for each object.

Adherence to this guideline does **not** guarantee portability because the size of the *int* type may determine whether or not an expression is subject to integer promotion. For example, an expression with type `int16_t` will not be promoted if *int* is implemented using 16 bits but will be promoted if *int* is implemented using 32 bits. This is discussed in more detail in the section on integer promotion in Appendix C.

Note: defining a specific-length type whose size is **not** the same as the implemented type is counter-productive both in terms of storage requirements and in terms of portability. Care should be taken to avoid defining types with the wrong size.

If abstract types are defined in terms of a specific-length type then it is not necessary, and may even be undesirable, for those abstract types to specify the size or sign. For example, the following code defines an abstract type representing mass in kilograms but does not indicate its size or sign:

```
typedef uint16_t mass_kg_t;
```

It might be desirable not to apply this guideline when interfacing with The Standard Library or code outside the project's control.

Exception

1. The basic numerical types may be used in a *typedef* to define a specific-length type.
2. For function `main`, an *int* may be used rather than the *typedef*s as a return type. Therefore `int main (void)` is permitted.
3. For function `main` an *int* may be used rather than the *typedef*s for the input parameter `argc`.
4. For function `main` a *char* may be used rather than the *typedef*s for the input parameter `argv`.

Therefore `int main( int argc, char *argv[] )` is permitted (C99 Section 5.1.2.2.1).

Example

```
/* Non-compliant - int used to define an object                    */
int x = 0;

/* Compliant     - int used to define specific-length type         */
typedef int SINT_16;
```

```
/* Non-compliant - no sign or size specified                          */
typedef int speed_t;

/* Compliant     - further abstraction does not need specific length */
typedef int16_t torque_t;
```

Dir 4.7	If a function returns error information, then that error information shall be tested

Category Required

Applies to C90, C99

Amplification

The list of functions that are deemed to return error information shall be determined by the project.

The error information returned by a function shall be tested in a meaningful manner.

Rationale

A function (whether it is part of The Standard Library, a third party library or a user defined function) may be deemed to provide some means of indicating the occurrence of an error. This may be via an error flag, some special return value or some other means. Whenever such a mechanism is provided by a function the calling program shall check for the indication of an error as soon as the function returns.

However, note that the checking of input values to functions is considered a more robust means of error prevention than trying to detect errors after the function has completed (see Dir 4.11).

Exception

If it can be shown, for example by checking arguments, that a function cannot return an error indication then there is no need to perform a check.

See also

Dir 4.11, Rule 17.7

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

Category Advisory

Applies to C90, C99

Amplification

The implementation of an object should be hidden by means of a pointer to an incomplete type.

Rationale

If a pointer to a structure or union is never dereferenced, then the implementation details of the object are not needed and its contents should be protected from unintentional changes.

Hiding the implementation details creates an *opaque type* which may be referenced via a pointer but whose contents may not be accessed.

Example

```
/* Opaque.h */

#ifndef OPAQUE_H
#define OPAQUE_H

typedef struct OpaqueType *pOpaqueType;

#endif

/* Opaque.c */

#include "Opaque.h"

struct OpaqueType
{
  /* Object implementation */
};

/* UseOpaque.c */

#include "Opaque.h"

void f ( void )
{
  pOpaqueType pObject;

  pObject = GetObject ( );   /* Get a handle to an OpaqueType object */

  UseObject ( pObject );     /* Use it...                            */
}
```

Dir 4.9 A function should be used in preference to a *function-like macro* where they are interchangeable

[Koenig 78–81]

Category Advisory

Applies to C90, C99

Amplification

This guideline applies only where a function is permitted by the syntax and *constraints* of the language standard.

Rationale

In most circumstances, functions should be used instead of macros. Functions perform argument type-checking and evaluate their arguments once, thus avoiding problems with potential multiple *side effects*. In many debugging systems, it is easier to step through execution of a function than a macro. Nonetheless, macros may be useful in some circumstances.

Some of the factors that should be considered when deciding whether to use a function or a macro are:

- The benefits of function argument and result type-checking;
- The availability of *inline functions* in C99, although note that the extent to which *inline* is acted upon is implementation-defined;

- The trade-off between code size and execution speed;
- Whether the possibility for compile-time evaluation is important: macros with constant arguments are more likely to be evaluated at compile-time than corresponding function calls;
- Whether the arguments would be valid for a function: macro arguments are textual whereas function arguments are expressions;
- Ease of understanding and maintainability.

Example

The following example is compliant. The *function-like macro* cannot be replaced with a function because it has a C operator as an argument:

```
#define EVAL_BINOP( OP, L, R ) ( ( L ) OP ( R ) )

uint32_t  x = EVAL_BINOP ( +, 1, 2 );
```

In the following example, the use of a macro to initialize an object with static storage duration is compliant because a function call is not permitted here.

```
#define DIV2(X) ( ( X ) / 2 )

void f ( void )
{
  static uint16_t x = DIV2 ( 10 );   /* Compliant - call not permitted */
         uint16_t y = DIV2 ( 10 );   /* Non-compliant - call permitted */
}
```

See also

Rule 13.2, Rule 20.7

Dir 4.10	Precautions shall be taken in order to prevent the contents of a *header file* being included more than once
Category	Required
Applies to	C90, C99

Rationale

When a translation unit contains a complex hierarchy of nested *header files*, it is possible for a particular *header file* to be included more than once. This can be, at best, a source of confusion. If this multiple inclusion leads to multiple or conflicting definitions, then this can result in undefined or erroneous behaviour.

Example

```
/* file.h */
#ifndef FILE_H
/* Non-compliant - does not #define FILE_H */
#endif
```

In order to facilitate checking, the contents of the header should be protected from being included more than once using one of the following two forms:

```
<start-of-file>
#if !defined ( identifier )
#define identifier
  /* Contents of file */
#endif
<end-of-file>

<start-of-file>
#ifndef identifier
#define identifier
  /* Contents of file */
#endif
<end-of-file>
```

Note: the identifier used to test and record whether a given *header file* has already been included shall be unique across all *header files* in the project.

Note: comments are permitted anywhere within these forms.

Dir 4.11 The validity of values passed to library functions shall be checked

C90 [Undefined 60, 63, 96; Implementation 45–47]
C99 [Unspecified 30, 31, 44, 48–50;
Undefined 102, 103, 107, 112, 180, 181, 183, 187, 189;
Implementation J.3(8–11)]

Category Required

Applies to C90, C99

Amplification

The nature and organization of the project will determine which libraries, and functions within those libraries, should be subject to this directive.

Rationale

Many functions in The Standard Library are not required by The Standard to check the validity of parameters passed to them. Even where checking is required by The Standard, or where compiler writers claim to check parameters, there is no guarantee that adequate checking will take place.

Similarly, the interface description for functions in other libraries may not specify the checks performed by those functions. There is also a risk that the specified checks are not performed adequately.

The programmer shall provide appropriate checks of input values for all library functions which have a restricted input domain (The Standard Library, third-party libraries, and in-house libraries).

Examples of functions from The Standard Library that have a restricted domain and need checking are:

- Many of the maths functions in `<math.h>`, for example:
  - Negative numbers must not be passed to the *sqrt* or *log* functions;
  - The second parameter of *fmod* should not be zero;
- Some implementations can produce unexpected results when the function *toupper* is passed an argument which is not a lowercase letter (and similarly for *tolower*);
- The character testing functions in `<ctype.h>` exhibit undefined behaviour if passed invalid values;
- The *abs* function applied to the most negative integer gives undefined behaviour.

Although most of the math library functions in `<math.h>` define allowed input domains, the values they return when a domain error occurs may vary from one compiler to another. Therefore pre-checking the validity of the input values is particularly important for these functions.

The programmer should identify any domain constraints which should sensibly apply to a function being used (which may or may not be documented in the interface description), and provide appropriate checks that the input value(s) lies within this domain. Of course the value may be restricted further, if required, by knowledge of what the parameter represents and what constitutes a sensible range of values for the parameter.

There are a number of ways in which the requirements of this guideline might be satisfied, including the following:

- Check the values before calling the function;
- Check the values in the called library function — this is particularly applicable for in-house designed libraries, though it could apply to bought-in libraries if the supplier can demonstrate that they have built in the checks;
- Produce "wrapped" versions of functions that perform the checks then call the original function;
- Demonstrate statically that the input parameters can never take invalid values.

See also

Dir 4.1, Dir 4.7

Dir 4.12 Dynamic memory allocation shall not be used

Category	Required
Applies to	C90, C99

Amplification

This rule applies to all dynamic memory allocation packages including:

- Those provided by The Standard Library;
- Third-party packages.

Rationale

The Standard Library's dynamic memory allocation and deallocation routines can lead to undefined behaviour as described in Rule 21.3. Any other dynamic memory allocation system is likely to exhibit undefined behaviours that are similar to those of The Standard Library.

The specification of third-party routines shall be checked to ensure that dynamic memory allocation is not being used inadvertently.

If a decision is made to use dynamic memory, care shall be taken to ensure that the software behaves in a predictable manner. For example, there is a risk that:

- Insufficient memory may be available to satisfy a request — care must be taken to ensure that there is a safe and appropriate response to an allocation failure;
- There is a high variance in the execution time required to perform allocation or deallocation depending on the pattern of usage and resulting degree of fragmentation.

Example

For convenience, these examples are based around use of The Standard Library's dynamic memory functions as their interfaces are well-known.

In this example, the behaviour is undefined following the first call to *free* because the value of the pointer `p` becomes indeterminate. Although the value stored in the pointer is unchanged following the call to *free*, it is possible, on some targets, that the memory to which it points no longer exists and the act of copying that pointer could cause a memory exception.

```
#include <stdlib.h>

void f ( void )
{
  char     *p = ( char * ) malloc ( 10 );
  char     *q;

  free ( p );
  q = p;          /* Undefined behaviour - value of p is indeterminate    */


  p = ( char * ) malloc ( 20 );
  free ( p );
  p = NULL;       /* Assigning NULL to freed pointer makes it determinate */
}
```

See also

Dir 4.1, Rule 18.7, Rule 21.3, Rule 22.1, Rule 22.2

Dir 4.13 Functions which are designed to provide operations on a resource should be called in an appropriate sequence

Category Advisory

Applies to C90, C99

Amplification

A set of functions providing operations on a resource typically has three kinds of operation:

1. Allocation of the resource, e.g. opening a file;
2. Deallocation of the resource, e.g. closing a file;
3. Other operations, e.g. reading from a file.

For each such set of functions, all uses of its operations should occur in an appropriate sequence.

Rationale

Static analyser tools are capable of providing path analysis checks that can identify paths through a program that result in the deallocation function of a sequence not being called. In order to maximize the benefits of such automated checks, developers are therefore encouraged to enable these checks by designing and declaring sets of balanced functions to the static analyser.

Example

```
/* These functions are intended to be paired */
extern mutex_t mutex_lock   ( void );
extern void    mutex_unlock ( mutex_t m );

extern int16_t x;

void f ( void )
{
  mutex_t m = mutex_lock ( );

  if ( x > 0 )
  {
    mutex_unlock ( m );
  }
  else
  {
    /* Mutex not unlocked on this path */
  }
}
```

See also

Rule 22.1, Rule 22.2, Rule 22.6

8 Rules

8.1 A standard C environment

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

[MISRA Guidelines Table 3], [IEC 61508-7: Table C.1], [ISO 26262-6: Table 1]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The program shall use only those features of the C language and its library that are specified in the chosen version of The Standard (see Section 3.1).

The Standard permits implementations to provide language extensions and the use of such extensions is permitted by this rule.

Except when making use of a language extension, a program shall not:

- Contain any violations of the language syntax described in The Standard;
- Contain any violations of the *constraints* imposed by The Standard.

A program shall not exceed the translation limits imposed by the implementation. The minimum translation limits are specified by The Standard but an implementation may provide higher limits.

Note: a conforming implementation generates a diagnostic for syntax and *constraint* violations but be aware that:

- The diagnostic need not necessarily be an error but could, for example, be a warning;
- The program may be translated and an executable generated despite the presence of a syntax or *constraint* violation;

Note: a conforming implementation does not need to generate a diagnostic when a translation limit is exceeded; an executable may be generated but it is not guaranteed to execute correctly.

Rationale

Problems associated with language features that are outside the supported versions of ISO/IEC 9899 have not been considered during development of these guidelines.

There is anecdotal evidence of some non-conforming implementations failing to diagnose *constraint* violations, for example in [38] p135, example 2 entitled "Error of writing into the const area".

Example

Some C90 compilers provide support for *inline functions* using the *__inline* keyword. A C90 program that uses *__inline* will be compliant with this rule provided that it is intended to be translated using such a compiler.

Many compilers for embedded targets provide additional keywords that qualify object types with attributes of the memory area in which the object is located, for example:

- *__zpage* — the object can be accessed using a short instruction
- *__near* — a pointer to the object can be held in 16 bits
- *__far* — a pointer to the object can be held in 24 bits

A program using these additional keywords will be compliant with this rule provided that the compiler supports those keywords as a language extension.

See also

Dir 2.1, Rule 1.2

Rule 1.2	Language extensions should not be used
Category	Advisory
Analysis	Undecidable, Single Translation Unit
Applies to	C90, C99

Rationale

A program that relies on language extensions may be less portable than one that does not. Although The Standard requires that a conforming implementation document any extensions that it provides to the language, there is a risk that this documentation might not provide a full description of the behaviour in all circumstances.

If this rule is not applied, the decision to use each language extension should be justified in the project's design documentation. The methods by which valid use of each extension will be assured, for example checking the compiler and its diagnostics, should also be documented.

It is recognized that it is necessary to use language extensions in embedded systems. The Standard requires that an extension does not alter the behaviour of any strictly conforming program. For example, a compiler might implement, as an extension, full evaluation of binary logical operators even though The Standard specifies that evaluation stops as soon as the result can be determined. Such an extension does not conform to The Standard because *side effects* in the right-hand operand of a logical AND operator would always occur, giving rise to a different behaviour.

See also

Rule 1.1

Rule 1.3	There shall be no occurrence of undefined or critical unspecified behaviour
Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

Some undefined and unspecified behaviours are dealt with by specific rules. This rule prevents all other undefined and critical unspecified behaviours. Appendix H lists the undefined behaviours and those unspecified behaviours that are considered critical.

Rationale

Any program that gives rise to undefined or unspecified behaviour may not behave in the expected manner. In many cases, the effect is to make the program non-portable but it is also possible for more serious problems to occur. For example, undefined behaviour might affect the result of a computation. If correct operation of the software is dependent on this computation then system safety might be compromised. The problem is particularly difficult to detect if the undefined behaviour only manifests itself on rare occasions.

Many of the MISRA C guidelines have been designed to avoid certain undefined and unspecified behaviours. For example, compliance with all of Rule 11.4, Rule 11.8 and Rule 19.2 ensures that it is not possible in C to create a non-*const* qualified pointer to an object declared with a *const*-qualified type. This avoids C90 [Undefined 39] and C99 [Undefined 61]. However, other behaviours are not covered by specific guidelines for example because:

- It is unlikely that the behaviour will be encountered;
- There is no practical guidance that can be given other than the obvious statement that the behaviour should be avoided.

Instead of introducing a guideline for each undefined and critical unspecified behaviour, the MISRA C Guidelines directly address those that are considered most important and most likely to occur in practice. Those behaviours that do not have specific guidelines are all covered together by this single rule. Appendix H lists all undefined and critical unspecified behaviours, along with the MISRA C guidelines that prevent their occurrence. It therefore indicates which behaviours are expected to be prevented by this rule and which behaviours are covered by other rules.

Note: some implementations may provide well-defined behaviour for some of the undefined and unspecified behaviours listed in The Standard. If such well-defined behaviours are relied upon, including by means of a language extension, it will be necessary to deviate this rule in respect of those behaviours.

See also

Dir 4.1

8.2 Unused code

Rule 2.1	A project shall not contain *unreachable code*

[IEC 61508-7 Section C.5.9], [DO-178C Section 6.4.4.3.c]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

Provided that a program does not exhibit any undefined behaviour, *unreachable code* cannot be executed and cannot have any effect on the program's outputs. The presence of *unreachable code* may therefore indicate an error in the program's logic.

A compiler is permitted to remove any *unreachable code* although it does not have to do so. *Unreachable code* that is **not** removed by the compiler wastes resources, for example:

- It occupies space in the target machine's memory;
- Its presence may cause a compiler to select longer, slower jump instructions when transferring control around the *unreachable code*;
- Within a loop, it might prevent the entire loop from residing in an instruction cache.

It is sometimes desirable to insert code that appears to be unreachable in order to handle exceptional cases. For example, in a *switch* statement in which every possible value of the controlling expression is covered by an explicit *case*, a *default* clause shall be present according to Rule 16.4. The purpose of the *default* clause is to trap a value that should not normally occur but that may have been generated as a result of:

- Undefined behaviour present in the program;
- A failure of the processor hardware.

If a compiler can prove that a *default* clause is unreachable, it may remove it, thereby eliminating the defensive action. On the assumption that the defensive action is important, it will be necessary either to demonstrate that the compiler does not eliminate the code despite it being unreachable, or to take steps to make the defensive code reachable. The former course of action requires a deviation against this rule, probably with a review of the object code or unit testing being used to support such a deviation. The latter course of action can usually be achieved by means of a *volatile* access. For example, a compiler might determine that the range of values held by `x` is covered by the *case* clauses in a *switch* statement such as:

```
uint16_t  x;

switch ( x )
```

By forcing `x` to be accessed by means of a *volatile* qualified *lvalue*, the compiler has to assume that the controlling expression could take any value:

```
switch ( *( volatile uint16_t * ) &x )
```

Note: code that has been conditionally excluded by pre-processor directives is not subject to this rule as it is not presented to the later phases of translation.

Example

```
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               */
}
```

See also

Rule 14.3, Rule 16.4

Rule 2.2 There shall be no *dead code*

[IEC 61508-7 Section C.5.10], [ISO 26262-6 Section 9.4.5], [DO-178C Section 6.4.4.3.c]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

Any operation that is executed but whose removal would not affect program behaviour constitutes *dead code*. Operations that are introduced by language extensions are assumed always to have an effect on program behaviour.

Note: The behaviour of an embedded system is often determined not just by the nature of its actions, but also by the time at which they occur.

Note: unreachable code is not *dead code* as it cannot be executed.

Rationale

The presence of *dead code* may be indicative of an error in the program's logic. Since *dead code* may be removed by a compiler, its presence may cause confusion.

Exception

A cast to *void* is assumed to indicate a value that is intentionally not being *used*. The cast is therefore not *dead code* itself. It is treated as using its operand which is therefore also not *dead code*.

Example

In this example, it is assumed that the object pointed to by `p` is used in other functions.

```
extern volatile uint16_t v;

extern char *p;

void f ( void )
{
  uint16_t  x;

  ( 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            */
  x = 3;            /* Non-compliant - the = operator is dead
                     *               - x is not subsequently read         */
  *p++;             /* Non-compliant - result of * operator is not used   */
  ( *p )++;         /* Compliant     - *p is incremented                  */
}
```

In the following compliant example, the *__asm* keyword is a language extension, not a function call operation, and is therefore not *dead code*:

```
__asm ( "NOP" );
```

In the following example, the function `g` does not contain *dead code*, and is not itself *dead code* because it does not contain any operations. However, the call to the function is dead because it could be removed without affecting program behaviour.

```
void g ( void )
{
  /* Compliant - there are no operations in this function */
}

void h ( void )
{
  g ( );   /* Non-compliant - the call could be removed   */
}
```

See also

Rule 17.7

Rule 2.3 A project should not contain unused type declarations

Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99

Rationale

If a type is declared but not used, then it is unclear to a reviewer if the type is redundant or it has been left unused by mistake.

Example

```
int16_t unusedtype ( void )
{
  typedef int16_t local_Type;   /* Non-compliant */

  return 67;
}
```

Rule 2.4 A project should not contain unused tag declarations

Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99

Rationale

If a tag is declared but not used, then it is unclear to a reviewer if the tag is redundant or it has been left unused by mistake.

Example

In the following example, the tag `state` is unused and the declaration could have been written without it.

```
void unusedtag ( void )
{
  enum state { S_init, S_run, S_sleep };   /* Non-compliant */
}
```

In the following example, the tag `record_t` is used only in the *typedef* of `record1_t` which is used in the rest of the translation unit whenever the type is needed. This *typedef* can be written in a compliant manner by omitting the tag as shown in the definition of `record2_t`.

```
typedef struct record_t                  /* Non-compliant */
{
  uint16_t  key;
  uint16_t  val;
} record1_t;

typedef struct                           /* Compliant     */
{
  uint16_t  key;
  uint16_t  val;
} record2_t;
```

Rule 2.5 A project should not contain unused macro declarations

Category Advisory

Analysis Decidable, System

Applies to C90, C99

Rationale

If a macro is declared but not used, then it is unclear to a reviewer if the macro is redundant or it has been left unused by mistake.

Example

```
void use_macro ( void )
{
#define SIZE 4
/* Non-compliant - DATA not used */
#define DATA 3
  use_int16 ( SIZE );
}
```

Rule 2.6 A function should not contain unused label declarations

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

If a label is declared but not used, then it is unclear to a reviewer if the label is redundant or it has been left unused by mistake.

Example

```
void unused_label ( void )
{
  int16_t x = 6;

label1:              /* Non-compliant */
  use_int16 ( x );
}
```

Rule 2.7 There should be no unused parameters in functions

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

Most functions will be specified as using each of their parameters. If a function parameter is unused, it is possible that the implementation of the function does not match its specification. This rule highlights such potential mismatches.

Example

```
void withunusedpara ( uint16_t *para1,
                      int16_t   unusedpara )    /* Non-compliant - unused */
{
  *para1 = 42U;
}
```

8.3 Comments

Rule 3.1	The character sequences /* and // shall not be used within a comment
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

If a comment starting sequence, /* or //, occurs within a /* comment, is it quite likely to be caused by a missing */ comment ending sequence.

If a comment starting sequence occurs within a // comment, it is probably because a region of code has been commented-out using //.

Exception

The sequence // is permitted within a // comment.

Example

Consider the following code fragment:

```
/* some comment, end comment marker accidentally omitted

<<New Page>>

Perform_Critical_Safety_Function( X );

/* this comment is non-compliant */
```

In reviewing the page containing the call to the function, the assumption is that it is executed code. Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed.

In the following C99 example, the presence of // comments changes the meaning of the program:

```
x = y // /*
      + z
      // */
    ;
```

This gives `x = y + z;` but would have been `x = y;` in the absence of the two // comment start sequences.

See also

Dir 4.4

Rule 3.2	Line-splicing shall not be used in // comments
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Amplification

Line-splicing occurs when the \ character is immediately followed by a new-line character. If the source file contains multibyte characters, they are converted to the source character set before any splicing occurs.

Rationale

If the source line containing a // comment ends with a \ character in the source character set, the next line becomes part of the comment. This may result in unintentional removal of code.

Note: line-splicing is described in Section 5.1.1.2(2) of both C90 and C99.

Example

In the following non-compliant example, the physical line containing the *if* keyword is logically part of the previous line and is therefore a comment.

```
extern bool_t b;

void f ( void )
{
  uint16_t  x = 0;    // comment \
  if ( b )
  {
    ++x;              /* This is always executed */
  }
}
```

See also

Dir 4.4

8.4 Character sets and lexical conventions

Rule 4.1	Octal and hexadecimal escape sequences shall be terminated
	C90 [Implementation 11], C99 [Implementation J.3.4(7, 8)]
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

An octal or hexadecimal escape sequence shall be terminated by either:

- The start of another escape sequence, or
- The end of the character constant or the end of a string literal.

Rationale

There is potential for confusion if an octal or hexadecimal escape sequence is followed by other characters. For example, the character constant `'\x1f'` consists of a single character whereas the character constant `'\x1g'` consists of the two characters `'\x1'` and `'g'`. The manner in which multi-character constants are represented as integers is implementation-defined.

The potential for confusion is reduced if every octal or hexadecimal escape sequence in a character constant or string literal is terminated.

Example

In this example, each of the strings pointed to by `s1`, `s2` and `s3` is equivalent to "Ag".

```
const char *s1 = "\x41g";    /* Non-compliant                            */
const char *s2 = "\x41" "g"; /* Compliant - terminated by end of literal */
const char *s3 = "\x41\x67"; /* Compliant - terminated by another escape */

int c1 = '\141t';            /* Non-compliant                            */
int c2 = '\141\t';           /* Compliant - terminated by another escape */
```

See also

C90: Section 6.1.3.4, C99: Section 6.4.4.4

Rule 4.2	Trigraphs should not be used
Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Trigraphs are denoted by a sequence of two question marks followed by a specified third character (e.g. `??-` represents a ~ (tilde) character and `??)` represents a `]`). They can cause accidental confusion with other uses of two question marks.

Note: the so-called digraphs:

```
<: :> <% %> %: %:%:
```

are permitted because they are tokens. Trigraphs are replaced wherever they appear in the program prior to preprocessing.

Example

For example the string

```
"(Date should be in the form ??-??-??)"
```

would not behave as expected, actually being interpreted by the compiler as

```
"(Date should be in the form ~~]"
```

8.5 Identifiers

Rule 5.1	*External identifiers* shall be distinct

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

This rule requires that different *external identifiers* be distinct within the limits imposed by the implementation.

The definition of distinct depends on the implementation and on the version of the C language that is being used:

- In C90 the **minimum** requirement is that the first 6 characters of *external identifiers* are significant but their case is not required to be significant;
- In C99 the **minimum** requirement is that the first 31 characters of *external identifiers* are significant, with each universal character or corresponding extended source character occupying between 6 and 10 characters.

In practice, many implementations provide greater limits. For example it is common for *external identifiers* in C90 to be case-sensitive and for at least the first 31 characters to be significant.

Rationale

If two identifiers differ only in non-significant characters, the behaviour is undefined.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit.

Note: In C99, if an extended source character appears in an *external identifier* and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies.

Example

In the following example, the definitions all occur in the same translation unit. The implementation in question supports 31 significant case-sensitive characters in *external identifiers*.

```
/*      12345678901234567890123456789012*********    Characters    */
int32_t engine_exhaust_gas_temperature_raw;
int32_t engine_exhaust_gas_temperature_scaled;   /* Non-compliant */

/*      12345678901234567890123456789012*********    Characters    */
int32_t engine_exhaust_gas_temp_raw;
int32_t engine_exhaust_gas_temp_scaled;          /* Compliant     */
```

In the following non-compliant example, the implementation supports 6 significant case-insensitive characters in *external identifiers*. The identifiers in the two translation units are different but are not distinct in their significant characters.

```
/* file1.c */
int32_t abc = 0;


/* file2.c */
int32_t ABC = 0;
```

See also

Dir 1.1, Rule 5.2, Rule 5.4, Rule 5.5

Rule 5.2 Identifiers declared in the same *scope* and name space shall be distinct

C90 [Undefined 7], C99 [Undefined 28]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule does not apply if both identifiers are *external identifiers* because this case is covered by Rule 5.1.

This rule does not apply if either identifier is a *macro identifier* because this case is covered by Rule 5.4 and Rule 5.5.

The definition of distinct depends on the implementation and on the version of the C language that is being used:

- In C90 the **minimum** requirement is that the first 31 characters are significant;
- In C99 the **minimum** requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

Rationale

If two identifiers differ only in non-significant characters, the behaviour is undefined.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long identifiers may impair the readability of code. While many automatic code generation systems produce long identifiers, there is a good argument for keeping identifier lengths well below this limit.

Example

In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage.

The identifier `engine_exhaust_gas_temperature_local` is compliant with this rule. Although it is not distinct from the identifier `engine_exhaust_gas_temperature_raw`, it is in a different scope. However, it is not compliant with Rule 5.3.

```
/*             12345678901234567890123456789*********       Characters */
extern int32_t engine_exhaust_gas_temperature_raw;
static int32_t engine_exhaust_gas_temperature_scaled;   /* Non-compliant */

void f ( void )
{
  /*        12345678901234567890123456789*********          Characters */
  int32_t   engine_exhaust_gas_temperature_local;       /* Compliant     */
}

/*             12345678901234567890123456789*********       Characters */
static int32_t engine_exhaust_gas_temp_raw;
static int32_t engine_exhaust_gas_temp_scaled;          /* Compliant     */
```

See also

Dir 1.1, Rule 5.1, Rule 5.3, Rule 5.4, Rule 5.5

Rule 5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

An identifier declared in an inner scope shall be distinct from any identifier declared in an outer scope.

The definition of distinct depends on the implementation and the version of the C language that is being used:

- In C90 the **minimum** requirement is that the first 31 characters are significant;
- In C99 the **minimum** requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

Rationale

If an identifier is declared in an inner scope but is not distinct from an identifier that already exists in an outer scope, then the inner-most declaration will "hide" the outer one. This may lead to developer confusion.

Note: An identifier declared in one name space does not hide an identifier declared in a different name space.

The terms outer and inner scope are defined as follows:

- Identifiers that have file scope can be considered as having the outermost scope;
- Identifiers that have block scope have a more inner scope;
- Successive, nested blocks, introduce more inner scopes.

Example

```
void fn1 ( void )
{
  int16_t i;       /* Declare an object "i"                              */

  {
    int16_t i;     /* Non-compliant - hides previous "i"                 */

    i = 3;         /* Could be confusing as to which "i" this refers     */
  }
}

struct astruct
{
  int16_t m;
};

extern void g ( struct astruct *p );

int16_t xyz = 0;                       /* Declare an object "xyz"        */

void fn2 ( struct astruct xyz )        /* Non-compliant - outer "xyz" is
                                        * now hidden by parameter name   */
{
  g ( &xyz );
}

uint16_t speed;

void fn3 ( void )
{
  typedef float32_t speed;   /* Non-compliant - type hides object        */
}
```

See also

Rule 5.2, Rule 5.8

Rule 5.4 *Macro identifiers* shall be distinct

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule requires that, when a macro is being defined, its name be distinct from:

- the names of the other macros that are currently defined; and
- the names of their parameters.

It also requires that the names of the parameters of a given macro be distinct from each other but does not require that macro parameters names be distinct across two different macros.

The definition of distinct depends on the implementation and on the version of the C language that is being used:

- In C90 the **minimum** requirement is that the first 31 characters of *macro identifiers* are significant;
- In C99 the **minimum** requirement is that the first 63 characters of *macro identifiers* are significant.

In practice, implementations may provide greater limits. This rule requires that *macro identifiers* be distinct within the limits imposed by the implementation.

Rationale

If two *macro identifiers* differ only in non-significant characters, the behaviour is undefined. Since macro parameters are active only during the expansion of their macro, there is no issue with parameters in one macro being confused with parameters in another macro.

If portability is a concern, it would be prudent to apply this rule using the minimum limits specified in The Standard.

Long *macro identifiers* may impair the readability of code. While many automatic code generation systems produce long *macro identifiers*, there is a good argument for keeping *macro identifier* lengths well below this limit.

Note: In C99, if an extended source character appears in a macro name and that character does not have a corresponding universal character, The Standard does not specify how many characters it occupies.

Example

In the following example, the implementation in question supports 31 significant case-sensitive characters in *macro identifiers*.

```
/*       12345678901234567890123456789012*********             Characters */
#define  engine_exhaust_gas_temperature_raw     egt_r
#define  engine_exhaust_gas_temperature_scaled  egt_s   /* Non-compliant */

/*       12345678901234567890123456789012*********             Characters */
#define  engine_exhaust_gas_temp_raw            egt_r
#define  engine_exhaust_gas_temp_scaled         egt_s   /* Compliant     */
```

See also

Rule 5.1, Rule 5.2, Rule 5.5

Rule 5.5 Identifiers shall be distinct from macro names

C90 [Undefined 7], C99 [Unspecified 7; Undefined 28]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule requires that the names of macros that exist prior to preprocessing be distinct from the identifiers that exist after preprocessing. It applies to identifiers, regardless of scope or name space, and to any macros that have been defined regardless of whether the definition is still in force when the identifier is declared.

The definition of distinct depends on the implementation and the version of the C language that is being used:

- In C90 the **minimum** requirement is that the first 31 characters are significant;
- In C99 the **minimum** requirement is that the first 63 characters are significant, with each universal character or extended source character counting as a single character.

Rationale

Keeping macro names and identifiers distinct can help to avoid developer confusion.

Example

In the following non-compliant example, the name of the *function-like macro* `Sum` is also used as an identifier. The declaration of the object `Sum` is not subject to macro-expansion because it is not followed by a `(` character. The identifier therefore exists after preprocessing has been performed.

```
#define Sum(x, y) ( ( x )  + ( y ) )

int16_t Sum;
```

The following example is compliant because there is no instance of the identifier `Sum` after preprocessing.

```
#define Sum(x, y) ( ( x )  + ( y ) )

int16_t x = Sum ( 1, 2 );
```

In the following example, the implementation in question supports 31 significant case-sensitive characters in identifiers that do not have external linkage. The example is non-compliant because the macro name is not distinct from an identifier name with internal linkage in the first 31 characters.

```
/*             12345678901234567890123456789012********* Characters */
#define        low_pressure_turbine_temperature_1  lp_tb_temp_1
static int32_t low_pressure_turbine_temperature_2;
```

See also

Rule 5.1, Rule 5.2, Rule 5.4

Rule 5.6 A *typedef* name shall be a unique identifier

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

A *typedef* name shall be unique across all name spaces and translation units. Multiple declarations of the same *typedef* name are only permitted by this rule if the type definition is made in a *header file* and that *header file* is included in multiple source files.

Rationale

Reusing a *typedef* name either as another *typedef* name or as the name of a function, object or enumeration constant, may lead to developer confusion.

Exception

The *typedef* name may be the same as the structure, union or enumeration tag name associated with the *typedef*.

Example

```
void func ( void )
{
  {
    typedef unsigned char u8_t;
  }
  {
    typedef unsigned char u8_t;     /* Non-compliant - reuse */
  }
}

typedef float mass;

void func1 ( void )
{
  float32_t mass = 0.0f;            /* Non-compliant - reuse */
}

typedef struct list
{
  struct list *next;
  uint16_t     element;
} list;                             /* Compliant - exception */

typedef struct
{
  struct chain
  {
    struct chain *list;
    uint16_t      element;
  } s1;
  uint16_t length;
} chain;                            /* Non-compliant - tag "chain" not
                                     * associated with typedef           */
```

See also

Rule 5.7

Rule 5.7	A tag name shall be a unique identifier
Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

The tag shall be unique across all name spaces and translation units.

All declarations of the tag shall specify the same type.

Multiple complete declarations of the same tag are only permitted by this rule if the tag is declared in a *header file* and that *header file* is included in multiple source files.

Rationale

Reusing a tag name may lead to developer confusion.

There is also undefined behaviour associated with reuse of tag names in C90 although this is not listed in The Standard's Annex. This undefined behaviour was recognized in C99 as a *constraint* in Section 6.7.2.3.

Exception

The tag name may be the same as the *typedef* name with which it is associated.

Example

```
struct stag
{
  uint16_t a;
  uint16_t b;
};

struct stag a1 = { 0, 0 };    /* Compliant - compatible with above        */
union  stag a2 = { 0, 0 };    /* Non-compliant - declares different type
                               * from struct stag.
                               * Constraint violation in C99              */
```

The following example also violates Rule 5.3

```
struct deer
{
  uint16_t a;
  uint16_t b;
};

void foo ( void )
{
  struct deer
  {
    uint16_t a;
  };                          /* Non-compliant - tag "deer" reused        */
}

typedef struct coord
{
  uint16_t x;
  uint16_t y;
} coord;                      /* Compliant by Exception                   */
```

```
struct elk
{
  uint16_t x;
};

struct elk       /* Non-compliant - declaration of different type
                  * Constraint violation in C99                          */
{
  uint32_t x;
};
```

See also

Rule 5.6

Rule 5.8 Identifiers that define objects or functions with external linkage shall be unique

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

An identifier used as an *external identifier* shall not be used for any other purpose in any name space or translation unit, even if it denotes an object with no linkage.

Rationale

Enforcing uniqueness of identifier names in this manner helps avoid confusion. Identifiers of objects that have no linkage need not be unique since there is minimal risk of such confusion.

Example

In the following example, `file1.c` and `file2.c` are both part of the same project.

```
/* file1.c */
int32_t   count;           /* "count" has external linkage             */
void foo ( void )          /* "foo" has external linkage               */
{
  int16_t index;           /* "index" has no linkage                   */
}

/* file2.c */
static void foo ( void )   /* Non-compliant - "foo" is not unique
{                           * (it is already defined with external
                            * linkage in file1.c)                      */
  int16_t count;           /* Non-compliant - "count" has no linkage
                            * but clashes with an identifier with
                            * external linkage                         */
  int32_t index;           /* Compliant - "index" has no linkage       */
}
```

See also

Rule 5.3

Rule 5.9	Identifiers that define objects or functions with internal linkage should be unique
Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99

Amplification

The identifier name should be unique across all name spaces and translation units. Any identifier used in this way should not have the same name as any other identifier, even if that other identifier denotes an object with no linkage.

Rationale

Enforcing uniqueness of identifier names in this manner helps avoid confusion.

Exception

An *inline function* with internal linkage may be defined in more than one translation unit provided that all such definitions are made in the same *header file* that is included in each translation unit.

Example

In the following example, `file1.c` and `file2.c` are both part of the same project.

```
/* file1.c */
static int32_t count;       /* "count" has internal linkage              */
static void foo ( void )    /* "foo" has internal linkage                */
{
  int16_t count;            /* Non-compliant - "count" has no linkage
                             * but clashes with an identifier with
                             * internal linkage                          */
  int16_t index;            /* "index" has no linkage                    */
}

void bar1 ( void )
{
  static int16_t count;     /* Non-compliant - "count" has no linkage
                             * but clashes with an identifier with
                             * internal linkage                          */
  int16_t index;            /* Compliant - "index" is not unique but
                             * has no linkage                            */
  foo ( );
}

/* End of file1.c */

/* file2.c */
static int8_t count;        /* Non-compliant - "count" has internal
                             * linkage but clashes with other
                             * identifiers of the same name              */
static void foo ( void )    /* Non-compliant - "foo" has internal
                             * linkage but clashes with a function of
                             * the same name                             */
{
  int32_t index;            /* Compliant - both "index" and "nbytes"     */
  int16_t nbytes;           /* are not unique but have no linkage        */
}
```

```
void bar2 ( void )
{
  static uint8_t nbytes;   /* Compliant - "nbytes" is not unique but
                            * has no linkage and the storage class is
                            * irrelevant                                */
}

/* End of file2.c */
```

See also

Rule 8.10

8.6 Types

Rule 6.1 Bit-fields shall only be declared with an appropriate type

C90 [Undefined 38; Implementation 29], C99 [Implementation J.3.9(1, 2)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The appropriate bit-field types are:

- C90: either *unsigned int* or *signed int*;
- C99: one of:
  - either *unsigned int* or *signed int*;
  - another explicitly signed or explicitly unsigned integer type that is permitted by the implementation;
  - *_Bool*.

Note: It is permitted to use *typedef*s to designate an appropriate type.

Rationale

Using *int* is implementation-defined because bit-fields of type *int* can be either *signed* or *unsigned*.

The use of *enum*, *short*, *char* or any other type for bit-fields is not permitted in C90 because the behaviour is undefined.

In C99, the implementation may define other integer types that are permitted in bit-field declarations.

Example

The following example is applicable to C90 and to C99 implementations that do not provide any additional bit-field types. It assumes that the *int* type is 16-bit.

```
typedef unsigned int UINT_16;

struct s {
   unsigned int b1:2;   /* Compliant                                    */
   int          b2:2;   /* Non-compliant - plain int not permitted      */
   UINT_16      b3:2;   /* Compliant - typedef designating unsigned int */
   signed long  b4:2;   /* Non-compliant even if long and int are the
                         * same size                                    */
};
```

Rule 6.2 Single-bit named bit fields shall not be of a signed type

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

According to the C99 Standard Section 6.2.6.2, a single-bit signed bit-field has a single (one) sign bit and no (zero) value bits. In any representation of integers, 0 value bits cannot specify a meaningful value.

A single-bit signed bit-field is therefore unlikely to behave in a useful way and its presence is likely to indicate programmer confusion.

Although the C90 Standard does not provide so much detail regarding the representation of types, the same considerations apply as for C99.

Note: this rule does not apply to unnamed bit fields as their values cannot be accessed.

8.7 Literals and constants

Rule 7.1 Octal constants shall not be used

[Koenig 9]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

Developers writing constants that have a leading zero might expect them to be interpreted as decimal constants.

Note: this rule does not apply to octal escape sequences because the use of a leading \ character means that there is less scope for confusion.

Exception

The integer constant zero (written as a single numeric digit), is strictly speaking an octal constant, but is a permitted exception to this rule.

Example

```
extern uint16_t code[ 10 ];

code[ 1 ] = 109;   /* Compliant     - decimal 109 */
code[ 2 ] = 100;   /* Compliant     - decimal 100 */
code[ 3 ] = 052;   /* Non-Compliant - decimal 42  */
code[ 4 ] = 071;   /* Non-Compliant - decimal 57  */
```

Rule 7.2	A “u” or “U” suffix shall be applied to all integer constants that are represented in an unsigned type
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to:

- Integer constants that appear in the controlling expressions of *#if* and *#elif* preprocessing directives;
- Any other integer constants that exist after preprocessing.

Note: during preprocessing, the type of an integer constant is determined in the same manner as after preprocessing except that:

- All signed integer types behave as if they were *long* (C90) or *intmax_t* (C99);
- All unsigned integer types behave as if they were *unsigned long* (C90) or *uintmax_t* (C99).

Rationale

The type of an integer constant is a potential source of confusion, because it is dependent on a complex combination of factors including:

- The magnitude of the constant;
- The implemented sizes of the integer types;
- The presence of any suffixes;
- The number base in which the value is expressed (i.e. decimal, octal or hexadecimal).

For example, the integer constant `40000` is of type *signed int* in a 32-bit environment but of type *signed long* in a 16-bit environment. The value `0x8000` is of type *unsigned int* in a 16-bit environment, but of type *signed int* in a 32-bit environment.

Note:

- Any value with a “U” suffix is of unsigned type;
- An unsuffixed decimal value less than 2^{31} is of signed type.

But:

- An unsuffixed hexadecimal value greater than or equal to 2^{15} may be of signed or unsigned type;
- For C90, an unsuffixed decimal value greater than or equal to 2^{31} may be of signed or unsigned type.

Signedness of constants should be explicit. If a constant is of an unsigned type, applying a "U" suffix makes it clear that the programmer understands that the constant is unsigned.

Note: this rule does not depend on the context in which a constant is used; promotion and other conversions that may be applied to the constant are not relevant in determining compliance with this rule.

Example

The following example assumes a machine with a 16-bit *int* type and a 32-bit *long* type. It shows the type of each integer constant determined in accordance with The Standard. The integer constant `0x8000` is non-compliant because it has an unsigned type but does not have a "U" suffix.

Constant	Type	Compliance
`32767`	*signed int*	Compliant
`0x7fff`	*signed int*	Compliant
`32768`	*signed long*	Compliant
`32768u`	*unsigned int*	Compliant
`0x8000`	*unsigned int*	Non-compliant
`0x8000u`	*unsigned int*	Compliant

Rule 7.3 The lowercase character "l" shall not be used in a literal suffix

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

Using the uppercase suffix "L" removes the potential ambiguity between "1" (digit 1) and "l" (letter "el") when declaring literals.

Example

Note: the examples containing the *long long* suffix are applicable only to C99.

```
const int64_t    a = 0L;
const int64_t    b = 0l;     /* Non-compliant */
const uint64_t   c = 0Lu;
const uint64_t   d = 0lU;    /* Non-compliant */
const uint64_t   e = 0ULL;
const uint64_t   f = 0Ull;   /* Non-compliant */
const int128_t   g = 0LL;
const int128_t   h = 0ll;    /* Non-compliant */
const float128_t m = 1.2L;
const float128_t n = 2.4l;   /* Non-compliant */
```

Rule 7.4 A string literal shall not be *assigned* to an object unless the object's type is "pointer to *const*-qualified *char*"

C90 [Undefined 12], C99 [Unspecified 14; Undefined 30]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

No attempt shall be made to modify a string literal or wide string literal directly.

The result of the address-of operator, `&`, applied to a string literal shall not be *assigned* to an object unless that object's type is "pointer to array of *const*-qualified *char*".

The same considerations apply to wide string literals. A wide string literal shall not be *assigned* to an object unless the object's type is "pointer to *const*-qualified *wchar_t*". The result of the address-of operator, `&`, applied to a wide string literal shall not be *assigned* to an object unless that object's type is "pointer to array of *const*-qualified *wchar_t*".

Rationale

Any attempt to modify a string literal results in undefined behaviour. For example, some implementations may store string literals in read-only memory in which case an attempt to modify the string literal will fail and may also result in an exception or crash.

This rule, when applied in conjunction with others, prevents a string literal from being modified.

It is explicitly unspecified in C99 whether string literals that share a common ending are stored in distinct memory locations. Therefore, even if an attempt to modify a string literal appears to succeed, it is possible that another string literal might be inadvertently altered.

Example

The following example shows an attempt to modify a string literal directly.

```
"0123456789"[0] = '*';   /* Non-compliant                                  */
```

These examples show how to prevent modification of string literals indirectly.

```
/* Non-compliant - s is not const-qualified */
char *s = "string";

/* Compliant - p is const-qualified; additional qualifiers are permitted */
const volatile char *p = "string";

extern void f1 ( char *s1 );

extern void f2 ( const char *s2 );

void g ( void )
{
  f1 ( "string" );       /* Non-compliant - parameter s1 is not
                          * const-qualified                          */
  f2 ( "string" );       /* Compliant                                */
}

char *name1 ( void )
{
  return ( "MISRA" );    /* Non-compliant - return type is not
                          * const-qualified                          */
}

const char *name2 ( void )
{
  return ( "MISRA" );    /* Compliant                                */
}
```

See also

Rule 11.4, Rule 11.8

8.8 Declarations and definitions

Rule 8.1	Types shall be explicitly specified
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90

Rationale

The C90 standard permits types to be omitted in some circumstances, in which case the *int* type is implicitly specified. Examples of the circumstances in which an implicit *int* might be used are:

- Object declarations;
- Parameter declarations;
- Member declarations;
- *typedef* declarations;
- Function return types.

The omission of an explicit type might lead to confusion. For example, in the declaration:

```
extern void g ( char c, const k );
```

the type of `k` is *const int* whereas *const char* might have been expected.

Example

The following examples show compliant and non-compliant object declarations:

```
extern         x;   /* Non-compliant - implicit int type */
extern int16_t x;   /* Compliant - explicit type         */
const          y;   /* Non-compliant - implicit int type */
const int16_t  y;   /* Compliant - explicit type         */
```

The following examples show compliant and non-compliant function type declarations:

```
extern f ( void );                          /* Non-compliant - implicit
                                             * int return type          */
extern int16_t f ( void );                  /* Compliant                */

extern void g ( char c, const k );          /* Non-compliant - implicit
                                             * int for parameter k      */
extern void g ( char c, const int16_t k );  /* Compliant                */
```

The following examples show compliant and non-compliant type definitions:

```
typedef ( *pfi ) ( void );             /* Non-compliant - implicit int
                                        * return type                   */
typedef int16_t ( *pfi ) ( void );     /* Compliant                     */
typedef void ( *pfv ) ( const x );     /* Non-compliant - implicit int
                                        * for parameter x               */
typedef void ( *pfv ) ( int16_t x );   /* Compliant                     */
```

The following examples show compliant and non-compliant member declarations:

```
struct str
{
  int16_t x;   /* Compliant                                  */
  const   y;   /* Non-compliant - implicit int for member y */
} s;
```

See also

Rule 8.2

Rule 8.2 Function types shall be in *prototype form* with named parameters

C90 [Undefined 22–25], C99 [Undefined 36–39, 73, 79]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The early version of C, commonly referred to as K&R C [30], did not provide a mechanism for checking the number of arguments or their types against the corresponding parameters. The type of an object or function did not have to be declared in K&R C since the default type of an object and the default return type of a function was *int*.

The C90 standard introduced function prototypes, a form of function declarator in which the parameter types were declared. This permitted argument types to be checked against parameter types. It also allowed the number of arguments to be checked except when a function prototype specified that a variable number of arguments was expected. The C90 standard did not **require** the use of function prototypes for reasons of backward compatibility with existing code. For the same reason, it continued to permit types to be omitted in which case the type would default to *int*.

The C99 standard removed the default *int* type from the language but continued to allow K&R-style function types in which there was no means to supply parameter type information in a declaration and it was optional to supply parameter type information in a definition.

The mismatch between the number of arguments and parameters, their types and the expected and actual return type of a function provides potential for undefined behaviour. The purpose of this rule along with Rule 8.1 and Rule 8.4 is to avoid this undefined behaviour by requiring parameter types and function return types to be specified explicitly. Rule 17.3 ensures that this information is available at the time of a function call, thereby requiring the compiler to diagnose any mismatch that is detected.

This rule also requires that names be specified for all the parameters in a declaration. The parameter names can provide useful information regarding the function interface and a mismatch between a declaration and definition might be indicative of a programming error.

Note: An empty parameter list is **not** valid in a prototype. If a function type has no parameters its *prototype form* uses the keyword *void*.

Example

The first example shows declarations of some functions and the corresponding definitions for some of those functions.

```
/* Compliant                                                    */
extern int16_t func1 ( int16_t n );

/* Non-compliant - parameter name not specified                 */
extern void func2 ( int16_t );

/* Non-compliant - not in prototype form                        */
static int16_t func3 ( );

/* Compliant - prototype specifies 0 parameters                 */
static int16_t func4 ( void );

/* Compliant                                                    */
int16_t func1 ( int16_t n )
{
  return n;
}

/* Non-compliant - old style identifier and declaration list */
static int16_t func3 ( vec, n )
int16_t *vec;
int16_t  n;
{
  return vec[ n - 1 ];
}
```

This example section shows the application of the rule to function types other than in function declarations and definitions.

```
/* Non-compliant - no prototype                                  */
int16_t ( *pf1 ) (  );

/* Compliant - prototype specifies 0 parameters                  */
int16_t ( *pf1 ) ( void );

/* Non-compliant - parameter name not specified                  */
typedef int16_t ( *pf2_t ) ( int16_t );

/* Compliant                                                     */
typedef int16_t ( *pf3_t ) ( int16_t n );
```

See also

Rule 8.1, Rule 8.4, Rule 17.3

Rule 8.3 All declarations of an object or function shall use the same names and type qualifiers

C90 [Undefined 10], C99 [Undefined 14], [Koenig 59–62]

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

Storage class specifiers are not included within the scope of this rule.

Rationale

Using types and qualifiers consistently across declarations of the same object or function encourages stronger typing.

Specifying parameter names in function prototypes allows the function definition to be checked for interface consistency with its declarations.

Exception

Compatible versions of the same basic type may be used interchangeably. For example, *int*, *signed* and *signed int* are all equivalent.

Example

```
extern void f ( signed int );
       void f (        int );    /* Compliant     - Exception       */
extern void g ( int * const );
       void g ( int *       );   /* Non-compliant - type qualifiers */
```

Note: all the above are not compliant with Dir 4.6.

```
extern int16_t func ( int16_t num, int16_t den );

/* Non-compliant - parameter names do not match */
int16_t func ( int16_t den, int16_t num )
{
  return num / den;
}
```

In this example the definition of `area` uses a different type name for the parameter `h` from that used in the declaration. This does not comply with the rule even though `width_t` and `height_t` are the same basic type.

```
typedef uint16_t width_t;
typedef uint16_t height_t;
typedef uint32_t area_t;

extern area_t area ( width_t w, height_t h );

area_t area ( width_t w, width_t h )
{
  return ( area_t ) w * h;
}
```

This rule does not require that a function pointer declaration use the same names as a function declaration. The following example is therefore compliant.

```
extern void f1 ( int16_t x );
extern void f2 ( int16_t y );

void f ( bool_t b )
{
  void ( *fp1 ) ( int16_t z ) = b ? f1 : f2;
}
```

See also

Rule 8.4

Rule 8.4 A compatible declaration shall be visible when an object or function with external linkage is defined

C90 [Undefined 24], C99 [Undefined 39]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A compatible declaration is one which declares a compatible type for the object or function being defined.

Rationale

If a declaration for an object or function is visible when that object or function is defined, a compiler must check that the declaration and definition are compatible. In the presence of function prototypes, as required by Rule 8.2, checking extends to the number and type of function parameters.

The recommended method of implementing declarations of objects and functions with external linkage is to declare them in a *header file*, and then include the *header file* in all those code files that need them, including the one that defines them (See Rule 8.5).

Example

In these examples there are no declarations or definitions of objects or functions other than those present in the code.

```
extern int16_t count;
       int16_t count = 0;        /* Compliant                              */

extern uint16_t speed = 6000u;   /* Non-compliant - no declaration
                                  * prior to this definition              */
uint8_t pressure = 101u;         /* Non-compliant - no declaration
                                  * prior to this definition              */

extern void func1 ( void );
extern void func2 ( int16_t x, int16_t y );
extern void func3 ( int16_t x, int16_t y );

void func1 ( void )
{
  /* Compliant */
}
```

The following non-compliant definition of `func3` also violates Rule 8.3.

```
void func2 ( int16_t x, int16_t y )
{
  /* Compliant                                  */
}

void func3 ( int16_t x, uint16_t y )
{
  /* Non-compliant - parameter types different */
}

void func4 ( void )
{
  /* Non-compliant - no declaration of func4 before this definition      */
}

static void func5 ( void )
{
  /* Compliant - rule does not apply to objects/functions with internal
   * linkage                                                              */
}
```

See also

Rule 8.2, Rule 8.3, Rule 8.5, Rule 17.3

Rule 8.5 An external object or function shall be declared once in one and only one file

[Koenig 66]

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Amplification

This rule applies to non-defining declarations only.

Rationale

Typically, a single declaration will be made in a *header file* that will be included in any translation unit in which the identifier is defined or used. This ensures consistency between:

- The declaration and the definition;
- Declarations in different translation units.

Note: there may be many *header files* in a project, but each external object or function shall only be declared in one *header file*.

Example

```
/* featureX.h */
extern int16_t a;   /* Declare a */

/* file.c */
#include "featureX.h"

int16_t a = 0;      /* Define a  */
```

See also

Rule 8.4

Rule 8.6 An identifier with external linkage shall have exactly one external definition

C90 [Undefined 44], C99 [Undefined 78], [Koenig 55, 63–65]

Category	Required
Analysis	Decidable, System
Applies to	C90, C99

Rationale

The behaviour is undefined if an identifier is used for which multiple definitions exist (in different files) or no definition exists at all. Multiple definitions in different files are not permitted by this rule even if the definitions are the same. It is undefined behaviour if the declarations are different, or initialize the identifier to different values.

Example

In this example the object `i` is defined twice.

```
/* file1.c */
int16_t i = 10;

/* file2.c */
int16_t i = 20;    /* Non-compliant - two definitions of i    */
```

In this example the object `j` has one tentative definition and one external definition.

```
/* file3.c */
int16_t j;         /* Tentative definition                    */
int16_t j = 1;     /* Compliant - external definition         */
```

The following example is non-compliant because the object `k` has two external definitions. The tentative definition in `file4.c` becomes an external definition at the end of the translation unit.

```
/* file4.c */
int16_t k;         /* Tentative definition - becomes external */

/* file5.c */
int16_t k = 0;     /* External definition                     */
```

Rule 8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

[Koenig 56, 57]

Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99

Rationale

Restricting the visibility of an object by giving it internal linkage or no linkage reduces the chance that it might be accessed inadvertently. Similarly, reducing the visibility of a function by giving it internal linkage reduces the chance of it being called inadvertently.

Compliance with this rule also avoids any possibility of confusion between an identifier and an identical identifier in another translation unit or a library.

Rule 8.8 The *static* storage class specifier shall be used in all declarations of objects and functions that have internal linkage

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

Since definitions are also declarations, this rule applies equally to definitions.

Rationale

The Standard states that if an object or function is declared with the *extern* storage class specifier and another declaration of the object or function is already visible, the linkage is that specified by the

earlier declaration. This can be confusing because it might be expected that the *extern* storage class specifier creates external linkage. The *static* storage class specifier shall therefore be consistently applied to objects and functions with internal linkage.

Example

```
static int32_t x = 0;         /* definition: internal linkage  */
extern int32_t x;             /* Non-compliant                 */

static int32_t f ( void );    /* declaration: internal linkage */
int32_t f ( void )            /* Non-compliant                 */
{
  return 1;
}

static int32_t g ( void );    /* declaration: internal linkage */
extern int32_t g ( void )     /* Non-compliant                 */
{
  return 1;
}
```

Rule 8.9 An object should be defined at block scope if its identifier only appears in a single function

Category	Advisory
Analysis	Decidable, System
Applies to	C90, C99

Rationale

Defining an object at block scope reduces the possibility that the object might be accessed inadvertently and makes clear the intention that it should not be accessed elsewhere.

Within a function, whether objects are defined at the outermost or innermost block is largely a matter of style.

It is recognized that there are situations in which it may not be possible to comply with this rule. For example, an object with static storage duration declared at block scope cannot be accessed directly from outside the block. This makes it impossible to set up and check the results of unit test cases without using indirect accesses to the object. In this kind of situation, some projects may prefer not to apply this rule.

Example

In this compliant example, `i` is declared at block scope because it is a *loop counter*. There is no need for other functions in the same file to use the same object for any other purpose.

```
void func ( void )
{
  int32_t i;

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

In this compliant example, the function `count` keeps track of the number of times it has been called and returns that number. No other function needs to know the details of the implementation of `count` so the call counter is defined with block scope.

```
uint32_t count ( void )
{
  static uint32_t call_count = 0;

  ++call_count;
  return call_count;
}
```

Rule 8.10 An *inline function* shall be declared with the static storage class

C99 [Unspecified 20; Undefined 67]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Rationale

If an *inline function* is declared with external linkage but not defined in the same translation unit, the behaviour is undefined.

A call to an *inline function* declared with external linkage may call the external definition of the function, or it may use the inline definition. Although this should not affect the behaviour of the called function, it might affect execution timing and therefore have an impact on a real-time program.

Note: an *inline function* can be made available to several translation units by placing its definition in a *header file*.

See also

Rule 5.9

Rule 8.11 When an array with external linkage is declared, its size should be explicitly specified

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to non-defining declarations only. It is possible to define an array and specify its size implicitly by means of initialization.

Rationale

Although it is possible to declare an array with incomplete type and access its elements, it is safer to do so when the size of the array may be explicitly determined. Providing size information for each declaration permits them to be checked for consistency. It may also permit a static checker to perform some array bounds analysis without needing to analyse more than one translation unit.

Example

```
extern int32_t array1[ 10 ];   /* Compliant     */
extern int32_t array2[ ];      /* Non-compliant */
```

Rule 8.12	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

An implicitly-specified enumeration constant has a value 1 greater than its predecessor. If the first enumeration constant is implicitly-specified then its value is 0.

An explicitly-specified enumeration constant has the value of the associated constant expression.

If implicitly-specified and explicitly-specified constants are mixed within an enumeration list, it is possible for values to be replicated. Such replication may be unintentional and may give rise to unexpected behaviour.

This rule requires that any replication of enumeration constants be made explicit, thus making the intent clear.

Example

In the following examples the `green` and `yellow` enumeration constants are given the same value.

```
/* Non-compliant - yellow replicates implicit green */
enum colour { red = 3, blue, green, yellow = 5 };

/* Compliant                                        */
enum colour { red = 3, blue, green = 5, yellow = 5 };
```

Rule 8.13	A pointer should point to a *const*-qualified type whenever possible
Category	Advisory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

A pointer should point to a *const*-qualified type unless either:

- It is used to modify an object, or
- It is copied to another pointer that points to a type that is not *const*-qualified by means of either:
  - *Assignment*, or
  - Memory move or copying functions.

For the purposes of simplicity, this rule is written in terms of pointers and the types that they point to. However, it applies equally to arrays and the types of the elements that they contain. An array should have elements with *const*-qualified type unless either:

- Any element of the array is modified, or
- It is copied to a pointer that points to a type that is not *const*-qualified by the means described above.

Rationale

This rule encourages best practice by ensuring that pointers are not inadvertently used to modify objects. Conceptually, it is equivalent to initially declaring:

- All arrays to have elements with *const*-qualified type, and
- All pointers to point to *const*-qualified types.

and then removing *const*-qualification only where it is necessary to comply with the *constraints* of the language standard.

Example

In the following non-compliant example, `p` is not used to modify an object but the type to which it points is not *const*-qualified.

```
uint16_t f ( uint16_t *p )
{
  return *p;
}
```

The code would be compliant if the function were defined with:

```
uint16_t g ( const uint16_t *p )
```

The following example violates a *constraint* because an attempt is made to use a *const*-qualified pointer to modify an object.

```
void h ( const uint16_t *p )
{
  *p = 0;
}
```

In the following example, the pointer `s` is *const*-qualified but the type it points to is not. Since `s` is not used to modify an object, this is non-compliant.

```
#include <string.h>

char last_char ( char * const s )
{
  return s[ strlen ( s ) - 1u ];
}
```

The code would be compliant if the function were defined with:

```
char last_char ( const char * const s )
```

In this non-compliant example, none of the elements of the array `a` are modified but the element type is not *const*-qualified.

```
uint16_t first ( uint16_t a[ 5 ] )
{
  return a[ 0 ];
}
```

The code would be compliant if the function were defined with:

```
uint16_t first ( const uint16_t a[ 5 ] )
```

Rule 8.14 The *restrict* type qualifier shall not be used

C99 [Undefined 65, 66]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Rationale

When used with care the *restrict* type qualifier may improve the efficiency of code generated by a compiler. It may also allow improved static analysis. However, to use the *restrict* type qualifier the programmer must be sure that the memory areas operated on by two or more pointers do not overlap.

There is a significant risk that a compiler will generate code that does not behave as expected if *restrict* is used incorrectly.

Example

The following example is compliant because the MISRA C Guidelines do not apply to The Standard Library functions. The programmer must ensure that the areas defined by `p`, `q` and `n` do not overlap.

```
#include <string.h>

void f ( void )
{
  /* memcpy has restrict-qualified parameters */
  memcpy ( p, q, n );
}
```

The following example is non-compliant because a function has been defined using *restrict*.

```
void user_copy ( void * restrict p, void * restrict q, size_t n )
{
}
```

8.9 Initialization

Rule 9.1 The value of an object with automatic storage duration shall not be read before it has been set

C90 [Undefined 41], C99 [Undefined 10, 17]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

For the purposes of this rule, an array element or structure member shall be considered as a discrete object.

Rationale

According to The Standard, objects with static storage duration are automatically initialized to zero unless initialized explicitly. Objects with automatic storage duration are not automatically initialized and can therefore have indeterminate values.

Note: it is sometimes possible for the explicit initialization of an automatic object to be ignored. This will happen when a jump to a label using a *goto* or *switch* statement "bypasses" the declaration of the object; the object will be declared as expected but any explicit initialization will be ignored.

Example

```
void f ( bool_t b, uint16_t *p )
{
  if ( b )
  {
    *p = 3U;
  }
}

void g ( void )
{
  uint16_t u;

  f ( false, &u );

  if ( u == 3U )
  {
    /* Non-compliant - u has not been assigned a value */
  }
}
```

In the following non-compliant C99 example, the *goto* statement jumps past the initialization of `x`.

Note: This example is also non-compliant with Rule 15.1.

```
{
  goto L1;

  uint16_t x = 10u;

L1:
  x = x + 1u;    /* Non-compliant - x has not been assigned a value */
}
```

See also

Rule 15.1, Rule 15.3

Rule 9.2 The initializer for an aggregate or union shall be enclosed in braces

C90 [Undefined 42], C99 [Undefined 76, 77]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to initializers for both objects and subobjects.

An initializer of the form `{ 0 }`, which sets all values to 0, may be used to initialize subobjects without nested braces.

Note: this rule does not itself require explicit initialization of objects or subobjects.

Rationale

Using braces to indicate initialization of subobjects improves the clarity of code and forces programmers to consider the initialization of elements in complex data structures such as multi-dimensional arrays or arrays of structures.

Exception

1. An array may be initialized using a string literal.
2. An automatic structure or union may be initialized using an expression with compatible structure or union type.
3. A designated initializer may be used to initialize part of a subobject.

Example

The following three initializations, which are permitted by The Standard, are equivalent. The first form is not permitted by this rule because it does not use braces to show subarray initialization explicitly.

```
int16_t y[ 3 ][ 2 ] = { 1, 2, 0, 0, 5, 6 };             /* Non-compliant */
int16_t y[ 3 ][ 2 ] = { { 1, 2 }, { 0 }, { 5, 6 } };    /* Compliant     */
int16_t y[ 3 ][ 2 ] = { { 1, 2 }, { 0, 0 }, { 5, 6 } }; /* Compliant     */
```

In the following example, the initialization of `z1` is compliant by virtue of Exception 3 because a designated initializer is used to initialize the subobject `z1[ 1 ]`. The initialization of `z2` is also compliant for the same reason. The initialization of `z3` is non-compliant because part of the subobject `z3[ 1 ]` is intialized with a positional initializer but is not enclosed in braces. The initialization of `z4` is compliant because a designated initializer is used to initialize the subobject `z4[ 0 ]` and the initializer for subobject `z4[ 1 ]` is brace-enclosed.

```
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 z3[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 0 ] = 0, 1 };    /* Non-compliant */
int16_t z4[ 2 ][ 2 ] = { [ 0 ][ 1 ] = 0, { 0, 1 } };    /* Compliant     */
```

The first line in the following example initializes 3 subarrays without using nested braces. The second and third lines show equivalent ways to write the same initializer.

```
float32_t a[ 3 ][ 2 ] = { 0 };                          /* Compliant     */
float32_t a[ 3 ][ 2 ] = { { 0 }, { 0 }, { 0 } };        /* Compliant     */
float32_t a[ 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       */
```

Rule 9.3 Arrays shall not be partially initialized

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

If any element of an array object or subobject is explicitly initialized, then the entire object or subobject shall be explicitly initialized.

Rationale

Providing an explicit initialization for each element of an array makes it clear that every element has been considered.

Exception

1. An initializer of the form `{ 0 }` may be used to explicitly initialize all elements of an array object or subobject.
2. An array whose initializer consists **only** of designated initializers may be used, for example to perform a sparse initialization.
3. An array initialized using a string literal does not need an initializer for every element.

Example

```
/* Compliant */
int32_t x[ 3 ] = { 0, 1, 2 };

/* Non-compliant - y[ 2 ] is implicitly initialized */
int32_t y[ 3 ] = { 0, 1 };
```

```
/* Non-compliant - t[ 0 ] and t[ 3 ] are implicitly initialized */
float32_t t[ 4 ]  = { [ 1 ] = 1.0f, 2.0f };

/* Compliant - designated initializers for sparse matrix */
float32_t z[ 50 ] = { [ 1 ] = 1.0f, [ 25 ] = 2.0f };
```

In the following compliant example, each element of the array `arr` is initialized:

```
float32_t arr[ 3 ][ 2 ] =
{
  { 0.0f, 0.0f },
  { PI / 4.0f, -PI / 4.0f },
  { 0 }   /* initializes all elements of array subobject arr[ 2 ] */
};
```

In the following example, array elements 6 to 9 are implicitly initialized to `'\0'`:

```
char h[ 10 ] = "Hello";   /* Compliant by Exception 3 */
```

Rule 9.4 An element of an object shall not be initialized more than once

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Amplification

This rule applies to initializers for both objects and subobjects.

The provision of *designated initializers* in C99 allows the naming of the components of an aggregate (structure or array) or of a union to be initialized within an initializer list and allows the object's elements to be initialized in any order by specifying the array indices or structure member names they apply to (elements having no initialization value assume the default for uninitialized objects).

Rationale

Care is required when using *designated initializers* since the initialization of object elements can be inadvertently repeated leading to overwriting of previously initialized elements. The C99 Standard does not specify whether the *side effects* in an overwritten initializer occur or not although this is not listed in Annex J.

In order to allow sparse arrays and structures, it is acceptable to only initialize those which are necessary to the application.

Example

Array initialization:

```
/*
 * Required behaviour using positional initialization
 * Compliant - a1 is -5, -4, -3, -2, -1
 */
int16_t a1[ 5 ] = { -5, -4, -3, -2, -1 };

/*
 * Similar behaviour using designated initializers
 * Compliant - a2 is -5, -4, -3, -2, -1
 */
int16_t a2[ 5 ] = { [ 0 ] = -5, [ 1 ] = -4, [ 2 ] = -3,
                    [ 3 ] = -2, [ 4 ] = -1              };
```

```
/*
 * Repeated designated initializer element values overwrite earlier ones
 * Non-compliant - a3 is -5, -4, -2, 0, -1
 */
int16_t a3[ 5 ] = { [ 0 ] = -5, [ 1 ] = -4, [ 2 ] = -3,
                    [ 2 ] = -2, [ 4 ] = -1              };
```

In the following non-compliant example, it is unspecified whether the *side effect* occurs or not:

```
uint16_t *p;

void f ( void )
{
  uint16_t a[ 2 ] = { [ 0 ] = *p++, [ 0 ] = 1 };
}
```

Structure initialization:

```
struct mystruct
{
  int32_t a;
  int32_t b;
  int32_t c;
  int32_t d;
};

/*
 * Required behaviour using positional initialization
 * Compliant - s1 is 100, -1, 42, 999
 */
struct mystruct s1 = { 100, -1, 42, 999 };

/*
 * Similar behaviour using designated initializers
 * Compliant - s2 is 100, -1, 42, 999
 */
struct mystruct s2 = { .a = 100, .b = -1, .c = 42, .d = 999 };

/*
 * Repeated designated initializer element values overwrite earlier ones
 * Non-compliant - s3 is 42, -1, 0, 999
 */
struct mystruct s3 = { .a = 100, .b = -1, .a = 42, .d = 999 };
```

Rule 9.5	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Amplification

The rule applies equally to an array subobject that is a flexible array member.

Rationale

If the size of an array is not specified explicitly, it is determined by the highest index of any of the elements that are initialized. When using designated initializers it may not always be clear which initializer has the highest index, especially when the initializer contains a large number of elements.

To make the intent clear, the array size shall be declared explicitly. This provides some protection if, during development of the program, the indices of the initialized elements are changed as it is a *constraint* violation (C99 Section 6.7.8) to initialize an element outside the bounds of an array.

Example

```
/* Non-compliant - probably unintentional to have single element */
int a1[ ] = { [ 0 ] = 1 };

/* Compliant                                                     */
int a2[ 10 ] = { [ 0 ] = 1 };
```

8.10 The *essential type model*

8.10.1 Rationale

The rules in this section collectively define the *essential type model* and restrict the C type system so as to:

1. Support a stronger system of type-checking;
2. Provide a rational basis for defining rules to control the use of implicit and explicit type conversions;
3. Promote portable coding practices;
4. Address some of the type conversion anomalies found within ISO C.

The *essential type model* does this by allocating an *essential type* to those objects and expressions which ISO C considers to be of arithmetic type. For example, adding an *int* to a *char* gives a result having *essentially character* type rather than the *int* type that is actually produced by integer promotion.

The full rationale behind the *essential type model* is given in Appendix C with Appendix D providing a comprehensive definition of the *essential type* of any arithmetic expression.

8.10.2 Essential type

The *essential type* of an object or expression is defined by its *essential type category* and size.

The *essential type category* of an expression reflects its underlying behaviour and may be:

- *Essentially Boolean*;
- *Essentially character*;
- *Essentially enum*;
- *Essentially signed*;
- *Essentially unsigned*;
- *Essentially floating*.

Note: each enumerated type is a unique *essentially enum type* identified as *enum<i>*. This allows different enumerated types to be handled as distinct types, which supports a stronger system of type-checking. One exception is the use of an enumerated type to define a Boolean value in C90. Such types are considered to have *essentially Boolean* type. Another exception is the use of *anonymous enumerations* as defined in Appendix D. *Anonymous enumerations* are a way of defining a set of related constant integers and are considered to have an *essentially signed* type.

When comparing two types of the same type category, the terms *wider* and *narrower* are used to describe their relative sizes as measured in bytes. Two different types are sometimes implemented with the same size.

The following table shows how the standard integer types map on to *essential type categories*.

Essential type category					
Boolean	character	signed	unsigned	enum<i>	floating
_Bool	char	signed char signed short signed int signed long signed long long	unsigned char unsigned short unsigned int unsigned long unsigned long long	named enum	float double long double

Note: C99 implementations may provide *extended integer types* each of which would be allocated a location appropriate to its rank and signedness (see C99 Section 6.3.1.1).

The restrictions enforced by the rules in this section also apply to the compound assignment operators (e.g. ^=) as these are equivalent to assigning the result obtained from the use of one of the arithmetic, bitwise or shift operators. For example:

```
u8a += u8b + 1U;
```

is equivalent to:

```
u8a = u8a + ( u8b + 1U );
```

Rule 10.1 Operands shall not be of an inappropriate *essential type*

C90 [Unspecified 23; Implementation 14, 17, 19, 32]
C99 [Undefined 13, 49; Implementation J3.4(2, 5), J3.5(5), J3.9(6)]

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Amplification

In the following table a number within a cell indicates where a restriction applies to the use of an *essential type* as an operand to an operator. These numbers correspond to paragraphs in the Rationale section below and indicate why each restriction is imposed.

Operator	Operand	Essential type category of arithmetic 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

Operator	Operand	Essential type category of arithmetic operand					
		Boolean	character	enum	signed	unsigned	floating
<< >>	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						

Under this rule the ++ and –– operators behave the same way as the binary + and – operators.

Other rules place further restrictions on the combination of *essential types* that may be used within an expression.

Rationale

1. The use of an expression of *essentially floating type* for these operands is a *constraint* violation.
2. An expression of *essentially Boolean type* should always be used where an operand is interpreted as a Boolean value.
3. An operand of *essentially Boolean type* should not be used where an operand is interpreted as a numeric value.
4. An operand of *essentially character type* should not be used where an operand is interpreted as a numeric value. The numeric values of character data are implementation-defined.
5. An operand of *essentially enum type* should not be used in an arithmetic operation because an *enum* object uses an implementation-defined integer type. An operation involving an *enum* object may therefore yield a result with an unexpected type. Note that an enumeration constant from an *anonymous enum* has *essentially signed type*.
6. Shift and bitwise operations should only be performed on operands of *essentially unsigned type*. The numeric value resulting from their use on *essentially signed types* is implementation-defined.
7. The right hand operand of a shift operator should be of *essentially unsigned type* to ensure that undefined behaviour does not result from a negative shift.
8. An operand of *essentially unsigned type* should not be used as the operand to the unary minus operator, as the signedness of the result is determined by the implemented size of *int*.

Exception

A non-negative *integer constant expression* of *essentially signed type* may be used as the right hand operand to a shift operator.

Example

```
enum enuma { a1, a2, a3 } ena, enb;    /* Essentially enum<enuma>          */
enum { K1 = 1, K2 = 2 };               /* Essentially signed               */
```

The following examples are non-compliant. The comments refer to the numbered rationale item that results in the non-compliance.

```
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           */
```

The following example is non-compliant with this rule and also violates Rule 10.3:

```
ena += a1         /* Rationale 5 - enum type used in arithmetic operation */
```

The following examples are compliant:

```
bla && blb
bla ? u8a : u8b

cha - chb
cha > chb

ena > a1
K1 * s8a          /* Compliant as K1 from anonymous enum                  */

s8a + s16b
-( s8a ) * s8b
s8a > 0
--s16b

u8a + u16b
u8a & 2U

u8a > 0U
u8a << 2U
u8a << 1          /* Compliant by exception                               */

f32a + f32b
f32a > 0.0
```

The following is compliant with this rule but violates Rule 10.2

```
cha + chb
```

See also

Rule 10.2

Rule 10.2 Expressions of *essentially character type* shall not be used inappropriately in addition and subtraction operations

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The appropriate uses are:

1. For the + operator, one operand shall have *essentially character type* and the other shall have *essentially signed type* or *essentially unsigned type*. The result of the operation has *essentially character type*.
2. For the – operator, the first operand shall have *essentially character type* and the second shall have *essentially signed type*, *essentially unsigned type* or *essentially character type*. If both operands have *essentially character type* then the result has the *standard type* (usually *int* in this case) else the result has *essentially character type*.

Rationale

Expressions with *essentially character type* (character data) shall not be used arithmetically as the data does not represent numeric values.

The uses above are permitted as they allow potentially reasonable manipulation of character data. For example:

- Subtraction of two operands with *essentially character type* might be used to convert between digits in the range '0' to '9' and the corresponding ordinal value;
- Addition of an *essentially character type* and an *essentially unsigned type* might be used to convert an ordinal value to the corresponding digit in the range '0' to '9';
- Subtraction of an *essentially unsigned type* from an *essentially character type* might be used to convert a character from lowercase to uppercase.

Example

The following examples are compliant:

```
'0' + u8a   /* Convert u8a to digit   */
s8a + '0'   /* Convert s8a to digit   */
cha - '0'   /* Convert cha to ordinal */
'0' - s8a   /* Convert -s8a to digit  */
```

The following examples are non-compliant:

```
s16a - 'a'
'0' + f32a
cha + ':'
cha - ena
```

See also

Rule 10.1

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*

C90 [Undefined 15; Implementation 16]
C99 [Undefined 15, 16; Implementation 3.5(4)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The following operations are covered by this rule:

1. *Assignment* as defined in the Glossary;
2. The conversion of the *constant expression* in a *switch* statement's *case* label to the promoted type of the controlling expression.

Rationale

The C language allows the programmer considerable freedom and will permit assignments between different arithmetic types to be performed automatically. However, the use of these implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. Further details of concerns with the C type system can be found in Appendix C.

The use of stronger typing, as enforced by the MISRA *essential type model*, reduces the likelihood of these problems occurring.

Exception

1. A non-negative *integer constant expression* of *essentially signed type* may be *assigned* to an object of *essentially unsigned type* if its value can be represented in that type.
2. The initializer `{ 0 }` may be used to initialize an aggregate or union type.

Example

```
enum enuma { A1, A2, A3   } ena;
enum enumb { B1, B2, B3   } enb;
enum       { K1=1, K2=128 };
```

The following are compliant:

```
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      */
```

The following are non-compliant as they have different *essential type categories*:

```
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                            */
```

The following are non-compliant as they contain assignments to a narrower *essential type*:

```
s8a  = K2;                    /* Constant value does not fit            */
u16a = u32a;                  /* uint32_t to uint16_t                   */

use_uint16 ( u32a );          /* uint32_t to uint16_t                   */

uint8_t foo1 ( uint16_t x )
{
   return x;                  /* uint16_t to uint8_t                    */
}
```

See also

Rule 10.4, Rule 10.5, Rule 10.6

Rule 10.4 Both operands of an operator in which the *usual arithmetic conversions* are performed shall have the same *essential type category*

C90 [Implementation 21], C99 [Implementation 3.6(4)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to operators that are described in *usual arithmetic conversions* (see C90 Section 6.2.1.5, C99 Section 6.3.1.8). This includes all the binary operators, excluding the shift, logical `&&`, logical `||` and comma operators. In addition, the second and third operands of the ternary operator are covered by this rule.

Note: the increment and decrement operators are not covered by this rule.

Rationale

The C language allows the programmer considerable freedom and will permit conversions between different arithmetic types to be performed automatically. However, the use of these implicit conversions can lead to unintended results, with the potential for loss of value, sign or precision. Further details of concerns with the C type system can be found in Appendix C.

The use of stronger typing, as enforced by the MISRA *essential type model*, allows implicit conversions to be restricted to those that should then produce the answer expected by the developer.

Exception

The following are permitted to allow a common form of character manipulation to be used:

1. The binary + and += operators may have one operand with *essentially character* type and the other operand with an *essentially signed* or *essentially unsigned* type;
2. The binary – and –= operators may have a left-hand operand with *essentially character* type and a right-hand operand with an *essentially signed* or *essentially unsigned* type.

Example

```
enum enuma { A1, A2, A3 } ena;
enum enumb { B1, B2, B3 } enb;
```

The following are compliant as they have the same *essential type category*:

```
ena > A1
u8a + u16b
```

The following is compliant by exception 1:

```
cha += u8a
```

The following is non-compliant with this rule and also violates Rule 10.3:

```
s8a += u8a   /* signed and unsigned         */
```

The following are non-compliant:

```
u8b + 2      /* unsigned and signed         */
enb > A1     /* enum<enumb> and enum<enuma> */
ena == enb   /* enum<enuma> and enum<enumb> */
u8a += cha   /* unsigned and char           */
```

See also

Rule 10.3, Rule 10.7

Rule 10.5 The value of an expression should not be cast to an inappropriate *essential type*

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Amplification

The casts which should be avoided are shown in the following table, where values are cast (explicitly converted) to the *essential type category* of the first column.

Essential type category	from					
to	*Boolean*	*character*	*enum*	*signed*	*unsigned*	*floating*
Boolean		Avoid	Avoid	Avoid	Avoid	Avoid
character	Avoid					Avoid
enum	Avoid	Avoid	Avoid*	Avoid	Avoid	Avoid
signed	Avoid					
unsigned	Avoid					
floating	Avoid	Avoid				

* *Note:* an enumerated type may be cast to an enumerated type provided that the cast is to the same *essential enumerated* type. Such casts are redundant.

Casting from *void* to any other type is not permitted as it results in undefined behaviour. This is covered by Rule 1.3.

Rationale

An explicit cast may be introduced for legitimate functional reasons, for example:

- To change the type in which a subsequent arithmetic operation is performed;
- To truncate a value deliberately;
- To make a type conversion explicit in the interests of clarity.

However, some explicit casts are considered inappropriate:

- In C99, the result of a cast or assignment to *_Bool* is always 0 or 1. This is not necessarily the case when casting to another type which is defined as *essentially Boolean*;
- A cast to an *essentially enum type* may result in a value that does not lie within the set of enumeration constants for that type;
- A cast from *essentially Boolean* to any other type is unlikely to be meaningful;
- Converting between floating and character types is not meaningful as there is no precise mapping between the two representations.

Exception

An *integer constant expression* with the value 0 or 1 of either signedness may be cast to a type which is defined as *essentially Boolean*. This allows the implementation of non-C99 Boolean models.

Example

```
( bool_t ) false     /* Compliant - C99 'false' is essentially Boolean */
( int32_t ) 3U       /* Compliant                                      */
( bool_t ) 0         /* Compliant - by exception                       */
( bool_t ) 3U        /* Non-compliant                                  */

( int32_t ) ena      /* Compliant                                      */
( enum enuma ) 3     /* Non-compliant                                  */
( char ) enc         /* Compliant                                      */
```

See also

Rule 10.3, Rule 10.8

8.10.3 Composite operators and expressions

Some of the concerns mentioned in Appendix C can be avoided by restricting the implicit and explicit conversions that may be applied to non-trivial expressions. These include:

- The confusion about the type in which integer expressions are evaluated, as this depends on the type of the operands after any integer promotion. The type of the result of an arithmetic operation depends on the implemented size of *int*;
- The common misconception among programmers that the type in which a calculation is conducted is influenced by the type to which the result is assigned or cast. This false expectation may lead to unintended results.

In addition to the previous rules, the *essential type model* places further restrictions on expressions whose operands are *composite expressions*, as defined below.

The following are defined as *composite operators* in this document:

- Multiplicative (`*`, `/`, `%`)
- Additive (binary +, binary –)
- Bitwise (`&`, `|`, `^`)
- Shift (<<, >>)
- Conditional (`?:`) if either the second or third operand is a *composite expression*

A compound assignment is equivalent to an assignment of the result of its corresponding *composite operator.*

A *composite expression* is defined in this document as a non-*constant expression* which is the direct result of a *composite operator.*

Note:

- The result of a compound assignment operator is not a *composite expression*;
- A parenthesized *composite expression* is also a *composite expression*;
- A *constant expression* is not a *composite expression.*

Rule 10.6	The value of a *composite expression* shall not be assigned to an object with wider *essential type*

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule covers the assigning operations described in Rule 10.3.

Rationale

The rationale is described in the introduction on *composite operators and expressions* (see Section 8.10.3).

Example

The following are compliant:

```
u16c = u16a + u16b;                /* Same essential type                */
u32a = ( uint32_t ) u16a + u16b;   /* Cast causes addition in uint32_t   */
```

The following are non-compliant:

```
u32a = u16a + u16b;                /* Implicit conversion on assignment  */
use_uint32 ( u16a + u16b );        /* Implicit conversion of fn argument */
```

See also

Rule 10.3, Rule 10.7, Section 8.10.3

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*

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The rationale is described in the introduction on *composite operators and expressions* (see Section 8.10.3).

Restricting implicit conversions on *composite expressions* means that sequences of arithmetic operations within an expression must be conducted in exactly the same *essential type*. This reduces possible developer confusion.

Note: this does not imply that all operands in an expression are of the same *essential type*.

The expression `u32a + u16b + u16c` is compliant as both additions will notionally be performed in type `uint32_t`. In this case only *non-composite expressions* are implicitly converted.

The expression `(u16a + u16b) + u32c` is non-compliant as the left addition is notionally performed in type `uint16_t` and the right in type `uint32_t`, requiring an implicit conversion of the *composite expression* `u16a + u16b` to `uint32_t`.

Example

The following are compliant:

```
  u32a * u16a + u16b                    /* No composite 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       */
```

The following are non-compliant:

```
  u32a * ( u16a + u16b )   /* Implicit conversion of ( u16a + u16b )     */
 u32a += ( u16a + u16b )   /* Implicit conversion of ( u16a + u16b )     */
```

See also

Rule 10.4, Rule 10.6, Section 8.10.3

Rule 10.8	The value of a *composite expression* shall not be cast to a different *essential type category* or a wider *essential type*

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The rationale is described in the introduction on *composite operators and expressions* (see Section 8.10.3).

Casting to a wider type is not permitted as the result may vary between implementations. Consider the following:

```
( uint32_t ) ( u16a + u16b );
```

On a 16-bit machine the addition will be performed in 16 bits with the result wrapping modulo-2 **before** it is cast to 32 bits. However, on a 32-bit machine the addition will take place in 32 bits and would preserve high-order bits that would have been lost on a 16-bit machine.

Casting to a narrower type with the same *essential type category* is acceptable as the explicit truncation of the result always leads to the same loss of information.

Example

```
( uint16_t ) ( u32a + u32b )   /* Compliant                              */
( uint16_t ) ( s32a + s32b )   /* Non-compliant - different essential
                                *                 type category         */
( uint16_t ) s32a              /* Compliant     - s32a is not composite */
( uint32_t ) ( u16a + u16b )   /* Non-compliant - cast to wider
                                *                 essential type        */
```

See also

Rule 10.5, Section 8.10.3

8.11 Pointer type conversions

Pointer types can be classified as follows:

- Pointer to object;
- Pointer to function;
- Pointer to incomplete;
- Pointer to *void*;
- A *null pointer constant*, i.e. the value 0, optionally cast to *void* *.

The only conversions involving pointers that are permitted by The Standard are:

- A conversion from a pointer type into *void*;
- A conversion from a pointer type into an arithmetic type;
- A conversion from an arithmetic type into a pointer type;
- A conversion from one pointer type into another pointer type.

Although permitted by the language *constraints*, conversion between pointers and any arithmetic types other than integer types is undefined.

The following permitted pointer conversions do **not** require an explicit cast:

- A conversion from a pointer type into *_Bool* (C99 only);
- A conversion from a *null pointer constant* into a pointer type;
- A conversion from a pointer type into a compatible pointer type provided that the destination type has all the type qualifiers of the source type;
- A conversion between a pointer to an object or incomplete type and *void* *, or qualified version thereof, provided that the destination type has all the type qualifiers of the source type.

In C99, any implicit conversion that does not fall into this subset of pointer conversions violates a *constraint* (C99 Sections 6.5.4 and 6.5.16.1).

In C90, any implicit conversion that does not fall into this subset of pointer conversions leads to undefined behaviour (C90 Sections 6.3.4 and 6.3.16.1).

The conversion between pointer types and integer types is implementation-defined.

Rule 11.1 Conversions shall not be performed between a pointer to a function and any other type

C90 [Undefined 24, 27–29], C99 [Undefined 21, 23, 39, 41]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A pointer to a function shall only be converted into or from a pointer to a function with a compatible type.

Rationale

The conversion of a pointer to a function into or from any of:

- Pointer to object;
- Pointer to incomplete;
- *void* *

results in undefined behaviour.

If a function is called by means of a pointer whose type is not compatible with the called function, the behaviour is undefined. Conversion of a pointer to a function into a pointer to a function with a different type is permitted by The Standard. Conversion of an integer into a pointer to a function is also permitted by The Standard. However, both are prohibited by this rule in order to avoid the undefined behaviour that would result from calling a function using an incompatible pointer type.

Exception

1. A *null pointer constant* may be converted into a pointer to a function;
2. A pointer to a function may be converted into *void*;
3. A function type may be implicitly converted into a pointer to that function type.

Note: exception 3 covers the implicit conversions described in C90 Section 6.2.2.1 and C99 Section 6.3.2.1. These conversions commonly occur when:

- A function is called directly, i.e. using a function identifier to denote the function to be called;
- A function is assigned to a function pointer.

Example

```
typedef void ( *fp16 ) ( int16_t n );
typedef void ( *fp32 ) ( int32_t n );

#include <stdlib.h>                     /* To obtain macro NULL          */

fp16 fp1 = NULL;                        /* Compliant - exception 1       */
fp32 fp2 = ( fp32 ) fp1;                /* Non-compliant - function
                                         * pointer into different
                                         * function pointer              */

if ( fp2 != NULL )                      /* Compliant - exception 1       */
{
}

fp16 fp3 = ( fp16 ) 0x8000;             /* Non-compliant - integer into
                                         * function pointer              */
fp16 fp4 = ( fp16 ) 1.0e6F;             /* Non-compliant - float into
                                         * function pointer              */
```

In the following example, the function call returns a pointer to a function type. Casting the return value into *void* is compliant with this rule.

```
typedef fp16 ( *pfp16 ) ( void );

pfp16 pfp1;

( void ) ( *pfp1 ( ) );    /* Compliant - exception 2 - cast function
                            * pointer into void                         */
```

The following examples show compliant implicit conversions from a function type into a pointer to that function type.

```
extern void f ( int16_t n );

f ( 1 );              /* Compliant - exception 3 - implicit conversion
                       * of f into pointer to function                  */
fp16 fp5 = f;         /* Compliant - exception 3                        */
```

Rule 11.2 Conversions shall not be performed between a pointer to an incomplete type and any other type

C90 [Undefined 29], C99 [Undefined 21, 22, 41]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A pointer to an incomplete type shall not be converted into another type.

A conversion shall not be made into a pointer to incomplete type.

Although a pointer to *void* is also a pointer to an incomplete type, this rule does not apply to pointers to *void* as they are covered by Rule 11.5.

Rationale

Conversion into or from a pointer to an incomplete type may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to an incomplete type into or from a floating type always results in undefined behaviour.

Pointers to an incomplete type are sometimes used to hide the representation of an object. Converting a pointer to an incomplete type into a pointer to object would break this encapsulation.

Exception

1. A *null pointer constant* may be converted into a pointer to an incomplete type.
2. A pointer to an incomplete type may be converted into *void*.

Example

```
struct s;                   /* Incomplete type                        */
struct t;                   /* A different incomplete type            */
struct s *sp;
struct t *tp;
int16_t  *ip;

#include <stdlib.h>         /* To obtain macro NULL                   */

ip = ( int16_t  * ) sp;     /* Non-compliant                          */
sp = ( struct s * ) 1234;   /* Non-compliant                          */
tp = ( struct t * ) sp;     /* Non-compliant - casting pointer into a
                             * different incomplete type              */
sp = NULL;                  /* Compliant - exception 1                */
```

```
struct s *f ( void );

( void ) f ( );             /* Compliant - exception 2                  */
```

See also

Rule 11.5

Rule 11.3 A cast shall not be performed between a pointer to object type and a pointer to a different object type

C90 [Undefined 20], C99 [Undefined 22, 34]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to the unqualified types that are pointed to by the pointers.

Rationale

Casting a pointer to object into a pointer to a different object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Even if conversion is known to produce a pointer that is correctly aligned, the behaviour may be undefined if that pointer is used to access an object. For example, if an object whose type is *int* is accessed as a *short* the behaviour is undefined even if *int* and *short* have the same representation and alignment requirements. See C90 Section 6.3, C99 Section 6.5, paragraph 7 for details.

Exception

It is permitted to convert a pointer to object type into a pointer to one of the object types *char*, *signed char* or *unsigned char*. The Standard guarantees that pointers to these types can be used to access the individual bytes of an object.

Example

```
uint8_t  *p1;
uint32_t *p2;

/* Non-compliant - possible incompatible alignment */
p2 = ( uint32_t * ) p1;

extern uint32_t read_value ( void );
extern void print ( uint32_t n );

void f ( void )
{
  uint32_t  u    = read_value ( );
  uint16_t *hi_p = ( uint16_t * ) &u;   /* Non-compliant even though
                                         * probably correctly aligned   */

  *hi_p = 0;     /* Attempt to clear high 16-bits on big-endian machine */
  print ( u );   /* Line above may appear not to have been performed    */
}
```

The following example is compliant because the rule applies to the unqualified pointer types. It does not prevent type qualifiers from being added to the object type.

```
const short *p;
const volatile short *q;

q = ( const volatile short * ) p;    /* Compliant */
```

The following example is non-compliant because the unqualified pointer types are different, namely "pointer to *const*-qualified *int*" and "pointer to *int*".

```
int * const * pcpi;
const int * const * pcpci;

pcpci = ( const int * const * ) pcpi;
```

See also

Rule 11.4, Rule 11.5, Rule 11.8

Rule 11.4 A conversion should not be performed between a pointer to object and an integer type

C90 [Undefined 20; Implementation 24]
C99 [Undefined 21, 34; Implementation J.3.7(1)]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A pointer should not be converted into an integer.

An integer should not be converted into a pointer.

Rationale

Conversion of an integer into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to object into an integer may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

Note: the C99 types *intptr_t* and *uintptr_t*, declared in `<stdint.h>`, are respectively signed and unsigned integer types capable of representing pointer values. Despite this, conversions between a pointer to object and these types is not permitted by this rule because their use does not avoid the undefined behaviour associated with misaligned pointers.

Casting between a pointer and an integer type should be avoided where possible, but may be necessary when addressing memory mapped registers or other hardware specific features. If casting between integers and pointers is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under Rule 11.3.

Exception

A *null pointer constant* that has integer type may be converted into a pointer to object.

Example

```
uint8_t *PORTA = ( uint8_t * ) 0x0002;     /* Non-compliant */

uint16_t *p;

int32_t   addr = ( int32_t ) &p;           /* Non-compliant */
uint8_t  *q    = ( uint8_t * ) addr;       /* Non-compliant */
bool_t    b    = ( bool_t ) p;             /* Non-compliant */

enum etag { A, B } e = ( enum etag ) p;    /* Non-compliant */
```

See also

Rule 11.3, Rule 11.7, Rule 11.9

Rule 11.5 A conversion should not be performed from pointer to *void* into pointer to object

C90 [Undefined 20], C99 [Undefined 22, 34]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Conversion of a pointer to *void* into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour. It should be avoided where possible but may be necessary, for example when dealing with memory allocation functions. If conversion from a pointer to object into a pointer to *void* is used, care should be taken to ensure that any pointers produced do not give rise to the undefined behaviour discussed under Rule 11.3.

Exception

A *null pointer constant* that has type pointer to *void* may be converted into pointer to object.

Example

```
uint32_t *p32;
void     *p;
uint16_t *p16;

p   = p32;                /* Compliant - pointer to uint32_t into
                           * pointer to void                          */
p16 = p;                  /* Non-compliant                            */
p   = ( void * ) p16;     /* Compliant                                */
p32 = ( uint32_t * ) p;   /* Non-compliant                            */
```

See also

Rule 11.2, Rule 11.3

Rule 11.6 A cast shall not be performed between pointer to *void* and an arithmetic type

C90 [Undefined 29; Implementation 24]
C99 [Undefined 21, 41; Implementation J.3.7(1)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Conversion of an integer into a pointer to *void* may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to *void* into an integer may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

Conversion between any non-integer arithmetic type and pointer to *void* is undefined.

Exception

An *integer constant expression* with value 0 may be cast into pointer to *void*.

Example

```
void     *p;
uint32_t  u;

/* Non-compliant - implementation-defined */
p = ( void * ) 0x1234u;

/* Non-compliant - undefined              */
p = ( void * ) 1024.0f;

/* Non-compliant - implementation-defined */
u = ( uint32_t ) p;
```

Rule 11.7 A cast shall not be performed between pointer to object and a non-integer arithmetic type

C90 [Undefined 29; Implementation 24]
C99 [Undefined 21, 41; Implementation J.3.7(1)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

For the purposes of this rule a non-integer arithmetic type means one of:

- *Essentially Boolean*;
- *Essentially character*;
- *Essentially enum*;
- *Essentially floating*.

Rationale

Conversion of an *essentially Boolean*, *essentially character* or *essentially enum* type into a pointer to object may result in a pointer that is not correctly aligned, resulting in undefined behaviour.

Conversion of a pointer to object into an *essentially Boolean*, *essentially character* or *essentially enum* type may produce a value that cannot be represented in the chosen integer type resulting in undefined behaviour.

Conversion of a pointer to object into or from an *essentially floating* type results in undefined behaviour.

Example

```
int16_t  *p;
float32_t f;

f = ( float32_t ) p;    /* Non-compliant */
p = ( int16_t * ) f;    /* Non-compliant */
```

See also

Rule 11.4

Rule 11.8 A cast shall not remove any *const* or *volatile* qualification from the type pointed to by a pointer

C90 [Undefined 12, 39, 40], C99 [Undefined 30, 61, 62]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Any attempt to remove the qualification associated with the addressed type by using casting is a violation of the principle of type qualification.

Note: the qualification referred to here is not the same as any qualification that may be applied to the pointer itself.

Some of the problems that might arise if a qualifier is removed from the addressed object are:

- Removing a *const* qualifier might circumvent the read-only status of an object and result in it being modified;
- Removing a *const* qualifier might result in an exception when the object is accessed;
- Removing a *volatile* qualifier might result in accesses to the object being optimized away.

Note: removal of the C99 *restrict* type qualifier is benign.

Example

```
         uint16_t          x;
         uint16_t * const  cpi = &x;   /* const pointer               */
         uint16_t * const *pcpi;       /* pointer to const pointer    */
         uint16_t *       *ppi;
   const uint16_t         *pci;        /* pointer to const            */
volatile uint16_t         *pvi;        /* pointer to volatile         */
         uint16_t         *pi;

pi = cpi;                              /* Compliant - no conversion
                                                    no cast required   */
pi  = (uint16_t *)pci;                 /* Non-compliant               */
pi  = (uint16_t *)pvi;                 /* Non-compliant               */
ppi = (uint16_t * *)pcpi;              /* Non-compliant               */
```

See also

Rule 11.3

Rule 11.9	The macro `NULL` shall be the only permitted form of integer *null pointer constant*
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

An *integer constant expression* with the value 0 shall be derived from expansion of the macro `NULL` if it appears in any of the following contexts:

- As the value being *assigned* to a pointer;
- As an operand of an == or `!=` operator whose other operand is a pointer;
- As the second operand of a `?:` operator whose third operand is a pointer;
- As the third operand of a `?:` operator whose second operand is a pointer.

Ignoring whitespace and any surrounding parentheses, any such *integer constant expression* shall represent the entire expansion of `NULL`.

Note: a *null pointer constant* of the form `(void *)0` is permitted, whether or not it was expanded from `NULL`.

Rationale

Using `NULL` rather than 0 makes it clear that a *null pointer constant* was intended.

Example

In the following example, the initialization of `p2` is compliant because the *integer constant expression* 0 does not appear in one of the contexts prohibited by this rule.

```
int32_t *p1 = 0;              /* Non-compliant */

int32_t *p2 = ( void * ) 0;   /* Compliant     */
```

In the following example, the comparison between `p2` and `(void *)0` is compliant because the *integer constant expression* 0 appears as the operand of a cast and not in one of the contexts prohibited by this rule.

```
#define MY_NULL_1 0
#define MY_NULL_2 ( void * ) 0

if ( p1 == MY_NULL_1 )     /* Non-compliant */
{
}
if ( p2 == MY_NULL_2 )     /* Compliant     */
{
}
```

The following example is compliant because use of the macro `NULL` provided by the implementation is always permitted, even if it expands to an *integer constant expression* with value 0.

```
/* Could also be stdio.h, stdlib.h and others */
#include <stddef.h>

extern void f ( uint8_t *p );

/* Compliant for any conforming definition of NULL, such as:
 *    0
 *    ( void * ) 0
 *    ( ( ( 0 ) ) )
 *    ( ( ( 1 - 1 ) ) )
 */
f ( NULL );
```

See also

Rule 11.4

8.12 Expressions

Rule 12.1	The precedence of operators within expressions should be made explicit

Category Advisory

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Amplification

The following table is used in the definition of this rule.

Description	Operator or Operand	Precedence
Primary	identifier, constant, string literal, `(` expression `)`	16 (high)
Postfix	`[]` `()` (function call) `.` `->` ++ (post-increment) `--` (post-decrement) `()` `{}` (C99: compound literal)	15
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
Assignment	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`	1
Comma	`,`	0 (low)

The precedences used in this table are chosen to allow a concise description of the rule. They are not necessarily the same as those that might be encountered in other descriptions of operator precedence.

For the purposes of this rule, the precedence of an expression is the precedence of the element (operand or operator) at the root of the parse tree for that expression.

For example: the parse tree for the expression `a << b + c` can be represented as:

```
    <<
   /  \
  a    +
      / \
     b   c
```

The element at the root of this parse tree is `'<<'` so the expression has precedence 10.

The following advice is given:

- The operand of the *sizeof* operator should be enclosed in parentheses;
- An expression whose precedence is in the range 2 to 12 should have parentheses around any operand that has both:
  - Precedence of less than 13, and
  - Precedence greater than the precedence of the expression.

Rationale

The C language has a relatively large number of operators and their relative precedences are not intuitive. This can lead less experienced programmers to make mistakes. Using parentheses to make operator precedence explicit removes the possibility that the programmer's expectations are incorrect. It also makes the original programmer's intention clear to reviewers or maintainers of the code.

It is recognized that overuse of parentheses can clutter the code and reduce its readability. This rule aims to achieve a compromise between code that is hard to understand because it contains either too many or too few parentheses.

Examples

The following example shows expressions with a unary or postfix operator whose operands are either *primary-expressions* or expressions whose top-level operators have precedence 15.

```
a[ i ]->n;      /* Compliant - no need to write ( a[ i ] )->n          */
*p++;           /* Compliant - no need to write *( p++ )               */
sizeof x + y;   /* Non-compliant - write either sizeof ( x ) + y
                 * or sizeof ( x + y )                                 */
```

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 );
```

The following example shows a variety of mixed-operator expressions:

```
/* Compliant - no need to write f ( ( a + b ), c ) */
x = f ( a + b, c );

/* Non-compliant
 * Operands of conditional operator (precedence 2) are:
 *   == precedence 8 needs parentheses
 *   a  precedence 16 does not need parentheses
 *   -  precedence 11 needs parentheses
 */
x = a == b ? a : a - b;

/* Compliant */
x = ( a == b ) ? a : ( a - b );
```

```
/* Compliant
 * Operands of << operator (precedence 10) are:
 *   a  precedence 16 does not need parentheses
 *  ( E ) precedence 16 already parenthesized
 */
x = a << ( b + c );

/* Compliant
 * Operands of && operator (precedence 4) are:
 *   a  precedence 16 does not need parentheses
 *   && precedence 4 does not need parentheses
 */
if ( a && b && c )
{
}

/* Compliant
 * Operands of && operator (precedence 4) are:
 *   defined(X) precedence 14 does not need parentheses
 *   (E) precedence 16 already parenthesized
 */
#if defined ( X ) && ( ( X + Y ) > Z )

/* Compliant
 * Operands of && operator (precedence 4) are:
 *   !defined ( X ) precedence 14 does not need parentheses
 *    defined ( Y ) precedence 14 does not need parentheses
 * Operand of ! operator (precedence 14) is:
 *    defined ( X ) precedence 14 does not need parentheses
 */
#if !defined ( X ) && defined ( Y )
```

Note: this rule does not require the operands of a `,` operator to be parenthesized. Use of the `,` operator is prohibited by Rule 12.3.

```
x = a, b; /* Compliant - parsed as ( x = a ), b */
```

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

C90 [Undefined 32], C99 [Undefined 48]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

If the right hand operand is negative, or greater than or equal to the width of the left hand operand, then the behaviour is undefined.

If, for example, the left hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number in the range 0 to 15.

See Section 8.10 for a description of *essential type* and the limitations on the *essential types* for the operands of shift operators.

There are various ways of ensuring this rule is followed. The simplest is for the right hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure

that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at run time or by review). Otherwise both limits will need to be checked.

Example

```
u8a = u8a << 7;                 /* Compliant     */
u8a = u8a << 8;                 /* Non-compliant */
u16a = ( uint16_t ) u8a << 9;   /* Compliant     */
```

To assist in understanding the following examples, it should be noted that the *essential type* of `1u` is *essentially unsigned char*, whereas the *essential type* of `1UL` is *essentially unsigned long*.

```
1u << 10u;                      /* Non-compliant */
( uint16_t ) 1u << 10u;         /* Compliant     */
1UL << 10u;                     /* Compliant     */
```

Rule 12.3 The comma operator should not be used

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Use of the comma operator is generally detrimental to the readability of code, and the same effect can usually be achieved by other means.

Example

```
f ( ( 1, 2 ), 3 );   /* Non-compliant - how many parameters? */
```

The following example is non-compliant with this rule and other rules:

```
for ( i = 0, p = &a[ 0 ]; i < N; ++i, ++p )
{
}
```

Rule 12.4 Evaluation of *constant expressions* should not lead to unsigned integer wrap-around

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to expressions that satisfy the *constraints* for a *constant expression*, whether or not they appear in a context that requires a *constant expression*.

If an expression is not evaluated, for example because it appears in the right operand of a logical AND operator whose left operand is always false, then this rule does not apply.

Rationale

Unsigned integer expressions do not strictly overflow, but instead wrap-around. Although there may be good reasons to use modulo arithmetic at run-time, it is less likely for its use to be intentional at compile-time.

Example

The expression associated with a *case* label is required to be a *constant expression*. If an unsigned wrap-around occurs during evaluation of a *case* expression, it is likely to be unintentional. On a machine with a 16-bit *int* type, any value of `BASE` greater than or equal to 65 024 would result in wrap-around in the following example:

```
#define BASE 65024u

switch ( x )
{
  case BASE + 0u:
    f ( );
    break;
  case BASE + 1u:
    g ( );
    break;
  case BASE + 512u:   /* Non-compliant - wraps to 0 */
    h ( );
    break;
}
```

The controlling expression of a *#if* or *#elif* preprocessor directive is required to be a *constant expression*.

```
#if 1u + ( 0u - 10u )   /* Non-compliant as ( 0u - 10u ) wraps */
```

In this example, the expression `DELAY + WIDTH` has the value 70 000 but this will wrap-around to 4 464 on a machine with a 16-bit *int* type.

```
#define DELAY 10000u
#define WIDTH 60000u

void fixed_pulse ( void )
{
  uint16_t off_time16 = DELAY + WIDTH;   /* Non-compliant */
}
```

This rule does not apply to the expression `c + 1` in the following compliant example as it accesses an object and therefore does not satisfy the *constraints* for a *constant expression*:

```
const uint16_t c = 0xffffu;

void f ( void )
{
  uint16_t y = c + 1u;   /* Compliant */
}
```

In the following example, the sub-expression `( 0u - 1u )` leads to unsigned integer wrap-around. In the initialization of `x`, the sub-expression is not evaluated and the expression is therefore compliant. However, in the initialization of `y`, it may be evaluated and the expression is therefore non-compliant.

```
bool_t b;

void g ( void )
{
  uint16_t x = ( 0u == 0u ) ? 0u : ( 0u - 1u );   /* Compliant     */
  uint16_t y = b ? 0u : ( 0u - 1u );              /* Non-compliant */

}
```

8.13 Side effects

Rule 13.1 *Initializer lists* shall not contain *persistent side effects*

C99 [Unspecified 17, 22]

Category	Required
Analysis	Undecidable, System
Applies to	C99

Rationale

C90 *constrains* the initializers for automatic objects with aggregate types to contain only *constant expressions*. However, C99 permits automatic aggregate initializers to contain expressions that are evaluated at run-time. It also permits compound literals which behave as anonymous initialized objects. The order in which *side effects* occur during evaluation of the expressions in an *initializer list* is unspecified and the behaviour of the initialization is therefore unpredictable if those *side effects* are *persistent*.

Example

```
volatile uint16_t v1;

void f ( void )
{
  /* Non-compliant - volatile access is persistent side effect */
  uint16_t a[ 2 ] = { v1, 0 };
}

void g ( uint16_t x, uint16_t y )
{
  /* Compliant - no side effects                                */
  uint16_t a[ 2 ] = { x + y, x - y };
}

uint16_t x = 0u;

extern void p ( uint16_t a[ 2 ] );

void h ( void )
{
  /* Non-compliant - two side effects                           */
  p ( ( uint16_t[ 2 ] ) { x++, x++ } );
}
```

See also

Rule 13.2

Rule 13.2 The value of an expression and its *persistent side effects* shall be the same under all permitted evaluation orders

C90 [Unspecified 7–9; Undefined 18], C99 [Unspecified 15–18; Undefined 32]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

Between any two adjacent sequence points or within any full expression:

1. No object shall be modified more than once;
2. No object shall be both modified and read unless any such read of the object's value contributes towards computing the value to be stored into the object;
3. There shall be no more than one modification access with *volatile*-qualified type;
4. There shall be no more than one read access with *volatile*-qualified type.

Note: An object might be accessed indirectly, by means of a pointer or a called function, as well as being accessed directly by the expression.

Note: This Amplification is intentionally stricter than the headline of the rule. As a result, expressions such as:

```
x = x = 0;
```

are not permitted by this rule even though the value and the *persistent side effects*, provided that `x` is not *volatile*, are independent of the order of evaluation or *side effects*.

Sequence points are summarized in Annex C of both the C90 and C99 standards. The sequence points in C90 are a subset of those in C99.

Full expressions are defined in Section 6.6 of the C90 standard and Section 6.8 of the C99 standard.

Rationale

The Standard gives considerable flexibility to compilers when evaluating expressions. Most operators can have their operands evaluated in any order. The main exceptions are:

- Logical AND `&&` in which the second operand is evaluated only if the first operand evaluates to non-zero;
- Logical OR `||` in which the second operand is evaluated only if the first operand evaluates to zero;
- The conditional operator `?:` in which the first operand is always evaluated and then either the second or third operand is evaluated;
- The `,` operator in which the first operand is evaluated and then the second operand is evaluated.

Note: The presence of parentheses may alter the order in which operators are applied. However, this does not affect the order of evaluation of the lowest-level operands, which may be evaluated in any order.

Many of the common instances of the unpredictable behaviour associated with expression evaluation can be avoided by following the advice given by Rule 13.3 and Rule 13.4.

Examples

When the `COPY_ELEMENT` macro is invoked in this non-compliant example, `i` is read twice and modified twice. It is unspecified whether the order of operations on `i` is:

- Read, modify, read, modify, or
- Read, read, modify, modify.

```
#define COPY_ELEMENT ( index ) ( a[( index )] = b[( index )] )

COPY_ELEMENT ( i++ );
```

In this non-compliant example the order in which `v1` and `v2` are read is unspecified.

```
extern volatile uint16_t v1, v2;
uint16_t t;

t = v1 + v2;
```

In this compliant example `PORT` is read and modified.

```
extern volatile uint8_t PORT;

PORT = PORT & 0x80u;
```

The order of evaluation of function arguments is unspecified as is the order in which *side effects* occur as shown in this non-compliant example.

```
uint16_t i = 0;

/*
 * Unspecified whether this call is equivalent to:
 *      f ( 0, 0 )
 * or   f ( 0, 1 )
 */
f ( i++, i );
```

The relative order of evaluation of a function designator and function arguments is unspecified. In this non-compliant example, if the call to `g` modifies `p` then it is unspecified whether the function designator `p->f` uses the value of `p` prior to the call of `g` or after it.

```
p->f ( g ( &p ) );
```

See also

Dir 4.9, Rule 13.1, Rule 13.3, Rule 13.4

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

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15; Undefined 32]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A function call is considered to be a *side effect* for the purposes of this rule.

All sub-expressions of the full expression are treated as if they were evaluated for the purposes of this rule, even if specified as not being evaluated by The Standard.

Rationale

The use of increment and decrement operators in combination with other operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement with the potential for undefined behaviour (covered by Rule 13.2).

It is clearer to use these operations in isolation from any other operators.

Example

The expression:

```
u8a = u8b++
```

is non-compliant. The non-compliant expression statement:

```
u8a = ++u8b + u8c--;
```

is clearer when written as the following sequence:

```
++u8b;
u8a = u8b + u8c;
u8c--;
```

The following are all compliant because the only *side effect* in each expression is caused by the increment or decrement operator.

```
x++;
a[ i ]++;
b.x++;
c->x++;
++( *p );
*p++;
( *p )++;
```

The following are all non-compliant because they contain a function call as well as an increment or decrement operator:

```
if ( ( f ( ) + --u8a ) == 0u )
{
}

g ( u8b++ );
```

The following are all non-compliant even though the sub-expression containing the increment or decrement operator or some other *side effect* is not evaluated:

```
u8a = ( 1u == 1u ) ? 0u : u8b++;

if ( u8a++ == ( ( 1u == 1u ) ? 0u : f ( ) ) )
{
}
```

See also

Rule 13.2

Rule 13.4 The result of an assignment operator should not be *used*

C90 [Unspecified 7, 8; Undefined 18], C99 [Unspecified 15, 18; Undefined 32] [Koenig 6]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies even if the expression containing the assignment operator is not evaluated.

Rationale

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;
- It introduces additional *side effects* into a statement making it more difficult to avoid the undefined behaviour covered by Rule 13.2.

Example

```
x = y;                          /* Compliant                              */
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               */
a = b = c = 0;
```

See also

Rule 13.2

Rule 13.5	The right hand operand of a logical `&&` or `||` operator shall not contain *persistent side effects*
Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

The evaluation of the right-hand operand of the `&&` and `||` operators is conditional on the value of the left-hand operand. If the right-hand operand contains *side effects* then those *side effects* may or may not occur which may be contrary to programmer expectations.

If evaluation of the right-hand operand would produce *side effects* which are not *persistent* at the point in the program where the expression occurs then it does not matter whether the right-hand operand is evaluated or not.

The term *persistent side effect* is defined in Appendix J.

Example

```
uint16_t f ( uint16_t y )
{
  /* These side effects are not persistent as seen by the caller */
  uint16_t temp = y;

  temp = y + 0x8080U;

  return temp;
}
```

```
uint16_t h ( uint16_t y )
{
  static uint16_t temp = 0;

  /* This side effect is persistent                                  */
  temp = y + temp;

  return temp;
}

void g ( void )
{
  /* Compliant - f ( ) has no persistent side effects                */
  if ( ishigh && ( a == f ( x ) ) )
  {
  }

  /* Non-compliant - h ( ) has a persistent side effect              */
  if ( ishigh && ( a == h ( x ) ) )
  {
  }
}

volatile uint16_t v;
         uint16_t x;

/* Non-compliant - access to volatile v is persistent */
if ( ( x == 0u ) || ( v == 1u ) )
{
}

/* Non-compliant if fp points to a function with persistent side effects */
( fp != NULL ) && ( *fp ) ( 0 );
```

Rule 13.6 The operand of the *sizeof* operator shall not contain any expression which has potential *side effects*

C99 [Unspecified 21]

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

Any expressions appearing in the operand of a *sizeof* operator are not normally evaluated. This rule mandates that the evaluation of any such expression shall not contain *side effects*, whether or not it is actually evaluated.

A function call is considered to be a *side effect* for the purposes of this rule.

Rationale

The operand of a *sizeof* operator may be either an expression or may specify a type. If the operand contains an expression, a possible programming error is to expect that expression to be evaluated when it is actually not evaluated in most circumstances.

The C90 standard states that expressions appearing in the operand are not evaluated at run-time.

In C99, expressions appearing in the operand are usually not evaluated at run-time. However, if the operand contains a variable-length array type then the array size expression will be evaluated if

necessary. If the result can be determined without evaluating the array size expression then it is unspecified whether it is evaluated or not.

Exception

An expression of the form `sizeof ( v )`, where `v` is an *lvalue* with a *volatile* qualified type that is not a variable-length array, is permitted.

Example

```
volatile int32_t i;
         int32_t j;
         size_t  s;

s = sizeof ( j );         /* Compliant             */
s = sizeof ( j++ );       /* Non-compliant         */
s = sizeof ( i );         /* Compliant - exception */
s = sizeof ( int32_t );   /* Compliant             */
```

In this example the final *sizeof* expression illustrates how it is possible for a variable-length array size expression to have no effect on the size of the type. The operand is the type "array of *n* pointers to function with parameter type array of `v int32_t`". Because the operand has variable-length array type, it is evaluated. However, the size of the array of *n* function pointers is unaffected by the parameter list for those function pointer types. Therefore, the *volatile*-qualified object `v` may or may not be evaluated and its *side effects* may or may not occur.

```
volatile uint32_t v;

void f ( int32_t n )
{
  size_t s;

  s = sizeof ( int32_t[ n ] );                        /* Compliant     */
  s = sizeof ( int32_t[ n++ ] );                      /* Non-compliant */
  s = sizeof ( void ( *[ n ] ) ( int32_t a[ v ] ) );  /* Non-compliant */
}
```

See also

Rule 18.8

8.14 Control statement expressions

The term *loop counter* is used by some of the rules in this section. A *loop counter* is defined as an object, array element or member of a structure or union which satisfies the following:

1. It has a scalar type;
2. Its value varies monotonically on each iteration of a given instance of a loop; and
3. It is involved in a decision to exit the loop.

Note: the second condition means that the value of the *loop counter* must change on each iteration of the loop and it must always change in the same direction for a given instance of the loop. It may, however, change in different directions on different instances, for example sometimes reading the elements of an array backwards and sometimes reading them forwards.

According to this definition, a loop need not have just one *loop counter*: it is possible for a loop to have no *loop counter* or to have more than one. See Rule 14.2 for further restrictions on *loop counters* in *for* loops.

The following code fragments show examples of loops and their corresponding *loop counters*.

In this loop, `i` is a *loop counter* because it has a scalar type, varies monotonically (increasing) and is involved in the loop termination condition.

```
for ( uint16_t i = 0; a[ i ] < 10; ++i )
{
}
```

The following loop has no *loop counter*. The object `count` has scalar type and varies monotonically but is not involved in the termination condition.

```
extern volatile bool_t b;
uint16_t  count = 0;

while ( b )
{
  count = count + 1U;
}
```

In the following code, both `i` and `sum` are scalar and monotonically varying (decreasing and increasing respectively). However `sum` is not a *loop counter* because it is not involved in the decision to exit the loop.

```
uint16_t sum = 0;

for ( uint16_t i = 10U; i != 0U; --i )
{
  sum += i;
}
```

In the following loop, `p` is a *loop counter*. It is not involved in the loop control expression but it is involved in a decision to exit the loop by means of the *break* statement.

```
extern volatile bool_t b;

extern char *p;

do
{
  if ( *p == '\0' )
  {
    break;
  }

  ++p;
} while ( b );
```

The *loop counter* in the following example is `p->count`.

```
struct s
{
  uint16_t count;
  uint16_t a[ 10 ];
};

extern struct s *p;

for ( p->count = 0U; p->count < 10U; ++( p->count ) )
{
  p->a[ p->count ] = 0U;
}
```

Rule 14.1	A *loop counter* shall not have *essentially floating* type
Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

When using a floating-point *loop counter*, accumulation of rounding errors may result in a mismatch between the expected and actual number of iterations. This can happen when a loop step that is not a power of the floating-point radix is rounded to a value that can be represented.

Even if a loop with a floating-point *loop counter* appears to behave correctly on one implementation, it may give a different number of iterations on another implementation.

Example

In the following non-compliant example, the value of `counter` is unlikely to be 1 000 at the end of the loop.

```
uint32_t counter = 0u;

for ( float32_t f = 0.0f; f < 1.0f; f += 0.001f )
{
  ++counter;
}
```

The following compliant example uses an integer *loop counter* to guarantee 1 000 iterations and uses it to generate `f` for use within the loop.

```
float32_t f;

for ( uint32_t counter = 0u; counter < 1000u; ++counter )
{
  f = ( float32_t ) counter * 0.001f;
}
```

The following *while* loop is non-compliant because `f` is being used as a *loop counter*.

```
float32_t f = 0.0f;

while ( f < 1.0f )
{
  f += 0.001f;
}
```

The following *while* loop is compliant because `f` is not being used as a *loop counter*.

```
float32_t f;
uint32_t  u32a;

f = read_float32 ( );

do
{
  u32a = read_u32 ( );
  /* f does not change in the loop so cannot be a loop counter */
} while ( ( ( float32_t ) u32a - f ) > 10.0f );
```

See also

Rule 14.2

Rule 14.2 A *for* loop shall be well-formed

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The three clauses of a *for* statement are the:

First clause which

- Shall be empty, or
- Shall assign a value to the *loop counter*, or
- Shall define and initialize the *loop counter* (C99).

Second clause which

- Shall be an expression that has no *persistent side effects*, and
- Shall use the *loop counter* and optionally *loop control flags*, and
- Shall not use any other object that is modified in the *for* loop body.

Third clause which

- Shall be an expression whose only *persistent side effect* is to modify the value of the *loop counter*, and
- Shall not use objects that are modified in the *for* loop body.

There shall only be one *loop counter* in a *for* loop, which shall not be modified in the *for* loop body.

A *loop control flag* is defined as a single identifier denoting an object with *essentially Boolean type* that is used in the *Second clause*.

The behaviour of a *for* loop body includes the behaviour of any functions called within that statement.

Rationale

The *for* statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyse.

Exception

All three clauses may be empty, for example `for ( ; ; )`, so as to allow for infinite loops.

Example

In the following C99 example, `i` is the *loop counter* and `flag` is a *loop control flag*.

```
bool_t flag = false;

for ( int16_t i = 0; ( i < 5 ) && !flag; i++ )
{
  if ( C )
  {
    flag = true;                /* Compliant - allows early termination
                                 * of loop                                */
  }

  i = i + 3;                    /* Non-compliant - altering the loop
                                 * counter                                */
}
```

See also

Rule 14.1, Rule 14.3, Rule 14.4

Rule 14.3 Controlling expressions shall not be invariant

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

This rule applies to:

- Controlling expressions of *if*, *while*, *for*, *do ... while* and *switch* statements;
- The first operand of the `?:` operator.

Rationale

If a controlling expression has an invariant value, it is possible that there is a programming error. Any code that cannot be reached due to the presence of an invariant expression may be removed by the compiler. This might have the effect of removing defensive code, for instance, from the executable.

Exception

1. Invariants that are used to create infinite loops are permitted.
2. A *do ... while* loop with an *essentially Boolean* controlling expression that evaluates to 0 is permitted.

Example

```
s8a = ( u16a < 0u ) ? 0 : 1;    /* Non-compliant - u16a always >= 0 */

if ( u16a <= 0xffffu )
{
                                /* Non-compliant - always true      */
}
```

```
if ( 2 > 3 )
{
                                  /* Non-compliant - always false     */
}

for ( s8a = 0; s8a < 130; ++s8a )
{
                                  /* Non-compliant - always true      */
}

if ( ( s8a < 10 ) && ( s8a > 20 ) )
{
                                  /* Non-compliant - always false     */
}

if ( ( s8a < 10 ) || ( s8a > 5 ) )
{
                                  /* Non-compliant - always true      */
}

while ( s8a > 10 )
{
  if ( s8a > 5 )
  {
                                  /* Non-compliant - s8a not volatile */
  }
}

while ( true )
{
  /* Compliant by exception 1 */
}

do
{
  /* Compliant by exception 2 */
} while ( 0u == 1u );

const uint8_t numcyl = 4u;

/*
 * Non-compliant - compiler is permitted to assume that numcyl always
 * has value 4
 */
if ( numcyl == 4u )
{
}

const volatile uint8_t numcyl_cal = 4u;

/*
 * Compliant - compiler assumes numcyl_cal may be changed by
 * an external method, e.g. automotive calibration tool, even
 * though the program cannot modify its value
 */
if ( numcyl_cal == 4u )
{
}
```

```
uint16_t n;     /* 10 <= n <= 100 */
uint16_t sum;

sum = 0;

for ( uint16_t i = ( n - 6u ); i < n; ++i )
{
  sum += i;
}

if ( ( sum % 2u ) == 0u )
{
  /*
   * Non-compliant - sum is the sum of 6 consecutive non-negative
   * integers so must be an odd number. The controlling expression
   * of the if statement will always be false.
   */
}
```

See also

Rule 2.1, Rule 14.2

Rule 14.4	The controlling expression of an *if* statement and the controlling expression of an *iteration-statement* shall have *essentially Boolean* type
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The controlling expression of a *for* statement is optional. The rule does not require the expression to be present but **does** require it to have *essentially Boolean type* if it is present.

Rationale

Strong typing requires the controlling expression of an *if* statement or *iteration-statement* to have *essentially Boolean* type.

Example

```
int32_t *p, *q;

while ( p )           /* Non-compliant - p is a pointer */
{
}

while ( q != NULL )   /* Compliant                      */
{
}

while ( true )        /* Compliant                      */
{
}
```

```
extern bool_t flag;

while ( flag )          /* Compliant                    */
{
}

int32_t i;

if ( i )                /* Non-compliant                */
{
}

if ( i != 0 )           /* Compliant                    */
{
}
```

See also

Rule 14.2, Rule 20.8

8.15 Control flow

Rule 15.1	The *goto* statement should not be used
Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

In some cases a total ban on *goto* requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the *goto* they replace. Therefore, if this rule is not followed, the restricted use of *goto* is allowed where that use follows the guidance in Rule 15.2 and Rule 15.3.

See also

Rule 9.1, Rule 15.2, Rule 15.3, Rule 15.4

Rule 15.2	The *goto* statement shall jump to a label declared later in the same function
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

Restricting the use of *goto* so that "back" jumps are prohibited ensures that iteration only occurs if the iteration statements provided by the language are used, helping to minimize visual code complexity.

Example

```
void f ( void )
{
  int32_t j = 0;

L1:
  ++j;

  if ( 10 == j )
  {
    goto L2;          /* Compliant     */
  }

  goto L1;            /* Non-compliant */

L2:
  ++j;
}
```

See also

Rule 15.1, Rule 15.3, Rule 15.4

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
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

For the purposes of this rule, a *switch-clause* that does not consist of a compound statement is treated as if it were a block.

Rationale

Unconstrained use of *goto* can lead to programs that are unstructured and extremely difficult to understand.

Preventing jumps between blocks, or into nested blocks, helps to minimize visual code complexity.

Note: C99 is more restrictive when variably modified types are used. An attempt to make a jump from outside the scope of an identifier with a variably modified type into such a scope results in a *constraint* violation.

Example

```
void f1 ( int32_t a )
{
  if ( a <= 0 )
  {
    goto L2;            /* Non-compliant */
  }

  goto L1;              /* Compliant     */

  if ( a == 0 )
  {
    goto L1;            /* Compliant     */
  }

  goto L2;              /* Non-compliant */

L1:
  if ( a > 0 )
  {
  L2:
    ;
  }
}
```

In the following example, the label `L1` is defined in a block which encloses the block containing the *goto* statement. However, the jump crosses from one *switch-clause* to another and, since *switch-clauses* are treated like blocks for the purposes of this rule, the *goto* statement is non-compliant.

```
switch ( x )
{
  case 0:
    if ( x == y )
    {
      goto L1;
    }
    break;
  case 1:
    y = x;
L1:
    ++x;
    break;
  default:
    break;
}
```

See also

Rule 9.1, Rule 15.1, Rule 15.2, Rule 15.4, Rule 16.1

Rule 15.4	There should be no more than one *break* or *goto* statement used to terminate any iteration statement
Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Restricting the number of exits from a loop helps to minimize visual code complexity. The use of one *break* or *goto* statement allows a single secondary exit path to be created when early loop termination is required.

Example

Both of the following nested loops are compliant as each has a single *break* used for early loop termination.

```
for ( x = 0; x < LIMIT; ++x )
{
  if ( ExitNow ( x ) )
  {
    break;
  }

  for ( y = 0; y < x; ++y )
  {
    if ( ExitNow ( LIMIT - y ) )
    {
      break;
    }
  }
}
```

The following loop is non-compliant as there are multiple *break* and *goto* statements used for early loop termination.

```
for ( x = 0; x < LIMIT; ++x )
{
  if ( BreakNow ( x ) )
  {
    break;
  }
  else if ( GotoNow ( x ) )
  {
    goto EXIT;
  }
  else
  {
    KeepGoing ( x );
  }
}

EXIT:
  ;
```

In the following example, the inner *while* loop is compliant because there is a single *goto* statement that can cause its early termination. However, the outer *while* loop is non-compliant because it can be terminated early either by the *break* statement or by the *goto* statement in the inner *while* loop.

```
while ( x != 0u )
{
  x = calc_new_x ( );

  if ( x == 1u )
  {
    break;
  }

  while ( y != 0u )
  {
    y = calc_new_y ( );

    if ( y == 1u )
    {
      goto L1;
    }
  }
}

L1:
z = x + y;
```

See also

Rule 15.1, Rule 15.2, Rule 15.3

Rule 15.5 A function should have a single point of exit at the end

[IEC 61508-3 Table B.9], [ISO 26262-6 Table 8]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A function should have no more than one *return* statement.

When a *return* statement is used, it should be the final statement in the compound statement that forms the body of the function.

Rationale

A single point of exit is required by IEC 61508 and ISO 26262 as part of the requirements for a modular approach.

Early returns may lead to the unintentional omission of function termination code.

If a function has exit points interspersed with statements that produce *persistent side effects*, it is not easy to determine which *side effects* will occur when the function is executed.

Example

In the following non-compliant code example, early returns are used to validate the function parameters.

```
bool_t f ( uint16_t n, char *p )
{
  if ( n > MAX )
  {
    return false;
  }

  if ( p == NULL )
  {
    return false;
  }

  return true;
}
```

See also

Rule 17.4

Rule 15.6 The body of an *iteration-statement* or a *selection-statement* shall be a *compound-statement*

[Koenig 24]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The body of an *iteration-statement* (*while*, *do ... while* or *for*) or a *selection-statement* (*if*, *else*, *switch*) shall be a *compound-statement*.

Rationale

It is possible for a developer to mistakenly believe that a sequence of statements forms the body of an *iteration-statement* or *selection-statement* by virtue of their indentation. The accidental inclusion of a semi-colon after the controlling expression is a particular danger, leading to a null control statement. Using a *compound-statement* clearly defines which statements actually form the body.

Additionally, it is possible that indentation may lead a developer to associate an *else* statement with the wrong *if*.

Exception

An *if* statement immediately following an *else* need not be contained within a *compound-statement*.

Example

The layout for the *compound-statement* and its enclosing braces are style issues which are not addressed by this document; the style used in the following examples is not mandatory.

Maintenance to the following

```
while ( data_available )
  process_data ( );              /* Non-compliant */
```

could accidentally give

```
while ( data_available )
  process_data ( );              /* Non-compliant */
  service_watchdog ( );
```

where `service_watchdog()` should have been added to the loop body. The use of a *compound-statement* significantly reduces the chance of this happening.

The next example appears to show that `action_2()` is the *else* statement to the first *if*.

```
if ( flag_1 )
  if ( flag_2 )                  /* Non-compliant */
    action_1 ( );                /* Non-compliant */
else
  action_2 ( );                  /* Non-compliant */
```

when the actual behaviour is

```
if ( flag_1 )
{
  if ( flag_2 )
  {
    action_1 ( );
  }
  else
  {
    action_2 ( );
  }
}
```

The use of *compound-statement*s ensures that *if* and *else* associations are clearly defined.

The exception allows the use of *else if*, as shown below

```
if ( flag_1 )
{
  action_1 ( );
}
else if ( flag_2 )               /* Compliant by exception */
{
  action_2 ( );
}
else
{
  ;
}
```

The following example shows how a spurious semi-colon could lead to an error

```
while ( flag );                  /* Non-compliant */
{
  flag = fn ( );
}
```

The following example shows the compliant method of writing a loop with an empty body:

```
while ( !data_available )
{
}
```

Rule 15.7	All *if ... else if* constructs shall be terminated with an *else* statement
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A final *else* statement shall always be provided whenever an *if* statement is followed by a sequence of one or more *else if* constructs. The *else* statement shall contain at least either one *side effect* or a comment.

Rationale

Terminating a sequence of *if ... else if* constructs with an *else* statement is defensive programming and complements the requirement for a *default* clause in a *switch* statement (see Rule 16.5).

The *else* statement is required to have a *side effect* or a comment to ensure that a positive indication is given of the desired behaviour, aiding the code review process.

Note: a final *else* statement is not required for a simple *if* statement.

Example

The following example is non-compliant as there is no explicit indication that no action is to be taken by the terminating *else*.

```
if ( flag_1 )
{
  action_1 ( );
}
else if ( flag_2 )
{
  action_2 ( );
}

/* Non-compliant */
```

The following shows a compliant terminating *else*.

```
else
{
  ;   /* No action required - ; is optional */
}
```

See also

Rule 16.5

8.16 Switch statements

Rule 16.1	All *switch* statements shall be well-formed
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

A *switch* statement shall be considered to be well-formed if it conforms to the subset of C *switch* statements that is specified by the following syntax rules. If a syntax rule given here has the same name as one defined in The Standard then it replaces the standard version for the scope of the *switch* statement; otherwise, all syntax rules given in The Standard are unchanged.

switch-statement:
switch (*switch-expression*) { *case-label-clause-list* *final-default-clause-list* }
switch (*switch-expression*) { *initial-default-clause-list* *case-label-clause-list* }

case-label-clause-list:
case-clause-list
case-label-clause-list *case-clause-list*

case-clause-list:
case-label *switch-clause*
case-label *case-clause-list*

case-label:
case *constant-expression*:

final-default-clause-list:
default: *switch-clause*
case-label *final-default-clause-list*

initial-default-clause-list:
default: *switch-clause*
default: *case-clause-list*

switch-clause:
statement-list$_{opt}$ break;
C90: { *declaration-list*$_{opt}$ *statement-list*opt break; }
C99: { *block-item-list*$_{opt}$ break; }

Except where explicitly permitted by this syntax, the *case* and *default* keywords may not appear anywhere within a *switch* statement body.

Note: some of the restrictions imposed on *switch* statements by this rule are expounded in the rules referenced in the "See also" section. It is therefore possible for code to violate both this rule **and** one of the more specific rules.

Note: the term ***switch label*** is used within the text of the specific *switch* statement rules to denote either a *case* label or a *default* label.

Rationale

The syntax for the *switch* statement in C is not particularly rigorous and can allow complex, unstructured behaviour. This and other rules impose a simple and consistent structure on the *switch* statement.

Example

The remaining rules in this section give examples that are also relevant to this rule.

See also

Rule 15.3, Rule 16.2, Rule 16.3, Rule 16.4, Rule 16.5, Rule 16.6

Rule 16.2 A *switch label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

The Standard permits a *switch label*, *i.e.* a *case* label or *default* label, to be placed before any statement contained in the body of a *switch* statement, potentially leading to unstructured code. In order to prevent this, a *switch label* shall only appear at the outermost level of the compound statement forming the body of a *switch* statement.

Example

```
switch ( x )
{
  case 1:          /* Compliant     */
    if ( flag )
    {
  case 2:          /* Non-compliant */
      x = 1;
    }
    break;
  default:
    break;
}
```

See also

Rule 16.1

Rule 16.3 An unconditional *break* statement shall terminate every *switch-clause*

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

If a developer fails to end a *switch-clause* with a *break* statement, then control flow "falls" into the following *switch-clause* or, if there is no such clause, off the end and into the statement following the *switch* statement. Whilst falling into a following *switch-clause* is sometimes intentional, it is often an error. An unterminated *switch-clause* occurring at the end of a *switch* statement may fall into any *switch-clauses* which are added later.

To ensure that such errors can be detected, the last statement in every *switch-clause* shall be a *break* statement, or if the *switch-clause* is a compound statement, the last statement in the compound statement shall be a *break* statement.

Note: a *switch-clause* is defined as containing at least one statement. Two consecutive labels, *case* or *default*, do not have any intervening statement and are therefore permitted by this rule.

Example

```
switch ( x )
{
  case 0:
    break;          /* Compliant - unconditional break                */
  case 1:           /* Compliant - empty fall through allows a group  */
  case 2:
    break;          /* Compliant                                      */
  case 4:
    a = b;          /* Non-compliant - break omitted                  */
  case 5:
    if ( a == b )
    {
      ++a;
      break;        /* Non-compliant - conditional break              */
    }
  default:
    ;               /* Non-compliant - default must also have a break */
}
```

See also

Rule 16.1

Rule 16.4	Every *switch* statement shall have a *default* label
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The *switch-clause* following the *default* label shall, prior to the terminating *break* statement, contain either:

- A statement, or
- A comment.

Rationale

The requirement for a *default* label is defensive programming. Any statements following the *default* label are intended to take some appropriate action. If no statements follow the label then the comment can be used to explain why no specific action has been taken.

Example

```
int16_t x;

switch ( x )
{
  case 0:
    ++x;
    break;
  case 1:
  case 2:
    break;
                    /* Non-compliant - default label is required      */
}

int16_t x;

switch ( x )
{
  case 0:
    ++x;
    break;
  case 1:
  case 2:
    break;
  default:          /* Compliant - default label is present           */
    errorflag = 1;  /* should be non-empty if possible                */
    break;
}

enum Colours
{ RED, GREEN, BLUE } colour;

switch ( colour )
{
  case RED:
    next = GREEN;
    break;
  case GREEN:
    next = BLUE;
    break;
  case BLUE:
    next = RED;
    break;
                    /* Non-compliant - no default label.
                     * Even though all values of the enumeration are
                     * handled there is no guarantee that colour takes
                     * one of those values                            */
}
```

See also

Rule 2.1, Rule 16.1

Rule 16.5	A *default* label shall appear as either the first or the last *switch label* of a *switch* statement
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

This rule makes it easy to locate the *default* label within a *switch* statement.

Example

```
switch ( x )
{
  default:   /* Compliant - default is the first label               */
  case 0:
    ++x;
    break;
  case 1:
  case 2:
    break;
}

switch ( x )
{
  case 0:
    ++x;
    break;
  default:   /* Non-compliant - default is mixed with the case labels */
    x = 0;
    break;
  case 1:
  case 2:
    break;
}

switch ( x )
{
  case 0:
    ++x;
    break;
  case 1:
  case 2:
    break;
  default:   /* Compliant - default is the final label               */
    x = 0;
    break;
}
```

See also

Rule 15.7, Rule 16.1

Rule 16.6 Every *switch* statement shall have at least two *switch-clauses*

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

A *switch* statement with a single path is redundant and may be indicative of a programming error.

Example

```
switch ( x )
{
  default:   /* Non-compliant - switch is redundant */
    x = 0;
    break;
}

switch ( y )
{
  case 1:
  default:   /* Non-compliant - switch is redundant */
    y = 0;
    break;
}

switch ( z )
{
  case 1:
    z = 2;
    break;
  default:   /* Compliant                           */
    z = 0;
    break;
}
```

See also

Rule 16.1

Rule 16.7	A *switch-expression* shall not have *essentially Boolean type*
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The Standard requires the controlling expression of a *switch* statement to have an integer type. Since the type that is used to implement Boolean values is an integer, it is possible to have a *switch* statement controlled by a Boolean expression. In this instance an *if-else* construct would be more appropriate.

Example

```
switch ( x == 0 )    /* Non-compliant - essentially Boolean             */
{                    /* In this case an "if-else" would be more logical */
  case false:
    y = x;
    break;
  default:
    y = z;
    break;
}
```

8.17 Functions

Rule 17.1	The features of <stdarg.h> shall not be used

C90 [Undefined 45, 70–76], C99 [Undefined 81, 128–135]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

None of *va_list*, *va_arg*, *va_start*, *va_end* and, for C99, *va_copy* shall be used.

Rationale

The Standard lists many instances of undefined behaviour associated with the features of <stdarg.h>, including:

- *va_end* not being used prior to end of a function in which *va_start* was used;
- *va_arg* being used in different functions on the same *va_list*;
- The type of an argument not being compatible with the type specified to *va_arg*.

Example

```
#include <stdarg.h>

void h ( va_list ap )            /* Non-compliant                    */
{
  double y;

  y = va_arg ( ap, double );     /* Non-compliant                    */
}

void f ( uint16_t n, ... )
{
  uint32_t x;

  va_list  ap;                   /* Non-compliant                     */

  va_start ( ap, n );            /* Non-compliant                     */
  x = va_arg ( ap, uint32_t );   /* Non-compliant                     */

  h ( ap );

  /* undefined - ap is indeterminate because va_arg used in h ( )     */
  x = va_arg ( ap, uint32_t );   /* Non-compliant                     */

  /* undefined - returns without using va_end ( )                     */
}

void g ( void )
{
  /* undefined - uint32_t:double type mismatch when f uses va_arg ( ) */
  f ( 1, 2.0, 3.0 );
}
```

Rule 17.2 Functions shall not call themselves, either directly or indirectly

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

Recursion carries with it the danger of exceeding available stack space, which can lead to a serious failure. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Rule 17.3 A function shall not be declared implicitly

C90 [Undefined 6, 22, 23]

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C90

Rationale

Provided that a function call is made in the presence of a prototype, a *constraint* ensures that the number of arguments matches the number of parameters and that each argument can be assigned to its corresponding parameter.

If a function is declared implicitly, a C90 compiler will assume that the function has a return type of *int*. Since an implicit function declaration does not provide a prototype, a compiler will have no information about the number of function parameters and their types. Inappropriate type conversions may result in passing the arguments and assigning the return value, as well as other undefined behaviour.

Example

If the function `power` is declared as:

```
extern double power ( double d, int n );
```

but the declaration is **not** visible in the following code then undefined behaviour will occur.

```
void func ( void )
{
  /* Non-compliant - return type and both argument types incorrect */
  double sq1 = power ( 1, 2.0 );
}
```

See also

Rule 8.2, Rule 8.4

Rule 17.4 All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression

C90 [Undefined 43], C99 [Undefined 82]

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The expression given to the *return* statement provides the value that the function returns. If a non-*void* function does not return a value but the calling function uses the returned value, the behaviour is undefined. This can be avoided by ensuring that, in a non-*void* function:

- Every *return* statement has an expression, and
- Control cannot reach the end of the function without encountering a *return* statement.

Note: C99 *constrains* every *return* statement in a non-*void* function to return a value.

Example

```
int32_t absolute ( int32_t v )
{
  if ( v < 0 )
  {
    return v;
  }

  /*
   * Non-compliant - control can reach this point without
   * returning a value
   */
}

uint16_t lookup ( uint16_t v )
{
  if ( ( v < V_MIN ) || ( v > V_MAX ) )
  {
    /* Non-compliant - no value returned. Constraint in C99 */
    return;
  }

  return table[ v ];
}
```

See also

Rule 15.5

Rule 17.5	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
Category	Advisory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

If a parameter is declared as an array with a specified size, the corresponding argument in each function call should point into an object that has at least as many elements as the array.

Rationale

The use of an array declarator for a function parameter specifies the function interface more clearly than using a pointer. The minimum number of elements expected by the function is explicitly stated, whereas this is not possible with a pointer.

A function parameter array declarator which does not specify a size is assumed to indicate that the function can handle an array of any size. In such cases, it is expected that the array size will be communicated by some other means, for example by being passed as another parameter, or by terminating the array with a sentinel value.

The use of an array bound is recommended as it allows out-of-bounds checking to be implemented within the function body and extra checks on parameter passing. It is legal in C to pass an array of the incorrect size to a parameter with a specified size, which can lead to unexpected behaviour.

Example

```
/*
 * Intent is that function does not access outside the range
 * array1[ 0 ] .. array1[ 3 ]
 */
void fn1 ( int32_t array1[ 4 ] );

/* Intent is that function handles arrays of any size           */
void fn2 ( int32_t array2[ ] );

void fn ( int32_t *ptr )
{
  int32_t arr3[ 3 ] = { 1, 2, 3 };
  int32_t arr4[ 4 ] = { 0, 1, 2, 3 };

  /* Compliant - size of array matches the prototype            */
  fn1 ( arr4 );

  /* Non-compliant - size of array does not match prototype     */
  fn1 ( arr3 );

  /* Compliant only if ptr points to at least 4 elements        */
  fn1 ( ptr );

  /* Compliant                                                  */
  fn2 ( arr4 );

  /* Compliant                                                  */
  fn2 ( ptr );
}
```

See also

Rule 17.6

Rule 17.6 The declaration of an array parameter shall not contain the *static* keyword between the **[ ]**

C99 [Undefined 71]

Category	Mandatory
Analysis	Decidable, Single Translation Unit
Applies to	C99

Rationale

The C99 language standard provides a mechanism for the programmer to inform the compiler that an array parameter contains a specified minimum number of elements. Some compilers are able to take advantage of this information to generate more efficient code for some types of processor.

If the guarantee made by the programmer is not honoured, and the number of elements is less than the minimum specified, the behaviour is undefined.

The processors used in typical embedded applications are unlikely to provide the facilities required to take advantage of the additional information provided by the programmer. The risk of the program failing to meet the guaranteed minimum number of elements outweighs any potential performance increase.

Example

There is no use of this C99 language feature that is compliant with this rule. The examples show some of the undefined behaviour that can arise from its use.

```
/* Non-compliant - uses static in array declarator */
uint16_t total ( uint16_t n, uint16_t a[ static 20 ] )
{
  uint16_t i;

  uint16_t sum = 0U;

  /* Undefined behaviour if a has fewer than 20 elements */
  for ( i = 0U; i < n; ++i )
  {
    sum = sum + a[ i ];
  }

  return sum;
}

extern uint16_t v1[ 10 ];
extern uint16_t v2[ 20 ];

void g ( void )
{
  uint16_t x;

  x = total ( 10U, v1 );   /* Undefined - v1 has 10 elements but needs
                            *             at least 20                   */
  x = total ( 20U, v2 );   /* Defined but non-compliant                 */
}
```

See also

Rule 17.5

Rule 17.7	The value returned by a function having non-*void* return type shall be *used*
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

It is possible to call a function without using the return value, which may be an error. If the return value of a function is intended not to be *used* explicitly, it should be cast to the *void* type. This has the effect of using the value without violating Rule 2.2.

Example

```
uint16_t func ( uint16_t para1 )
{
  return para1;
}
```

```
uint16_t x;

void discarded ( uint16_t para2 )
{
  func ( para2 );            /* Non-compliant - value discarded */
  ( void ) func ( para2 );   /* Compliant                       */
  x = func ( para2 );        /* Compliant                       */
}
```

See also

Dir 4.7, Rule 2.2

Rule 17.8 A function parameter should not be modified

Category	Advisory
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

A function parameter behaves in the same manner as an object that has automatic storage duration. While the C language permits parameters to be modified, such use can be confusing and conflict with programmer expectations. It may be less confusing to copy the parameter to an automatic object and modify that copy. With a modern compiler, this will not usually result in any storage or execution time penalty.

Programmers who are unfamiliar with C, but who are used to other languages, may modify a parameter believing that the effects of the modification will be felt in the calling function.

Example

```
int16_t glob = 0;

void proc ( int16_t para )
{
  para = glob;               /* Non-compliant */
}

void f ( char *p, char *q )
{
  p = q;                     /* Non-compliant */
  *p = *q;                   /* Compliant     */
}
```

8.18 Pointers and arrays

Rule 18.1 A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand

C90 [Undefined 30], C99 [Undefined 43, 44, 46, 59]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

Creation of a pointer to one beyond the end of the array is well-defined by The Standard and is permitted by this rule. Dereferencing a pointer to one beyond the end of an array results in undefined behaviour and is forbidden by this rule.

This rule applies to all forms of array indexing:

```
integer_expression + pointer_expression
pointer_expression + integer_expression
pointer_expression - integer_expression
pointer_expression += integer_expression
pointer_expression -= integer_expression
++pointer_expression
pointer_expression++
--pointer_expression
pointer_expression--
pointer_expression [ integer_expression ]
integer_expression [ pointer_expression ]
```

Note: a subarray is also an array.

Note: for purposes of pointer arithmetic, The Standard treats an object that is not a member of an array as if it were an array with a single element (C90 Section 6.3.6, C99 Section 6.5.6).

Rationale

Although some compilers may be able to determine at compile time that an array boundary has been exceeded, no checks are generally made at run-time for invalid array subscripts. Using an invalid array subscript can lead to erroneous behaviour of the program.

Run-time derived array subscript values are of the most concern since they cannot easily be checked by static analysis or manual review. Code of a defensive programming nature should, where possible and practicable, be provided to check such subscript values against valid ones and, if required, appropriate action be taken.

It is undefined behaviour if the result obtained from one of the above expressions is not a pointer to an element of the array pointed to by `pointer_expression` or an element one beyond the end of that array. See C90 Section 6.3.6 and C99 Section 6.5.6 for further information.

Multi-dimensional arrays are “arrays of arrays”. This rule does not allow pointer arithmetic that results in the pointer addressing a different subarray. Array subscripting over “internal” boundaries shall not be used, as such behaviour is undefined.

Example

The use of the + operator will also violate Rule 18.4.

```
int32_t f1 ( int32_t * const a1, int32_t a2[ 10 ] )
{
  int32_t *p = &a1[ 3 ];        /* Compliant/non-compliant depending on
                                 * the value of a1                        */
  return *( a2 + 9 );           /* Compliant                              */
}

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        */

int32_t *p3 = &c[ 11 ];        /* Non-compliant - undefined, points to
                                *                 two beyond              */
data = *p2;                    /* Non-compliant - undefined, dereference
                                *                 one beyond              */

data = f1 ( &b, c );
data = f1 ( c, c );

p1++;                          /* Compliant                               */
c[ -1 ] = 0;                   /* Non-compliant - undefined, array
                                *                 bounds exceeded         */
data = c[ 10 ];                /* Non-compliant - undefined, dereference
                                *                 of address one beyond   */
data = *( &data + 0 );         /* Compliant     - C treats data as an
                                *                 array of size 1         */

d[ 3 ][ 1 ] = 0;               /* Compliant                               */
data = *( *( d + 3 ) + 1 );    /* Compliant                               */
data = d[ 2 ][ 3 ];            /* Non-compliant - undefined, internal
                                *                 boundary exceeded       */

p1 = d[ 1 ];                   /* Compliant                               */
data = p1[ 1 ];                /* Compliant     - p1 addresses an array
                                *                 of size 2               */
}
```

The following example illustrates pointer arithmetic applied to members of a structure. Because each member is an object in its own right, this rule prevents the use of pointer arithmetic to move from one member to the next. However, it does not prevent arithmetic on a pointer to a member provided that the resulting pointer remains within the bounds of the member object.

```
struct
{
  uint16_t x;
  uint16_t y;
  uint16_t z;
  uint16_t a[ 10 ];
} s;

uint16_t *p;
```

```
void f3 ( void )
{
  p = &s.x;
  ++p;          /* Compliant     - p points one beyond s.x                */
  p[ 0 ] = 1;   /* Non-compliant - undefined, dereference of address one
                 *                 beyond s.x which is not necessarily
                 *                 the same as s.y                        */
  p[ 1 ] = 2;   /* Non-compliant - undefined                              */

p = &s.a[ 0 ]; /* Compliant     - p points into s.a                      */
p = p + 8;     /* Compliant     - p still points into s.a                */
p = p + 3;     /* Non-compliant - undefined, p points more than one
                *                 beyond s.a                             */
}
```

See also

Dir 4.1, Rule 18.4

Rule 18.2 Subtraction between pointers shall only be applied to pointers that address elements of the same array

C90 [Undefined 31], C99 [Undefined 45]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

This rule applies to expressions of the form:

```
pointer_expression_1 - pointer_expression_2
```

It is undefined behaviour if `pointer_expression_1` and `pointer_expression_2` do not point to elements of the same array or the element one beyond the end of that array.

Example

```
#include <stddef.h>

void f1 ( int32_t *ptr )
{
  int32_t   a1[ 10 ];
  int32_t   a2[ 10 ];
  int32_t  *p1 = &a1[ 1 ];
  int32_t  *p2 = &a2[ 10 ];
  ptrdiff_t diff;

  diff = p1 - a1;           /* Compliant     */
  diff = p2 - a2;           /* Compliant     */
  diff = p1 - p2;           /* Non-compliant */
  diff = ptr - p1;          /* Non-compliant */
}
```

See also

Dir 4.1, Rule 18.4

Rule 18.3 The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object

C90 [Undefined 33], C99 [Undefined 50]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

Attempting to make comparisons between pointers will produce undefined behaviour if the two pointers do not point to the same object.

Note: it is permissible to address the next element beyond the end of an array, but accessing this element is not allowed.

Example

```
void f1 ( void )
{
  int32_t  a1[ 10 ];
  int32_t  a2[ 10 ];
  int32_t *p1 = a1;

  if ( p1 < a1 )       /* Compliant     */
  {
  }
  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_1.upb )   /* Compliant     */
  {
  }
  if ( &limits_1.lwb > &limits_2.upb )    /* Non-Compliant */
  {
  }
}
```

See also

Dir 4.1

Rule 18.4	The +, -, += and -= operators should not be applied to an expression of pointer type
Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Array indexing using the array subscript syntax, `ptr[expr]`, is the preferred form of pointer arithmetic because it is often clearer and hence less error prone than pointer manipulation. Any explicitly calculated pointer value has the potential to access unintended or invalid memory addresses. Such behaviour is also possible with array indexing, but the subscript syntax may ease the task of manual review.

Pointer arithmetic in C can be confusing to the novice. The expression `ptr+1` may be mistakenly interpreted as the addition of 1 to the address held in `ptr`. In fact the new memory address depends on the size in bytes of the pointer's target. This misunderstanding can lead to unexpected behaviour if *sizeof* is applied incorrectly.

When used with caution however, pointer manipulation using ++ can in some cases be considered more natural; e.g. sequentially accessing locations during a memory test where it is more convenient to treat the memory space as a contiguous set of locations and the address bounds can be determined at compilation time.

Exception

Subject to Rule 18.2, pointer subtraction between two pointers is allowed.

Example

```
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;       /* Non-compliant                             */
  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 */
    }
  }
}
```

In the following example, Rule 18.1 may also be violated if `p1` does not point to an array with at least six elements and `p2` does not point to an array with at least 4 elements.

```
void fn3 ( uint8_t *p1, uint8_t p2[ ] )
{
  p1++;                   /* Compliant     */
  p1 = p1 + 5;            /* Non-compliant */
  p1[ 5 ] = 0U;           /* Compliant     */

  p2++;                   /* Compliant     */
  p2 = p2 + 3;            /* Non-compliant */
  p2[ 3 ] = 0U;           /* Compliant     */
}

uint8_t a1[ 16 ];
uint8_t a2[ 16 ];
uint8_t data = 0U;

void fn4 ( void )
{
  fn3 ( a1, a2 );
  fn3 ( &data, &a2[ 4 ] );
}
```

See also

Rule 18.1, Rule 18.2

Rule 18.5 Declarations should contain no more than two levels of pointer nesting

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

No more than two *pointer declarators* should be applied consecutively to a type. Any *typedef-name* appearing in a declaration is treated as if it were replaced by the type that it denotes.

Rationale

The use of more than two levels of pointer nesting can seriously impair the ability to understand the behaviour of the code, and should therefore be avoided.

Example

```
typedef int8_t * INTPTR;

void function ( int8_t ** arrPar[ ] )  /* Non-compliant */
{
   int8_t **  obj2;                    /* Compliant     */
   int8_t *** obj3;                    /* Non-compliant */
   INTPTR *   obj4;                    /* Compliant     */
   INTPTR *   const * const obj5;      /* Non-compliant */
   int8_t **  arr[ 10 ];               /* Compliant     */
   int8_t **  ( *parr )[ 10 ];         /* Compliant     */
   int8_t *   ( **pparr )[ 10 ];       /* Compliant     */
}

struct s
{
   int8_t *   s1;                      /* Compliant     */
   int8_t **  s2;                      /* Compliant     */
   int8_t *** s3;                      /* Non-compliant */
};

struct s *   ps1;                      /* Compliant     */
struct s **  ps2;                      /* Compliant     */
struct s *** ps3;                      /* Non-compliant */

int8_t **  (   *pfunc1 )( void );      /* Compliant     */
int8_t **  (  **pfunc2 )( void );      /* Compliant     */
int8_t **  ( ***pfunc3 )( void );      /* Non-compliant */
int8_t *** (  **pfunc4 )( void );      /* Non-compliant */
```

Note:

- `arrPar` is of type pointer to pointer to pointer to `int8_t` because parameters declared with array type are converted to a pointer to the initial element of the array — this is three levels and is non-compliant;
- `arr` is of type array of pointer to pointer to `int8_t` — this is compliant;
- `parr` is of type pointer to array of pointer to pointer to `int8_t` — this is compliant;
- `pparr` is of type pointer to pointer to array of pointer to `int8_t` — this is compliant.

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

C90 [Undefined 9, 26], C99 [Undefined 8, 9, 40]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The address of an object might be copied by means of:

- *Assignment*;
- Memory move or copying functions.

Rationale

The address of an object becomes indeterminate when the lifetime of that object expires. Any use of an indeterminate address results in undefined behaviour.

Example

```
int8_t *func ( void )
{
  int8_t local_auto;

  return &local_auto;      /* Non-compliant - &local_auto is indeterminate
                            *                 when func returns           */
}
```

In the following example, the function `g` stores a copy of its pointer parameter `p`. If `p` **always** points to an object with static storage duration then the code is compliant with this rule. However, in the example given, `p` does point to an object with automatic storage duration. In such a case, copying the parameter `p` is non-compliant.

```
uint16_t *sp;

void g ( uint16_t *p )
{
  sp = p;                  /* Non-compliant - address of f's parameter u
                            *                 copied to static sp         */
}

void f ( uint16_t u )
{
  g ( &u );
}

void h ( void )
{
  static uint16_t *q;

  uint16_t x = 0u;

  q = &x;                  /* Non-compliant - &x stored in object with
                            *                 greater lifetime            */
}
```

Rule 18.7 Flexible array members shall not be declared

C99 [Undefined 59]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Flexible array members are most likely to be used in conjunction with dynamic memory allocation which is banned by Dir 4.12 and Rule 21.3.

The presence of flexible array members modifies the behaviour of the *sizeof* operator in ways that might not be expected by a programmer. The assignment of a structure that contains a flexible array member to another structure of the same type may not behave in the expected manner as it copies only those elements up to but not including the start of the flexible array member.

Example

```
#include <stdlib.h>

struct s
{
  uint16_t len;
  uint32_t data[ ];   /* Non-compliant - flexible array member */
} str;

struct s *copy ( struct s *s1 )
{
  struct s *s2;

  /* Omit malloc ( ) return check for brevity */
  s2 = malloc ( sizeof ( struct s ) + ( s1->len * sizeof ( uint32_t ) ) );

  *s2 = *s1;          /* Only copies s1->len */

  return s2;
}
```

See also

Dir 4.12, Rule 21.3

Rule 18.8 Variable-length array types shall not be used

C99 [Unspecified 21; Undefined 69, 70]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Rationale

Variable-length array types are specified when the size of an array declared in a block or a function prototype is not an *integer constant expression*. They are typically implemented as a variable size object stored on the stack. Their use can therefore make it impossible to determine statically the amount of memory that must be reserved for a stack.

If the size of a variable-length array is negative or zero, the behaviour is undefined.

If a variable-length array is used in a context in which it is required to be compatible with another array type, possibly itself variable-length, then the size of the array types shall be identical. Further, all sizes shall evaluate to positive integers. If these requirements are not met, the behaviour is undefined.

If a variable-length array type is used in the operand of a *sizeof* operator, under some circumstances it is unspecified whether the array size expression is evaluated or not.

Each instance of a variable-length array type has its size fixed at the start of its lifetime. This gives rise to behaviour that might be confusing, for example:

```
void f ( void )
{
  uint16_t n = 5;

  typedef uint16_t Vector[ n ];  /* An array type with 5 elements     */

  n = 7;

  Vector   a1;                    /* An array type with 5 elements     */

  uint16_t a2[ n ];               /* An array type with 7 elements     */
}
```

Example

There is no use of variable-length arrays that is compliant with this rule. The examples show some of the undefined behaviour that can arise from their use.

```
void f ( int16_t n )
{
  uint16_t vla[ n ];            /* Non-compliant - Undefined if n <= 0 */
}

void g ( void )
{
  f ( 0 );                      /* Undefined */
  f ( -1 );                     /* Undefined */
  f ( 10 );                     /* Defined   */
}
```

```
void h ( uint16_t n,
         uint16_t a[ 10 ][ n ] )     /* Non-compliant */
{
  uint16_t ( *p )[ 20 ];

  /* Undefined unless n == 20: incompatible types otherwise */
  p = a;
}
```

See also

Rule 13.6

8.19 Overlapping storage

Rule 19.1 An object shall not be assigned or copied to an overlapping object

C90 [Undefined 34, 55], C99 [Undefined 51, 94]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

The behaviour is undefined when two objects are created which have some overlap in memory and one is assigned or copied to the other.

Exception

The following are permitted because the behaviour is well-defined:

1. Assignment between two objects that overlap exactly and have compatible types (ignoring their type qualifiers)
2. Copying between objects that overlap partially or completely using The Standard Library function `memmove`

Example

This example also violates Rule 19.2 because it uses unions.

```
void fn ( void )
{
  union
  {
    int16_t i;
    int32_t j;
  } a = { 0 }, b = { 1 };

  a.j = a.i;                /* Non-compliant           */
  a = b;                    /* Compliant - exception 1 */
}
```

```
#include <string.h>

int16_t a[ 20 ];

void f ( void )
{
  memcpy ( &a[ 5 ], &a[ 4 ], 2u * sizeof ( a[ 0 ] ) ); /* Non-compliant */
}

void g ( void )
{
  int16_t  *p = &a[ 0 ];
  int16_t  *q = &a[ 0 ];

  *p = *q;   /* Compliant - exception 1 */
}
```

See also

Rule 19.2

Rule 19.2 The *union* keyword should not be used

C90 [Undefined 39, 40; Implementation 27], C99 [Unspecified 10; Undefined 61, 62]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

A union member can be written and the same member can then be read back in a well-defined manner.

However, if a union member is written and then a different union member is read back, the behaviour depends on the relative sizes of the members:

- If the member read is wider than the member written then the value is unspecified;
- Otherwise, the value is implementation-defined.

The Standard permits the bytes of a union member to be accessed by means of another member whose type is array of *unsigned char*. However, since it is possible to access bytes with unspecified values, unions should not be used.

If this rule is not followed, the kinds of behaviour that need to be determined are:

- Padding — how much padding is inserted at the end of the union;
- Alignment — how are members of any structures within the union aligned;
- Endianness — is the most significant byte of a word stored at the lowest or highest memory address;
- Bit-order — how are bits numbered within bytes and how are bits allocated to bit fields.

Example

In this non-compliant example, a 16-bit value is stored into a union but a 32-bit value is read back resulting in an unspecified value being returned.

```
uint32_t zext ( uint16_t s )
{
  union
  {
    uint32_t ul;
    uint16_t us;
  } tmp;

  tmp.us = s;
  return tmp.ul;   /* unspecified value */
}
```

See also

Rule 19.1

8.20 Preprocessing directives

Rule 20.1 *#include* directives should only be preceded by preprocessor directives or comments

C90 [Undefined 56], C99 [Undefined 96, 97]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The rule shall be applied to the contents of a file before preprocessing occurs.

Rationale

To aid code readability, all the *#include* directives in a particular code file should be grouped together near the top of the file.

Additionally, using *#include* to include a standard *header file* within a declaration or definition, or using part of The Standard Library before the inclusion of the related standard *header file* leads to undefined behaviour.

Example

```
/* f.h */
xyz = 0;

/* f.c */

int16_t
#include "f.h"    /* Non-compliant */

/* f1.c */
#define F1_MACRO

#include "f1.h"   /* Compliant     */
#include "f2.h"   /* Compliant     */

int32_t i = 0;

#include "f3.h"   /* Non-compliant */
```

Rule 20.2 The ', " or \ characters and the /* or // character sequences shall not occur in a *header file* name

C90 [Undefined 14], C99 [Undefined 31]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The behaviour is undefined if:

- The ', " or \ characters, or the /* or // character sequences are used between < and > delimiters in a header name preprocessing token;
- The ' or \ characters, or the /* or // character sequences are used between the " delimiters in a header name preprocessing token.

Note: although use of the \ character results in undefined behaviour, many implementations will accept the / character in its place.

Example

```
#include "fi'le.h"   /* Non-compliant */
```

Rule 20.3 The *#include* directive shall be followed by either a <filename> or "filename" sequence

C90 [Undefined 48], C99 [Undefined 85]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies after macro replacement has been performed.

Rationale

The behaviour is undefined if a *#include* directive does not use one of the following forms:

- `#include <filename>`
- `#include "filename"`

Example

```
#include "filename.h"          /* Compliant     */
#include <filename.h>          /* Compliant     */
#include another.h            /* Non-compliant */

#define HEADER "filename.h"
#include HEADER                /* Compliant */
#define FILENAME file2.h
#include FILENAME              /* Non-compliant */

#define BASE "base"
#define EXT  ".ext"
#include BASE EXT              /* Non-compliant - strings are concatenated
                                * after preprocessing     */

#include "./include/cpu.h"     /* Compliant - filename may include a path */
```

Rule 20.4 A macro shall not be defined with the same name as a keyword

C90 [Undefined 56], C99 [Undefined 98]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to all keywords, including those that implement language extensions.

Rationale

Using macros to change the meaning of keywords can be confusing. The behaviour is undefined if a standard header is included while a macro is defined with the same name as a keyword.

Example

The following example is non-compliant because it alters the behaviour of the *int* keyword. Including a standard header in the presence of this macro results in undefined behaviour.

```
#define int some_other_type
#include <stdlib.h>
```

The following example shows that it is non-compliant to redefine the keyword *while* but it is compliant to define a macro that expands to statements.

```
#define while( E ) for ( ; ( E ) ; )  /* Non-compliant - redefined while */
#define unless( E ) if ( ! ( E ) )    /* Compliant                       */

#define seq( S1, S2 ) do { \
   S1; S2; } while ( false )          /* Compliant                       */
#define compound( S ) { S; }          /* Compliant                       */
```

The following example is compliant in C90, but not C99, because *inline* is not a keyword in C90.

```
/* Remove inline if compiling for C90 */
#define inline
```

See also

Rule 21.1

Rule 20.5 *#undef* should not be used

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The use of *#undef* can make it unclear which macros exist at a particular point within a translation unit.

Example

```
#define QUALIFIER volatile

#undef QUALIFIER             /* Non-compliant */

void f ( QUALIFIER int32_t p )
{
  while ( p != 0 )
  {
    ;                        /* Wait...  */
  }
}
```

Rule 20.6 Tokens that look like a preprocessing directive shall not occur within a macro argument

C90 [Undefined 50], C99 [Undefined 87]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

An argument containing sequences of tokens that would otherwise act as preprocessing directives leads to undefined behaviour.

Example

```
#define M( A ) printf ( #A )

#include <stdio.h>

void main ( void )
{
  M (
#ifdef SW            /* Non-compliant */
       "Message 1"
#else                /* Non-compliant */
       "Message 2"
#endif               /* Non-compliant */
    );
}
```

The above may print

```
#ifdef SW "Message 1" #else "Message 2" #endif
```

or

```
"Message 2"
```

or exhibit some other behaviour.

Rule 20.7 Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses

[Koenig 78–81]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

If the expansion of any macro parameter produces a token, or sequence of tokens, that form an expression then that expression, in the fully-expanded macro, shall either:

- Be a parenthesized expression itself; or
- Be enclosed in parentheses.

Note: this does not necessarily require that all macro parameters are parenthesized; it is acceptable for parentheses to be provided in macro arguments.

Rationale

If parentheses are not used, then operator precedence may not give the desired results when macro substitution occurs.

If a macro parameter is not being used as an expression then the parentheses are not necessary because no operators are involved.

Example

In the following non-compliant example,

```
#define M1( x, y ) ( x * y )

r = M1 ( 1 + 2, 3 + 4 );
```

the macro expands to give:

```
r = ( 1 + 2 * 3 + 4 );
```

The expressions `1 + 2` and `3 + 4` are derived from expansion of parameters `x` and `y` respectively, but neither is enclosed in parentheses. The value of the resulting expression is 11, whereas the result 21 might have been expected.

The code could be written in a compliant manner either by parenthesizing the macro arguments, or by writing an alternative version of the macro that inserts parentheses during expansion, for example:

```
r = M1 ( ( 1 + 2 ), ( 3 + 4 ) );    /* Compliant */

#define M2( x, y ) ( ( x ) * ( y ) )

r = M2 ( 1 + 2, 3 + 4 );            /* Compliant */
```

The following example is compliant because the first expansion of `x` is as the operand of the `##` operator, which does not produce an expression. The second expansion of `x` is as an expression which is parenthesized as required.

```
#define M3( x ) a ## x = ( x )

int16_t M3 ( 0 );
```

The following example is compliant because expansion of the parameter `M` as a member name does not produce an expression. Expansion of the parameter `S` produces an expression, with structure or union type, which does require parentheses.

```
#define GET_MEMBER( S, M ) ( S ).M

v = GET_MEMBER ( s1, minval );
```

The following compliant example shows that it is not always necessary to parenthesize every instance of a parameter, although this is often the easiest method of complying with this rule.

```
#define F( X ) G( X )
#define G( Y ) ( ( Y ) + 1 )

int16_t x = F ( 2 );
```

The fully-expanded macro is `( ( 2 ) + 1 )`. Tracing back through macro expansion, the value `2` arises from expansion of parameter `Y` in macro `G` which in turn arises from parameter `X` in macro `F`. Since `2` is parenthesized in the fully-expanded macro, the code is compliant.

See also

Dir 4.9

Rule 20.8 The controlling expression of a *#if* or *#elif* preprocessing directive shall evaluate to 0 or 1

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule does not apply to controlling expressions in preprocessing directives which are not evaluated. Controlling expressions are not evaluated if they are within code that is being excluded and cannot have an effect on whether code is excluded or not.

Rationale

Strong typing requires the controlling expression of conditional inclusion preprocessing directives to have a Boolean value.

Example

```
#define FALSE 0
#define TRUE  1

#if FALSE           /* Compliant                          */
#endif

#if 10              /* Non-compliant                      */
#endif
```

```
#if ! defined ( X ) /* Compliant                                */
#endif


#if A > B           /* Compliant assuming A and B are numeric */
#endif
```

See also

Rule 14.4

Rule 20.9 All identifiers used in the controlling expression of *#if* or *#elif* preprocessing directives shall be *#define*'d before evaluation

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

As well as using a *#define* preprocessor directive, identifiers may effectively be *#define*'d in other, implementation-defined, ways. For example some implementations support:

- Using a compiler command-line option, such as `-D` to allow identifiers to be defined prior to translation;
- Using environment variables to achieve the same effect;
- Pre-defined identifiers provided by the compiler.

Rationale

If an attempt is made to use a macro identifier in a preprocessor directive, and that identifier has not been defined, then the preprocessor will assume that it has a value of zero. This may not meet developer expectations.

Example

The following examples assume that the macro `M` is undefined.

```
#if M == 0          /* Non-compliant                                */
                    /* Does 'M' expand to zero or is it undefined?  */
#endif

#if defined ( M ) /* Compliant - M is not evaluated                 */
#if M == 0        /* Compliant - M is known to be defined           */
                  /* 'M' must expand to zero.                       */
#endif
#endif

/* Compliant - B is only evaluated in ( B == 0 ) if it is defined */
#if defined ( B ) && ( B == 0 )
#endif
```

Rule 20.10 The # and ## preprocessor operators should not be used

C90 [Unspecified 12; Undefined 51, 52], C99 [Unspecified 25; Undefined 3, 88, 89]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. In some cases it is therefore not possible to predict the result of macro expansion.

The use of the ## operator can result in code that is obscure.

Note: Rule 1.3 covers the undefined behaviour that arises if either:

- The result of a # operator is not a valid string literal; or
- The result of a ## operator is not a valid preprocessing token.

See also

Rule 20.11

Rule 20.11 A macro parameter immediately following a # operator shall not immediately be followed by a ## operator

C90 [Unspecified 12], C99 [Unspecified 25]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

The order of evaluation associated with multiple #, multiple ## or a mix of # and ## preprocessor operators is unspecified. The use of # and ## is discouraged by Rule 20.10. In particular, the result of a # operator is a string literal and it is extremely unlikely that pasting this to any other preprocessing token will result in a valid token.

Example

```
#define A( x )      #x          /* Compliant     */
#define B( x, y )   x ## y      /* Compliant     */
#define C( x, y )   #x ## y     /* Non-compliant */
```

See also

Rule 20.10

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

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Rationale

A macro parameter that is used as an operand of a # or ## operator is **not** expanded prior to being used. The same parameter appearing elsewhere in the replacement text **is** expanded. If the macro parameter is itself subject to macro replacement, its use in mixed contexts within a macro replacement may not meet developer expectations.

Example

In the following non-compliant example, the macro parameter `x` is replaced with `AA` which is subject to further macro replacement when not used as the operand of ##.

```
#define AA         0xffff
#define BB( x )    ( x ) + wow ## x   /* Non-compliant */

void f ( void )
{
  int32_t wowAA = 0;

  /* Expands as wowAA = ( 0xffff ) + wowAA; */
  wowAA = BB ( AA );
}
```

In the following compliant example, the macro parameter `X` is not subject to further macro replacement.

```
int32_t speed;
int32_t speed_scale;
int32_t scaled_speed;

#define SCALE( X ) ( ( X ) * X ## _scale )

/* expands to scaled_speed = ( ( speed ) * speed_scale ); */
scaled_speed = SCALE ( speed );
```

Rule 20.13 A line whose first token is # shall be a valid preprocessing directive

Category Required

Analysis Decidable, Single Translation Unit

Applies to C90, C99

Amplification

White-space is permitted between the # and preprocessing tokens.

Rationale

A preprocessor directive may be used to conditionally exclude source code until a corresponding *#else*, *#elif* or *#endif* directive is encountered. A malformed or invalid preprocessing directive contained

within the excluded source code may not be detected by the compiler, possibly leading to the exclusion of more code than was intended.

Requiring all preprocessor directives to be syntactically valid, even when they occur within an excluded block of code, ensures that this cannot happen.

Example

In the following example all the code between the *#ifndef* and *#endif* directives may be excluded if `AAA` is defined. The developer intended that `AAA` be assigned to `x`, but the *#else* directive was entered incorrectly and not diagnosed by the compiler.

```
#define AAA 2

int32_t foo ( void )
{
  int32_t x = 0;

#ifndef AAA
  x = 1;
#else1        /* Non-compliant */
  x = AAA;
#endif

  return x;
}
```

The following example is compliant because the text `#start` appearing in a comment is not a token.

```
/*
#start is not a token in a comment
 */
```

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
Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Rationale

Confusion can arise when blocks of code are included or excluded by the use of conditional compilation directives which are spread over multiple files. Requiring that a *#if* directive be terminated within the same file reduces the visual complexity of the code and the chance that errors will be made during maintenance.

Note: #if directives may be used within included files provided they are terminated within the same file.

Example

```
/* file1.c */
#ifdef A              /* Compliant     */
#include "file1.h"
#endif
/* End of file1.c */
```

```
/* file2.c */
#if 1                   /* Non-compliant */
#include "file2.h"
/* End of file2.c*/

/* file1.h */
#if 1                   /* Compliant     */
#endif
/* End of file1.h */

/* file2.h */
#endif
/* End of file1.h */
```

8.21 Standard libraries

Rule 21.1 *#define* and *#undef* shall not be used on a reserved identifier or reserved macro name

C90 [Undefined 54, 57, 58, 62, 71]
C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to the following:

- Identifiers or macro names beginning with an underscore;
- Identifiers in file scope described in Section 7, "Library", of The Standard;
- Macro names described in Section 7, "Library", of The Standard as being defined in a standard header.

This rule also prohibits the use of *#define* or *#undef* on the identifier *defined* as this results in explicitly undefined behaviour.

This rule does **not** include those identifiers or macro names that are described in the section of the applicable C standard entitled "Future Library Directions".

The Standard states that defining a macro with the same name as:

- A macro defined in a standard header, or
- An identifier with file scope declared in a standard header

is well-defined provided that the header is not included. This rule does not permit such definitions on the grounds that they are likely to cause confusion.

Note: the macro `NDEBUG` is not defined in a standard header and may therefore be *#define*'d.

Rationale

Reserved identifiers and reserved macro names are intended for use by the implementation. Removing or changing the meaning of a reserved macro may result in undefined behaviour.

Example

```
#undef  __LINE__                /* Non-compliant - begins with _        */
#define _GUARD_H 1              /* Non-compliant - begins with _        */
#undef  _BUILTIN_sqrt           /* Non-compliant - the implementation
                                 * may use _BUILTIN_sqrt for other
                                 * purposes, e.g. generating a sqrt
                                 * instruction                          */

#define defined                 /* Non-compliant - reserved identifier  */
#define errno my_errno          /* Non-compliant - library identifier   */
#define isneg( x ) ( ( x ) < 0 ) /* Compliant     - rule doesn't include
                                 *                 future library
                                 *                 directions           */
```

See also

Rule 20.4

Rule 21.2 A reserved identifier or macro name shall not be declared

C90 [Undefined 57, 58, 64, 71], C99 [Undefined 93, 100, 101, 104, 108, 116, 118, 130]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

See the Amplification for Rule 21.1 for a description of the relevant identifiers and macro names.

Rationale

The implementation is permitted to rely on reserved identifiers behaving as described in The Standard and may treat them specially. If reserved identifiers are reused, the program may exhibit undefined behaviour.

Example

In the following non-compliant example, the function *memcpy* is declared explicitly. The compliant method of declaring this function is to include `<string.h>`.

```
/*
 * Include <stddef.h> to define size_t
 */
#include <stddef.h>
extern void *memcpy ( void *restrict s1, const void *restrict s2, size_t n );
```

An implementation is permitted to provide a *function-like macro* definition for each Standard Library function in addition to the library function itself. This feature is often used by compiler writers to generate efficient inline operations in place of the call to a library function. Using a *function-like macro*, the call to a library function can be replaced with a call to a reserved function that is detected by the compiler's code generation phase and replaced with the inline operation. For example, the fragment of `<math.h>` that declares *sqrt* might be written using a *function-like macro* that generates a call to `_BUILTIN_sqrt` which is replaced with an inline `SQRT` instruction on processors that support it:

```
extern double sqrt ( double x );

#define sqrt( x ) ( _BUILTIN_sqrt ( x ) )
```

The following non-compliant code might interfere with the compiler's built-in mechanism for handling `sqrt` and therefore produce undefined behaviour:

```
#define _BUILTIN_sqrt( x ) ( x )            /* Non-compliant           */
#include <math.h>

float64_t x = sqrt ( ( float64_t ) 2.0 );   /* sqrt may not behave as
                                             * defined in The Standard */
```

Rule 21.3 The memory allocation and deallocation functions of `<stdlib.h>` shall not be used

C90 [Unspecified 19; Undefined 9, 91, 92; Implementation 69]
C99 [Unspecified 39, 40; Undefined 8, 9, 168–171; Implementation J.3.12(35)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The identifiers *calloc*, *malloc*, *realloc* and *free* shall not be used and no macro with one of these names shall be expanded.

Rationale

Use of dynamic memory allocation and deallocation routines provided by The Standard Library can lead to undefined behaviour, for example:

- Memory that was not dynamically allocated is subsequently freed;
- A pointer to freed memory is used in any way;
- Accessing allocated memory before storing a value into it.

Note: this rule is a specific instance of Dir 4.12.

See also

Dir 4.12, Rule 18.7, Rule 22.1, Rule 22.2

Rule 21.4 The standard *header file* <setjmp.h> shall not be used

C90 [Unspecified 14; Undefined 64–67]
C99 [Unspecified 32; Undefined 118–121, 173]
[Koenig 74]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

None of the facilities that are specified as being provided by <setjmp.h> shall be used.

Rationale

setjmp and *longjmp* allow the normal function call mechanisms to be bypassed. Their use may lead to undefined and unspecified behaviour.

Rule 21.5 The standard *header file* <signal.h> shall not be used

C90 [Undefined 67–69; Implementation 48–52]
C99 [Undefined 122–127; Implementation J.3.12(12)]
[Koenig 74]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

None of the facilities that are specified as being provided by <signal.h> shall be used.

Rationale

Signal handling contains implementation-defined and undefined behaviour.

Rule 21.6 The Standard Library input/output functions shall not be used

C90 [Unspecified 2–5, 16–18; Undefined 77–89; Implementation 53–68]
C99 [Unspecified 3–6, 34–37; Undefined 138–166, 186; Implementation J.3.12(14–32)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

This rule applies to the functions that are specified as being provided by <stdio.h> and, in C99, their wide-character equivalents specified in Sections 7.24.2 and 7.24.3 of the C99 Standard as being provided by <wchar.h>.

None of these identifiers shall be used and no macro with one of these names shall be expanded.

Rationale

Streams and file I/O have unspecified, undefined and implementation-defined behaviours associated with them.

See also

Rule 22.1, Rule 22.3, Rule 22.4, Rule 22.5, Rule 22.6

Rule 21.7 The *atof*, *atoi*, *atol* and *atoll* functions of `<stdlib.h>` shall not be used

C90 [Undefined 90], C99 [Undefined 113]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The identifiers *atof*, *atoi*, *atol* and, for C99 only *atoll*, shall not be used and no macro with one of these names shall be expanded.

Rationale

These functions have undefined behaviour associated with them when the string cannot be converted.

Rule 21.8 The library functions *abort*, *exit*, *getenv* and *system* of `<stdlib.h>` shall not be used

C90 [Undefined 93; Implementation 70–73]
C99 [Undefined 172, 174, 175; Implementation J.3.12(36–38)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The identifiers *abort*, *exit*, *getenv* and *system* shall not be used and no macro with one of these names shall be expanded.

Rationale

These functions have undefined and implementation-defined behaviours associated with them.

Rule 21.9 The library functions *bsearch* and *qsort* of <stdlib.h> shall not be used

C90 [Unspecified 20, 21], C99 [Unspecified 41, 42; Undefined 176–178]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

The identifiers *bsearch* and *qsort* shall not be used and no macro with one of these names shall be expanded.

Rationale

If the comparison function does not behave consistently when comparing elements, or it modifies any of the elements, the behaviour is undefined.

Note: the unspecified behaviour, which relates to the treatment of elements that compare as equal, can be avoided by ensuring that the comparison function never returns 0. When two elements are otherwise equal, the comparison function could return a value that indicates their relative order in the initial array.

The implementation of *qsort* is likely to be recursive and will therefore place unknown demands on stack resource. This is of concern in embedded systems as the stack is likely to be a fixed, often small, size.

Rule 21.10 The Standard Library time and date functions shall not be used

C90 [Unspecified 22; Undefined 80, 97; Implementation 75, 76]
C99 [Unspecified 43, 44; Undefined 146, 154, 182; Implementation J.3.12(39–42)]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C90, C99

Amplification

None of the facilities that are specified as being provided by <time.h> shall be used.

In C99, the identifier *wcsftime* shall not be used and no macro with this name shall be expanded.

Rationale

The time and date functions have unspecified, undefined and implementation-defined behaviours associated with them.

Rule 21.11 The standard *header file* <tgmath.h> shall not be used

C99 [Undefined 184, 185]

Category	Required
Analysis	Decidable, Single Translation Unit
Applies to	C99

Amplification

None of the facilities that are specified as being provided by <tgmath.h> shall be used.

Rationale

Using the facilities of <tgmath.h> may result in undefined behaviour.

Example

```
#include <tgmath.h>

float f1, f2;

void f ( void )
{
  f1 = sqrt ( f2 );    /* Non-compliant - generic sqrt used      */
}

#include <math.h>

float f1, f2;

void f ( void )
{
  f1 = sqrtf ( f2 );   /* Compliant - float version of sqrt used */
}
```

Rule 21.12 The exception handling features of <fenv.h> should not be used

C99 [Unspecified 27, 28; Undefined 109–111; Implementation J.3.6(8)]

Category	Advisory
Analysis	Decidable, Single Translation Unit
Applies to	C99

Amplification

The identifiers *feclearexcept*, *fegetexceptflag*, *feraiseexcept*, *fesetexceptflag* and *fetestexcept* shall not be used and no macro with one of these names shall be expanded.

The macros *FE_INEXACT*, *FE_DIVBYZERO*, *FE_UNDERFLOW*, *FE_OVERFLOW*, *FE_INVALID* and *FE_ALL_EXCEPT*, along with any implementation-defined floating-point exception macros, shall not be used.

Rationale

In some circumstances, the values of the floating-point status flags are unspecified and attempts to access them may lead to undefined behaviour.

The order in which exceptions are raised by the *feraiseexcept* function is unspecified and could therefore result in a program that has been designed for a certain order not operating correctly.

Example

```
#include <fenv.h>

void f ( float32_t x, float32_t y )
{
  float32_t z;

  feclearexcept ( FE_DIVBYZERO );          /* Non-compliant */

  z = x / y;

  if ( fetestexcept ( FE_DIVBYZERO ) )     /* Non-compliant */
  {
  }

  else
  {
#pragma STDC FENV_ACCESS ON

    z = x * y;
  }

  if ( z > x )
  {
#pragma STDC FENV_ACCESS OFF

    if ( fetestexcept ( FE_OVERFLOW ) )    /* Non-compliant */
    {
    }
  }
}
```

8.22 Resources

Many of the rules in this section are applicable only when rules in other sections have been deviated.

Rule 22.1	All resources obtained dynamically by means of Standard Library functions shall be explicitly released
Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The Standard Library functions that allocate resources are *malloc*, *calloc*, *realloc* and *fopen*.

Rationale

If resources are not explicitly released then it is possible for a failure to occur due to exhaustion of those resources. Releasing resources as soon as possible reduces the possibility that exhaustion will occur.

Example

```
#include <stdlib.h>

int main ( void )
{
  void *b = malloc ( 40 );

  /* Non-compliant - dynamic memory not released */
  return 1;
}
```

```
#include <stdio.h>

int main ( void )
{
  FILE *fp = fopen ( "tmp", "r" );

  /* Non-compliant - file not closed             */
  return 1;
}
```

In the following non-compliant example, the handle on “tmp-1” is lost when “tmp-2” is opened.

```
#include <stdio.h>

int main ( void )
{
  FILE *fp;

  fp = fopen ( "tmp-1", "w" );

  fprintf ( fp, "*" );

  /* File "tmp-1" should be closed here, but stream 'leaks'. */

  fp = fopen ( "tmp-2", "w" );

  fprintf ( fp, "!" );

  fclose ( fp );

  return ( 0 );
}
```

See also

Dir 4.12, Dir 4.13, Rule 21.3, Rule 21.6

Rule 22.2 A block of memory shall only be freed if it was allocated by means of a Standard Library function

C90 [Undefined 92], C99 [Undefined 169]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

The Standard Library functions that allocate memory are *malloc*, *calloc* and *realloc*.

A block of memory is freed when its address is passed to *free* and potentially freed when its address is passed to *realloc*. Once freed, a block of memory is no longer considered to be allocated and therefore cannot subsequently be freed again.

Rationale

Freeing non-allocated memory, or freeing the same allocated memory more than once leads to undefined behaviour.

Example

```
#include <stdlib.h>

void fn ( void )
{
  int32_t a;

  /* Non-compliant - a does not point to allocated storage     */
  free ( &a );
}

void g ( void )
{
  char *p = ( char * ) malloc ( 512 );
  char *q = p;

  free ( p );

  /* Non-compliant - allocated block freed a second time       */
  free ( q );

  /* Non-compliant - allocated block may be freed a third time */
  p = ( char * ) realloc ( p, 1024 );
}
```

See also

Dir 4.12, Dir 4.13, Rule 21.3

Rule 22.3 The same file shall not be open for read and write access at the same time on different streams

C90 [Implementation 61], C99 [Implementation J.3.12(22)]

Category	Required
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

This rule applies to files opened with The Standard Library functions. It may also apply to similar features provided by the execution environment.

Rationale

The Standard does not specify the behaviour if a file is both written and read via different streams.

Note: it is acceptable to open a file multiple times for read-only access.

Example

```
#include <stdio.h>

void fn ( void )
{
  FILE *fw = fopen ( "tmp", "r+" );   /* "r+" opens for read/write */
  FILE *fr = fopen ( "tmp", "r" );    /* Non-compliant             */
}
```

See also

Rule 21.6

Rule 22.4 There shall be no attempt to write to a stream which has been opened as read-only

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

The Standard does not specify the behaviour if an attempt is made to write to a read-only stream. For this reason it is considered unsafe to write to a read-only stream.

Example

```
#include <stdio.h>

void fn ( void )
{
  FILE *fp = fopen ( "tmp", "r" );

  ( void ) fprintf ( fp, "What happens now?" );   /* Non-compliant */

  ( void ) fclose ( fp );
}
```

See also

Rule 21.6

Rule 22.5 A pointer to a `FILE` object shall not be dereferenced

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Amplification

A pointer to a `FILE` object shall not be dereferenced directly or indirectly (e.g. by a call to `memcpy` or `memcmp`).

Rationale

The Standard (C90 Section 7.9.3(6), C99 Section 7.19.3(6)) states that the address of a `FILE` object used to control a stream may be significant and a copy of the object may not give the same behaviour. This rule ensures that such a copy cannot be made.

The direct manipulation of a `FILE` object is prohibited as this may be incompatible with its use as a stream designator.

Example

```
#include <stdio.h>

FILE *pf1;
FILE *pf2;
FILE  f3;
```

```
pf2 = pf1;      /* Compliant     */
 f3 = *pf2;     /* Non-compliant */
```

The following example assumes that `FILE *` specifies a complete type with a member named `pos`:

```
pf1->pos = 0;   /* Non-compliant */
```

See also

Rule 21.6

Rule 22.6 The value of a pointer to a `FILE` shall not be used after the associated stream has been closed

C99 [Undefined 140]

Category	Mandatory
Analysis	Undecidable, System
Applies to	C90, C99

Rationale

The Standard states that the value of a `FILE` pointer is indeterminate after a close operation on a stream.

Example

```
#include <stdio.h>

void fn ( void )
{
  FILE *fp;
  void *p;

  fp = fopen ( "tmp", "w" );

  if ( fp == NULL )
  {
    error_action ( );
  }

  fclose ( fp );

  fprintf ( fp, "?" );   /* Non-compliant */
  p = fp;                /* Non-compliant */
}
```

See also

Dir 4.13, Rule 21.6

9 References

[1] MISRA *Guidelines for the Use of the C Language In Vehicle Based Software*, ISBN 0-9524159-9-0, Motor Industry Research Association, Nuneaton, April 1998

[2] ISO/IEC 9899:1990, *Programming languages — C*, International Organization for Standardization, 1990

[3] Hatton L., *Safer C – Developing Software for High-integrity and Safety-critical Systems*, ISBN 0-07-707640-0, McGraw-Hill, 1994

[4] ISO/IEC 9899:1990/COR 1:1995, *Technical Corrigendum 1*, 1995

[5] ISO/IEC 9899:1990/AMD 1:1995, *Amendment 1*, 1995

[6] ISO/IEC 9899:1990/COR 2:1996, *Technical Corrigendum 2*, 1996

[7] ANSI X3.159-1989, *Programming languages — C*, American National Standards Institute, 1989

[8] ISO/IEC 9899:1999, *Programming languages — C*, International Organization for Standardization, 1999

[9] ISO/IEC 9899:1999/COR 1:2001, *Technical Corrigendum 1*, 2001

[10] ISO/IEC 9899:1999/COR 2:2004, *Technical Corrigendum 2*, 2004

[11] ISO/IEC 9899:1999/COR 3:2007, *Technical Corrigendum 3*, 2007

[12] ISO/IEC 9899:1999 Committee Draft WG14/N1256, *Programming languages — C*, International Organization for Standardization, September 2007

[13] ISO/IEC 9899:2011, *Programming languages — C*, International Organization for Standardization, 2011

[14] ISO/IEC 9899:2011/COR 1:2012, *Technical Corrigendum 1*, 2012

[15] MISRA *Development Guidelines for Vehicle Based Software*, ISBN 0-9524156-0-7, Motor Industry Research Association, Nuneaton, November 1994

[16] MISRA AC AGC *Guidelines for the application of MISRA-C:2004 in the context of automatic code generation*, ISBN 978-1-906400-02-6, MIRA Limited, Nuneaton, November 2007

[17] MISRA AC GMG *Generic modelling design and style guidelines*, ISBN 978-1-906400-06-4, MIRA Limited, Nuneaton, May 2009

[18] MISRA AC SLSF *Modelling design and style guidelines for the application of Simulink and Stateflow*, ISBN 978-1-906400-07-1, MIRA Limited, Nuneaton, May 2009

[19] MISRA AC TL *Modelling style guidelines for the application of Targetlink in the context of automatic code generation*, ISBN 978-1-906400-01-9, MIRA Limited, Nuneaton, November 2007

[20] CRR80, *The Use of Commercial Off-the-Shelf (COTS) Software in Safety Related Applications*, ISBN 0-7176-0984-7, HSE Books

[21] ISO 9001:2008, *Quality management systems — Requirements*, International Organization for Standardization, 2008

[22] ISO 90003:2004, *Software engineering — Guidelines for the application of ISO 9001:2000 to computer software*, ISO, 2004

[23] ISO 26262:2011, *Road vehicles — Functional safety*, ISO, 2011

[24] DO-178C/ED-12C, *Software Considerations in Airborne Systems and Equipment Certification*, RTCA, 2011

[25] The TickIT Guide, *Using ISO 9001:2000 for Software Quality Management System Construction, Certification and Continual Improvement*, Issue 5, British Standards Institution, 2001

[26] Straker D., *C Style: Standards and Guidelines*, ISBN 0–13-116898-3, Prentice Hall 1991

[27] Fenton N.E. and Pfleeger S.L., *Software Metrics: A Rigorous and Practical Approach*, 2nd Edition, ISBN 0-534-95429-1, PWS, 1998

[28] MISRA Report 5 *Software Metrics*, Motor Industry Research Association, Nuneaton, February 1995

[29] MISRA Report 6 *Verification and Validation*, Motor Industry Research Association, Nuneaton, February 1995

[30] Kernighan B.W., Ritchie D.M., *The C programming language*, 2nd edition, ISBN 0-13-110362-8, Prentice Hall, 1988 (note: The 1st edition is not a suitable reference document as it does not describe ANSI/ISO C)

[31] Koenig A., *C Traps and Pitfalls*, ISBN 0-201-17928-8, Addison-Wesley, 1988

[32] IEC 61508:2010, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, International Electromechanical Commission, in 7 parts published in 2010

[33] EN 50128:2011, *Railway applications — Communications, signalling and processing systems — Software for railway control and protection*, CENELEC, 2011

[34] IEC 62304:2006, *Medical device software — Software life cycle processes*, IEC, 2006

[35] ANSI/IEEE Std 754, *IEEE Standard for Binary Floating-Point Arithmetic*, 1985

[36] ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*, International Organization for Standardization, 2003

[37] Goldberg D., *What Every Computer Scientist Should Know about Floating-Point Arithmetic*, Computing Surveys, March 1991

[38] Software Engineering Center, Information-technology Promotion Agency, Japan (IPA/SEC), *Embedded System development Coding Reference (ESCR) [C language edition]* Version 1.1, SEC Books, 2012

Appendix A Summary of guidelines

The implementation

Dir 1.1	Required	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood

Compilation and build

Dir 2.1	Required	All source files shall compile without any compilation errors

Requirements traceability

Dir 3.1	Required	All code shall be traceable to documented requirements

Code design

Dir 4.1	Required	Run-time failures shall be minimized
Dir 4.2	Advisory	All usage of assembly language should be documented
Dir 4.3	Required	Assembly language shall be encapsulated and isolated
Dir 4.4	Advisory	Sections of code should not be "commented out"
Dir 4.5	Advisory	Identifiers in the same name space with overlapping visibility should be typographically unambiguous
Dir 4.6	Advisory	*typedefs* that indicate size and signedness should be used in place of the basic numerical types
Dir 4.7	Required	If a function returns error information, then that error information shall be tested
Dir 4.8	Advisory	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden
Dir 4.9	Advisory	A function should be used in preference to a *function-like macro* where they are interchangeable
Dir 4.10	Required	Precautions shall be taken in order to prevent the contents of a *header file* being included more than once
Dir 4.11	Required	The validity of values passed to library functions shall be checked
Dir 4.12	Required	Dynamic memory allocation shall not be used
Dir 4.13	Advisory	Functions which are designed to provide operations on a resource should be called in an appropriate sequence

A standard C environment

Rule 1.1	Required	The program shall contain no violations of the standard C syntax and *constraints*, and shall not exceed the implementation's translation limits
Rule 1.2	Advisory	Language extensions should not be used
Rule 1.3	Required	There shall be no occurrence of undefined or critical unspecified behaviour

Unused code

Rule 2.1	Required	A project shall not contain *unreachable* code
Rule 2.2	Required	There shall be no *dead code*
Rule 2.3	Advisory	A project should not contain unused type declarations
Rule 2.4	Advisory	A project should not contain unused tag declarations
Rule 2.5	Advisory	A project should not contain unused macro declarations
Rule 2.6	Advisory	A function should not contain unused label declarations
Rule 2.7	Advisory	There should be no unused parameters in functions

Comments

Rule 3.1	Required	The character sequences `/*` and `//` shall not be used within a comment
Rule 3.2	Required	Line-splicing shall not be used in `//` comments

Character sets and lexical conventions

Rule 4.1	Required	Octal and hexadecimal escape sequences shall be terminated
Rule 4.2	Advisory	Trigraphs should not be used

Identifiers

Rule 5.1	Required	*External identifiers* shall be distinct
Rule 5.2	Required	Identifiers declared in the same scope and name space shall be distinct
Rule 5.3	Required	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope
Rule 5.4	Required	*Macro identifiers* shall be distinct
Rule 5.5	Required	Identifiers shall be distinct from macro names
Rule 5.6	Required	A *typedef* name shall be a unique identifier
Rule 5.7	Required	A tag name shall be a unique identifier

Rule 5.8	Required	Identifiers that define objects or functions with external linkage shall be unique
Rule 5.9	Advisory	Identifiers that define objects or functions with internal linkage should be unique

Types

Rule 6.1	Required	Bit-fields shall only be declared with an appropriate type
Rule 6.2	Required	Single-bit named bit fields shall not be of a signed type

Literals and constants

Rule 7.1	Required	Octal constants shall not be used
Rule 7.2	Required	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type
Rule 7.3	Required	The lowercase character "l" shall not be used in a literal suffix
Rule 7.4	Required	A string literal shall not be *assigned* to an object unless the object's type is "pointer to *const*-qualified *char*"

Declarations and definitions

Rule 8.1	Required	Types shall be explicitly specified
Rule 8.2	Required	Function types shall be in *prototype form* with named parameters
Rule 8.3	Required	All declarations of an object or function shall use the same names and type qualifiers
Rule 8.4	Required	A compatible declaration shall be visible when an object or function with external linkage is defined
Rule 8.5	Required	An external object or function shall be declared once in one and only one file
Rule 8.6	Required	An identifier with external linkage shall have exactly one external definition
Rule 8.7	Advisory	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit
Rule 8.8	Required	The *static* storage class specifier shall be used in all declarations of objects and functions that have internal linkage
Rule 8.9	Advisory	An object should be defined at block scope if its identifier only appears in a single function
Rule 8.10	Required	An *inline function* shall be declared with the static storage class
Rule 8.11	Advisory	When an array with external linkage is declared, its size should be explicitly specified

Rule 8.12	Required	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique
Rule 8.13	Advisory	A pointer should point to a const-qualified type whenever possible
Rule 8.14	Required	The *restrict* type qualifier shall not be used

Initialization

Rule 9.1	Mandatory	The value of an object with automatic storage duration shall not be read before it has been set
Rule 9.2	Required	The initializer for an aggregate or union shall be enclosed in braces
Rule 9.3	Required	Arrays shall not be partially initialized
Rule 9.4	Required	An element of an object shall not be initialized more than once
Rule 9.5	Required	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly

The essential type model

Rule 10.1	Required	Operands shall not be of an inappropriate *essential* type
Rule 10.2	Required	Expressions of *essentially character type* shall not be used inappropriately in addition and subtraction operations
Rule 10.3	Required	The value of an expression shall not be assigned to an object with a narrower *essential type* or of a different *essential type category*
Rule 10.4	Required	Both operands of an operator in which the *usual arithmetic conversions* are performed shall have the same *essential type category*
Rule 10.5	Advisory	The value of an expression should not be cast to an inappropriate *essential type*
Rule 10.6	Required	The value of a *composite expression* shall not be assigned to an object with wider *essential type*
Rule 10.7	Required	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*
Rule 10.8	Required	The value of a *composite expression* shall not be cast to a different *essential type category* or a wider *essential type*

Pointer type conversions

Rule 11.1	Required	Conversions shall not be performed between a pointer to a function and any other type
Rule 11.2	Required	Conversions shall not be performed between a pointer to an incomplete type and any other type
Rule 11.3	Required	A cast shall not be performed between a pointer to object type and a pointer to a different object type

Rule 11.4	Advisory	A conversion should not be performed between a pointer to object and an integer type
Rule 11.5	Advisory	A conversion should not be performed from pointer to *void* into pointer to object
Rule 11.6	Required	A cast shall not be performed between pointer to *void* and an arithmetic type
Rule 11.7	Required	A cast shall not be performed between pointer to object and a non-integer arithmetic type
Rule 11.8	Required	A cast shall not remove any *const* or *volatile* qualification from the type pointed to by a pointer
Rule 11.9	Required	The macro `NULL` shall be the only permitted form of integer *null pointer constant*

Expressions

Rule 12.1	Advisory	The precedence of operators within expressions should be made explicit
Rule 12.2	Required	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
Rule 12.3	Advisory	The comma operator should not be used
Rule 12.4	Advisory	Evaluation of *constant expressions* should not lead to unsigned integer wrap-around

Side effects

Rule 13.1	Required	*Initializer* lists shall not contain *persistent side effects*
Rule 13.2	Required	The value of an expression and its *persistent side effects* shall be the same under all permitted evaluation orders
Rule 13.3	Advisory	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
Rule 13.4	Advisory	The result of an assignment operator should not be *used*
Rule 13.5	Required	The right hand operand of a logical `&&` or `||` operator shall not contain *persistent side effects*
Rule 13.6	Mandatory	The operand of the *sizeof* operator shall not contain any expression which has potential *side effects*

Control statement expressions

Rule 14.1	Required	A *loop counter* shall not have *essentially floating* type
Rule 14.2	Required	A *for* loop shall be well-formed
Rule 14.3	Required	Controlling expressions shall not be invariant
Rule 14.4	Required	The controlling expression of an *if* statement and the controlling expression of an *iteration-statement* shall have *essentially Boolean* type

Control flow

Rule 15.1	Advisory	The *goto* statement should not be used
Rule 15.2	Required	The *goto* statement shall jump to a label declared later in the same function
Rule 15.3	Required	Any label referenced by a *goto* statement shall be declared in the same block, or in any block enclosing the *goto* statement
Rule 15.4	Advisory	There should be no more than one *break* or *goto* statement used to terminate any iteration statement
Rule 15.5	Advisory	A function should have a single point of exit at the end
Rule 15.6	Required	The body of an *iteration-statement* or a *selection-statement* shall be a *compound-statement*
Rule 15.7	Required	All *if ... else if* constructs shall be terminated with an *else* statement

Switch statements

Rule 16.1	Required	All *switch* statements shall be well-formed
Rule 16.2	Required	A *switch label* shall only be used when the most closely-enclosing compound statement is the body of a *switch* statement
Rule 16.3	Required	An unconditional *break* statement shall terminate every *switch-clause*
Rule 16.4	Required	Every *switch* statement shall have a *default* label
Rule 16.5	Required	A *default* label shall appear as either the first or the last *switch label* of a *switch* statement
Rule 16.6	Required	Every *switch* statement shall have at least two *switch-clauses*
Rule 16.7	Required	A *switch-expression* shall not have *essentially Boolean* type

Functions

Rule 17.1	Required	The features of `<stdarg.h>` shall not be used
Rule 17.2	Required	Functions shall not call themselves, either directly or indirectly
Rule 17.3	Mandatory	A function shall not be declared implicitly

Rule 17.4	Mandatory	All exit paths from a function with non-*void* return type shall have an explicit *return* statement with an expression
Rule 17.5	Advisory	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements
Rule 17.6	Mandatory	The declaration of an array parameter shall not contain the *static* keyword between the `[ ]`
Rule 17.7	Required	The value returned by a function having non-*void* return type shall be *used*
Rule 17.8	Advisory	A function parameter should not be modified

Pointers and arrays

Rule 18.1	Required	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand
Rule 18.2	Required	Subtraction between pointers shall only be applied to pointers that address elements of the same array
Rule 18.3	Required	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object
Rule 18.4	Advisory	The +, –, += and -= operators should not be applied to an expression of pointer type
Rule 18.5	Advisory	Declarations should contain no more than two levels of pointer nesting
Rule 18.6	Required	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
Rule 18.7	Required	Flexible array members shall not be declared
Rule 18.8	Required	Variable-length array types shall not be used

Overlapping storage

Rule 19.1	Mandatory	An object shall not be assigned or copied to an overlapping object
Rule 19.2	Advisory	The *union* keyword should not be used

Preprocessing directives

Rule 20.1	Advisory	*#include* directives should only be preceded by preprocessor directives or comments
Rule 20.2	Required	The `'`, `"` or `\` characters and the `/*` or `//` character sequences shall not occur in a *header file* name
Rule 20.3	Required	The *#include* directive shall be followed by either a `<filename>` or `"filename"` sequence
Rule 20.4	Required	A macro shall not be defined with the same name as a keyword
Rule 20.5	Advisory	*#undef* should not be used

Rule 20.6	Required	Tokens that look like a preprocessing directive shall not occur within a macro argument
Rule 20.7	Required	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses
Rule 20.8	Required	The controlling expression of a *#if* or *#elif* preprocessing directive shall evaluate to 0 or 1
Rule 20.9	Required	All identifiers used in the controlling expression of *#if* or *#elif* preprocessing directives shall be *#define*'d before evaluation
Rule 20.10	Advisory	The `#` and `##` preprocessor operators should not be used
Rule 20.11	Required	A macro parameter immediately following a `#` operator shall not immediately be followed by a `##` operator
Rule 20.12	Required	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
Rule 20.13	Required	A line whose first token is `#` shall be a valid preprocessing directive
Rule 20.14	Required	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

Standard libraries

Rule 21.1	Required	*#define* and *#undef* shall not be used on a reserved identifier or reserved macro name
Rule 21.2	Required	A reserved identifier or macro name shall not be declared
Rule 21.3	Required	The memory allocation and deallocation functions of `<stdlib.h>` shall not be used
Rule 21.4	Required	The standard header file `<setjmp.h>` shall not be used
Rule 21.5	Required	The standard header file `<signal.h>` shall not be used
Rule 21.6	Required	The Standard Library input/output functions shall not be used
Rule 21.7	Required	The *atof*, *atoi*, *atol* and *atoll* functions of `<stdlib.h>` shall not be used
Rule 21.8	Required	The library functions *abort*, *exit*, *getenv* and *system* of `<stdlib.h>` shall not be used
Rule 21.9	Required	The library functions *bsearch* and *qsort* of `<stdlib.h>` shall not be used
Rule 21.10	Required	The Standard Library time and date functions shall not be used
Rule 21.11	Required	The standard *header file* `<tgmath.h>` shall not be used
Rule 21.12	Advisory	The exception handling features of `<fenv.h>` should not be used

Resources

Rule 22.1	Required	All resources obtained dynamically by means of Standard Library functions shall be explicitly released
Rule 22.2	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function
Rule 22.3	Required	The same file shall not be open for read and write access at the same time on different streams
Rule 22.4	Mandatory	There shall be no attempt to write to a stream which has been opened as read-only
Rule 22.5	Mandatory	A pointer to a `FILE` object shall not be dereferenced
Rule 22.6	Mandatory	The value of a pointer to a `FILE` shall not be used after the associated stream has been closed

Appendix B Guideline attributes

Rule	Category	Applies to	Analysis
Dir 1.1	Required	C90, C99	
Dir 2.1	Required	C90, C99	
Dir 3.1	Required	C90, C99	
Dir 4.1	Required	C90, C99	
Dir 4.2	Advisory	C90, C99	
Dir 4.3	Required	C90, C99	
Dir 4.4	Advisory	C90, C99	
Dir 4.5	Advisory	C90, C99	
Dir 4.6	Advisory	C90, C99	
Dir 4.7	Required	C90, C99	
Dir 4.8	Advisory	C90, C99	
Dir 4.9	Advisory	C90, C99	
Dir 4.10	Required	C90, C99	
Dir 4.11	Required	C90, C99	
Dir 4.12	Required	C90, C99	
Dir 4.13	Advisory	C90, C99	
Rule 1.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 1.2	Advisory	C90, C99	Undecidable, Single Translation Unit
Rule 1.3	Required	C90, C99	Undecidable, System
Rule 2.1	Required	C90, C99	Undecidable, System
Rule 2.2	Required	C90, C99	Undecidable, System
Rule 2.3	Advisory	C90, C99	Decidable, System
Rule 2.4	Advisory	C90, C99	Decidable, System
Rule 2.5	Advisory	C90, C99	Decidable, System
Rule 2.6	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 2.7	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 3.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 3.2	Required	C99	Decidable, Single Translation Unit
Rule 4.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 4.2	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 5.1	Required	C90, C99	Decidable, System
Rule 5.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 5.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 5.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 5.5	Required	C90, C99	Decidable, Single Translation Unit
Rule 5.6	Required	C90, C99	Decidable, System
Rule 5.7	Required	C90, C99	Decidable, System
Rule 5.8	Required	C90, C99	Decidable, System
Rule 5.9	Advisory	C90, C99	Decidable, System
Rule 6.1	Required	C90, C99	Decidable, Single Translation Unit

Rule	Category	Applies to	Analysis
Rule 6.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 7.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 7.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 7.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 7.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 8.1	Required	C90	Decidable, Single Translation Unit
Rule 8.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 8.3	Required	C90, C99	Decidable, System
Rule 8.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 8.5	Required	C90, C99	Decidable, System
Rule 8.6	Required	C90, C99	Decidable, System
Rule 8.7	Advisory	C90, C99	Decidable, System
Rule 8.8	Required	C90, C99	Decidable, Single Translation Unit
Rule 8.9	Advisory	C90, C99	Decidable, System
Rule 8.10	Required	C99	Decidable, Single Translation Unit
Rule 8.11	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 8.12	Required	C90, C99	Decidable, Single Translation Unit
Rule 8.13	Advisory	C90, C99	Undecidable, System
Rule 8.14	Required	C99	Decidable, Single Translation Unit
Rule 9.1	Mandatory	C90, C99	Undecidable, System
Rule 9.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 9.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 9.4	Required	C99	Decidable, Single Translation Unit
Rule 9.5	Required	C99	Decidable, Single Translation Unit
Rule 10.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.5	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 10.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 10.8	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.4	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 11.5	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 11.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.8	Required	C90, C99	Decidable, Single Translation Unit
Rule 11.9	Required	C90, C99	Decidable, Single Translation Unit
Rule 12.1	Advisory	C90, C99	Decidable, Single Translation Unit

Rule	Category	Applies to	Analysis
Rule 12.2	Required	C90, C99	Undecidable, System
Rule 12.3	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 12.4	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 13.1	Required	C99	Undecidable, System
Rule 13.2	Required	C90, C99	Undecidable, System
Rule 13.3	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 13.4	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 13.5	Required	C90, C99	Undecidable, System
Rule 13.6	Mandatory	C90, C99	Decidable, Single Translation Unit
Rule 14.1	Required	C90, C99	Undecidable, System
Rule 14.2	Required	C90, C99	Undecidable, System
Rule 14.3	Required	C90, C99	Undecidable, System
Rule 14.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 15.1	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 15.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 15.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 15.4	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 15.5	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 15.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 15.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.5	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 16.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 17.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 17.2	Required	C90, C99	Undecidable, System
Rule 17.3	Mandatory	C90	Decidable, Single Translation Unit
Rule 17.4	Mandatory	C90, C99	Decidable, Single Translation Unit
Rule 17.5	Advisory	C90, C99	Undecidable, System
Rule 17.6	Mandatory	C99	Decidable, Single Translation Unit
Rule 17.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 17.8	Advisory	C90, C99	Undecidable, System
Rule 18.1	Required	C90, C99	Undecidable, System
Rule 18.2	Required	C90, C99	Undecidable, System
Rule 18.3	Required	C90, C99	Undecidable, System
Rule 18.4	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 18.5	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 18.6	Required	C90, C99	Undecidable, System
Rule 18.7	Required	C99	Decidable, Single Translation Unit

Rule	Category	Applies to	Analysis
Rule 18.8	Required	C99	Decidable, Single Translation Unit
Rule 19.1	Mandatory	C90, C99	Undecidable, System
Rule 19.2	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 20.1	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 20.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.5	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 20.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.8	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.9	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.10	Advisory	C90, C99	Decidable, Single Translation Unit
Rule 20.11	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.12	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.13	Required	C90, C99	Decidable, Single Translation Unit
Rule 20.14	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.1	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.2	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.3	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.4	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.5	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.6	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.7	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.8	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.9	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.10	Required	C90, C99	Decidable, Single Translation Unit
Rule 21.11	Required	C99	Decidable, Single Translation Unit
Rule 21.12	Advisory	C99	Decidable, Single Translation Unit
Rule 22.1	Required	C90, C99	Undecidable, System
Rule 22.2	Mandatory	C90, C99	Undecidable, System
Rule 22.3	Required	C90, C99	Undecidable, System
Rule 22.4	Mandatory	C90, C99	Undecidable, System
Rule 22.5	Mandatory	C90, C99	Undecidable, System
Rule 22.6	Mandatory	C90, C99	Undecidable, System

Appendix C Type safety issues with C

ISO C may be considered to exhibit poor type safety as it permits a wide range of implicit type conversions to take place. These type conversions can compromise safety as their implementation-defined aspects can cause developer confusion.

An explanation of these conversions follows, along with some examples to show how they can lead to developer confusion.

C.1 Type conversions

C.1.1 Implicit conversions

Implicit type conversions within ISO C permit the use of mixed types within expressions and may occur even when the types used are all the same. There are three types of implicit conversion:

1. An *integer promotion* conversion to *signed int* or *unsigned int* takes place whenever a bit-field or object with *(un)signed char*, *(un)signed short* or *enum* type is used arithmetically as described in C90 Section 6.2.1.1, C99 Section 6.3.1.1. *Integer promotion* preserves the original value, but the signedness may change (e.g. *unsigned char* will be converted to *signed int*);

2. The *usual arithmetic conversions* convert (balance) operands to a common type whenever operands with different types are used with certain operators, as described in C90 Section 6.2.1.5, C99 Section 6.3.1.8;

3. Conversion on *assignment*.

An expression may contain none, one, two or all of these implicit conversions, as the examples below show (`si` and `ui` are 32-bit signed and unsigned integers and `u8` is an 8-bit *unsigned char*):

Expression	Promote	Balance	Convert	Notes
`si = si + si;`				
`si = uc + uc;`	Yes			`uc` promoted to *signed int*
`ui = si + ui;`		Yes		`si` is balanced to *unsigned int*
`ui = ui + uc;`	Yes	Yes		`uc` promoted to *signed int* and balanced to *unsigned int*
`ui = si;`			Yes	`si` converted to *unsigned int*
`ui = uc + uc;`	Yes		Yes	`uc` promoted to *signed int* and result of expression converted to *unsigned int*
`si = si + ui;`		Yes	Yes	`si` is balanced to *unsigned int* and result of expression converted to *signed int*
`si = ui + uc;`	Yes	Yes	Yes	`uc` promoted to *signed int*, balanced to *unsigned int* and result of expression converted to *signed int*

C.1.2 Explicit conversions

Explicit type conversions (casts) may be introduced for functional reasons:

- To change the type in which a subsequent arithmetic operation is performed;
- To truncate a value deliberately;
- To make a type conversion explicit in the interests of clarity.

ISO C does not require all explicit casts to be checked, allowing conversions to take place between incompatible types.

C.1.3 Concerns with conversions

Implicit and explicit conversion can lead to several concerns, including:

- **Loss of value**: e.g. conversion to a type where the magnitude of the value cannot be represented;
- **Loss of sign**: e.g. conversion from a signed type to an unsigned type resulting in loss of sign;
- **Loss of precision**: e.g. conversion from a floating type to an integer type with consequent loss of precision;
- **Loss of layout**: e.g. conversion from a pointer to one type to a pointer to a different type leading to an incompatible storage layout.

Whilst many type conversions are safe, the only ones that can be guaranteed safe for all data values and all possible conforming implementations are:

- Conversion of an integer value to a wider integer type of the same signedness;
- Conversion of a floating type to a wider floating type.

Depending on the implemented integer sizes, other type conversions may also be safe. However, any potentially dangerous type conversions should be identified by making them explicit.

C.2 Developer confusion

As the sizes in which types are implemented can vary between implementations, the contexts in which implicit conversions take place can also vary. This places a requirement on the developer to maintain a mental model of the type system in use for any particular project. It is relatively easy for a developer to use the wrong model from time-to-time, especially if they are working on multiple projects having different implementations.

C.2.1 Type widening in integer promotion

The type in which integer expressions are evaluated depends on the type of the operands after any integer promotion. Multiplying two 8-bit values will always give a result that is at least 16 bits wide. However, multiplying two 16-bit values will only give a 32-bit result when the implemented size of *int* is at least 32 bits. It is safer never to rely on the type-widening obtained by integer promotion as the associated implementation-defined behaviour may lead to developer confusion. Consider the following example:

```
uint16_t u16a = 40000;          /* unsigned short or unsigned int ? */
uint16_t u16b = 30000;          /* unsigned short or unsigned int ? */
uint32_t u32x;                  /* unsigned int or unsigned long  ? */

u32x = u16a + u16b;             /* u32x = 70000 or 4464 ?           */
```

The expected result is possibly 70 000, but the value assigned to `u32x` will depend on the implemented size of *int*. If this is 32 bits, the addition will occur in 32-bit signed arithmetic and the "correct" value will be obtained. If it is only 16 bits, the addition will take place in 16-bit unsigned arithmetic, wraparound will occur and will yield the value 4 464 (70 000 % 65 536). Wraparound in unsigned arithmetic is well defined but, as its effects depend on the type in which the arithmetic is performed, its intentional use should documented.

C.2.2 Evaluation type confusion

A similar problem arises where a developer is deceived into thinking that the type in which a calculation is conducted is influenced by the type to which the result is assigned or converted. For example, in the following code the two 16-bit objects are added together in 16-bit arithmetic (assuming *int* is 16-bit), and the result is converted to type `uint32_t` on assignment.

```
u32x = u16a + u16b;
```

It is not unusual for developers to be deceived into thinking that the addition is performed in 32-bit arithmetic, because of the type of `u32x`.

Confusion of this nature is not confined to integer arithmetic or to implicit conversions. The following examples demonstrate some statements in which the result is well defined but the calculation may not be performed in the type that the developer expects:

```
u32a = (uint32_t)(u16a * u16b);     /* May not be a 32-bit operation    */
f64a = u16a / u16b;                 /* Not floating point division      */
f32a = (float32_t)(u16a / u16b);    /* Not floating point division      */
f64a = f32a + f32b;                 /* May be a low precision operation */
f64a = (float64_t)(f32a + f32b);    /* May be a low precision operation */
```

C.2.3 Change of signedness in arithmetic operations

Integer promotion may lead to the result of an expression not meeting developer expectations. For example, the expression `10 - u16a` yields a result that is dependent on the size of *int*. If `u16a` has a value of 100 and *int* is 32 bits, then the result is signed with a value of -90. However, if *int* is 16 bits the result will be unsigned with a value of 65 446.

C.2.4 Change of signedness in bitwise operations

Integer promotion arising when bitwise operators are applied to small unsigned types can lead to confusion. For example a bitwise complement operation on an operand of type *unsigned char* will generally yield a result of type (*signed*) *int* with a negative value. The operand is promoted to type *int* before the operation and the extra high order bits are set by the complement process. The number of extra bits, if any, is dependent on the size of an *int* and it is particularly hazardous if the complement operation is followed by a right shift as this leads to implementation-defined behaviour.

```
u8a = 0xff;

if ( ~u8a == 0x00U )  /* This test will always fail */
```

Appendix D Essential types

The C language as defined by the ISO C standard contains a number of weaknesses and inconsistencies and, unfortunately, the *standard type* of an expression is not always fully descriptive of the essential nature of the data being represented. For example:

- Integer promotion The set of integer types do not behave in a consistent way. As a result of integer promotion, an expression which is intrinsically *unsigned* (e.g. *unsigned char* + *unsigned char*) typically has a *standard type* of *signed int*.
- Integer constants Integer constants only exist for *signed/unsigned int*, *long* and *long long* types. This complicates the issue of ensuring type consistency (for example when passing constants as function arguments).
- Character constants The *standard type* of a character constant (e.g. `'x'`) is defined to be *int* (not *char*). This obscures the distinction between character data (i.e. "characters") and numeric data (i.e. "number values").
- Logical expressions A Boolean type does not exist in the C90 language. Type *_Bool* was introduced into C99, but unfortunately, for reasons of backwards compatibility, the result of equality (`==, !=`), relational (`<, <=, >=, >`), and logical (`&&, ||, !`) operators is still of type *int* rather than type *_Bool*.
- Bit-fields ISO C does not define a bit-field type. This means that it is not possible, for example, to cast to a bit-field type. The *standard type* of a bit-field as defined by ISO C is one of *_Bool*, *signed int* or *unsigned int*. The signedness of a bit-field of type *int* is implementation-defined.
- Enumerations An enumeration has a type which is implementation-defined.
  The implementation is free to use any integer type which is capable of representing the enumeration. If the enumeration includes negative values, the type will be a signed integer type. If the enumeration consists only of non-negative values, the type may be either a signed or unsigned integer type.
- Enumeration constants Regardless of the type used to implement an enumeration, an enumeration constant is always of type *int*.

D.1 The essential type category of expressions

MISRA C:2012 introduces the concept of *essential type* to help mitigate the issues identified above.

The *essential type category* of an expression is one of:

- *Essentially Boolean*;
- *Essentially character*;
- *Essentially enum*;
- *Essentially signed*;
- *Essentially unsigned*;
- *Essentially floating*.

The following table shows how the standard integer types map on to *essential type categories*. *Note:* C99 implementations may provide *extended integer types* each of which would be allocated a location appropriate to its rank and signedness (see C99 Section 6.3.1.1).

Essential type category					
Boolean	character	signed	unsigned	enum<i>	floating
_Bool	char	signed char signed short signed int signed long signed long long	unsigned char unsigned short unsigned int unsigned long unsigned long long	named enum	float double long double

The concept of *rank* is particularly relevant when describing the set of signed and unsigned types; *signed* and *unsigned long long* share the "highest" rank and *signed* and *unsigned char* share the "lowest" rank. In the ISO C99 standard, "rank" is a term applied only to integer types.

The *essential type* of an expression only differs from the standard C type (*standard type*) in expressions where the *standard type* is either *signed int* or *unsigned int*.

In the following paragraphs the specific situations are defined where *essential type* and *standard type* are different.

D.2 The essential type of character data

An object with a *standard type* of *char* (i.e. single byte character) has *essentially char type*.

D.3 The signed and unsigned type of lowest rank (STLR and UTLR)

The *Essential Type Model* introduces the concept of the *Signed Type of Lowest Rank* (the *STLR*) and the *Unsigned Type of Lowest Rank* (the *UTLR*). These allow *integer constant expressions* and bit-fields to be used within expressions having an *essential type* with a rank lower than that of *int*. In the case of *integer constant expressions*, this is a convenience as it avoids the need to cast the expression, or its operands, to a type with lower rank. Similarly, for an implementation that does not permit bit-fields to have a type whose rank is lower than that of *int*, it avoids the need to cast bit-fields in some situations.

1. The *STLR* is the *signed type* having the lowest rank required to represent the value of a particular *integer constant expression* or both the maximum and minimum values of a bit-field;
2. The *UTLR* is the *unsigned type* having the lowest rank required to represent the value of a particular *integer constant expression* or the maximum value of a bit-field.

D.4 The essential type of bit-fields

The *essential type* of a bit-field is determined by the first of the following that applies:

1. For a bit-field which is implemented with an *essentially Boolean type* it is *essentially Boolean*;
2. For a bit-field which is implemented with a *signed type* it is the *STLR* which is able to represent the bit-field;
3. For a bit-field which is implemented with an *unsigned type* it is the *UTLR* which is able to represent the bit-field.

D.5 The essential type of enumerations

Two distinct types of enumeration need to be considered:

1. A *named enum type* is an enumeration which has a *tag* or which is used in the definition of any object, function or type;
2. An *anonymous enum type* is an enumeration which does not have a *tag* and which is not used in the definition of any object, function or type. This will typically be used to define a set of constants, which may or may not be related.

A *named enum type* is distinct from all other *named enum types*, even if declared in an inner scope with exactly the same tag and enumeration constants. Each instance of a *named enum type* is denoted as *enum<i>* in this document, with the *i* being different for each such type.

An alias for a *named enum type* created using *typedef* denotes that *named enum type* and is not a new type.

The *essential type* of an *anonymous enum type* is its *standard type*.

The following all have *named enum type* and have distinct *essential types*:

```
        enum ETAG { A, B, C };
typedef enum      { A, B, C } ETYPE;
typedef enum ETAG { A, B, C } ETYPE;
        enum      { A, B, C } x;
```

The following *typedefs* both refer to the same *enum<i>*:

```
typedef enum      { A, B, C } ETYPE;
typedef ETYPE                 FTYPE;
```

D.6 The essential type of literal constants

The ISO C99 standard defines the following constants of *integer type*:

- Integer constants;
- Enumeration constants;
- Character constants.

Note: a constant of *integer type* is not necessarily an *integer constant*.

Integer constants

1. If the *standard type* of an integer constant is *signed int* then its *essential type* is the *STLR*;
2. If the *standard type* of an integer constant is *unsigned int* then its *essential type* is the *UTLR*.

Enumeration constants

The *standard type* of an enumeration constant in C is always *int*, regardless of the implemented type of the enumeration and regardless of the type of any initializer expression used to define its value. For example, if an enumeration constant is initialized with an unsigned value (e.g. `500U`), the constant will still be considered to have a *standard type* of *signed int*.

The *essential type* of an enumeration constant is determined as follows:

1. If an enumeration defines a *named enum type* then the *essential type* of its enumeration constants is *enum<i>*;

   1.1 If a *named enum type* is used to define an *essentially Boolean type* then the *essential type* of its enumeration constants is *essentially Boolean*.

2. If an enumeration defines an *anonymous enum type* then the *essential type* of each enumeration constant is the *STLR* of its value.

In the following, each of the enumeration constants, and the object `x`, have *essential type* of *enum<i>*:

```
enum ETAG { A = 8, B = 64, C = 128 } x;
```

The following *anonymous enum type* defines a set of constants. The *essential type* of each constant is the *STLR*. Therefore on a machine with 8-bit *char* and 16-bit *short* types, `A` and `B` have an *essential type* of *signed char* but `C` has an *essential type* of *signed short*.

```
enum { A = 8, B = 64, C = 128 };
```

Character constants

The *standard type* of a character constant (e.g. `'q'`, `'xy'`) is *int*, not *char*.

1. If a character constant consists of a single character then its *essential type* is *char*;
2. Else the *essential type* is the same as its *standard type*.

Boolean constants

The C99 standard provides no syntax to explicitly define a constant of *_Bool* type. The library header `<stdbool.h>` defines *false* and *true* in terms of constants 0 and 1 of type *int*, but these have an *essential type* of *essentially Boolean*. If these are not used or are not available, a *constant expression* of type *_Bool* may be declared using an explicit cast or an operator that delivers an *essentially Boolean* value.

Tools may also provide additional ways of identifying *essentially Boolean types*. For example, a tool could be configured to recognize:

```
enum Bool {False, True};
```

The use of a cast to define a *constant expression* of Boolean type (e.g. `(_Bool)0`) is only appropriate if the expression is not to be used in a *#if* or *#elif* preprocessing directive. Use of a cast within a preprocessing directive is a syntax error.

D.7 The essential type of expressions

The *essential type* of any expression not listed in this section is the same as its *standard type*.

Comma (,)

The *essential type* of the result is the *essential type* of the right hand operand.

Relational (< <= >= >), Equality (== !=) and Logical (&& || !)

The result of the expression is *essentially Boolean*.

Shift (<< >>)

1. If the left hand operand is *essentially unsigned* then:

    1.1 If both operands are *integer constant expressions* then the *essential type* of the result is the *UTLR* of the result;

    1.2 Else the *essential type* of the result is the *essential type* of the left hand operand.

2. Else the *essential type* is the *standard type*.

Bitwise complement (~)

1. If the operand is *essentially unsigned* then:

    1.1 If the operand is an *integer constant expression* then the *essential type* of the result is the *UTLR* of the result;

    1.2 Else the *essential type* of the result is the *essential type* of the operand.

2. Else the *essential type* is the *standard type*.

Unary plus (+)

1. If the operand is *essentially signed* or *essentially unsigned* then the *essential type* of the result is the *essential type* of the operand;

2. Else the *essential type* is the *standard type*.

Unary minus (–)

1. If the operand is *essentially signed* then:

    1.1 If the expression is an *integer constant expression* then the *essential type* of the result is the *STLR* of the whole expression;

    1.2 Else the *essential type* of the result is the *essential type* of the operand.

2. Else the *essential type* is the *standard type*.

Conditional (? :)

1. If the *essential type* of the second and third operands is the same then the result has the same *essential type*;

2. Else if the second and third operands are both *essentially signed* then the *essential type* of the result is the *essential type* of the one with the highest rank;

3. Else if the second and third operands are both *essentially unsigned* then the *essential type* of the result is the *essential type* of the one with the highest rank;

4. Else the *essential type* is the *standard type*.

Operations subject to the usual arithmetic conversions (* / % + – & | ^)

1. If the operands are both *essentially signed* then:

    1.1 If the expression is an *integer constant expression* then the *essential type* of the result is the *STLR* of the result;

    1.2 Else the *essential type* of the result is the *essential type* of the operand with the highest rank.

2. Else if the operands are both *essentially unsigned* then:

    2.1 If the expression is an *integer constant expression* then the *essential type* of the result is the *UTLR* of the result;

    2.2 Else the *essential type* of the result is the *essential type* of the operand with the highest rank.

3. Else the *essential type* is the *standard type*.

Addition with *char*

1. If one operand is *essentially character* and the other is *essentially signed* or *essentially unsigned* then the *essential type* of the result is *char*;

2. Else the *essential type* is the *standard type*.

Subtraction with *char*

1. If the first operand is *essentially character* and the second is *essentially signed* or *essentially unsigned* then the *essential type* of the result is *char*;

2. Else the *essential type* is the *standard type*.

Appendix E Applicability to automatically generated code

E.1 Guideline categories for automatically generated code

Most MISRA C guidelines have the same category for automatically generated code as they do for manually generated code.

This section lists those guidelines for which the category is different. For convenience, guidelines with similar rationales are grouped together.

E.1.1 Additional categories

An additional guideline category is used in automatically generated code:

E.1.1.1 Readability

If a guideline has the "readability" category then it is not necessary to comply with it **provided that** the automatically generated code is not intended to be read, reviewed or modified by human programmers. If the code **is** being used by humans then the category remains the same as for manually generated code.

E.1.2 Hiding identifiers

Rule 5.3	Advisory

An automatic code generator is capable of tracking the identifiers that it uses and should not be confused about any identifier reuse. The guideline is therefore advisory when applied to automatically generated code.

When manually generated code is injected into automatically generated code or *vice-versa* it is possible for an identifier to be hidden without the code generator being aware. This guideline therefore remains required if the identifier that is being hidden, or that is hiding another identifier, is declared in manually generated code.

E.1.3 Octal constants

Rule 7.1	Advisory

An automatic code generator is unlikely to generate an octal constant unintentionally.

E.1.4 Compatible declarations with external linkage

Rule 8.4	Advisory	Rule 8.5	Advisory

These guidelines require:

- A compatible declaration to be visible when an object or function with external linkage is defined;
- Each object or function with external linkage to be declared in only one file.

Together, they provide a mechanism for ensuring that all translation units using a declaration of the object or function are doing so in a manner that is compatible with the definition. However, this is not the only strategy for guaranteeing compatibility. For example, an automatic code generator might hold the declaration of each object or function in a data dictionary and use that declaration in each translation unit that needs it.

E.1.4.1 The restrict type qualifier

Rule 8.14	Advisory

When an automatic code generator can determine that two pointers do not alias it may wish to use the *restrict* type qualifier.

E.1.5 Essential type

Rule 10.1	Advisory	Rule 10.2	Advisory	Rule 10.3	Advisory	Rule 10.4	Advisory
Rule 10.6	Advisory	Rule 10.7	Advisory	Rule 10.8	Advisory	Rule 14.4	Advisory
Rule 20.8	Advisory						

The *essential type* concept is one method for avoiding problems associated with *integer promotion* and the *usual arithmetic conversions*. An automatic code generator may adopt a different approach to avoiding such problems which is equally valid. Further, the automatic code generator is aware of both C's implicit type conversion rules and the sizes of the types on the target machine.

The rules relating to *essential type* are therefore all given advisory category when applied to automatically generated code.

E.1.6 Loop counters

Rule 14.1	Advisory

An automatic code generator is aware of the floating-point implementation and will therefore be aware of the implications of using a floating-point *loop counter*.

E.1.7 Labels and goto

Rule 15.2	Advisory	Rule 15.3	Advisory

An automatic code generator may need to generate a backwards jump, for example in the implementation of a state-machine. It is very unlikely to generate a jump into a block of injected manually-generated code and should therefore be able to avoid jumping over initialization.

E.1.8 Switch statements

Rule 16.1	Advisory	Rule 16.2	Advisory	Rule 16.3	Advisory	Rule 16.4	Advisory
Rule 16.5	Advisory	Rule 16.6	Advisory	Rule 16.7	Advisory		

If an automatic code generator determines that it can optimize away a *default clause* that is suggested by the model, for example every value for the controlling expression is covered by a *case clause*, then it may only do so if it inserts a comment into the generated C that explains why the *default clause* is absent. This comment can be reviewed and accepted as part of the MISRA C compliance argument.

An automatic code generator is capable of handling *switch clauses* that fall through into subsequent clauses.

Unusual *switch* statements such as those that have a controlling expression with *essentially Boolean* type, or that have only one *switch clause*, might be indicative of an error in human generated code but may be necessary or desirable in automatically generated code.

E.1.9 Readability

Dir 4.5	Readability	Rule 2.3	Readability	Rule 2.4	Readability	Rule 2.5	Readability
Rule 2.6	Readability	Rule 2.7	Readability	Rule 5.9	Readability	Rule 7.2	Readability
Rule 7.3	Readability	Rule 9.2	Readability	Rule 9.3	Readability	Rule 9.5	Readability
Rule 11.9	Readability	Rule 13.3	Readability	Rule 14.2	Readability	Rule 15.7	Readability
Rule 17.5	Readability	Rule 17.7	Readability	Rule 17.8	Readability	Rule 18.5	Readability
Rule 20.5	Readability						

These guidelines are for the benefit of human developers and reviewers.

If any identifier with internal linkage is declared in injected code, then Rule 5.9 as applied to that identifier is advisory for the entire translation unit.

If code is injected into an automatically generated *for* loop then Rule 14.2 is required insofar as the injected code causes a non-compliance.

E.2 Documentation requirements for automatic code generation tools

The developer of an automatic code generation tool shall provide documentation for each item listed in this section.

E.2.1 Implementation-defined behaviour and language extensions

In accordance with Dir 1.1, the use of any implementation-defined behaviour on which automatically generated code depends shall be documented.

In accordance with Rule 1.2, the use of language extensions by automatically generated code, and the methods by which their use is assured, shall be documented.

In accordance with Dir 4.2, the use of assembly language by automatically generated code shall be documented.

E.2.2 The essential type model

If an automatic code generator does not use the *essential type* model, then the strategy that it uses in its place shall be documented.

E.2.3 Run-time errors

An automatic code generator should select an appropriate code generation strategy to minimize the possibility for run-time failures, as required by Dir 4.1. If the possibility for run-time failures remains, the error handling strategy used by the code generator shall be documented. Since it is likely that manually generated code will be needed in order to handle the error, the error handling interface shall be documented as part of the overall error handling strategy.

Appendix F Process and tools checklist

This Appendix provides a checklist of the development process and tool use guidance that need to be followed in order to claim MISRA C compliance as described in Section 5.3.

Section	Guidance
3.1	A choice has been made between C90 and C99
4.2	There is a process for dealing with deficiencies in the C implementation
5.3.1	The translator has been configured to accept the correct version of the C language
5.3.1	The translator has been configured to generate an appropriate level of diagnostic information
5.3.1	The translator has been configured appropriately for the target machine
5.3.1	The translator's optimization level has been configured appropriately
5.3.3	There is a process for investigating and resolving any diagnostic messages produced by the translator
5.3	There is a compliance matrix showing how compliance with each MISRA C guideline is to be checked
5.3.2	The analysis tools have been configured to accept the correct version of the C language
5.3.2	The analysis process can deal with any language extensions that have been used
5.3.2	The analysis tools have been configured for the implementation, for example to be aware of the sizes of the integer types
5.3.3	There is a process for investigating and resolving any diagnostic messages produced by the analysis tools
5.4	There is a deviation process for recording and approving deviations
5.2.1	There is a process for ensuring that the program has sufficient resources, such as processing time and stack space
5.2.1	There is a process for demonstrating and recording the absence of run-time errors, for example in module designs

Appendix G Implementation-defined behaviour checklist

This Appendix provides the checklist of implementation-defined behaviours referred to in Dir 1.1.

C90 Annex	Item	Implementation-defined behaviour
G.3.1	1	How a diagnostic is identified (5.1.1.3).
G.3.2	1	The semantics of the arguments to main (5.1.2.2.1).
G.3.3	1	The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2).
	2	The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2).
	3	Whether case distinctions are significant in an identifier with external linkage (6.1.2).
G.3.4	1	The members of the source and execution character sets, except as explicitly specified in this International Standard (5.2.1).
	3	The number of bits in a character in the execution character set (5.2.4.2.1).
	4	The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4).
	5	The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (6.1.3.4).
	6	The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4).
	7	The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4).
G.3.5	1	The representations and sets of values of the various types of integers (6.1.2.5).
	4	The sign of the remainder on integer division (6.3.5).
G.3.6	1	The representations and sets of values of the various types of floating-point numbers (6.1.2.5).
	2	The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3).
	3	The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4).
	4	The order of allocation of bit-fields within a unit (6.5.2.1).
	5	Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).
G.3.10	1	What constitutes an access to an object that has volatile-qualified type (6.5.5.3).
G.3.13	1	Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).
	2	The method for locating includable source files (6.8.2).
	3	The support of quoted names for includable source files (6.8.2).
	4	The mapping of source file character sequences (6.8.2).
	5	The behavior on each recognized #pragma directive (6.8.6).
	6	The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.8.8).
G.3.14	5	Whether the mathematics functions set the integer expression errno to the value of the macro ERANGE on underflow range errors (7.5.1).
	30	The set of environment names and the method for altering the environment list used by the getenv function (7.10.4.4).

C99 Annex	Item	Implementation-defined behaviour
J.3.1	1	How a diagnostic is identified (3.10, 5.1.1.3).
J.3.2	2	The name and type of the function called at program startup in a freestanding environment (5.1.2.1).
	3	The effect of program termination in a freestanding environment (5.1.2.1).
	4	An alternative manner in which the main function may be defined (5.1.2.2.1).
	5	The values given to the strings pointed to by the argv argument to main (5.1.2.2.1).
	10	The set of environment names and the method for altering the environment list used by the getenv function (7.20.4.5).
J.3.3	1	Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).
	2	The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
J.3.4	1	The number of bits in a byte (3.6).
	2	The values of the members of the execution character set (5.2.1).
	4	The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).
	6	The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).
	7	The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).
	8	The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).
	9	The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).
	10	The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).
	11	The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).
J.3.5	1	Any extended integer types that exist in the implementation (6.2.5).
	2	Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

C99 Annex	Item	Implementation-defined behaviour
J.3.6	1	The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (5.2.4.2.2).
	2	The rounding behaviors characterized by non-standard values of FLT_ROUNDS (5.2.4.2.2).
	3	The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (5.2.4.2.2).
	4	The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).
	5	The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).
	6	How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).
	7	Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (6.5).
	8	The default state for the FENV_ACCESS pragma (7.6.1).
	9	Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).
	10	The default state for the FP_CONTRACT pragma (7.12.2).
	11	Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).
	12	Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).
J.3.9	2	Allowable bit-field types other than _Bool, signed int, and unsigned int (6.7.2.1).
	3	Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).
	4	The order of allocation of bit-fields within a unit (6.7.2.1).
J.3.10	1	What constitutes an access to an object that has volatile-qualified type (6.7.3).
J.3.11	1	The locations within #pragma directives where header name preprocessing tokens are recognized (6.4, 6.4.7).
	2	How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).
	3	Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).
	4	Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).
	5	The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).
	6	How the named source file is searched for in an included delimited header (6.10.2).
	7	The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).
	9	Whether the # operator inserts a character before the character that begins a universal character name in a character constant or string literal (6.10.3.2).
	10	The behavior on each recognized non-STDC #pragma directive (6.10.6).
	11	The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).

C99 Annex	Item	Implementation-defined behaviour
J.3.12	1	Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).
	4	Whether the feraiseexcept function raises the “inexact” floating-point exception in addition to the “overflow” or “underflow” floating-point exception (7.6.2.3).
	5	Strings other than C and that may be passed as the second argument to the setlocale function (7.11.1.1).
	6	The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than 0 (7.12).
	9	The values returned by the mathematics functions on underflow range errors, whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, and whether the “underflow” floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero. (7.12.1).
	11	The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).
	33	The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the strtod, strtof, strtold, wcstod, wcstof, or wcstold function (7.20.1.3, 7.24.4.1.1).
	34	Whether or not the strtod, strtof, strtold, wcstod, wcstof, or wcstold function sets errno to ERANGE when underflow occurs (7.20.1.3, 7.24.4.1.1).
	37	The termination status returned to the host environment by the abort, exit, or _Exit function (7.20.4.1, 7.20.4.3, 7.20.4.4).
	44	Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.9).
J.3.13	1	The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.18.2, 7.18.3).
	2	The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

Appendix H Undefined and critical unspecified behaviour

This Appendix identifies the undefined and critical unspecified behaviours that are referred to by Rule 1.3.

H.1 Undefined behaviour

The columns in this table have the following meanings:

- **C90 Id** is the number of the undefined behaviour in Annex G of The C90 Standard; where the behaviour is mentioned in the body of The Standard but not listed in Annex G, a * character is shown;
- **C99 Id** is the number of the undefined behaviour in Annex J of The C99 Standard;
- **Decidable?** is "Yes" or "No" according to whether detecting instances of the behaviour is, in general, decidable or not;
- **Guidelines** lists the MISRA C Guidelines which, if complied with, avoid the undefined behaviour;
- **Notes** provides additional notes on the behaviour including information on rules that may help to avoid the behaviour even if it cannot be totally avoided.

If a particular undefined behaviour has no entry in the "Guidelines" column then an instance of that behaviour in a program is a violation of Rule 1.3.

Note: it is assumed that any code that is written in other language and linked with the program does not introduce undefined behaviour directly or indirectly. For example, assembly language modules might define overlapping objects which, if accessed from C, would lead to undefined behaviour even though this might not be apparent from the C source code.

Id		Decidable	Guidelines	Notes
C90	C99			
	1		N/A	This behaviour is listed in C99 but each such instance is also given its own entry in Annex J. The entry for this behaviour is therefore redundant.
1	2	Yes		
	3	Yes		
	4	Yes		
2		Yes		
	5	Yes		
3		Yes	Rule 20.10	
	6	Yes		
5		Yes	Rule 5.2	
6		Yes	Rule 17.3	
8	7	Yes		
	8	No	Dir 4.12, Rule 18.6, Rule 21.3	
9		No	Dir 4.12, Rule 18.6, Rule 21.3	

Id		Decidable	Guidelines	Notes
C90	C99			
	9	No	Dir 4.12, Rule 18.6, Rule 21.3	
41		No	Rule 9.1	
	10	No		Compliance with Rule 9.1 avoids a common cause of this undefined behaviour but it is not sufficient to avoid all situations in which an indeterminate value might arise.
	11	No		The following rules help to avoid this behaviour: Rule 9.1, Rule 11.2, Rule 11.3, Rule 11.4, Rule 11.5 and Rule 19.1. However, if a trap representation is copied into an object that does not have character type, for example using memmove, memcpy or via a pointer to character type as permitted by the exception of Rule 11.3, it is not possible to avoid this behaviour.
	12	No	Rule 11.2, Rule 11.3, Rule 11.4, Rule 11.5	
	13	No		The following rules help to avoid this behaviour: Rule 9.1, Rule 10.1, Rule 11.2, Rule 11.3, Rule 11.4, Rule 11.5, and Rule 19.1. However, if the Exception of Rule 11.3 is used then it is not possible to prevent generation of a negative zero.
10	14	Yes	Rule 5.6, Rule 5.7, Rule 8.3	
15		No	Dir 4.1, Rule 10.3	
	15	No	Dir 4.1, Rule 10.3	
	16	No	Dir 4.1, Rule 10.3	
	17	No	Rule 9.1, Rule 11.2, Rule 11.3, Rule 11.4, Rule 11.5, Rule 19.1	
16	18	Yes		
	19	Yes		
17	20	Yes		
*	21	No	Rule 11.1, Rule 11.2, Rule 11.4, Rule 11.6	
	22	No	Rule 11.2, Rule 11.3, Rule 11.5	
27	23	No	Rule 11.1	
4	24	Yes		
*	25	Yes		
	26	Yes		
	27	Yes		
7	28	Yes	Rule 5.1, Rule 5.2, Rule 5.3, Rule 5.4, Rule 5.5	
	29	Yes	Rule 21.2	
11		Yes		
12	30	No	Rule 7.4, Rule 11.4, Rule 11.8	
13		Yes		

Id		Decidable	Guidelines	Notes
C90	C99			
14		Yes	Rule 20.2	
	31	Yes	Rule 20.2	
18	32	No	Rule 13.2, Rule 13.3, Rule 13.4	
19	33	No	Dir 4.1	
20		No	Rule 11.3, Rule 11.4, Rule 11.5	
	34	No	Rule 11.3, Rule 11.4, Rule 11.5	
	35	Yes		
21		Yes		
22	36	No	Rule 8.2, Rule 17.3	Rule 17.3 is only applicable to, and only required for, C90
25		No	Rule 8.4, Rule 8.5, Rule 11.1, Rule 21.2, Rule 17.3	
	37	No	Rule 8.4, Rule 8.5, Rule 11.1, Rule 21.2	
23		No	Rule 8.2, Rule 17.3	
	38	No	Rule 8.2	
24		No	Rule 5.6, Rule 5.7, Rule 8.3, Rule 8.4, Rule 8.5, Rule 11.1, Rule 21.2	
	39	No	Rule 5.6, Rule 5.7, Rule 8.2, Rule 8.3, Rule 8.4, Rule 8.5, Rule 11.1, Rule 21.2	
26	40	No	Dir 4.1	
28		Yes	Rule 11.1	
29	41	Yes	Rule 11.1, Rule 11.2, Rule 11.6, Rule 11.7	
	42	No	Dir 4.1	
30	43	No	Rule 18.1	
*	44	No	Rule 18.1	
31	45	No	Rule 18.2	
	46	No	Rule 18.1	
*	47	No		
32	48	No	Rule 12.2	
	49	No		Compliance with Rule 10.1 avoids this undefined behaviour except when the expression being left-shifted has an unsigned type that is promoted to a signed type.
33	50	No	Rule 18.3	
34	51	No	Rule 19.1	
*	52	Yes		
*	53	Yes		

Id		Decidable	Guidelines	Notes
C90	C99			
*	54	Yes		
*	55	Yes		
35	56	Yes		
36	57	Yes		
37	58	Yes		
38		Yes	Rule 6.1	
	59	No	Rule 18.7	
	60	Yes		
39	61	No	Rule 11.4, Rule 11.8, Rule 19.2	
40	62	No	Rule 11.4, Rule 11.8, Rule 19.2	
*	63	Yes		
*	64	Yes		
	65	No	Rule 8.14	
	66	No	Rule 8.14	
	67	Yes	Rule 8.10	
*	68	Yes		
	69	No	Rule 18.8	
	70	No	Rule 18.8	
	71	No	Rule 17.6	
	72	Yes		
*	73	Yes	Rule 8.2, Rule 11.1	
*	74	No		
*	75	Yes		
42		Yes	Rule 9.2	
	76	Yes	Rule 9.2	
	77	Yes	Rule 9.2	
44	78	Yes	Rule 8.6	
	79	Yes	Rule 8.2	
*	80	Yes		
45	81	Yes	Rule 17.1	
43	82	Yes	Rule 17.4	
46	83	Yes		
47	84	Yes		
48	85	Yes	Rule 20.3	
	86	Yes		
49		Yes		
50	87	Yes	Rule 20.6	

Id		Decidable	Guidelines	Notes
C90	C99			
51	88	Yes	Rule 20.10	
52	89	Yes	Rule 20.10	
53	90	Yes		
	91	Yes		
	92	Yes		
54	93	Yes	Rule 21.1	
55	94	No		Compliance with Rule 19.1 avoids a common cause of this undefined behaviour but does not prevent copying part of an object to another part of the same object, such as an array.
*	95	Yes		
56		Yes	Rule 17.3, Rule 20.1, Rule 20.4, Rule 21.2	
	96	Yes	Rule 20.1	
	97	Yes	Rule 20.1, Rule 21.2	
	98	Yes	Rule 20.4	
57		Yes	Rule 21.1, Rule 21.2	
	99	Yes	Rule 21.2	
	100	Yes	Rule 21.1, Rule 21.2	
	101	Yes	Rule 21.1	
60	102	No	Dir 4.11	
*	103	No	Dir 4.11	
61		Yes	Rule 17.3, Rule 21.2	
62	104	Yes		Compliance with Rule 21.1 prevents undefinition of the macro but no rule prevents the macro expansion from being suppressed, e.g. by means of `(assert)(E)`
	105	Yes		
	106	Yes		
63	107	No	Dir 4.11	
58		Yes	Rule 21.1	
	108	Yes		Compliance with Rule 21.1 prevents undefinition of the macro but no rule prevents the macro expansion from being suppressed, e.g. by means of `(errno)` if it is implemented as a function-like macro. Compliance with Rule 21.2 prevents definition of the identifier errno.
	109	No		Compliance with Rule 21.12 avoids some instances of this undefined behaviour but does not prevent floating-point control modes from being changed.
	110	No	Rule 21.12	
	111	No	Rule 21.12	
	112	No	Dir 4.11	
90		No	Rule 21.7	
94		No		

Id		Decidable	Guidelines	Notes
C90	C99			
	113	No		
*	114	No		
*	115	No		
	116	Yes	Rule 21.1, Rule 21.2	
	117	Yes		
64		Yes	Rule 21.1, Rule 21.2, Rule 21.4	
	118	Yes	Rule 21.1, Rule 21.2, Rule 21.4	
65	119	Yes	Rule 21.4	
*	120	No	Rule 21.4	
66	121	No	Rule 21.4	
67		No	Rule 21.4, Rule 21.5	Compliance with either rule is sufficient to avoid the undefined behaviour.
*	122	No	Rule 21.5	
*	123	No	Rule 21.5	
	124	No	Rule 21.5	
68		No	Rule 21.5	
	125	No	Rule 21.5	
69	126	No	Rule 21.5	
*	127	No	Rule 21.5	
*	128	No		Compliance with Rule 17.1 avoids instances of this undefined behaviour that arise through improper use of the features of <stdarg.h>.
70	129	No	Rule 17.1	
71		Yes	Rule 17.1, Rule 21.1, Rule 21.2	
	130	Yes	Rule 17.1, Rule 21.1, Rule 21.2	
75		No	Rule 17.1	
76		No	Rule 17.1	
	131	No	Rule 17.1	
	132	Yes	Rule 17.1	
73		No	Rule 17.1	
74		No	Rule 17.1	
	133	No	Rule 17.1	
	134	No	Rule 17.1	
72	135	Yes	Rule 17.1	
59	136	Yes		
	137	Yes		
	138	No	Rule 21.6	
	139	No	Rule 21.6	

Id		Decidable	Guidelines	Notes
C90	C99			
*	140	No	Rule 21.6	If Rule 21.6 is deviated then Rule 22.6 provides protection against this undefined behaviour. Rule 21.6 is preferred as it is decidable.
77	141	No	Rule 21.6	
	142	No	Rule 21.6	
78	143	No	Rule 21.6	
*	144	No	Rule 21.6	
79		No	Rule 21.6	
85		No	Rule 21.6	
	145	No	Rule 21.6	
	146	No	Rule 21.6, Rule 21.10	
*	147	No	Rule 21.6	
*	148	No	Rule 21.6	
83		No	Rule 21.6	
84		No	Rule 21.6	
	149	No	Rule 21.6	
82		No	Rule 21.6	
87		No	Rule 21.6	
	150	No	Rule 21.6	
*	151	No	Rule 21.6	
	152	No	Rule 21.6	
81	153	No	Rule 21.6	
97		No	Rule 21.10	
80	154	No	Rule 21.6, Rule 21.10	
86	155	No	Rule 21.6	
89	156	No	Rule 21.6	
*	157	No	Rule 21.6	
	158	No	Rule 21.6	
88	159	No	Rule 21.6	
*	160	No	Rule 21.6	
*	161	No	Rule 21.6	
*	162	No	Rule 21.6	
*	163	No	Rule 21.6	
*	164	No	Rule 21.6	
*	165	No	Rule 21.6	
*	166	No	Rule 21.6	
*	167	No	Rule 21.3	
91	168	No	Rule 21.3	
92	169	No	Rule 21.3, Rule 22.2	

Id		Decidable	Guidelines	Notes
C90	C99			
*	170	No	Rule 21.3	
*	171	No	Rule 21.3	
93	172	No	Rule 21.8	
	173	No	Rule 21.4	
*	174	No		Compliance with Rule 21.8 avoids this undefined behaviour in respect of getenv only.
	175	No	Rule 21.8	
	176	No	Rule 21.9	
	177	No	Rule 21.9	
*	178	No	Rule 21.9	
95	179	No		
96	180	No	Dir 4.11	
	181	No	Dir 4.11	
*	182	No		Compliance with Rule 21.10 avoids this undefined behaviour except in respect of wcsxfrm.
	183	No	Dir 4.11	
	184	Yes	Rule 21.11	
	185	Yes	Rule 21.11	
	186	No	Rule 21.6	
	187	No	Dir 4.11	
	188	No		
	189	No	Dir 4.11	
	190	No		
	191	No		

H.2 Critical unspecified behaviour

The columns in this table have the following meanings:

- **C90 Id** is the number of the unspecified behaviour in Annex G of The C90 Standard; where the behaviour is mentioned in the body of The Standard but not listed in Annex G, a * character is shown;
- **C99 Id** is the number of the unspecified behaviour in Annex J of The C99 Standard;
- **Critical?** is "Yes" or "No" according to whether reliance on this behaviour is likely to lead to unexpected program operation;
- **Guidelines** lists the MISRA C Guidelines which, if complied with, avoid the unspecified behaviour;
- **Notes** provides additional notes on the behaviour including information on rules that may help to avoid the behaviour even if it cannot be totally avoided.

If a particular unspecified behaviour is marked as being critical but has no entry in the "Guidelines" column then reliance on that behaviour in a program is a violation of Rule 1.3.

Note: it is assumed that any code that is written in other language and linked with the program does not rely on unspecified behaviour directly or indirectly. For example, assembly language functions might attempt to access parameters despite their layout being unspecified.

Id		Critical	Guidelines	Notes
C90	C99			
1	1	No		
	2	No		
2	3	No	Rule 21.6	
3	4	No	Rule 21.6	
4	5	No	Rule 21.6	
5	6	No	Rule 21.6	
	7	Yes	Rule 5.1	
6		Yes		
	8	Yes		
	9	Yes		
	10	Yes	Rule 19.2	
	11	Yes		
	12	Yes		
	13	Yes		Compliance with Rule 10.1 avoids generation of negative zeros when operating on expressions that have a signed type before promotion.
	14	Yes	Rule 7.4	
7, 8	15	Yes	Rule 13.2	
9	16	Yes	Rule 13.2	
	17	Yes	Rule 13.1	
7	18	Yes	Rule 13.2	
10	19	No		

Id		Critical	Guidelines	Notes
C90	C99			
	20	Yes	Rule 8.10	
	21	Yes	Rule 13.6, Rule 18.8	
7	22	Yes	Rule 13.1	
11	23	No		
*	24	Yes		
12	25	Yes	Rule 20.10, Rule 20.11	
13	26	No		
	27	Yes	Rule 21.12	
	28	Yes	Rule 21.12	
	29	No		
	30	Yes	Dir 4.11	
	31	Yes	Dir 4.11	
14	32	No	Rule 21.4	
15	33	No	Rule 17.1	
	34	Yes	Rule 21.6	
16	35	Yes	Rule 21.6	
17	36	Yes	Rule 21.6	
18	37	Yes	Rule 21.6	
	38	No		
19	39	No	Rule 18.1, Rule 18.2, Rule 18.3, Rule 21.3	Compliance with either Rule 21.3 or all of Rule 18.1, Rule 18.2 and Rule 18.3 will avoid this unspecified behaviour
	40	Yes	Rule 21.3	
20	41	Yes	Rule 21.9	
21	42	Yes	Rule 21.9	
22	43	Yes	Rule 21.10	
	44	Yes	Rule 21.10	
	45	Yes		
	46	Yes		
	47	Yes		
	48	Yes	Dir 4.11	
	49	Yes	Dir 4.11	
	50	Yes	Dir 4.11	

Appendix I Example deviation record

Project	F10_BCM	Deviation ID	R_00102
MISRA C Ref	Rule 10.6	Status	Approved
Source	Tool: MMMC	Scope	Project

Raised by	E C Unwin	Approved by	D B Stevens
	Signature		*Signature*
Position	Software Team Leader	Position	Engineering Director
Date	27-Jul-2012	Date	12-Aug-2012

I.1 Summary

The rationale for MISRA C:2012 Rule 10.6 is that it avoids potential developer confusion regarding the type in which some arithmetic operations take place. Unfortunately, because of a "feature" of the compiler being used on this project, it is not possible to make the code comply with the rule without incurring serious run-time performance degradation. The impact of this is that the software's timing requirements cannot be satisfied in all cases and there is a significant risk that the vehicle's legislated hydrocarbon emissions target will not be met.

I.2 Detailed Description

The project makes use of several variable time-step integrators which accumulate a multiple-precision long-term sum. The quantity to be integrated and the integration time-step are both 16-bit unsigned quantities which need to be multiplied to give a 32-bit result which is then accumulated.

Since the C compiler in use on this project implements the *int* type in 32-bits, the code to compute the 32-bit product is:

```
extern uint16_t qty, time_step;

uint32_t  prod = ( uint32_t ) qty * ( uint32_t ) time_step;
```

Clearly, even though the operands of the multiplication operator are 32-bit, there is no possibility that the product is too large for 32-bits because both operands were zero-extended from 16-bits.

In accordance with our standard development procedures, all object modules are passed through a worst-case execution time analysis tool to ensure that each function meets its execution time budget as specified in the architectural design. The analyser highlighted that the code generated for these integrators was far in excess of the budget laid out for them. Investigation revealed that the reason for the excessive execution time was that the compiler was generating a call to a "shift-and-add" style of long multiplication routine. This is surprising because the processor is equipped with an *IMUL* instruction that is capable of multiplying two 16-bit unsigned integers to deliver a 32-bit result in a single clock cycle. Although the multiplication operands are 32-bit, the compiler has the means to know that the most significant 16-bits of these operands are 0 so it should be capable of selecting the *IMUL* instruction.

Experimentation with the compiler suggests that it will select the *IMUL* instruction provided that the operands of the multiplication are implicitly converted from 16-bit to 32-bit, i.e.

```
uint32_t  prod = qty * time_step;
```

This is particularly odd because the behaviour of the code as described by the C Standard is independent of whether the conversion is implicit or explicit. While the program will generate the same results regardless of whether *IMUL* or a library call is used for multiplication, the library call requires a worst case of 100 cycles to execute. The compiler vendor has confirmed in writing that this is the behaviour they expect under the circumstances.

I.3 Justification

Since this type of integrator is used in several functions in the project and is executed at least once every 100 microseconds on average, the performance of the library function is not acceptable. At the specified CPU internal frequency of 25 MHz this means that 4% of the time is spent just on these multiplications. This in itself is insignificant and can be contained in the headroom available in the overall timing budget. However, the design specifies that the integrator shall make its result available within a maximum of 10 microseconds in order to satisfy timing requirements of other functions. The failure to meet this requirement means that there is significant risk in achieving the emissions target, the commercial implications of which are in excess of $10m.

Our preferred solution to this problem is to write the integrators using implicit conversions. This would require deviation against MISRA C:2012 Rule 10.6 for instances of such an integrator. The code is functionally identical to that generated by the MISRA-compliant code but executes up to 100 times faster.

The following other options were considered:

- Increase the clock speed — to achieve the required performance would require a 10-fold increase in clock but the processor's maximum PLL frequency is 100 MHz;
- Change processor — not commercially viable given that hardware design validation is well underway; the additional costs to the project would be around $250 000 and there would be a timing impact too;
- Change compiler — there is no other commercially recognized compiler for this processor; there is an unsupported public domain compiler but it is not considered of suitable quality for this project;
- Recode the library routine — the library uses a base-2 long multiplication; it could be recoded to implement a base-65 536 long multiplication using 3 *IMUL* instructions but we are reluctant to make changes to the compiler vendor's code; we have sought their views on this approach and received the response that "they could not support us making changes to their library".

I.4 Conditions under which the deviation is requested

This deviation is requested for all instances of variable time-step integrators within the project.

I.5 Consequences of non-compliance

There are no consequences associated with non-compliance with MISRA C:2012 Rule 10.6 in the circumstances described in this deviation record.

There are no additional verification and validation requirements resulting from this deviation.

I.6 Actions to control reporting

The MMMC tool used to check compliance with this rule provides a facility whereby a diagnostic message can be suppressed within an expression. Since all integrators are of the form:

```
prod = qty * time_step;
```

a macro can be used to implement the integrator and suppress the warning. The following macro will be used to implement the multiplication and assignment of its result to the product term:

```
/* Violates Rule 10.6: See deviation R_00102 */
#define INTEG(prod, qty, time_step) \
   ( /* -mmmc-R10_6 */ (prod) = (qty) * (time_step) /* -mmmc-pop */ )
```

Although this macro could be implemented as a function, the overhead of the call and return is excessive given the simplicity of the operation being performed. A macro is therefore preferred to a function in this instance even though this means violating Dir 4.9.

Appendix J Glossary

Assigned

An expression is *assigned* if it is the subject of an *assignment*.

Assignment

It is sometimes convenient to use the term *assignment* to denote any operation which takes place as if it were by assignment. The operations covered by this term are:

- Assignment by means of one of the assignment operators;
- Passing an argument to a function, in which case the argument is copied as if by assignment to the corresponding parameter;
- Returning an expression from a function, in which case the result is copied as if by assignment to an object with the function's return type;
- Using an expression to initialize all or part of an object, including a compound literal in C99, in which case the expression is copied as if by assignment to the destination.

Code

Code consists of everything within a translation unit that is not excluded by conditional compilation.

Dead code

Any operation whose result does not affect the program's behaviour is *dead code*.

Note: initialization is not the same as an assignment operation and is therefore not a candidate for *dead code*.

Examples of *dead code* include:

- Storing values into non-*volatile* objects that are never subsequently used;
- An expression statement that does not assign a value to an object.

The following are never *dead code*:

- Accessing a *volatile* object;
- Using a language extension.

Since the time at which actions occur is often an important aspect of the behaviour of an embedded program, it follows that code whose effect is to insert a delay is not *dead code*. However, any such code is very likely to be written with reference to timer hardware and will therefore involve *volatile* object accesses in any event. For example:

```
extern const volatile uint16_t MILLISEC_TIMER;

void delay_ms ( uint16_t n )
{
  uint16_t  start_time = MILLISEC_TIMER;

  while ( ( MILLISEC_TIMER - start_time ) < n )
  {
    /* wait */
  }
}
```

External identifier

An *external identifier* is an identifier with external linkage and must therefore denote either an object or a function.

Header file

A *header file* is any file that is the subject of a *#include* directive.

Note: the filename extension is not significant.

Inline function

An *inline function* is one that is declared with the *inline* function specifier.

Loop counter

A *loop counter* is defined in Section 8.14 .

Macro identifier

A *macro identifier* is an identifier that is either a macro name or a macro parameter.

Opaque type

A type whose implementation detail is not made available to its users.

Persistent side effect

A *side effect* is said to be *persistent* at a particular point in execution if it might have an effect on the execution state at that point. All of the following *side effects* are *persistent* at a given point in the program:

- Modifying a file;
- Modifying an object;
- Accessing a *volatile* object.

When a function is called, it may have *side effects*. Modifying a file or accessing a *volatile* object are persistent as viewed by the calling function. However any objects modified by the called function, whose lifetimes have ended by the time it returns, do not affect the caller's execution state. Any *side effects* arising from modifying such objects are not *persistent* as viewed by the caller.

For example:

```
uint16_t f ( void )
{
  uint16_t  temp = 1;           /* Side effect of modifying temp is not
                                 * persistent to the caller */

  return ( temp );
}

x = f ( );
```

Project

A *project* consists of the *code* from the set of translation units used to build a single executable. A typical embedded controller might consist of several logically distinct projects, for example:

- Primary bootloader;
- Secondary bootloader;
- Main application.

Prototype form

A function type is said to be in *prototype form* if it includes a prototype, *i.e.* it specifies a list of parameter types and, optionally, names. For example, the following are all in *prototype form*:

```
void  f ( void );            /* Prototype specifying no parameters      */
void  g ( int16_t, char );   /* Prototype specifying types but no names */
void  h ( uint32_t n );      /* Prototype specifying type and name      */
```

The following are not in *prototype form*:

```
void      f ( );               /* No information about parameters      */

void g ( x, y )                /* Function definition "K&R"-style      */
int16_t x;
char y;
{
}
```

Standard type

The *standard type* of an expression is its type as determined according to The Standard.

Unreachable code

Code is unreachable if, in the absence of hardware failures and ignoring the effects of undefined behaviour, there is no combination of program inputs that can cause it to be executed.

Unreachable code may result from the structure of the program's control flow graph. For example, any code following a *return* statement is unreachable and some operands of logical or conditional operators may be unreachable:

```
uint16_t  y;

uint16_t f ( void )
{
  uint16_t  x;

  return true ? y : ( y + 1U );     /* y + 1 is never executed     */
  x = 10U;                          /* statement is never executed */
}
```

Unreachable code may also result when there is a control flow path to a statement but that path can never be followed, for example:

```
uint16_t  x, y;

switch ( 2u * x )
{
case 0:
  y = 10u;
  break;
case 1:
  y = 0u;   /* unreachable: 2 * x can never be odd */
  break;

case 2:
  y = x;
  break;
default:
  break;
}
```

Used

An expression is said to be *used* if it appears in one of the following contexts:

- The operand of an operator, including a parameter of a function call operator;
- The controlling expression of an *if*, *while*, *for*, *do ... while*, or *switch* statement;
- The value returned by a *return* statement;
- An *initializer*.

This term should not be confused with reading the value of an object, for example:

```
uint16_t  x = 3;
uint16_t *p = &x;               /* x is used but not read */

return *p;                      /* x is read but not used */
```

ISBN 978-1-906400-10-1 paperback
ISBN 978-1-906400-11-8 PDF