Software and Safety

Anthony Williams

Woven by Toyota
https://www.woven.toyota

November 2025

Definitions

Definitions

Safety Critical Software

The consequence for failure is injury or loss of life.

Definitions

Software with Safety Consequences

Failure can cause quality-of-life problems for users — e.g. loss of money, loss of work, loss of trust, exposure of personal details, identity theft.

Software for Dangerous Devices

Software for Dangerous Devices

- Automotive control software — steering, brakes, engine control

Software for Dangerous Devices

- Automotive control software — steering, brakes, engine control
- Aircraft cockpit software

Software for Dangerous Devices

- Automotive control software — steering, brakes, engine control
- Aircraft cockpit software
- Factory machinery control software

Software for Dangerous Devices

- Automotive control software — steering, brakes, engine control
- Aircraft cockpit software
- Factory machinery control software
- Nuclear power plant control software

Software in Medical Devices

Software in Medical Devices

- Pacemakers

Software in Medical Devices

- Pacemakers
- Defibrilators

Software in Medical Devices

- Pacemakers
- Defibrilators
- Artificial lungs

Software in Medical Devices

- Pacemakers
- Defibrilators
- Artificial lungs
- CT Scanners

Software Designed to Increase Safety

Software Designed to Increase Safety

- Driving assistance software

Software Designed to Increase Safety

- Driving assistance software
- Automatic cut-off on cookers

Software Designed to Increase Safety

- Driving assistance software
- Automatic cut-off on cookers
- Fire alarm and sprinkler systems

Software for Management Tasks

Software for Management Tasks

- Traffic light control software

Software for Management Tasks

- Traffic light control software
- Air traffic management software

Software for Management Tasks

- Traffic light control software
- Air traffic management software
- Railway management software

Personal Data Management

woven by TOYOTA

Personal Data Management

- Payment software

Personal Data Management

- Payment software
- Banking software

woven by TOYOTA

Personal Data Management

- Payment software
- Banking software
- Phone software

Personal Data Management

- Payment software
- Banking software
- Phone software
- Social Media

Incidental Software

Incidental Software

- Games

Incidental Software

- Games
- Internet-connected devices

Incidental Software

- Games
- Internet-connected devices
- Web browsers

Incidental Software

- Games
- Internet-connected devices
- Web browsers
- Spreadsheets

Software Development Standards

Safety Critical Software has Standards to conform to.

Software Development Standards

Safety Critical Software has Standards to conform to.

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

If your software is not “Safety Critical”, then these don’t apply.

Software Development Standards

If your software is not “Safety Critical”, then these don’t apply.

You should still consider “safety”, and the potential impact of failure on your users.

Problem Causes

Problem Causes

A big cause of problems is circumstances the developers didn't expect to happen:

	Knowns	Unknowns
known	known Knowns	known Unknowns
unknown	unknown Knowns	unknown Unknowns

Problem Causes

A big cause of problems is circumstances the developers didn't expect to happen:

- Events they thought "would never happen"

	Knowns	Unknowns
known	known Knowns	known Unknowns
unknown	unknown Knowns	unknown Unknowns

Problem Causes

A big cause of problems is circumstances the developers didn't expect to happen:

- Events they thought "would never happen"
- Events they **didn't even consider as possibilities**

	Knowns	Unknowns
known	known Knowns	known Unknowns
unknown	unknown Knowns	unknown Unknowns

Problem Causes

Another big cause is **mistakes**

Problem Causes

Another big cause is **mistakes**

The developer **intended** one thing, but the code does something else

Problem Causes

Another big cause is **mistakes**

The developer **intended** one thing, but the code does something else

We commonly call these cases **bugs**

Problem Causes

Other causes can include:

Problem Causes

Other causes can include:

- Hardware failure — sensors, motors, displays, storage, …

Problem Causes

Other causes can include:

- Hardware failure — sensors, motors, displays, storage, …
- “Bad” user input — steering off a bridge, entering incorrect drug dosage, …

Problem Causes

Other causes can include:

- Hardware failure — sensors, motors, displays, storage, …
- “Bad” user input — steering off a bridge, entering incorrect drug dosage, …

From a safety point of view, these are **design** issues.

Problem Causes

Other causes can include:

- Hardware failure — sensors, motors, displays, storage, …
- “Bad” user input — steering off a bridge, entering incorrect drug dosage, …

From a safety point of view, these are **design** issues.

Can we design the **system** to handle these?

Another Definition

undefined behavior

behavior for which this document imposes no requirements

— C++23 §3.65 [defns.undefined]

Another Definition

> ***undefined behavior***
> *behavior for which this document imposes no requirements*
>
> — *C++23 §3.65 [defns.undefined]*

"no requirements" means the code can do **anything at all**.

Undefined Behaviour

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

— ISO 26262

Undefined Behaviour

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

Code with UB can do **anything at all**. This is likely **unintended functionality**, so must be avoided.

Undefined Behaviour

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

Code with UB can do **anything at all**. This is likely **unintended functionality**, so must be avoided.

Safety-Critical code **must provide evidence**.

Memory Safety?

There is a lot of publicity about the lack of Memory Safety being a **big problem**.

woven by TOYOTA

Memory Safety?

There is a lot of publicity about the lack of Memory Safety being a **big problem**.

Such problems are often an example of Undefined Behaviour:

woven by TOYOTA

Memory Safety?

There is a lot of publicity about the lack of Memory Safety being a **big problem**.

Such problems are often an example of Undefined Behaviour:

- Lifetime errors

Memory Safety?

There is a lot of publicity about the lack of Memory Safety being a **big problem**.

Such problems are often an example of Undefined Behaviour:

- Lifetime errors
- Bounds errors

woven by TOYOTA

Undefined Behaviour

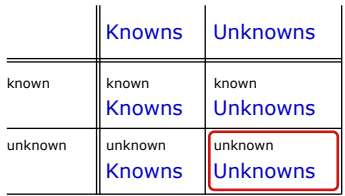
Undefined behaviour is often a **consequence of an unexpected event**.

	Knowns	Unknowns
known	known Knowns	known Unknowns
unknown	unknown Knowns	unknown Unknowns

Undefined Behaviour

Undefined behaviour is often a **consequence of an unexpected event**.

If we are serious about “Safety”, we need to work hard to ensure we **eliminate unexpected events**.



Consequences

This all impacts how you develop software.

Consequences

This all impacts how you develop software.

In particular, it affects how you **specify** that software, and **test** that software.

Validating Input

A tester walks into a bar...

Validating Input

A tester walks into a bar...

- Orders 1 beer

Validating Input

A tester walks into a bar...

- Orders 1 beer
- Orders 99 beers

Validating Input

A tester walks into a bar...

- Orders 1 beer
- Orders 99 beers
- Orders an orange juice

Validating Input

A tester walks into a bar...

- Orders 1 beer
- Orders 99 beers
- Orders an orange juice
- Orders -1 beers

Validating Input

A tester walks into a bar...

- Orders 1 beer
- Orders 99 beers
- Orders an orange juice
- Orders -1 beers
- Orders four candles

Validating Input

A tester walks into a bar...

- Orders 1 beer
- Orders 99 beers
- Orders an orange juice
- Orders -1 beers
- Orders four candles
- Orders a `DROP TABLE DRINKS`

Validating Input

Your software needs to have an **intended** behaviour for **every possible input**.

Validating Input

Your software needs to have an **intended** behaviour for **every possible input**.

Including network traffic, file contents, and interactions with external software or devices.

Validating Input

Your software needs to have an **intended** behaviour for **every possible input**.

Including network traffic, file contents, and interactions with external software or devices.

This is a **design** problem.

Validating Input

Design interactions so input is checkable.

Validating Input

Design interactions so input is checkable.

Check input for **structure** before **interpreting** values:

Validating Input

Design interactions so input is checkable.

Check input for **structure** before **interpreting** values:

- Use a checksum to validate

Validating Input

Design interactions so input is checkable.

Check input for **structure** before **interpreting** values:

- Use a checksum to validate
- Use a **schema** to check format

Validating Input

Design interactions so input is checkable.

Check input for **structure** before **interpreting** values:

- Use a checksum to validate
- Use a **schema** to check format
- Validate sizes and offsets

Validating Input

Design interactions so input is checkable.

Check input for **structure** before **interpreting** values:

- Use a checksum to validate
- Use a **schema** to check format
- Validate sizes and offsets

Decide on behaviour for invalid input.

Validating Input

Choosing appropriate behaviour is the tricky part.

Validating Input

Choosing appropriate behaviour is the tricky part.

- What if the input is invalid?

Validating Input

Choosing appropriate behaviour is the tricky part.

- What if the input is invalid?
- What if the input is valid **on its own**, but not **in the current system state**?

Validating Input

Choosing appropriate behaviour is the tricky part.

- What if the input is invalid?
- What if the input is valid **on its own**, but not **in the current system state**?
- What if the “input” is that something **external** isn’t working as expected?

Validating Input

Choosing appropriate behaviour is the tricky part.

- What if the input is invalid?
- What if the input is valid **on its own**, but not **in the current system state**?
- What if the “input” is that something **external** isn’t working as expected?

Can you design the system so these are “expected” and handled gracefully?

Validating Input

Input validation is also a **testing** problem.

Validating Input

Input validation is also a **testing** problem.

Malformed and invalid inputs can outnumber valid inputs.

Validating Input

Input validation is also a **testing** problem.

Malformed and invalid inputs can outnumber valid inputs.

Usually it is impractical to test with all possible inputs.

Validating Input

White box analysis of problematic inputs should feed tests.

Validating Input

White box analysis of problematic inputs should feed tests.

Fuzz testing can help fill holes.

Validating Input

White box analysis of problematic inputs should feed tests.

Fuzz testing can help fill holes.

Test all layers of your end product: low level libraries, middleware, application processes, the whole “system”

Identifying Undefined Behaviour

Testing only gets us so far.

Identifying Undefined Behaviour

Testing only gets us so far.

One of the **most dangerous** consequences of UB is when **the code does exactly what you naively expected, in testing**.

Identifying Undefined Behaviour

Testing only gets us so far.

One of the **most dangerous** consequences of UB is when **the code does exactly what you naively expected, in testing**.

The UB is still there, and can potentially cause safety issues in production.

Identifying Undefined Behaviour

Testing only gets us so far.

One of the **most dangerous** consequences of UB is when **the code does exactly what you naively expected, in testing**.

The UB is still there, and can potentially cause safety issues in production.

Memory Safety issues often fall into this category.

Identifying Undefined Behaviour

Static Analysis can identify some UB at build time.

Identifying Undefined Behaviour

Static Analysis can identify some UB at build time.

Sanitizers and **hardened** libraries can identify some UB at runtime.

Identifying Undefined Behaviour

Static Analysis can identify some UB at build time.

Sanitizers and **hardened** libraries can identify some UB at runtime.

Use sanitizers in testing and hardened libraries in testing and production.

Contracts

Preconditions specify what is required to be true when you use a function or class.

Contracts

Preconditions specify what is required to be true when you use a function or class.

Postconditions specify what will be true when a function returns.

Checking Contracts: if

You can often check preconditions in code:

```
// precondition: ptr must not be null
void foo(bar* ptr) {
  if(ptr == nullptr) { // contract check
    precondition_violated();
  }
  do_something(*ptr);
}
```

Checking Contracts: macros

You can often check preconditions in code:

```
// precondition: ptr must not be null
void foo(bar* ptr) {
  PRECONDITION(ptr != nullptr);


  do_something(*ptr);
}
```

Checking Contracts: C++26

You can often check preconditions in code:

```
// precondition: ptr must not be null
void foo(bar* ptr)
  pre(ptr != nullptr) { // C++26 syntax


  do_something(*ptr);
}
```

Checking Contracts: Static Analysis

You can often check preconditions in code:

```
// precondition: ptr must not be null
void foo(bar* ptr) {
  /*@ requires @*/        // CodeQL Annotation
  assert(ptr != nullptr);

  do_something(*ptr);
}
```

Checking Contracts: Performance

A checked contract requires evaluating the condition, and branching to the handler on failure.

```
void foo(bar* ptr) {
  PRECONDITION(ptr != nullptr);
  do_something(*ptr);
}
```

```
"foo(bar*)":
  test rdi, rdi
  je .L3
  jmp "do_something(bar&)"
"foo(bar*) [clone .cold]":
.L3:
  push rax
  call "violation_handler"
```

Checking Contracts: Performance

Compilers are good at optimizing out redundant checks.

```
void baz(bar* ptr){
  PRECONDITION(ptr!=nullptr);
  foo(ptr);
  foo(ptr);
}
```

```
"baz(bar*)":
  sub rsp, 24
  test rdi, rdi
  je .L9
  mov QWORD PTR [rsp+8], rdi
  call "do_something(bar&)"
  mov rdi, QWORD PTR [rsp+8]
  add rsp, 24
  jmp "do_something(bar&)"
"baz(bar*) [clone .cold]":
.L9:
  call "violation_handler"
```

Checking Contracts: Performance

Compilers are good at optimizing out redundant checks.

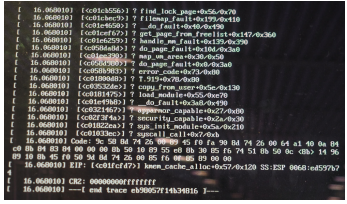
```
void loop(){
  bar values[10]{};

  for(unsigned i=0;i<10;++i) {
    baz(&values[i]);
  }
}
```

```
"loop()":
  push rbx
  sub rsp, 16
  lea rbx, [rsp+6]
.L12:
  mov rdi, rbx
  call "do_something(bar&)"
  mov rdi, rbx
  add rbx, 1
  call "do_something(bar&)"
  lea rax, [rsp+16]
  cmp rax, rbx
  jne .L12
  add rsp, 16
  pop rbx
  ret
```

Violated Contracts

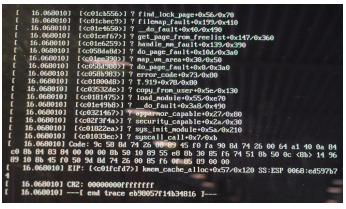
If a contract is violated, **you have a bug**.



Violated Contracts

If a contract is violated, **you have a bug**.

In testing, notify the developer ASAP, with the maximum information.

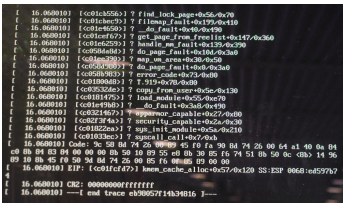


Violated Contracts

If a contract is violated, **you have a bug**.

In testing, notify the developer ASAP, with the maximum information.

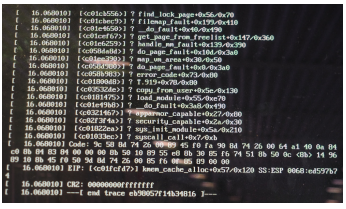
- Break into debugger



Violated Contracts

If a contract is violated, **you have a bug**.

In testing, notify the developer ASAP, with the maximum information.

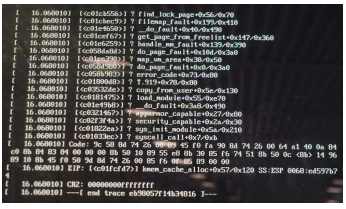


- Break into debugger
- Stack trace and core dump

Violated Contracts

If a contract is violated, **you have a bug**.

In testing, notify the developer ASAP, with the maximum information.



- Break into debugger
- Stack trace and core dump
- Custom logging of state

Violated Contracts

In production, the violation handler should do **the safest thing** for your use case.

Violated Contracts

In production, the violation handler should do **the safest thing** for your use case.

Safety Critical code might trigger a watchdog to reset to a “known safe state”.

Violated Contracts

In production, the violation handler should do **the safest thing** for your use case.

Safety Critical code might trigger a watchdog to reset to a “known safe state”.

Other code may do something else.

Violated Contracts

In production, the violation handler should do **the safest thing** for your use case.

Safety Critical code might trigger a watchdog to reset to a “known safe state”.

Other code may do something else.
Including continuing.

Simple Ways to Avoid UB

Simple Ways to Avoid UB

- Initialize all your variables

Simple Ways to Avoid UB

- Initialize all your variables
- Use checked functions like `at`

Simple Ways to Avoid UB

- Initialize all your variables
- Use checked functions like `at`
- Use `span` and `string_view` rather than raw pointers

Simple Ways to Avoid UB

- Initialize all your variables
- Use checked functions like `at`
- Use `span` and `string_view` rather than raw pointers
- Use range-based `for` loops rather than indexing

Simple Ways to Avoid UB

- Initialize all your variables
- Use checked functions like `at`
- Use `span` and `string_view` rather than raw pointers
- Use range-based `for` loops rather than indexing
- Use RAII to manage lifetimes

Defense in Depth

Use the “Swiss Cheese” approach:

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems
- Use static analysis

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems
- Use static analysis
- Test with potentially problematic input

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems
- Use static analysis
- Test with potentially problematic input
- Fuzz test

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems
- Use static analysis
- Test with potentially problematic input
- Fuzz test
- Use sanitizers and hardened libraries

Defense in Depth

Use the “Swiss Cheese” approach:

- Design your system to minimize the potential for problems
- Use static analysis
- Test with potentially problematic input
- Fuzz test
- Use sanitizers and hardened libraries
- Use contracts

Software and Safety

- Many types of software have safety consequences

Software and Safety

- Many types of software have safety consequences
- Think carefully about desired behaviour

Software and Safety

- Many types of software have safety consequences
- Think carefully about desired behaviour
- What is the “safe” behaviour if a problem is discovered?

Software and Safety

- Many types of software have safety consequences
- Think carefully about desired behaviour
- What is the “safe” behaviour if a problem is discovered?
- Use code patterns for avoiding problems

Software and Safety

- Many types of software have safety consequences
- Think carefully about desired behaviour
- What is the “safe” behaviour if a problem is discovered?
- Use code patterns for avoiding problems
- Use “Swiss Cheese” defense in depth

Announcement

Arene Base Library

Woven by Toyota is releasing our foundation-level C++ library for Safety-Critical Systems as Open Source.

Arene Base Library

Woven by Toyota is releasing our foundation-level C++ library for Safety-Critical Systems as Open Source.

It will be released under the Apache License 2.0 with LLVM exception.

Arene Base Library

- Designed for C++14 and C++17 to comply with AUTOSAR and MISRA coding guidelines

Arene Base Library

- Designed for C++14 and C++17 to comply with AUTOSAR and MISRA coding guidelines
- Includes a subset of C++ Standard Library for embedded platforms

Arene Base Library

- Designed for C++14 and C++17 to comply with AUTOSAR and MISRA coding guidelines
- Includes a subset of C++ Standard Library for embedded platforms
- Provides containers without dynamic memory allocation such as `inline_vector`, `inline_map` and `inline_string`

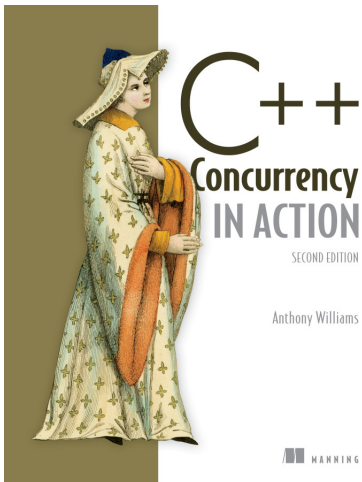
Arene Base Library

- Designed for C++14 and C++17 to comply with AUTOSAR and MISRA coding guidelines
- Includes a subset of C++ Standard Library for embedded platforms
- Provides containers without dynamic memory allocation such as `inline_vector`, `inline_map` and `inline_string`
- Provides backports of more recent library features such as `string_view`, `span` and `mdspan`

Arene Base Library

- Designed for C++14 and C++17 to comply with AUTOSAR and MISRA coding guidelines
- Includes a subset of C++ Standard Library for embedded platforms
- Provides containers without dynamic memory allocation such as `inline_vector`, `inline_map` and `inline_string`
- Provides backports of more recent library features such as `string_view`, `span` and `mdspan`
- Includes enforced preconditions for bounds checks

My Book



C++ Concurrency in Action
Second Edition

Covers C++17 and the
first Concurrency TS

45% discount from Manning until
24th November 2025 with code
`meetingcpp25`

`https://hubs.la/Q03RQ9Q40`

Questions?

Image Attributions

The Woven by Toyota logo is copyright Woven by Toyota.

The following images are used (cropped and resized) under Creative Commons Licenses:

Creative Commons Attribution 2 (CC-BY-2)

https://commons.wikimedia.org/wiki/File:AirBaltic_Bombardier_CS300_launch_event_%2831581897816%29.jpg
https://www.flickr.com/photos/salsaboy/3844860345/

Creative Commons Attribution 4 (CC-BY-4)

https://universe.roboflow.com/final-year-project-e9b7f/car-warning-light-symbol

Image Attributions

Create Commons Attribution Share-Alike 2 (CC-BY-SA-2)

https://commons.wikimedia.org/wiki/File:
Giant_Assassin_Bug_(Platymeris_guttatipennis)_(11838791835).jpg

Create Commons Attribution Share-Alike 3 (CC-BY-SA-3)

https://commons.wikimedia.org/wiki/File:UnknownUnknownsEN.svg

Create Commons Attribution Share-Alike 4 (CC-BY-SA-4)

https://commons.wikimedia.org/wiki/File:CPI_Microlyth_Pacemaker.jpg
https://commons.wikimedia.org/wiki/File:
Stack_trace_at_Katajanokka_Terminal.jpg

Image Attributions

The following images are Public Domain:

https://www.pexels.com/photo/a-person-making-a-payment-using-smartphone-5239806/
https://www.pexels.com/photo/a-woman-playing-video-game-7915492/
https://www.pexels.com/photo/dogs-lying-beside-the-gray-laptop-9040443/
https://commons.wikimedia.org/wiki/File:Masskruege.jpg
https://www.publicdomainpictures.net/en/view-image.php?image=42717&picture=ekg-electrocardiogram
https://picryl.com/media/a-sailor-operates-the-spn-43-air-search-radar-system-while-standing-approach-8473ae
https://openclipart.org/detail/337098/contract-signed
https://www.needpix.com/photo/1106547/