Parallelism Safety-Critical Guidelines for C++

MICHAEL WONG, ANDREAS WEIS,
ILYA BURYLOV& CHRISTOF MEERWALD

Staff Engineer at Woven Planet

Andreas Weis (he/him)

/ ComicSansMS

@DerGhulbus

Co-organizer of the Munich C++ User Group (MUC++)

Member of WG21 (ISO C++) and MISRA C++

Working on the Runtime framework for the Arene platform at Woven Planet

woven planet

Ilya Burylov

Principle Engineer

An architect of C++ software solutions for autonomous driving market

Contribution into functional safety MISRA standard

Contribution into WG21 in threading, vectorization and numerics.

Contribution into SYCL

Distinguished Engineer

- Chair of SYCL Heterogeneous Programming Language
- RISC-V Datacenter/CLoud COmputign Chair
- ISO C++ Directions Group past Chair
- Past CEO OpenMP
- ISOCPP.org Director, VP
  http://isocpp.org/wiki/faq/wg21#michael-wong
- michael@codeplay.com
- fraggamuffin@gmail.com
- Head of Delegation for C++ Standard for Canada
- Chair of Programming Languages for Standards Council of Canada
  Chair of WG21 SG19 Machine Learning
  Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
- Editor: C++ SG5 Transactional Memory Technical Specification
- Editor: C++ SG1 Concurrency Technical Specification
- MISRA C++ and AUTOSAR
- Chair of Standards Council Canada TC22/SC32 Electrical and electronic components (SOTIF)
- Chair of UL4600 Object Tracking
- http://wongmichael.com/about
- C++11 book in Chinese: https://www.amazon.cn/dp/B00ETOV2OQ

Michael Wong

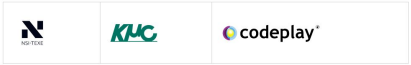
Argonne and Oak Ridge National Laboratories Award Codeplay® Software to Further Strengthen SYCL™ Support Extending the Open Standard Software for AMD GPUs

17 June 2021

LEMONT, IL, and OAK RIDGE, TN, and EDINBURGH, UK, June 17, 2021 - Argonne National Laboratory (ANL) in collaboration with Oak Ridge National Laboratory (ORNL), has awarded Codeplay a contract implementing the oneAPI DPC++ compiler, an implementation of the SYCL open standard software, to support AMD GPU-based high-performance compute (HPC) supercomputers

NSITEXE, Kyoto Microcomputer and Codeplay Software are bringing open standards programming to RISC-V Vector processor for HPC and AI systems

29 October 2020



Implementing OpenCL™ and SYCL™ for the popular RISC-V processors will make it easier to port existing HPC and AI software for embedded systems

NERSC, ALCF, Codeplay Partner on SYCL for Next-generation Supercomputers

02 February 2021

The **National Energy Research Scientific Computing Center** (NERSC) at **Lawrence Berkeley National Laboratory** (Berkeley Lab), in collaboration with the **Argonne Leadership Computing Facility** (ALCF) at Argonne National Laboratory, has signed a contract with **Codeplay Software** to enhance the LLVM SYCL™ GPU compiler capabilities for NVIDIA® A100 GPUs.

We build GPU compilers for some of the most powerful supercomputers in the world

Senior Software Engineer
at Edison Design Group

Christof Meerwald

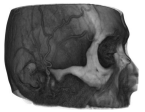
C++ Compiler Front End Development

Member of WG21 (ISO C++ - Core Working Group)

Agenda

- Adding safety to parallelism for both MISRA and C++ CG
  - This year: focus on what we intend to do for C++CG by hazards
- Deep dive to C++CG rules
  - Deadlocks and rejected rules
  - Lifetime violation and new/modified rules
- CG+MISRA: the close to ideal safety partners
  - Ongoing analysis of future C++ parallelism for safety

Safety Critical API Evolution



New Generation Safety Critical APIs for Graphics, Compute and Display

minimize API surface area , reduce ambiguity, UB, increase determinism

Industry Need
for CPU/GPU Acceleration APIs designed to ease system safety certification

Rendering Compute Display

RISC-V

SCSC FOR EVERYONE WORKING IN SYSTEM SAFETY WWW.SCSC.UK

ISO International Organization for Standardization

UNECE WP.29

C++

SYCL

ISO 26262

ISO/PAS 21448

UL 4600

MISRA

SAE INTERNATIONAL

ISO/IEC JTC 1/SC 42
Artificial intelligence

Comparing coding standards

Coding Standard	**C++ Versions**
Autosar	C++14 dormant
Misra	C++03 (working to C++17)
High Integrity CPP	C++11 dormant
JSF	C++03 dormant
C++ CG	C++11/14/17/20/latest
CERT C++	C++14 dormant

MISRAC++ParallelConcurrencyHeteroRulesOverview

File Edit View Insert Format Tools Add-ons Help Last edit was made seconds ago by Michael Wong

Share

100% Normal text Arial 27.9

Outline

Note: this is an early draft WIP. It's known to be incomplet and incorrekt, and it has lots of badformatting.

Table of Content

Table of contents

Heading numbers format 1.2.3

Display until level 6

Stage 1: extensive deep analysis of 81 rules

- Started in 2019 at a MISRA meeting
  - Why are there no rules for parallelism in MISRA?
- 2019-2021: Phase 1 complete
  - Reviewed 81 rules pulled from
    - C++CG
    - HIC++
    - REphrase H2020 project
    - CERT C++
    - JSF++ (no parallel rules)
    - WG23 (no parallel rules)
    - Added some from our own contributions
- Many joined, average 5-8 per meeting
  - Also consulted outside concurrency and safety experts
- Shared Drive of Phase 1 analysis:
  - https://docs.google.com/document/d/14E0BYqsH_d7fMKvXvaZWoNWtIC65cYBw0aZp4dlev0Q/edit#heading=h.yt0hxah53p9e

MISRAC++ParallelConcurrencyHeteroRulesStatusPhase1

File Edit View Insert Format Data Tools Add-ons Help Last edit was 3 days ago

Share

A6: 0.3.6 [5] Do not use std::thread

	A	B	C	D	E	F	G
1	Rule	Category	decidable via human review	decidable via tools	status	Destination: Tools vs C++ Core guideline	Reason for keeping and
2	0.2.2 [2] Do not use platform specific multi-threading facilities	advisory	easy	partially	consider later		only partially detectable, e.g
3	0.3.1 0.3.x [82] Make std::threads unjoinable on all paths	advisory	complex	yes, on system level	consider later		use [7] instead
4	0.3.4 [3] A thread shall not access objects whose lifetime has expired	required	complex	partially	accept for initial revision	CP23 is high level vs more specific	it may exclude certain techr
5	0.3.5 [4] Thread callable object may receive only global or static objects via pointer or	?mandatory	easy	partially	consider later		complex behavior of detach
6	0.3.6 [5] Do not use std::thread	advisory	easy	yes, on local level	accept for initial revision	CP 25 is different	straightforward and decidab
7	0.3.8 [7] Do not call std::thread::detach() function? Join on all Available exit paths	required	easy	yes, on local level	accept for initial revision	CP 26	better than [82] in decidabili
8	0.3.9 [8] Verify resource management assumptions of std::thread with the implementat	directive	complex	no	consider later		directive - keep directive for
9	0.4.1 [9] Do not call member functions of std::mutex, std::timed_mutex, std::recursive_	required	easy	yes, on local level	accept for initial revision	CP 20 is close, we are a little more specific but it is t	straightforward and decidab
10	0.4.3 [11] Use std::lock(), std::try_lock() or std::scoped_lock to acquire multiple mutex	required	easy	yes, on local level	accept for initial revision	based on CP21	straightforward and decidab
11	0.4.4 [12] Do not destroy objects of the following types std::mutex, std::timed_mutex, s	mandatory	complex	yes, on system level	accept for initial revision	NO CG, but not good for CG as it is a clear error, no	clear UB related
12	0.4.5 [13] Mutexes locked with std::lock or std::try_lock shall be wrapped with std::loc	required	easy	yes, on local level	accept for initial revision	NO CG, might be good for CG	straightforward and decidab
13	0.4.6 [14] Do not call virtual functions and callable objects passed by argument of the	advisory	complex	yes, on system level	consider later		
14	0.4.8 [16] Objects of std::lock_guards, std::unique_locks, std::shared_lock and std::sc	required	easy	yes, on local level	accept for initial revision	based on CP24 add shared_lock	straightforward and decidab
15	0.4.9 [17] Define a mutex together with the data it guards. Use synchronized_value<T>	directive	complex	no	consider later		related API is not yet confir
16	0.4.11 [19] There shall be no code path which results in locking of the non-recursive m	mandatory	complex	yes, on system level	accept for initial revision	no CG, but hard for human, so nard for CG	clear UB related
17	0.4.12 [20] The order of nested locks unlock shall form a DAG	required	complex	yes, on system level	accept for initial revision	no CG, but hard for human, so hard for CG	should enspire tools detecti
18	0.4.13 [21] std::recursive_mutex and std::recursive_timed_mutex should not be used	advisory	easy	yes, on local level	accept for initial revision	good for CG, good for MISRA	a sign of too complex soluti
19	0.4.14 [22] There should be a code path, where at least one member functions is calle	advisory	easy	yes, on system level	drop		not a safety concern
20	0.5.1 [23] std::condition_variable::wait, std::condition_variable::wait_for, std::conditio	required	easy	yes, on local level	accept for initial revision	already in CG	straightforward and decidab
21	0.5.3 [25] std::conditional_variable::notify_one() can be used if all threads must perfor	m the same set of oper	?	?	consider later		
22	0.5.4 [26] Do not use std::condition_variable_any on a std::mutex	advisory	easy	yes, on local level	accept for initial revision	good for CG/tools	straightforward and decidab
23	0.6.1 [27]Use only std::memory_order_seq_cst for atomic operations	required	easy	yes, on local level	accept for initial revision	good for CG/tools, more specific then CGF dont use	straightforward and decidab
24	0.7.1 [28] Use a future to return a value from a concurrent task		?	?	drop		hardly formalizable
25	0.7.2 [29] Use an async() to spawn a concurrent task		?	?	drop		to be replace with [5]
26	0.8.1 [30] Don't try to use volatile for synchronization		?	?	drop		to be replace with [32]
27	0.8.2 [31] Use volatile only to talk to non-C++ memory		?	?	drop		should not be in scope of pa
28	0.8.3 [32] Volatile variables shall not be accessed from different threads.	required	complex	may be, on system level	accept for initial revision	good for tools, meta for CG CP8	should enspire tools detecti
29	0.9.1 [33] Bit-fields of the same object, which are not separated by not-bit-field or zero	required	complex	may be, on system level	consider later		very small use case
30	0.9.2 [34] Synchronize access to data shared between threads using a single lock	advisory	complex	may be, on system level	consider later		not perfectly formalizable

Лист1

Rule decidability

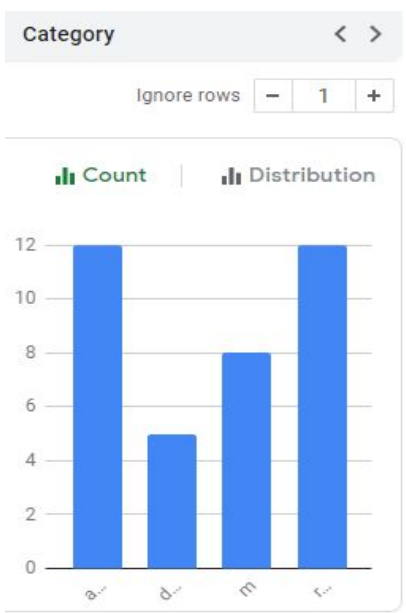
- Human review
  - Generally simple rules
  - Code snippets
  - Basic syntax matches intention
- Automated tool
  - Static scope: can be convoluted but doable and simple for this generation of tools
  - Dynamic scope: much more complex, hard even for tools of this generation, may be doable with whole program analysis
    - Intention is hidden
- Both Human and Automated tools
  - Generally simple cases
  - Intention is shown in syntax
- Neither are good
  - Very hard cases, dynamic scope, whole program analysis
  - Intention is not clear
    - In these cases we wonder if an [[intention:]] attribute might help

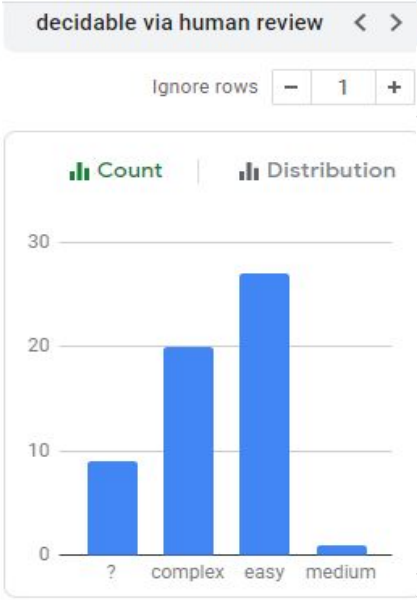
Where should parallel/concurrency/hetero rules go?

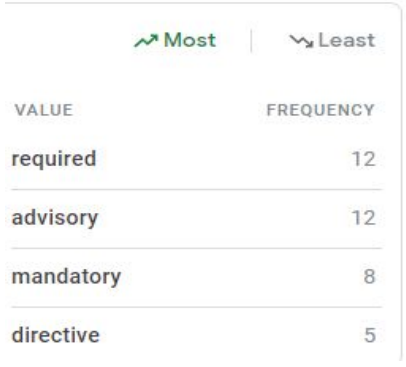
Human decidable	**Tool decidable**	**Suitable tools in order of preference**
Easy	Easy	C++CG, MISRA tools
Easy	Hard	C++CG, Tools will be meta or undecidable, lots of false positive May be bad rule for tools
Hard	Easy	MISRA tools, CG Meta
Hard	Hard	Neither, META directive; Code guidelines Obvious rules, but hard to verify Might not be a good rule anyway Need a new [[intention::] attribute

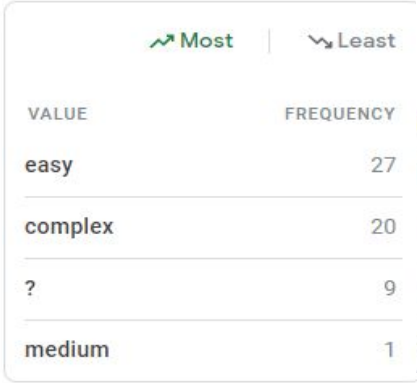
Stage 2: collate

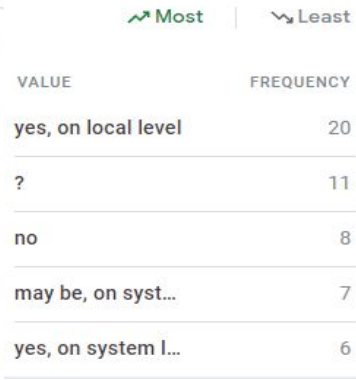
- Category
  - Mandatory: 8
  - Required: 12
  - Advisory: 12
  - Directive: 5
- Decidable by humans
  - Easy: 27
  - Medium: 1
  - Complex: 20
  - Unknown yet: 9
- Decidable via automated tools
  - Yes, on a local level: 20
  - Yes, on a system level: 6
  - Maybe, on a system level: 7
  - No: 8
  - Unknown yet: 11





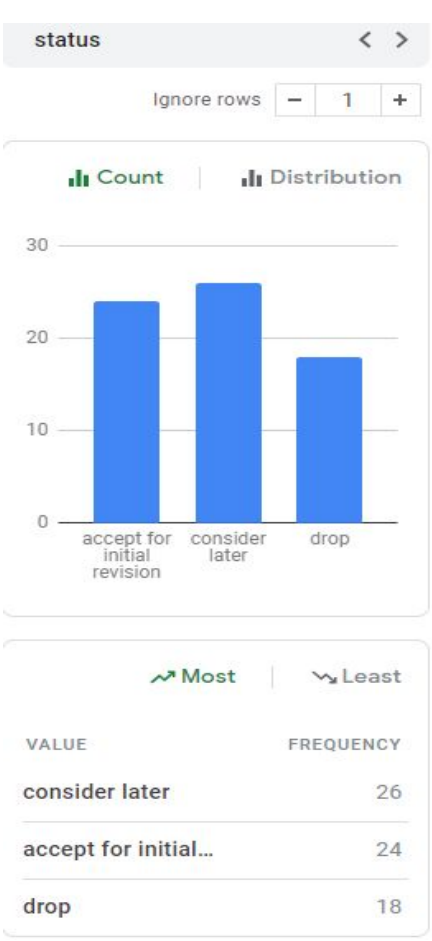






CG, Misra, both or neither

- Accepted: for initial entry 24
  - CG+tools: 12
  - Tools+CG: 5
  - Modifies CG: 4
  - Same as CG: 3
- Deferred for future: 26
- Rejected: 18
- Shared drive of Status from Phase 1:
  - https://docs.google.com/spreadsheets/d/1f-NX2z6axlyv5P0mh4aeNfKO7KLSVSTtZrTwS2YO02M/edit#gid=0



More focus on CG contribution in 2022

- We had 24 rules ready for MISRA and C+CG in 2021
- C++ CG is thin on parallel, only 4 sections have about 36 rules, rest ??? placeholders for future
  - Mostly contributed by Bjarne/Herb
  - We aim to contribute new, or modify, or add to ?? in these section ongoing basis
  - Many parallelism experts, but only very few of those are also safety critical/guideline experts
  - We aim to grow both CG and Misra

C++CG parallel

Concurrency and parallelism rule summary:

- CP.1: Assume that your code will run as part of a multi-threaded program
- CP.2: Avoid data races
- CP.3: Minimize explicit sharing of writable data
- CP.4: Think in terms of tasks, rather than threads
- CP.8: Don't try to use `volatile` for synchronization
- CP.9: Whenever feasible use tools to validate your concurrent code

See also:

- CP.con: Concurrency
- CP.coro: Coroutines
- CP.par: Parallelism
- CP.mess: Message passing
- CP.vec: Vectorization
- CP.free: Lock-free programming
- CP.etc: Etc. concurrency rules

CP.coro: Coroutines

This section focuses on uses of coroutines.

Coroutine rule summary:

- CP.51: Do not use capturing lambdas that are coroutines
- CP.52: Do not hold locks or other synchronization primitives across suspension points
- CP.53: Parameters to coroutines should not be passed by reference

CP.par: Parallelism

By "parallelism" we refer to performing a task (more or less) simultaneously ("in parallel with") on many data items.

Parallelism rule summary:

- ???
- ???
- Where appropriate, prefer the standard-library parallel algorithms
- Use algorithms that are designed for parallelism, not algorithms with unnecessary dependency on linear evaluation

Concurrency rule summary:

- CP.20: Use RAII, never plain `lock()`/`unlock()`
- CP.21: Use `std::lock()` or `std::scoped_lock` to acquire multiple `mutex`es
- CP.22: Never call unknown code while holding a lock (e.g., a callback)
- CP.23: Think of a joining `thread` as a scoped container
- CP.24: Think of a `thread` as a global container
- CP.25: Prefer `gsl::joining_thread` over `std::thread`
- CP.26: Don't `detach()` a thread
- CP.31: Pass small amounts of data between threads by value, rather than by reference or pointer
- CP.32: To share ownership between unrelated `thread`s use `shared_ptr`
- CP.40: Minimize context switching
- CP.41: Minimize thread creation and destruction
- CP.42: Don't `wait` without a condition
- CP.43: Minimize time spent in a critical section
- CP.44: Remember to name your `lock_guard`s and `unique_lock`s
- CP.50: Define a `mutex` together with the data it guards. Use `synchronized_value<T>` where possible
- ??? when to use a spinlock
- ??? when to use `try_lock()`
- ??? when to prefer `lock_guard` over `unique_lock`
- ??? Time multiplexing
- ??? when/how to use `new thread`

Vectorization rule summary:

- ???
- ???

Message passing rules summary:

- CP.60: Use a `future` to return a value from a concurrent task
- CP.61: Use `async()` to spawn concurrent tasks
- message queues
- messaging libraries

Lock-free programming rule summary:

- CP.100: Don't use lock-free programming unless you absolutely have to
- CP.101: Distrust your hardware/compiler combination
- CP.102: Carefully study the literature
- how/when to use atomics
- avoid starvation
- use a lock-free data structure rather than hand-crafting specific lock-free access
- CP.110: Do not write your own double-checked locking for initialization
- CP.111: Use a conventional pattern if you really need double-checked locking
- how/when to compare and swap

CP.etc: Etc. concurrency rules

These rules defy simple categorization:

- CP.200: Use `volatile` only to talk to non-C++ memory
- CP.201: ??? Signals

C++CG parallelism rules By hazard

Row Labels	Count of hazards
Deadlocks	4
Elegance guidelines/bad design/ excessive complicated design	2
Expert Only	1
Lifetime violations	4
REJECTED for CG Undefined Behaviour	1
undefined behavior DEADLOCKS	1
(blank)	
Grand Total	**13**

Of the 24 accepted initially, 18 for CG

	Rule	Category	decidable via human re	decidable via tools	status	hazards	Destination: To	Reason
2	***0.3.4 [3] A thread shall not access objects whose lifetime has expire***	required	complex	partially	accept for initial revision	Lifetime violations	Modifies CG	CP23 is high leve
3	***0.3.6 [5] Do not use std::thread***	advisory	easy	yes, on local level	accept for initial revision	Lifetime violations	Modifies CG	CP 25 is different
4	***0.3.8 [7] Do not call std::thread::detach() function? Join on all Availa***	required	easy	yes, on local level	accept for initial revision	Lifetime violations	Same as CG	CP 26
5	***0.4.1 [9] Do not call member functions of std::mutex, std::timed_mut***	required	easy	yes, on local level	accept for initial revision	Deadlocks	Modifies CG	CP 20 is close, w
6	***0.4.3 [11] Use std::lock(), std::try_lock() or std::scoped_lock to acqu***	required	easy	yes, on local level	accept for initial revision	Deadlocks	Same as CG	based on CP21
7	***0.4.4 [12] Do not destroy objects of the following types std::mutex, s***	mandatory	complex	yes, on system level	accept for initial revision	REJECTED for CG Un	Tools+CG	NO CG, but not g
8	***0.4.5 [13] Mutexes locked with std::lock or std::try_lock shall be wra***	required	easy	yes, on local level	accept for initial revision	Deadlocks	CG+Tools	NO CG, might be
9	***0.4.8 [16] Objects of std::lock_guards, std::unique_locks, std::share***	required	easy	yes, on local level	accept for initial revision	Lifetime violations	Modifies CG	based on CP24 a
10	***0.4.11 [19] There shall be no code path which results in locking of th***	mandatory	complex	yes, on system level	accept for initial revision	undefined behavior D	Tools+CG	no CG, but hard f
11	***0.4.12 [20] The order of nested locks unlock shall form a DAG***	required	complex	yes, on system level	accept for initial revision	Deadlocks	Tools+CG	no CG, but hard f
12	***0.4.13 [21] std::recursive_mutex and std::recursive_timed_mutex sh***	advisory	easy	yes, on local level	accept for initial revision	Elegance guidelines/ba	CG+Tools	good for CG, goo
13	***0.5.1 [23] std::condition_variable::wait, std::condition_variable::wait***	required	easy	yes, on local level	accept for initial revision		Same as CG	already in CG
14	***0.5.4 [26] Do not use std::condition_variable_any on a std::mutex***	advisory	easy	yes, on local level	accept for initial revision	Elegance guidelines/ba	CG+Tools	good for CG/tools
15	***0.6.1 [27]Use only std::memory_order_seq_cst for atomic operations***	required	easy	yes, on local level	accept for initial revision	Expert Only	CG+Tools	good for CG/tools
16	**0.8.3 [32] Non-Atomic Volatile variables shall not be accessed from c**	required	complex	may be, on system level	accept for initial revision		Tools+CG	good for tools, me
17	***0.10.5 [39] Use std::call_once to ensure a function is called exactly c***	advisory	easy	may be, on system level	accept for initial revision		CG+Tools	new for CG? CP1
18	***0.12.4 [51] Always explicitly specify a launch policy for std::async***	required	easy	yes, on local level	accept for initial revision		CG+Tools	cg, tools
19	***0.12.5 [52] Access to mutable members shall be synchronised in cor***	advisory	easy	yes, on local level	accept for initial revision		CG+Tools	
20	***0.12.11 [58] Objects of type std::mutex shall not have dynamic stora***	mandatory	easy	yes, on local level	accept for initial revision		CG+Tools	cg, tools
21	***0.13.1 [64] Use higher-level standard facilities to implement parallelis***	directive			accept for initial revision		CG+Tools	
22	***0.13.5 [68] Functor used with a parallel algorithm shall always return***	mandatory	easy	yes, on local level	accept for initial revision		CG+Tools	
23	***0.13.7 [70] Catch handlers enclosing algorithms with execution polic***	mandatory	easy	yes, on local level	accept for initial revision		CG+Tools	
24	***0.13.8 [71] The binary_op used with std::reduce or std::transform_re***	directive	complex	may be, on system level	accept for initial revision		Tools+CG	
25	***0.13.9 [72] The Function argument used with an algorithm shall not u***	mandatory	easy	yes, on local level	accept for initial revision		CG+Tools	
26	***0.14.9 [81] Do not discard the result of mutex types' try_lock* functio***	mandatory	easy	yes, on local level	accept for initial revision		CG+Tools	good for CG, no C

Agenda

- Adding safety to parallelism for both MISRA and C++ CG
  - This year: focus on what we intend to do for C++CG by hazards
- Deep dive to C++CG rules
  - Deadlocks and rejected rules
  - Lifetime violation and new/modified rules
- CG+MISRA: the close to ideal safety partners
  - Ongoing analysis of future C++ parallelism for safety

Deadlocks

Current CG focused on deadlocks prevention:

- CP.20: Use RAII, never plain `lock()`/`unlock()`
- CP.21: Use `std::lock()` or `std::scoped_lock` to acquire multiple `mutex`es
- CP.22: Never call unknown code while holding a lock (e.g., a callback)
- CP.50: Define a `mutex` together with the data it guards. Use `synchronized_value<T>` where possible
- CP.52: Do not hold locks or other synchronization primitives across suspension points

RAII for mutexes

Do not call member functions of std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex, std::shared_mutex and std::shared_timed_mutex objects.

CP.20: Use RAII, never plain `lock()`/`unlock()`

Modification to be proposed:

CP.20: Use RAII, never plain **locking and unlocking member functions of mutexes**

Effects:

- Extends the scope of the rule to timed mutexes
- Encourages RAII for try_lock() use cases
- std::lock() was in the grey area of CP.20 rule, now it is explicitly out of its scope

Multiple mutexes

Use std::lock(), std::try_lock() or std::scoped_lock to acquire multiple mutexes in same scope.

CP.21: Use std::lock() or std::scoped_lock to acquire multiple mutexes

Modification to be proposed:

CP.21: Use std::lock(), **std::try_lock()** or std::scoped_lock to acquire multiple mutexes

Effects:

- std::try_lock() is the reasonable straightforward way to try to acquire multiple mutexes or do something else, if it does not happen
  - mutexes can be easily adopted by std::scoped_lock, on success

Destroying locked mutexes

Do not destroy objects of the following types std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex, std::shared_mutex, std::shared_timed_mutex if object is in locked or shared locked state.

We decided to keep that rule for MISRA but Reject it for CG.

The reason:

- The rule is correct - you should never do that, but…
- It is not a rule for a human - no one will do that on purpose.
- Such thing in the code will be marked as a bug regardless of the rule existence
- At the same time MISRA rule will encourage code analysis tools to develop methodologies for automatic detection of such hard to find cases

Correct order for locking and unlocking

The order of nested locks unlock shall form a DAG.

We do not have definite conclusion for CG - there are reasons to Reject it in the current form.

But there are more reasons to keep it in MISRA…

The reasons:

- This rule is too generic to be easily applied by human.
  - Having it in CG may not be useful enough
- This rule is too specific to be easily applied by a tool.
  - It is almost impossible to build a full graph of the application to check its properties
  - But keeping it in MISRA may inspire tool developers to check such properties at least on visible subgraphs

Agenda

- Adding safety to parallelism for both MISRA and C++ CG
  - This year: focus on what we intend to do for C++CG by hazards
- Deep dive to C++CG rules
  - Deadlocks and rejected rules
  - Lifetime violation and new/modified rules
- CG+MISRA: the close to ideal safety partners
  - Ongoing analysis of future C++ parallelism for safety

Lifetime rules

MISRA: ***A thread shall not access objects whose lifetime has expired***

- Rule is Undecidable by tools, requires System-level analysis
- Required for MISRA as it catches dangerous undefined behavior in code
- Quality of actual diagnostics depends on the quality of implementation of the checker tool
- Core Guidelines wants rules that can be decided efficiently through local reasoning

⇒ Required for MISRA, but rejected for Core Guidelines

Lifetime rules

Current CG rules for lifetime :

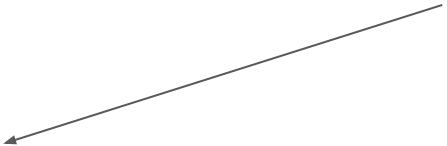
- `CP.23: Think of a joining thread as a scoped container`
- `CP.24: Think of a thread as a global container`
- `CP.26: Don’t detach() a thread`

Promotes scope-based reasoning about lifetimes

But what does it mean?

Scope-based reasoning for lifetimes

```
void f() {

  MyClass obj;

  std::jthread t([&obj]() {

    do_work(obj);

  });

}
```



For data accessed by a thread:
Lifetime of the data must not end before the lifetime of the thread

Is this example safe?

- Yes, because lifetime of the thread is clearly bounded (same as with a function call)
- Unless you detach the jthread

`CP.23: Think of a joining thread as a scoped container`

Enforcement: Forbid detach() of joining threads.

Thread and Joining Thread

`CP.24: Think of a thread as a global container`

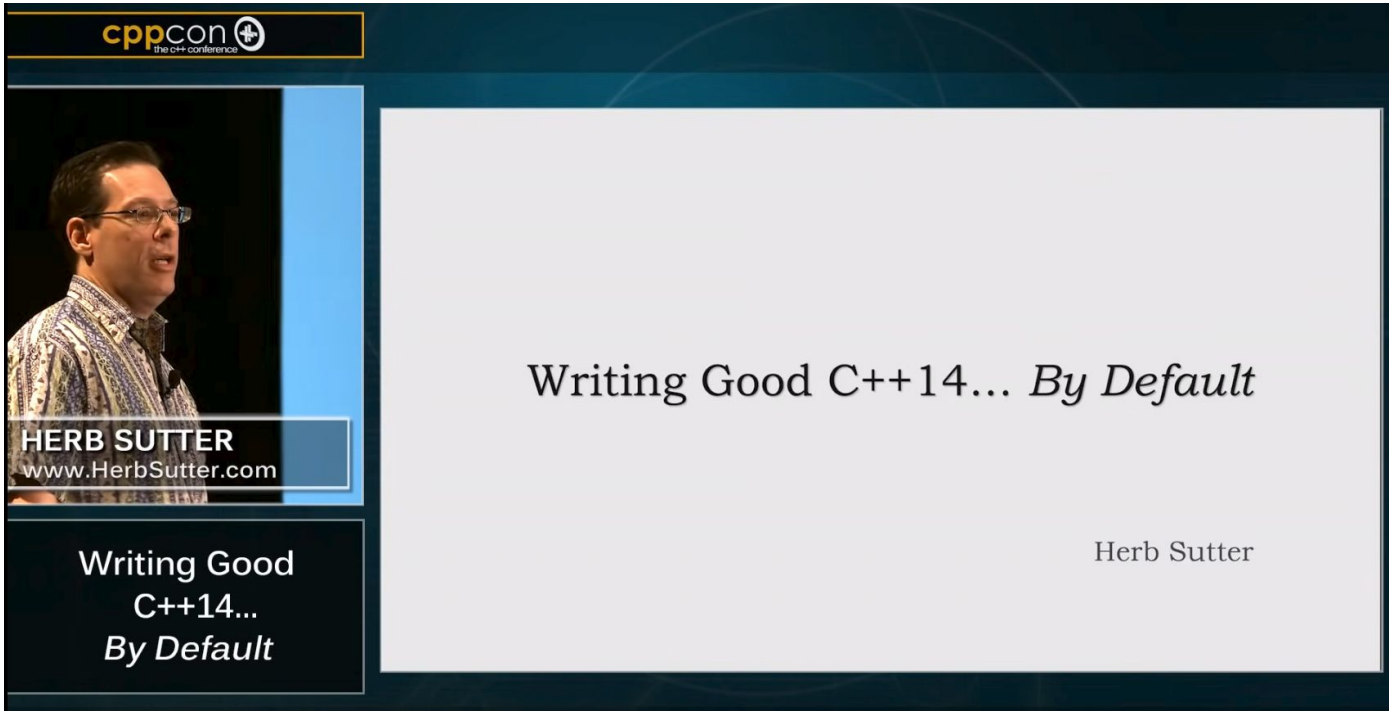
Enforcement: Disallow capturing of local variables

`CP.26: Don’t detach() a thread`

If a tool can prove that CP.26 was followed, a thread can be treated as a joining thread.

But why the analogy with local and global containers?

Core Guidelines Lifetime Model



Core Guidelines Lifetime Model

Core Guidelines: `Pro.lifetime: Lifetime safety`

P1179R1 Lifetime safety: Preventing common dangling

- Assumption: Checker already enforces lifetime profile for containers
- If all threads are scoped, the same check used for lifetime of data stored by containers can also be used for checking lifetime of data captured by thread

Lifetime checking in MISRA

- MISRA currently does not have a lifetime safety profile à la P1179
- We decided not to follow the mental model of CG here but instead stick with the general (Undecidable, System)-rule

Modification to be proposed: Adjust the rules to mention `std::jthread` alongside `gsl::joining_thread`.

New Core Guideline rules

- ***0.4.5 [13] Mutexes locked with std::lock or std::try_lock shall be wrapped with std::lock_guard, std::unique_lock or std::shared_lock with adopt_lock tag within the same scope***
- ***0.4.13 [21] std::recursive_mutex and std::recursive_timed_mutex should not be used***
- ***0.5.4 [26] Do not use std::condition_variable_any on a std::mutex***
- ***0.6.1 [27]Use only std::memory_order_seq_cst for atomic operations***
- ***0.12.4 [51] Always explicitly specify a launch policy for std::async***
- ***0.12.11 [58] Objects of type std::mutex shall not have dynamic storage duration.***

Modifications to existing Core Guidelines rules

- ***0.4.8 [16] Objects of std::lock_guards, std::unique_locks, std::shared_lock and std::scoped_lock classes shall always be named Remember to name your lock_guards and unique_locks***
  - CP.44: Remember to name your lock_guards and unique_locks
- ***0.10.5 [39] Use std::call_once to ensure a function is called exactly once (rather than the Double-Checked Locking pattern)***
  - CP.110: Do not write your own double-checked locking for initialization
- ***0.4.13 [21] std::recursive_mutex and std::recursive_timed_mutex should not be used***
  - CP.22: Never call unknown code while holding a lock (e.g., a callback)

MISRA exclusive rules

These are rules that we accepted for MISRA, but decided that they are not a good match for the Core Guidelines

- ***0.4.4 [12] Do not destroy objects of the following types std::mutex, std::timed_mutex, std::recursive_mutex, std::recursive_timed_mutex, std::shared_mutex, std::shared_timed_mutex if object is in locked or shared locked state***
- ***0.4.11 [19] There shall be no code path which results in locking of the non-recursive mutex within the scope when this mutex is already locked***
- ***0.4.12 [20] The order of nested locks unlock shall form a DAG***

Agenda

- Adding safety to parallelism for both MISRA and C++ CG
  - This year: focus on what we intend to do for C++CG by hazards
- Deep dive to C++CG rules
  - Deadlocks and rejected rules
  - Lifetime violation and new/modified rules
- CG+MISRA: the close to ideal safety partners
  - Ongoing analysis of future C++ parallelism for safety

Safety for Parallel/concurrency for C++20

Asynchronous Agents

C++11: thread,lambda function, TLS, async

C++ 20: Jthreads +interrupt _token:, stop_source,stop_token, stop_callback, request_stop(),stop_requested(), coroutines

Concurrent collections

C++11: Async, packaged tasks, promises, futures, atomics,

C++ 17: ParallelSTL, control false sharing

C++ 20: Vec execution policy, Algorithm un-sequenced policy, span

Mutable shared state/low Latency

C++11: locks, memory model, mutex, condition variable, atomics, static init/term,

C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence,
C++ 17: scoped _lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies
C++20: atomic_ref, Latches and barriers, :atomic<std::shared_ptr<T>>, atomic<std::weak_ptr<T>>
Atomics & padding bits
Simplified atomic init
Atomic C/C++ compatibility
Semaphores, atomic<T> waiting: var.wait(), var.notify_one(), var.notify_all()
Fixed gaps in memory model , Improved atomic flags, Repair memory model

Heterogeneous/Distributed/Accel

C++11: lambda

C++14: generic lambda

C++17: , progress guarantees, TOE, execution policies

C++20: atomic_ref,, span

Ongoing work but we do know a few things

- atomic<std::shared_ptr<T>>
  - May or may not be lock free
  - If lock-free, likely not end-to-end lockfree
  - Slow under high contention
- Atomic_ref: all access access to that object must use atomic_ref
- Semaphores: attempting to acquire a slot when the count is 0 will either block or fail
- Jthreads: surprising if you are used to pthreads, but not if you know RAII
  - Cooperative cancellation, if target doesn't check, nothing happens!

CP.coro: Coroutines

This section focuses on uses of coroutines.

Coroutine rule summary:

- CP.51: Do not use capturing lambdas that are coroutines
- CP.52: Do not hold locks or other synchronization primitives across suspension points
- CP.53: Parameters to coroutines should not be passed by reference

How to use future C++ parallel + TS2 (or IS26) safely

Deferred reclamation can be applied readily to most concurrent linked data structures

- HP
  - Not hard to convert ref count to HP
  - No blocking concerns as Reclamation objects are bounded
  - HP now being amenable to synchronous cleanup in future Cannot have external dependencies in destructors
- RCU
  - Reader might block reclamation if unbounded, so an unbounded amount of memory might remain unclaimed
  - But in safety critical,memory is bounded by the maximum duration of RCU read-side critical section X max amount of memory retired per unit of time
  - In safety if you use static allocation then you will not have new injections and this is actually good as it will not block reclamation
  - If you recycle a fixed number of statically allocated blocks, then blocking in an RCU reader is less damaging to updates than blocking in an reader-writer-locking reader.
  - An RCU reader typically only blocks recycling of memory, allowing updates to proceed concurrently with RCU readers.
  - In contrast, a reader-writer-locking reader blocks updates entirely.
- Coroutines:
  - Similar to things like std::mutex, RCU readers should not span a coroutine suspension point (unless special non-standard extensions or use cases are applied).
  - Similar to reference counting, hazard pointers can be held across coroutine suspension points, and further can be passed from one thread to another.
- Both hazard pointers and RCU can have debugging issues due to thread switching

What is the difference C++CG and MISRA C++

- These are the best 2 guidelines, both are actively updated
- CG is a coding guideline, safety is a by-product
  - Human evaluation and some machine evaluation
  - State rules in positive: e.g. Do this ….
  - Aim for more elegance which can be safe but not necessarily safe
  - Updated as new C++ comes out, does not maintain older C++ versions
  - Relies on local static analysis, MS has implemented a lot of it, and one is in clang,-tidy
  - Need to do for good design
  - Lots of sequential rules with a few parallel rules
  - Top down
- MISRA is a safety guideline, not about elegance
  - Trap accidental coding mistakes that can kill
  - State rules mostly in negative: Don't do this …
  - Aim for Machine automated checkable, large number of rules hard for anyone to check individually
  - Updated slower then C++CG because safety compilers support is at least one level behind
  - Coverity, sonarsource, Klocwork, Helix QAC, Axivion …
  - Kind of mechanical, makes money for tool makers
  - Bottom up
  - 2022: all sequential, with ours as the only parallel which will enter after MISRA NEXT

Conclusion

What you need for safety:

- You need both
- MISRA C++ to have a good sense of what can be automatically checked now, and
  - Use MISRA certified tools to support API safety
  - work with safety certified compilers for ABI safety which might be C++17 today or older
- C++ CG to see what is coming, what makes code elegant and by extension safe
  - To reduce the amount of changes in future
  - Overlapping coverage using both to cover safety with elegance
  - See Bjarne's keynote.