C++ONLINE

ANTHONY WILLIAMS

TALK:

WRITING A BASE LEVEL LIBRARY FOR SAFETY CRITICAL CODE

2025

Overview

Overview

- Introduction

Overview

- Introduction
- Standards

Overview

- Introduction
- Standards
- Tooling

Overview

- Introduction
- Standards
- Tooling
- Testing

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling
- Design Impacts

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling
- Design Impacts
- Summary

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling
- Design Impacts
- Summary

Introduction

Cars have hundreds of **Electronic Control Units** (ECUs).

Introduction

Cars have hundreds of **Electronic Control Units** (ECUs).

Many perform operations that are vital for **safe** operation of the vehicle.

Introduction

Cars have hundreds of **Electronic Control Units** (ECUs).

Many perform operations that are vital for **safe** operation of the vehicle.

Lots of the software for these is written in C++.

Introduction

Cars have hundreds of **Electronic Control Units** (ECUs).

Many perform operations that are vital for **safe** operation of the vehicle.

Lots of the software for these is written in C++.

My team's code is intended to be usable by all C++ code in the vehicle.

Introduction

Safety Critical Software

Software where the consquence for failure is injury or loss of life.

Introduction

This impacts how you develop software. You need to:

Introduction

This impacts how you develop software. You need to:

- Analyse the effects of errors

Introduction

This impacts how you develop software. You need to:

- Analyse the effects of errors
- Minimize the chance of errors

Introduction

This impacts how you develop software. You need to:

- Analyse the effects of errors
- Minimize the chance of errors
- Defend against the bad effects of errors

Introduction

This impacts how you develop software. You need to:

- Analyse the effects of errors
- Minimize the chance of errors
- Defend against the bad effects of errors
- Provide evidence of the above for your **Safety Case**

Overview

- Introduction
- **Standards**
- Tooling
- Testing
- Error Handling
- Design Impacts
- Summary

woven by TOYOTA

Standards

There are two kinds of standards that Safety Critical Software cares about:

Standards

There are two kinds of standards that Safety Critical Software cares about:

- Standards covering the **Software Development Process** as a whole

Standards

There are two kinds of standards that Safety Critical Software cares about:

- Standards covering the **Software Development Process** as a whole
- Coding Standards

Software Development Standards

The “baseline” standard is IEC 61508: “Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems”

Software Development Standards

There are also industry-specific standards:

Software Development Standards

There are also industry-specific standards:

- ISO 26262 for automotive software

Software Development Standards

There are also industry-specific standards:

- ISO 26262 for automotive software
- DO-178C for aerospace software

Software Development Standards

There are also industry-specific standards:

- ISO 26262 for automotive software
- DO-178C for aerospace software
- IEC 62304 and related standards for medical device software

Software Development Standards

There are also industry-specific standards:

- ISO 26262 for automotive software
- DO-178C for aerospace software
- IEC 62304 and related standards for medical device software
- CENELEC EN 50128 for railway software

Software Development Standards

There are also industry-specific standards:

- ISO 26262 for automotive software
- DO-178C for aerospace software
- IEC 62304 and related standards for medical device software
- CENELEC EN 50128 for railway software
- etc.

Software Development Standards

These standards cover everything about the software development **process**:

Software Development Standards

These standards cover everything about the software development **process**:

- Requirements

Software Development Standards

These standards cover everything about the software development **process**:

- Requirements
- Design

Software Development Standards

These standards cover everything about the software development **process**:

- Requirements
- Design
- Testing

woven by TOYOTA

Software Development Standards

These standards cover everything about the software development **process**:

- Requirements
- Design
- Testing

What they **don't** cover is language-specific coding standards.

Safety Integrity Levels

The standards define **Safety Integrity Levels** based on the consequences of software failure.

Safety Integrity Levels

The standards define **Safety Integrity Levels** based on the consequences of software failure.

Each piece of software is assigned an SIL based on the role it plays, and any external failure mitigation.

Safety Integrity Levels

The standards define **Safety Integrity Levels** based on the consequences of software failure.

Each piece of software is assigned an SIL based on the role it plays, and any external failure mitigation.

These then determine the specific rules to follow.

Safety Integrity Levels

ISO 26262 defines 4 **Automotive Safety Integrity Levels** (ASIL), plus **Quality Management** (QM).

woven by TOYOTA

Safety Integrity Levels

ISO 26262 defines 4 **Automotive Safety Integrity Levels** (ASIL), plus **Quality Management** (QM).

DO-178C defines 5 **Design Assurance Levels** (DAL) or **Item Development Assurance Levels** (IDAL)

Safety Integrity Levels

ISO 26262 defines 4 **Automotive Safety Integrity Levels** (ASIL), plus **Quality Management** (QM).

DO-178C defines 5 **Design Assurance Levels** (DAL) or **Item Development Assurance Levels** (IDAL)

Other standards provide similar definitions

Safety Integrity Levels

Different standards order Integrity Levels differently.

Safety Integrity Levels

Different standards order Integrity Levels differently.

ISO26262: QM is the lowest level, then ASIL-A, and ASIL-D is the highest.

Safety Integrity Levels

Different standards order Integrity Levels differently.

ISO26262: QM is the lowest level, then ASIL-A, and ASIL-D is the highest.

DO178C: DAL-E is the lowest level, DAL-A is the highest.

Safety Integrity Levels

The higher the integrity level, the more stringent the requirements on design, testing, reviews, and other verification and validation activities.

Safety Integrity Levels

The higher the integrity level, the more stringent the requirements on design, testing, reviews, and other verification and validation activities.

e.g. for ASIL-A, a “walk-through” of the design may be enough, but for ASIL-D it is not: you need to do detailed inspection and analysis.

Coding Standards

Safety-critical software coding standards specify which language constructs you can use, and how you can use the others.

Coding Standards

They may be very specific:

> *Rule M5-18-1*
> *The comma operator shall not be used.*
> — *AUTOSAR C++14 Guidelines*

Coding Standards

or more general:

> *Rule 6.0.1*
> *Block scope declarations shall not be visually ambiguous*
> — *MISRA C++: 2023*

Coding Standards

Coding Standards lag behind language standards:

Coding Standards

Coding Standards lag behind language standards:

- MISRA C++: 2008 — C++03

Coding Standards

Coding Standards lag behind language standards:

- MISRA C++: 2008 — C++03
- AUTOSAR C++14 Guidelines (2017) — C++14

woven by TOYOTA

Coding Standards

Coding Standards lag behind language standards:

- MISRA C++: 2008 — C++03
- AUTOSAR C++14 Guidelines (2017) — C++14
- MISRA C++: 2023 — C++17

Language Versions

Conformance with **process standards** such as ISO 26262 requires **coding guidelines** for used programming languages.

Language Versions

Conformance with **process standards** such as ISO 26262 requires **coding guidelines** for used programming languages.

This limits the use of newer standards until there is a suitable coding guideline.

woven by TOYOTA

Overview

- Introduction
- Standards
- **Tooling**
- Testing
- Error Handling
- Design Impacts
- Summary

woven by TOYOTA

Tooling

Any tool that impacts the **generated code** or the **verification and validation evidence** needs to be suitably **Qualified**.

Tooling

Any tool that impacts the **generated code** or the **verification and validation evidence** needs to be suitably **Qualified**.

- Compiler

Tooling

Any tool that impacts the **generated code** or the **verification and validation evidence** needs to be suitably **Qualified**.

- Compiler
- Code Generator

woven by TOYOTA

Tooling

Any tool that impacts the **generated code** or the **verification and validation evidence** needs to be suitably **Qualified**.

- Compiler
- Code Generator
- Static Analyser

Tooling

Any tool that impacts the **generated code** or the **verification and validation evidence** needs to be suitably **Qualified**.

- Compiler
- Code Generator
- Static Analyser
- Testing Tools

Tooling

The qualification requirements depend on:

Tooling

The qualification requirements depend on:

- the Safety Integrity Level of the software

Tooling

The qualification requirements depend on:

- the Safety Integrity Level of the software
- the consequences of an error introduced by the tool

Tooling

The qualification requirements depend on:

- the Safety Integrity Level of the software
- the consequences of an error introduced by the tool
- the chance of an error going undetected

Static Analysis

Safety Critical Coding Standards have lots of rules.

Static Analysis

Safety Critical Coding Standards have lots of rules.

Enforcing by code review is **tedious** and **error-prone**.

Static Analysis

Safety Critical Coding Standards have lots of rules.

Enforcing by code review is **tedious** and **error-prone**.

Static analysis tools are **essential**.

Static Analysis

We run Static Analysis in our CI system. You cannot merge to `main` unless the static analysis passes.

Static Analysis

We run Static Analysis in our CI system. You cannot merge to `main` unless the static analysis passes.

You can also run it locally. This is slow, but better than waiting for CI to fail before making a change.

Static Analysis

We run Static Analysis in our CI system. You cannot merge to `main` unless the static analysis passes.

You can also run it locally. This is slow, but better than waiting for CI to fail before making a change.

The CI analysis result is part of our **Safety Case**.

woven by TOYOTA

Static Analysis

We run Static Analysis in our CI system. You cannot merge to `main` unless the static analysis passes.

You can also run it locally. This is slow, but better than waiting for CI to fail before making a change.

The CI analysis result is part of our **Safety Case**.

General Guideline: Automate Required Checks

Overview

- Introduction
- Standards
- Tooling
- **Testing**
- Error Handling
- Design Impacts
- Summary

woven by TOYOTA

Testing

You can't just review the code and say "LGTM. Ship it!"

Testing

You can't just review the code and say "LGTM. Ship it!"

You need to verify that it really does what is intended.

Testing

You can't just review the code and say "LGTM. Ship it!"

You need to verify that it really does what is intended.

That means testing.

Testing

You can't just review the code and say "LGTM. Ship it!"

You need to verify that it really does what is intended.

That means testing.

Automated testing, for repeatability and **evidence**.

woven by TOYOTA

Testing Coverage

We would like “100% coverage” from our tests.

Testing Coverage

We would like “100% coverage” from our tests.

What does that mean?

Line Coverage

100% Line coverage $\Rightarrow$ every line is covered by a test.

Line Coverage

100% Line coverage ⇒ every line is covered by a test.

```
unsigned foo(unsigned a, unsigned b) {
  if(a < b) return 0;
  return a + b;
}

TEST(Foo, FooWithOneAndTwoReturnsZero) {
  ASSERT_EQ(foo(1, 2), 0);
}
```

Line Coverage

100% Line coverage ⇒ every line is covered by a test.

```
unsigned foo(unsigned a, unsigned b) {
  if(a < b) return 0;
  return a + b; // not covered
}

TEST(Foo, FooWithOneAndTwoReturnsZero) {
  ASSERT_EQ(foo(1, 2), 0);
}
```

Line Coverage

100% Line coverage can be tricky

woven by TOYOTA

Line Coverage

100% Line coverage can be tricky

```
enum class colour {red, green};
void foo(unsigned arg) {
  switch(get_colour(arg)){
  case colour::red: do_red_things(); break;
  case colour::green: do_green_things(); break;
  default: error_invalid_colour();
  }
}
```

Line Coverage

100% Line coverage can be tricky

```
enum class colour {red, green};
void foo(unsigned arg) {
  switch(get_colour(arg)){
  case colour::red: do_red_things(); break;
  case colour::green: do_green_things(); break;
  default: error_invalid_colour();
  }
}
```

The `default` case might be impossible to reach.

woven by TOYOTA

Branch Coverage

100% Branch coverage $\Rightarrow$ every branch is covered by a test.

Branch Coverage

100% Branch coverage ⇒ every branch is covered by a test.

```
unsigned bar(unsigned a, unsigned b) {
  if(a < b)
    a = 23;
  return a + b;
}

TEST(Bar, BarWithOneAndTwoReturnsTwentyFive) {
  ASSERT_EQ(bar(1, 2), 25);
}
```

Branch Coverage

100% Branch coverage ⇒ every branch is covered by a test.

```
unsigned bar(unsigned a, unsigned b) {
  if(a < b) // "else" branch is not covered
    a = 23;
  return a + b;
}

TEST(Bar, BarWithOneAndTwoReturnsTwentyFive) {
  ASSERT_EQ(bar(1, 2), 25);
}
```

Modified Condition/Decision Coverage

M C/D C is about ensuring all aspects of complex conditions are checked, without exhaustive combination checking.

Modified Condition/Decision Coverage

M C/D C is about ensuring all aspects of complex conditions are checked, without exhaustive combination checking.

```
if(A && (B || C)) { foo(); } else { bar(); }
```

Modified Condition/Decision Coverage

M C/D C is about ensuring all aspects of complex conditions are checked, without exhaustive combination checking.

```
if(A && (B || C)) { foo(); } else { bar(); }
```

What are the possible conditions to consider?

Full condition coverage

```
if(A && (B || C)) { foo(); } else { bar(); }
```

A	B	C	Result
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	false
false	true	true	false
false	true	false	false
false	false	true	false
false	false	false	false

Modified Condition/Decision Coverage

This is exponential in the number of conditions, and is generally untenable.

Modified Condition/Decision Coverage

This is exponential in the number of conditions, and is generally untenable.

M C/D C deals with this by limiting the choices:

For each condition X there must be two invocations where the result is different and X is the only changed input.

Modified Condition/Decision Coverage

```
if(A && (B || C)) { foo(); } else { bar(); }
```

A	B	C	Result	Change
false	true	false	false	-
true	true	false	true	A
true	**false**	false	false	B
true	false	**true**	true	C

woven by TOYOTA

Modified Condition/Decision Coverage

100% **M C/D C** ensures that each element in the overall condition affects the outcome

Requirements Coverage

the implemented software unit shall be verified ...to provide evidence for ...confidence in the absence of unintended functionality

— *ISO 26262*

Requirements Coverage

> *the implemented software unit shall be verified ...to provide evidence for ...confidence in the absence of unintended functionality*
>
> *— ISO 26262*

To have confidence that there is nothing unintended happening, you need to know what **is** intended.

Requirements Coverage

> *the implemented software unit shall be verified ...to provide evidence for ...confidence in the absence of unintended functionality*
>
> *— ISO 26262*

To have confidence that there is nothing unintended happening, you need to know what **is** intended.

That means you need to **review your tests against your requirements**.

Requirements Coverage

100% line, branch, and M C/D coverage is no use if the software doesn’t do what you want.

Requirements Coverage

100% line, branch, and M C/D coverage is no use if the software doesn't do what you want.

Tests need to be reviewed against requirements, to ensure that all the necessary cases are covered, including boundary conditions and error cases.

Quantities of Test code

The vast majority of our codebase is test code.

Quantities of Test code

The vast majority of our codebase is test code.

For every line of code there is approximately 4-5 lines of test code.

Quantities of Test code

The vast majority of our codebase is test code.

For every line of code there is approximately 4-5 lines of test code.

Test code is often parameterised to reduce duplication.

Overview

- Introduction
- Standards
- Tooling
- Testing
- **Error Handling**
- Design Impacts
- Summary

woven by TOYOTA

Error Handling

"What if something goes wrong?"

Error Handling

“What if something goes wrong?”

This needs to be always on your mind when writing Safety Critical Software.

Error Handling

There are three types of error to handle:

Error Handling

There are three types of error to handle:

- Invalid input

Error Handling

There are three types of error to handle:

- Invalid input
- Undesired behaviour of **other code or systems**

Error Handling

There are three types of error to handle:

- Invalid input
- Undesired behaviour of **other code or systems**
- Logic errors

Error Handling

There are three types of error to handle:

- Invalid input
- Undesired behaviour of **other code or systems**
- Logic errors

Failure to anticipate the first two is a **logic error**.

Invalid Input

Any input you receive from a user or external system could be invalid.

Invalid Input

Any input you receive from a user or external system could be invalid.

- Accidental error

Invalid Input

Any input you receive from a user or external system could be invalid.

- Accidental error
- Corruption in transit

Invalid Input

Any input you receive from a user or external system could be invalid.

- Accidental error
- Corruption in transit
- Hardware fault

Invalid Input

Any input you receive from a user or external system could be invalid.

- Accidental error
- Corruption in transit
- Hardware fault
- External conditions outside design constraints

Invalid Input

Invalid Input must therefore be **expected** and **incorporated in the design.**

Invalid Input

Invalid Input must therefore be **expected** and **incorporated in the design.**

$\Rightarrow$ validate input, log validation errors, report errors, use default values, skip operations, etc.

Undesired behaviour of external systems

External code or systems might not behave as desired.

Undesired behaviour of external systems

External code or systems might not behave as desired.

- Transient problem due to external conditions

Undesired behaviour of external systems

External code or systems might not behave as desired.

- Transient problem due to external conditions
- Fault in other system

Undesired behaviour of external systems

External code or systems might not behave as desired.

- Transient problem due to external conditions
- Fault in other system
- External conditions outside design constraints

Undesired behaviour of external systems

External code or systems might not behave as desired.

- Transient problem due to external conditions
- Fault in other system
- External conditions outside design constraints
- Corruption of request or response

Undesired behaviour of external systems

Undesired Behaviour must therefore be **expected** and **incorporated in the design.**

Undesired behaviour of external systems

Undesired Behaviour must therefore be **expected** and **incorporated in the design**.

⇒ check error codes, verify success, propagate errors, retry operations, etc.

woven by TOYOTA

Watchdogs

Safety Critical **Systems** usually need to keep running.

Watchdogs

Safety Critical **Systems** usually need to keep running.

Such systems therefore have **watchdogs** — something that resets the system if it fails.

Watchdogs and Logic Errors

It is often safer to reset the system than continue after a **logic error.**

Watchdogs and Logic Errors

It is often safer to reset the system than continue after a **logic error**.

A logic error means the system is in a state you didn't anticipate.

Watchdogs and Logic Errors

It is often safer to reset the system than continue after a **logic error**.

A logic error means the system is in a state you didn't anticipate.

Continuing therefore has unknown consequences.

Logic Errors

Eliminate **logic errors** through testing and review.

Logic Errors

Eliminate **logic errors** through testing and review.

Terminate/Reset/Reboot if a logic error is detected.

Preconditions

Precondition

Something that **must** be true when a function is invoked.

Preconditions

Precondition

Something that **must** be true when a function is invoked.

It is a **logic error** to invoke a function when its preconditions are not met.

Preconditions

If the caller cannot check, it shouldn't be a precondition.

Preconditions

If the caller cannot check, it shouldn't be a precondition.

Failure to validate external input is a **logic error**.

Preconditions

Our `PRECONDITION` macro terminates the process on violations:

```
/// frobnigate the widget with the specified index
/// @pre index is in the range [0, max_valid_index())
void frobnigate(std::size_t index) {
  PRECONDITION(index < max_valid_index());
  // ...
}
```

Preconditions

Calling a function when its precondition is not satisfied is **never intended**.

Preconditions

Calling a function when its precondition is not satisfied is **never intended**.

⇒ **Something is wrong**, but **we don't know what**.

Preconditions

Calling a function when its precondition is not satisfied is **never intended**.

⇒ **Something is wrong**, but **we don't know what**.

⇒ We should stop running to minimize harm. The **watchdog** will keep the **system** going.

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling
- **Design Impacts**
- Summary

Design Impacts

Eliminate or **Mitigate** as many sources of error as possible.

Design Impacts

Eliminate or **Mitigate** as many sources of error as possible.

Favour **compilation errors** over **runtime errors.**

Design Impacts

Eliminate or **Mitigate** as many sources of error as possible.

Favour **compilation errors** over **runtime errors.**

Make things **correct by construction.**

Use `-Werror`

Crank up your compiler warnings, and set warnings as errors.

Use `-Werror`

Crank up your compiler warnings, and set warnings as errors.

Compiler warnings are the first line of defense against accidental errors.

Use `-Werror`

Crank up your compiler warnings, and set warnings as errors.

Compiler warnings are the first line of defense against accidental errors.

Eliminating warnings usually makes your code clearer **for reviewers.**

woven by TOYOTA

Use `[[nodiscard]]`

Ignoring error codes can change small problems into big problems.

Use `[[nodiscard]]`

Ignoring error codes can change small problems into big problems.

Discarding RAII return values can lead to resources being released too soon.

Use `[[nodiscard]]`

Ignoring error codes can change small problems into big problems.

Discarding RAII return values can lead to resources being released too soon.

Discarding raw resource handles can lead to leaks.

Use Strong Types

Create abstractions for everything.

Use Strong Types

Create abstractions for everything.

Use `span` instead of pointers.

Use Strong Types

Create abstractions for everything.

Use `span` instead of pointers.

Use a `quantity` template to ensure that you can't pass values in the wrong units.

woven by TOYOTA

Use Strong Types

Create abstractions for everything.

Use `span` instead of pointers.

Use a `quantity` template to ensure that you can't pass values in the wrong units.

Create `foo_id` and `bar_id` types rather than using `unsigned` to avoid mixing.

Resource Acquisition Is Initialization

Every resource should be managed with a type that **acquires** or **takes ownership** of the resource in its constructor, and **releases the resource in its destructor**.

Why use RAII?

Local objects always get their destructors run when the block is exited, **however that exit happens.**

Why use RAII?

Local objects always get their destructors run when the block is exited, **however that exit happens.**

`break`, `continue`, `return`, `throw` all exit blocks, and all run destructors.

Why use RAII?

Local objects always get their destructors run when the block is exited, **however that exit happens.**

`break`, `continue`, `return`, `throw` all exit blocks, and all run destructors.

This simplifies code for releasing resources and reduces the chance of errors

Why use RAII?

```
some_result foo() {
  some_resource x = acquire_resource();
  if(!do_stuff(x)) { // returns true
    return first_result;
  }
  release_resource(x);
  return second_result;
}
```

Why use RAII?

```
some_result foo() {
  some_resource x = acquire_resource();
  if(!do_stuff(x)) { // returns false
    return first_result;
  }
  release_resource(x); // SKIPPED
  return second_result;
}
```

woven by TOYOTA

Why use RAII?

```
some_result foo() {
  RAIIClass x; // acquires resource
  if(!do_stuff(x)) { // returns true
    return first_result;
  }
  // NO EXPLICIT CLEANUP
  return second_result;
} // releases resource
```

Why use RAII?

```
some_result foo() {
  RAIIClass x; // acquires resource
  if(!do_stuff(x)) { // returns false
    return first_result;
  }
  // NO EXPLICIT CLEANUP
  return second_result;
} // STILL releases resource
```

Avoid Dynamic Memory Allocation

Dynamic memory allocation has three big downsides:

Avoid Dynamic Memory Allocation

Dynamic memory allocation has three big downsides:

- Large worst-case times

Avoid Dynamic Memory Allocation

Dynamic memory allocation has three big downsides:

- Large worst-case times
- Unpredictable timing

Avoid Dynamic Memory Allocation

Dynamic memory allocation has three big downsides:

- Large worst-case times
- Unpredictable timing
- Fragmentation

Avoid Dynamic Memory Allocation

Safety-Critical Coding Standards often say “don’t use dynamic memory” for these reasons.

Avoid Dynamic Memory Allocation

Safety-Critical Coding Standards often say “don’t use dynamic memory” for these reasons.

$\Rightarrow$ Pre-allocate memory

Preallocation

- Use `optional` to allocate space, but delay construction

Preallocation

- Use `optional` to allocate space, but delay construction
- Use “inline” containers that contain the storage directly

woven by TOYOTA

Preallocation

- Use `optional` to allocate space, but delay construction
- Use “inline” containers that contain the storage directly
- Use static-storage-duration objects

Preallocation

- Use `optional` to allocate space, but delay construction
- Use “inline” containers that contain the storage directly
- Use static-storage-duration objects
- Allocate on startup:

Preallocation

- Use `optional` to allocate space, but delay construction
- Use “inline” containers that contain the storage directly
- Use static-storage-duration objects
- Allocate on startup:
  - On the heap with `std::make_unique`, or

Preallocation

- Use `optional` to allocate space, but delay construction
- Use “inline” containers that contain the storage directly
- Use static-storage-duration objects
- Allocate on startup:
  - On the heap with `std::make_unique`, or
  - On the stack in `main`

Avoid recursion

> *Functions shall not call themselves, either directly or indirectly.*
>
> — *AUTOSAR C++14 Guidelines*

Avoid recursion

> *Functions shall not call themselves, either directly or indirectly.*
>
> — *AUTOSAR C++14 Guidelines*

Every recursive function can be transformed into an iterative form with auxiliary data.

Avoid recursion

> *Functions shall not call themselves, either directly or indirectly.*
>
> — *AUTOSAR C++14 Guidelines*

Every recursive function can be transformed into an iterative form with auxiliary data.

This forces you to consider the storage requirements up front.

Splitting to facilitate testing

Q: How do you test `private` member functions?

Splitting to facilitate testing

Q: How do you test `private` member functions?

A: Extract a class where they are `public` members, and test that.

woven by TOYOTA

Splitting to facilitate testing

Q: How do you test `private` member functions?

A: Extract a class where they are `public` members, and test that.

The same applies to complex logic in functions.

Splitting to facilitate testing

Q: How do you test `private` member functions?

A: Extract a class where they are `public` members, and test that.

The same applies to complex logic in functions.

Extract a function for each branch of a conditional or `switch` and test those separately.

woven by TOYOTA

Splitting to facilitate testing

Smaller classes and functions are:

Splitting to facilitate testing

Smaller classes and functions are:

- easier to review

Splitting to facilitate testing

Smaller classes and functions are:

- easier to review
- easier to test

Splitting to facilitate testing

Smaller classes and functions are:

- easier to review
- easier to test
- easier for static analysers to check

Overview

- Introduction
- Standards
- Tooling
- Testing
- Error Handling
- Design Impacts
- **Summary**

Summary

- Safety Critical Software means lives are at risk

Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards

Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards
- Static Analysis is vital

woven by TOYOTA

Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards
- Static Analysis is vital
- You need lots of Automated Tests

Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards
- Static Analysis is vital
- You need lots of Automated Tests
- You still need extensive code review

woven by TOYOTA

Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards
- Static Analysis is vital
- You need lots of Automated Tests
- You still need extensive code review
- You must think about error cases

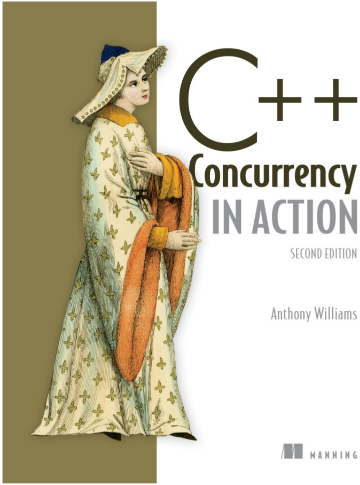
Summary

- Safety Critical Software means lives are at risk
- Follow Process Standards and Coding Standards
- Static Analysis is vital
- You need lots of Automated Tests
- You still need extensive code review
- You must think about error cases
- Your design must reflect these considerations

woven by TOYOTA

Questions?

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the first Concurrency TS

cplusplusconcurrencyinaction.com

woven by TOYOTA