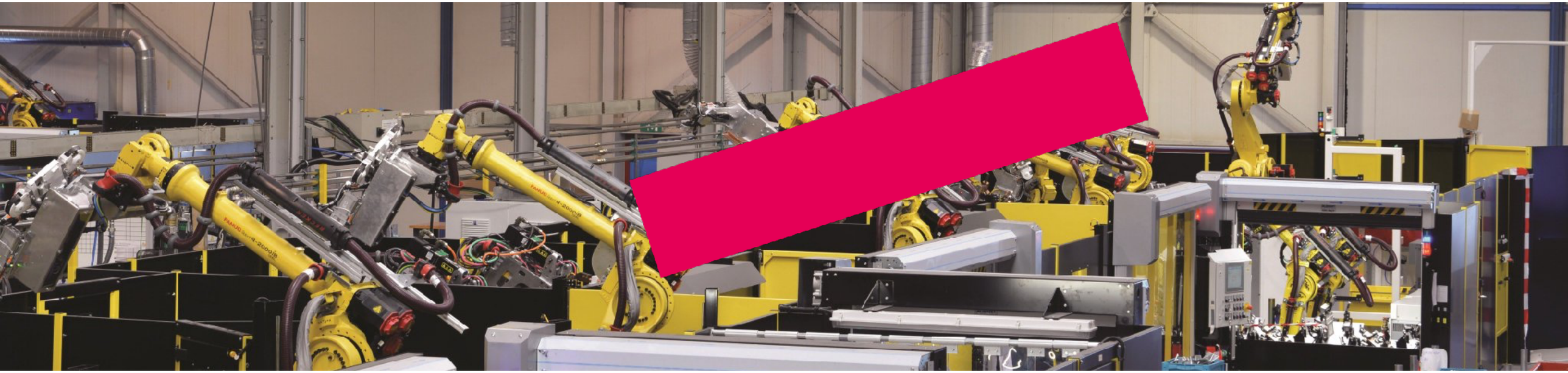
Programming 6



Workshop on Clean Code

johan.korten@han.nl

V1.0 Nov 2025

HAN_UNIVERSITY
OF APPLIED SCIENCES

Workshop Topics overview

Clean code (beyond SOLID)

CMake

Unit testing

Commenting

Patterns (Clean architecture)

A few reasons why do we need clean code…

All software:

- Maintainability (Code is read 10x more than written)
- Collaboration (Teams need shared understanding)

Mr. Clean Code: Robert C. Martin aka Uncle Bob

https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

Clean code: Maintainability

How do we solve maintainability and collaboration mostly?

Clean code: Maintainability

Clean code versus Clean architecture

A few reasons why do we need clean code…

All software:

- Maintainability (Code is read 10x more than written)
- Collaboration (Teams need shared understanding)

- Especially embedded:
  - Safety - Lives depend on it (medical, automotive, aerospace)
  - Certification - IEC 62304, ISO 26262, DO-178C compliance required
  - Resource constraints - Limited RAM/ROM, no dynamic allocation
  - Longevity - Products run 10-20+ years unchanged (e.g. MOBA)
  - Debugging difficulty - No printf, limited tooling, real-time constraints
  - Efficiency - E.g. battery life, etc.

When Code Quality Kills 1/2

Case 1: Therac-25 (1985-1987)

The Incident:

Radiation therapy machine delivered ***doses 100x higher than intended***

At least 6 accidents between 1985-1987

3 patients died, others severely injured

Code Problems:

- Race condition: changing treatment mode within 8 seconds left turntable in wrong position
- One-byte flag overflow: every 256th increment reset safety check to zero
- Hardware safety interlocks removed, relying solely on software
- Reused code from previous machines (Therac-6, Therac-20) without hardware safeguards
- Cryptic error messages ("MALFUNCTION 54") with no documentation
- No formal specifications, design documents, or test plans

When Code Quality Kills 2/2

Case 2: Toyota Unintended Acceleration (2000s-2010)

The Incident:

Reports of ***unintended acceleration*** led to investigation

NHTSA estimated **89 deaths** in the prior decade (by 2010)

$3+ billion in settlements and fines

Multiple recalls involving millions of vehicles

The Code Problems:

- [Error cannot fit on single slide ;)]

When Code Quality Kills 2/2 (Toyota Unintended Acceleration)

Case 2: Toyota Unintended Acceleration (2000s-2010)

The Code Problems (NASA Investigation (2010-2011)):

- Found **7,134** MISRA-C violations (checking only 35 of 127 rules)
- **347** violations of subset of MISRA-C rules they checked
- **243** violations of NASA's own "Power of 10" safety rules
- Concluded: no definitive electronic cause found, but system was highly complex

- Civil Litigation Expert Analysis (2013): Michael Barr's team found in 2005 Camry code:
  - **81,514** MISRA-C violations (checking full 2004 ruleset)
  - **11,000** global variables
  - **67** functions rated "untestable" (cyclomatic complexity >50)
  - Stack overflow risks (**94%** usage, not 41% as Toyota claimed)
  - Critical variables not mirrored for safety
  - No watchdog protection on critical task
  - Recursion in safety-critical code (MISRA violation)
  - Toyota's Standards: Used only 11 of 127 MISRA-C rules; violated 5 of those 11

When Code Quality Kills: Resume

Both cases show:

"It compiles" ≠ "It's safe"

Standards exist for reasons - written in blood

Testing alone is insufficient - code must be maintainable and verifiable

Overconfidence kills - both companies dismissed early warnings

These weren't syntax errors. These were design and process failures that standards like MISRA specifically prevent. Let's learn what those standards are and why they matter."

When Code Quality Kills: Resume

From SOLID to safety-critical coding:

- SOLID helps with maintainability, but embedded systems need more…
- Clean code matters differently in embedded: safety, certification, resource constraints

Trade-offs: why *software engineering* vs programming or computer science

Performance vs safety, readability vs efficiency

MISRA	Code Smells	Readability
Safety	Design	Communication
Correctness	Maintainability	Understanding
Standards	Clean Architecture	Clarity
↓	↓	↓
Prevents Crashes	Reduces Tech Debt	Enables Maintenance

MISRA Compliance on C++:

“While popular, no programming language can guarantee that the final executable code will behave exactly as the developer intended.

The language has several drawbacks which are discussed in the following sub-sections.”

MISRA Compliance on C++: Essentials

Essential MISRA Reading in a C++ context:

- Chapter 2: "Disadvantages of C++" - discusses why standards are needed
- Chapter 4: Type conversions and casts
- Chapter 6: Basic concepts (particularly control flow)
- Chapter 8: Declarations and definitions
- Chapter 15: Resource management (RAII)

MISRA Compliance on C++: Essentials

Chapter 2: "Disadvantages of C++" - discusses why standards are needed

Undefined Behavior: C++ has many cases where behavior is not specified by the standard (buffer overflows, dereferencing null pointers, signed integer overflow)

Implementation-Defined Behavior: Compiler-specific behaviors that can vary

Unspecified Behavior: Standard allows multiple valid outcomes

Complexity: C++ is a large, complex language with many features that can interact in unexpected ways

Backward Compatibility: Legacy features retained for compatibility can be dangerous in safety-critical contexts

Lack of Built-in Safety: No automatic bounds checking, memory management, or initialization

Standards like MISRA restrict the language to a safer, more predictable subset by eliminating undefined behaviors and dangerous constructs.

MISRA Compliance on C++: Essentials

Chapter 4: Type conversions and casts

Implicit Conversions Are Dangerous: Can lose data silently (int to unsigned, float to int, pointer conversions)

Signed/Unsigned Mixing: Major source of bugs in embedded systems

Narrowing Conversions: Losing precision without warning

Pointer Casts: Violate type safety and can cause alignment issues

MISRA Approach:

- Prefer explicit casts over implicit conversions
- Use static_cast for safe conversions
- Avoid reinterpret_cast and C-style casts
- Never cast away const

MISRA Compliance on C++: Essentials

Chapter 6: Basic concepts (particularly control flow)

Goto Considered Harmful: Makes code difficult to analyze and verify

Single Entry/Exit: Functions should have clear, predictable flow

Loop Bounds: All loops should have deterministic, analyzable bounds

Switch Statements: Must have default cases, no fall-through

Cyclomatic Complexity: Keep functions simple and testable

MISRA Compliance on C++: Essentials

Chapter 8: Declarations and definitions

Initialization Is Mandatory: All variables must be initialized before use

Scope Minimization: Declare variables in smallest possible scope

One Declaration Per Line: Improves readability and reduces errors

Consistent Naming: Clear, meaningful names

Avoid Shadowing: Don't reuse names in nested scopes

MISRA Compliance on C++: Essentials

Chapter 15: Resource management (RAII)

RAII is Fundamental: Resources tied to object lifetime

No Naked new/delete: Use smart pointers

Deterministic Cleanup: Destructors must not fail

Exception Safety: Resources must be released even when exceptions occur

Manager Classes: Scoped, unique, and general managers for resource handling

MISRA Compliance on C++: Violation Categories

Mandatory

Must be followed - No exceptions allowed

Cannot be deviated from under any circumstances

Typically covers critical safety issues or undefined behavior

Violation = Immediate failure in certification

MISRA Compliance on C++: Violation Categories

Required (Most Common)

Should be followed - Deviations require formal justification

Can be deviated from, but you must:

1. Document why you're deviating
2. Provide technical justification
3. Get approval from project safety authority
4. Prove the deviation is safe

Covers important safety/quality issues

MISRA Compliance on C++: Violation Categories

Advisory

Recommended - Should follow when practical

Deviations don't require formal documentation

Best practices that improve code quality

May have performance and/or practical trade-offs

MISRA Compliance on C++: Examples

Disclaimer: the code examples show what pitfalls to avoid but sometimes also introduce other code smells (that is often done for simplicity reasons).

We will check our final example using the MISRA C++ 2023 rules and cppcheck!

Example MISRA C++ Rule 8.4.1: A variable shall be initialized before it is read

```cpp
// VIOLATION - Uninitialized variable
void badFunction() {
    int temperature;

    if (sensorReady()) {
        temperature = readSensor();
    }

    // BUG: temperature might be uninitialized here!
    processTemperature(temperature);
}

// COMPLIANT - Always initialized
void goodFunction() {
    int temperature = 0;  // Initialize with sensible default

    if (sensorReady()) {
        temperature = readSensor();
    }

    processTemperature(temperature);
}
```

Example MISRA C++ Rule 8.4.1: A variable shall be initialized before it is read

```
13 // COMPLIANT - Always initialized
14 void goodFunction() {
15     int temperature = 0;  // Initialize with sensible default
16
17     if (sensorReady()) {
18         temperature = readSensor();
19     }
20
21     processTemperature(temperature);
22 }
23
24 // BETTER - Initialize at declaration
25 void betterFunction() {
26     if (!sensorReady()) {
27         return;  // Early exit if sensor not ready
28     }
29
30     int temperature = readSensor();  // Initialized immediately
31     processTemperature(temperature);
32 }
```

Example MISRA C++ Rule 8.4.1: A variable shall be initialized before it is read

```
1 // Therac-25 had issues like this:
2 uint8_t treatmentMode;  // Uninitialized!
3
4 if (operatorChangedMode) {
5     treatmentMode = newMode;
6 }
7
8 // If operator didn't change mode, treatmentMode contains garbage
9 applyRadiation(treatmentMode);  // DANGEROUS!
```

Example MISRA C++ Rule 7.1.1:
A variable which is not modified shall be const qualified

```cpp
// VIOLATION - Not using const
class TemperatureSensor {
    float threshold;

public:
    TemperatureSensor(float t) : threshold(t) {}

    // This function doesn't modify the object but isn't marked const!
    bool isOverheating(float temp) {
        return temp > threshold;
    }
};

void checkSensors(TemperatureSensor* sensor) {  // Non-const pointer
    float reading = 75.5f;

    // Can accidentally modify sensor here!
    // sensor->threshold = 100.0f;  // Oops!

    if (sensor->isOverheating(reading)) {
        alarm();
    }
}
```

Example MISRA C++ Rule 7.1.1:
A variable which is not modified shall be const qualified

```
25 // COMPLIANT - Proper const usage
26 class TemperatureSensor {
27     const float threshold;  // const member - can't be changed after construction
28
29 public:
30     TemperatureSensor(float t) : threshold(t) {}
31
32     // const member function - documents that object isn't modified
33     bool isOverheating(const float temp) const {
34         return temp > threshold;
35     }
36
37     float getThreshold() const {  // Getter is const
38         return threshold;
39     }
40 };
41
42 void checkSensors(const TemperatureSensor* sensor) {  // const pointer
43     const float reading = 75.5f;  // const variable
44
45     // Compiler prevents accidental modification:
46     // sensor->threshold = 100.0f;  // ERROR: const object!
47
48     if (sensor->isOverheating(reading)) {
49         alarm();
50     }
51 }
```

Example MISRA C++ Rule 7.1.1:
A variable which is not modified shall be const qualified

```cpp
// PRACTICAL EXAMPLE
class PIDController {
    const float kP, kI, kD;  // Constants don't change
    float integral, prevError;  // State variables that do change

public:
    PIDController(const float p, const float i, const float d)
        : kP(p), kI(i), kD(d), integral(0.0f), prevError(0.0f) {}

    float compute(const float setpoint, const float measured) {
        const float error = setpoint - measured;  // Local const
        integral += error;
        const float derivative = error - prevError;

        prevError = error;  // Modify state

        return kP * error + kI * integral + kD * derivative;
    }

    // const function - just reads state
    float getIntegral() const {
        return integral;
    }
};
```

Example MISRA C++ Rule 15.0.1:
A class that appears to manage resources should use RAII

```cpp
// VIOLATION - Manual resource management
class DataLogger {
    FILE* logFile;

public:
    DataLogger(const char* filename) {
        logFile = fopen(filename, "w");
    }

    void log(const char* message) {
        if (logFile) {
            fprintf(logFile, "%s\n", message);
        }
    }

    // PROBLEM: User must remember to call close()!
    void close() {
        if (logFile) {
            fclose(logFile);
            logFile = nullptr;
        }
    }
};

void buggyCode() {
    DataLogger logger("system.log");
    logger.log("Starting...");

    if (errorCondition()) {
        return;  // BUG: File never closed! Memory leak!
    }

    logger.log("Done");
    logger.close();  // Only closed if we get here
}
```

Example MISRA C++ Rule 15.0.1:
A class that appears to manage resources should use RAII

```cpp
// COMPLIANT - RAII pattern
class DataLogger {
    FILE* logFile;

public:
    // Constructor acquires resource
    DataLogger(const char* filename) : logFile(nullptr) {
        logFile = fopen(filename, "w");
        if (!logFile) {
            throw std::runtime_error("Failed to open log file");
        }
    }

    // Destructor releases resource automatically
    ~DataLogger() {
        if (logFile) {
            fclose(logFile);
            logFile = nullptr;
        }
    }

    // Prevent copying (would cause double-free)
    DataLogger(const DataLogger&) = delete;
    DataLogger& operator=(const DataLogger&) = delete;

    void log(const char* message) {
        if (logFile) {
            fprintf(logFile, "%s\n", message);
        }
    }
};
```

```cpp
void correctCode() {
    DataLogger logger("system.log");
    logger.log("Starting...");

    if (errorCondition()) {
        return;  // File automatically closed by destructor!
    }

    logger.log("Done");
    // File automatically closed here too
}

// EVEN BETTER - Use standard library RAII
void bestCode() {
    std::ofstream logger("system.log");
    logger << "Starting...\n";

    if (errorCondition()) {
        return;  // Automatically closed!
    }

    logger << "Done\n";
    // Automatically closed
}
```

Example MISRA C++ Rule Rule
18.0.2: Do not use new or delete; use smart pointers instead

```cpp
// VIOLATION - Raw new/delete
void processData() {
    int* buffer = new int[1000];  // Manual allocation

    readData(buffer, 1000);

    if (errorCondition()) {
        return;  // BUG: Memory leak!
    }

    processBuffer(buffer, 1000);

    delete[] buffer;  // Only freed if we get here
}

class Sensor {
    int* readings;
    size_t count;

public:
    Sensor(size_t n) : count(n) {
        readings = new int[n];  // Manual allocation
    }

    // PROBLEM: If we forget destructor, memory leaks!
    // PROBLEM: If we copy this object, double-free!
    ~Sensor() {
        delete[] readings;
    }
};
```

Example MISRA C++ Rule Rule
18.0.2: Do not use new or delete; use smart pointers instead

```cpp
// COMPLIANT - Use std::unique_ptr for single ownership
void processData() {
    std::unique_ptr<int[]> buffer(new int[1000]);
    // Or better: auto buffer = std::make_unique<int[]>(1000);

    readData(buffer.get(), 1000);

    if (errorCondition()) {
        return;  // Memory automatically freed!
    }

    processBuffer(buffer.get(), 1000);

    // Memory automatically freed here too
}

// BETTER - Use std::vector (preferred for arrays)
void processDataBetter() {
    std::vector<int> buffer(1000);

    readData(buffer.data(), buffer.size());

    if (errorCondition()) {
        return;  // Automatically cleaned up!
    }

    processBuffer(buffer.data(), buffer.size());
}
```

Example MISRA C++ Rule Rule
18.0.2: Do not use new or delete; use smart pointers instead

```cpp
// COMPLIANT - Smart pointer in class
class Sensor {
    std::unique_ptr<int[]> readings;
    size_t count;

public:
    Sensor(size_t n) : readings(std::make_unique<int[]>(n)), count(n) {}

    // No need for explicit destructor - unique_ptr handles it!
    // Copying is automatically prevented by unique_ptr

    int getReading(size_t index) const {
        if (index < count) {
            return readings[index];
        }
        return 0;
    }
};
```

Example MISRA C++ Rule Rule
18.0.2: Do not use new or delete; use smart pointers instead

```cpp
// BEST - Use std::vector in class
class SensorBetter {
    std::vector<int> readings;

public:
    explicit SensorBetter(size_t n) : readings(n, 0) {}

    // No destructor needed!
    // Copyable and moveable automatically if needed

    int getReading(size_t index) const {
        if (index < readings.size()) {
            return readings[index];
        }
        return 0;
    }

    size_t size() const { return readings.size(); }
};

```

Example MISRA C++ Rule Rule
18.0.2: Do not use new or delete; use smart pointers instead

```cpp
// ALTERNATIVE std::shared_ptr for shared ownership
class DataCache {
    std::shared_ptr<std::vector<int>> data;

public:
    DataCache() : data(std::make_shared<std::vector<int>>(1000)) {}

    // Multiple objects can share this data safely
    std::shared_ptr<std::vector<int>> getData() const {
        return data;  // Reference count increases
    }
};  // Reference count decreases, memory freed when last owner destroyed
```

MISRA Compliance on C++:

Key Rules:

- Rule 8.4.1 (uninitialized variables)
- Rule 8.4.2 (no goto)
- Rule 7.1.1 (const correctness)
- Rule 15.0.1 (RAII)
- Rule 18.0.2 (no raw new/delete)

MISRA Violations Hunt

```cpp
// How many MISRA violations?
#define MAX_SENSORS 10

class SensorArray {
    int* data;
    int size;

public:
    SensorArray(int s) {
        size = s;
        data = new int[size];
    }

    int getValue(int index) {
        return data[index];
    }

    void process() {
        int result;
        for (int i = 0; i <= size; i++) {
            if (data[i] < 0)
                goto error;
            result = data[i] * 2;
        }
        return;

    error:
        result = -1;
    }
};
```

```cpp
void main() {
    SensorArray sensors(MAX_SENSORS);
    sensors.process();
}
```

https://gist.github.com/jakorten/e9ffce1eb57fda54c74704e4bac91cd6

MISRA Compliance on C++:

```
Scanning directory: Example
✓ Found 1 C++ files

✓ Loaded 146 MISRA C++ rules
[1/1] Checking our_wonderful_code.cpp... 13 violations

Scan Complete!
  Total violations: 13
  Files with issues: 1/1

✓ Report saved to report_example.md

Summary by Priority:
  🟡 Required: 8
  🔵 Advisory: 5
```

Tip: in the Repo is also “MoreCode_Anonymized” you could also try to check for violations.
Note: it is very, very likely to find violations in ALL our (more serious) code, that is not the point!!!

More on (static) code checking:

```
# MISRA Checking Quick Reference

## Free Tools (Use in practice)
- cppcheck --enable=all mycode.cpp
- clang-tidy mycode.cpp -- -std=c++17
- gcc/clang with: -Wall -Wextra -Wpedantic -Werror

## What They Check (MISRA-like)
✓ Uninitialized variables
✓ Memory leaks
✓ Null pointer dereferences
✓ Buffer overflows
✓ Dead code
✓ Complexity metrics

## What They Miss (Need commercial tools)
✗ Full MISRA rule compliance
✗ Certification-grade reports
✗ Deep flow analysis
```

How do we check our code…

CPPCheck
cpplint to check Google C++ Style Guide compliance

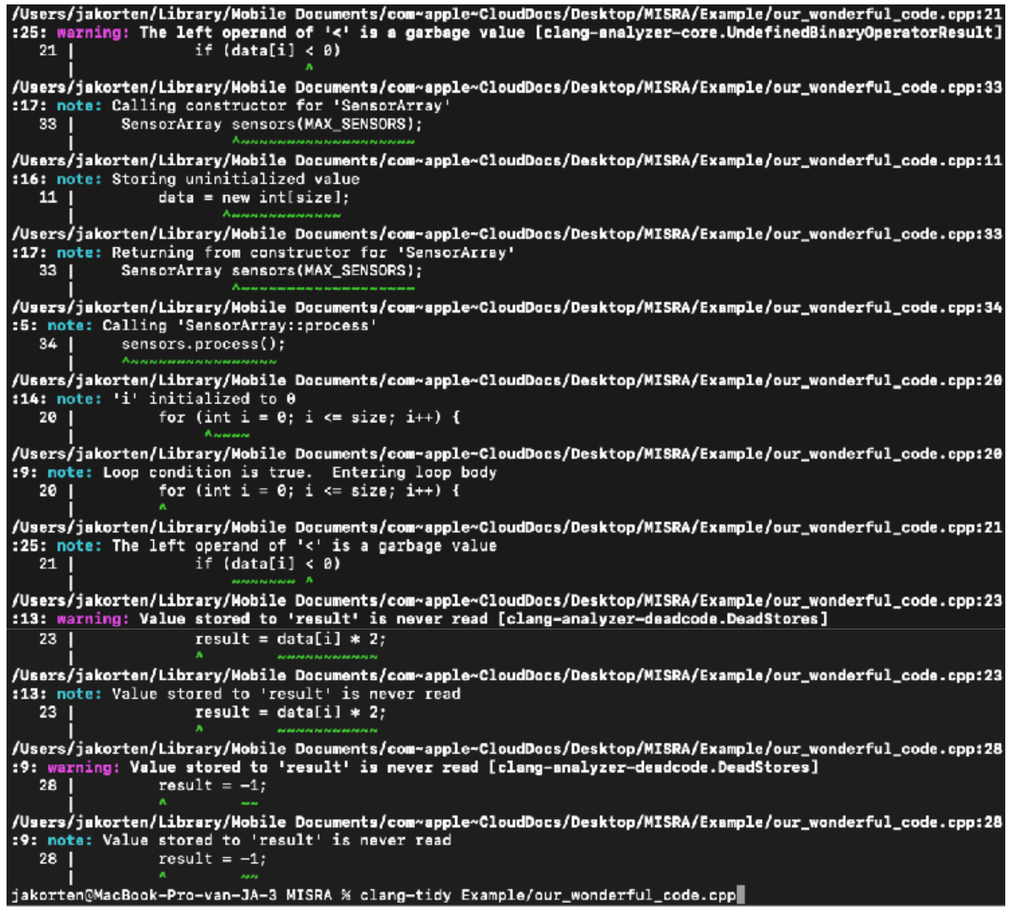
and other static code analyzers

Remember: they are just one of many in our bag of tools.

More on this next week!

Clang-tidy

```
clang-tidy Example/our_wonderful_code.cpp
```



Readibility

Clean code is simple and direct.

Clean code reads like well-written prose.

Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.”

- Grady Booch author of Object Oriented Analysis and Design with Applications (Booch, Jacobson and Rumbaugh were behind UML)

Kent Beck: Order matters!

Code is written once but read hundreds of times. Write for the hundreds of times.” - Kent Beck

- First: Make it WORK (on target hardware; correctness)
- Second: Make it RIGHT (safe, maintainable)
- Make it FIT (within memory constraints)
- Make it FAST (meet real-time deadlines)

Kent Beck: Habits

"I'm not a great programmer; I'm just a good programmer with great habits."

The habits:

✅ Write tests

✅ Refactor continuously

✅ Name things carefully

✅ Keep it simple

✅ Make it readable

Trade-offs: why *software engineering* vs programming or computer science

Exercise 1: "Spot the Issues"

- Give students the bad code, ask them to find:
- 5 MISRA violations
- 5 code smells
- Which ones overlap?

Exercise 2: "Prioritize the Fixes"

- Which would you fix first? Why?
- Safety vs maintainability trade-offs

Exercise 3: "Refactor Step-by-Step"

- Start with bad code
- Fix MISRA violations
- Remove code smells
- Compare: did fixing one help the other?

Trade-offs: why *software engineering* vs programming or computer science

Combat "vibe coding": Emphasize that these aren't arbitrary rules - each has a failure case. Show real bugs that resulted from violations.

Make it embedded-relevant: Use examples from Arduino/microcontroller context they know - ISRs, hardware registers, timing-critical code.

Code style: Naming Conventions

C++ Code Style Best Practices

Classes: PascalCase (e.g., SensorController, MotorDriver)

Functions/methods: camelCase (e.g., readSensor(), calculateSpeed())

Variables: camelCase (e.g., sensorValue, targetPosition)

Constants: UPPER_SNAKE_CASE (e.g., MAX_SPEED, DEFAULT_TIMEOUT)

Member variables: prefix with m_ (e.g., m_currentState, m_sensorData)

Code style: Code Quality

Always use const for values that don't change

Prefer references (&) over pointers when ownership isn't transferred

Initialize variables at declaration

Avoid magic numbers - use named constants

Write self-documenting code with clear names

(And of course the rest that we already discussed)

Code style: Comments (later workshop)

Explain why, not what (code shows what)

Document complex algorithms and hardware interactions

Keep comments up-to-date with code changes

Code smells

```
python3 code_smell_scanner.py  MoreCode_Anonymized
```

```
Scanning 6 C++ files...


================================================================================
CODE SMELL ANALYSIS REPORT
================================================================================

Summary:
  Total code smells: 99
  Files with issues: 6

By Severity:
  🔴 Critical  :   1 (  1.0%)
  🟡 Major     :   9 (  9.1%)
  🔵 Minor     :  89 ( 89.9%)

By Type:
  • Magic Number            :  89 ( 89.9%) ████████████████████
  • Dead Code               :   5 (  5.1%) ▌
  • Duplicate Code          :   4 (  4.0%)
  • God Class               :   1 (  1.0%)


Top Recommendations:
  🔴 CRITICAL: Address Large/God classes immediately
  🟡 MAJOR: Eliminate duplicate code through abstraction
  🔵 MINOR: Replace magic numbers with named constants
```

Code style

Of course companies might have their own style guidelines with do’s and don’ts!

Code style at file level

Kolpackov's "Canonical Project Structure" (ISO C++ Paper P1204R0, 2018) argues that split layouts make "navigating between corresponding headers and sources cumbersome" and that "in practice, private headers are placed into include/, defeating the purpose”

Code style: mixing (natural) languages

Sutter & Alexandrescu (2005): Coding standards promote consistency and international collaboration Micro-os-plus

Google C++ Style Guide: Emphasizes clear, unambiguous naming for global readability Google

JSF AV C++ (Safety-critical standard): Consistent, descriptive English identifiers required Bjarne Stroustrup

Bottom Line: For embedded systems that may be maintained internationally or in professional contexts, English identifiers are mandatory. Dutch variable names create barriers to code review, collaboration, and professional development.

Recommended Refactoring:

```cpp
// Current (Dutch)          → Recommended (English)
algemeen/                   → common/ or general/
dataPakket.h                → DataPacket.h
algdef.h                    → CommonDef.h or GeneralDef.h
```

Project Structure Recommendation:

```
rgbData/
  RGBData.h
  RGBData.cpp
common/                              // English equivalent
  DataPacket.h
  DataPacket.cpp
  CommonDef.h
  CommonDef.cpp
```

Take away message

Starting today, write code that is:
✅ Safe (MISRA)
✅ Maintainable (No smells)
✅ Readable (Clear intent)

Don't just make it work. Make it RIGHT.

https://github.com/AEAEmbedded/ESE_PROG/tree/main/Workshops

Great Resources

C++ Core Guidelines (Stroustrup et al., ISO C++ Standards Committee)

Sutter & Alexandrescu (2005): C++ Coding Standards: 101 Rules, Guidelines, and Best Practices

JSF AV C++ (Lockheed Martin, 2005): Safety-critical embedded systems standard (221 rules)

MISRA C++:2023: Automotive/medical device coding guidelines for critical systems

Google C++ Style Guide: Industry best practices for large-scale development

Kolpackov's "Canonical Project Structure" (ISO C++ Paper P1204R0, 2018)

Sutter & Alexandrescu (2005): Coding standards promote consistency and international collaboration

“

That’s all…

Any questions?