MISRA-C (QUICK overview)

Jaydeep Shah (radhey04ec@gmail.com)

What is MISRA C?

MISRA C = Motor Industry Software Reliability Association.

A set of coding guidelines created for safety-critical systems:

- Automotive
- Aerospace
- Defense
- Medical devices
- Industrial controllers

These systems cannot afford crashes, so MISRA provides strict rules to avoid unsafe C coding.

Why MISRA C Exists (Real reasons)?

- Normal C language is:
- Too flexible
- Allows dangerous behaviors
- No runtime protections
- Undefined behavior (UB)
- Implementation dependent features

MISRA C removes all risky features → makes software predictable, testable, safe.

MISRA C compliance is not a built-in feature of most standard IDEs; it is typically achieved using specialized third-party static analysis tools. Tools like Parasoft C/C++test integrate into popular IDEs to provide comprehensive MISRA rule enforcement and reporting.

Quick overview of some useful rules that need to follow during development:

1) Always initialize variables

Rule idea: Never read an automatic (local) variable before you set it to a known value.

Bad example:

```
int sum(void) {
   int s;            // uninitialized
   s += 10;       // UB: reading s to add to it
   return s;
}
```

Good example:

```
void f(void) {
   int x = 0;      // initialized
   if (x == 0) {
      do_something();
   }
}
```

2) No magic numbers

Rule idea: Use named constants instead of bare numbers so intent and range are clear.

Bad example:

```
buffer[index] = value;     // assume index < 256 somewhere else
if (speed > 63) stop();
```

Good example:

```
#define MAX_SPEED 63
#define BUFFER_SIZE 256

if (speed > MAX_SPEED) stop();
if (index < BUFFER_SIZE) buffer[index] = value;
```

3) Avoid dynamic memory in safety-critical code:

Rule idea: Prefer static (compile-time) allocations in safety-critical embedded systems.
It means No malloc() or calloc().

Why this is bad / effects

- Non-deterministic timing: allocation may take variable time.
- Fragmentation: long run-time can fragment heap, later malloc() fails.
- Failure handling complexity: must check every allocation and handle failures — often dangerous.
- Hard to certify: dynamic memory makes formal verification harder.

4) Bounds-check arrays

Rule idea: Never index arrays without first checking that the index is within valid bounds.

Bad example:

```
char buf[10];
```

```
buf[len] = '\0';        // if len >= 10 -> overflow
```

Good example:

```
if (len < sizeof(buf)) {
   buf[len] = '\0';
} else {
   // handle error
}
```

5) Avoid complex expressions with side-effects

Rule idea: Don't write expressions that change multiple variables or rely on unsequenced behavior. Break into simple steps.

How one-liner violates rule

```
a = b++  +  ++c;
```

modifying b and c inside expression and using them is confusing and can be UB depending on sequence points/ordering.

Bad example:

```
i = ++j + j++; // undefined (modifies j multiple times without sequence)
arr[i++] = i;   // undefined: modifying i and reading i in same expression
```

Good example: (Break into steps)

```
j = j + 1;
int tmp = j + (j - 1); // or whatever intended
i = tmp;

int idx = i;
i = i + 1;
arr[idx] = i;
```

6) Use braces {} for all control blocks

Rule idea: Always put braces around if, for, while bodies, even single-line ones.

How one-liner violates rule?

if (cond) do_a(); do_b(); — visually may look like both are in if, but only first is.

7) Switch must have default case

Rule idea: Always include a default: branch in switch so unexpected enumeration values are handled.

Good example:

```
switch (state) {
   case STATE_IDLE: do_idle(); break;
   case STATE_RUN:  do_run(); break;
   default:        handle_unexpected_state(); break;
}
```

8) Check all return values

Rule idea: Functions that return status or size must be checked and handle.

10) Explicit conversions

Rule idea: Avoid implicit casts that change type or sign; use explicit casts and check ranges.

That means no implicit narrowing or mixed signed/unsigned.

here is simple example

uint8_t x = 300; or int i = -1; unsigned u = i; — implicit conversions silently change value.

Bad example:

```
uint8_t x = 300;       // 300 truncated modulo 256 -> 44
```

```
int a = -1; unsigned b = a; // b becomes large positive
if (i < u) ...        // surprising result
```

Good example:

```
int temp = 300;
if (temp >= 0 && temp <= UINT8_MAX) {
   uint8_t x = (uint8_t)temp;
} else {
   // handle error
}
```