



Mid-Term Presentation On-Demand Parsing in Clang

Vipul Cariappa

2026-08-12

Recap: The Project

Problem Statement

- 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.
- Modern compilers eagerly parse the entirety of the included code, resulting in increased compilation time and higher memory consumption.

Proposed Solution

- 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.
- This form of on-demand parsing can avoid processing unused code paths, thereby improving efficiency.

On-Demand Parsing (#210893)

Introduced a new flag `-fparse-functions-ondemand` to defer parsing for function.

```
static void foo() { printf("foo()\n"); }

void main() {
    // foo(); // foo not used
    printf("main()\n");
}
```

```
└─FunctionDecl 0x5647add39aa0 <line:3:1, col:17> col:13 foo 'void ()' static internal-linkage
└─<<<NULL>>>
└─FunctionDecl 0x5647add39b68 <line:4:1, line:7:1> line:4:6 invalid main 'void ()' external-linkage
└─CompoundStmt 0x5647add39dc0 <col:13, line:7:1>
└─CallExpr 0x5647add39d78 <line:6:3, col:20> 'int'
└─ImplicitCastExpr 0x5647add39d60 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
└─DeclRefExpr 0x5647add39d10 <col:3> 'int (const char *, ...)' lvalue Function 0x5647add39880 'printf' 'int (const char *, ...)'
└─ImplicitCastExpr 0x5647add39da8 <col:10> 'const char *' <ArrayToPointerDecay>
└─StringLiteral 0x5647add39cf0 <col:10> 'const char[8]' lvalue "main()\n"
```

Note that the current implementation is designed such that only function with internal-linkage is defer. All other functions are parsed eagerly. This is done to preserve symbols that other Translation Units (TU)s may depend on from this TU.

Example 1: Functions in Anonymous Namespace

```
namespace {
    void foo() { printf("foo()\n"); }
}
```

```
void main() {} // foo unused
```

```
└─NamespaceDecl 0x557db102ca60 <line:3:1, line:5:1> line:3:11 external-linkage
└─FunctionDecl 0x557db102cb80 <line:4:3, col:12> col:8 foo 'void ()' internal-linkage
└─<<<NULL>>>
└─UsingDirectiveDecl 0x557db102cad0 <line:3:11, <invalid sloc>> <invalid sloc> implicit Namespace 0x557db102ca60 <col:1, line:5:1>
└─FunctionDecl 0x557db102cc60 <line:7:1, col:14> col:6 invalid main 'void ()' external-linkage
└─CompoundStmt 0x557db102cd40 <col:13, col:14>
```

Example 2: Templated Functions

```
template <int N>
static void foo() { printf("foo<%d>()\n", N); }

void main() {} // foo unused
```

```
└─FunctionTemplateDecl 0x56263cb9ecc8 <line:3:1, line:4:17> col:13 foo internal-linkage
  └─NonTypeTemplateParmDecl 0x56263cb9eb58 <line:3:11, col:15> col:15 'int' depth 0 index 0 N
    └─FunctionDecl 0x56263cb9ec20 <line:4:1, col:17> col:13 foo 'void ()' static
      └─<<<NULL>>>
    └─FunctionDecl 0x56263cb9edb0 <line:6:1, col:14> col:6 invalid main 'void ()' external-linkage
      └─CompoundStmt 0x56263cb9ee90 <col:13, col:14>
```

Example 3: Methods

```
namespace {  
    struct A {  
        void foo() { printf("A::foo()\n"); }  
        template <int N>  
        void bar() { printf("A::bar<%d>()\n", N); }  
    };  
}  
  
void main() {  
    A a;  
    a.foo(); // A::bar unused  
}
```

Example 3: Methods (AST)

```

NamespaceDecl 0x55fbd9d98ba0 <line:3:1, line:9:1> line:3:11 external-linkage
├─CXXRecordDecl 0x55fbd9d98c68 <line:4:3, line:8:3> line:4:10 referenced struct A definition internal-linkage
│   └─CXXMethodDecl 0x55fbd9d98e90 <line:5:5, col:40> col:10 used foo 'void ()' implicit-inline internal-linkage
│       └─CompoundStmt 0x55fbd9dbbf50 <col:16, col:40>
│           └─CallExpr 0x55fbd9dbbf08 <col:18, col:37> 'int'
│               └─ImplicitCastExpr 0x55fbd9dbbef0 <col:18> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
│                   └─DeclRefExpr 0x55fbd9dbbe98 <col:18> 'int (const char *, ...)' lvalue Function 0x55fbd9d989d0 'printf' 'int (const char *, ...)'
│                       └─ImplicitCastExpr 0x55fbd9dbbf38 <col:25> 'const char *' <ArrayToPointerDecay>
│                           └─StringLiteral 0x55fbd9dbbe70 <col:25> 'const char[10]' lvalue "A::foo()\n"
├─FunctionTemplateDecl 0x55fbd9d99090 <line:6:5, line:7:14> col:10 bar internal-linkage
│   └─NonTypeTemplateParmDecl 0x55fbd9d98f50 <line:6:15, col:19> col:19 'int' depth 0 index 0 N
│       └─CXXMethodDecl 0x55fbd9d98fe8 <line:7:5, col:14> col:10 bar 'void ()' implicit-inline
│           └─<<<NULL>>>
├─FunctionDecl 0x55fbd9d99138 <line:11:1, line:14:1> line:11:6 invalid main 'void ()' external-linkage
│   └─CompoundStmt 0x55fbd9dbbda0 <col:13, line:14:1>
│       └─DeclStmt 0x55fbd9dbbd10 <line:12:3, col:6>
│           └─VarDecl 0x55fbd9d99260 <col:3, col:5> col:5 used a 'A' callinit
│               └─CXXConstructExpr 0x55fbd9dbbce8 <col:5> 'A' 'void () noexcept'
├─CXXMemberCallExpr 0x55fbd9dbbd78 <line:13:3, col:9> 'void'
│   └─MemberExpr 0x55fbd9dbbd48 <col:3, col:5> '<bound member function type>' .foo 0x55fbd9d98e90
│       └─DeclRefExpr 0x55fbd9dbbd28 <col:3> 'A' lvalue Var 0x55fbd9d99260 'a' 'A'

```

Example 4.1: Virtual Methods

```
namespace {
    struct A {
        virtual void foo() { printf("A::foo()\n"); }
        template <int N>
        void bar() { printf("A::bar<%d>()\n", N); }
    };
}

void main() {} // struct A unused
```

```
└─NamespaceDecl 0x5623131deab0 <line:3:1, line:9:1> line:3:11 external-linkage
  └─CXXRecordDecl 0x5623131deb78 <line:4:3, line:8:3> line:4:10 struct A definition internal-linkage
    └─CXXMethodDecl 0x5623131deda0 <line:5:5, col:22> col:18 foo 'void ()' virtual implicit-inline internal-linkage
      └─<<<NULL>>>
    └─FunctionTemplateDecl 0x5623131defa0 <line:6:5, line:7:14> col:10 bar internal-linkage
      └─NonTypeTemplateParmDecl 0x5623131dee60 <line:6:15, col:19> col:19 'int' depth 0 index 0 N
        └─CXXMethodDecl 0x5623131deef8 <line:7:5, col:14> col:10 bar 'void ()' implicit-inline
          └─<<<NULL>>>
  └─FunctionDecl 0x562313200430 <line:11:1, col:14> col:6 invalid main 'void ()' external-linkage
    └─CompoundStmt 0x562313200510 <col:13, col:14>
```

Example 4.2: Virtual Methods

```
namespace {  
    struct A {  
        virtual void foo() { printf("A::foo()\n"); }  
        template <int N>  
        void bar() { printf("A::bar<%d>()\n", N); }  
    };  
}  
  
void main() {  
    A a;  
}
```

Example 4.2: Virtual Methods (AST)

```

- NamespaceDecl 0x55642a01daa0 <line:3:1, line:9:1> line:3:11 external-linkage
  - CXXRecordDecl 0x55642a01db68 <line:4:3, line:8:3> line:4:10 referenced struct A definition internal-linkage
    - CXXMethodDecl 0x55642a01dd90 <line:5:5, col:48> col:18 used foo 'void ()' virtual implicit-inline internal-linkage
      - CompoundStmt 0x55642a03fd30 <col:24, col:48>
        - CallExpr 0x55642a03fce8 <col:26, col:45> 'int'
          - ImplicitCastExpr 0x55642a03fcd0 <col:26> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
            - DeclRefExpr 0x55642a03fc78 <col:26> 'int (const char *, ...)' lvalue Function 0x55642a01d8d0 'printf' 'int (const char *, ...)'
          - ImplicitCastExpr 0x55642a03fd18 <col:33> 'const char *' <ArrayToPointerDecay>
            - StringLiteral 0x55642a03fc50 <col:33> 'const char[10]' lvalue "A::foo()\n"
        - FunctionTemplateDecl 0x55642a01df90 <line:6:5, line:7:14> col:10 bar internal-linkage
          - NonTypeTemplateParmDecl 0x55642a01de50 <line:6:15, col:19> col:19 'int' depth 0 index 0 N
          - CXXMethodDecl 0x55642a01dee8 <line:7:5, col:14> col:10 bar 'void ()' implicit-inline
            - <<<NULL>>>
      - FunctionDecl 0x55642a03f420 <line:11:1, line:13:1> line:11:6 invalid main 'void ()' external-linkage
        - CompoundStmt 0x55642a03fb90 <col:13, line:13:1>
          - DeclStmt 0x55642a03fb78 <line:12:3, col:6>
            - VarDecl 0x55642a03f530 <col:3, col:5> col:5 a 'A' callinit
              - CXXConstructExpr 0x55642a03fb50 <col:5> 'A' 'void () noexcept'

```

If the struct is used all virtual functions are parsed. Because the first field of a the struct is pointer to the vtable, and the vtable needs entries for all the virtual functions.

Lookup Semantic

When parsing a deferred function at a later stage, the lookup table would have entries to things that are defined after it. But these entries should not be used, as it can end up producing non-standard conforming code.

```
static int foo(int) { return 1; }  
static int bar() { return foo(0.0); } // only foo(int) in lookup table  
  
static int foo(double) { return 2; } // should not be used to parse `bar`  
  
int main() { return printf("%d\n", bar()); } // forces parsing of bar here
```

This is done by filtering the lookup results that appear after the declaration of the lookup point.

```
$ clang lookup.cpp -o lookup  
$ ./lookup  
1
```

```
$ clang -fparse-functions-ondemand \  
      lookup.cpp -o lookup  
$ ./lookup  
1
```

Example 1: Class/Structs

```
namespace {
    struct A {
        void foo() {
            printf("A::foo(%d)\n", n); // n is defined after this lookup point
        };
        void bar();
        int n = 10;
    };

    // out-of-line definition
    void A::bar() { printf("A::bar(%d)\n", n); }
}

int main() {
    A a;
    a.foo();
    a.bar();
}
```

Here we cannot filter lookup if `n`. We check if the lookup result is part of the same `RecordDecl` or part of one of the inherited parent.

```
$ clang lookup-class.cpp -o lookup
$ ./lookup
A::foo(10)
A::bar(10)
```

```
$ clang -fparse-functions-ondemand \
    lookup-class.cpp -o lookup
$ ./lookup
A::foo(10)
A::bar(10)
```

Example 2: Lambda Call Operator

```
void callback(std::function<int()> f) {  
    printf("callback::f() = %d\n", f()); // lookup of operator()  
}  
  
int main() {  
    callback([]() { return 1; }); // operator() defined here  
}
```

```
$ clang++ lookup-lambda.cpp -o lookup  
$ ./lookup  
callback::f() = 1
```

```
$ clang++ -fparse-functions-ondemand \  
    lookup-lambda.cpp -o lookup  
$ ./lookup  
callback::f() = 1
```

Example 3: Lookup in template parameter

```
namespace {
template <typename T>
typename T::result call_bar(T &t) { // lookup of T::result
    typename T::result res = t.bar();
    printf("t.bar() = (%d)\n", res);
    return res;
}

struct T {
    using result = int; // T::result defined here
    result bar() { return 0; }
};
} // namespace

int main() {
    T t;
    return call_bar(t);
}
```

We check if we are currently performing template parameter substitution to determine if the lookup result needs to be filtered.

```
$ clang++ lookup-template.cpp -o lookup
$ ./lookup
t.bar() = (0)
```

```
$ clang++ -fparse-functions-ondemand \
    lookup-template.cpp -o lookup
$ ./lookup
t.bar() = (0)
```

Correctness & Benchmarks

Correctness: clang builds with `-fparse-functions-ondemand` enabled, and `check-clang` passes.

Building clang (2480 TUs, `-ftime-trace`, 24 threads):

	Frontend (parse)	Backend (codegen)	clang	check-clang
Off	3331.8 s	3314.9 s	280.37 s	255.93 s
On	3290.4 s	3315.1 s	281.56 s	251.40 s

Roughly **40 s** less parsing time.

Micro-benchmark: Parsing

```
#include <bits/stdc++.h>
int main() {}
```

On-demand	Time
Off	0.7 s
On	0.4 s

Future Work

- Currently we are only deferring the parsing of functions. We are also interested in deferring parsing entire classes/structs.
- Check the implementation on clang-repl, but by not just deferring internal-linkage functions, but all functions.

Thank You

Any questions or comments.