

libclang: Thinking Beyond the Compiler

Clang is not just a great compiler...

Clang is not just a great compiler...

- Clang is also a library for processing source code
 - Translates text into **Abstract Syntax Trees**
 - Resolves identifiers and symbols
 - Expands macros
 - Makes implicit information explicit

Clang is not just a great compiler...

- Clang is also a library for processing source code
 - Translates text into **Abstract Syntax Trees**
 - Resolves identifiers and symbols
 - Expands macros
 - Makes implicit information explicit
- Clang obsessively tracks source-level location information

Introducing libclang

Introducing libclang

- Parsing

Introducing libclang

- Parsing
- Indexing and cross-referencing

Introducing libclang

- Parsing
- Indexing and cross-referencing
- Syntax highlighting

Introducing libclang

- Parsing
- Indexing and cross-referencing
- Syntax highlighting
- Code completion

Parsing Source Code

Parsing Source Code

```
int main(int argc, char *argv[]) {
```

```
    return 0;
```

```
}
```

Parsing Source Code

```
int main(int argc, char *argv[]) {
    CXIndex Index = clang_createIndex(0, 0);
```

```
clang_disposeIndex(Index);  
return 0;  
}
```

Parsing Source Code

```
int main(int argc, char *argv[]) {  
    CXIndex Index = clang_createIndex(0, 0);  
    CXTranslationUnit TU = clang_parseTranslationUnit(Index, 0,  
                                                       argv, argc, 0, 0, CXTranslationUnit_None);  
  
    clang_disposeTranslationUnit(TU);  
    clang_disposeIndex(Index);  
    return 0;  
}
```

Parsing Source Code

```
int main(int argc, char *argv[]) {  
    CXIndex Index = clang_createIndex(0, 0);  
    CXTranslationUnit TU = clang_parseTranslationUnit(Index, 0,  
                                                       argv, argc, 0, 0, CXTranslationUnit_None);  
  
    clang_disposeTranslationUnit(TU);  
    clang_disposeIndex(Index);  
    return 0;  
}
```

Parsing Source Code

```
int main(int argc, char *argv[]) {
    CXIndex Index = clang_createIndex(0, 0);
    CXTranslationUnit TU = clang_parseTranslationUnit(Index, 0,
                                                    argv, argc, 0, 0, CXTranslationUnit_None);
    for (unsigned I = 0, N = clang_getNumDiagnostics(TU); I != N; ++I) {
        CXDiagnostic Diag = clang_getDiagnostic(TU, I);
        CXString String = clang_formatDiagnostic(Diag,
                                                clang_defaultDiagnosticDisplayOptions());
        fprintf(stderr, "%s\n", clang_getCString(String));
        clang_disposeString(String);
    }
    clang_disposeTranslationUnit(TU);
    clang_disposeIndex(Index);
    return 0;
}
```

Parsing Source Code—Results

- Given list.c:

```
struct List { ... };  
int sum(union List *L) { /* ... */ }
```

- Run our syntax-checker:

```
$ syntax-check -I../.. list.c
```

```
list.c:2:9: error: use of 'List' with tag type that  
does not match previous declaration  
list.c:1:8: note: previous use is here
```

Parsing Source Code—Results

- Given list.c:

```
struct List { ... };  
int sum(union List *L) { /* ... */ }
```

- Run our syntax-checker:

```
list.c:2:9: error: use of 'List' with tag type that does not match  
          previous declaration  
int sum(union List *Node) {  
        ^~~~~~  
        struct  
list.c:1:8: note: previous use is here  
struct List {  
      ^
```

Dissecting Diagnostics

Dissecting Diagnostics

- Core diagnostic information:
 - `enum CXDiagnosticSeverity`
`clang_getDiagnosticSeverity(CXDiagnostic Diag);`
 - `CXSourceLocation`
`clang_getDiagnosticLocation(CXDiagnostic Diag);`
 - `CXString clang_getDiagnosticSpelling(CXDiagnostic Diag);`

Dissecting Diagnostics

- Core diagnostic information:
 - `enum CXDiagnosticSeverity`
`clang_getDiagnosticSeverity(CXDiagnostic Diag);`
 - `CXSourceLocation`
`clang_getDiagnosticLocation(CXDiagnostic Diag);`
 - `CXString clang_getDiagnosticSpelling(CXDiagnostic Diag);`
- Highlighted source ranges

Dissecting Diagnostics

- Core diagnostic information:
 - `enum CXDiagnosticSeverity`
`clang_getDiagnosticSeverity(CXDiagnostic Diag);`
 - `CXSourceLocation`
`clang_getDiagnosticLocation(CXDiagnostic Diag);`
 - `CXString clang_getDiagnosticSpelling(CXDiagnostic Diag);`
- Highlighted source ranges
- Fix-its:
 - `unsigned clang_getDiagnosticNumFixIts(CXDiagnostic Diag);`
 - `CXString clang_getDiagnosticFixIt(CXDiagnostic Diag,
 unsigned FixIt,
 CXSourceRange *ReplacementRange);`

Indexing & Cross-Referencing

Indexing and Cross-Referencing

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

Indexing and Cross-Referencing

- Walk the Abstract Syntax Tree

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

Indexing and Cross-Referencing

- Walk the Abstract Syntax Tree
 - Declarations

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

Indexing and Cross-Referencing

- Walk the Abstract Syntax Tree

- Declarations
- References

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

Indexing and Cross-Referencing

- Walk the Abstract Syntax Tree

- Declarations
- References
- Statements & expressions

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

Indexing and Cross-Referencing

- Walk the Abstract Syntax Tree

- Declarations
- References
- Statements & expressions
- Macro definitions & instantiations

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

CXCursor: AST Node, Simplified

- `typedef struct { ... } CXCursor;`
- Unifies AST nodes (declaration, reference, expression, statement, etc.)
 - Source location and extent
 - Name and symbol resolution
 - Type
 - Child nodes

Example CXCursor: struct List

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Example CXCursor: struct List

- Top-level cursor C for List:
 - clang_getCursorKind(C)
== CXCursor_StructDecl
 - clang_getCursorSpelling(C)
== "List"

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Example CXCursor: struct List

- Top-level cursor C for List:
 - clang_getCursorKind(C)
== CXCursor_StructDecl
 - clang_getCursorSpelling(C)
== "List"
 - clang_getCursorLocation(C)



```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Example CXCursor: struct List

- Top-level cursor C for List:
 - clang_getCursorKind(C)
== CXCursor_StructDecl
 - clang_getCursorSpelling(C)
== "List"
 - clang_getCursorLocation(C)
 - clang_getCursorExtent(C)

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Example CXCursor: struct List

- Top-level cursor C for List:

- clang_getCursorKind(C)
== CXCursor_StructDecl

- clang_getCursorSpelling(C)
== "List"

- clang_getCursorLocation(C)

- clang_getCursorExtent(C)

- clang_visitChildren(C, ...);

```
struct List {  
    → int Data;  
    → struct List *Next;  
};
```

Reference Cursors

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Reference Cursors

- Reference cursor R for List:
 - `clang_getCursorKind(R)`
 `== CXCursor_TypeRef`
 - `clang_getCursorSpelling(R)`
 `== "List"`
 - `clang_getCursorLocation(R)`

```
struct List {  
    int Data;  
    struct List *Next;  
};
```



Reference Cursors

- Reference cursor R for List:
 - `clang_getCursorKind(R)`
 `== CXCursor_TypeRef`
 - `clang_getCursorSpelling(R)`
 `== "List"`
 - `clang_getCursorLocation(R)`
 - `clang_getCursorExtent(R)`

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Reference Cursors

- Reference cursor R for List:

- `clang_getCursorKind(R)`
 `== CXCursor_TypeRef`
- `clang_getCursorSpelling(R)`
 `== "List"`
- `clang_getCursorLocation(R)`
- `clang_getCursorExtent(R)`
- `clang_getCursorReferenced(R) == C`

```
struct List {  
    int Data;  
    struct List *Next;  
};
```



Local Renaming of Declarations

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Local Renaming of Declarations

- Walk all cursors in the AST, recursively:

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Local Renaming of Declarations

- Walk all cursors in the AST, recursively:
 - Identify the cursor C corresponding to the declaration we want to rename

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Local Renaming of Declarations

- Walk all cursors in the AST, recursively:
 - Identify the cursor C corresponding to the declaration we want to rename
 - Find each cursor R: `clang_getCursorReferenced(R) = C`

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

Local Renaming of Declarations

- Walk all cursors in the AST, recursively:
 - Identify the cursor C corresponding to the declaration we want to rename
 - Find each cursor R: `clang_getCursorReferenced(R) = C`
- Perform textual replacement of the name at the locations of C and each R

```
struct MyList {  
    int Data;  
    struct MyList *Next;  
};
```

Spanning Translation Units

- **Unified Symbol Resolutions** provide a stable name for declarations
 - Every entity with external linkage has a USR
 - USRs are stable across translation units, time

```
CXString clang_getCursorUSR(CXCursor C);
```

Syntax Coloring

What's Under My Cursor?

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```

What's Under My Cursor?

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```



What's Under My Cursor?

- Kind: CXCursor_DeclRefExpr
- Type: int
- References: variable declaration "result"

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```



What's Under My Cursor?

```
CXSourceLocation
```

```
clang_getLocation(CXTranslationUnit TU,  
                  CXFile File,  
                  unsigned Line,  
                  unsigned Column);
```

```
CXCursor clang_getCursor(CXTranslationUnit TU,  
                          CXSourceLocation Where);
```

```
int result = 0;  
for (; Node; Node = Node->Next)  
    result = result + Node->Data;  
return result;  
}
```



What's Under My Cursor?

- Kind:
CXCursor_UnexposedExpr
- Type: int

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```



What's Under My Cursor?

- Kind:
CXCursor_MemberRefExpr
- Type: struct List *
- References: field
declaration "Next"

```
struct List {  
    int Data;  
    struct List *Next;  
};
```

```
int sum(struct List *Node) {  
    int result = 0;  
    for (; Node; Node = Node->Next)  
        result = result + Node->Data;  
    return result;  
}
```



Syntax Coloring

```
struct MyList {  
    int Data;  
    struct MyList *Next;  
};
```

Syntax Coloring

```
struct MyList {  
    int Data;  
    struct MyList *Next;  
};
```

Syntax Coloring

- `void clang_tokenize(CXTranslationUnit, CXSourceRange, CXToken **Tokens, unsigned *NumTokens);`

```
struct MyList {  
    int Data;  
    struct MyList *Next;  
};
```

Syntax Coloring

- `void clang_tokenize(CXTranslationUnit, CXSourceRange, CXToken **Tokens, unsigned *NumTokens);`
 - Keyword [1:1-1:7]: "struct"
 - Identifier [1:8-1:14]: "MyList"
 - Punctuation [1:15-1:16]: "{"
 - Keyword [2:3-2:6]: "int"
 - Identifier [2:7-2:11]: "Data"
- ```
struct MyList {
 int Data;
 struct MyList *Next;
};
```

  - Punctuation [2:11-2:12]: ";"
  - Keyword [3:3-3:9]: "struct"
  - Identifier [3:10-3:16]: "MyList"

# Syntax Coloring

- `void clang_annotateTokens(CXTranslationUnit, CXToken *Tokens, unsigned NumTokens, CXCursor *Cursors);`
  - Keyword [1:1-1:7]: "struct"
  - Identifier [1:8-1:14]: "MyList"
  - Punctuation [1:15-1:16]: "{"
  - Keyword [2:3-2:6]: "int"
  - Identifier [2:7-2:11]: "Data"
- `struct MyList {  
 int Data;  
 struct MyList *Next;  
};`
  - Punctuation [2:11-2:12]: ";"
  - Keyword [3:3-3:9]: "struct"
  - Identifier [3:10-3:16]: "MyList"

# Syntax Coloring

- `void clang_annotateTokens(CXTranslationUnit, CXToken *Tokens, unsigned NumTokens, CXCursor *Cursors);`
  - Keyword [1:1-1:7]: "struct"
  - Identifier [1:8-1:14]: "MyList" (MyList struct declaration)
  - Punctuation [1:15-1:16]: "{" `struct MyList {`
  - Keyword [2:3-2:6]: "int" `int Data;`
  - Identifier [2:7-2:11]: "Data" (Data member declaration) `struct MyList`
  - Punctuation [2:11-2:12]: ";" `*Next;`
  - Keyword [3:3-3:9]: "struct" `};`
  - Identifier [3:10-3:16]: "MyList" (MyList type reference)

# Code Completion

# What Can I Do <here>?

# What Can I Do <here>?

- Code completion suggests what we can type, e.g.
  - “Data”
  - “Next”

```
int sum(struct List *Node) {
 int result = 0;
 for (; Node; Node = Node->Next)
 result = result + Node->
```

# What Can I Do <here>?

- Code completion suggests what we can type, e.g.
  - “Data”
  - “Next”
- Implemented in the parser:
  - Insert  token in the lexer
  - Parse  token and call context-sensitive callback function
  - Form completion results

```
int sum(struct List *Node) {
 int result = 0;
 for (; Node; Node = Node->Next)
 result = result + Node->
```

# What Can I Do <here>? (Cont.)

# What Can I Do <here>? (Cont.)

```
int sum(struct List *Node) {
 int result = 0;
 for (; Node; Node = Node->Next)
 result = result + 
```

# What Can I Do <here>? (Cont.)

- Reference a local variable (result, Node)
- Call a function (e.g., sum)
- Use any C expression (e.g., sizeof)
- Use any defined macro (e.g., NULL)
- In C++, name any type (e.g., List)
- In C++, refer into a namespace (e.g., std::)

```
int sum(struct List *Node) {
 int result = 0;
 for (; Node; Node = Node->Next)
 result = result + 
```

# Semantic Strings

# Semantic Strings

- Code completion results are composed of “chunks”:
  - **Typed Text**, which the user types and is inserted into the file
  - **Text**, which is inserted into the buffer
  - **Informative Text**, which is shown but not inserted
  - **Placeholders**, which indicate code the user should modify
  - **Result Type**, which indicates the type of the result
  - **Cursor Kind**, which is the kind of entity in the completion

# Semantic Strings

- Code completion results are composed of “chunks”:
  - **Typed Text**, which the user types and is inserted into the file
  - **Text**, which is inserted into the buffer
  - **Informative Text**, which is shown but not inserted
  - **Placeholders**, which indicate code the user should modify
  - **Result Type**, which indicates the type of the result
  - **Cursor Kind**, which is the kind of entity in the completion
- Example completion: `int sum(struct List *Node)`
  - Cursor kind: `CXCursor_FunctionDecl`

Result Type  
`int`

Typed Text  
`sum`

Text  
`(`

Placeholder  
`struct List *Node`

Text  
`)`

```

34
35 if (CXXConstructorDecl *Constructor = dynamic_cast<type>(expression))
36 {
37 typename cast_rety<X, Y>::ret_type llvm::dyn_cast<class X>(Y const &Val)
38 {
39 typename cast_rety<X, Y *>::ret_type llvm::dyn_cast_or_null<class X>(Y *Val)
40 dynamic_cast<type>(expression)
41 }
42 // generate a forwarding call.
43 for (CXXMethodDecl::param_iterator P = Method->param_begin(),
44 PEnd = Method->param_end(); P != PEnd; ++P)

```

# Wrap-Up

# Getting Started with libclang

- User-level tools
  - Xcode 4 Developer Preview
  - Emacs code completion “demo” (see `clang/utis/clang-completion-mode.el`)
  - Vim code completion (see `llvm/utis/vim/`)
- Developer-centric resources
  - `c-index-test` provides a command-line interface to the API
  - Doxygen documentation at <http://clang.llvm.org>

# libclang Supports Development Tools

- Simple C API covers a variety of features
  - Parsing
  - Indexing
  - Cross-referencing
  - Mapping between source code and ASTs
  - Syntax coloring
  - Code completion
- Great basis for development tools

