Balancing Coding Rules with Embedded Systems Constrains

Stefan Fuhrmann, Efficientware GmbH

How Following ~~Orders~~Coding Rules Can Ruin Your Day

Stefan Fuhrmann, Efficientware GmbH

Rules are Tools; They Aren't Your Product

Stefan Fuhrmann, Efficientware GmbH

A Plea Against Coding Rule Totalitarianism

Stefan Fuhrmann, Efficientware GmbH

Balancing Coding Rules with Embedded Systems Constrains

Stefan Fuhrmann, Efficientware GmbH

My Background

- 2017 @ Bosch:
  SW optimization team, tech lead
- L4 full autonomous driving cars
  (10x squeeze to 40 ARM cores)

- Branched out to other projects
- Lots of firefighting

- 2024: Co-founder Efficientware

Non-Controversial Statement

"Runtime performance and efficient resource usage are *quality* attributes.

They impose non-functional requirements onto your system that *need to be fulfilled* like any other requirement."

(ISO 26262 et al.)

Non-Controversial Statement

"Runtime performance and efficient resource usage are *quality* attributes.

They impose non-functional requirements onto your system that *need to be fulfilled* like any other requirement."

(ISO 26262 et al.)

- Efficiency allows you to do more and better, at a lower price.
- Think smartphones:
  Need bigger model because of bloated apps and terrible battery life?

Foto: Bundesregierung

The Problem

- 1968: "Software Crisis"
- Design and language pitfalls

- Best practices, guidelines, coding rules, traditions and cultures

The Problem

- 1968: "Software Crisis"
- Design and language pitfalls

- Best practices, guidelines, coding rules, traditions and cultures

- Belief: sticking to the rules almost guarantees a successful product
- Safety: fewer automatisms and "zero cost X" in certified tool chain

- The rules may incur runtime and memory footprint penalties
- Making people understand and accept the conflict is hard

- Our experience: Teaching the underlying issues helps

Over-Confidence – How Rules can Fail you

- Goal: No part of your system is in an undefined state.
- Rule: Initialize all variables (=state), prefer {} syntax.

Over-Confidence – How Rules can Fail you

- Goal: No part of your system is in an undefined state.
- Rule: Initialize all variables (=state), prefer {} syntax.

```
// Your system must pick one of several
// variants; there are no defaults.
enum Selection {
  variantA = 1,
  variantB = 2
};
Selection selection{};
```

Over-Confidence – How Rules can Fail you

- Goal: No part of your system is in an undefined state.
- Rule: Initialize all variables (=state), prefer {} syntax.

```
// Your system must pick one of several
// variants; there are no defaults.
enum Selection {
  variantA = 1,
  variantB = 2
};
Selection selection{};

// Pray that someone assigns a different
// value before you run this:
switch(selection) {
  case variantA: foo(); break;
  case variantB: bar(); break;
  // You always add a default, right?!
  default: hcf(); // halt & catch fire
}
```

Over-Confidence – How Rules can Fail you

- Goal: No part of your system is in an undefined state.
- Rule: Initialize all variables (=state), prefer {} syntax.
- The value of `selection` is well-defined from a language POV but semantically undefined.
- Zero-init via {} not a complete fix!
- No constructors in `enum class`
- Redesign is required

```
// Your system must pick one of several
// variants; there are no defaults.
enum Selection {
  variantA = 1,
  variantB = 2
};
Selection selection{};

// Pray that someone assigns a different
// value before you run this:
switch(selection) {
  case variantA: foo(); break;
  case variantB: bar(); break;
  // You always add a default, right?!
  default: hcf(); // halt & catch fire
}
```

Recurring Problem Areas

- Initialization
- Inlining
- Templates

Cost of Initialization – Curly Braces

```
template<typename T, size_t N>
struct Container {
  T data[N];
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// What does the initialization look like?
Container<Data, 100> c{};

Container<Data, 100> foo() {
  return {};
}
```

Cost of Initialization – Curly Braces

```
template<typename T, size_t N>
struct Container {
  T data[N];
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// What does the initialization look like?
Container<Data, 100> c{};

Container<Data, 100> foo() {
  return {};
}
```

Compiler must pick a trade-off:

1. Pre-bake full pattern + memcpy
2. Constructor function for Data + generic array construction helper
3. Inline create 16-byte pattern and store it 100 times in a loop
4. Inline create 16-byte pattern and write it in 100 store instructions

Each one works well for either easy or complicated patterns, small or large repetition counts, scarce ROM or runtime.

Cost of Initialization – Curly Braces

1. Pre-bake full pattern + memcpy
   GCC 14: (if static, no memcpy) https://godbolt.org/z/v3GzMfhjz
   GCC 8: (shared pattern) https://godbolt.org/z/c69vY9eTn
   TI CL430: (1 pattern per instance) https://godbolt.org/z/P7bWz3xhG
2. Constructor function for Data + generic array construction helper
   No example found. Something with -Os? -Og?
3. Inline create 16-byte pattern and store it 100 times in a loop
   GCC 14: https://godbolt.org/z/v3GzMfhjz
4. Inline create 16-byte pattern and write it in 100 store instructions
   Clang 19: https://godbolt.org/z/996Er37q9

Double Initialization

```
template<typename T, size_t N>
struct Container {
  T data[N];
  size_t best; // complication?!
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// What does the initialization look like?
Container<Data, 100> c{};

Container<Data, 100> foo() {
  return {};
}
```

Double Initialization

```
template<typename T, size_t N>
struct Container {
  T data[N];
  size_t best; // complication?!
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// What does the initialization look like?
Container<Data, 100> c{};

Container<Data, 100> foo() {
  return {};
}
```

```
foo():
  stp x29, x30, [sp, -16]!
  mov x0, x8
  mov w1, 0
  mov x29, sp
  mov x2, 1608
  bl memset
  mov x8, x0
  adrp x0, .LC0
  add x1, x8, 1600
  ldr q31, [x0, #:lo12:.LC0]
  mov x0, x8
  str q31, [x0], 16
.L2:
  str q31, [x0], 16
  cmp x1, x0
  bne .L2
  ldp x29, x30, [sp], 16
  ret
.LC0:
  .word 1068925085
  .word 1158833091
  .word 1191552333
  .word -1064882825
c:
  .word 1068925085
  .word 1158833091
  ...
```

init loop for data

pattern for Data

static c fully initialized

Double Initialization

```
template<typename T, size_t N>
struct Container {
  T data[N];
  size_t best; // complication?!
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// What does the initialization look like?
Container<Data, 100> c{};

Container<Data, 100> foo() {
  return {};
}
```

- Curly braces without parameter are zero-initializers (`c{}`)
- Compiler fails to eliminate the initial zeroing for complicated cases.
- GCC 14: (static o.k., local bad) https://godbolt.org/z/hd7xv15Wb

- Additional `memset` before constructing actual content
- Even worse: repeated `memset` for nested structures

Triple Initialization

```
template<typename T, size_t N>
struct Container {
  T data[N];
  size_t best;
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

void foo() {
  // Double initialization + overwrite
  Container<Data, 100> buffer{};
  readData(buffer);
}
```

Triple Initialization

```
template<typename T, size_t N>
struct Container {
  T data[N];
  size_t best;
};

struct Data {
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

void foo() {
  // Double initialization + overwrite
  Container<Data, 100> buffer{};
  readData(buffer);
}
```

- Nothing surprising but providing buffers is a typical pattern.
- Be mindful about the minimum initialization requirements of your data types
  (e.g. `vector`-like vs `array`-like)

Reduce Initialization Costs

```
template<typename T, size_t N>
struct Container {
  Container(): best{0} {}  // add that
  T data[N];
  size_t best;
};

struct Data {
  Data();  // no large pre-baked patterns
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// This is no longer zero-initialization!
Container<Data, 100> c{};
// No data being initialized; only c.best
Container<int, 100> u{};
```

- Syntax hack: provide a default constructor
- `c{}` is now a constructor call
- Gets rid of the `memset`

Reduce Initialization Costs

```
template<typename T, size_t N>
struct Container {
  Container(): best{0} {}  // add that
  T data[N];
  size_t best;
};

struct Data {
  Data();  // no large pre-baked patterns
  float a{1.4258F};
  float b{2342.235F};
  float c{34213.3F};
  float d{-4.2243F};
};

// This is no longer zero-initialization!
Container<Data, 100> c{};
// No data being initialized; only c.best
Container<int, 100> u{};
```

- Syntax hack: provide a default constructor
- `c{}` is now a constructor call
- Gets rid of the `memset`

- A non-inlined constructor for Data fixes the ROM usage problem but makes the code slower

- Types are no longer trivial
- Sub-structures initialized only if required by them (`int` does not) or the constructor (we don't).

Honorable Mention: RVO

```
// RVO: result not memset + memcpy'ed,
//      return value is memset directly.
std::array<char, 10000> foo() {
  std::array<char, 10000> result{};
  return result;
}

// TI CL430 21.6.1
foo():
  ...
  MOV.W #$P$T1$2+0,r13
  MOV.W #10000,r14
  CALL #memcpy ; 10k pre-baked zeros
  ...
  MOV.W r10,r12
  MOV.W #10000,r14
  CALL #memcpy ; did not RVO
  ...
  RET
```

- URVO mandatory since C++17
- Some embedded compilers support URVO but not NRVO, some neither
- Stack size increases
- At odds with modern C++ trend towards value semantics
- Passing outputs by reference frowned upon by rules.

Inlining

- Myth: "the `inline` keyword just a hint; the compiler knows best anyway"
- Reality: correct use will likely make or break your product

Inlining – Compiler Philosophies

Do as the user tells you

- Don't inline if not marked as such
- Inline others, even large functions (depending on internal metrics)
- Often found in embedded compilers

Inline is automatic

- Keyword relevant for ODR only (functions in header-only libs etc.)
- Inline anything the compiler sees fit
- May not inline larger functions
- Deep "forwarding" call hierarchies may not be inlined

Inlining – Suggestions

- Don't rely on link-time-optimization (LTO) to inline across source files
  -> may not be available in your embedded tool chain

- Explicitly mark exactly those functions as `inline` that shall be inlined
- Avoid inlining large functions as results vary greatly by compiler
- Caveat: function definitions inside a C++ class definition are implicitly inline

- Pro-tip: Use profilers to see if you missed something

Inlining – Complication

- Example: TI C6x compiler inlines anything that's called only once
  -> great results for static functions in the same file

- C/C++: one compilation unit at a time (1 source file + includes)
  -> "only once" is decided for each source file separately

- Large functions in shared headers can cause serious binary bloat (templates!)

C++ Templates – Code Bloat Problem

- Great for data containers: generic and type-safe at the same time
- Problem: function code often binary identical between instances but language *requires* different function addresses for different instances
- Result: Bloated and slower binary

C++ Templates – Code Bloat Problem

- Great for data containers: generic and type-safe at the same time
- Problem: function code often binary identical between instances but language *requires* different function addresses for different instances
- Result: Bloated and slower binary

- Example: compare generated code `vector<int*>::resize()` vs. `vector<size_t>::resize()`

- Compare using ARM64 GCC 14.2.1: https://gcc.godbolt.org/z/PbKTPeK1j

C++ Templates – Code Bloat Problem

- Great for data containers: generic and type-safe at the same time
- Problem: function code often binary identical between instances but language *requires* different function addresses for different instances
- Result: Bloated and slower binary

- Solution 1: Avoid function pointers and use a linker with `--icf=safe`
- Solution 2: De-templatize

- May conflict with DRY rule.

- Example: compare generated code `vector<int*>::resize()` vs. `vector<size_t>::resize()`

- Compare using ARM64 GCC 14.2.1: https://gcc.godbolt.org/z/axhd13ETP

C++ Templates – Accidental Code Bloat

```cpp
#include <array>

template<typename C>
typename C::value_type best(const C& c) {
  typename C::value_type result{};
  for (auto&& e : c) {
    if (e > result) {
      result = e;
    }
  }
  return result;
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/65vjTM49P

C++ Templates – Accidental Code Bloat

```cpp
#include <array>

template<typename C>
typename C::value_type best(const C& c) {
  typename C::value_type result{};
  for (auto&& e : c) {
    if (e > result) {
      result = e;
    }
  }
  return result;
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/65vjTM49P

```
std::array<int, 100ul>::value_type
best<std::array<int,
  100ul> >(std::array<int, 100ul> const&):
    movi v31.4s, 0
    add x1, x0, 400
  .L2:
    ldr q30, [x0], 16
    smax v31.4s, v31.4s, v30.4s
    cmp x1, x0
    bne .L2
    smaxv s31, v31.4s
    fmov w0, s31
    ret

std::array<int, 200ul>::value_type
best<std::array<int,
  200ul> >(std::array<int, 200ul> const&):
    movi v31.4s, 0
    add x1, x0, 800
.L6:
    ldr q30, [x0], 16
    smax v31.4s, v31.4s, v30.4s
    cmp x1, x0
    bne .L6
    smaxv s31, v31.4s
    fmov w0, s31
    ret
```

C++ Templates – Accidental Code Bloat

```
#include <array>

template<typename C>
typename C::value_type best(const C& c) {
  typename C::value_type result{};
  for (auto&& e : c) {
    if (e > result) {
      result = e;
    }
  }
  return result;
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/65vjTM49P

- Particularly nasty: capacities and options as template parameters
- Variant: strongly typed ints, floats

C++ Templates – Accidental Code Bloat

```
#include <array>

template<typename C>
typename C::value_type best(const C& c) {
  typename C::value_type result{};
  for (auto&& e : c) {
    if (e > result) {
      result = e;
    }
  }
  return result;
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/65vjTM49P

- Particularly nasty: capacities and options as template parameters
- Variant: strongly typed ints, floats
- Solution: Move logic to common functions with fewer / no template parameters
- Advise: Use MAP file to find relevant template instantiations
- Use -O0 to get a full picture

C++ Templates – Accidental Code Bloat

```cpp
#include <array>

// actual logic goes here
template<typename I>
auto best(I first, I last) -> typename
  std::iterator_traits<I>::value_type;


template<typename C>
typename C::value_type best(const C& c) {
  // only determines the range,
  // i.e. evaluates the capacity parameter
  return best(c.begin(), c.end());
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/GscWPd3Yb

C++ Templates – Accidental Code Bloat

```cpp
#include <array>

// actual logic goes here
template<typename I>
auto best(I first, I last) -> typename
  std::iterator_traits<I>::value_type;


template<typename C>
typename C::value_type best(const C& c) {
  // only determines the range,
  // i.e. evaluates the capacity parameter
  return best(c.begin(), c.end());
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/GscWPd3Yb

```
std::array<int, 100ul>::value_type
best<std::array<int, 100ul>>(std::array<int, 100ul>
const&):
    add x1, x0, #400
    b std::iterator_traits<int const*>::value_type
          best<int const*>(int const*, int const*)

std::array<int, 200ul>::value_type
best<std::array<int, 200ul>>(std::array<int, 200ul>
const&):
    add x1, x0, #800
    b std::iterator_traits<int const*>::value_type
          best<int const*>(int const*, int const*)
```

C++ Templates – Accidental Code Bloat

```cpp
#include <array>

// actual logic goes here
template<typename I>
auto best(I first, I last) -> typename
  std::iterator_traits<I>::value_type;


template<typename C>
typename C::value_type best(const C& c) {
  // only determines the range,
  // i.e. evaluates the capacity parameter
  return best(c.begin(), c.end());
}

using std::array;
template int best(const array<int, 100>&);
template int best(const array<int, 200>&);
```

https://gcc.godbolt.org/z/GscWPd3Yb

- Do not encode capacities in iterator types.
- STL *tends* to do a good job here

- Reduce number of parameter combinations
- Replace template parameters with function parameters

- Using templates needs monitoring and may add design restrictions

C++ Templates – Inlining Problem

- Template definitions traditionally kept in headers
- Problem: template function definitions visible in all including translation units
  -> compiler might decide to inline even large functions

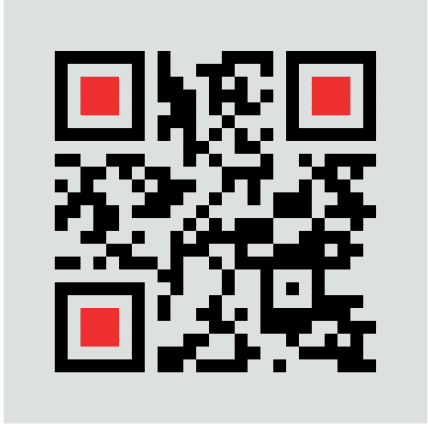
- Solution 1: De-templatize complicated inner logic
- Solution 2: Move definition to source file + explicitly instantiate

Summary

- Rules are crystalized experience – follow them
- However, idealized code may be too large or too slow for your product.

- All your coding rules must allow for ad-hoc exceptions.
- Try to tailor rules to work well with you tool chain.
- If you run out of resources, check the generated assembly for anomalies.

- Pro-tip: Do so early on in your project



https://effw.net/embo25