
# Code Analysis++

ANASTASIA KAZAKOVA


## About me

- Anastasia Kazakova, @anastasiak2512
- C++ Dev: Embedded, Networking
- C++ Tools PMM and .NET Tools Marketing Lead, JetBrains
- St. Petersburg C++ UG: https://www.meetup.com/St-Petersburg-CPP-User-Group/
- C++ Russia: https://cppconf.ru/en/

# Why Code Analysis?

# Software Quality

**Anastasia Kazakova**
@anastasiak2512

While preparing for my workshop at #CppOnSea, I want to ask you: reply with the very first thing that comes to your mind when you think about software quality.

Readability

Maintainability

tools

fuzzer

battery life

Repeatable tests

less UB

Robustness

Orthogonality

SW helps solving problems

The Last Spike

Languages

Memory management

Expressive code

Simplicity

Work as intended

Documented

Reviews

Reliability

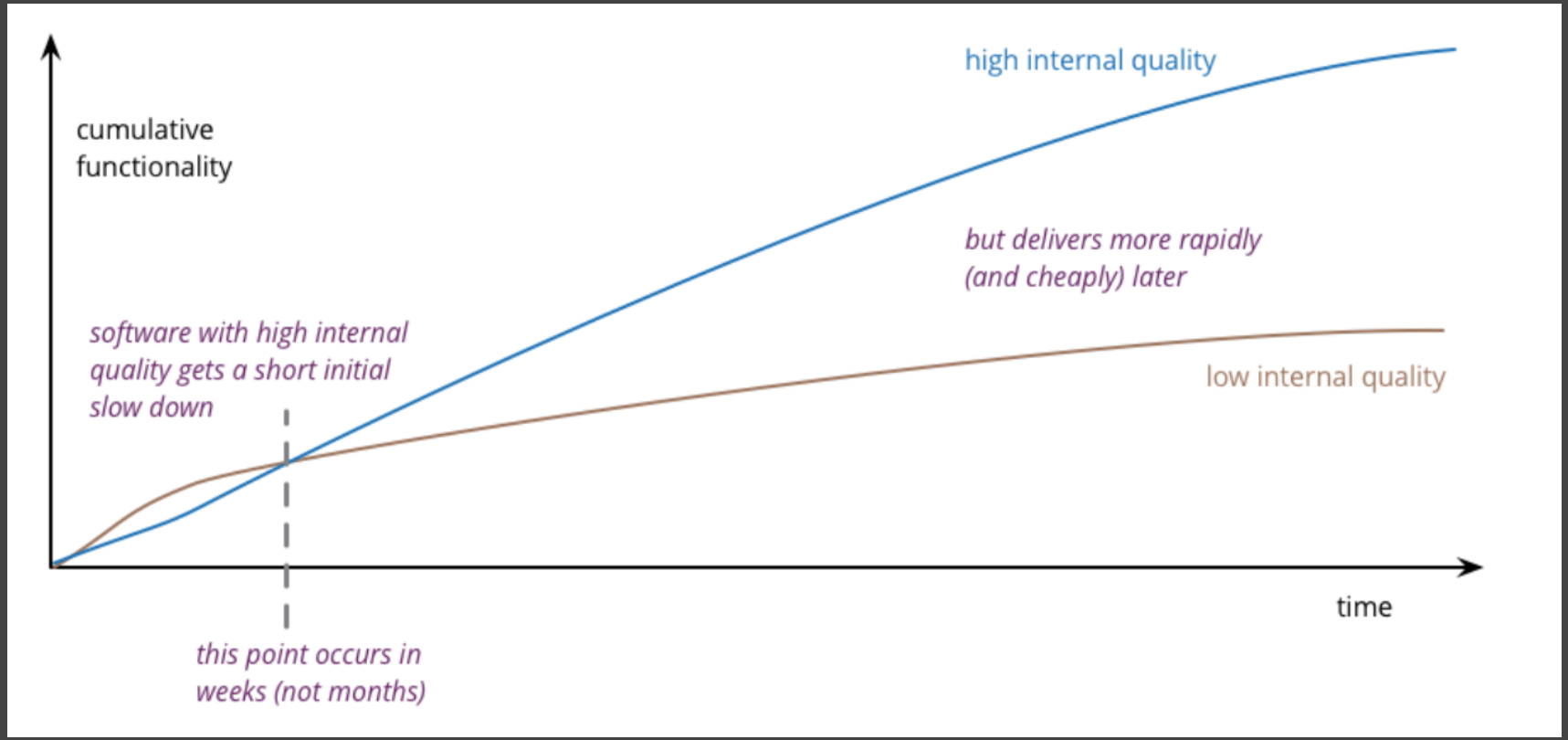
Efficiency

Security

Maintainability

Size

**High quality software is cheaper to produce!**



# Developer Frustration

| Frustration Points | Major % |
|---|---|
| Managing libraries my application depends on | 48 % |
| Build times | 45 % |
| Managing CMake projects | 32 % |
| Setting up a CI pipeline from scratch | 31 % |
| Concurrency safety: Races, deadlocks, performance bottlenecks | 27 % |
| Setting up a dev env from scratch | 26 % |
| Managing Makefiles | 23 % |
| Parallelism support | 22 % |
| Managing MSBuild projects | 18 % |
| Debugging issues in my code | 18 % |
| Memory safety: Bounds safety issues | 16 % |
| Memory safety: Use-after-delete/free | 15 % |
| Security issues: disclosure, vulnerabilities, exploits | 11 % |
| Memory safety: Memory leaks | 11 % |
| Type safety: Using an object as the wrong type | 10 % |
| Moving existing code to the latest language standard | 7 % |

## C++ developer frustration

```
template<class T, int ... X>
T pi(T(X...));
int main() {
   return pi<int, 42>;
}
```

“Problem is, just because the “features” are there, some people will use them. If you’re coding alone, all is peachy. But working in a team? 10 ways of doing 1 thing != good language.”
Twitter, @ArenMook, 24 Dec 2018

“With a sufficient number of uses of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody.”

(Hyrums Law, Software Engineering at Google,
by Titus Winter, Tom Manshrek, Hyrum Wright)

# Undefined Behavior

## Undefined Behavior

- data races
- memory accesses outside of array bounds
- signed integer overflow
- null pointer dereference
- access to an object through a pointer of a different type
- etc.

Compilers are not required to diagnose undefined behavior!

## Undefined Behavior

Fun with NULL pointers, part 1: https://lwn.net/Articles/342330/

```
static unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    unsigned int mask = 0;

    if (!tun)
        return POLLERR;
```

## Why code analysis
–

- Improve software quality
- Lower developer frustration
- Avoid UB

# Language

## Language helps!

- Lifetime safety: http://wg21.link/p1179
  - Owner & Pointer
  - Built-in compiler check
  - Current LLVM implementation gives 5% overhead
  - Annotations to help analysis: *gsl::SharedOwner*, *gsl::Owner*, *gsl::Pointer*

```
void sample1() {
    int* p = nullptr;
    {
        int x = 0;
        p = &x;
        *p = 42; //OK
    }
    *p = 42;      //ERROR
}
```

## Language helps!

- Lifetime safety: http://wg21.link/p1179
- *std::source_location*: since C++20
  - To avoid macro-styled logging and tracing
  - Part of bigger effort

## Language helps!

- Lifetime safety: http://wg21.link/p1179
- *std::source_location*: since C++20
- Contracts: http://wg21.link/p2388, http://wg21.link/p2182
  - [[pre ]], [[post ]], [[assert ]]
  - MVP discussion, *ignore* or *check&abort* modes

## Language helps!

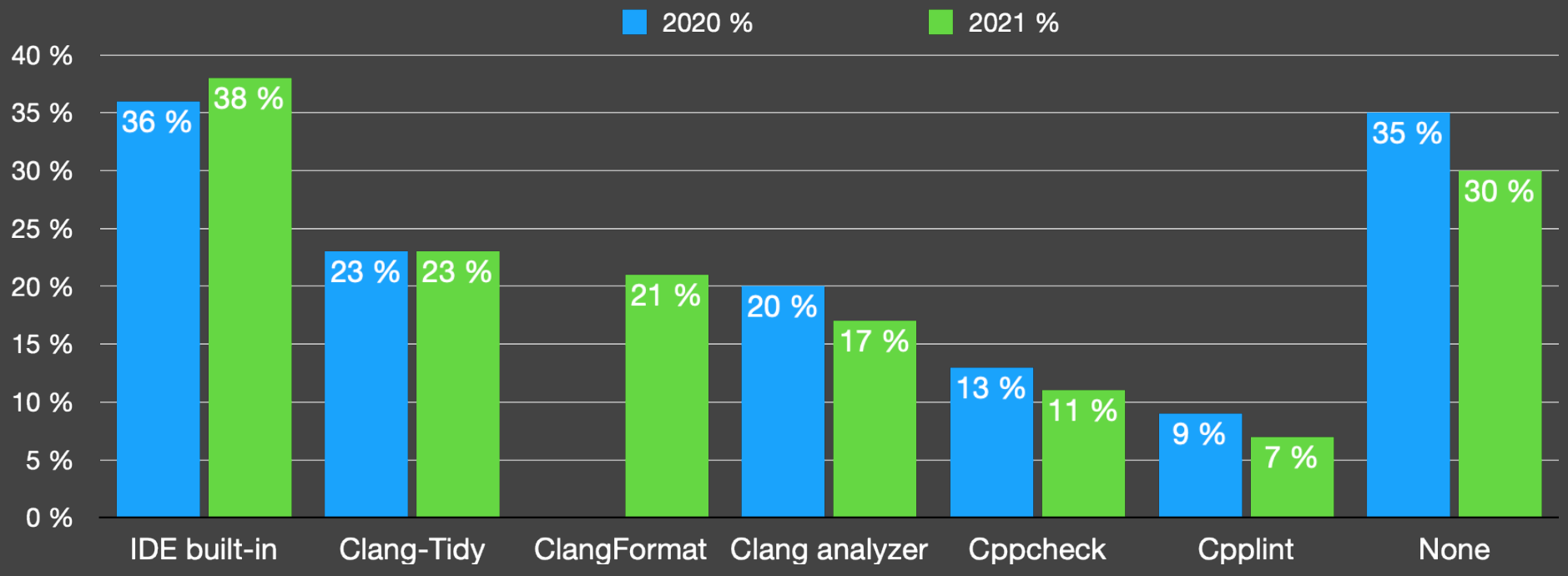
- Lifetime safety: http://wg21.link/p1179
- *std::source_location*: since C++20
- Contracts: http://wg21.link/p2388, http://wg21.link/p2182
- Parameter passing: https://github.com/hsutter/708/blob/main/708.pdf
  - in / inout / out / move / forward semantics
  - Still under discussion, no implementation so far

## Compiler vs Analyzer
–

| Language & Compiler | Stand-alone analyzer |
| --- | --- |
| Core tool – hard to update | Side tool – any adopted by the team is ok |
| Code base might requires specific compiler versions | No strong requirements for analyzer version |
| Set of checks is defined by compiler vendor | Custom checks are possible |
| Standard to everyone | Depends on the tool |

# Tooling

**What do you use for guideline enforcement or other code quality/analysis?**



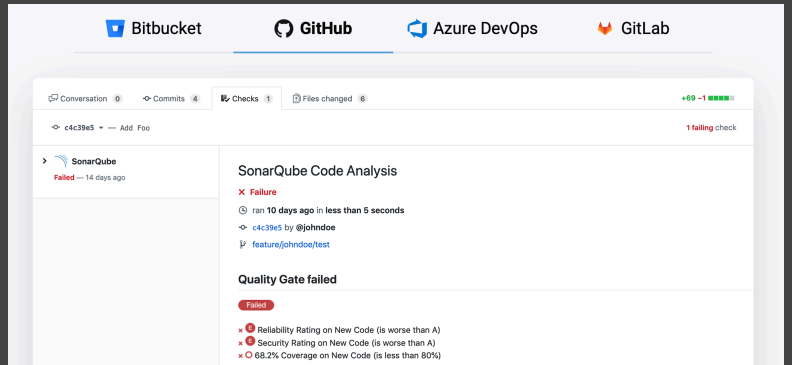
## Code Analysis: CI

–

https://www.sonarsource.com

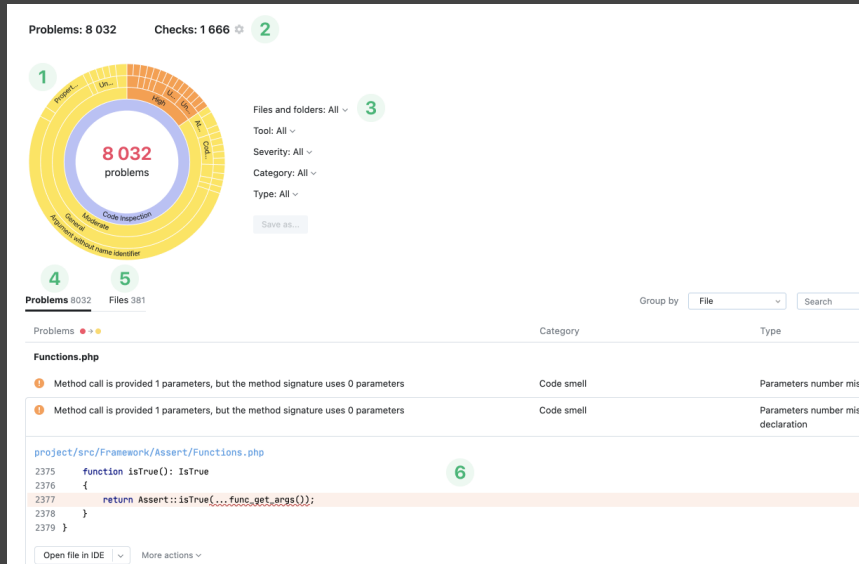
https://rules.sonarsource.com/cpp

- Linter 549 rules
- CI/CD integration
- Code reviews
- PR decorations




https://www.jetbrains.com/qodana/

- Linters from JetBrains IDEs
- CI/CD integrations
- Java (released), Php/Python/JS (EAP), C++ (coming soon)



## Static analysis tools

–

- Compiler errors and warnings

**Compiler errors and warnings**
–

[-Wsign-compare]

```
int a = -27;
unsigned b = 20U;
if (a > b)
    return 27;
return 42;
```

[-Wsizeof-pointer-memaccess]

```
int x = 100;
int *ptr = &x;
memset(ptr, 0, sizeof (ptr));
```

[-Wmisleading-indentation]

```
if (some_condition(cond))
    foo();
    bar();
```

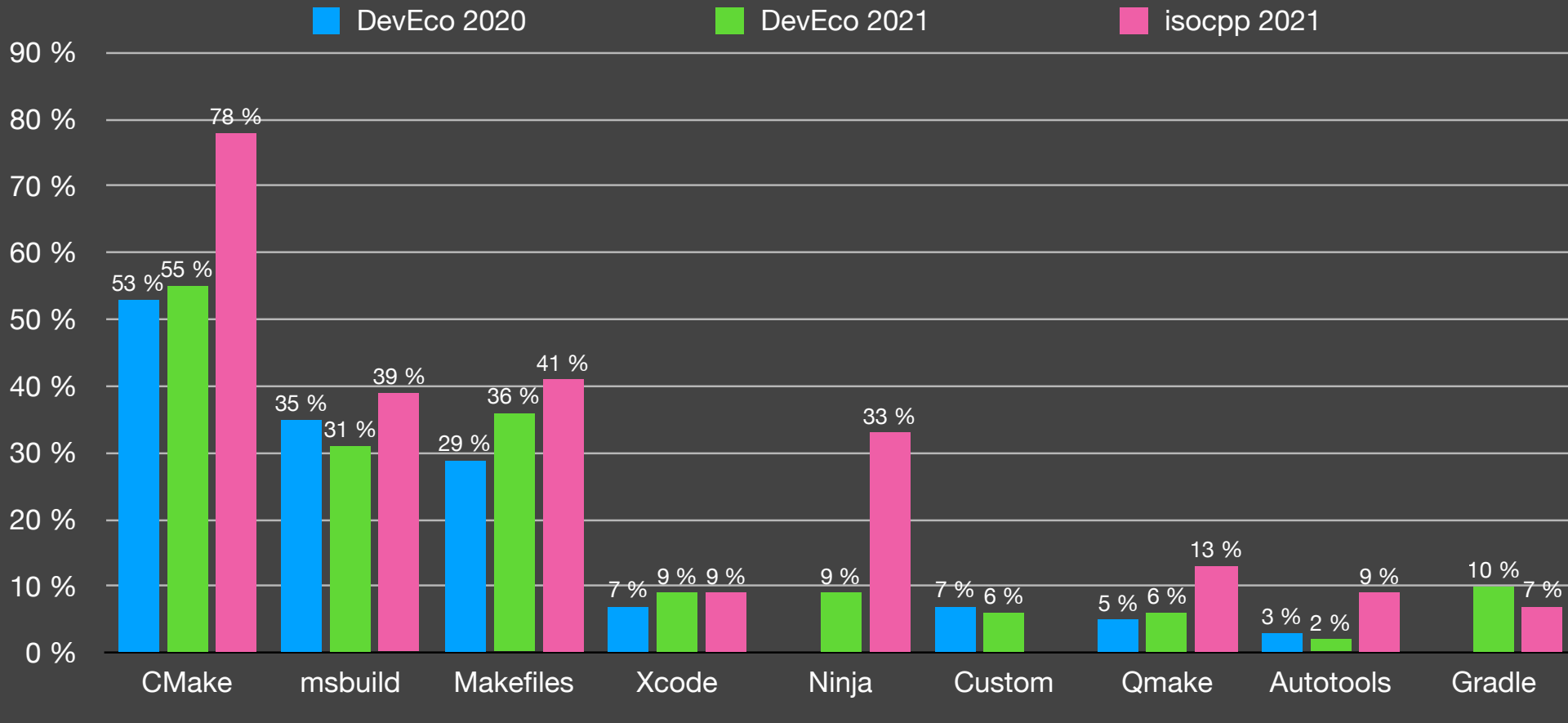
## Compiler errors and warnings

```
if (MSVC)
    # warning level 4 and all warnings as errors
    add_compile_options(/W4 /WX)
else()
    # lots of warnings and all warnings as errors
    add_compile_options(-Wall -Wextra -Werror)
endif()
```

---

```
CXXFLAGS += -Wall -Wextra -Werror
```

## Project models



## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
  - Clang experimental -Wlifetime

## Lifetime safety

```
std::string get_string();
void dangling_string_view()
{
   std::string_view sv = get_string();
   auto c = sv.at(0);
}
```

Object backing the pointer will be destroyed at the end of the full-expression

## Lifetime safety

```
void dangling_iterator()
{
   std::vector<int> v = { 1, 2, 3 };
   auto it = v.begin();
   *it = 0;
   v.push_back(4);
   *it = 0;
}
```

Using invalid operator

## Lifetime safety

```
struct [[gsl::Owner(int)]] MyIntOwner {...};
struct [[gsl::Pointer(int)]] MyIntPointer {...};

MyIntPointer test5() {
  const MyIntOwner owner = MyIntOwner();
  auto pointer = MyIntPointer(owner);
  return pointer;
}
```

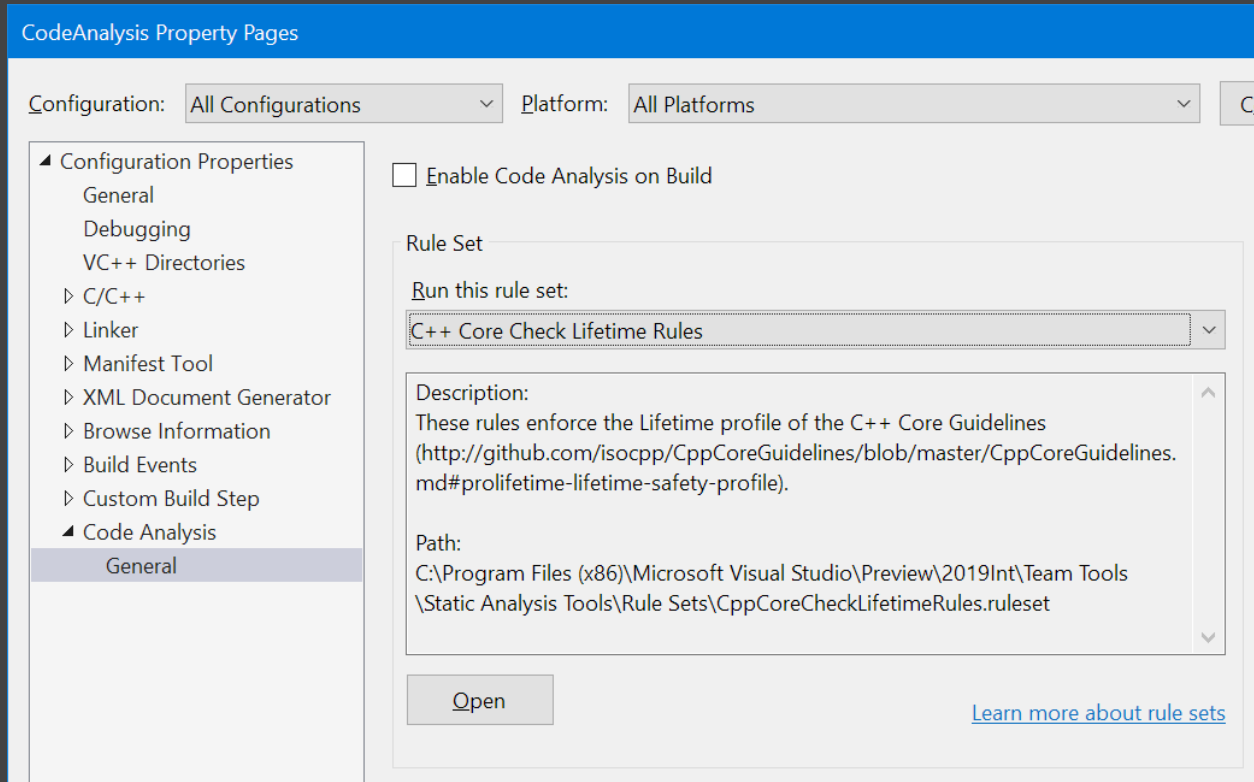
The address of the local variable may escape the function

**Lifetime safety**

–

https://devblogs.microsoft.com/cppblog/lifetime-profile-update-in-visual-studio-2019-preview-2/

Visual Studio 2019

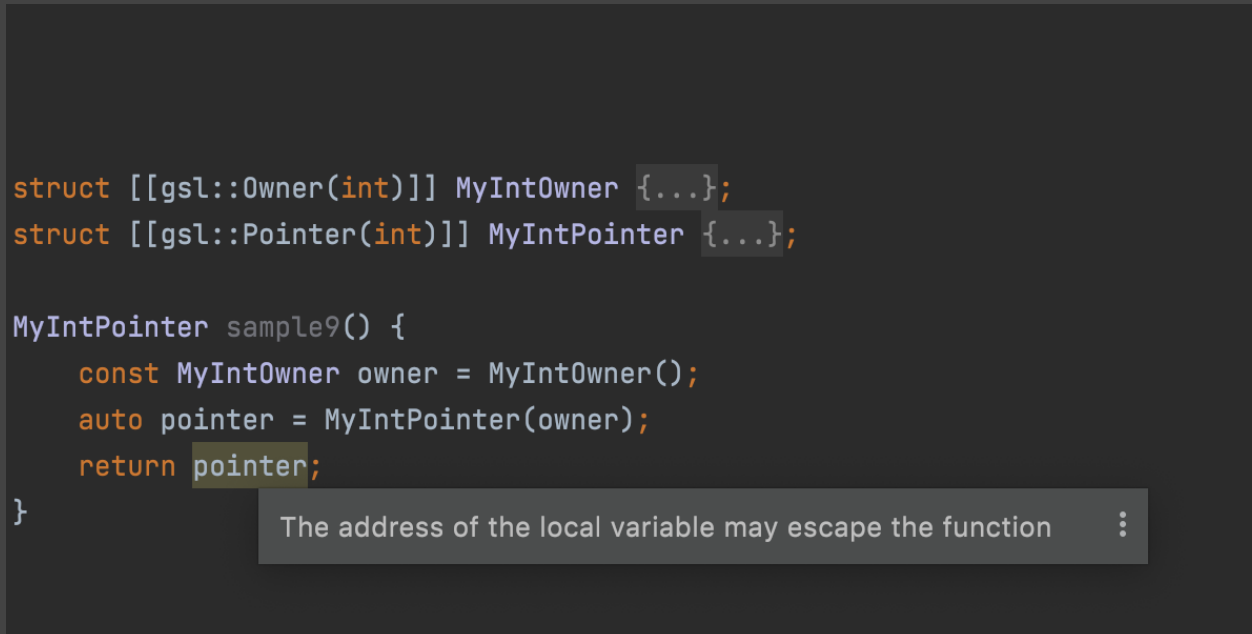


## Lifetime safety

Lifetime analysis in CLion since 2021.2:

https://www.jetbrains.com/clion/whatsnew/#scope-2021-2-code-analysis

https://github.com/anastasiak2512/code_analysis_pp



## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis

## Data Flow Analysis

- DFA analyzes the data:
  - Function parameters/ arguments
  - Function return value
  - Fields and global variables

```
enum class Color { Red, Blue, Green, Yellow };

void do_shadow_color(int shadow) {
    Color cl1, cl2;

    if (shadow)
        cl1 = Color::Red, cl2 = Color::Blue;
    else
        cl1 = Color::Green, cl2 = Color::Yellow;
    if (cl1 == Color::Red || cl2 == Color::Yellow) {...}
}
```

Condition is always true when reached

## Data Flow Analysis

```
void linked_list::process() {
    for (node *pt = head; pt != nullptr; pt = pt->next) {
        delete pt;
    }
}
```

Local variable may point to deallocated memory

## Data Flow Analysis

```
static void delete_ptr(int* p) {
    delete p;
}

int handle_pointer() {
    int* pt = new int;
    delete_ptr(pt);
    *pt = 1;
    return 0;
}
```

Local variable may point to deallocated memory

## Data Flow Analysis

- DFA local/global:
  - Constant conditions
  - Dead code
  - Endless loops
  - Infinite recursion
  - Unused values
  - Null dereference
  - Escape analysis
  - Dangling pointers

```
class Deref {
    int* foo() {
        return nullptr;
    }

public:
    void bar() {
        int* buffer = foo();
        buffer[0] = 0;
    }
};
```

Null dereferencing

## Data Flow Analysis

- DFA global-only:
  - Constant function result
  - Constant function parameter
  - Unreachable calls of function

```
bool always_false() {
    return false;
}

static void foo() {}

void bar(int p) {
    if (always_false())
        foo();
}
```

Unreachable calls

## Data Flow Analysis

CLion:

- Local DFA since 1.x
- Local DFA on Clang since 2020.1
- Global (TU) DFA since 2021.1
- Lifetimes in 2021.2

PVS-Studio:

- Value Range Analysis

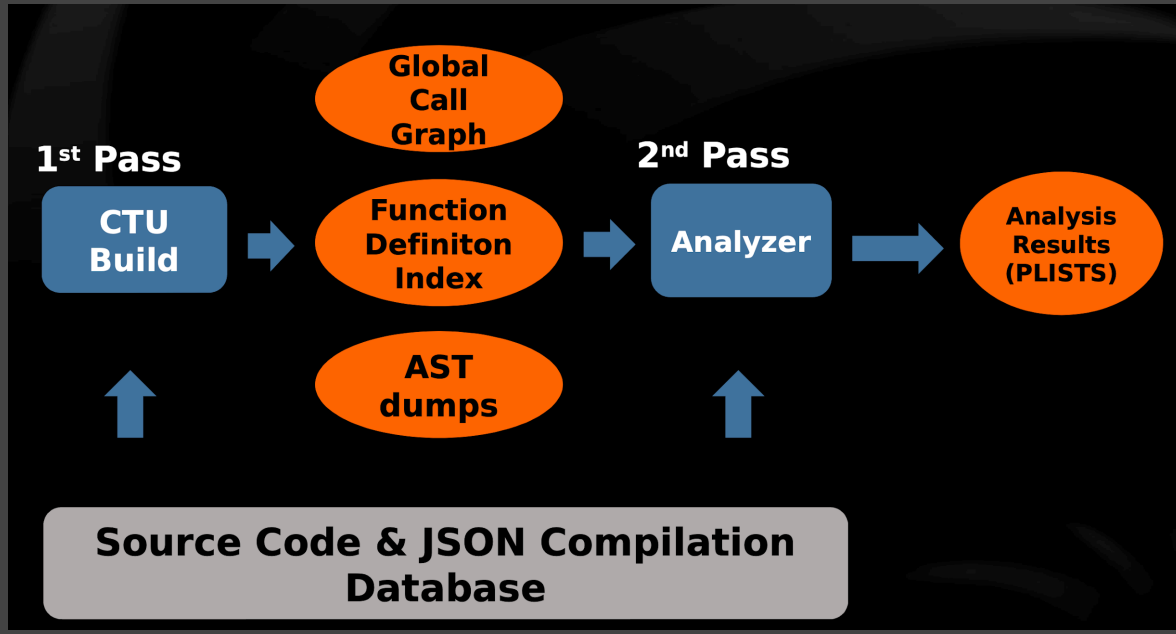
## Data Flow Analysis: CTU

Cross Translation Unit (CTU) Analysis:

https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html

CodeChecker

https://github.com/Ericsson/codechecker



## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines

## C++ Core Guidelines

"Within C++ is a smaller, simpler, safer language struggling to get out."

(c) Bjarne Stroustrup

https://github.com/isocpp/CppCoreGuidelines

## C++ Core Guidelines: toolable

- *F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const*
  - E1: Parameter being passed by value has a size > 2 * sizeof(void*) => suggest reference to const
  - E2. Parameter passed by reference to const has a size < 2 * sizeof(void*) => suggest passing by value
  - E3. Warn when a parameter passed by reference to const is moved
- *F.43: Never (directly or indirectly) return a pointer or a reference to a local object*

## C++ Core Guidelines: not really

- *F.1: “Package” meaningful operations as carefully named functions*
  - Detect identical and similar lambdas used in different places
- *F.2: A function should perform a single logical operation*
  - >1 “out” parameter – suspicious, >6 parameters – suspicious => action?
  - Rule of one screen: 60 lines by 140 characters => action?
- *F.3: Keep functions short and simple*
  - Rule of one screen => action?
  - Cyclomatic complexity “more than 10 logical path through” => action?

## Finding code duplicates

https://stackoverflow.com/questions/191614/how-to-detect-code-duplication-during-development

- CCFinderX
- Duplo
- Simian
- ...others

```
template<class T, int ... X>
T pi(T(X...));
int main() {
   return pi<int, 42>;
}
```

## C++ Core Guidelines: should we?

–

- *F.4: If a function might have to be evaluated at compile time, declare it constexpr*
- *F.5: If a function is very small and time-critical, declare it inline*
- *F.6: If your function may not throw, declare it noexcept*

## C++ Core Guidelines: tools

–

- Guidelines Support Library
- Visual Studio C++ Core Guidelines checkers
- Clang-Tidy: *cppcoreguidelines-**
- Sonar (Qube, Lint, Cloud)
- CLion, ReSharper C++

## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy

## Clang-Tidy

https://clang.llvm.org/extra/clang-tidy/checks/list.html

abseil-* (18),

android-* (15),

cert-* (35),

Clang Static Analyzer,

cppcoreguidelines-* (31),

google-* (22),

modernize-* (31),

performance-* (15), …

*,<disabled-checks>

*vs*

-*,<enabled-checks>

## Static analysis tools
–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy
- Domain-specific analysis tools:
    - MISRA/AUTOSAR, Clazy (Qt), Unreal Header Tool (UE), …

## MISRA
–

| Certification stage | Development stage |
| --- | --- |
| Must have | Good to have |
| High costs | Low costs |
| Defined checks and error messages | Flexible set of checks, detailed messages |
| Rule violations messages only | Checks + Quick-fixes |

**We all care about the same!**

–

- C++ Core Guidelines
  - F.55: Don't use va_arg arguments
  - ES.34: Don't define a (C-style) variadic function
- MISRA
  - MISRA C:2004, 16.1 - Functions shall not be defined with a variable number of arguments.
  - MISRA C++:2008, 8-4-1 - Functions shall not be defined using the ellipsis notation.
- CERT
  - DCL50-CPP. - Do not define a C-style variadic function

## MISRA

–

- CLion MISRA: https://confluence.jetbrains.com/display/CLION/MISRA+checks+supported+in+CLion
  - MISRA C 2012 (63 / 166)
  - MISRA C++ 2008 (64 / 211)
- SonarLint MISRA:
  - https://rules.sonarsource.com/cpp/tag/misra-c++2008 (51 rules)
  - https://rules.sonarsource.com/cpp/tag/misra-c2004 (14 rules)
  - https://rules.sonarsource.com/cpp/tag/misra-c2012 (10 rules)
- PVS-Studio, Cevelop, etc.

## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy
- Domain-specific analysis tools
- Style
    - Formatting, Naming, Syntax style, …

## Formatting

- ClangFormat
  - Formatting standard in C++ nowadays
  - Breaking compatibility
  - Fuzzy parsing

## Naming

- Naming
  - camelCase, PascalCase, SCREAMING_SNAKE_CASE
  - Google style, LLVM, Unreal Engine conversions
  - Requires Rename refactoring + support in code generation/refactorings

## Syntax style

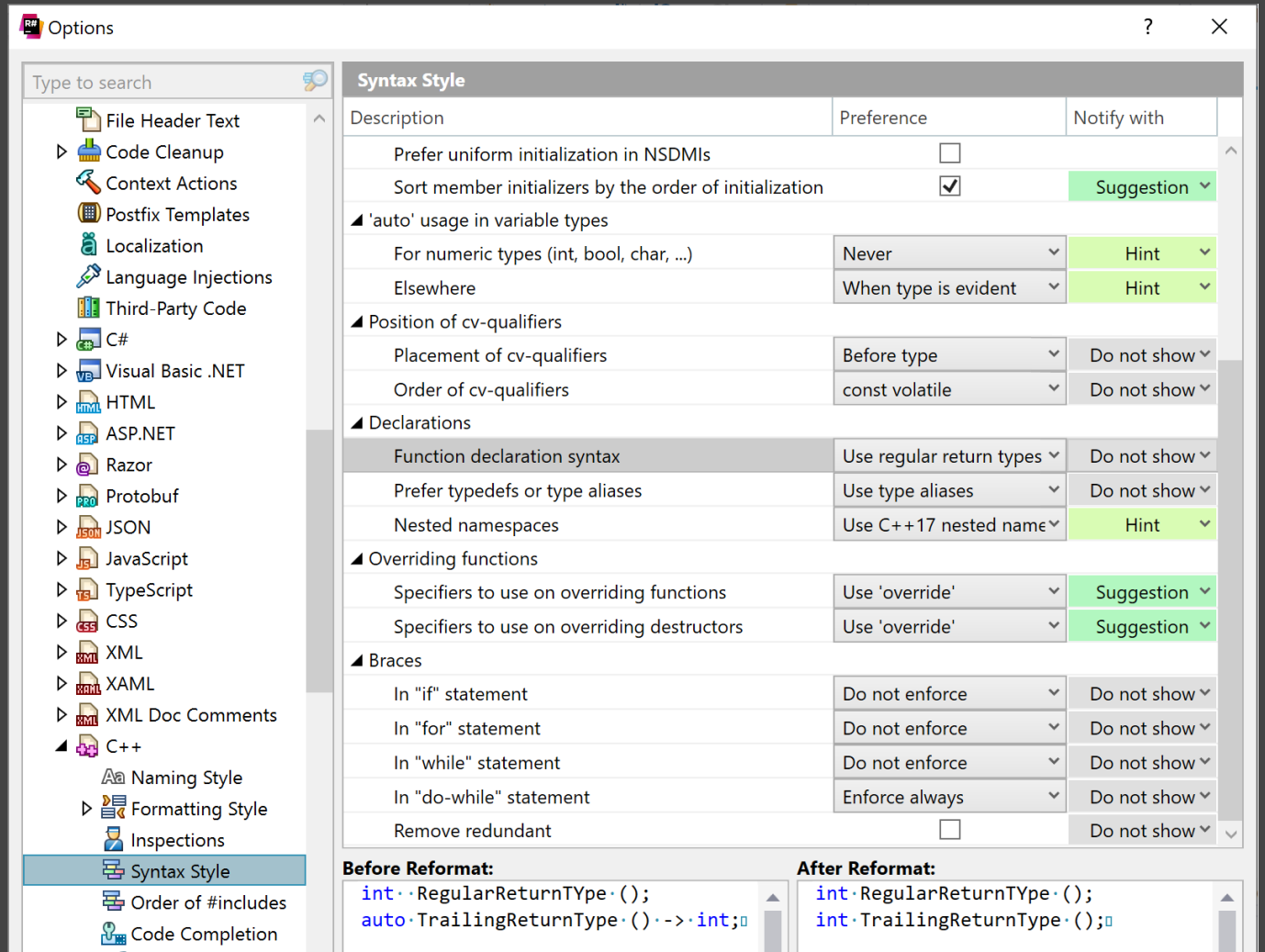
- Syntax style
  - Auto: “Almost Always Auto”, “When Evident”, …
  - “East const” vs. “West const”
  - Typedefs vs. Type Aliases
  - Trailing return types vs. regular
  - Override, final, virtual

## Syntax style

–

ReSharper C++

since 2021.1



## Static analysis tools

–

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy
- Domain-specific analysis tools: Clazy, MISRA/AUTOSAR, etc.
- Style: Formatting, Naming, Syntax style

## References


1. [Is High Quality Software Worth the Cost? By Martin Fowler](https://martinfowler.com/articles/is-quality-worth-cost.html)
2. [Aras Pranckevičius](https://twitter.com/aras_p/status/1076947443823136768)
3. [Tim Sweeney](https://twitter.com/TimSweeneyEpic/status/1409028887279984640)
4. [2021 Annual C++ Developer Survey "Lite"](https://isocpp.org/files/papers/CppDevSurvey-2021-04-summary.pdf)
5. [Lifetime safety: Preventing common dangling](http://wg21.link/p1179)
6. [Lifetime analysis in VS](https://devblogs.microsoft.com/cppblog/lifetime-profile-update-in-visual-studio-2019-preview-2/)
7. [Cross Translation Unit (CTU) Analysis](https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html)
8. [CodeChecker by Ericsson](https://github.com/Ericsson/codechecker)
9. [ReSharper C++: Syntax Style](https://blog.jetbrains.com/rscpp/2021/03/30/resharper-cpp-2021-1-syntax-style/)