



C++20 for flight software

Wojciech Cierpucha

ADCSS - 2024 N7 Space sp. z o.o.

Project goals

- C++ is an established and mature programming language
- Recent language revisions look very different from C++98
- As of early 2022, no ESA mission took advantage of C++20
- Common fears of using C++ in mission-critical embedded software
 - Implicit code generation
 - Compatibility with ECSS requirements
 - Lack of reusable qualified software
 - Interoperability difficulties with existing software
 - Runtime and object code size overhead

Agenda

- Useful C++ features
- Compiler support
- Coding guidelines
- Additional tooling
- RTOS integration
- C++ Standard Library applicability
- Memory allocation control
- Error handling
- Runtime overhead
- Notable C++20 features

Useful C++ features and their C counterparts

- Component reuse and code generalization
- Metaprogramming and compile-time computation
- Deterministic and reliable resource management in C++
- Conditional compilation and configurability.

Useful C++ features - component reuse and code generalization

C- style

- `void*` casting
- macro substitution

`void *`

`≠`

Polymorphism

C++- style

- Inheritance
- Polymorphism
 - Static polymorphism
 - Templates
 - Overloading
 - CRTP
 - Dynamic polymorphism
 - Overriding

```
void print(int i) { logger << i << "\n"; }
void print(double d) { logger << d << "\n"; }

template <class T> T add(T a, T b) { return a + b; }

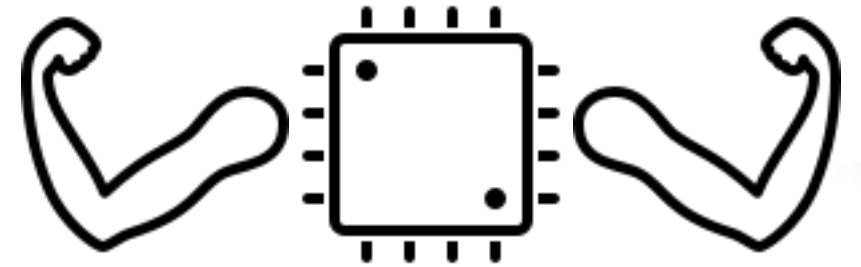
class Complex {
public:
    double r, i;
    Complex operator+(const Complex &c) {
        return {r + c.r, i + c.i};
    }
};

class SyncIndication {
public:
    virtual ~SyncIndication() = default;
    virtual void handle(std::uint8_t count) = 0;
};

class NmtSrvSyncInd : public SyncIndication {
public:
    void handle(std::uint8_t count) override {
        // ...
    }
};
```

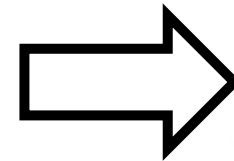
Useful C++ features –compile-time facilities

- Example applications:
 - Precompute CRC16 lookup table
 - Enforce buffer sizes at compile-time
- **templates**
- **constexpr**
- **constexpr** – immediate functions
- **constexpr**



Useful C++ features - Deterministic and reliable resource management in C++

- Most examples use heap memory as a managed resource – familiar to anybody
- In flight software applicable for
 - RTEMS mutexes
 - Opened files
 - Reference-counted datapool objects (e.g. TC/TM stores)
- Resource Acquisition is initialization- RAII
 - Constructor and destructors
 - Management of resource life cycle.
 - Constructor will allocate a resource for you or take ownership of it
 - Destructor will clean a resource for you
 - Copy constructor and copy assignment operator
 - Management of resource copying.
 - Will make a deep copy of the resource if necessary
 - Move constructor and move assignment operator
 - Management of resource ownership



Useful C++ features - Conditional compilation and reconfigurability.

- Pre-processor directives
 - + Can detect toolchain / environment
 - + Can be used to pass compilation configuration
 - - Doesn't follow code indentation
 - - Not all of the code is compiled
 - - Doesn't enforce type safety
- constexpr and constexpr if
 - - can't detect toolchain / environment
 - - can't be used to pass compilation configuration
 - + follows code indentation
 - + all of the code is compiled
 - + enforces static type safety

constexpr > #ifdef
constexpr if > #elif..

```
#ifndef BUF_SIZE // make it possible to pass  
                // compilation configuration  
#define BUF_SIZE 32 // default value  
#endif
```

```
namespace example  
{ // enforce type-safety with constexpr  
  constexpr std::size_t buf_size = BUF_SIZE;  
} // namespace example  
char buf[example::buf_size];
```


Compiler support

Toolchain support

ARM

- Can be created from scratch based on GCC or Clang with GNU or LLVM libs
- ARM Ltd
- Linaro
- Vendor specific e.x.:
 - STM32
 - Microchip

SPARC

- Can be created from scratch based on GCC or Clang with GNU or LLVM libs
- Frontgrade Gaisler
 - BCC2
 - LEON 2-5
 - GNU and LLVM
 - C++11 support
 - Some features of C++ ver. up to 20 should be supported
 - BCC
 - LEON 3,4
 - GNU based

RISC-V

- Can be created from scratch based on GCC or Clang with GNU or LLVM libs
- SiFive
- RISC-V International
- PlatformIO Labs
- Embecosm
- Yocto

C++ Core Guidelines

- Living document, not versioned in practice
- Guidelines Support Library
- Rule investigation and tailoring for space projects
 - Exceptions
 - Dynamic memory allocation
 - GSL use

AUTOSAR C++

- Based on MISRA C++:2008
 - Original is too old to be applicable for modern language revisions
 - Updated for C++14 with some C++17 rules, if available
- Obsolete since 2019
 - But still very useful (until recently)
- High tool coverage
 - Only commercial offerings
- Rule investigation and tailoring for space projects
 - Exceptions
 - Dynamic memory allocation

Minimal C++20 for Flight Software Coding Guidelines

- None of investigated standards were up-to-date
- Common sense rules enforceable using free tools
 - With flight-themed examples
 - With traceability to AUTOSAR C++, MISRA C++, C++ Core Guidelines, SEI CERT C++ rules
 - Enforceable using free tools
 - Annex U rules taken into account
 - Attached **clang-tidy** configuration file
- 43 general guidelines
- 5 guidelines related to dynamic memory allocation

MISRA C++:2023

- Investigation planned, not performed
- Merge of AUTOSAR C++ into MISRA
 - Updated for C++17
- Internally reviewed after available publicly
 - Tooling not there yet
 - Currently adopting it in one of our projects
 - The way to go in the future

Tools

- Most tools that target C++, support C as well
- Some tools can be used on target hardware, others on development machines
 - Additional x86 build for flight software is highly recommended
- Annex U rules can be enforced using free tools from this slide
- Static analysers
 - Compiler warnings (first line of defense)
 - Free linters and path sensitive analysers (*clang-tidy, cplint, clang-sa, cppcheck*)
 - Commercial software
- Dynamic analysers (*sanitizers* and *valgrind*)
 - **Critical** for finding memory correctness and concurrency bugs
- Debuggers
- Profilers (*vTune, WPA, perf_events, optick, valgrind, manual instrumentation* etc.)
 - A convenient profiler for target hardware would be nice...
- Code formatters (*clang-format, uncrustify, artistic style*)
- **#include** analysers (*include-what-you-use, cppinclude, clangd's includecleaner*)
 - Even when modules are stable, header files will stay for decades
- Build systems
- Documentation generators (*Doxygen, clang-doc, Sphinx* etc.)
- Unit testing frameworks (*GoogleTest, CppUTest, Catch2, etc.*)

RTOS compatibility

- Investigation into C++ compability with popular RTOSes
 - Only RTEMS SMP QDP investigated
- Multiple examples produced with a dockerized CMake build-system integrating RTEMS SMP QDP
- No functional compatibility issues discovered – it just works
- Schedulability analysis issues
 - Exceptions
 - RTTI
 - Heap allocation
 - ^ avoid / disable above

C++ support for RTEMS concurrency

- `std::mutex`, `std::condition_variable` can be made to work
 - But require the not-qualified POSIX API
- `std::shared_mutex` and `std::shared_lock` do not work
- Latches and Barriers are not compatible with the Barrier Manager

but

- `std::atomic<>` and the `<atomic>` standard header supported and recommended
- RTEMS mutex and condition variable can be easily wrapped in C++ Standard Library compatible interface
- Other locking RAII types (e.g. `std::unique_lock`) work with C++ wrappers over RTEMS primitives

Standard library implementations

libc++, version 15	libstdc++, version 12.2.0
Full support up to C++14, partial of C++17 and 20	Full support up to C++17, partial of C++20
Documentation: available online and possible to generate on request using CMake	Documentation: available online and possible to generate on request using Doxygen
License: Apache 2	License: GNU General Public License (GPL) v3 with the GCC Runtime Library Exception
Nesting level over 5: 18 offenders	Nesting level over 5: 75 offenders
Cyclomatic complexity over 15: 48 offenders	Cyclomatic complexity over 15: 217 offenders

Standard library test coverage

libc++

Directory	Line Coverage ↕	Functions ↕	Branches ↕
/llvm-project/libcxx/src	74.1 % 4209 / 5677	81.3 % 704 / 866	50.9 % 2146 / 4216
/llvm-project/libcxx/src/experimental	0.0 % 0 / 30	0.0 % 0 / 16	- 0 / 0
/llvm-project/libcxx/src/filesystem	89.5 % 1288 / 1439	96.2 % 177 / 184	64.3 % 909 / 1413
/llvm-project/libcxx/src/include	98.4 % 301 / 306	100.0 % 31 / 31	76.3 % 309 / 405
/llvm-project/libcxx/src/include/ryu	96.8 % 92 / 95	100.0 % 24 / 24	77.1 % 37 / 48
/llvm-project/libcxx/src/ryu	98.8 % 1106 / 1120	100.0 % 29 / 29	98.0 % 494 / 504
/llvm-project/libcxx/src/support/runtime	100.0 % 50 / 50	94.7 % 18 / 19	62.5 % 10 / 16
v1	64.1 % 3463 / 5405	77.4 % 1336 / 1725	38.0 % 2567 / 6748
v1/algorithm	79.6 % 528 / 663	36.8 % 243 / 661	13.3 % 360 / 2708
v1/bit	100.0 % 2 / 2	100.0 % 4 / 4	- 0 / 0
v1/charconv	98.1 % 51 / 52	100.0 % 13 / 13	87.5 % 21 / 24
v1/chrono	95.3 % 41 / 43	100.0 % 60 / 60	50.0 % 4 / 8
v1/debug_utils	100.0 % 2 / 2	6.7 % 1 / 15	- 0 / 0
v1/filesystem	95.5 % 212 / 222	99.1 % 116 / 117	63.1 % 77 / 122
v1/functional	76.2 % 99 / 130	86.5 % 32 / 37	70.0 % 14 / 20
v1/ios	66.7 % 4 / 6	60.0 % 3 / 5	- 0 / 0
v1/iterator	69.2 % 83 / 120	59.9 % 139 / 232	45.0 % 18 / 40
v1/memory	83.5 % 259 / 310	80.9 % 539 / 666	39.7 % 69 / 174
v1/random	0.0 % 0 / 21	0.0 % 0 / 10	0.0 % 0 / 2
v1/string	95.3 % 163 / 171	78.7 % 37 / 47	48.7 % 76 / 156
v1/type_traits	100.0 % 2 / 2	100.0 % 2 / 2	- 0 / 0
v1/utility	85.7 % 24 / 28	67.1 % 192 / 286	- 0 / 0
v1/experimental	0.0 % 0 / 1	0.0 % 0 / 1	- 0 / 0

libstdc++

Directory	Line/State ment coverage	Decision coverage
libstdc++ include/	91.6%	57.6%
libstdc++ include/bits/	88.0%	55.1%
libstdc++ src/	72.1%	50.4%

Dynamic allocation in the Standard Library

- Different parts of the Standard Library have different approaches to allocating memory
 - Some do not allocate at all (`std::array<>`, `std::tuple<>`, `std::variant<>`, `std::optional<>`, `std::atomic<>`)
 - Some may allocate depending on the size of stored data (`std::any`, `std::function<>`, `std::string`)
 - Some data structures allow control (`std::vector<>`, `std::list<>`, `std::string`)
 - Most algorithms do not allocate, but some need to
 - Some parts are of no interest in flight software (IO, regular expressions, OS-based thread or filesystem support)
- Inconsistent approach, requires knowledge/documentation

Custom allocation mechanisms

- User-provided allocators can be passed to some containers
- Emergency pool for exceptions
 - Can be controlled in recent GCC versions
- Overriding **new/delete** operators
 - Useful for logging and tracking unwanted memory operations
- Garbage collection
 - Abandoned
- Recommendations
 - Knowledge is key
 - Avoid using parts with no control

Exceptions vs error codes

- Many approaches to error handling
- Binary size overhead
- Runtime overhead on happy and error paths
- Readability
- Usability
- Default safety
- Composability

Exceptions: recommendations

- Exceptions either disabled or used for critical failures
 - This recommendation is specifically targeting real time software
- Mark reusable public interface as **noexcept**
 - Even with exceptions disabled

Runtime overhead – Language

- Investigation into runtime overhead of C++ constructs over C analogues
- Namespaces: no overhead
- Type conversions:
 - Possible overhead of implicit conversions of large types
 - Overhead in dynamic type conversions (RTTI, `typeid` and `dynamic_cast<>`)
- Function calls: no overhead
- Virtual dispatch:
 - Slight overhead over non-virtual functions, insignificant for non-trivial functions
 - Same as calling through function pointer
- Exceptions: nominal path is zero-cost in most implementations
- Data structures: two implementations of a ring buffer

Runtime overhead – Standard Library

- Tons of standard algorithms in C++ Standard Library, only 2 in C
- Binary search
 - Not much difference, `std::lower_bound` can benefit from inlining the comparator, but there isn't a lot of comparisons - $O(\log_2 N)$
- Sorting
 - `std::sort` is considerably faster than `qsort`
 - But adds a noticeable amount of code for each instantiation

C++20: Coroutines

- Resumable functions
 - Can simplify complex state machines
- Currently expert-friendly
 - Minimal support from the Standard Library
 - Depends on heap allocation (by default)
- Work quite well with RTEMS SMP QDP's GCC

C++20: Modules

- Probably the largest change to the language in years
 - Changes decades-old compilation model
- Isolation from accidental macros
- Better encapsulation built-in
 - Explicit export control
- Accidental ODR bugs are harder to introduce
- Compile time improvement potential is a nice bonus
- Header units as a middle ground
- **import std;** in C++23
- Not ready for use yet
 - Lots of fixes in Clang and GCC since we finished our report
 - Best practices still need to be formulated
 - Header files are not going anywhere

Conclusions

- Newer C++ revisions as viable as older ones
- But some new features do not have complete implementation yet
 - Mainly modules
- Challenges related to ECSS software reuse from the Standard Library
- Compile-time computation can only be covered indirectly
 - Does not produce object code to execute
- Avoid after initialization or always
 - Exceptions
 - RTTI
 - Heap

Where to find

- Available on ESSR
- 2 Documents
- Source code examples SW1
- Clang-tidy configuration DATA1

C++20 for the Flight Software Study Report

CPP-N7S-SR-001 rev. 1.0

C++20 for the Flight Software Coding guidelines

CPP-N7S-CG-001 rev. 1.0

Thank you for your attention



Wojciech Cierpucha
wcierpucha@n7space.com
www.n7space.com

Bonus slides

N7 Space

- Company devoted to space systems software development
- Joint venture between SPACEBEL and N7 Mobile
- Key activities:
 - Critical software development for real-time embedded systems
 - LEON and ARM based systems
 - Software qualification based on ECSS standards
 - Development and applications of MBSE tools
 - Formal architecture and data modelling for flight software
 - Model checking
 - Rich experience in TASTE and Capella ecosystems
- Engineering team with ~10 years of experience in space engineering
- Team size: 25



Example: templates generalization

```
size_t ser(uint8_t *buf, const size_t bufSize,    //
           const size_t offset,                  //
           const void *val, const size_t valSize) //
{
    size_t resultIdx = offset;

    if (bufSize >= offset + valSize)
    {
        memcpy(&buf[offset], val, valSize);
        resultIdx += valSize;
    }
    /*.. else() etc*/
    return resultIdx;
}

/*..*/
constexpr size_t SIZE = 10;
uint8_t buf[SIZE];
float val = 3.14;
```


Example: templates generalization

```
size_t ser(uint8_t *buf, const size_t bufSize,    //
           const size_t offset,                  //
           const void *val, const size_t valSize) // Function doesn't know the type nor its size
{
    size_t resultIdx = offset;

    if (bufSize >= offset + valSize)
    {
        memcpy(&buf[offset], val, valSize);
        resultIdx += valSize;
    }
    /*.. else() etc*/
    return resultIdx;
}

/*..*/
constexpr size_t SIZE = 10;
uint8_t buf[SIZE];
float val = 3.14;
ser(buf, SIZE, 1, &val, sizeof(val)); // User needs to remember to call sizeof or need to know the size
```

Example: templates generalization

```
size_t ser(uint8_t *buf, const size_t bufSize,    // Can we trust that a user will pass proper size values?
           const size_t offset,                //
           const void *val, const size_t valSize) // Function doesn't know the type nor its size
{
    size_t resultIdx = offset;

    if (bufSize >= offset + valSize)
    {
        memcpy(&buf[offset], val, valSize);
        resultIdx += valSize;
    }
    /*.. else() etc*/
    return resultIdx;
}

/*..*/
constexpr size_t SIZE = 10;
uint8_t buf[SIZE];
float val = 3.14;
ser(buf, SIZE, 1, &val, sizeof(val)); // User needs to remember to call sizeof or need to know the size
```

Example: templates generalization

```
template <typename T>
size_t serCpp(span<uint8_t> buf, // Buffer size deduced by span
              const size_t idx, //
              const T val)      // Type deduced by a compiler
{
    constexpr size_t bytesNum = sizeof(T); // Size always calculated by compiler
    size_t resultIdx = idx;
    if (buf.size() >= bytesNum + idx)
    {
        span<uint8_t> destination = buf.subspan(idx);
        const T source = val;
        std::memcpy(destination.data(), &source, bytesNum);
        resultIdx += bytesNum;
    }
    /*.. else() etc*/
    return resultIdx;
}

/*..*/
constexpr size_t SIZE = 10;
uint8_t buf[SIZE];
float val = 3.14;
serCpp(buf, 1, &val); // User don't care about calling sizeof or remembering the size
```

Example: compile time facility

```
template <int N> struct fact
{
    enum
    {
        val = fact<N - 1>::val * N
    };
};
```

```
template <> struct fact<0>
{
    enum
    {
        val = 1
    };
};
```

```
int main()
{
    std::printf("5! = %d\n",
fact<5>::val);
    return 0;
}
```

Example: compile time facility

```
template <int N> struct fact
{
    enum
    {
        val = fact<N - 1>::val * N
    };
};
```

```
template <> struct fact<0>
{
    enum
    {
        val = 1
    };
};
```

```
int main()
{
    std::printf("5! = %d\n",
fact<5>::val);
    return 0;
}
```

```
constexpr int fact(const int v)
{
    int result = 1;
    for (int i = 1; i <= v; ++i)
    {
        result *= i;
    }
    return result;
}
```

```
int main()
{
    int x = 5;
    int f5 = fact(x);
    constexpr int f6 = fact(6);
    std::printf("5! = %d\n", f5);
    std::printf("6! = %d\n", f6);
    return 0;
}
```

Example: compile time facility

```
template <int N> struct fact
{
    enum
    {
        val = fact<N - 1>::val * N
    };
};
```

```
template <> struct fact<0>
{
    enum
    {
        val = 1
    };
};
```

```
int main()
{
    std::printf("5! = %d\n",
fact<5>::val);
    return 0;
}
```

```
constexpr int fact(const int v)
{
    int result = 1;
    for (int i = 1; i <= v; ++i)
    {
        result *= i;
    }
    return result;
}
```

```
int main()
{
    int x = 5;
    int f5 = fact(x);
    constexpr int f6 = fact(6);
    std::printf("5! = %d\n", f5);
    std::printf("6! = %d\n", f6);
    return 0;
}
```

```
constexpr int fact(const int v)
{
    int result = 1;
    for (int i = 1; i <= v; ++i)
    {
        result *= i;
    }
    return result;
}
```

```
int main()
{
    // int x = 5;
    // int f5 = fact(x);
    constexpr int f6 = fact(6);
    // std::printf("5! = %d\n", f5);
    std::printf("6! = %d\n", f6);
    return 0;
}
```

Example: CRTP

```
template <typename Derived> class Animal
{
public:
    void says() { static_cast<Derived *>(this)->saysImpl(); }
    void saysImpl() { cout << "Base implementation\n"; }
};

class Fox : public Animal<Fox>
{
public:
    void saysImpl() { cout << "Wa-pa-pa-pow\n"; }
};
```

Example: Static init order fiasco

```
// Fiasco
// File1.cpp
int get_value() { return 42; }

int globVar1 = get_value();

// File2.cpp
extern int globVar1;
int globVar2 = globVar1 + 1;

int main()
{
    cout << "globVar2: " << globVar2 << "\n";
    // May not be 43
    return 0;
}
```

```
// Constinit solution
// File1.cpp
constexpr int get_value() { return 42; }
extern constinit const int globVar1
    = get_value();

// File2.cpp
extern constinit const int globVar1;
int globVar2 = globVar1 + 1;

int main()
{
    cout << "globVar2: " << globVar2 << "\n";
    // Guaranteed to be 43
    return 0;
}
```


Example: custom allocators for Standard Library containers

```
template <class T>
class simple_allocator {
public:
    using value_type = T;

    simple_allocator() = default;
    template <class U>
    simple_allocator(const simple_allocator<U> &) noexcept;

    [[nodiscard]] T *allocate(const std::size_t n);
    void deallocate(T *, const std::size_t n) const noexcept;

private:
    static T static_memory_pool[SIMPLE_ALLOC_POOL_SIZE];
    static std::size_t m_pool_counter;
};

template <class T, class U>
bool operator==(const simple_allocator<T> &,
                const simple_allocator<U> &) noexcept;
template <class T, class U>
bool operator!=(const simple_allocator<T> &,
                const simple_allocator<U> &) noexcept;
```

```
std::vector<int, simple_allocator<int>> vec{1, 2, 3,
4, 5};
```

```
vec.push_back(6);
vec.push_back(7);
vec.push_back(8);
vec.clear();
```

```
static char buffer[BUF_SIZE];
std::pmr::monotonic_buffer_resource alloc{buffer, BUF_SIZE};
```

```
std::pmr::pool_options options;
options.max_blocks_per_chunk = 32;
options.largest_required_pool_block = 512;
std::pmr::unsynchronized_pool_resource pool{options, &alloc};
```

```
std::pmr::vector<int> vec{&pool};
```

```
vec.push_back(6);
vec.push_back(7);
vec.push_back(8);
```

Example: C++ support for RTEMS concurrency

```
class RtemsMutex {
public:
    RtemsMutex(rtems_name name);
    ~RtemsMutex();
    // not copyable, not moveable
    RtemsMutex(const RtemsMutex &) = delete;
    RtemsMutex &operator=(const RtemsMutex &) = delete;
    const rtems_id &getId() const { return m_id; }
    void obtain();
    void release();
    // meet BasicLockable requirements
    void lock() { obtain(); }
    void unlock() { release(); }
    // meet Lockable requirements
    bool try_lock();
private:
    rtems_id m_id;
};
```

```
std::deque<int> sharedResource;
RtemsMutex mutex{rtems_build_name('M', 'T', 'X', '0')};

void produce(const int val) {
    std::scoped_lock lock(mutex);
    sharedResource.push_front(val);
} // unlock mutex in destructor

int consume() {
    std::scoped_lock lock(mutex);
    if (!m_sharedResource.empty()) {
        const int val = sharedResource.back();
        sharedResource.pop_back();
        return val;
    }
    return -1;
} // unlock mutex in destructor
```

Example: C++ and C ring buffer interface

```
template <typename T, std::size_t CAPACITY>
class rb_queue {
    std::array<T, CAPACITY> buffer;
public:
    using value_type = T;
    using size_type = std::size_t;
    /* Construction/destruction */
    rb_queue() = default;
    ~rb_queue() = default;
    /* Element access */
    T front() const;
    T back() const;
    /* Capacity */
    size_type capacity() const;
    size_type size() const;
    bool full() const;
    bool empty() const;
    /* Modifiers */
    void push(const T &value);
    void pop();
};
```

```
typedef int rb_queue_value_type;
/* Construction/destruction */
void rb_queue_init(rb_queue *q, rb_queue_value_type
                  buffer[], size_t capacity);
void rb_queue_destroy(rb_queue *q);
/* Element access */
rb_queue_value_type rb_queue_front(const rb_queue *const q);
rb_queue_value_type rb_queue_back(const rb_queue *const q);
/* Capacity */
bool rb_queue_empty(const rb_queue *q);
bool rb_queue_full(const rb_queue *q);
size_t rb_queue_capacity(const rb_queue *q);
size_t rb_queue_size(const rb_queue *const q);
/* Modifiers */
void rb_queue_push(rb_queue *q, rb_queue_value_type val);
void rb_queue_pop(rb_queue *q);
```

Guidelines: CISQ ISO/IEC 5055:2021

- Code Quality Standards
- ~140 rules based on CWEs
- Measures software quality
- 4 main pillars:
 - Reliability
 - Security
 - Performance efficiency
 - Maintainability
- Reviewed all rules and proposed alternative tailoring