Статический анализ кода++

Anastasia Kazakova
@anastasiak2512
C++ Tools PMM, JetBrains

High quality software is cheaper to produce

–

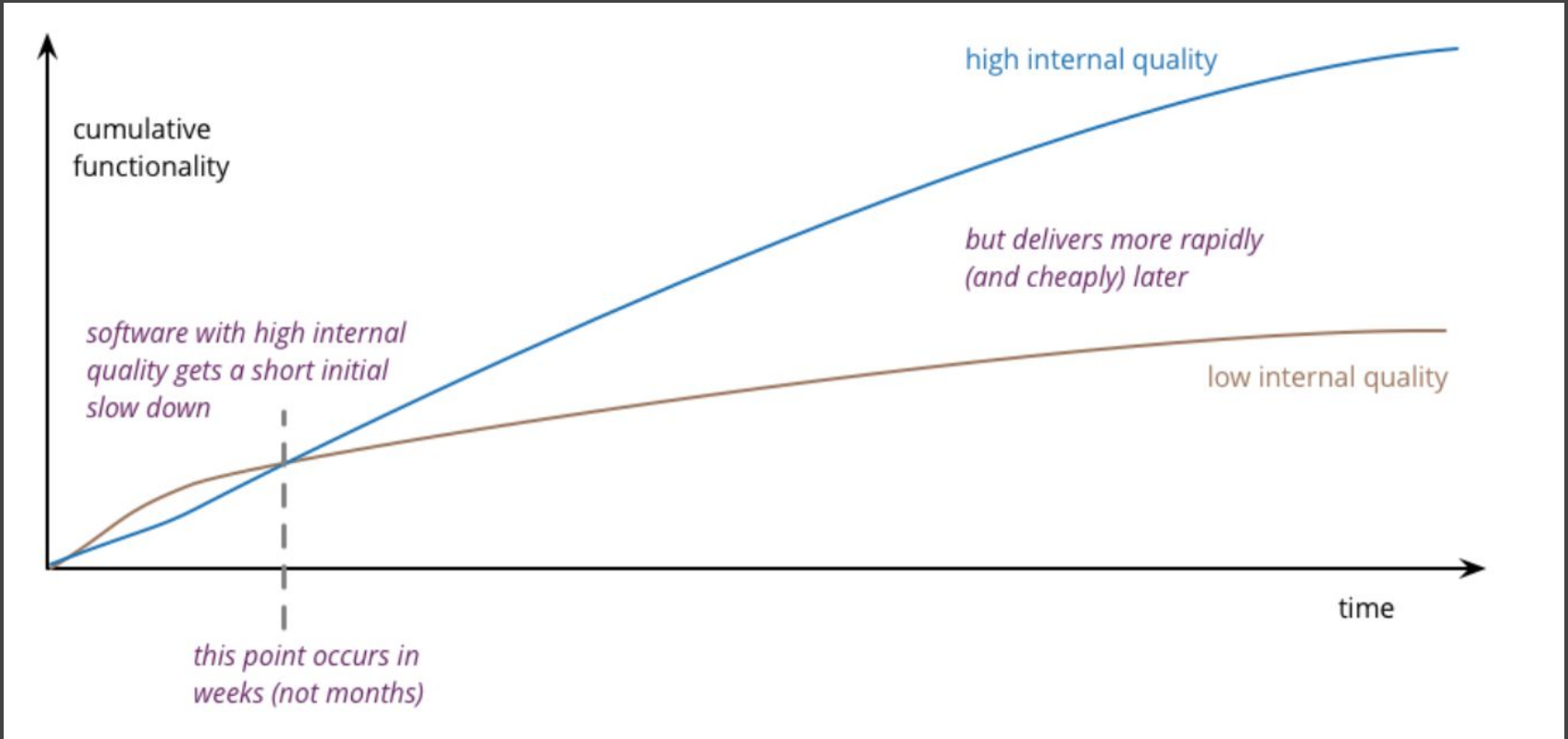
- Martin Fowler: Is High Quality Software Worth the Cost? 29 May 2019
  - Quality:
    - Works reliably and efficiently (no bugs)
    - Readable & Maintainable & Extendable
    - Secure
    - Size
  - We used to trade-off between quality and cost.

High quality software is cheaper to produce

—

- Martin Fowler: Is High Quality Software Worth the Cost? 29 May 2019
  - External vs internal quality

High quality software is cheaper to produce



Software quality: how-to

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

До компиляции

После компиляции

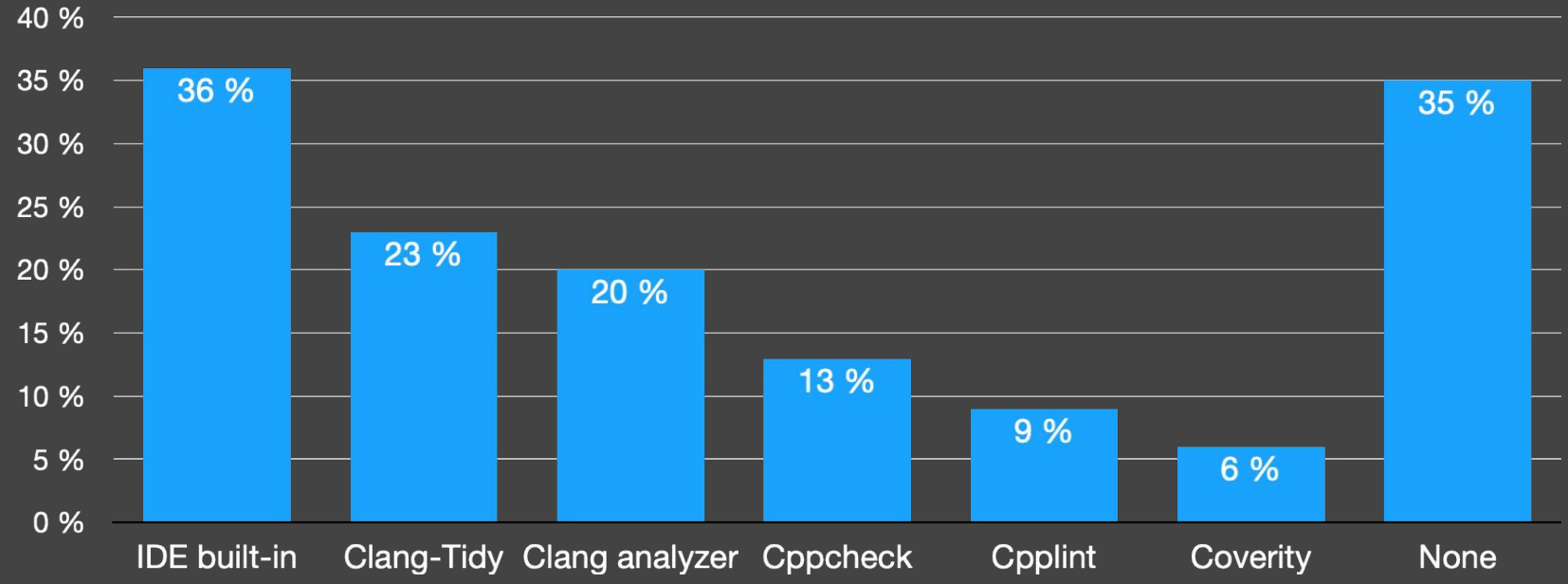
Static analysis on CI and later

–

- Static analysis reports
- SonarQube, Qodana
- Static analysis checks in Code Review
- Static analysis checks for Pull Requests

Dev Eco 2020: Static analysis

Which of the following tools do you or your team use for guideline enforcement or other code quality or analysis?



Dev Eco 2021

–

2021: https://surveys.jetbrains.com/s3/a33-developer-ecosystem-survey-2021

Static analysis: -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();
```

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;
    }
}
```

Dangling pointer

Data Flow Analysis: global (within TU)

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

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

Dangling pointer

Data Flow Analysis: local vs global

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

Local analysis

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

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

Global/ TU analysis

Data Flow Analysis: global (within TU)

–

- Private Entities
- Unsafe Entities

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

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

```
///Translation unit 2
void C::bar() {
    foo(3);
}
```

Data Flow Analysis: global (within 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 (within TU)

–

- 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

CLion 2021.1: Global DFA

https://blog.jetbrains.com/clion/2021/01/clion-starts-2021-1-eap-global-dfa-project-view-cmake/#global_data_flow_analysis

Data Flow Analysis: CTU

–

Cross Translation Unit (CTU) Analysis

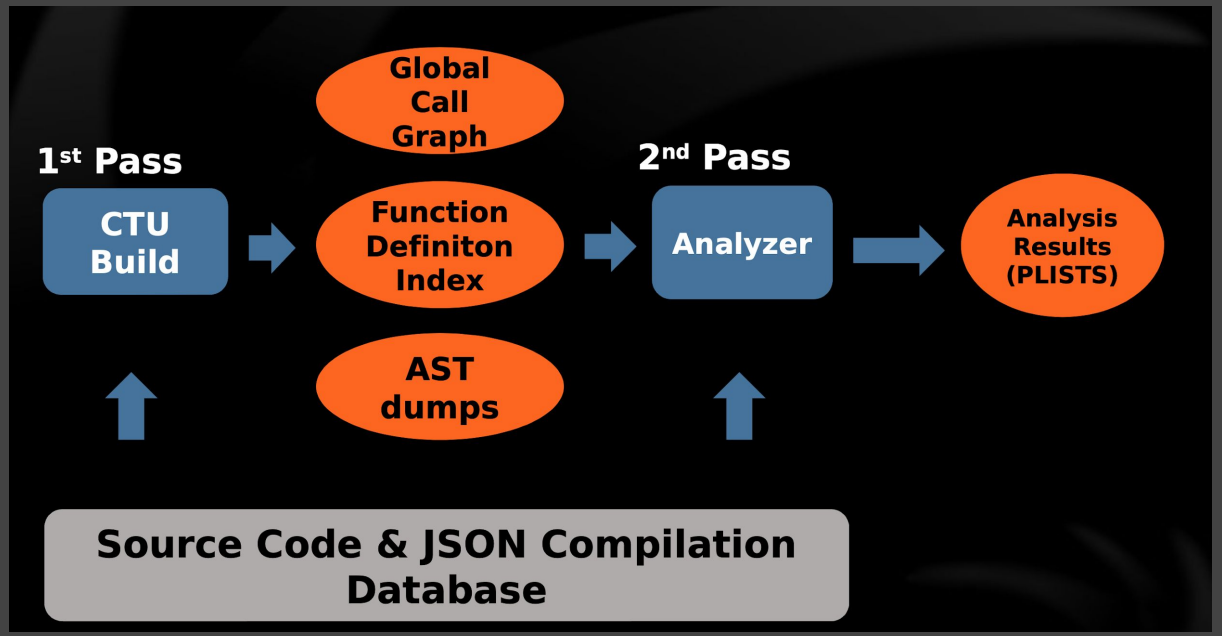
https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html

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

Data Flow Analysis: CTU – CodeChecker

–

CodeChecker https://github.com/Ericsson/codechecker



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 enforcements

–

- *F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const*

```
void f1(const string& s);  // OK: pass by reference to const; always cheap
void f2(string s);         // bad: potentially expensive
void f3(int x);            // OK: Unbeatable
void f4(const int& x);     // bad: overhead on access in f4()
```

- F.43: Never (directly or indirectly) return a pointer or a reference to a local object
- ES.10: Declare one name (only) per declaration
- ES.12: Do not reuse names in nested scopes

C++ Core Guidelines: not really toolable

–

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

Finding code duplicates

–

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

- CCFinderX
- Duplo
- Simian
- ...others

C++ Core Guidelines: should the tool interfere?

—

- *F.4: If a function may have to be evaluated at compile time, declare it constexpr*
  - Enforcement Impossible and unnecessary.
- *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*
  - Flag functions that are not noexcept, yet cannot throw.
  - Flag throwing swap, move, destructors, and default constructors. They should never throw.

C++ Core Guidelines: implementation

–

- Guidelines Support Library: https://github.com/Microsoft/GSL
- Visual Studio: https://docs.microsoft.com/en-us/cpp/code-quality/code-analysis-for-cpp-corecheck?view=msvc-160
- Clang-Tidy: *cppcoreguidelines-** https://clang.llvm.org/extra/clang-tidy/checks/list.html
- SonarSource: https://rules.sonarsource.com/cpp/tag/cppcoreguidelines
- CLion, ReSharper C++

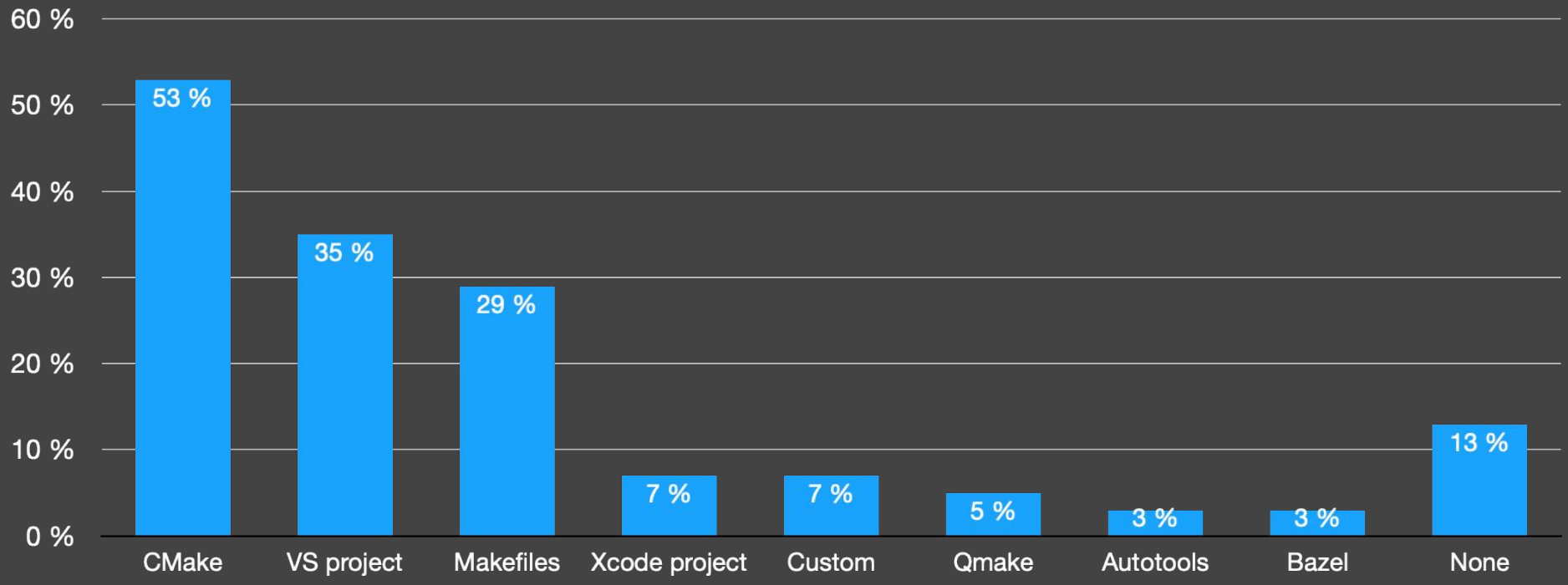
Static analysis tools

–

- Compiler errors and warnings (-Wall -Wextra)

Dev Eco 2020: Static analysis

Which project models or build systems do you regularly use, if any?



Static analysis tools

–

- Compiler errors and warnings (-Wall -Wextra)
- C++ Core Guidelines
- Data Flow Analysis
- 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>

-checks=-,clang-analyzer-*,-clang-analyzer-cplusplus**

-checks=-,cppcoreguidelines-**

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

Static analysis tools

–

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

Static analysis tools: intersections

- 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 (-Wall -Wextra)
- C++ Core Guidelines
- Data Flow Analysis
- Clang-Tidy
- Specific analysis:
  - LLVM coding standard, Clazy, MISRA/AUTOSAR, UHT, …
- Style & naming

Style & naming tools

- Clang-Format
  - Clang-Format Editor in ClangPowerTools
- Naming
  - Rename quick-fixes
  - Specific rules (UE naming)

Language helps!

–

- Lifetime proposal (compiler checks & annotations)
- Contracts (expects/ ensures/ assert)
- Parameter passing

Index

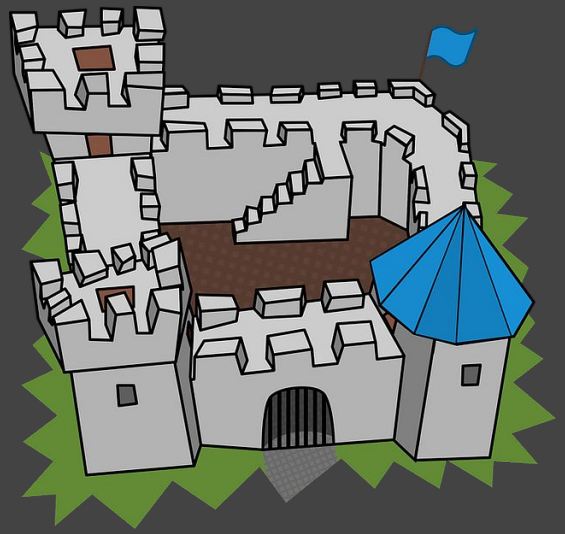
1. High quality software is cheaper to produce.
2. Static analysis tools usage.
3. Compiler warnings.
4. Data Flow Analysis: local -> TU -> CTU.
5. C++ Core Guidelines: from toolable checks to program semantics.
6. Clang-Tidy.
7. Style & naming.
8. Language helps!

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!