



On-Demand Parsing in Clang

Vipul Cariappa

2026-11-04

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.

Where it stands

- Two flags, `-fparse-functions-ondemand` and `-fparse-records-ondemand`, both restricted to incremental processing mode, i.e. *clang-repl*.

Functions: Repurposing MSVC's Lazy Template Parsing

- Clang already stores a function body as lexical tokens and parses it later with the `-fdelayed-template-parsing` flag for MSVC compatibility.
- We repurpose the same feature, extending it to lazily parse all functions, not just templated ones.
- A function declaration is parsed eagerly, and the declaration is inserted into the lookup table and marked with `Sema::MarkAsLateParsedTemplate`.
- The lexical token stream of the function body is stored without actually parsing it into the AST.
- `Sema::MarkFunctionReferenced` marks the function as needing a definition, and `Sema::ActOnEndOfTranslationUnit` parses it.

Example: A Function Nobody Called

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

bar(); // bar is used here, foo is never used
```

```
|-FunctionDecl 0x561483e125b0 <line:3:1, col:10> col:6 foo 'void ()' external-linkage
| `-<<<NULL>>>
|-FunctionDecl 0x561483e12678 <line:4:1, col:33> col:6 used bar 'void ()' external-linkage
| `-CompoundStmt 0x561483e129d0 <col:12, col:33>
|   `-CallExpr 0x561483e12988 <col:14, col:30> 'int'
|     |-ImplicitCastExpr 0x561483e12970 <col:14> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|     | `-DeclRefExpr 0x561483e12920 <col:14> 'int (const char *, ...)' lvalue Function 0x561483e12390 'printf' 'int (const char *, ...)'
|     `-ImplicitCastExpr 0x561483e129b8 <col:21> 'const char *' <ArrayToPointerDecay>
|       `-StringLiteral 0x561483e12900 <col:21> 'const char[7]' lvalue "bar()\n"
`-TopLevelStmtDecl 0x561483e12768 <line:6:1, col:5> col:1
  `-CallExpr 0x561483e12828 <col:1, col:5> 'void'
    `-ImplicitCastExpr 0x561483e12810 <col:1> 'void (*)()' <FunctionToPointerDecay>
      `-DeclRefExpr 0x561483e127c0 <col:1> 'void ()' lvalue Function 0x561483e12678 'bar' 'void ()'
```

`foo` keeps its declaration and its place in the lookup table but the body is never turned into an AST.

Example: Function Templates

```
template <int N> void used()   {}
template <int N> void unused() {}

used<1>(); // unused<N> is never instantiated
```

```
|-FunctionTemplateDecl 0x555abd2a01f8 <line:3:1, col:33> col:23 used external-linkage
| |-NonTypeTemplateParmDecl 0x555abd2a0088 <col:11, col:15> col:15 'int' depth 0 index 0 N
| |-FunctionDecl 0x555abd2a0150 <col:18, col:33> col:23 used 'void ()'
| | `-CompoundStmt 0x555abd2a0860 <col:32, col:33>
| `-FunctionDecl 0x555abd2a0698 <col:18, col:33> col:23 used used 'void ()' implicit_instantiation instantiated_from 0x555abd2a0150 external-linkage
|   |-TemplateArgument integral '1'
|   `-CompoundStmt 0x555abd2a0860 <col:32, col:33>
|-FunctionTemplateDecl 0x555abd2a0410 <line:4:1, col:30> col:23 unused external-linkage
| |-NonTypeTemplateParmDecl 0x555abd2a02d0 <col:11, col:15> col:15 'int' depth 0 index 0 N
| `-FunctionDecl 0x555abd2a0368 <col:18, col:30> col:23 unused 'void ()'
|   `-<<<NULL>>>
`-TopLevelStmtDecl 0x555abd2a0528 <line:6:1, col:9> col:1
  `-CallExpr 0x555abd2a0838 <col:1, col:9> 'void'
    `-ImplicitCastExpr 0x555abd2a0820 <col:1, col:7> 'void (*)()' <FunctionToPointerDecay>
      `-DeclRefExpr 0x555abd2a0790 <col:1, col:7> 'void ()' lvalue Function 0x555abd2a0698 'used' 'void ()' (FunctionTemplate 0x555abd2a01f8 'used')
```

The pattern is a deferred body like any other. `used<1>` gets one because the call named it; `unused<N>` is never asked for.

Example: Methods

```
struct A {
  void foo() {}
  template <int N> void bar() {}
};

A a;
a.foo(); // A::bar is never used
```

```
|-CXXRecordDecl 0x555822a5f050 <line:3:1, line:6:1> line:3:8 referenced struct A definition external-linkage
| |-CXXMethodDecl 0x555822a5f280 <line:4:3, col:15> col:8 used foo 'void ()' implicit-inline external-linkage
| | `-CompoundStmt 0x555822a810d0 <col:14, col:15>
| |-FunctionTemplateDecl 0x555822a5f480 <line:5:3, col:29> col:25 bar external-linkage
| | |-NonTypeTemplateParmDecl 0x555822a5f340 <col:13, col:17> col:17 'int' depth 0 index 0 N
| | `-CXXMethodDecl 0x555822a5f3d8 <col:20, col:29> col:25 bar 'void ()' implicit-inline
| |   `- <<<NULL>>>
|-VarDecl 0x555822a5f528 <line:8:1, col:3> col:3 used a 'A' callinit external-linkage
| `-CXXConstructExpr 0x555822a80f90 <col:3> 'A' 'void () noexcept'
`-TopLevelStmtDecl 0x555822a81000 <line:9:1, col:7> col:1
  `-CXXMemberCallExpr 0x555822a810a8 <col:1, col:7> 'void'
    `-MemberExpr 0x555822a81078 <col:1, col:3> '<bound member function type>' .foo 0x555822a5f280
      `-DeclRefExpr 0x555822a81058 <col:1> 'A' lvalue Var 0x555822a5f528 'a' 'A'
```

A method is a function, so the same machinery applies inside a class.

Example: Virtual Methods

```
struct A {
  virtual void foo() {}
  void bar() {}
};

A a; // constructing A needs the vtable, and so needs foo
```

```
|-CXXRecordDecl 0x56002f4fe050 <line:3:1, line:6:1> line:3:8 referenced struct A definition external-linkage
| |-CXXMethodDecl 0x56002f4fe280 <line:4:3, col:23> col:16 used foo 'void ()' virtual implicit-inline external-linkage
| | `-CompoundStmt 0x56002f51f7e0 <col:22, col:23>
| |-CXXMethodDecl 0x56002f4fe348 <line:5:3, col:12> col:8 bar 'void ()' implicit-inline external-linkage
| | `-<<<NULL>>>
`-VarDecl 0x56002f4fe9a0 <line:8:1, col:3> col:3 a 'A' callinit external-linkage
  `-CXXConstructExpr 0x56002f51f770 <col:3> 'A' 'void () noexcept'
```

The vtable needs the address of every virtual function, so `foo` is parsed as soon as `A` is used. `bar` still waits.

Classes: Using the Same Trick

- We use the same methodology as for functions to parse records on-demand.
- A record declaration is parsed eagerly, inserted into the lookup table and marked with `Sema::MarkAsLateParsedRecord`.
- The lexical token stream of the body, including the attributes and the base-clause, is stored. This makes the record an incomplete type.
- Whenever a complete type is required, it goes through `Sema::RequireCompleteType`, which is responsible for parsing the stored tokens into the AST.
- There are three other places where a complete type is required but which do not go through `Sema::RequireCompleteType`:
  - Out-of-line definition of methods.
  - Implicit or explicit instantiation.
  - Building implicit deduction guides.

Example: A Class Nobody Needed

```
struct Unused { int a; void f() { printf("Unused::f()\n"); } };

struct Used   { int b; void g() { printf("Used::g()\n");   } };

Used u; // only Used is required to be complete
```

```
├─CXXRecordDecl 0×55a71c02a620 <line:3:1, col:8> col:8 struct Unused external-linkage
├─CXXRecordDecl 0×55a71c02a6b0 <line:8:1, line:11:1> line:8:8 referenced struct Used definition external-linkage
│ ├─FieldDecl 0×55a71c02a910 <line:9:3, col:7> col:7 b 'int'
│ ├─CXXMethodDecl 0×55a71c02a9e0 <line:10:3, col:10> col:8 g 'void ()' implicit-inline external-linkage
│ │ `-<<<NULL>>>
`-VarDecl 0×55a71c02a798 <line:13:1, col:6> col:6 u 'Used' callinit external-linkage
  `-CXXConstructExpr 0×55a71c02b148 <col:6> 'Used' 'void () noexcept'
```

`Unused` never becomes a definition but `Used` does, and its member `g` is still a deferred body. So the two halves compose.

Example: Class Templates

```
template <typename T> struct Used   { T value; };
template <typename T> struct Unused { T value; };

Used<int> u; // Unused is never instantiated
```

```
|-ClassTemplateDecl 0x55c32a8907f0 <line:3:1, line:4:24> col:8 referenced Used external-linkage
| |-TemplateTypeParmDecl 0x55c32a890690 <line:3:11, col:20> col:20 referenced typename depth 0 index 0 T
| |-CXXRecordDecl 0x55c32a890760 <line:4:1, col:24> col:8 struct Used definition
| | `-FieldDecl 0x55c32a890e98 <col:15, col:17> col:17 value 'T'
| `-ClassTemplateSpecializationDecl 0x55c32a890af0 <line:3:1, line:4:24> col:8 referenced struct Used definition external-linkage instantiated_from 0x55c32a890760 implicit_instantiation
|   |-TemplateArgument type 'int'
|   | `-BuiltinType 0x55c32a83fb20 'int'
|   `-FieldDecl 0x55c32a891060 <col:15, col:17> col:17 value 'int'
|-ClassTemplateDecl 0x55c32a890980 <line:6:1, line:7:8> col:8 Unused external-linkage
| |-TemplateTypeParmDecl 0x55c32a890848 <line:6:11, col:20> col:20 typename depth 0 index 0 T
| `-CXXRecordDecl 0x55c32a8908f0 <line:7:1, col:8> col:8 struct Unused
`-VarDecl 0x55c32a890d28 <line:9:1, col:11> col:11 u 'Used<int>' callinit external-linkage
  `-CXXConstructExpr 0x55c32a8b4008 <col:11> 'Used<int>' 'void () noexcept'
```

`Used` is parsed because the instantiation needs the pattern. `Unused` stays a bare template declaration with no member-specification at all.

Example: Incomplete Is Often Enough

```
struct Pointee { int a; };
struct Sized   { int b; };

Pointee *p;              // a pointer does not need the definition
int n = sizeof(Sized);   // sizeof does
```

```
├─CXXRecordDecl 0×55f099091de0 <line:3:1, col:8> col:8 referenced struct Pointee external-linkage
├─CXXRecordDecl 0×55f099091e70 <line:4:1, col:25> col:8 referenced struct Sized definition external-linkage
│ `-FieldDecl 0×55f099092250 <col:18, col:22> col:22 b 'int'
├─VarDecl 0×55f099091fe0 <line:6:1, col:10> col:10 p 'Pointee *' external-linkage
`-VarDecl 0×55f099092060 <line:7:1, col:21> col:5 n 'int' cinit external-linkage
  `-ImplicitCastExpr 0×55f0990922d8 <col:9, col:21> 'int' <IntegralCast>
    `-UnaryExprOrTypeTraitExpr 0×55f0990922b8 <col:9, col:21> '__size_t':'unsigned long' sizeof 'Sized'
```

Everything that needs a complete type already went through one funnel, so declaring a pointer costs nothing and `sizeof` pays for exactly one body.

Putting the Parser Back Together

A deferred class body is parsed *in the middle of parsing an unrelated piece of code*. Therefore the parser needs to be restored to a state that is ready to parse the deferred record.

- **Lexical scopes and declaration contexts** are restored to the ones the record was written in, not the ones the parser happens to be in. Unqualified lookup walks this chain outwards.
- **Lookup tables.** A function's local declarations are reachable only through the scope chain, so a member of a local class cannot see them until they are put back.
- **Floating-point pragma state** as configured at the record's site, not as in effect where the type was first needed.
- …

Example: Scopes and Declaration Contexts

```
namespace N {
  int helper() { return 7; }
  struct S { int x = helper(); };   // deferred; `helper` is unqualified
}

namespace M {
  int k = sizeof(N::S);             // S's body is parsed here, inside M
}
```

`helper` is found by walking outwards from the scope the parser is in. At that moment we are parsing inside the scope `M`, where there is no `helper`. We need to save the current contexts and scopes and restore a state in which `S` can be parsed.

Example: The Lookup Table

```
void f() {
  typedef int MyInt;
  struct S {
    MyInt g() { MyInt v = 1; return v; }   // deferred body
  };
  S s;
  (void)s.g();                             // g is parsed here
}
```

A local class is parsed with its function, but its **methods are still deferred**. `MyInt` is a declaration of `f`, reachable only through the scope chain, so it has to be put back into the lookup table by hand before `g`'s body can see it, and taken out again afterwards.

Example: Floating-Point Pragmas

```
#pragma STDC FP_CONTRACT ON
struct Contracted {
  float f(float a, float b, float c) { return a * b + c; }
};
#pragma STDC FP_CONTRACT OFF

extern "C" float use(Contracted w) { return w.f(1, 2, 3); }
```

`Contracted` is first needed complete at `use`, where contraction is **off**. Without the pragma stack being restored to what was in effect where the member was *written*, `a * b + c` stops being an `llvm.fmuladd`, resulting in a silently different computation.

Semantic Problem: Name Lookup

A deferred body is parsed after everything below it in the file has been seen, so the lookup table has entries that the body must not be allowed to use.

```
static int foo(int) { return 1; }
static int bar() { return foo(0.0); }  // only foo(int) is visible here
static int foo(double) { return 2; }   // must not take part

__attribute__((used))
int main() { return printf("%d\n", bar()); } // forces bar to be parsed here
```

The cached tokens carry their original locations, so when constructing a named-value `LookupResult` we filter out the results that are written after the lookup point. Exempt from the filter:

- Lookups within members of a class.
- Lookups based on the template argument being substituted.
- The lambda call operator.

Exemption: Members of a Class

```
struct A {
  void foo() { printf("A::foo(%d)\n", n); }  // n is declared below
  void bar();
  int n = 10;
};

void A::bar() { printf("A::bar(%d)\n", n); }
```

The complete-class rule says a member body sees the whole class, however late in it a name appears. A lookup result that is a member of the same class, or of a base, is kept whatever the source order says.

`$INC` is `-Xclang -fincremental-extensions`, `$OD` is `-fparse-records-ondemand`.

```
$ clang++ $INC \
      lookup-class.cpp && ./a.out
A::foo(10)
A::bar(10)
```

```
$ clang++ $INC $OD \
      lookup-class.cpp && ./a.out
A::foo(10)
A::bar(10)
```

Exemption: The Lambda Call Operator

```
void callback(std::function<int()> f) {
  printf("callback::f() = %d\n", f());   // lookup of operator()
}

int main() { callback([]() { return 1; }); }   // operator() written here
```

The lambda is written *below* the body that calls through it, so the filter would hide the only `operator()` there is.

`$INC` is `-Xclang -fincremental-extensions`, `$OD` is `-fparse-records-ondemand`.

```
$ clang++ $INC \
      lookup-lambda.cpp && ./a.out
callback::f() = 1
```

```
$ clang++ $INC $OD \
      lookup-lambda.cpp && ./a.out
callback::f() = 1
```

Exemption: Template Argument Substitution

```
template <typename T>
typename T::result call_bar(T t) {              // lookup of T::result
  typename T::result res = t.bar();
  return res;
}

struct K { using result = int; result bar() { return 0; } };  // declared here
call_bar(K{});
```

A dependent name is looked up where the argument is *substituted*, not where the template was written, so the filter is switched off while a substitution is in progress.

`$INC` is `-Xclang -fincremental-extensions`, `$OD` is `-fparse-records-ondemand`.

```
$ clang++ $INC \
      lookup-template.cpp && ./a.out
t.bar() = (0)
```

```
$ clang++ $INC $OD \
      lookup-template.cpp && ./a.out
t.bar() = (0)
```

Benchmark 1: The Whole STL, Unused

```
#include <bits/stdc++.h>   // into clang-repl; nothing is done with it
```

Mean of 10 runs. Upstream is a separate build of the `release/23.x` branch (the base branch of this work), configured as a release build with assertions.

on-demand	net wall	net RSS
upstream	0.627 s	75.4 MB
on-demand off	0.628 s	74.9 MB
on-demand on	**0.347 s**	**66.1 MB**

−45 % time, −12 % memory against upstream.

net = measured minus the binary's start-up floor.

Benchmark 2: CppJIT Start-Up

```
import cppjit   # creates the clang-repl interpreter and preloads its headers
```

The upstream row is a second cppjit built against upstream LLVM from the same cppjit and CppInterOp sources.

on-demand	net wall	net RSS
upstream	1.260 s	166.5 MB
on-demand off	1.270 s	166.8 MB
on-demand on	**0.651 s**	**158.1 MB**

−48 % time, −5 % memory against upstream

Benchmark 3: When Everything Is Used

A generated header of ~20,000 lines with 400 units, each a free function, a function template, a class with methods and a class template, and a driver that uses **every one** of them.

on-demand	net wall	net RSS
upstream	0.174 s	33.0 MB
on-demand off	0.175 s	33.3 MB
on-demand on	0.204 s	44.5 MB
	+17 % time	**+35 % memory**

The memory overhead is in caching the tokens; the time overhead is restoring the parser state, including re-entering the scopes once per late parse.

Benchmark 4: CppJIT's Test Suite

674 tests, run serially. Mean of 3.

on-demand	wall	peak RSS
upstream `release/23.x`	41.82 s	761.1 MB
branch, flags off	41.68 s	761.1 MB
on	**38.49 s**	**747.4 MB**

−8 % time, −2 % memory against upstream.

A comparison only means anything if every side did the same work, so the harness runs the suite all three ways, and refuses to measure unless the reports agree — *674 tests, 623 passed, 43 skipped, 8 xfailed, 0 failed, identical on all three.*

Limitations: Semantics

The lookup filter and its exemptions are thinly tested. The source-order rule and every exemption carved out of it are only exercised by the regression tests. A construct that reaches a lookup from a late-parsed body along a path none of that covers is exactly where the next bug is.

An unused body is never diagnosed.

```
struct Undefined;              // declared, never defined
void use() { Undefined u; }    // nobody calls use(), so nobody ever complains
```

Consistent with the deferred model, but a deviation from the standard.

Limitations: Cost

Restoring the parser state is linear, and paid per deferred entity.

```
void Sema::LocalDeclsReentryRAII::reenter(Scope *S, DeclContext *DC) {
  for (Decl *D : DC→decls()) {      // every declaration of the parent scope ..
    S→AddDecl(D);
    SemaRef.IdResolver.AddDecl(D); // ... in, and out again on exit
  }
}
```

Ordinary parsing builds a lexical scope once and keeps it for every function in it. A late parse inserts the parent's declarations into the lookup table on entry and withdraws them on exit, separately for every deferred body.

The cached tokens are held for the whole session. A body that has been parsed still owns its token buffer, so the memory a deferral costs is never given back. Releasing it was tried and did not move the peak, so benchmark 3's memory regression is not an easy fix. It would bound a long-running session, which is the reason to come back to it.

AI Usage

- LLMs were heavily used to understand and navigate the code base.
- Almost all of the tests and the benchmarks were written using them.
- LLMs helped in debugging: I would give one a reproducer and it would point me to the buggy code path. They were also used to make sure that the cppjit test suite would pass, by creating standalone C++ reproducers of complex interop-based tests.
- For deferred parsing of functions, I wrote most of the feature by hand. Later in the development process it became the LLMs writing the code and me reviewing it.
- These slides were prepared with the help of LLMs.

Future Work

- **Cover what is still eager:** A class whose base-clause the scan cannot resolve, local classes, lambdas, partial & explicit specializations.
- **A cheaper parser state (re-)store mechanism:** A persistent cache, so that entering and leaving a lexical scope is not $O(\text{declarations in the parent})$.
- **Multi-Threaded Parsing:** This work sets up one of the things necessary in clang to potentially parse source files in parallel. Currently, the clang parser is not thread-safe, and making it thread-safe would be a multi-year project.
- **Beyond incremental mode:** Lazy parsing makes less sense in non-incremental (AOT) compilation, as the generated object files would need all the symbols defined in the source. With multi-threaded parsing, we have an application for the AOT use case.
- **Upstreaming.** Currently all the work done lives in my fork of LLVM here. Upstreaming this work would be a months-long endeavour, as it changes how source is parsed at a fundamental level.

Thank You

Any questions or comments?

Backup Slides

Still Eager: A Base-Clause the Scan Cannot Resolve

The body has to be cached *whole*, base-clause included, so the parser must find the opening { **before** it has parsed the base-clause. Brackets are tracked exactly; angles only heuristically, because < is also less-than.

```
template <bool> struct Base {};
constexpr int N = 1, M = 2;

struct Deferred : Base<(N < M)> { int x; };  // parenthesised: deferred
struct Eager    : Base<N < M>   { int x; };  // bare '<': scan gives up
```

In the second one the inner < follows a name, so the scan counts it as another angle, never comes back to depth zero, and runs into the ; that closes the declaration. The definition is then parsed the ordinary way.

A wrong guess costs performance, never correctness.

Still Eager: Member, Local and Lambda

A member or a local class is parsed with the context it was written in, so only **namespace-scope** classes are deferred.

```
struct Outer {
  struct Inner { int x; };   // parsed with Outer, not deferred again
  int y;
};
int n = sizeof(Outer);       // Outer's body is parsed here
```

```
void fn() {
  struct Local { int x; };   // parsed with fn's body
  Local l;
}
```

A lambda is excluded outright — the closure type has no name to look up and no second chance to be asked for:

```
auto g = [](int x) { return x + 1; };  // the closure type is never deferred
```

Deferring the *enclosing* entity still pays: nothing inside `Outer` or `fn` is parsed until `Outer` or `fn` is.

Still Eager: Template Specializations

A partial or explicit specialization is matched by its arguments, not found by name, so there is no single point where it can be asked for:

```
template <typename T> struct S       { int a; };  // primary: deferred
template <typename T> struct S<T *>  { int b; };  // partial:  eager
template <>           struct S<int>  { int c; };  // explicit: eager
```