Code Analysis++

Anastasia Kazakova

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

About me

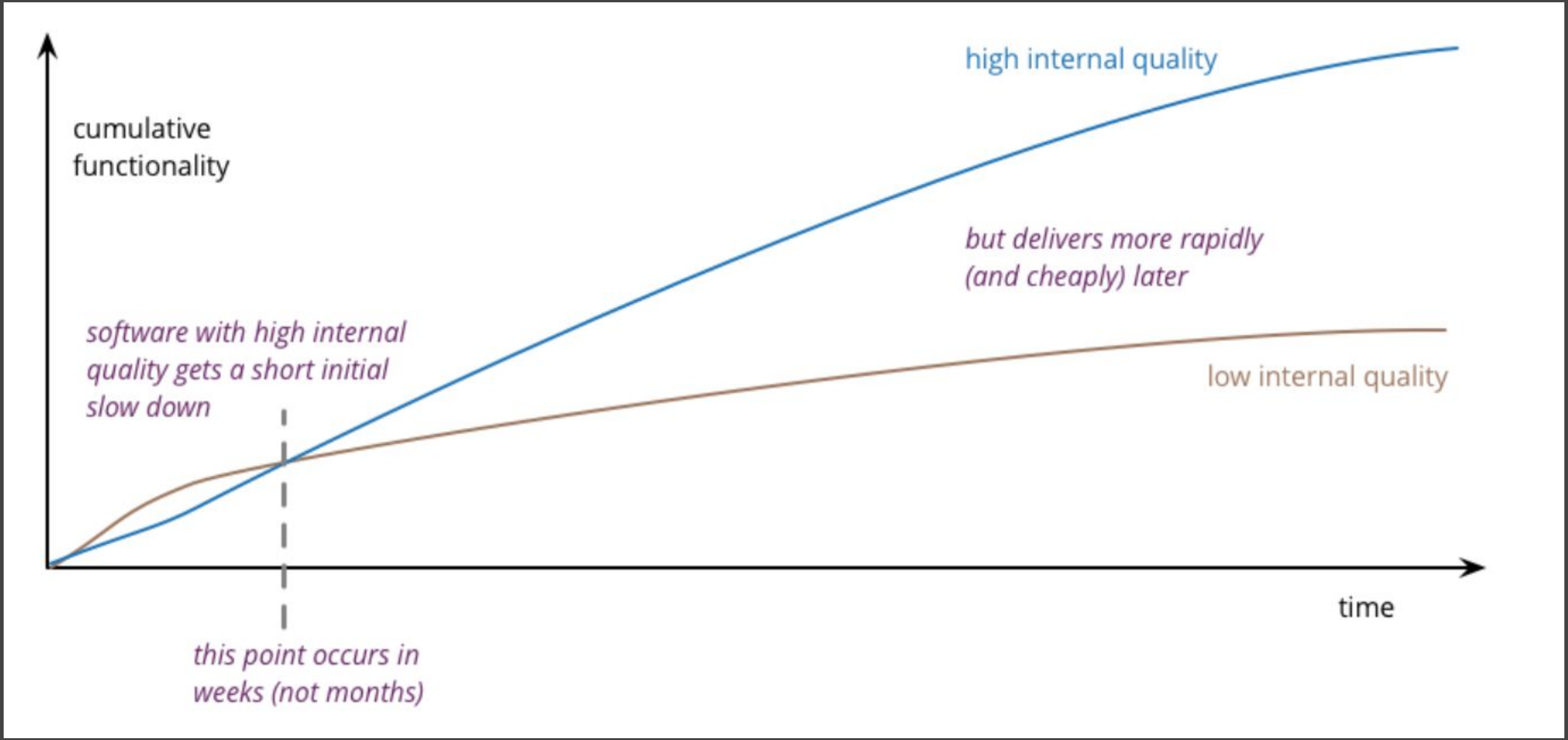
- Anastasia Kazakova
- @anastasiak2512
- C++ Dev, C++ Tools PMM and .NET Tools Marketing Lead, JetBrains
- St. Petersburg C++ UG: https://www.meetup.com/St-Petersburg-CPP-User-Group/
- C++ Russia: https://cppconf.ru/en/

Software quality

Software Quality

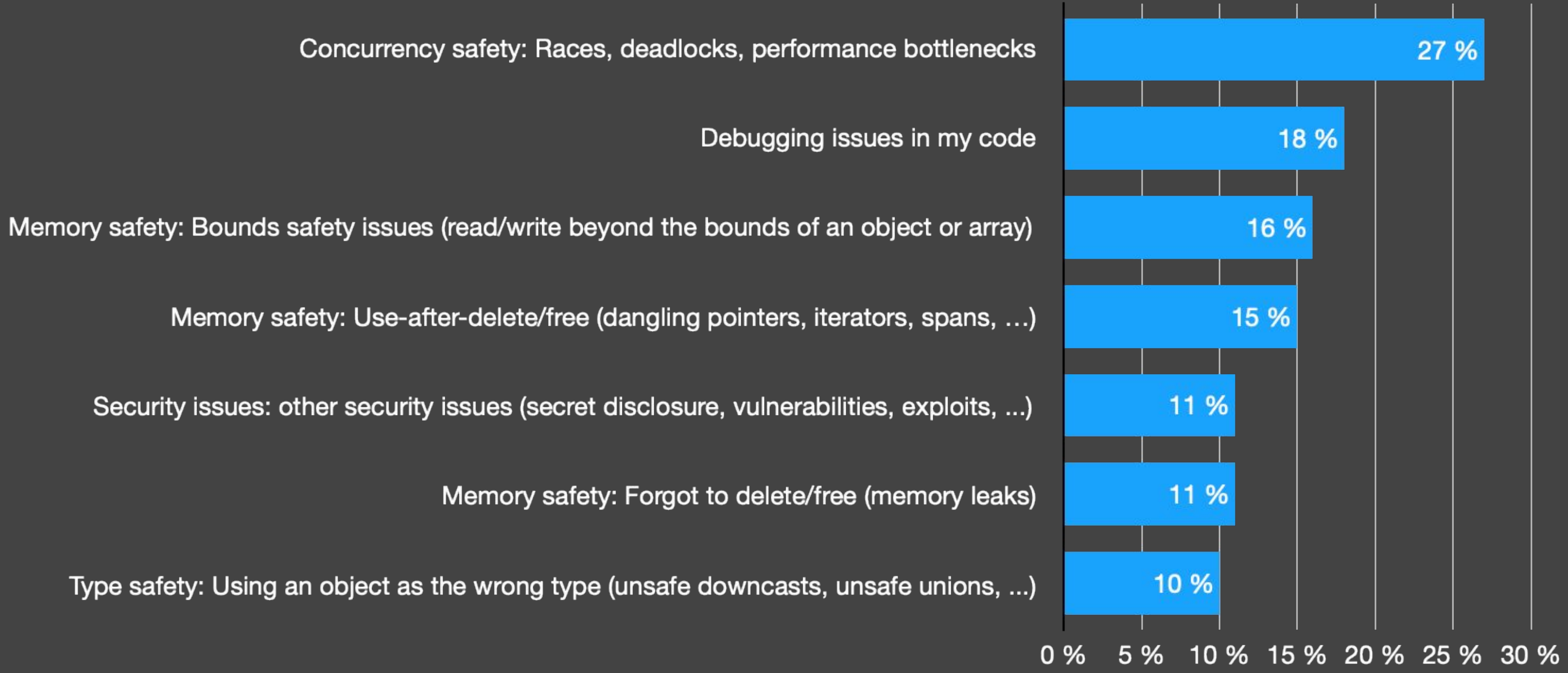
- Define quality
- Find a trade-off between quality and cost of development (spoiler: no!)
- External vs internal quality

High quality software is cheaper to produce!



Developer frustration

C++ development frustration: safety & security



C++ development frustration: style

```
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

Certification process

- Automotive, self-driving experience
- Embedded in general, MISRA/AUTOSAR
- ...

Undefined behavior

UB

- 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!

Why code analysis

- Improve software quality
- Lower developer frustration
- Avoid UB

Language

Language helps!

- Lifetime safety: http://wg21.link/p1179
  - Diagnostic w/o annotations
  - Annotations: gsl::SharedOwner, gsl::Owner, gsl::Pointer, etc. in libraries or user code

Language helps!

- Lifetime safety: http://wg21.link/p1179
- Contracts: http://wg21.link/p2358
  - For programming errors and failures of the C++ abstract machine
  - Assertions, pre- & post- conditions

Language helps!

- Lifetime safety: http://wg21.link/p1179
- Contracts: http://wg21.link/p2358
- Parameter passing: https://github.com/hsutter/708/blob/main/708.pdf
  - in / inout / out / move / forward semantics

Language helps!

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

Software quality: how-to

- Refactoring
- Pair programming
- Static analysis
- Unit testing
- Dynamic analysis
- Code review
- Other testing

Pre-compilation phrase

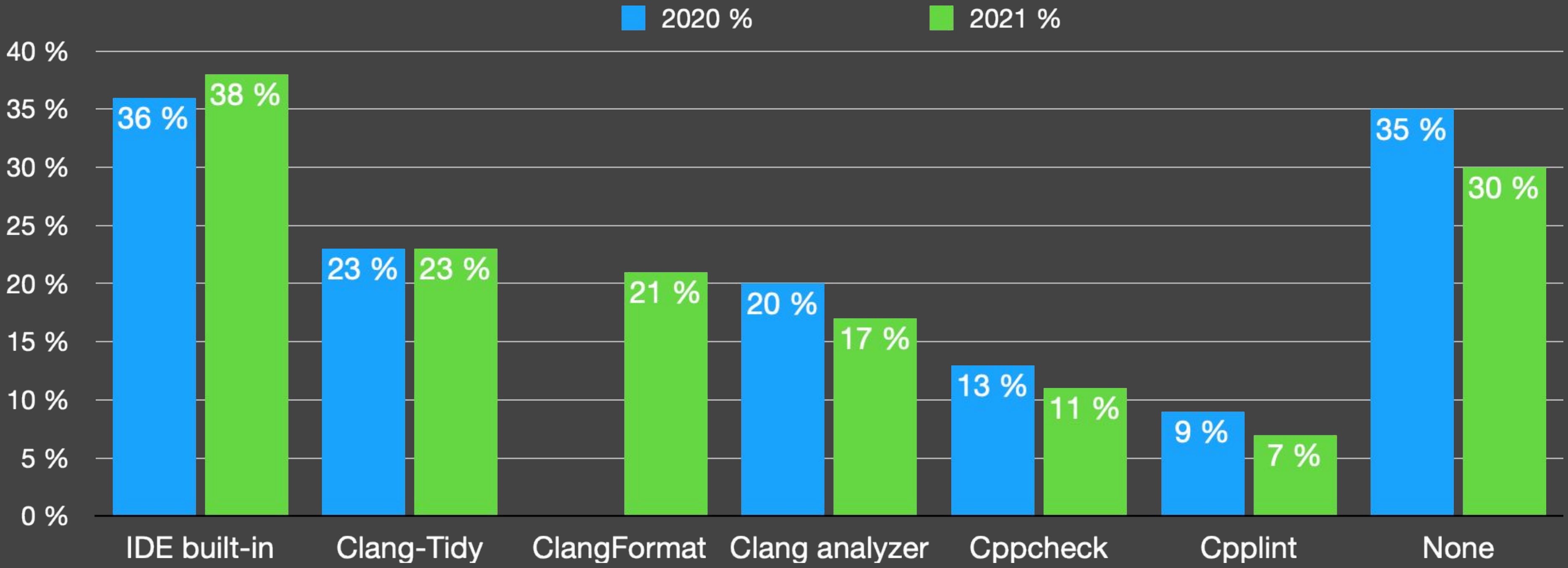
Post-compilation phrase

Static analysis after compilation

–

- Static analysis reports on CI
- Static analysis checks in Code Review
- Static analysis checks for Pull Requests

Dev Eco 2021: Static analysis



Static analysis tools

- Compiler errors and warnings

Compilers: -Wall -Wextra

[-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));
```

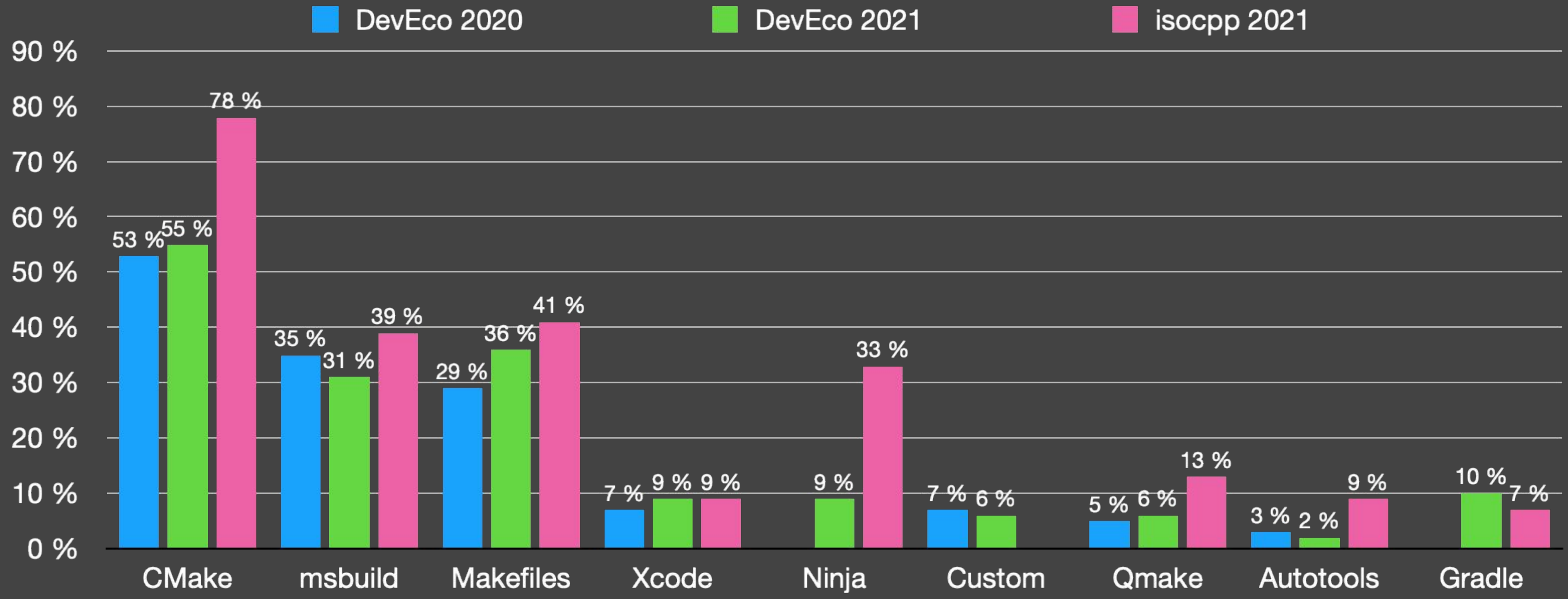
[-Wmisleading-indentation]

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

Compiler helps!

Compiler checks	**Stand-alone analyzer**
Check code after it’s written	Check code while writing it
Analysing the code with proper flags / vars	Should use compilation flags & env
Using specific compiler	Can get checks from other compiler
Different compiler flags	Checks are independent from compiler

Project models 2020 / 2021



Static analysis tools

- Compiler errors and warnings
- Lifetime safety

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
}
```

Static analysis tools

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

Data Flow Analysis

```
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: global (TU)

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

int handle_pointer() {
    int* pt = new int;
    delete_ptr(pt);
    *pt = 1;      Local variable may point to deallocated memory
    return 0;
}
```

Interprocedural dataflow analysis: how-to

- Private Entities
- Unsafe Entities

```
// TU #1
class C {
    void foo(int p);
    void bar();
    void test();
};

void C::foo(int p) {
    if (p == 2) ;
}
void C::test() {
    foo(2);
}
```

```
// TU #2
void C::bar() {
    foo(3);
}
```

Data Flow Analysis: local and global (TU)

- Constant conditions
- Dead code
- Null dereference
- Dangling pointers
- Endless loops
- Infinite recursion
- Unused values
- Escape analysis

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

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

Data Flow Analysis: global (TU)

- Constant function result
- Constant function parameter
- Unreachable calls of function

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

static void foo() {}
```

Unreachable calls

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

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

Data Flow Analysis: global (CTU)

Cross Translation Unit (CTU) Analysis

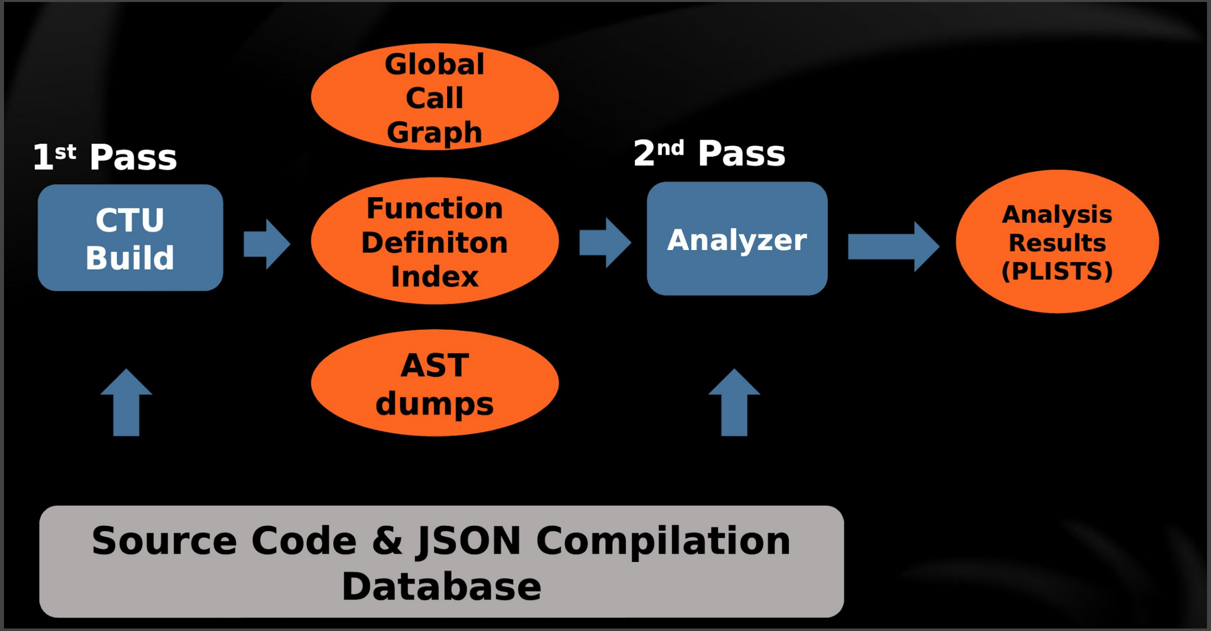
https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html

1. Pre-dumped PCH
2. Generate AST on-demand

Data Flow Analysis: CTU – CodeChecker

–

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

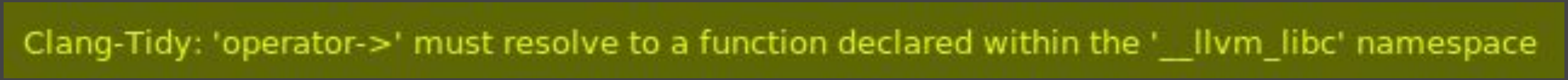
abseil-* (18), android-* (15), cert-* (35), Clang Static Analyzer, cppcoreguidelines-* (31), google-* (22), modernize-* (31), performance-* (15), ...

Clang-Tidy: */ -*

*,<disabled-checks>

vs

-*,<enabled-checks>



```
Clang-Tidy: 'operator->' must resolve to a function declared within the '__llvm_libc' namespace
```

Static analysis tools

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy
- Specific analysis:
    - LLVM coding standard, Clazy, MISRA/AUTOSAR, UHT, ...

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

Static analysis tools

- Compiler errors and warnings
- Lifetime safety
- Data Flow Analysis
- C++ Core Guidelines
- Clang-Tidy
- Specific analysis:
    - LLVM coding standard, Clazy, MISRA/AUTOSAR, UHT, …
- Style & Naming

Style & naming tools

- Clang-Format
  - Formatting standard in C++ nowadays
  - Breaking compatibility
  - Fuzzy parsing
- Naming
  - camelCase, PascalCase, SCREAMING_SNAKE_CASE
  - Google style, LLVM, Unreal Engine conversions
- Syntax style
  - “East const” vs. “West const”

Static analysis tools

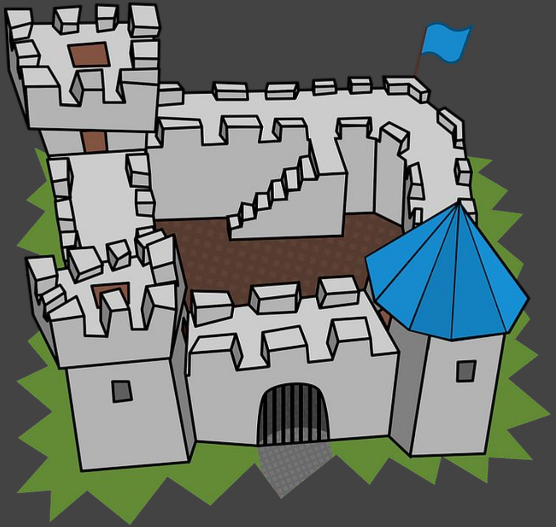
- Software quality & developers frustration
- Language helps!
- Tools help!
  - Compiler
  - Stand-alone analyzers, tools bundled into IDEs
  - C++ Core Guidelines
  - Clang-family tools

Gamifying Static Analysis

–

2018: http://www.bodden.de/pubs/db18gamifying.pdf

- Levels and decomposition
- Motivation
- Using CTA instead of issues
- Team collaborative work



Questions?

—

Thank you!

References

1. [Is High Quality Software Worth the Cost? By Martin Fowler](https://martinfowler.com/articles/is-quality-worth-cost.html)
2. [2021 Annual C++ Developer Survey "Lite"](https://isocpp.org/files/papers/CppDevSurvey-2021-04-summary.pdf)
3. [Lifetime safety: Preventing common dangling](http://wg21.link/p1179)
4. [Cross Translation Unit (CTU) Analysis](https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html)
5. [CodeChecker by Ericsson](https://github.com/Ericsson/codechecker)
6. [Gamifying Static Analysis](http://www.bodden.de/pubs/db18gamifying.pdf)