On-Demand Parsing in Clang

Contact Details:

Name: Vipul Cariappa
Email: vipulcariappa@gmail.com
Country/Timezone: Switzerland, CEST (UTC+2)
Proposed Start Date: 1st June 2026

Project Overview

A typical C++ translation unit often includes a substantial number of header files, while only a limited subset of the declarations and definitions contained within those headers is actually utilized. Despite this, modern compilers such as Clang eagerly parse the entirety of the included code. This approach leads to unnecessary computational overhead, resulting in increased compilation time and higher memory consumption.

The objective of this project is to design and implement a mechanism within the Clang compiler that defers the parsing of function and class definitions until the point at which they are first required. By introducing this form of on-demand parsing, the compiler can avoid processing unused code paths, thereby improving efficiency.

Technical Approach

Within class or struct definitions, Clang already employs a deferred parsing mechanism for method bodies through the use of `clang::Parser::ParsingClass::LateParsedDeclarations`. This is gone because the

bodies of methods within the class may refer to other methods or attributes that are defined later in the class's body. This project proposes extending a similar strategy to top-level functions and class definitions.

The proposed approach begins at the lexical analysis stage. The tokenizer converts the input character stream into a sequence of tokens, as per the standard compilation pipeline. When the parser encounters a function or class definition, instead of immediately parsing its body, it records the associated token stream, specifically the range between the opening `{` and closing `}` braces, into a dedicated storage structure (e.g., `DefinitionToks`). The parser then treats the construct as a declaration rather than a fully parsed definition.

The identification of matching braces can be efficiently achieved during tokenization in linear time. Later, when the compiler encounters a use of the corresponding entity, it checks whether the stored token stream is available. If so, the parser is invoked at that point to process the deferred definition. Note that this approach requires lexical analysis upfront and only defers the parsing to a later stage.

Challenges and Considerations

Silent Acceptance of Invalid Unused Code

Deferred parsing introduces the possibility that syntactically incorrect code may go undetected if it is never referenced:

```
int foo() {
    return 2 + ; // invalid code
}

int main() {
    return 0; // foo never used
}
```

In this scenario, the compiler would not emit an error for `foo`, as its body is never parsed. While this behavior is consistent with the deferred parsing model, it deviates from standard C++ requirements and must be carefully evaluated.

Impact on Name Lookup Semantics

Deferring parsing may alter the results of name lookup due to changes in the visible scope at the time of parsing:

```
int foo(int) { return 1; }
```

```
int bar() { return foo(0.0); }

int foo(double) { return 2; }

int main() { return bar(); }
```

According to the C++ standard, `main` should return `1`. However, if `bar` is parsed only after both overloads of `foo` are available, overload resolution may yield a different result.

To address this, the proposal introduces source-location-aware name lookup. By restricting lookup to declarations that precede the definition point of the entity being parsed, it is possible to preserve standard-compliant behavior.

Missing Symbols in Object Files

In large-scale projects, especially those involving static/dynamic libraries, many translation units do not contain a `main` function and may not directly reference all defined symbols. As a result, deferred parsing could prevent certain definitions from ever being compiled into object files.

This limitation suggests that the approach is particularly well-suited for `static`, `inline`, and templated entities, where unused definitions are less likely to impact linkage correctness.

JIT-Guided Parsing for Incremental Compilation

For incremental and interactive use cases, such as those supported by `clang-repl`, Just-In-Time (JIT) compilation can be leveraged to guide parsing decisions.

LLVM's Orc JIT infrastructure provides the `llvm::orc::MaterializationUnit` abstraction, which enables lazy generation of `llvm::Module` instances for individual symbols. In this model, each symbol is registered with a callback responsible for generating its corresponding IR only when the symbol is required.

By integrating deferred parsing with this mechanism, it becomes possible to extend laziness beyond code generation to the parsing phase itself. Function bodies would only be parsed when the associated symbol is materialized, aligning parsing behavior with runtime demand.

Tentative Timeline

Month 1: Familiarization
Develop a comprehensive understanding of Clang's frontend architecture, focusing on the lexer and parser components, as well as their interaction.

Month 2: Lexer Enhancements and Lookup Mechanism
Modify the lexical analysis phase to extract and store token streams corresponding to definition bodies. Implement a source location based name lookup mechanism to ensure correctness in deferred parsing scenarios.

Month 3: Deferred Parsing of Function Bodies
Leverage the infrastructure developed in previous stages to enable on-demand parsing of top-level function definitions.

Month 4: Deferred Parsing of Class and Struct Definitions
Extend the deferred parsing mechanism to support class and struct definitions, addressing any additional complexities related to scope and member visibility.

Month 5: Integration with Code Generation and JIT
Study and adapt Clang's CodeGen pipeline for incremental compilation. Integrate Orc JIT's `MaterializationUnit` to enable lazy compilation of symbols, ensuring that parsing and code generation are both demand driven.

Month 6: Testing, Benchmarking, and Validation
Conduct thorough testing and profiling of the prototype. Evaluate improvements in memory usage and compilation time, gather feedback from the community, and assess compliance with C++ standards.

Note: This timeline is indicative and subject to adjustment based on implementation complexity. Each phase will include dedicated time for unit testing and validation.

Expected Outcomes

- A functional prototype integrated into Clang that supports on-demand parsing of top-level functions and class/struct definitions, demonstrating the feasibility of the approach.
- Measurable improvements in compilation performance, including reduced peak memory usage and faster compile times on representative workloads, particularly for large translation units with sparse usage.
- Preservation of correct C++ semantics through mechanisms such as source-location-aware name lookup, ensuring that deferred parsing does not introduce any deviations from standard-compliant behavior.
- A comprehensive testing and benchmarking suite validating correctness, ensuring compatibility with Clang's existing test infrastructure, and demonstrating the practical impact of deferred parsing across real-world scenarios.

Benefits to the Community

This project has the potential to deliver meaningful improvements to the broader C++ and LLVM/Clang ecosystem by significantly reducing compilation time and peak memory usage,

particularly for large-scale codebases that rely heavily on header inclusion. By deferring the parsing of unused entities, developers can benefit from faster build cycles, leading to increased productivity and more efficient use of computational resources. Furthermore, in interactive environments powered by Clang, such as Clang-Repl, header inclusion effectively becomes a near no-op operation under this model, as definitions are only processed when explicitly required. This advancement directly enhances the responsiveness and performance of tools built on top of Clang-Repl, including frameworks such as CERN's ROOT, where rapid iteration and dynamic execution are critical. Overall, the proposed work contributes to a more scalable and efficient compilation model, benefiting both systems programmers and researchers working with complex C++ infrastructures.

Biographical Information

I am a final-year student pursuing a degree in Computer Science and Engineering at Ramaiah University of Applied Sciences, India. I am currently working at CERN, Switzerland, as part of the ROOT team, focusing on interoperability between C/C++ and Python. My work involves extensive use of Clang and Clang-Repl for runtime type introspection and dynamic generation of Python bindings. I have participated twice in Google Summer of Code (GSoC): first with GNU Octave, where I worked on Python interoperability, and subsequently with the Python Software Foundation under the LPython project, contributing to LLVM-based JIT compilation for statically typed Python. Additionally, I am a core developer of the CppInterOp project, which provides a C++ interpreter and reflection capabilities built on top of LLVM and Clang.