Building Safe and Reliable Surgical Robotics with C++

MILAD KHALEDYAN

Presenters

Milad Khaledyan
Software Robotics Engineer

Alexander Drew
Software Engineer

Johnson&Johnson
MedTech

Disclaimer: The views and opinions expressed in this presentation are solely those of the presenters and do not necessarily represent or reflect the views, positions, or policies of any company in the Johnson & Johnson Family of Companies.

Big Picture

1. Why Safety/Security in C++
2. Medical Device Failure Analysis
3. Brief Intro to Medical Device Standards, Documents, and Reports
4. Safety in C++ and What We Can Do
5. Coding Practices in Safety Critical Path
6. Final Words and Q&A

Safety/Security and C++

Security/Safety Concerns with C++

PRESS RELEASE | Nov. 10, 2022

America's Cyber Defense Agency

NATIONAL COORDINATOR FOR CRITICAL INFRASTRUCTURE SECURITY AND RESILIENCE

The Urgent Need for Memory Safety in Software Products

Released: September 20, 2023

Revised: December 06, 2023

The Case for Memory Safe Roadmaps

MITRE Common Weaknesses Enumeration

2023 CWE Top 25

Rank	ID	Name
1	CWE-787	Out-of-bounds Write
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
4	CWE-416	Use After Free
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
6	CWE-20	Improper Input Validation
7	CWE-125	Out-of-bounds Read
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
9	CWE-352	Cross-Site Request Forgery (CSRF)
10	CWE-434	Unrestricted Upload of File with Dangerous Type
11	CWE-862	Missing Authorization
12	CWE-476	NULL Pointer Dereference
13	CWE-287	Improper Authentication
14	CWE-190	Integer Overflow or Wraparound
15	CWE-502	Deserialization of Untrusted Data
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
18	CWE-798	Use of Hard-coded Credentials
19	CWE-918	Server-Side Request Forgery (SSRF)
20	CWE-306	Missing Authentication for Critical Function
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
22	CWE-269	Improper Privilege Management
23	CWE-94	Improper Control of Generation of Code ('Code Injection')
24	CWE-863	Incorrect Authorization
25	CWE-276	Incorrect Default Permissions

Recent Notable Talks on Safety in C++

Medical Device Failures Analysis

Medical Device Failure Analysis

2006-2011 FDA MAUDE database

Analysis of Safety-Critical Computer Failures in Medical Devices

Homa Alemzadeh, Ravishankar K. Iyer, and Zbigniew Kalbarczyk |
University of Illinois at Urbana-Champaign
Jai Raman | Rush University Medical Center

Malfunctioning medical devices are one of the leading causes of serious injury and death in the US. Analysis of human-written descriptions of recalls and adverse event reports reveals safety issues in these devices and provides insights on the future challenges in the design of safety-critical devices.

Electronic and computer-based devices are deployed widely in clinical and personalized settings, facilitated by shrinking hardware and increased portability and interconnectedness. But with ease of deployment comes a significant increase in device complexity and major challenges in reliability, patient safety, and security. Medical devices are often subject to a nonnegligible number of failures with potentially catastrophic impacts on patients. Between 2006 and 2011, 5,294 recalls and 1,154,451 adverse events were reported to the US Food and Drug Administration (FDA). As Figure 1 shows, since 2006, there was a 69.8 percent increase in the number of recalls and a 103.3 percent increase in the number of adverse events (reaching approximately 1,190 recalls [see Figure 1a], 92,600 patient injuries, and 4,590 deaths in 2011 [see Figure 1b]).

In this article, we focus on *computer-related recalls* related to failures of computer-based medical devices. During our measurement period, the number of computer-related recalls almost doubled, reaching an overall number of 1,210 (22.9 percent of all recalls), as Figure 1a shows. A study conducted during the six-year period between 1999 and 2005 attributed 1,261 recalls (33.4 percent) to software-based medical devices.[1] Our goal was to identify the major causes of computer-related failures in medical devices that impact patient safety. We define *computer-related failure* as any event causing a computer-based medical device to function improperly or present harm to patients or users owing to failures in the device's software, hardware, I/O, or battery.

We collected data from two public FDA databases: the Medical and Radiation Emitting Device Recalls database (referred to as the Recalls database) and the Manufacturer and User Facility Device Experience (MAUDE or Adverse Event Reports) database.[2] Through an in-depth study of recalls data, we characterized the computer-related failures based on

- *fault class*: the defective components that led to device failure,
- *failure mode*: the impact of failures on the device's safe functioning,
- *recovery action category*: the type of actions the manufacturer took to address the recall,
- *number of recalled devices*: the quantity of recalled devices distributed in the market, and

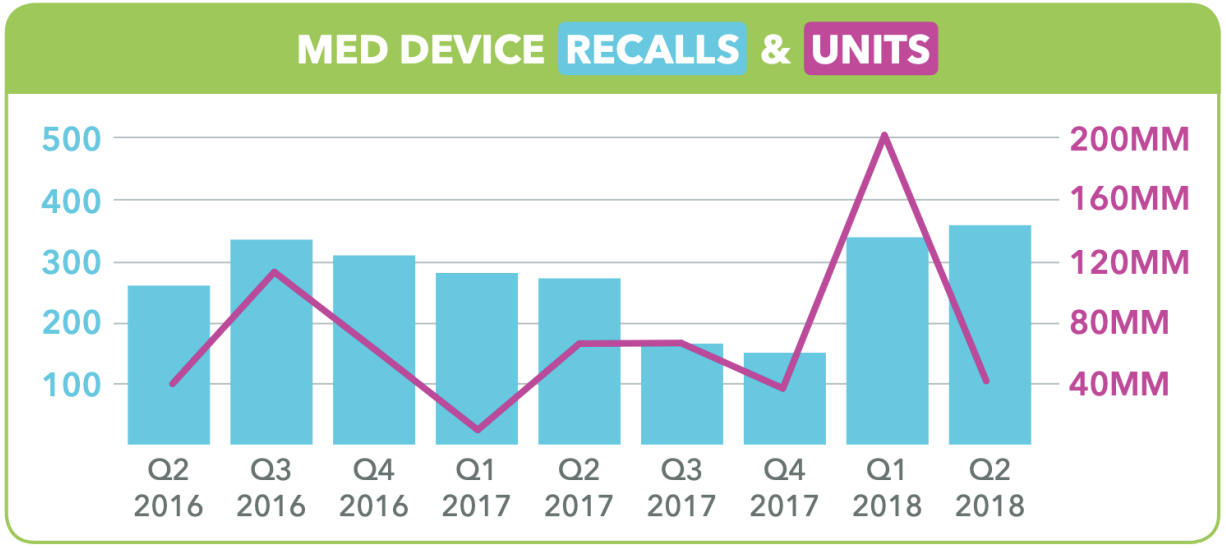
	Class I: high risk	Class II: medium risk	Class III: low risk	Total recalls
Software	14 (33.3%)	718 (65.6%)	46 (75.3%)	778 (64.3%)
Hardware	8 (19.0%)	158 (14.4%)	13 (27.4%)	179 (14.8%)
Other	10 (23.8%)	124 (11.3 %)	8 (12.3%)	142 (11.7%)
Battery	8 (19.0%)	57 (5.2%)	5 (6.8%)	70 (5.8%)
I/O	2 (4.8%)	38 (3.5%)	1 (2.7%)	41 (3.4%)
Total recalls	42 (3.5%)	1,095 (90.5%)	73 (6.0%)	1,210

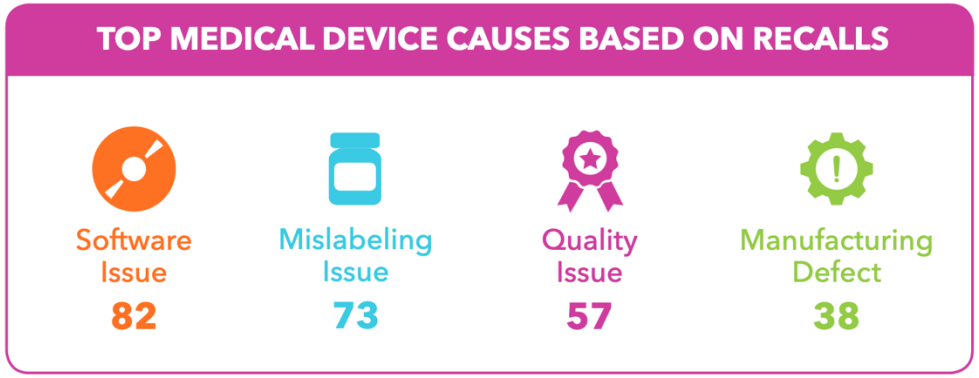
This study does NOT contain security related recalls!

Medical Device Failure Analysis

Recall Index Database

Stericycle® Expert Solutions™



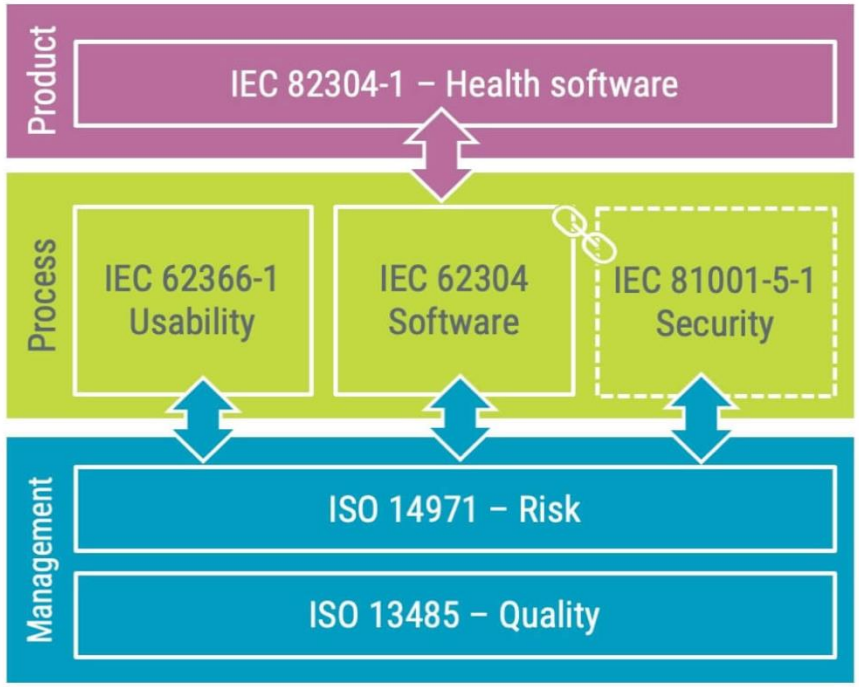


At 22.8%, software issues were the top cause of recalls for the ninth quarter in a row.

Regulatory Standards & Documents

Software Development in Medical Devices

100+ Standards, Documents and Technical reports



ISO/TS 82304-2 Health and wellness apps – Quality and reliability

IEC TR 60601-4-5 Guidance and interpretation – Safety-related technical security specifications

IEC 62443-4-1 Security for industrial automation and control systems – Part 4-1: Secure product development lifecycle

AAMI/CR510 Appropriate use of public cloud computing for quality systems and medical devices

AAMI TIR45 Guidance on the use of AGILE practices in the development of medical device software

AAMI CR34971 Application of ISO 14971 to Artificial Intelligence (AI) and Machine Learning (ML)

AAMI TIR57 Principles for medical device security - Risk management

www.medicaldevicehq.com

www.jhc-technology.com

Cybersecurity in Medical Devices: Quality System Considerations and Content of Premarket Submissions

Guidance for Industry and Food and Drug Administration Staff

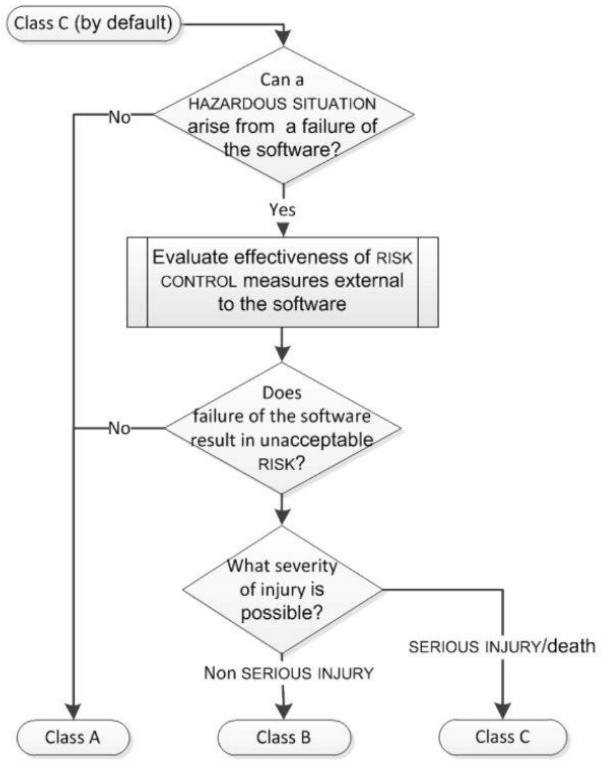
Document issued on September 27, 2023.

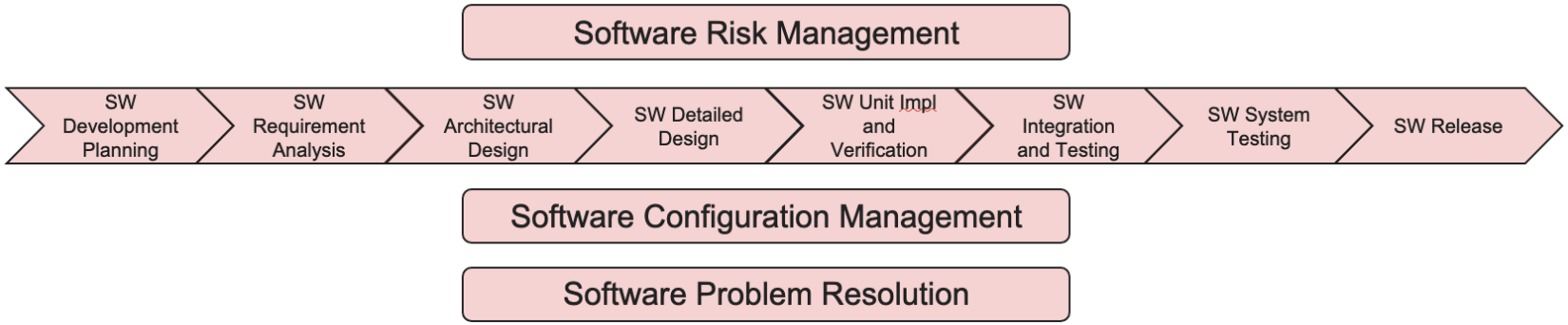
The draft of this document was issued on April 8, 2022.

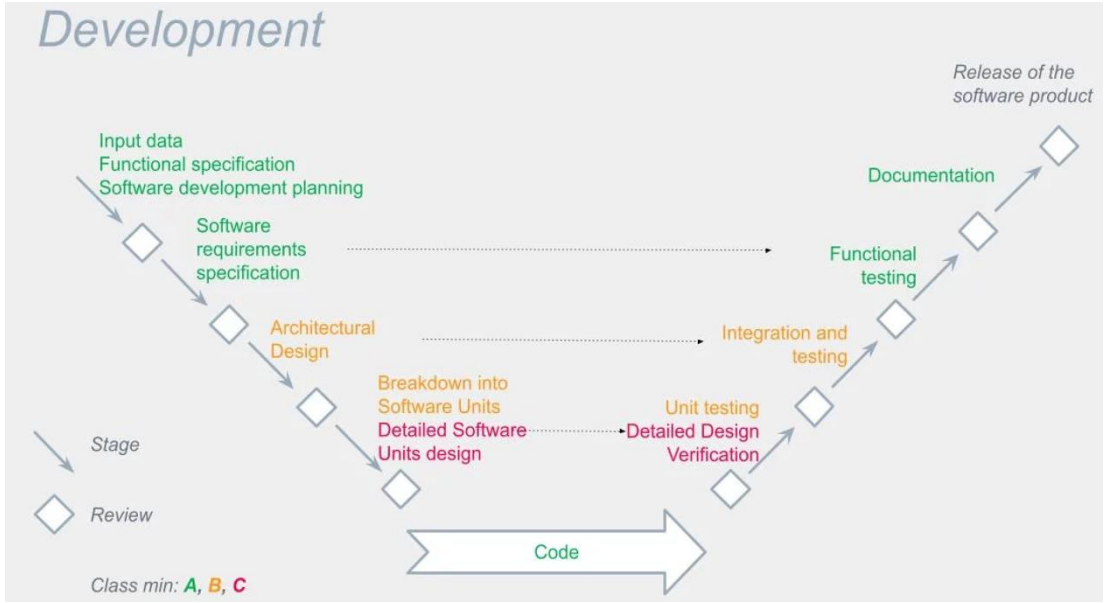
This document supersedes "Content of Premarket Submissions for Management of Cybersecurity in Medical Devices," issued October 2, 2014.

Software Development in Medical Devices

IEC 62304: Functional Safety

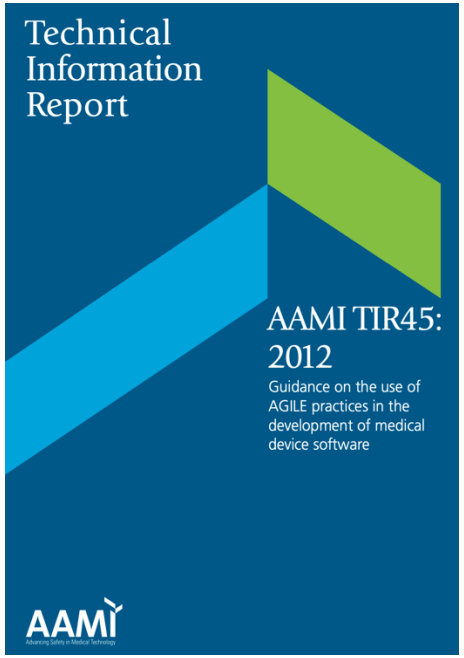






www.qualitiso.com

+



Software Development in Medical Devices

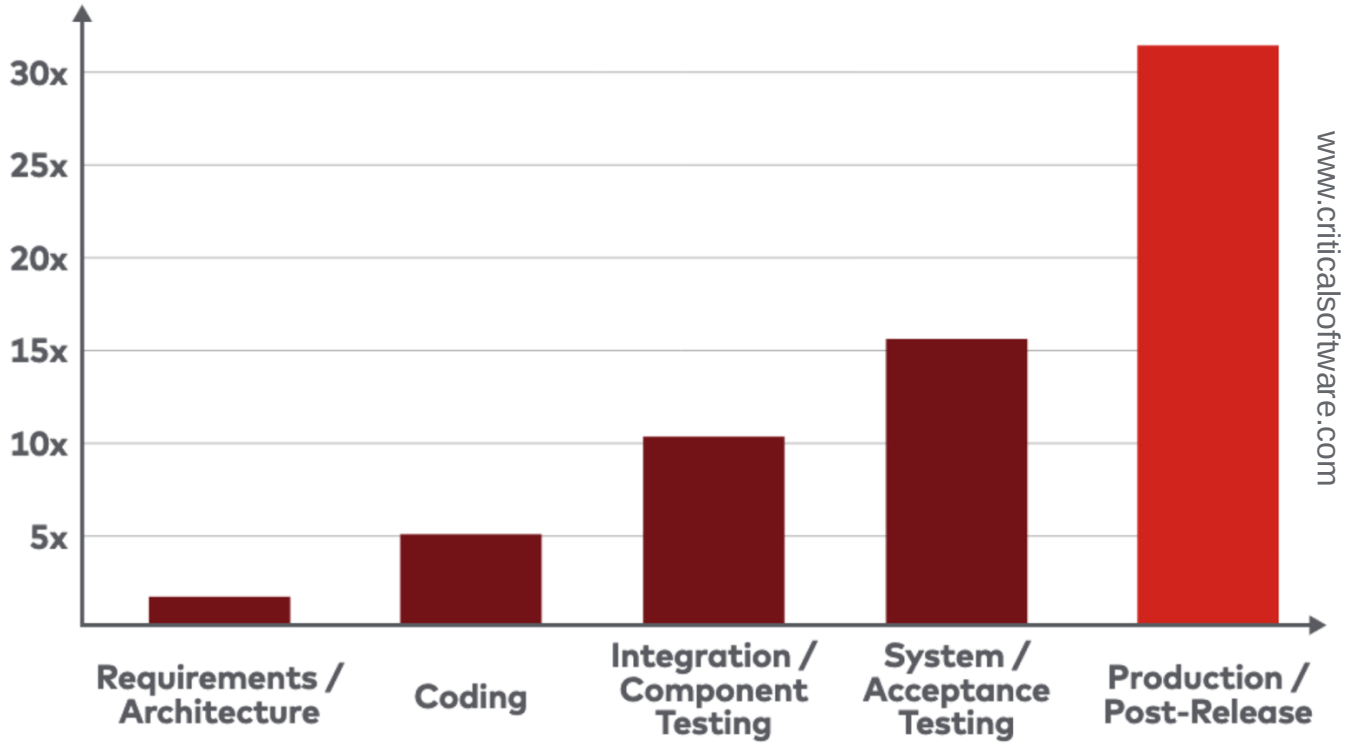
V&V

General Principles of Software Validation; Final Guidance for Industry and FDA Staff

Document issued on: January 11, 2002

This document supersedes the draft document, "General Principles of Software Validation, Version 1.1, dated June 9, 1997.

U.S. Department Of Health and Human Services
Food and Drug Administration
Center for Devices and Radiological Health
Center for Biologics Evaluation and Research



Relative cost to fix bugs

Software Development in Medical Devices

IEC/TR 80002-1:2009

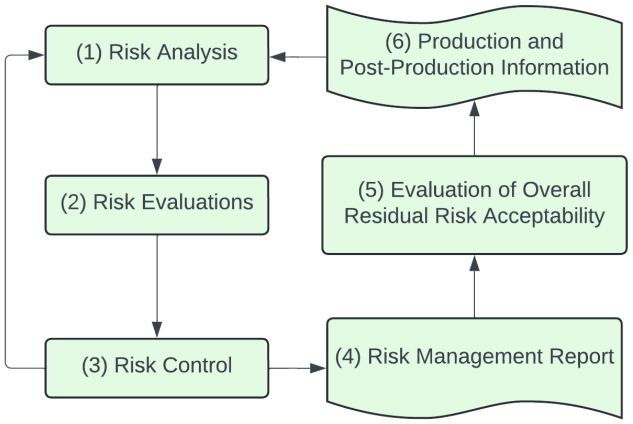


Table B.2 – Examples of software causes that can introduce side-effects *(continued)*

	VERIFICATION types: Analysis: static / dynamic / timing			
	Test (unit, integration)			
	Inspection			
Software causes	**RISK CONTROL measures**			
Miscellaneous				
Memory leaks	Avoid dynamic allocation of memory	✦	✦	D
SYSTEM deadlocks	Simple locking strategies (PROCESS can only have one resource locked at a given time), deadlock analysis			S
Re-entrancy	Ensure that all functions, including those in third-party libraries, which are called from interrupts (or multiple TASKS of different priorities) are designed to be re-entrant			D
Stack overflow	Run-time stack guards, high-water marks, stack analysis			S
Logic errors/syntax	Use source code analysis tool (such as Lint) and/or max. compiler warning level Dual DIVERSITY and cross-checks at critical control points	✦	✦	S
Infinite loops	Loop counters, loop timeouts, watchdog timer	✦	✦	
Code corruption	Power-up and run-time program image CRC check			
Dead code	If not removed insert an error check that will alarm or perform a safe shutdown if dead code (in custom or for off-the-shelf software components) begins to execute.			D
Incorrect conditional code	Ensure conditional compilation is used appropriately and only when necessary	✦		
Unintended macro side-effects	Use parenthesis around all macro parameters			S
Resource depletion	Stack, heap, and timing analyses			T
Incorrect alarm/alert prioritization	Stress testing			
Unauthorized features (“gold plating”, “back doors”, etc.)	Requirements and design reviews, trace matrices			
Incorrect order of operations/precedence	“Bread-crumbing”, Called from tracking			
SAFE state	Independent monitor			

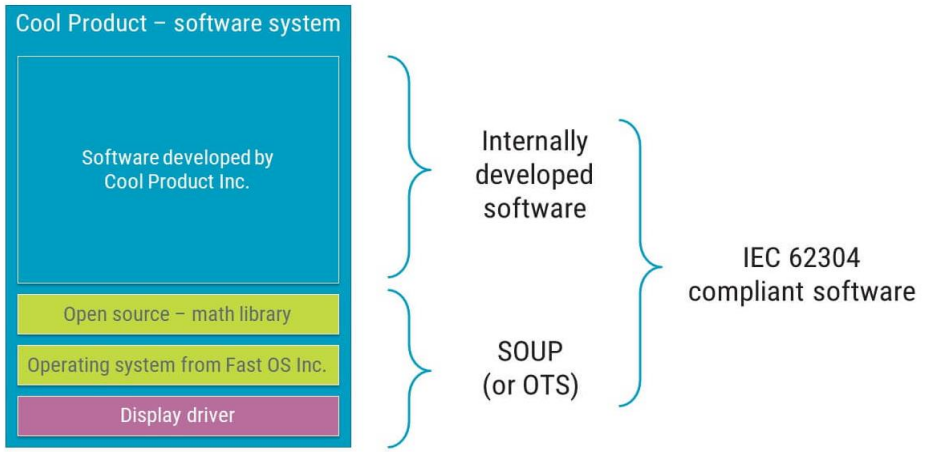
Software Development in Medical Devices

IEC 62304 & More

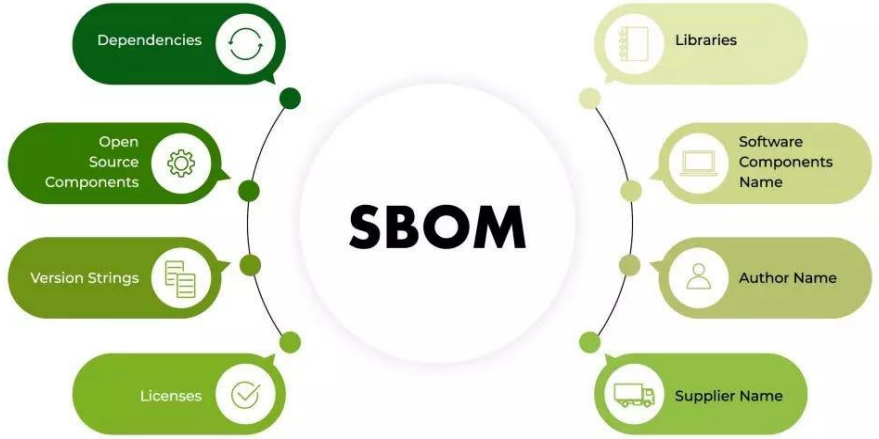
- **Software of Unknown Provenance (SOUP)**
  - Third-party libraries (Boost, Qt, etc.)
  - Properly documented/validated/verified/tested
  - Even the compiler needs to be verified!

- **Software Bill of Materials (SBOM)**
  - Vulnerability management
  - Compliance and reporting
  - Supply chain transparency

- **Cybersecurity Bill of Materials (CBOM)**
  - SBOM + Cybersecurity items
  - Usually interchangeable with SBOM



www.medicaldevicehq.com



www.chpk.medium.com

Software Development in Medical Devices

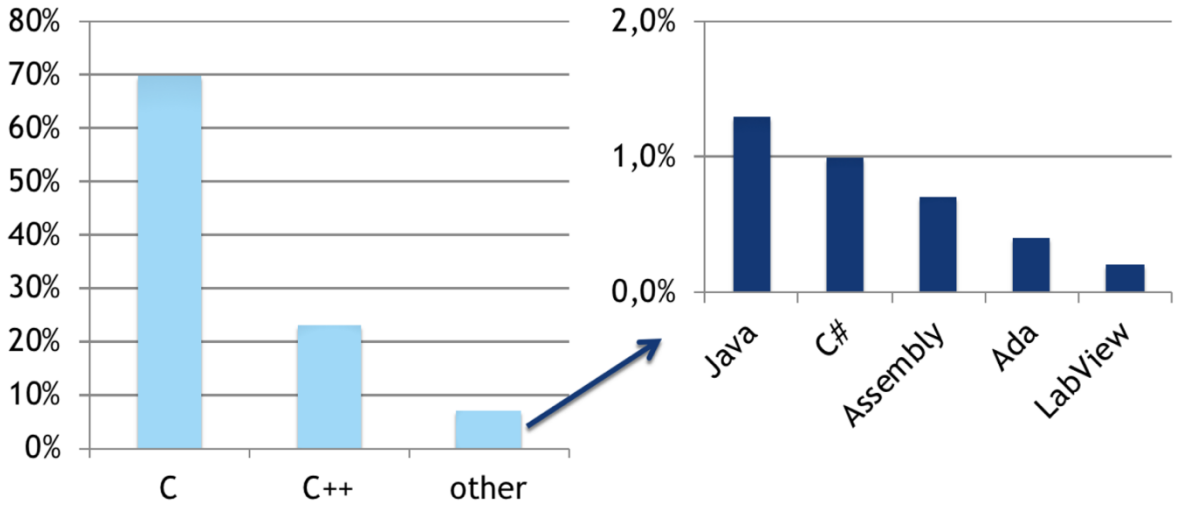
Is regulatory compliance enough?

- Standards are generic, high level, no specificity and prescriptiveness
- No strong language (only recommendations)
- Usability and human factors engineering aspects
- Requirements and system design errors
- Non-linear/indirect interactions between components or systems
- Demand for complex systems outpaces quality standards
- Perhaps less restrictive than automative and avionic

Software Development in Medical Devices

Barr Group 2018 Embedded Systems Safety & Security Survey

- **2 out of 3** products target Medical, Defense, Automative, and Industrial technologies
- **17%** lack of written coding standards
- **33%** don't perform static analysis
- **38%** don't comply with any safety standards
- **43%** don't do regular peer code reviews
- **41%** don't perform regression testing
- **1 in 6** designers are completely ignoring security



Strive to Achieve Correctness, Safety and Security

Definitions

Functional Correctness: Meeting specifications and requirements.

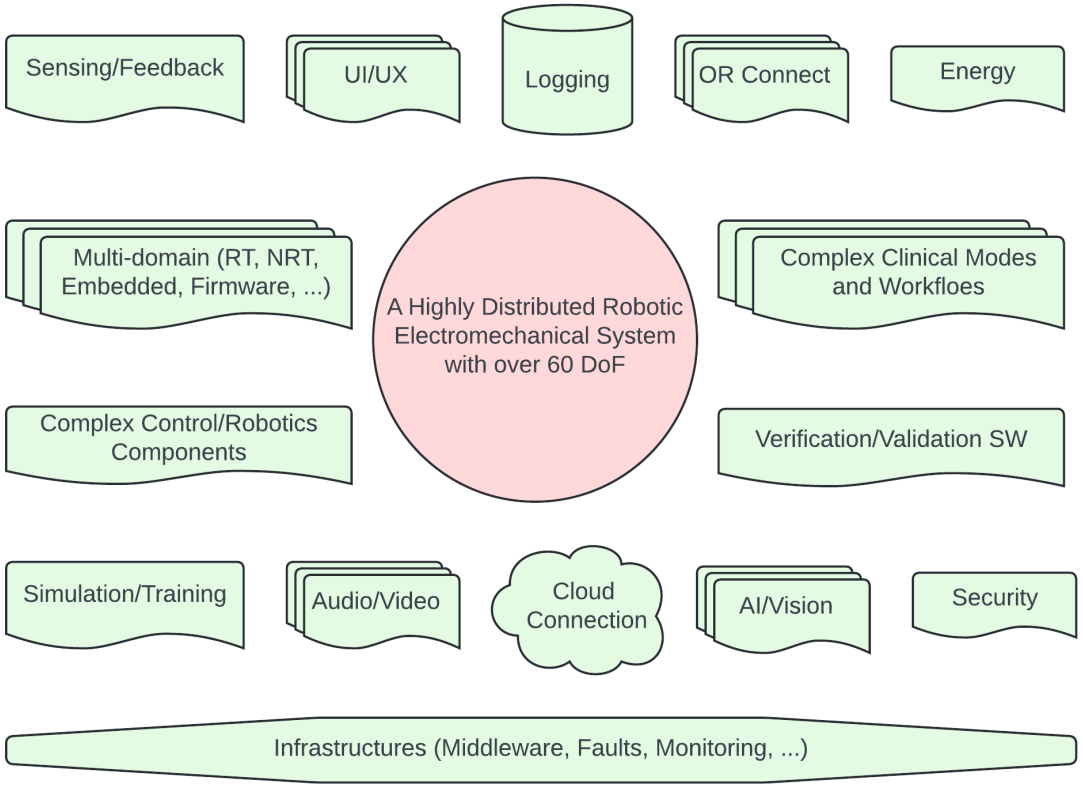
Functional Safety: Zero unintended behavior; Medical device operates correctly in response to inputs, including in failure scenarios (*Fail-safe Design*), to prevent harm or hazards to patient.

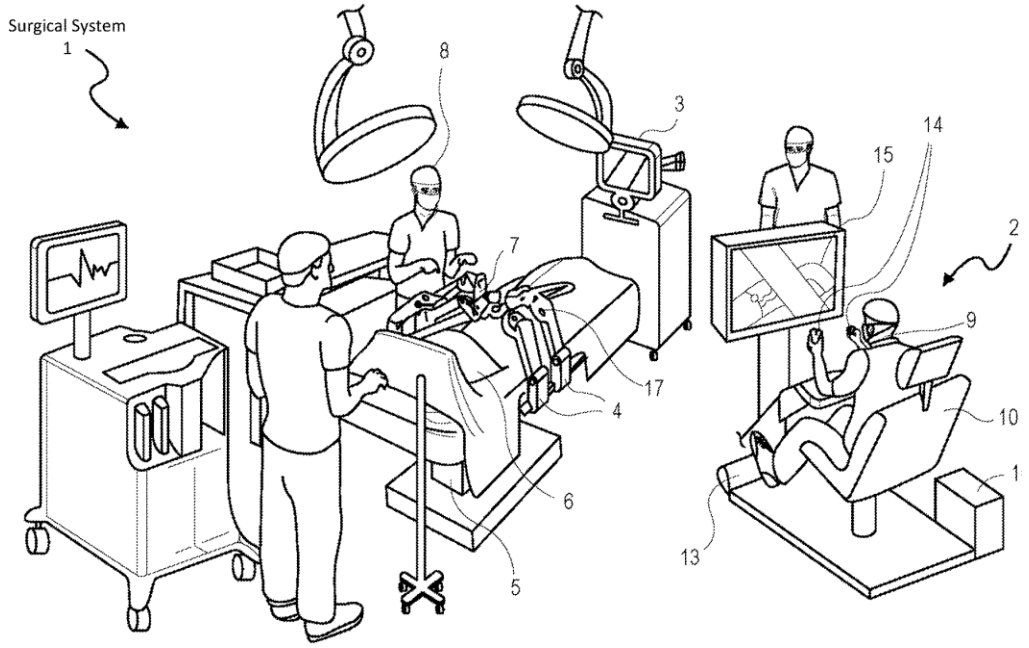
Security: Protection of systems, networks, and data from unauthorized access, attacks, damage, or theft.

Medical Device Use Case & C++

Robotically Assisted Surgical Platform

Our use case: A Class-C Medical Device

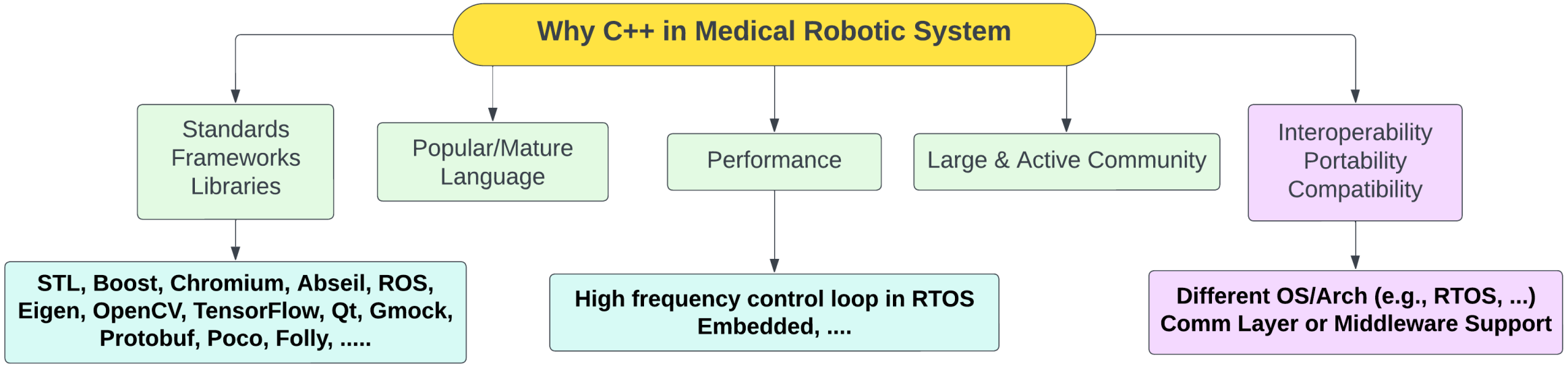




Millions of Lines of C++ Code

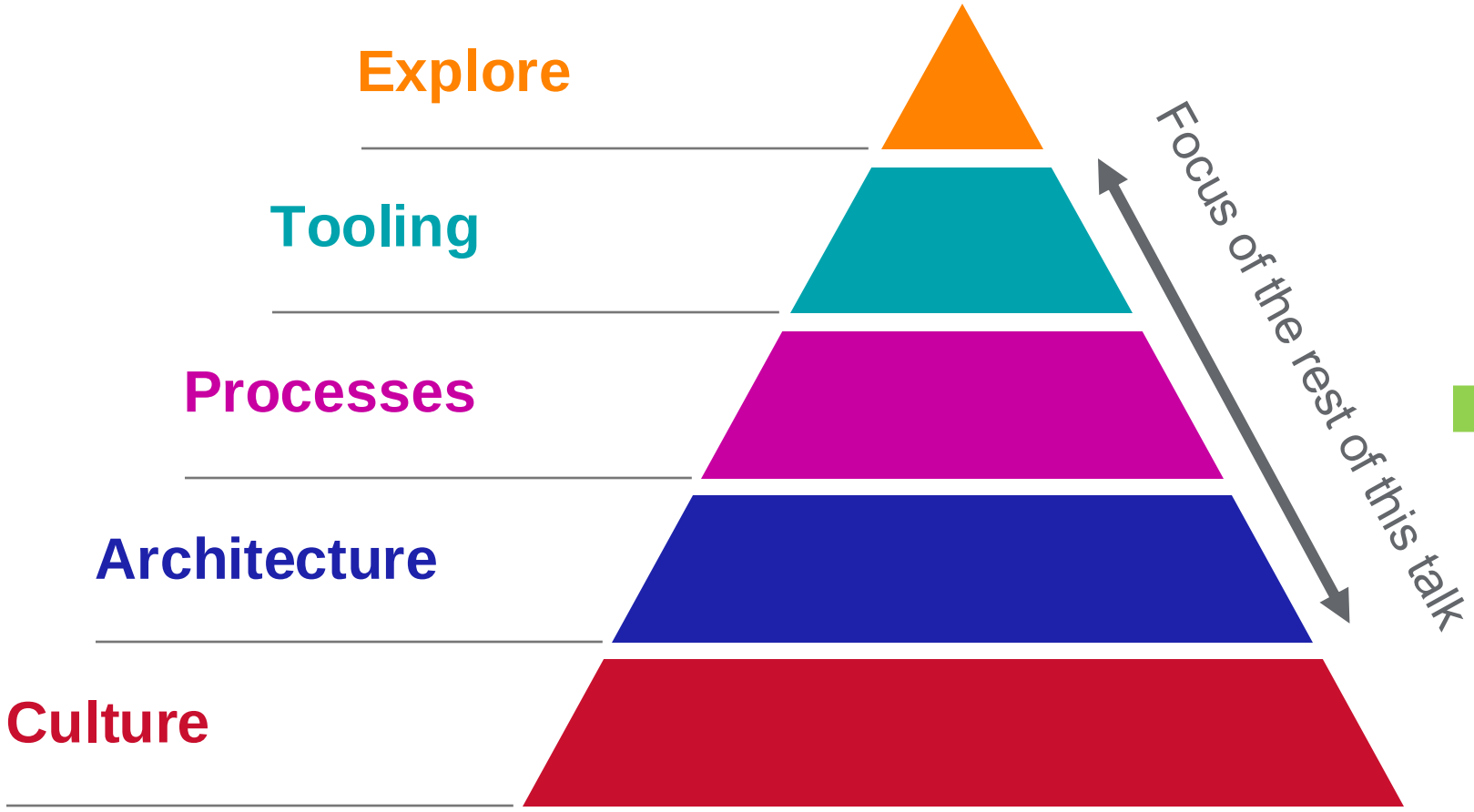
Why C++?

Reminder to a simple, but often overlooked, question!



What can we do then?

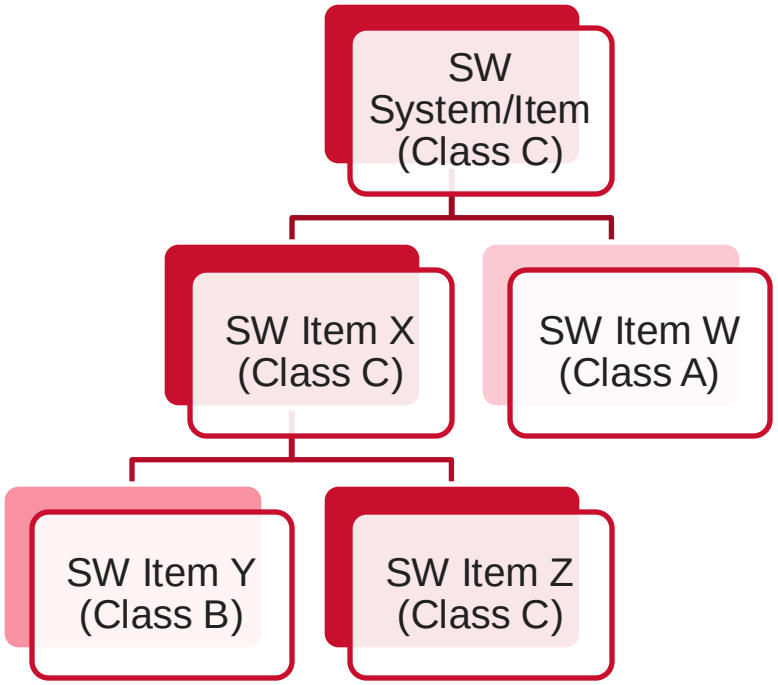
Can C++ usage be safe? **Mitigate Safety and Security Vulnerabilities**



Architecture

Risk Driven Architecture

Control and reduce risk



Overcoming Adverse Condition

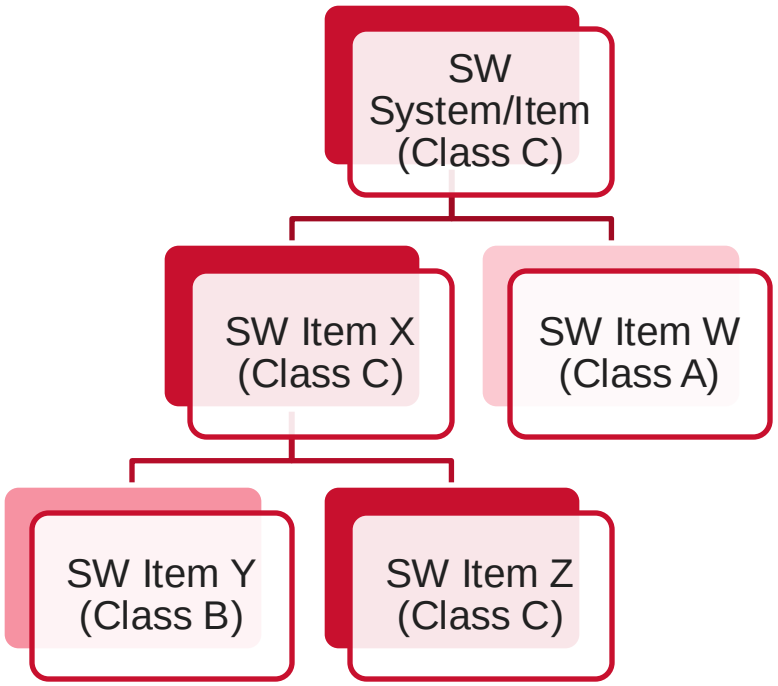
- Faults
- Limited Resources
- Limited Performance

Physical/Logical Segregation?

Reduce Coupling & Enhance Cohesion

Risk Driven Architecture

Safety Critical Path vs Non-Real Time Domain



Non-Real Time
- No strict timing guarantees
- e.g., Logging
- NRTPC

Soft Real Time
- Deadlines are more flexible but still affect performance
- e.g., Vision Overlays
- NRTPC, RTOS

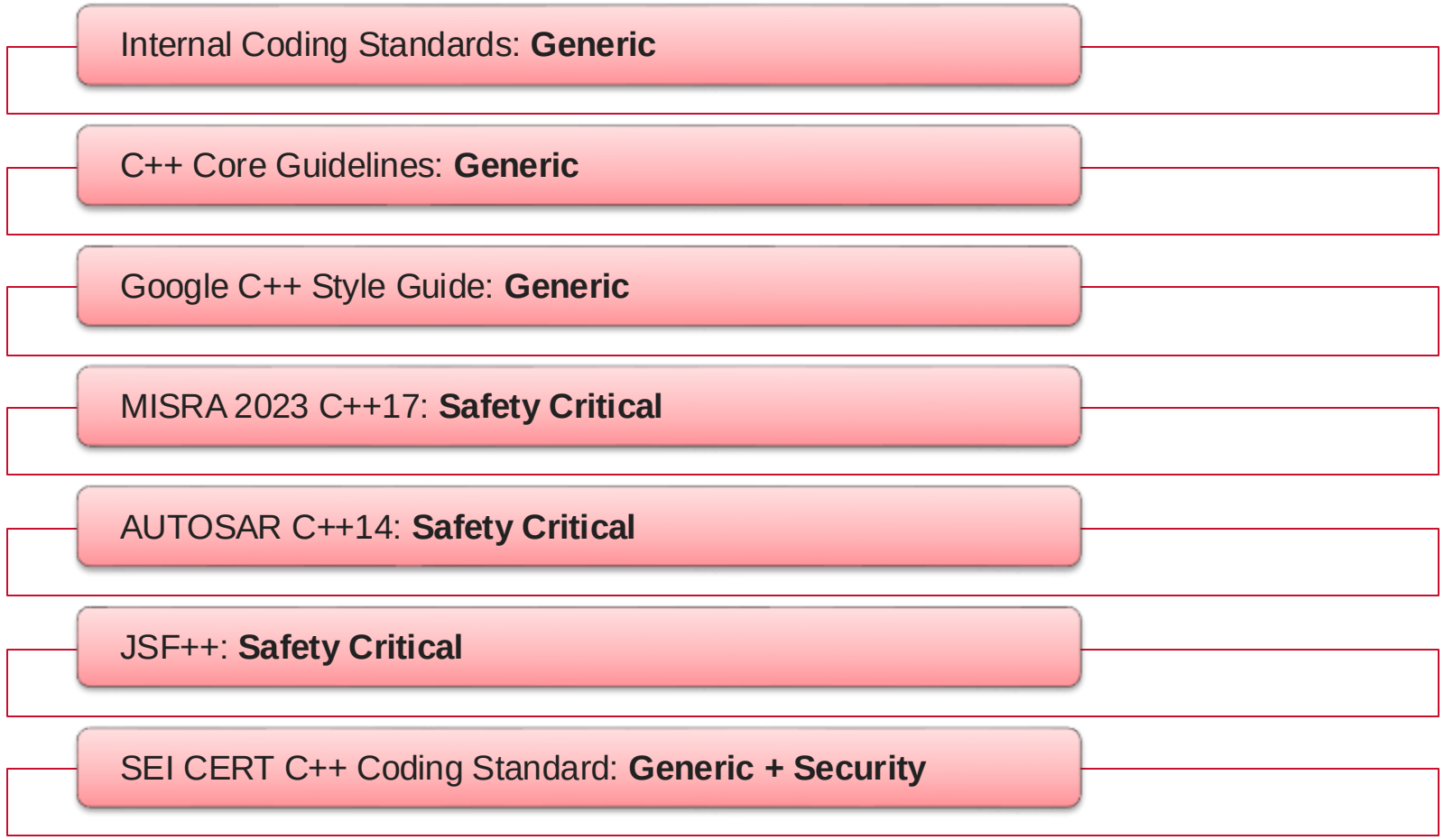
Firm Real Time
- Tiny amount of missed deadlines are ok.
- e.g., UDP comms
- RTOS

Hard Real time
- Deadlines are strict and non-negotiable
- e.g., Control Systems, Vision
- BareMetal, Embedded, RTOS

Processes & Tooling

Coding Standards

Coding standards are a key part of software acceptance criteria: IEC 62304 Annex B.5.5



Compiler Hardening

FDA Recommendation: Warnings are errors and ideally enable `-Werror -Wall -Wextra`

```
-O2 -Wall -Wformat -Wformat=2 -Wconversion -Wimplicit-fallthrough \
-Werror=format-security \
-U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=3 \
-D_GLIBCXX_ASSERTIONS \
-fstrict-flex-arrays=3 \
-fstack-clash-protection -fstack-protector-strong \
-Wl,-z,nodlopen -Wl,-z,noexecstack \
-Wl,-z,relro -Wl,-z,now \
-Wl,--as-needed -Wl,--no-copy-dt-needed-entries
```

Compiler Options Hardening Guide for C and C++

Compiler Hardening

Prioritize Memory and type safety

- Compiler Extensions should be disallowed

Compiler Flag	Supported Since	Description
`-Wconversion, -Wsign-conversion`	GCC 2.95.3, Clang 4.0.0	Enable implicit conversion warnings
`-fstack-protector-strong`	GCC 4.9.0 Clang 6.0.0	Enable run-time checks for stack-based buffer overflows. Can impact performance.
`-fno-delete-null-pointer-checks`	GCC 3.0.0, Clang 7.0.0	Force retention of null pointer checks
`-fno-strict-overflow`	GCC 4.2.0	Integer overflow may occur
`-fno-strict-aliasing`	GCC 2.95.3, Clang 18.0.0	Do not assume strict aliasing
`-ftrivial-auto-var-init`	GCC 12.0.0, Clang 8.0.0	Perform trivial auto variable initialization
`-Wconversion, -Wsign-conversion`	GCC 2.95.3, Clang 4.0.0	Enable implicit conversion warnings
`-fstack-clash-protection`	GCC 8.0.0, Clang 11.0.0	Enable run-time checks for variable-size stack allocation validity

Compiler Hardening

Prioritize Memory and type safety

- Treat obsolete C constructs as errors

Compiler Flag	Supported Since	Description
`-Werror=implicit`	GCC 2.95.3 Clang 2.6.0	Treat declarations that do not specify as type or functions used before being declared as errors
`-Werror=incompatible-pointer-types`	GCC 5.5.0 Clang 7.0.0	Treat conversion between pointers that have incompatible types as errors
`-Werror=int-conversion`	GCC 2.95.3 Clang 2.6.0	Treat implicit integer to pointer and pointer to integer conversions as errors

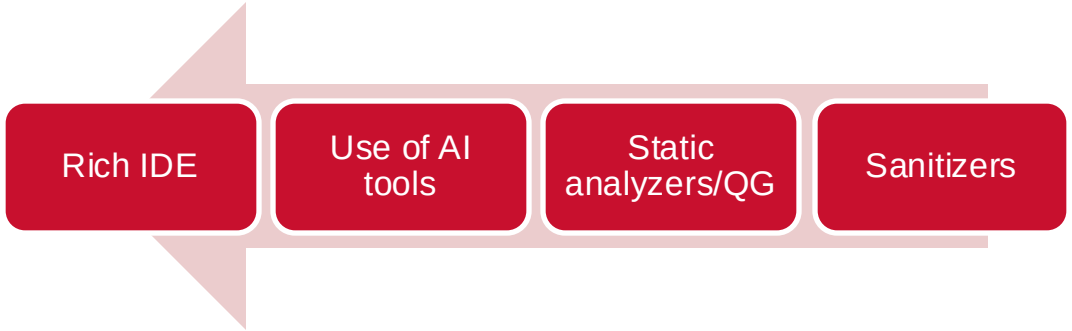
Compiler Hardening

Prioritize Memory, type and thread safety: sanitizers

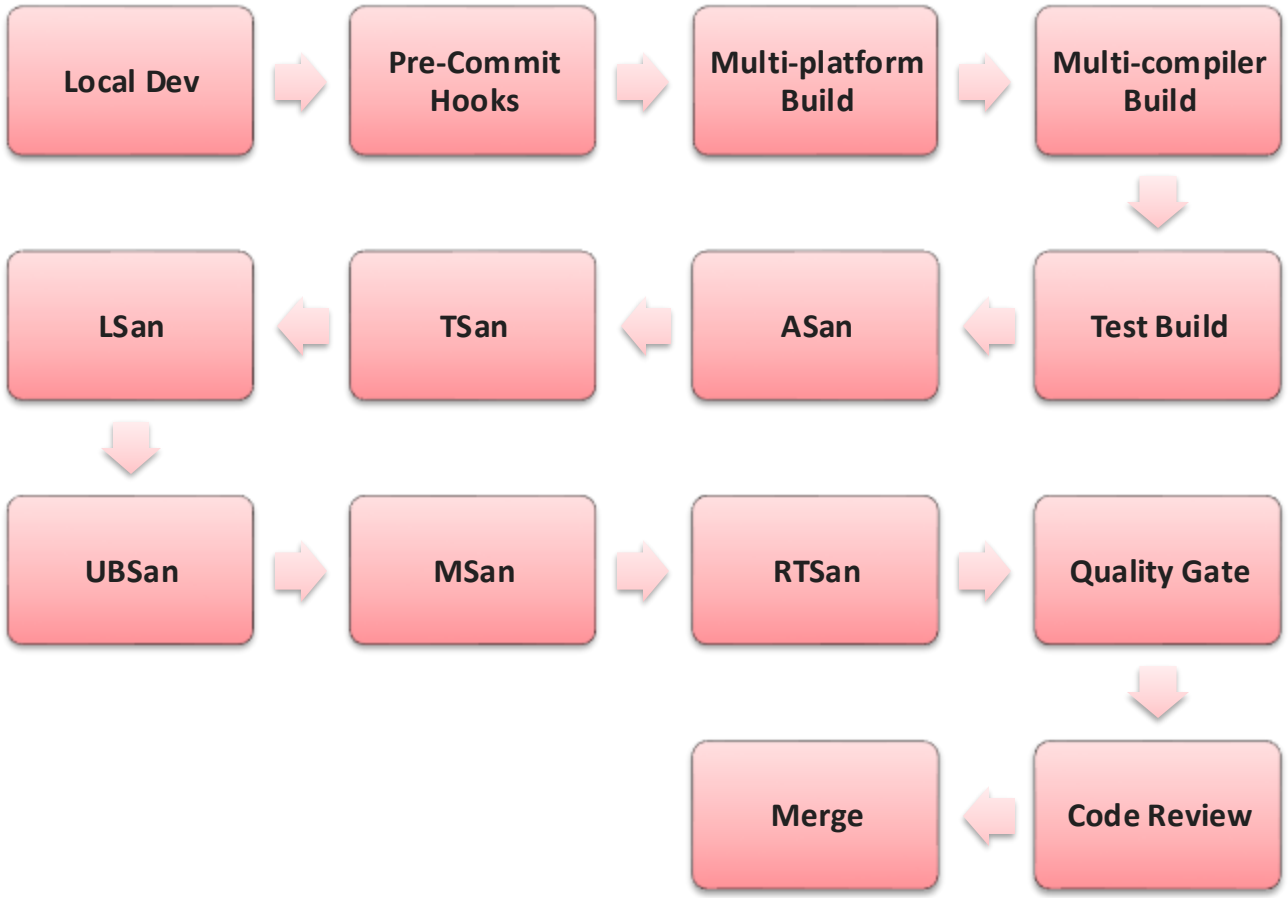
Compiler Flag	Supported Since	Description
`-fsanitize=address`	GCC 4.8.0 Clang 3.1.0	Enables AddressSanitizer to detect memory errors at run-time
`-fsanitize=thread`	GCC 4.8.0 Clang 3.2.0	Enables ThreadSanitizer to detect data race bugs at run time
`-fsanitize=leak`	GCC 4.8.0 Clang 3.1.0	Enables LeakSanitizer to detect memory leaks at run time
`-fsanitize=undefined`	GCC 4.9.0 Clang 3.3.0	Enables UndefinedBehaviorSanitizer to detect undefined behavior at run time
`-fsanitize=realtime`	Clang 20.0.0*	Enables RealtimeSanitizer to detect real time violations at run time

CI/CT/CD

Prefer pre-merge, pre-commit, and local dev before CI: Limited CD in Medical Devices

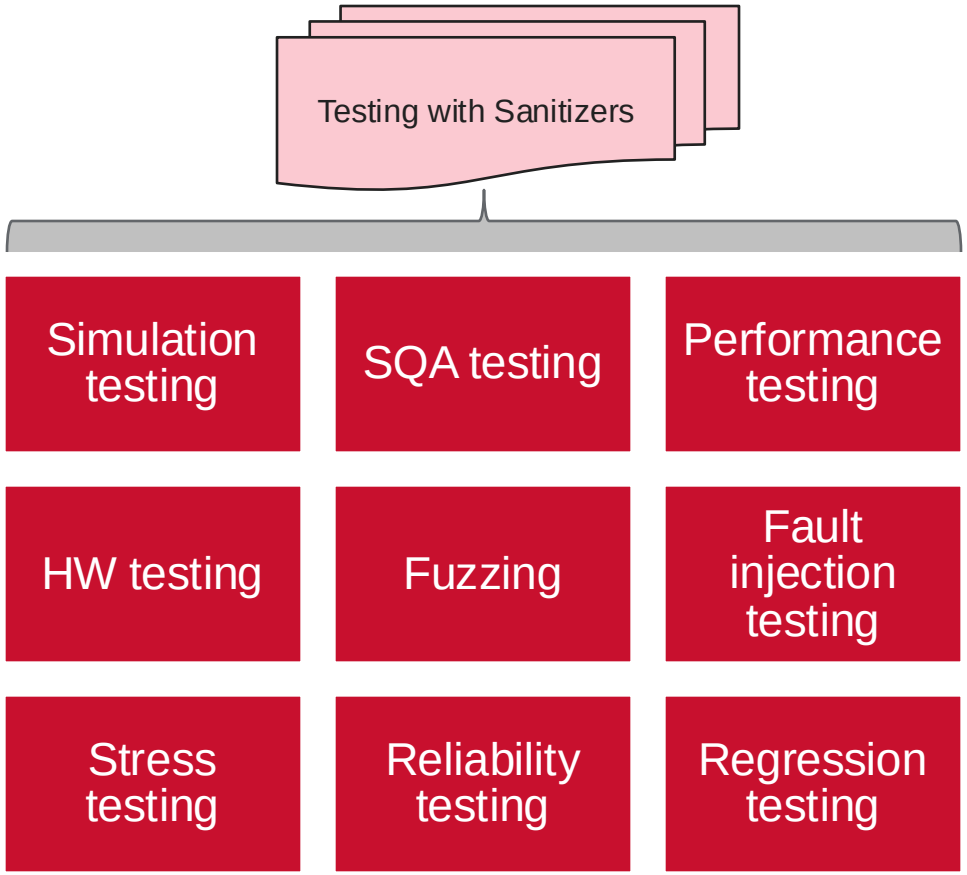


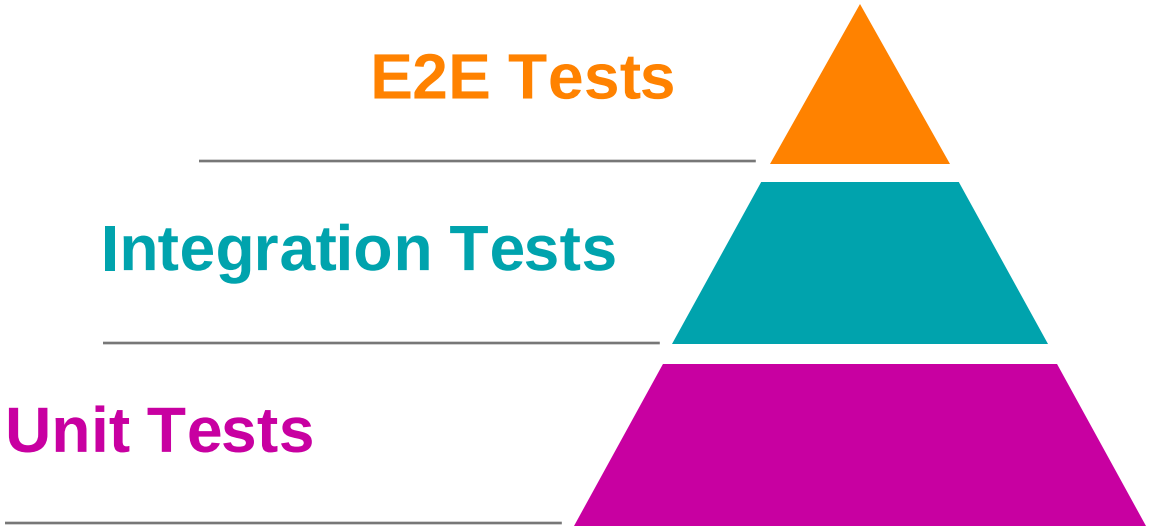
- No new bugs are introduced
- No new vulnerabilities are introduced
- All new security hotspots are reviewed
- New code has limited technical debt
- New code has limited duplication
- New code is properly covered by tests



Continuous Testing

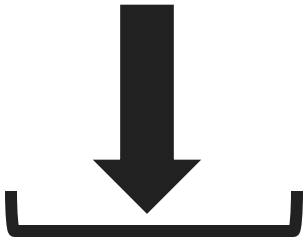
Shift left: Stress test the SW





Frequent Upgrades

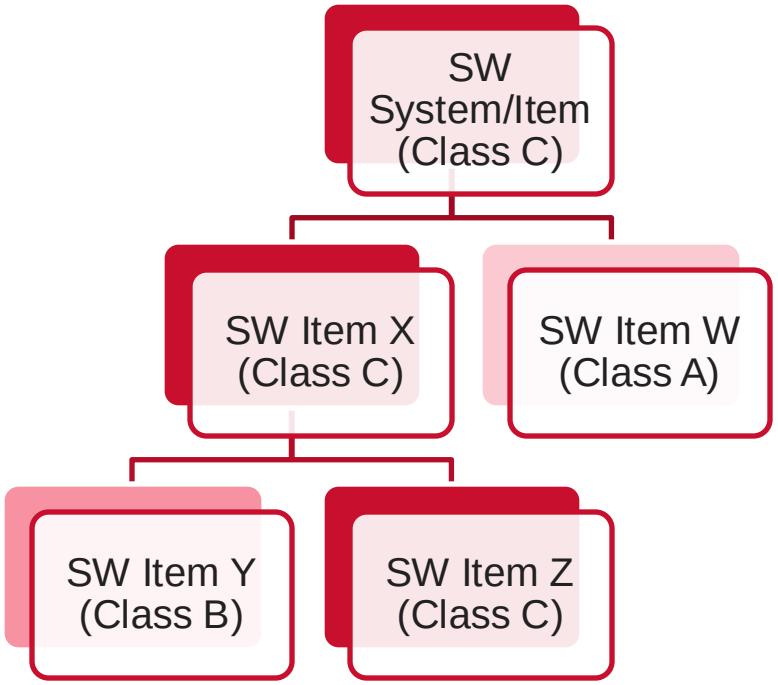
A superpower



- Being enabled to do frequent SW upgrades is a superpower.
- Upgrading SOUP
  - 3rd Party libraries, OS, Middleware, Compiler version, C++ version, ...
- What do we gain:
  - New features, Security++, Bug fixes, Stability and optimizations, A safer C++ version, Enhanced tooling, Compliance and possibly being safety certified (3rd party vendors), ...
- What can help:
  - Modular and decoupled architecture, Frequent tech debt reviews, Security/Compliance tracking, Established testing framework for upgrades, Proper dependency mgmt. (e.g., Conan), solid documentation, ...

Domain Driven Considerations

Safety Critical Path vs Non-Real Time Domain



Non-Real Time
- No strict timing guarantees
- e.g., Logging
- NRTPC

Soft Real Time
- Deadlines are more flexible but still affect performance
- e.g., Vision Overlays
- NRTPC, RTOS

Firm Real Time
- Tiny amount of missed deadlines are ok.
- e.g., UDP comms
- RTOS

Hard Real time
- Deadlines are strict and non-negotiable
- e.g., Control Systems, Vision
- BareMetal, Embedded, RTOS

Domain Driven Considerations

C++ usage in Safety Critical Path

- After system startup and once the robotic instruments are within patient's body, as much as possible, we should avoid:
  - **Unsafe Operations**
    - Possible UB, Pointer Arithmetic, Uninitialized variables, Out of bound read/write, improper casts, nullptr dereference, memory buffer violations, race condition, etc.
  - **Non-deterministic Behavior**
    - Dynamic memory allocation, exception handling
  - **OS Level Blocking Calls**
    - Unnecessary locking, I/O operations

Dynamic Memory Allocation

Dynamic Memory Allocation

The Cost

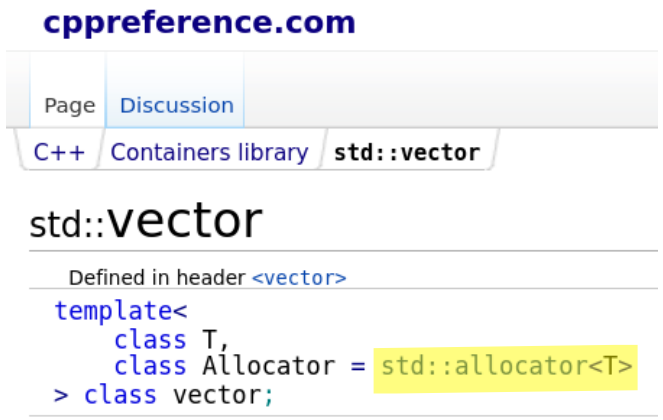
Dynamic Memory Allocation

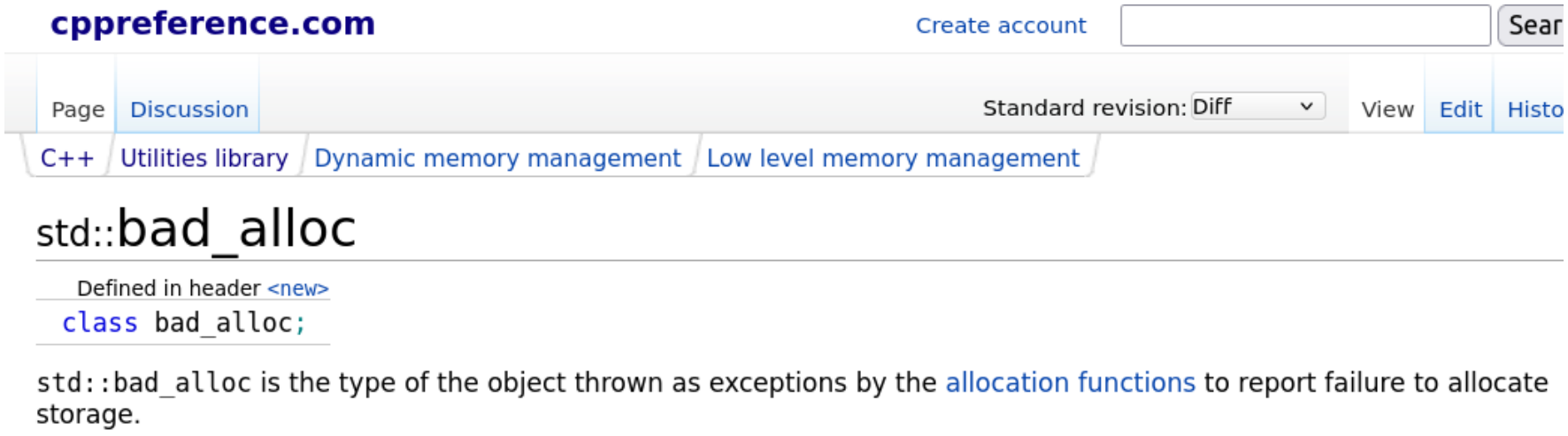
The Cost

Identifying Dynamic Memory Allocation

Dynamic Memory Allocation

Identifying DMA





Dynamic Memory Allocation

Identifying DMA

```
std::vector

std::map

std::set

std::string
```

```
std::unique_ptr

std::shared_ptr
```

```
std::function

std::any
```

Dynamic Memory Allocation

Identifying DMA

```
void Log(const std::string& msg) { /* ... */ }

int main() {
  Log("some static string constant");
}
```

Dynamic Memory Allocation

Identifying DMA

Hoping humans do the right thing is error-prone and almost never works, so leverage tools

- Static assertions
- Static analysis tools
- Dynamic analysis tools
- Unit tests that monitor allocation

Dynamic Memory Allocation

Identifying DMA

How to enforce avoiding DMA in critical sections?

UNIT TEST

Possible to override **`malloc/free`** but not trivial.
LD_PRELOAD your program with a dynamic lib that overrides malloc with `__wrap_malloc`.

```
class HeapMonitor {
   public:
    HeapMonitor() = default;
    static void SetHeapDetected() { is_heap_allocated_ = true; }

    bool IsHeapAllocatedAndReset() {
        bool heap_allocated = is_heap_allocated_;
        is_heap_allocated_ = false;
        return heap_allocated;
    }

   private:
    static bool is_heap_allocated_;
};

bool HeapMonitor::is_heap_allocated_ = false;

void* operator new(std::size_t s) {
    HeapMonitor::SetHeapDetected();
    return malloc(s);
}

void* operator new[](std::size_t s) {
    HeapMonitor::SetHeapDetected();
    return malloc(s);
}

void operator delete(void* p) { free(p); }
void operator delete[](void* p) { free(p); }
```

Avoiding Dynamic Memory Allocation

Dynamic Memory Allocation

Avoiding DMA

```
// Move all allocation to initialization
std::vector<std::unique_ptr<Sensor>> sensors = BuildSensors(configuration);

while (keep_going) {
  for (auto& sensor : sensors) {
    Use(*sensor);
  }
  // ...
}
```

Dynamic Memory Allocation

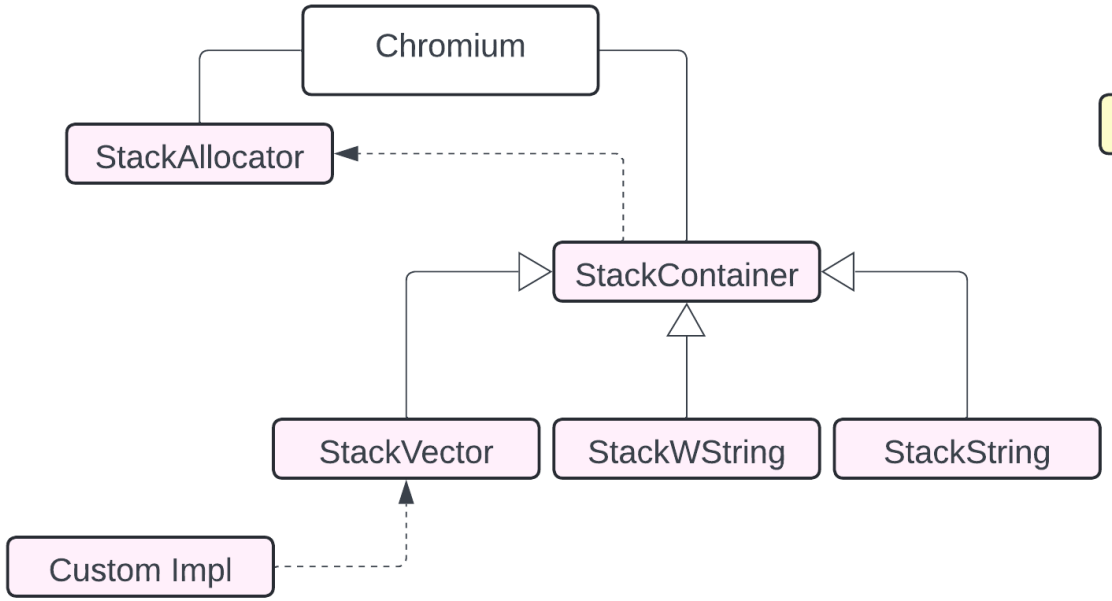
Avoiding DMA

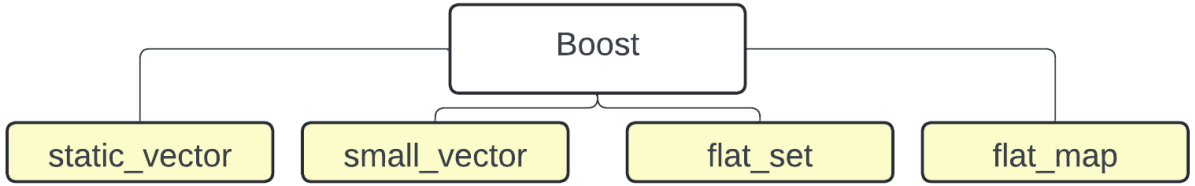
```
std::vector<T, Allocator = std::allocator<T>>
std::map<Key, T, Compare, Allocator>
std::set<Key, Compare, Allocator>
std::basic_string<CharT, Traits, Allocator>
```

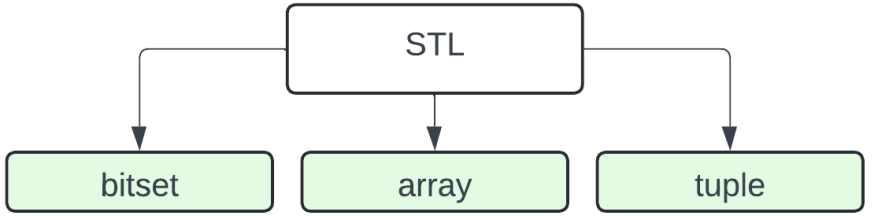
Provide the **Allocator** strategy for pre-allocating memory with a capacity or use off-the-shelf implementations.

Dynamic Memory Allocation

Avoiding DMA







Dynamic Memory Allocation

Avoiding DMA

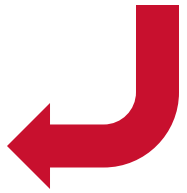
```
int main() {
  std::vector<std::unique_ptr<MyInterface>> workers;
  workers.push_back(
      std::unique_ptr<MyInterface>(new (std::nothrow) SensorReader));
  workers.push_back(
      std::unique_ptr<MyInterface>(new (std::nothrow) GravityStabilizer));

  if (std::any_of(workers.begin(), workers.end(),
                  [](const auto &w) { return w == nullptr; })) {
    std::cerr << "Error\n";
    return 1;
  }

  for (auto &worker : workers) {
    if (worker) {
      worker->DoWork();
    }
  }
}
```

Dynamic Memory Allocation

Avoiding DMA

```cpp
// Keep it generic but move to the stack
int main() {
  std::array<std::variant<SensorReader, GravityStabilizer>, 2> workers{
      SensorReader(), GravityStabilizer()};
  for (auto &worker : workers) {
    std::visit([](auto &w) { w.DoWork(); }, worker);
  }
}
```

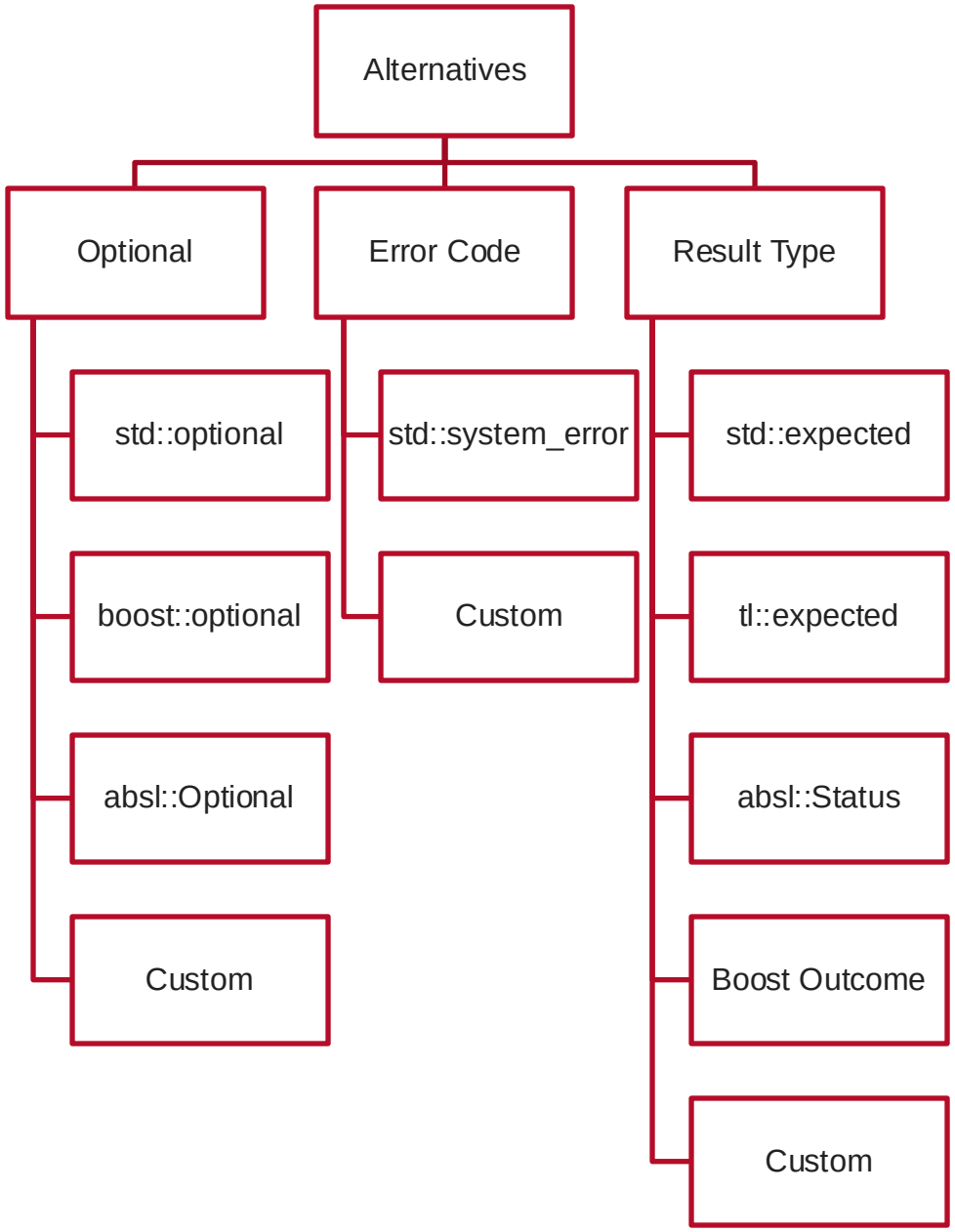
```cpp
// Nice, concrete types and descriptive functions
int main() {
  SensorReader sensor;
  GravityStabilizer stabalizer;
  sensor.Read();
  stabalizer.Apply();
}
```



Exception

Exceptions

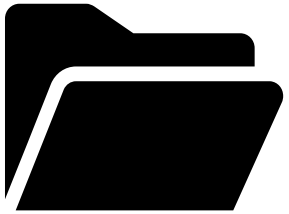
Alternatives



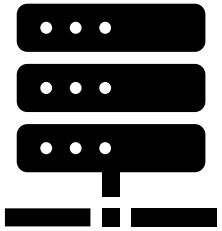
Blocking Calls

Blocking Calls

Identifying Blocking Calls



File IO



Network IO

Locking

Blocking Calls

Identifying Blocking Calls

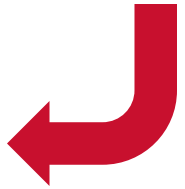
```
while (true) {
  // Waiting to acquire ...
  std::lock_guard lock(m);
  // Processing unknown amount per iteration
  while (!q.empty()) {
    const auto c = q.front();
    Process(c);
    // wait for next iteration ...
  }
}
```

Avoiding Blocking Calls

Blocking Calls

Avoiding Blocking Calls

```
std::mutex m;
std::queue<char> q;
// ...
std::lock_guard lock(m);
if (!q.empty()) { /* ... */ }
```



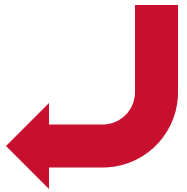
```
boost::lockfree::spsc_queue<MyType> q(42);
// ...
if (q.read_available() > 0) {
  // ...
}
```

Blocking Calls

Avoiding Blocking Calls

```
while (true) {
  w.consume_all(Process);
  // ...
}
```

```
while (true) {
  w.consume_one(Process);
  // ...
}
```



C++ Zero Overhead Abstraction for Safety

cppreference.com Create account

Page Discussion Standard revisior

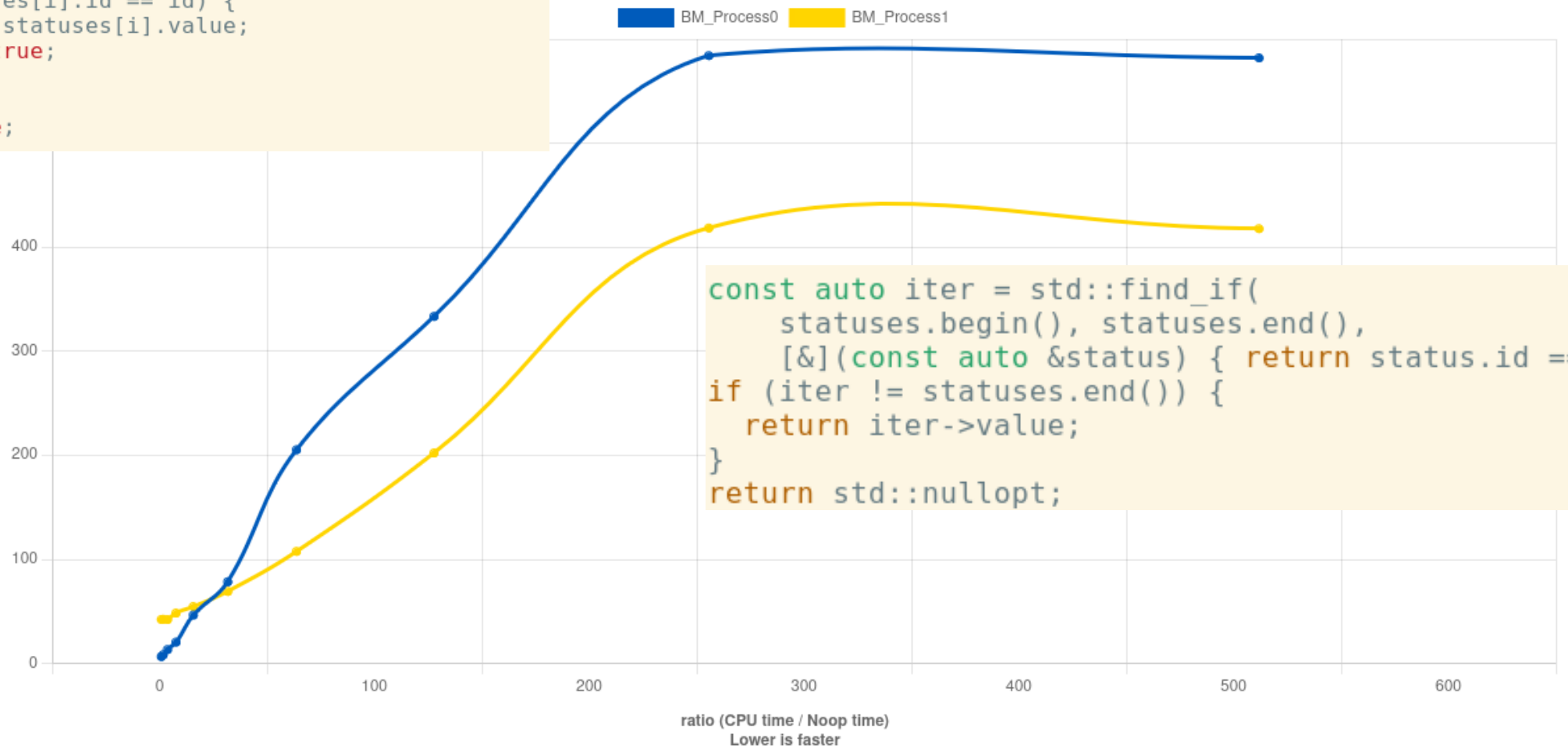
C++ C++ language

Zero-overhead principle

The *zero-overhead principle* is a C++ design principle that states:

1. You don't pay for what you don't use.
2. What you do use is just as efficient as what you could reasonably write by hand.

```cpp
for (std::size_t i = 0; i < statuses.size(); ++i) {
  if (statuses[i].id == id) {
    value = statuses[i].value;
    return true;
  }
}
return false;
```




```cpp
const auto iter = std::find_if(
    statuses.begin(), statuses.end(),
    [&](const auto &status) { return status.id == id; });
if (iter != statuses.end()) {
  return iter->value;
}
return std::nullopt;
```

https://quick-bench.com

Domain Driven Considerations

`unique_ptr` should be the default choice but is not a silver bullet

```cpp
template <class T>
void C_Cpp_Legacy_func(T* ptr) {
    // This function takes a raw ptr w/o ownership transfer
    // Do sth with |ptr|
}

template <class T>
void C_Cpp_3rdParty_Func(T* ptr) {
    // This function takes ownership of the resources of a raw pointer.
    // Do sth with |ptr|
    // ...
    // delete ptr and set to nullptr
    delete ptr;
    ptr = nullptr;
}

int main() {
    auto ptr = std::make_unique<int>(42);
    C_Cpp_Legacy_func(ptr.get());
    C_Cpp_3rdParty_Func(ptr.release());

    return 0;
}
```

Final Words

Trade offs

- Trading off **performance** for **safety**?
  - In many cases but **not always**! Avoiding DMA and exception throwing in the critical path
  - Performance is very subjective. 250 Hz control loop might be fast for some applications and too slow for others!
  - Characterize/measure performance implications!
- Trading off **correctness** for **safety**?
  - Usually, safer code is more correct!
  - Static analysis tools may require writing coding to address the issues, and this potentially introduces more bugs!
  - Dynamic analysis tools could potentially change the code correctness!
  - Degraded performance could mean less correct!

Trade offs

- Trading off **flexibility** for **safety**?
  - Most of the time but not always!
  - Avoiding unsafe constructs could hinder flexibility!
  - Increased use of wrapper types, stricter type checks, and encapsulation can reduce code flexibility!
  - Being more careful in general is less flexible!
- Trading off **cost** for **safety**?
  - All the tooling and processes to mitigate safety issues require resources: Engineering hours and infrastructures.

Takeaways

What have we learned?

- Building safe complex medical robotics is actually very hard
- Standards/regulations are necessary but not sufficient
- Software is a key part of medical device failures
- Testing alone is not sufficient. Fail but fail safely
- Safety in C++ is achieved through strong **culture**, robust **architecture**, effective **tooling**, and rigorous **processes**
- No simple solution at the end of the day but tradeoffs

Opportunities at JnJ MedTech

www.careers.jnj.com

Upcoming Talk: Thursday 2p by **Samuel Privett**

Why is my build so slow? Compilation Profiling and Visualization

Thank you

Questions?