Clang and Clang-Tidy

1. What is Clang?

Clang is a **compiler front-end** for C, C++, and Objective-C, built as part of the **LLVM project**. A compiler front-end is responsible for **understanding source code**, checking it for errors, and converting it into an internal representation.

Main responsibilities of Clang

- Read source code
- Perform lexical analysis (tokens)
- Perform parsing (syntax)
- Perform semantic analysis (meaning, types, scopes)
- Build an **Abstract Syntax Tree (AST)**

The most important output of Clang is the **AST**.

---

2. How Clang Works (Step by Step)

Step 1: Source code input

```
int add(int a, int b) {
    return a + b;
}
```

Step 2: Tokenization

Clang breaks code into tokens:

```
int | add | ( | int | a | , | int | b | ) | { | return | a | + | b | ; | }
```

Step 3: Parsing

Tokens are parsed according to grammar rules.

Step 4: AST generation

Clang builds an **Abstract Syntax Tree**:

```
FunctionDecl add
├── ParmVarDecl a
├── ParmVarDecl b
└── CompoundStmt
    └── ReturnStmt
        └── BinaryOperator '+'
            ├── DeclRefExpr a
            └── DeclRefExpr b
```

Step 5: Semantic analysis

Clang checks: - Are variables declared? - Are types correct? - Is the operation valid?

Step 6: Output

Depending on the tool: - Compiler: AST → LLVM IR → Machine code - Analysis tools: AST → checks → warnings

---

3. What is an AST (Abstract Syntax Tree)?

An AST is a **tree representation of source code structure**, not text.

Important points: - Formatting and comments are removed - Variable names are secondary - Structure and node types matter

Example:

```
a + b;
c + d;
```

Both produce the **same AST structure**.

---

4. What is Clang-Tidy?

Clang-tidy is a **static analysis and linting tool** built on top of Clang.

What clang-tidy does

- Uses Clang to build the AST
- Runs **checks** on the AST
- Reports warnings and suggestions

- Can automatically fix some issues

Typical uses

- Coding style enforcement
- Bug detection
- Performance improvements
- MISRA / CERT / coding guidelines

---

5. How Clang-Tidy Works Internally

High-level flow:

```
Source Code
   ↓
Clang Frontend
   ↓
AST
   ↓
Clang-Tidy Checks
   ↓
Warnings / Fixes
```

Important point:

> Clang-tidy **does not parse code itself**. It relies completely on Clang to build the AST.

---

6. Structure of Clang-Tidy Source Code

Inside LLVM source tree:

```
clang-tools-extra/clang-tidy/
├── ClangTidyCheck.h
├── ClangTidyCheck.cpp
├── ClangTidyModule.h
├── ClangTidyModule.cpp
├── tool/clang-tidy-main.cpp
└── checks/
```

Each rule in clang-tidy is implemented as a **ClangTidyCheck**.

---

7. Basic Clang-Tidy Check Structure

A clang-tidy check has two important functions:

```
class MyCheck : public ClangTidyCheck {
public:
  void registerMatchers(MatchFinder *Finder) override;
  void check(const MatchFinder::MatchResult &Result) override;
};
```

- `registerMatchers()` → describes **what AST pattern to look for**
- `check()` → runs when the pattern is found

---

8. Example: Clang-Tidy Check (Detect goto)

Check Header

```
class AvoidGotoCheck : public ClangTidyCheck {
public:
  AvoidGotoCheck(StringRef Name, ClangTidyContext *Context)
      : ClangTidyCheck(Name, Context) {}

  void registerMatchers(ast_matchers::MatchFinder *Finder) override;
  void check(const ast_matchers::MatchFinder::MatchResult &Result) override;
};
```

Check Implementation

```
void AvoidGotoCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(gotoStmt().bind("goto"), this);
}

void AvoidGotoCheck::check(const MatchFinder::MatchResult &Result) {
  const auto *GS = Result.Nodes.getNodeAs<GotoStmt>("goto");
  if (GS)
    diag(GS->getBeginLoc(), "avoid using goto");
}
```

---

9. How AST Conversion Happens in Clang-Tidy

Important clarification:

> In clang-tidy code, **you never write code to convert source to AST**.

Clang frontend automatically: - Parses source code - Builds AST - Passes AST to clang-tidy

You can **see** the AST using:

```
clang -Xclang -ast-dump -fsyntax-only file.c
```

---

10. Example: Convert Code to AST

Input code

```
int main() {
    int x = 1 + 2;
    return x;
}
```

AST (simplified)

```
FunctionDecl main
└── CompoundStmt
    ├── VarDecl x
    │   └── BinaryOperator '+'
    │       ├── IntegerLiteral 1
    │       └── IntegerLiteral 2
    └── ReturnStmt
        └── DeclRefExpr x
```

---

11. Comparing Two ASTs

Code A

```
a + b;
```

AST:

```
BinaryOperator '+'
├── DeclRefExpr
└── DeclRefExpr
```

Code B

```
c + d;
```

AST:

```
BinaryOperator '+'
├── DeclRefExpr
└── DeclRefExpr
```

Comparison result

- Structure: SAME
- Operator: SAME
- Node types: SAME

✅ ASTs match structurally

Non-matching example

```
a - b;
```

AST:

```
BinaryOperator '-'
```

❌ Does not match `+` AST

---

12. How Clang-Tidy Compares ASTs

Clang-tidy **does not compare full AST trees**.

Instead it: - Defines a **pattern AST** (matcher) - Searches for that pattern inside the full AST

Example matcher:

```
binaryOperator(hasOperatorName("+"))
```

---

13. Implementing and Running Clang-Tidy Checks

Without LLVM source

- You can run existing clang-tidy checks
- You cannot add new checks

With LLVM source

Steps: 1. Clone llvm-project 2. Add check under clang-tools-extra/clang-tidy 3. Register the module 4. Build clang-tidy 5. Run your custom check

---

14. Summary

- Clang converts source code into an AST
- AST is the core representation
- Clang-tidy analyzes the AST
- Clang-tidy checks are pattern matches on AST
- AST comparison is structural, not textual

---

15. One-Line Takeaway

> **Clang understands code, Clang-tidy reasons about code using the AST**