# Comprehensive Guide to MISRA C for Scoping and Compliance

## Introduction: The Purpose of MISRA

MISRA (Motor Industry Software Reliability Association) provides a set of guidelines for developing high-integrity software in embedded systems. While originally created for the automotive industry, these guidelines have been adopted across various sectors, including aerospace, medical, and industrial control systems. The primary goal of MISRA is to facilitate the creation of robust, safe, and secure code by restricting the use of certain language features that are known to be problematic, ambiguous, or error-prone.

A **MISRA compliance effort** is not about passing a test, but about a disciplined approach to software development. Scoping is the critical first step, where you define which rules apply to your project and how compliance will be verified.

## Core Concepts and Rule Classifications

Understanding the different classifications of MISRA rules is fundamental to scoping your project correctly.

### 1. Rules vs. Directives

* **Rules:** These are guidelines that can be checked by analyzing the source code alone. A static analysis tool can often determine if a rule has been violated.
* **Directives:** These are guidelines that require checking of the project's documentation, processes, or architecture. They cannot be verified by just looking at the code.

### 2. Rule Enforcement Categories

Each rule in MISRA C:2012 (and its successor, MISRA C:2023) falls into one of three categories. This classification dictates how strictly a rule must be followed.

* **Mandatory Rules:** These rules must be complied with under all circumstances. There is no official deviation process for Mandatory rules; a violation indicates a critical defect and means the project cannot claim full MISRA compliance. They typically deal with fundamental issues of code safety, such as avoiding undefined behavior.
* **Required Rules:** These rules must be complied with unless a formal deviation has been granted. A **deviation** is a formal, documented justification for why a rule must be violated in a specific instance (e.g., for performance reasons, or to interface with specific hardware). This is the largest category of rules.
* **Advisory Rules:** These are recommendations for good practice. They are intended to improve code clarity, maintainability, and efficiency. They should be followed where possible, but non-compliance does not require a formal deviation.

## Selected Rule Examples and Rationale

Below are examples of rules from each category, along with an explanation of their purpose and a code snippet to illustrate the concept. This is a partial list designed for educational purposes and is not a substitute for the official standard.

### Mandatory Rules

**Rule 1.3: There shall be no more than one definition for any object or function with internal linkage in the same translation unit.**

* **Purpose:** Prevents subtle linking errors. While a compiler might not complain, having multiple definitions can lead to unexpected behavior and makes code difficult to maintain.

**Rule 2.1: A project shall not contain any unreachable code.**

* **Purpose:** Unreachable code is dead code that can never be executed. It is a sign of a logic error or poor design, and it increases the cognitive load for maintenance and debugging.

**Rule 8.1: An object or function with external linkage shall have exactly one definition.**

* **Purpose:** Ensures the linker can successfully resolve all external symbols. Multiple definitions lead to a linker error, while a missing definition leads to an "unresolved external" error.

**Rule 20.9: The human-readable characters used to form identifiers shall be restricted to a subset of the characters available in the Latin-1 character set.**

* **Purpose:** Promotes code portability by preventing the use of extended or non-standard characters in identifiers.

### Required Rules

**Rule 1.2: The language shall not be extended by the implementation.**

* **Purpose:** Prevents the use of non-standard, compiler-specific extensions (like \_\_attribute\_\_) that make code non-portable.

**Rule 4.1: The value of an integer literal shall not be too large for a standard integer type.**

* **Purpose:** Prevents integer overflow at compile time. Example: A literal 2147483648 is too large for a 32-bit signed integer and can lead to unexpected behavior.

**Rule 10.3: The value of an expression shall not be assigned to an object with a narrower essential type or different essential type category.**

* **Purpose:** Prevents implicit type conversions that can lead to data loss or unexpected behavior.

**Rule 11.3: A cast shall not be performed between a pointer to object type and a pointer to a different object type.**

* **Purpose:** Prevents unsafe pointer type-punning, which can violate strict aliasing rules and lead to undefined behavior. A deviation might be necessary for low-level memory operations.

**Rule 12.4: The right-hand side of a shift operator shall be less than the width of the promoted left-hand side.**

* **Purpose:** Prevents undefined behavior from shifting by a number greater than or equal to the size of the data type.

### Advisory Rules

**Rule 1.1: The human-readable characters used to form identifiers should be restricted to a subset of the characters available in the Latin-1 character set.**

* **Purpose:** Recommends a best practice to ensure identifiers are easily readable and portable across different systems and text encodings.

**Rule 8.4: A function identifier should be declared globally before its first use.**

* **Purpose:** Promotes good coding practice by ensuring function prototypes are visible to the compiler before the function is called. This prevents implicit function declarations, which can hide errors.

**Rule 17.5: The parameter to printf shall be a constant string, not a pointer to a modifiable string.**

* **Purpose:** Prevents a common security vulnerability known as a format string attack. Using a constant string ensures the format specifiers cannot be tampered with at runtime.

**Rule 20.4: Dynamic heap memory allocation should not be used.**

* **Purpose:** Dynamic memory allocation (malloc, calloc, free) can lead to unpredictable behavior, memory fragmentation, and leaks, which are unacceptable in safety-critical systems. This rule recommends avoiding it entirely. A formal deviation is often required for embedded systems with non-trivial memory management.

## The Scoping and Compliance Process

Scoping is the process of defining your project's MISRA strategy. It involves these key steps:

### Step 1: Standard Selection

Choose the appropriate MISRA standard for your project (e.g., MISRA C:2023 for new C projects, AUTOSAR C++14 for new C++ projects).

### Step 2: Tooling and Coverage

Select one or more static analysis tools. No single tool can cover every rule, so a combination of tools and manual reviews is often necessary. The tool’s vendor should provide a **compliance matrix** that outlines which rules are checked automatically.

### Step 3: Rule Tailoring

This is the most crucial part of scoping. You must create a project-specific subset of the standard.

* **For Mandatory rules:** Confirm that your development processes and tool chain will enforce them.
* **For Required rules:** Decide which rules are relevant and which will require a formal deviation.
* **For Advisory rules:** Decide which of these recommendations your team will follow as a standard practice.

### Step 4: The Compliance Matrix

Create a central document, often a spreadsheet, that serves as your project's single source of truth for MISRA compliance. This matrix should list every rule and directive and specify:

* The rule's category.
* How compliance is verified (e.g., "Tool X," "Manual Review," "Peer Review").
* The status of compliance for each rule (Compliant, Not Applicable, Deviated).
* A reference to any formal deviation records.

This matrix is a living document that will be maintained and used throughout the project's lifecycle to track progress and provide evidence for audits.