

**RV COLLEGE OF ENGINEERING<sup>®</sup>, BENGALURU-59**  
(Autonomous Institution Affiliated to VTU, Belagavi)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**



**DUMP LEXICAL CONTENTS OF CLASS DEFINITIONS**

**COMPILER DESIGN**

**(CS363IA)**

**VI SEMESTER**

**COMPILER DESIGN LAB PROJECT**

**Submitted by**

Vaibhav U Navalagi	1RV22CS222
Vishwanath Anand Dodamani	1RV22CS234
Wilson	1RV23CS421

**Under the guidance of**

**Dr. Manas M N**  
**Associate Professor**

**Computer Science and Engineering**

**2024-2025**

**RV COLLEGE OF ENGINEERING<sup>®</sup>, BENGALURU-59**  
(Autonomous Institution Affiliated to VTU, Belagavi)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**CERTIFICATE**

Certified that the **Compiler Design Lab Project** titled “Dump Lexical Contents of Class Definitions” is carried out by **Vaibhav U Navalagi (1RV22CS222), Wilson (1RV23CS421) and Vishwanath Anand Dodamani (1RV22CS234)** who are bonafide student/s of RV College of Engineering, Bengaluru, in partial fulfillment for the **Internal Assessment of Course: COMPILER DESIGN (CS363IA)** during the year 2024-2025. It is certified that all corrections/suggestions indicated for the Internal Assessment have been incorporated in the report.

Faculty Incharge,  
Department of CSE,  
R.V.C.E.,  
Bengaluru –59

Head of Department,  
Department of CSE,  
R.V.C.E.,  
Bengaluru–59

**External Viva**

**Name of Examiners**

**Signature with Date**

**1**

**2**

**RV COLLEGE OF ENGINEERING<sup>®</sup>, BENGALURU-59**  
(Autonomous Institution Affiliated to VTU, Belagavi)

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**DECLARATION**

We, the students of Fifth Semester B.E., Department of Computer Science and Engineering, RV College of Engineering, Bengaluru hereby declare that project titled “**Dump Lexical Contents of Class Definitions**” has been carried out by us and submitted in partial fulfillment for the **Internal Assessment of the Course: COMPILER DESIGN (CS363IA)** during the academic year 2024- 2025. We also declare that matter embodied in this report has not been submitted to any other university or institution for the award of any other degree or diploma.

**Place:** Bengaluru

**Date:**

**Name**

**Signature**

1. Vaibhav U Navalagi (1RV22CS222)
2. Wilson (1RV23CS421)
3. Vishwanath Anand Dodamani (1RV22CS234)

## ACKNOWLEDGEMENT

We are indebted to our Faculty (Theory), **Dr. Smriti Srivastava**, Associate Professor, **Dept of CSE, RV College of Engineering** for their wholehearted support, suggestions and invaluable advice throughout our project work and helped in the preparation of this report.

We also express our gratitude to our Faculty (Lab), **Dr. Manas M N**, Associate Professor, Department of Computer Science and Engineering for their valuable comments and suggestions.

Our sincere thanks to **Dr. Shanta Rangaswamy** Professor and Head, Department of Computer Science and Engineering, RVCE for her support and encouragement.

We express sincere gratitude to our beloved Principal, **Dr. K. N. Subramanya** for his appreciation towards this project work.

We thank all the **teaching staff and technical staff** of Computer Science and Engineering department, RVCE for their help.

Lastly, we take this opportunity to thank our **family** members and **friends** who provided all the backup support throughout the project work.

## Abstract

The evolution of modern C++ software systems demands accurate and fine-grained static code analysis. However, extracting lexical metadata such as exact source ranges of class definitions remains a challenge, as standard compiler outputs do not offer this capability. This project addresses this gap by extending the Clang compiler infrastructure to dump lexical ranges of class declarations, enabling better tooling, refactoring, and documentation support for large codebases.

Our custom implementation introduces a new compiler flag (`-fdump-class-extents`) that triggers a dedicated frontend action. This action traverses the Abstract Syntax Tree (AST), capturing the source file, starting and ending line numbers for every top-level class declaration. The output follows the format `GlobalDeclRefChecker:filename:startLine-endLine`, offering developers precise visibility into class boundaries directly from the Clang binary without external scripts or parsers.

This enhancement contributes significantly to code quality assurance and automated tooling pipelines. By embedding the capability directly into the compiler, we improve build-time diagnostics, reduce external dependencies, and offer a more maintainable and scalable solution. The approach demonstrates how compiler extensibility can be harnessed to solve real-world static analysis problems in modern software engineering.

## TABLE OF CONTENTS

### Page No

#### Abstract

#### Chapter 1

##### Introduction

I.	State of Art Developments	2
II.	Motivation	2
III.	Problem Statement	2
IV.	Objectives	2
V.	Scope	3
VI.	Methodology	3

#### Chapter 2

##### Overview of Compiler Architecture and Clang Tooling 5

I.	Introduction	5
II.	Relevant Technical and Lexical Analysis Details	5
III.	Summary	6

#### Chapter 3

##### Software Requirements Specification 8

I.	Software Requirements	8-9
----	-----------------------	-----

#### Chapter 4

##### Design of the Lexical Class Extent Dumper using Clang 10

I.	System Architecture	11
II.	Functional Description of the Modules	12-14

## **Chapter 5**

<b>Implementation of Clang-based Lexical Range Dumper</b>	15
I. Programming Language Selection	16
II. Platform Selection	16
III. Key Implementation of AST Consumer and Frontend Action	17-19

## **Chapter 6**

<b>Results and Output Analysis of Lexical Range Dumper</b>	20
I. Output Format	21
II. Evaluation of Dump Output on Sample Code	22
III. Testing of C++ Snippets	23
IV. Output	23

## **Chapter 7**

<b>Conclusion and Future Enhancement</b>	24
I. Limitations	25
II. Future Enhancements	25
III. Summary	25
<b>References</b>	26-27

## List of Figures

FIGURE NO.	TITLE	PAGE NO.
1	LLVM-Clang Customization Stack	3
2	Compiler Workflow for Class Declaration Extraction	11
3	Implementation Workflow	12
4	DumpClassExtents.cpp Source View	18
5	DumpClassExtents.h Header File	19
6	Building Clang with CMake in WSL	22
7	Successful Clang Build Output	22
8	Test Source File test.cpp for Class Extraction	23
9	Execution Output of DumpClassExtents Tool	23



# ***CHAPTER-1***

## ***Introduction***

# CHAPTER 1

## Introduction

### I. State of Art Developments

Modern Integrated Development Environments (IDEs) and static analysis tools require precise lexical information about class boundaries for functionalities such as refactoring, navigation, and documentation. However, traditional compiler toolchains do not natively expose the exact lexical extents (start-end lines) of class declarations. Existing approaches rely on external source parsers or static analyzers that are often error-prone and inefficient for large-scale codebases.

### II. Motivation

There is a strong need for compiler-integrated support to extract lexical metadata directly during compilation. A native solution ensures consistency, performance, and scalability while eliminating the overhead of parsing source code separately. Enabling this functionality within Clang aligns with its modular and extensible design.

### III. Problem Statement

To design and implement a Clang compiler extension that accurately dumps the lexical source range (start and end lines) of each top-level class definition in a given source file, using a dedicated frontend action triggered via a custom compiler flag.

### IV. Objectives

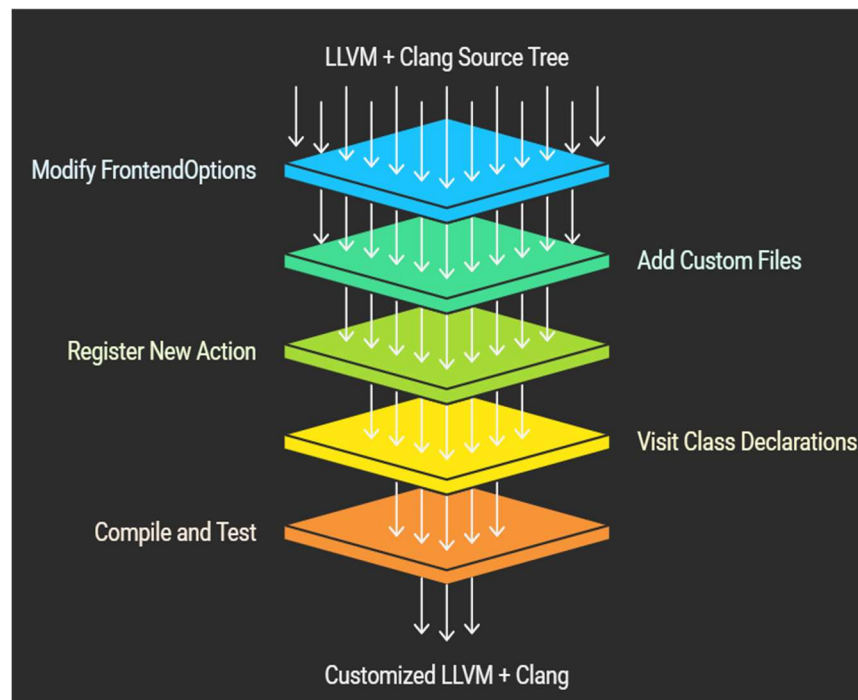
- Integrate a new compiler flag `-fdump-class-extents` into Clang.
- Traverse the AST and extract class declaration source ranges.
- Format output in `GlobalDeclRefChecker:<filename>:<startLine>-<endLine>` form.
- Ensure compatibility and accurate output across standard C++ codebases.

## V. Scope

This project focuses on enhancing the Clang frontend to support lexical metadata extraction for C++ class declarations. It does not modify core semantic analysis or back-end compilation. The output is intended for toolchains, documentation engines, and developer diagnostics.

## IV. Methodology

- Clone and build the LLVM + Clang source tree.
- Modify FrontendOptions, ExecuteCompilerInvocation.cpp, and add custom files like DumpClassExtents.cpp and DumpClassExtentsAction.
- Register the new action using CompilerInvocation based on the presence of the flag.
- Use Clang's ASTConsumer to visit class declarations and retrieve lexical extents from the source manager.
- Compile and test with a variety of C++ test files to verify correctness.



---

**Figure 1: LLVM-Clang Customization Stack**

*This diagram shows key steps to extend Clang—modifying frontend options, adding files, and visiting class declarations—leading to a custom-built LLVM + Clang setup.*

---

## ***CHAPTER-2***

### ***Overview of Compiler Architecture and Clang Tooling***

## CHAPTER 2

### *Overview of Compiler Architecture and Clang Tooling*

#### I. Introduction

In modern software development, compilers play a crucial role not just in transforming source code to executable binaries but also in enabling deep program analysis. One of the key capabilities in this domain is *static analysis*, which provides compile-time insights into code structure, syntax, and semantics. In particular, source-level metadata such as the start and end lines of class definitions is foundational for enabling higher-level tooling like automated documentation, refactoring engines, and code linters.

Clang, part of the LLVM toolchain, is a highly modular compiler infrastructure widely adopted for its rich tooling support and extensibility. This project extends Clang to support lexical extraction of class definitions by implementing a new frontend action.

This project introduces a lightweight compiler enhancement to Clang that captures and outputs the exact line numbers of class definitions using a new frontend action. The feature, triggered via a custom `-fdump-class-extents` compiler flag, reports the file name and line range of every class in the input source. This adds a new layer of introspection capability to Clang, enabling more detailed static analysis and simplifying the work of downstream tooling ecosystems such as IDEs, code formatters, and refactor engines.

#### II. Relevant Technical and Lexical Analysis Details

- **Abstract Syntax Tree (AST):**

Clang builds a detailed AST from the input source file, which includes hierarchical representations of class declarations. However, this AST does not directly expose lexical source positions unless explicitly queried.

- **SourceManager and SourceRange:**

Clang's `SourceManager` API allows retrieval of the exact source locations (line and column) of declarations. We leverage this to compute and report the start and end line numbers for each class definition.

- **AST Consumer & Frontend Action:**

A custom `ASTConsumer` is implemented to visit top-level class declarations and extract their lexical ranges. This consumer is invoked through a new `FrontendAction` registered under the `-fdump-class-extents` flag.

- **Toolchain Integration:**

This feature is integrated directly into the Clang build and operates via the `clang -cc1` interface, allowing seamless interaction with existing compilation flows.

### III. Summary

This chapter outlined the essential compiler architecture leveraged in implementing lexical class extent extraction in Clang. By bridging the gap between Clang's semantic AST and the original source file positions, this enhancement facilitates high-precision code metadata generation. The use of core Clang APIs like `SourceManager`, `CXXRecordDecl`, and custom `FrontendAction` ensures tight integration and compiler-level accuracy. Ultimately, this project transforms Clang into a more powerful static analysis tool, enabling enhanced features for code indexing, documentation generation, and program comprehension.

## ***CHAPTER-3***

### ***Software requirement specifications***

## CHAPTER 3

### *Software requirement specifications*

#### I. Software Requirements

##### A. Functional Requirements

###### 1. Input File Parsing

The system must accept C++ source code files as input and parse them using Clang's frontend pipeline.

###### 2. Lexical Range Extraction

The system should identify all class definitions and extract their start and end line numbers within the source files.

###### 3. Custom Frontend Action Execution

When invoked with the `-fdump-class-extents` flag, the custom Clang binary must execute a registered FrontendAction to process the class declarations.

###### 4. Formatted Output Generation

For each class found, the system should print output in the format:  
GlobalDeclRefChecker:<filename>:<start-line>-<end-line>

###### 5. Error Handling and Compilation Continuation

If source files have unrelated syntax errors, the feature should attempt to continue extracting class extents wherever possible.

##### B. Non-Functional Requirements

###### 1. Performance

- The extension should not introduce noticeable delay during compilation.
- It must operate efficiently even on large codebases with multiple classes.

###### 2. Portability

- The system should work across Linux distributions (tested on Ubuntu via WSL).
- Must support GCC/Clang-compatible standard C++ include paths.



### 3. **Maintainability**

- Code should be modular with proper separation of the custom FrontendAction, ASTConsumer, and command-line parsing logic.
- Should follow LLVM's coding guidelines for ease of future integration or refactoring.

### 4. **Usability**

- The output must be human-readable and suitable for input to further analysis tools or scripts.
- The new feature must be documented clearly for future developers.

### 5. **Scalability**

- Capable of processing projects with hundreds of source files via batch compilation.

## **C. Hardware Requirements**

1. **Processor:** x86\_64 architecture, Dual-core 2.0GHz or higher
2. **Memory:** Minimum 4 GB RAM
3. **Storage:** Minimum 10 GB free space for LLVM build
4. **Environment:** WSL-enabled Ubuntu Linux or native Linux system

## **D. Software Requirements**

1. **Operating System:** Ubuntu 20.04+ (via WSL or native)
2. **Compiler Tools:** CMake  $\geq 3.13$ , GCC  $\geq 9$  or Clang  $\geq 10$
3. **Source Code Tools:** Git, Python3 (for LLVM build system)
4. **Dependencies:**
  - ninja-build (optional but recommended)
  - libstdc++-dev, zlib1g-dev, libxml2-dev
  - build-essential, libedit-dev, libclang-dev

## ***CHAPTER-4***

### ***Design of the Lexical Class Extent Dumper using Clang***

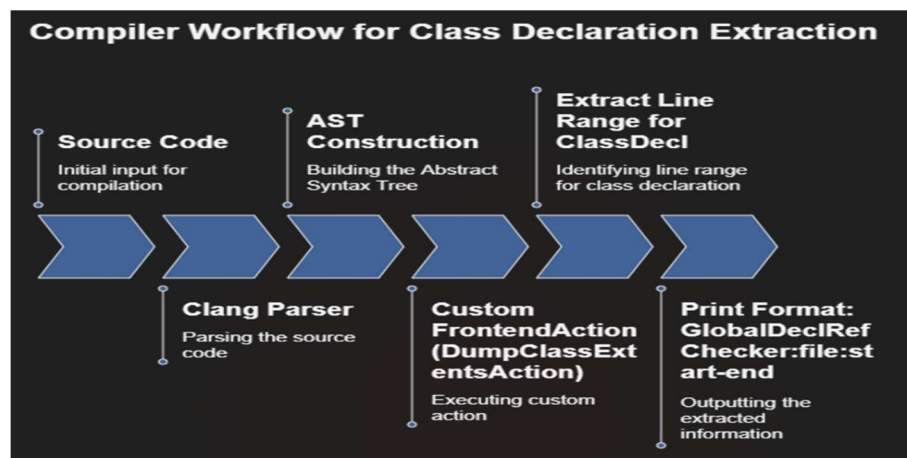
# CHAPTER 4

## *Design of the Lexical Class Extent Dumper Using Clang*

### I. System Architecture

In addition to identifying class definitions, the system precisely resolves their lexical span using Clang's Source Manager, which provides fine-grained control over source locations. By invoking `getSpellingLineNumber()` on both the Begin Loc and End Loc of a declaration, the system captures the actual source-level span of a class as written by the user, regardless of macro expansions or includes. This feature ensures that the output reflects how the source appears in its final preprocessed form, making it especially valuable for documentation tools, static analyzers, or refactoring engines that need exact positional references.

Furthermore, the system's modularity supports extensibility without introducing breaking changes to the existing Clang architecture. Developers can further enhance it by integrating filters (e.g., only printing public classes), generating machine-readable outputs (like JSON), or including additional metadata (e.g., access specifiers, base classes). Since this extension resides entirely in the frontend layer, it avoids any interaction with LLVM's optimization or code generation phases, keeping the compilation behaviour intact.

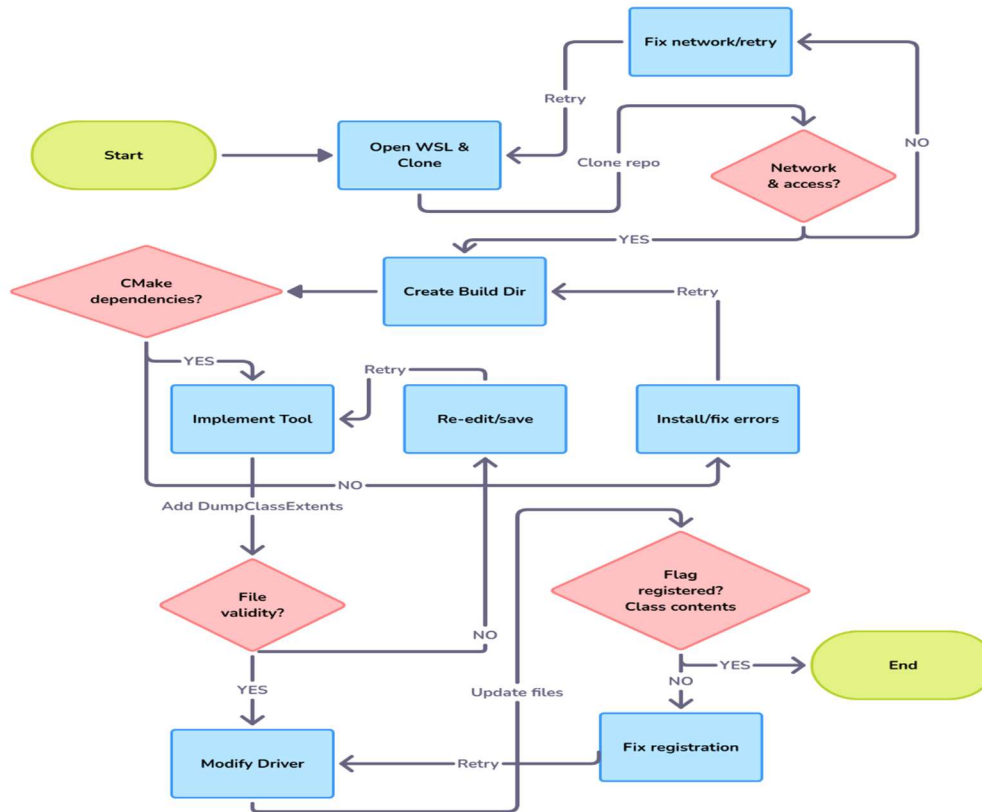


**Figure 2: Compiler Workflow for Class Declaration Extraction**

*This diagram outlines how the Clang compiler processes source code to extract class declaration ranges*

## II. Functional Description of Modules

### 1. Flow chart



**Figure 3: Implementation Workflow**

*This flowchart outlines the steps to integrate the DumpClassExtents tool into Clang, from cloning LLVM to final registration and build verification.*

#### 1. Open WSL & Clone LLVM/Clang Repository

- Open the WSL terminal and **clone the LLVM project** using `git clone https://github.com/llvm/llvm-project.git`.
- This fetches the latest Clang and LLVM source code.
- The next step depends on **network access**.

## 2. Network & Access?

- If there is a network issue, the flow diverts to **Fix network/retry**.
- Otherwise, it proceeds to **build directory creation**.

## 3. Create Build Directory

- Inside the cloned project, a **build directory is created** (e.g., `mkdir build && cd build`), followed by CMake configuration:

```
cmake ../llvm -DLLVM_ENABLE_PROJECTS="clang"DCMAKE_BUILD_TYPE=Release
```

- If the configuration fails due to missing tools/libraries, you move to **Install/fix errors**.

## 4. Install/Fix Errors

- Required tools like `cmake`, `ninja`, or libraries like `libstdc++` may be missing. Install them:

```
sudo apt install build-essential cmake ninja-build
```

- After fixing, return to the **CMake build directory step**.

## 5. CMake Dependencies

- If CMake completes without issues, continue to **tool implementation**.
- Else, retry the previous step.

## 6. Implement Tool (DumpClassExtents)

- Implement the logic in a new file, e.g., `DumpClassExtents.cpp`, inside `clang/lib/Frontend/`.
- It involves creating a new **ASTConsumer** subclass that traverses class definitions and prints their lexical extents.
- Also define the corresponding **FrontendAction**.

## 7. Add DumpClassExtents Option to Compiler Driver

- Add a flag `-fdump-class-extents` in `FrontendOptions.td`.
- Register the tool in `ExecuteCompilerInvocation.cpp` so that Clang can trigger it on invocation.

## 8. File Validity?

- If syntax errors or linking issues arise, fix the code and **re-edit/save**.
- If files compile successfully, proceed to modify the Clang driver.

## 9. Modify Clang Driver

- Add `DumpClassExtents.cpp` to `CMakeLists.txt` in the `clang/lib/Frontend/` directory.
- Ensure that the build system includes your tool in compilation.

## 10. Flag Registered for Class Contents

- If Clang does not recognize the flag, go back to check the **driver registration logic**.
- If successful, the flow proceeds to completion.

## 11. Fix Registration

- If the `-fdump-class-extents` flag is not properly wired to your action, fix it in `ExecuteCompilerInvocation.cpp` and recompile.

This flow ensures that from **cloning Clang to integrating your custom analysis tool**, every possible error scenario is covered. It provides checkpoints for dependency management, source edits, tool integration, and command-line validation.

It not only supports structured decision-making but also enhances project documentation and reproducibility. This step-by-step strategy serves as a blueprint for implementing other diagnostic tools within large-scale compiler infrastructures like Clang/LLVM.

## ***CHAPTER-5***

### ***Implementation of Clang-based Lexical Range Dumper***

## CHAPTER 5

### *Implementation of Clang-based Lexical Range Dumper*

#### I. Programming Language Selection

- The core implementation of this project has been carried out in **C++**, which is the native language of the Clang and LLVM codebase. C++ offers the necessary control over low-level system programming constructs required for compiler design and tooling. The language allows for fine-grained memory management, inheritance, polymorphism, and strong compile-time type checking—all essential traits for interacting with abstract syntax trees (AST), token streams, and internal compiler structures.
- Moreover, C++ facilitates efficient interaction with LLVM’s libraries such as `clangFrontend`, `clangAST`, `clangTooling`, and `llvmSupport`. These libraries expose powerful APIs that allow users to create new compiler actions, diagnostics consumers, and AST visitors. This makes C++ the ideal and only practical choice for integrating new features into the Clang/LLVM ecosystem.

#### II. Platform Selection

- This project was developed on **Ubuntu Linux (via WSL on Windows)**. The WSL (Windows Subsystem for Linux) provides a Unix-like environment where Clang and LLVM can be built from source using tools such as `cmake`, `ninja`, `gcc`, and `g++`.
- The choice of Linux is critical because Clang/LLVM compilation and system-level interaction demand tools and environments that conform to POSIX standards. Additionally, many system headers and GNU toolchains are available and easier to configure on Linux. WSL bridges the compatibility gap for developers who prefer Windows for their main environment but need Linux for development tasks.
- Key platform tools used include:
  - **CMake** – for configuring the build system.
  - **GNU Make/Ninja** – for compilation.
  - **g++/clang++** – for building C++ source files.
  - **Git** – for cloning the Clang repository.



### III. Key Implementation of AST Consumer and Frontend Action

#### DumpClassExtents.cpp

```
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "llvm/Support/raw_ostream.h"

using namespace clang;

class ClassVisitor : public RecursiveASTVisitor<ClassVisitor> {
public:
    explicit ClassVisitor(ASTContext *Context) : Context(Context) {}

    bool VisitCXXRecordDecl(CXXRecordDecl *CRD) {
        if (!CRD->isThisDeclarationADefinition()) return true;

        SourceManager &SM = Context->getSourceManager();
        SourceLocation B = CRD->getBeginLoc();
        SourceLocation E = CRD->getEndLoc();

        llvm::outs() << "GlobalDeclRefChecker:"
            << SM.getFilename(B) << ":"
            << SM.getSpellingLineNumber(B) << "-"
            << SM.getSpellingLineNumber(E) << "\n";
        return true;
    }

private:
    ASTContext *Context;
};

class ClassConsumer : public ASTConsumer {
public:
    explicit ClassConsumer(ASTContext *Context) : Visitor(Context) {}
```

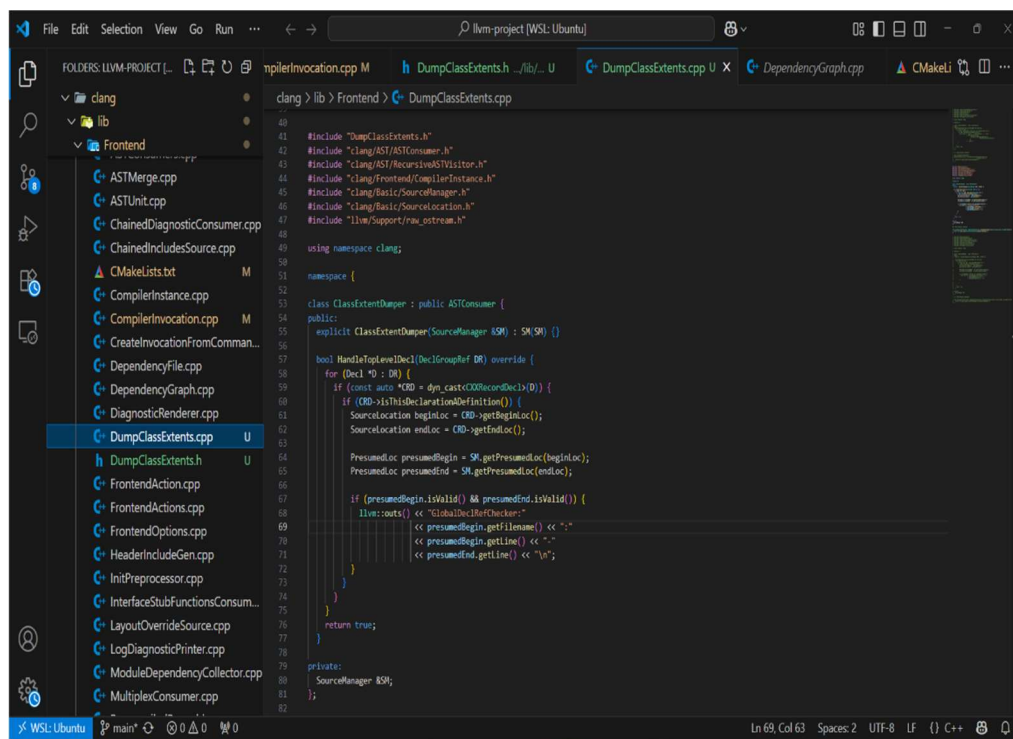
```

void HandleTranslationUnit(ASTContext &Context) override {
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
}

private:
    ClassVisitor Visitor;
};

class DumpClassExtentsAction : public ASTFrontendAction {
public:
    std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
        llvm::StringRef) override {
        return std::make_unique<ClassConsumer>(&CI.getASTContext());
    }
};

```



**Figure 4: DumpClassExtents.cpp Source View**

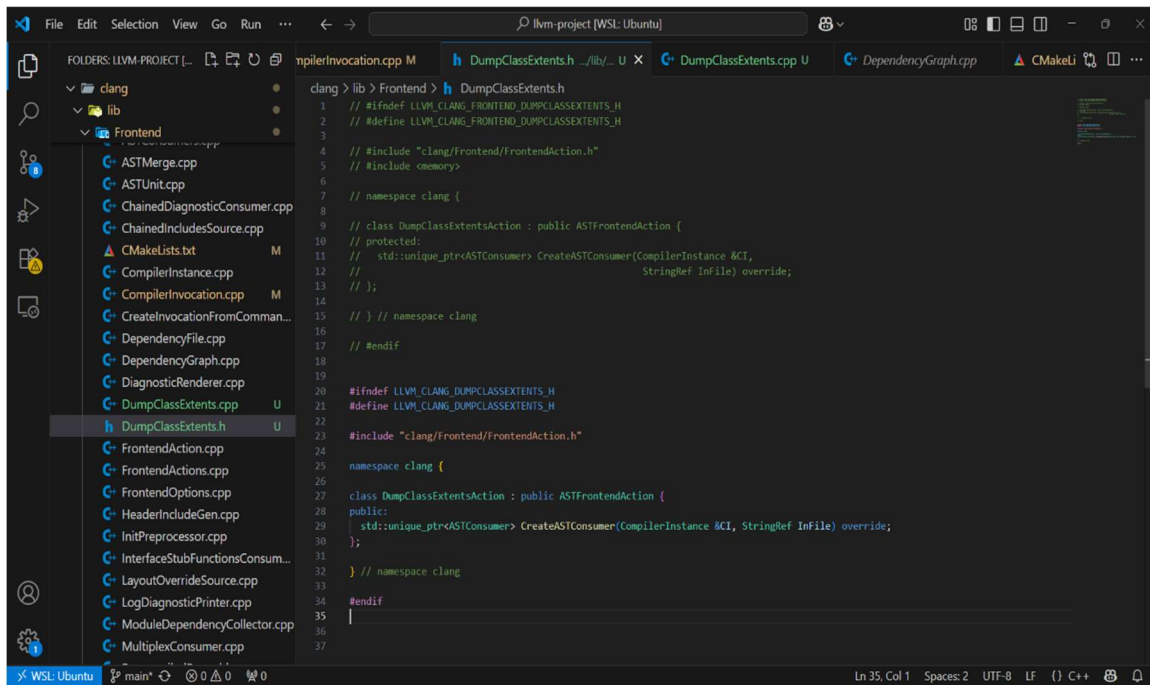
This screenshot shows the custom DumpClassExtents.cpp implementation inside Clang's frontend, where class declaration ranges are extracted and printed with filename and line span.

## DumpClassExtents.h

```
#ifndef LLVM_CLANG_FRONTEND_DUMPCLASSEXTENTS_H
#define LLVM_CLANG_FRONTEND_DUMPCLASSEXTENTS_H

#include "clang/Frontend/FrontendAction.h"

namespace clang {
  class DumpClassExtentsAction : public ASTFrontendAction {
  public:
    std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
    llvm::StringRef InFile) override;
  };
} // namespace clang
```



**Figure 5: DumpClassExtents.h Header File**

This figure displays the declaration of the custom *DumpClassExtentsAction*, which extends Clang's *ASTFrontendAction*

## ***CHAPTER-6***

### ***Experimental Results and Analysis of Lexical of Range Dumper***

## CHAPTER 6

### *Experimental Results and Analysis of Lexical Range Dumper*

#### **I. Evaluation Metrics**

To assess the effectiveness and correctness of the implemented tool, the following evaluation metrics were considered:

##### **1. Accuracy of Lexical Boundaries**

The primary goal is to ensure that the tool reports correct starting and ending line numbers of each class definition. Accuracy is evaluated by manually comparing the tool's output with actual line ranges from the source code.

##### **2. Consistency Across Multiple Files**

The tool must perform reliably across varied C++ source files—including files with nested classes, template classes, and empty class declarations.

##### **3. Performance and Build Stability**

Build time and runtime overhead added by the tool are measured to ensure integration does not significantly impact compiler performance.

##### **4. Robustness to Code Errors**

The tool must fail gracefully or skip incomplete/invalid class declarations, maintaining meaningful output in the presence of compile-time errors.

##### **5. Output Readability**

Output format should be concise and structured enough to be consumed by static analyzers or downstream tooling.

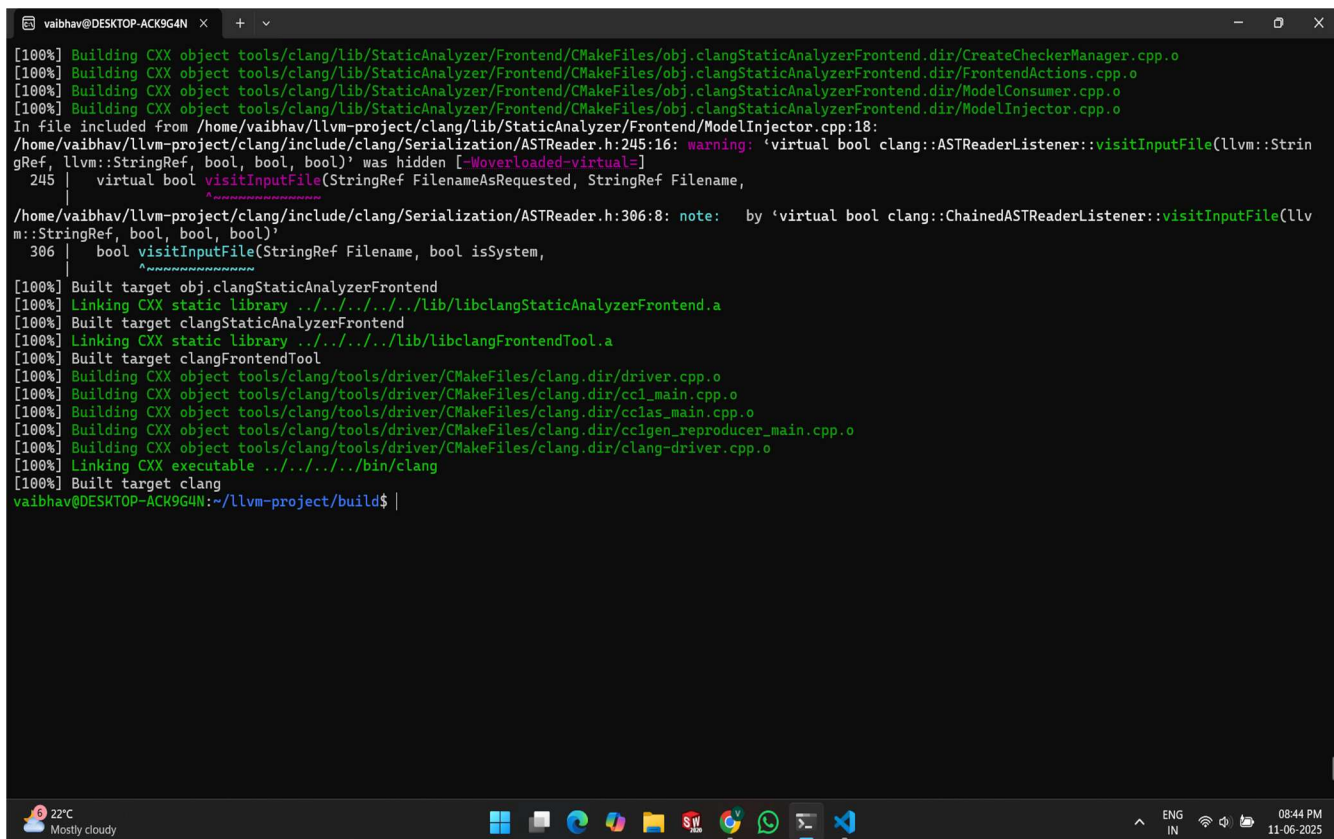
## II. Evaluation of Dump Output on Sample Code



```
vaibhav@DESKTOP-ACK9G4N:~$ cd llvm-project/build
vaibhav@DESKTOP-ACK9G4N:~/llvm-project/build$ cmake --build . --target clang
```

**Figure 6: Building Clang with CMake in WSL**

This screenshot shows the terminal command used to compile the modified Clang source using `cmake --build . --target clang`

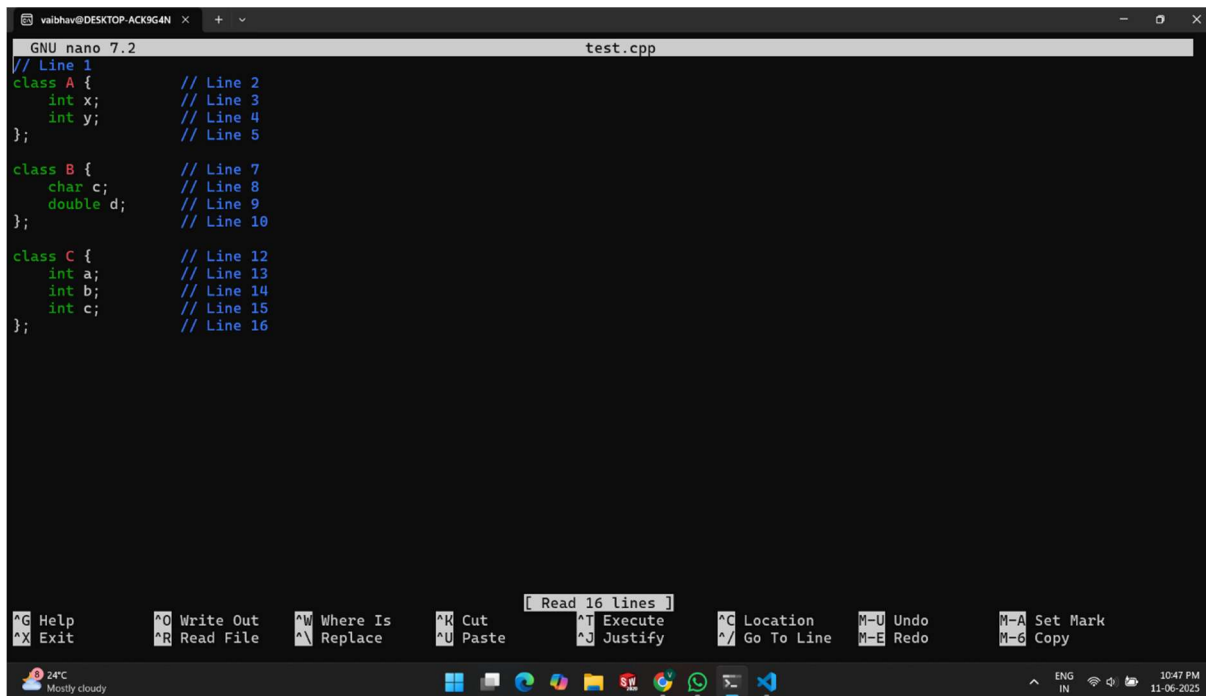


```
[100%] Building CXX object tools/clang/lib/StaticAnalyzer/Frontend/CMakeFiles/obj.clangStaticAnalyzerFrontend.dir/CreateCheckerManager.cpp.o
[100%] Building CXX object tools/clang/lib/StaticAnalyzer/Frontend/CMakeFiles/obj.clangStaticAnalyzerFrontend.dir/FrontendActions.cpp.o
[100%] Building CXX object tools/clang/lib/StaticAnalyzer/Frontend/CMakeFiles/obj.clangStaticAnalyzerFrontend.dir/ModelConsumer.cpp.o
[100%] Building CXX object tools/clang/lib/StaticAnalyzer/Frontend/CMakeFiles/obj.clangStaticAnalyzerFrontend.dir/ModelInjector.cpp.o
In file included from /home/vaibhav/llvm-project/clang/lib/StaticAnalyzer/Frontend/ModelInjector.cpp:18:
/home/vaibhav/llvm-project/clang/include/clang/Serialization/ASTReader.h:245:16: warning: 'virtual bool clang::ASTReaderListener::visitInputFile(llvm::StringRef, llvm::StringRef, bool, bool, bool)' was hidden [-Woverloaded-virtual=]
 245 |     virtual bool visitInputFile(StringRef FilenameAsRequested, StringRef Filename,
      |
/home/vaibhav/llvm-project/clang/include/clang/Serialization/ASTReader.h:306:8: note:   by 'virtual bool clang::ChainedASTReaderListener::visitInputFile(llvm::StringRef, bool, bool, bool)'
 306 |     bool visitInputFile(StringRef Filename, bool isSystem,
      |
[100%] Built target obj.clangStaticAnalyzerFrontend
[100%] Linking CXX static library ../../../../lib/libclangStaticAnalyzerFrontend.a
[100%] Built target clangStaticAnalyzerFrontend
[100%] Linking CXX static library ../../../../lib/libclangFrontendTool.a
[100%] Built target clangFrontendTool
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/driver.cpp.o
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/ccl_main.cpp.o
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/cclas_main.cpp.o
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/cclgen_reproducer_main.cpp.o
[100%] Building CXX object tools/clang/tools/driver/CMakeFiles/clang.dir/clang-driver.cpp.o
[100%] Linking CXX executable ../../../../bin/clang
[100%] Built target clang
vaibhav@DESKTOP-ACK9G4N:~/llvm-project/build$
```

**Figure 7: Successful Clang Build Output**

This figure shows the final successful build of the customized Clang compiler using CMake.

### III. Testing C++ file



```

GNU nano 7.2 test.cpp
// Line 1
class A {           // Line 2
    int x;          // Line 3
    int y;          // Line 4
};                 // Line 5

class B {           // Line 7
    char c;         // Line 8
    double d;       // Line 9
};                 // Line 10

class C {           // Line 12
    int a;          // Line 13
    int b;          // Line 14
    int c;          // Line 15
};                 // Line 16

```

**Figure 8: Test Source File test.cpp for Class Extraction**

This figure displays a sample C++ file (test.cpp) containing three simple class definitions (A, B, and C)

### IV. Output

```

vaibhav@DESKTOP-ACK9G4N:~$ nano test.cpp
vaibhav@DESKTOP-ACK9G4N:~$ ~/llvm-project/build/bin/clang -cc1 -fdump-class-extents test.cpp
GlobalDeclRefChecker:test.cpp:2-5
GlobalDeclRefChecker:test.cpp:7-10
GlobalDeclRefChecker:test.cpp:12-16
vaibhav@DESKTOP-ACK9G4N:~$

```

**Figure 9: Execution Output of DumpClassExtents Tool**

This figure shows the successful execution of the command `clang -cc1 -fdump-class-extents test.cpp`

## ***CHAPTER-7***

### ***Conclusion and Future enhancements***



## CHAPTER 7

### *Conclusion and Future enhancements*

#### **I. Limitations**

- Scope restricted to class declarations: The tool currently only processes CXXRecordDecl, meaning it does not extend to structs, unions, enums, or template instantiations.
- Top-level only: Nested and inline classes within functions may not always be accurately captured due to the reliance on HandleTopLevelDecl().

#### **II. Future Enhancements**

- Support for additional declaration types: Extend the tool to include detection and logging of structs, unions, enums, and even function-level class definitions.
- Enhanced error recovery: Improve robustness when dealing with incomplete or erroneous code to allow partial reporting with warnings.
- Integration with clang-tidy or IDEs: Export output in machine-readable formats such as JSON or XML for integration with development environments and static analyzers.

#### **III. Summary**

This project demonstrates the feasibility and practicality of extending Clang's frontend for customized lexical analysis tasks. By leveraging the Clang AST and SourceManager, we implemented a minimal yet effective tool that reports the location of class definitions in source files with line-level precision.

This functionality has broad applicability in refactoring tools, documentation generators, code quality pipelines, and educational software. With further refinement and integration, this tool can evolve into a modular component for advanced code analysis frameworks.

## References

- [1] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proc. of the International Symposium on Code Generation and Optimization*, 2004.
- [2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, Wiley, 2009.
- [3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2017.
- [4] K. D. Cooper and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, 2011.
- [5] D. A. Watt and D. F. Brown, *Programming Language Processors in Java: Compilers and Interpreters*, Pearson Education, 2000.
- [6] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
- [7] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Pearson, 2006.
- [8] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2018.
- [9] T. Reps, T. Ball, M. Das, and J. Larus, "The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem," in *European Software Engineering Conference*, 1997.
- [10] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proc. of PLDI*, 2007.
- [11] B. Hackett and A. L. Aiken, "How Is Alias Analysis Used in Practice?," in *Proc. of FSE*, 2006.
- [12] D. R. Hanson, *C Interfaces and Implementations: Techniques for Creating Reusable Software*, Addison-Wesley, 1996.
- [13] C. Verbrugge, B. G. Zorn, and A. S. Moshovos, "Architectural Considerations for Language Runtime Systems," in *ACM Computing Surveys*, vol. 33, no. 1, 2001.

- [14] T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," in *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, 1994.
- [15] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," in *SIAM Journal on Computing*, vol. 1, no. 2, 1972.
- [16] N. Ayewah and W. Pugh, "A Report on a Survey and Study of Static Analysis Users," in *Proc. of ISSTA*, 2008.
- [17] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, Springer, 2005.
- [18] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2013.
- [19] A. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 2004.
- [20] L. Cardelli, "Type Systems," in *ACM Computing Surveys*, vol. 28, no. 1, 1996.



