Large Scale Refactoring using Clang-Tidy

November 2022

Pierre Rousseau

Software Developer, Coveo

Agenda

1. Why do we need to refactor code?
2. What are LLVM and Clang?
3. What is Clang-Tidy?
4. Refactoring code using Clang-Tidy
5. Overview of Clang-Tidy internals
6. Writing a custom check for Clang-Tidy
7. Questions?
8. References

1. Why do we need to refactor code?

3

Why do we need to refactor code

Refactoring:

Improve code by making small changes without changing its functionality.

- Main reasons to refactor code
  - Make the code cleaner
    - ✓ Easier to understand
    - ✓ Easier to maintain
    - ✓ Easier to find bugs
  - Reduce technical debt: *things that should be fixed, changed or updated*
    - $ Cost associated with a technical debt
    - ✓ Reduce dependencies on aging libraries, frameworks, or style.

C++ Language and STL Evolution Over Time

- 1982: C++
  - Classes, inheritance, strong typing, inlining, default arguments, virtual functions, function name and operator overloading, references, constants, new/delete, Cfront.
- 1985: First edition of *The C++ Programming Language*

Personal interest

- 1989 and 1991: C++ 2.0 was released
  - Multiple inheritance, abstract classes, static member functions, const member functions, protected members.
  - Templates, exceptions, namespaces, bool.
- 1998: C++98 First ISO Standard for C++, STL

20 years ago

- 2003: C++03 Minor update

15 years ago

- 2011: C++11 Major update
  - New container classes, thread, promises, futures, TLS, shared_ptr, unique_ptr, new algorithms, =delete, =default, lambda expression , auto, decltype, Uniform Initialization Syntax, nullptr, Delegating Constructors, move semantic, …

10 years ago

- 2014: C++14 Minor update

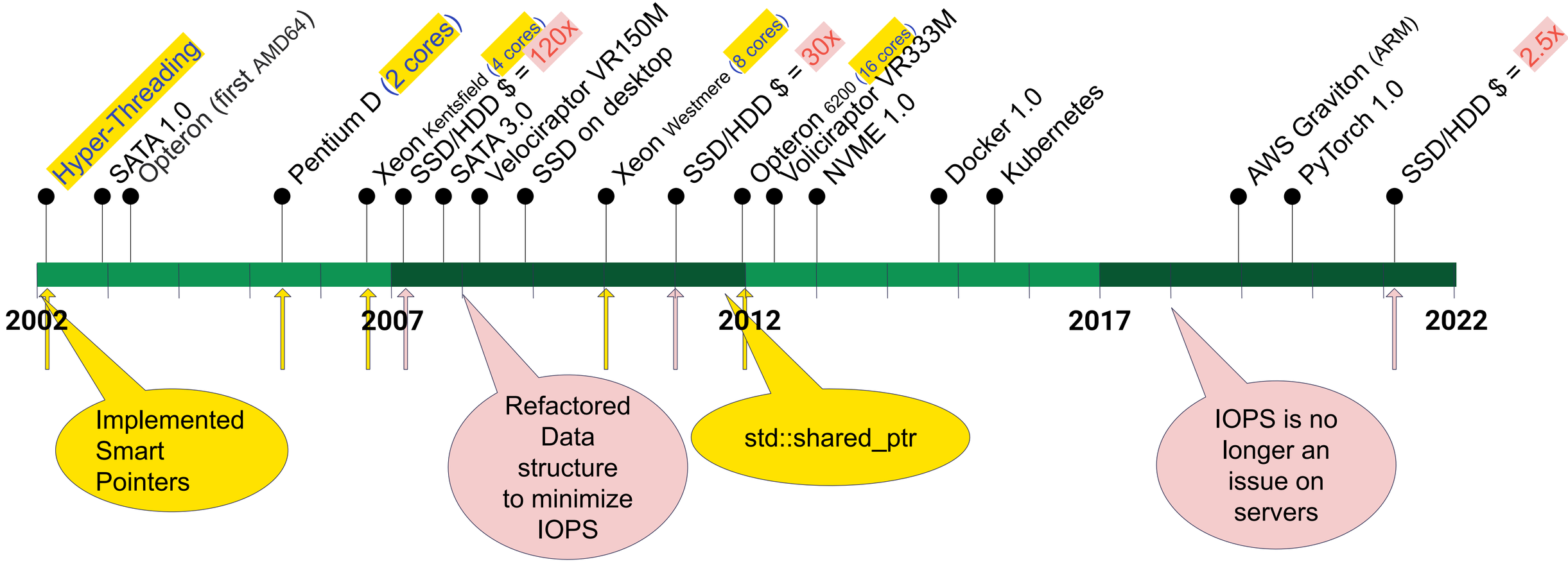
5 years ago

- 2017: C++17 Major update
  - Nested namespaces, variable declaration in if and switch, constexpr if, structured binding, fold expressions, [[attributes]], inline variables, std::variant, std::optional, std::any, std::invoke, std::apply, …
- 2020: C++20
  - Concepts, coroutine, ranges, 3-way comparisons, modules, …

near future

- 2023: C++23 …

Hardware and software evolution over time



Why automatic **large scale** refactoring

*Let say we want to refactor a class that would change **4%** of the lines in a **1,000,000 lines** code base.*

- This mean changing 40 KLOC
- Do we want humans to modify and revise 40 KLOC?

⚠️ How many bugs we would have in those 40 KLOC?

⚠️ Bigger the code review is ⇒ more difficult it gets to find bugs

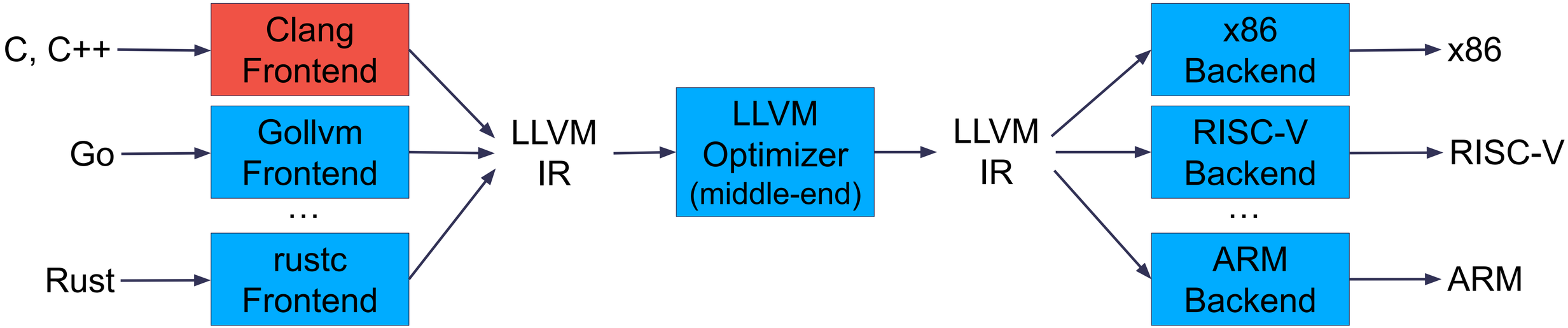
- Automated refactoring
  - Smaller code review (only refactoring tool, and sample output)
  - ⇒Fewer bugs

2. What are LLVM and Clang?

8

LLVM and Clang

- LLVM is a set of compilers and toolchains
- Open source, released in 2003



- IR: high-level assembly language, strongly typed RISC instructions
- IR: using an infinite set of registers

3. What is Clang-Tidy?

10

What is Clang-Tidy

- Clang-based linter for C++
- Configurable with *-checks=* option, or with *.clang-tidy* file
- Support many predefined checks for diagnosing
  - Typical programming errors 💣
  - Style violations 😈
  - Performance issues 🚣
- Many checks offer the possibility to fix the code (using *--fix*)
- New checks could be provided using a dynamic loaded library (Clang-14)

Example 1: Find Potential Problems

clang-tidy --checks="bugprone*" example1.cpp

```
class Example1 {
Public:
   // Lines skipped here …
   Example1& operator=(const Example1& that);
private:
   int* p = nullptr;
};

inline Example1& Example1::operator=(const Example1& that)
{
   p = that.p;
   return *this;
}
```

```
1 warning generated.
example1.cpp:12:28: warning: operator=() does not handle self-assignment properly
[bugprone-unhandled-self-assignment]
inline Example1& Example1::operator=(const Example1& that)
                           ^
```

--fix

4. Refactoring code using Clang-Tidy

Example 2: Modernize Code

```
clang-tidy --checks="modernize-loop-convert" --fix example2.cpp
```

Before:

```cpp
int Example2(std::vector<int>& vector)
{
    int zeroCount = 0;
    for (std::vector<int>::iterator it = vector.begin(); it != vector.end(); ++it) {
        if (*it == 0) {
            ++zeroCount;
        }
    }
```

After:

```cpp
int Example2(std::vector<int>& vector)
{
    int zeroCount = 0;
    for (int & it : vector) {
        if (it == 0) {
            ++zeroCount;
        }
    }
```

Example 3: Performance Improvement

```
clang-tidy --checks="performance-*" --fix example3.cpp
```

Before:

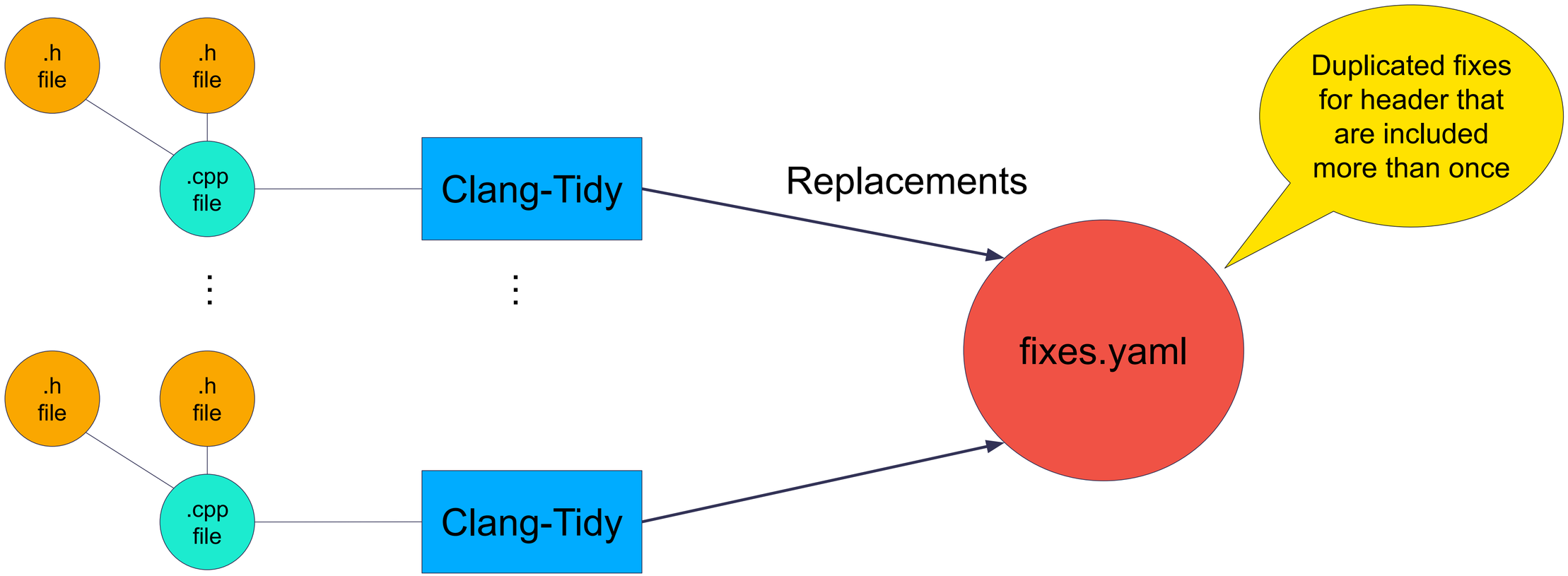
```
class Example3 {
public:
   explicit Example3(const std::string& text) : m_Text(text) {};
private:
   const std::string m_Text;
};
```

After:

```
class Example3 {
public:
   explicit Example3(std::string text) : m_Text(std::move(text)) {};
private:
   const std::string m_Text;
};
```

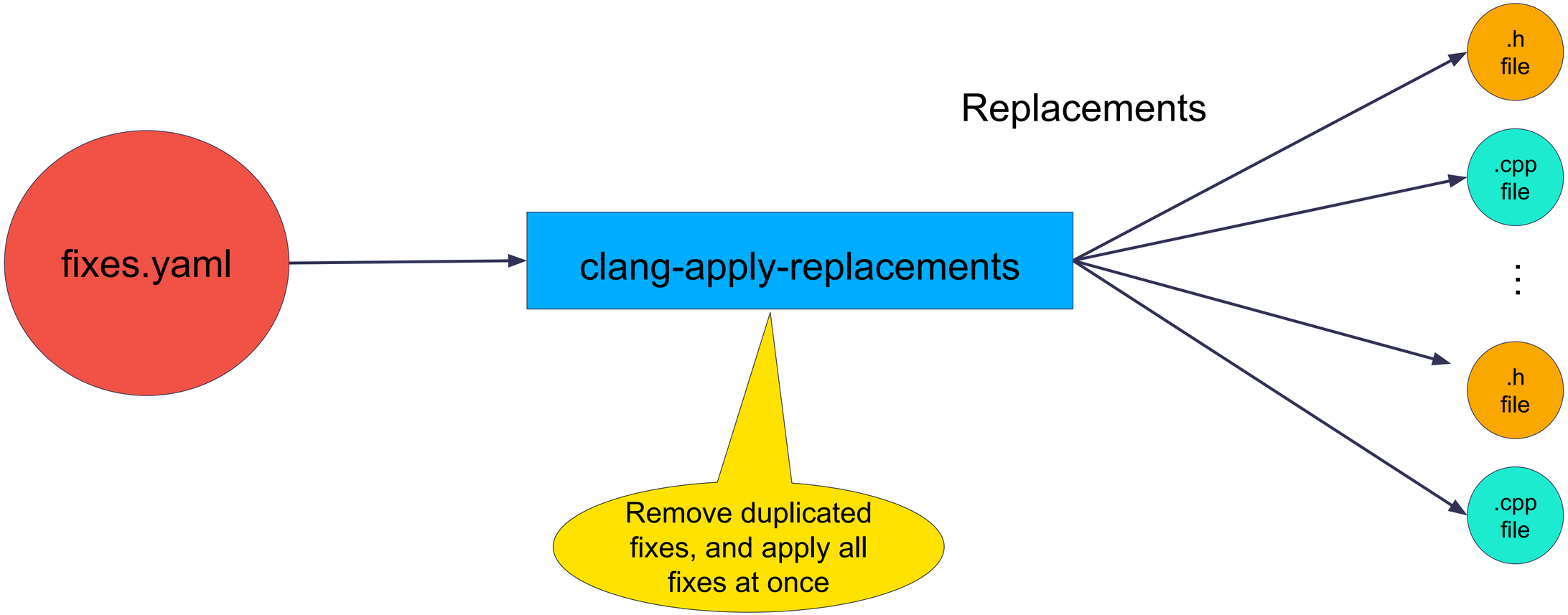
Parallel execution (Step 1)

```
run-clang-tidy -export-fixes=fixes.yaml …
```



Parallel execution (Step 2)

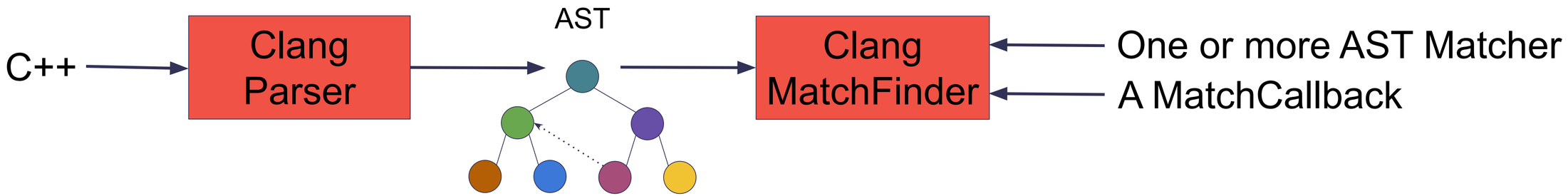
```
clang-apply-replacements .
```



5. Overview of Clang-Tidy internals

18

How Clang-Tidy works



- C++ code is parsed. The output is an AST (*abstract syntax tree*)
- Clang-Tidy uses a MatchFinder to find the code to inspect for a check
- For a check, you must supply:
  - One or more AST Matcher
  - A MatchCallBack
    - Typically, the callback outputs a message using *diag()*
    - Optionally, proposes a fix for the code

Dumping Clang AST

clang -cc1 -ast-dump patate.cpp

```
int Func(const int a, const int b)
{
    return  2 * a + b;
}
```

```
int Func(const int a, const int b)
{
    return  2  *  a  +  b ;
}
```

```
`-FunctionDecl 0x1aa9740 <patate.cpp:1:1, line:4:1> line:1:5 Func 'int (const int, const int)'
  |-ParmVarDecl 0x1aa95a8 <col:10, col:20> col:20 used a 'const int'
  |-ParmVarDecl 0x1aa9628 <col:23, col:33> col:33 used b 'const int'
  `-CompoundStmt 0x1aa9918 <line:2:1, line:4:1>
    `-ReturnStmt 0x1aa9908 <line:3:5, col:20>
      `-BinaryOperator 0x1aa98e8 <col:12, col:20> 'int' '+'
        |-BinaryOperator 0x1aa9890 <col:12, col:16> 'int' '*'
        | |-IntegerLiteral 0x1aa9838 <col:12> 'int' 2
        | `-ImplicitCastExpr 0x1aa9878 <col:16> 'int' <LValueToRValue>
        |   `-DeclRefExpr 0x1aa9858 <col:16> 'const int' lvalue ParmVar 0x1aa95a8 'a' 'const int'
        `-ImplicitCastExpr 0x1aa98d0 <col:20> 'int' <LValueToRValue>
          `-DeclRefExpr 0x1aa98b0 <col:20> 'const int' lvalue ParmVar 0x1aa9628 'b' 'const int'
```

- Some nodes in the AST contains pointers to other nodes
  - Example: the *DeclRefExpr* nodes above contain a pointer to a *ParmVarDecl* node

6. Writing a custom check for Clang-Tidy

Implementing a custom check

1. Derive from *ClangTidyCheck*
2. Derive from *ClangTidyModule*
3. Register your module and your custom check

Implementing a custom check

```
class ReviseLibV2APICheck : public ClangTidyCheck
{
public:
   ReviseLibV2APICheck(StringRef Name, ClangTidyContext* Context) : ClangTidyCheck(Name, Context)
   {
   }
   void registerMatchers(ast_matchers::MatchFinder* Finder) override;
   void check(const ast_matchers::MatchFinder::MatchResult& Result) override;
};
```

- Derive from *ClangTidyCheck* and implement 2 methods
  - *registerMatchers*: Register one or more AST matchers
  - *check*: Add code to check the matching AST, output diagnostic, and suggest a fix

Registering our custom check

```
namespace {

class ReviseLibV2APICheckModule : public ClangTidyModule
{
public:
   void addCheckFactories(ClangTidyCheckFactories& CheckFactories) override
   {
       CheckFactories.registerCheck<ReviseLibV2APICheck>("revise-libv2-api");
   }
};

}  // namespace

namespace clang::tidy {

// Register the module using this statically initialized variable.
static ClangTidyModuleRegistry::Add<::ReviseLibV2APICheckModule> X("revise-libv2", "Revise libv2 API calls.");

// This anchor is used to force the linker to link in the generated object file and thus register the module.
volatile int ReviseLibV2APICheckAnchorSource = 0;

}  // namespace clang::tidy
```

- Our fix is named *revise-libv2-api*

Refactor an API using our own custom check

Lib.h:
(v1)

```
void ProcessRecord(Record* record)
{
   // Do something with the record here ...
   delete record;
}
```

example4.cpp:
(original)

```
#include "lib.h"

void Example4()
{
   ProcessRecord(new Record("key", "value"));
}
```

---

lib.h (v2):

```
void ProcessRecord(std::shared_ptr<Record> record);
```

example4.cpp:
(target)

```
#include "lib.h"

void Example4()
{
   ProcessRecord(std::make_shared<Record>("key", "value"));
}
```

How to write an AST matcher

- Read the documentation
  https://clang.llvm.org/docs/LibASTMatchersReference.html
- 3 types of matchers:
  - **Node** matchers: match a specific type of AST nodes
  - **Narrowing** matchers: match attributes on AST nodes
  - **Traversal** matchers: allow traversal between AST nodes
- Clang-Query
  - Tool for prototyping AST matchers

Using Clang-Query to discover the AST matcher (Part 1)

```
$ clang-query example4.cpp
clang-query> m callExpr()

Match #1:

/usr/lib/gcc/x86_64-linux-gnu/11/../../../../incl                    lloc_traits.h:496:9: note: "root"
binds here
      { __a.deallocate(__p, __n); }
        ^~~~~~~~~~~~~~~~~~~~~~~~

. . . Lines skipped . . .
Match #3089:

./record.h:4:78: note: "root" binds here
   Record(std::string key, std::string data): m_Key(std::move(key)), m_Data(std::move(data)) {}
                                                                              ^~~~~~~~~~~~~~~


Match #3090:

example4.cpp:6:5: note: "root" binds here
   ProcessRecord(new Record("key", "value"));
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3090 matches.
clang-query>
```

First AST matcher to match all call expressions

3089 out of 3090 call expressions are coming from the headers.

Only one is from our code.

We need to refine out AST matcher to match only this call

Using Clang-Query to discover the AST matcher (Part 2)

```
clang-query> m callExpr(callee(functionDecl(hasName("ProcessRecord"))), hasArgument(0, cxxNewExpr()))

Match #1:

example4.cpp:6:5: note: "root" binds here
    ProcessRecord(new Record("key", "value"));
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 match.
clang-query>
```

Matcher	Matcher Kind	Description
`callExpr`	Node	Matches call expressions.
`callee`	Traversal	Checks if the call expression's callee's declaration matches the given matcher.
`functionDecl`	Node	Matches function declarations.
`hasName`	Narrowing	Matches NamedDecl nodes that have the specified name.
`hasArgument`	Narrowing	Matches the n'th argument of a call expression or a constructor call expression.
`cxxNewExpr`	Node	Matches new expressions.

Using Clang-Query to discover the AST matcher (Part 3)

```
clang-query> enable output dump
clang-query> m callExpr(callee(functionDecl(hasName("ProcessRecord"))), hasArgument(0, cxxNewExpr()))

Match #1:

example4.cpp:6:5: note: "root" binds here
   ProcessRecord(new Record("key", "value"));
   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Binding for "root":
CallExpr 0x2c84eb0 <example4.cpp:6:5, col:45> 'void'
|-ImplicitCastExpr 0x2c84e98 <col:5> 'void (*)(Record *)' <FunctionToPointerDecay>
| `-DeclRefExpr 0x2c84e48 <col:5> 'void (Record *)' lvalue Function 0x2c847a8 'ProcessRecord' 'void (Record *)'
`-CXXNewExpr 0x2c84e08 <col:19, col:44> 'Record *' Function 0x2681b90 'operator new' 'void *(std::size_t)'
  `-CXXConstructExpr 0x2c84dd0 <col:23, col:44> 'Record' 'void (std::string, std::string)'
    |-CXXBindTemporaryExpr 0x2c84cd8 <col:30> 'std::string':'std::basic_string' (CXXTemporary 0x2c84cd8)
    | `-ImplicitCastExpr 0x2c84cb8 <col:30> 'std::string':'std::basic_string' <ConstructorConversion>
    |   `-CXXConstructExpr 0x2c84c80 <col:30> 'std::string':'std::basic_string' 'void (const char *, const std::allocator &)'
    |     |-ImplicitCastExpr 0x2c84b98 <col:30> 'const char *' <ArrayToPointerDecay>
    |     | `-StringLiteral 0x2c84aa8 <col:30> 'const char[4]' lvalue "key"
    |     `-CXXDefaultArgExpr 0x2c84c60 <> 'const std::allocator':'const std::allocator' lvalue
    `-CXXBindTemporaryExpr 0x2c84db0 <col:37> 'std::string':'std::basic_string' (CXXTemporary 0x2c84db0)
      `-ImplicitCastExpr 0x2c84d90 <col:37> 'std::string':'std::basic_string' <ConstructorConversion>
        `-CXXConstructExpr 0x2c84d58 <col:37> 'std::string':'std::basic_string' 'void (const char *, const std::allocator &)'
          |-ImplicitCastExpr 0x2c84d20 <col:37> 'const char *' <ArrayToPointerDecay>
          | `-StringLiteral 0x2c84ac8 <col:37> 'const char[6]' lvalue "value"
          `-CXXDefaultArgExpr 0x2c84d38 <> 'const std::allocator':'const std::allocator' lvalue
```

We could use this command to dump the AST.
This is useful to refine the AST matcher.

Implementation of the *registerMatchers* method

- Our AST Matcher from Clang-Query:

```
callExpr(callee(functionDecl(hasName("ProcessRecord"))), hasArgument(0, cxxNewExpr()))
```

- The syntax for the AST Matcher in C++ is the same as in Clang-Query

```
void ReviseLibV2APICheck::registerMatchers(MatchFinder* Finder)
{
   Finder->addMatcher(callExpr(callee(functionDecl(hasName("ProcessRecord"))), hasArgument(0, cxxNewExpr()))
.bind("api_v1_call"), this);
}
```

- We use *bind("...")* to give a name to the matcher

Implementation of the *check* method without fixes

```
void ReviseLibV2APICheck::check(const MatchFinder::MatchResult& result)
{
    const auto* matchedDecl = result.Nodes.getNodeAs<CallExpr>("api_v1_call");

    if (matchedDecl != nullptr) {
        constexpr char message[] = "Use `std::make_shared<Record>` instead of `new Record`.";
        diag(matchedDecl->getBeginLoc(), message, DiagnosticIDs::Warning);
    }
}
```

```
$ clang-tidy --checks=-*,revise-* --load libReviseLibV2Check.so example4.cpp
example4.cpp:6:5: warning: Use `std::make_shared<Record>` instead of `new Record`.
[revise-libv2-api]
    ProcessRecord(new Record("key", "value"));
    ^
```

Implementation of the *check* method with fixes

```
void ReviseLibV2APICheck::check(const MatchFinder::MatchResult& result)
{
    const auto* matchedDecl = result.Nodes.getNodeAs<CallExpr>("api_v1_call");

    if (matchedDecl != nullptr) {
        const auto         arg1             = matchedDecl->getArg(0);
        const auto         arg1Type         = arg1->getType()->getPointeeType()->getTypeClassName();
        const std::string  fixText          = std::string("std::make_shared<") + arg1Type + ">";
        const auto         startNewTypeLoc  = arg1->child_begin()->getBeginLoc();
        const auto         endFixLoc        = startNewTypeLoc.getLocWithOffset(strlen(arg1Type) - 1);
        const auto         sourceRange      = SourceRange(arg1->getBeginLoc(), endFixLoc);

        constexpr char message[] = "Use `std::make_shared<Record>` instead of `new Record`.";
        const auto     fix       = FixItHint::CreateReplacement(sourceRange, fixText);

        diag(matchedDecl->getBeginLoc(), message, DiagnosticIDs::Warning) << fix;
    }
}
```

Text to insert

Begin and end position of the fix

Fix

Stream the fix into the diagnostic message

Example 4: Using our own check to refactor code

```
clang-tidy --checks="-*,revise-*" --load libReviseLibV2Check.so --fix example4.cpp
```

Before:

```
void Example4()
{
   ProcessRecord(new Record("key", "value"));
}
```

Output:

```
example4.cpp:7:5: warning: Use `std::make_shared<Record>` instead of `new Record`.
[revise-libv2-api]
   ProcessRecord(new Record("key", "value"));
   ^              ~~~~~~~~~~
                  std::make_shared
example4.cpp:7:19: note: FIX-IT applied suggested code changes
   ProcessRecord(new Record("key", "value"));
                 ^
clang-tidy applied 1 of 1 suggested fixes.
```

After:

```
void Example4()
{
   ProcessRecord(std::make_shared<Record>("key", "value"));
}
```

7. Questions?

8. References

References

- Stephen Kelly, 2018, blog
  - Exploring Clang Tooling Part 1: Extending Clang-Tidy
  - Exploring Clang Tooling Part 2: Examining the Clang AST with clang-query
  - Exploring Clang Tooling Part 3: Rewriting Code with clang-tidy
  - Exploring Clang Tooling – Using Build Tools with clang-tidy
- Stephen Kelly, 2018-2019, YouTube
  - Refactor your codebase with Clang tooling - code::dive 2018
  - The Future of AST Matcher-based Refactoring

References

- Clang-Tidy documentation
- Clang documentation, Matching the Clang AST
- Clang documentation, AST Matcher Reference
- Vince Bridger, tutorial 2022, Using Clang-Tidy for Customized Checkers and Large Scale Source Tree Refactoring
- Kevin Lalumière, https://github.com/coveooss/clang-tidy-plugin-example
- Source code used in this presentation: https://github.com/coveooss/clang-tidy-plugin-demo
- Internal Google Slides (available only within Coveo)