Extending Clang

Doug Gregor
A Platform for Tools

- Library-based architecture
- Compatibility with various language standards
- Accurate representation of source code
Extension Points

Lex ➔ Parse ➔ Sema ➔ CodeGen
Extension Points

- libclang
- Preprocessor callbacks, AST consumers
- Semantic analysis, static analyzer
- LLVM IR transformation and optimization
- Source-to-source translation
Source ⇔ AST Mapping
libclang: Clang C API

```c
struct List {
    int Data;
    struct List *Next;
};
```
libclang: Clang C API

• Where are all the declarations?

```c
struct List {
    int Data;
    struct List *Next;
};
```
libclang: Clang C API

- Where are all the declarations?
- Where are uses of List?
libclang: Clang C API

- Where are all the declarations?
- Where are uses of List?
- What is under my cursor?
libclang: Clang C API

- Where are all the declarations?
- Where are uses of List?
- What is under my cursor?
- What code completions work here?
libclang: Clang C API

- Where are all the declarations?
- Where are uses of List?
- What is under my cursor?
- What code completions work here?

See 2010 talk “libclang: Thinking Beyond the Compiler”
Preprocessor Callbacks

Lex → Parse → Sema → CodeGen
Preprocessor Callbacks

Lex $\rightarrow$ Parse $\rightarrow$ Sema $\rightarrow$ CodeGen
Preprocessor Callbacks

- Invoked for various preprocessor actions
- Macro definition/expansion
- Entering/leaving a file
- Pragmas, ifdefs
- Customize by overriding callbacks
class FindDependencies : public PPCallbacks {
public:
    void FileChanged(SourceLocation Loc, 
                      FileChangeReason Reason, 
                      SrcMgr::CharacteristicKind, 
                      FileID PrevFID) {
        if (Reason != EnterFile) return;
        if (const FileEntry *FE 
            = SM.getFileEntryForID(
                SM.getFileID(Loc)))
            std::cout << "Depends on " 
            << FE->getName() << "
";
    }
};
AST Consumers

Lex ➔ Parse ➔ Sema ➔ CodeGen
AST Consumers

Lex → Parse → Sema → CodeGen
class FindUnions : public ASTConsumer {
public:
    void HandleTagDeclDefinition(TagDecl *D) {
        if (D->isUnion()) {
            std::cout << "Union: "
                << D->getNameAsString()
                << "\n";
        }
    }
};
RecursiveASTVisitor

- Recursively walk any part of the AST
- `Call Visitor.Traverse<NodeType>(Node)`
- Customize by overriding visitation methods
- Used heavily within Clang itself
Finding Calls
Finding Calls

class FindCalls
  : public RecursiveASTVisitor<FindCalls> {

};
class FindCalls
    : public RecursiveASTVisitor<FindCalls> {

public:
    bool VisitCallExpr(CallExpr *Call) {
        if (FunctionDecl *Callee
            = Call->getDirectCallee())
            std::cout << "Call to "
                << Callee->getNameAsString()
                << "\n";
            return true;
    }
};
Warnings & Errors

Lex ➔ Parse ➔ Sema ➔ CodeGen
Warnings & Errors

Lex → Parse → Sema → CodeGen
A Terrible Diagnostic
A Terrible Diagnostic

typedef int N;
N::string str;
A Terrible Diagnostic

typedef int N;
N::string str;

t.cpp:2:1: error: expected a class or namespace
N::string str;
^
A Terrible Diagnostic
A Terrible Diagnostic

```plaintext
// DiagnosticSemaKinds.td
def err_expected_class_or_namespace
  : Error<"expected a class or namespace">;
```
A Terrible Diagnostic

// DiagnosticSemaKinds.td
def err_expected_class_or_namespace:
  Error<"expected a class or namespace">;

// SemaCXXScopeSpec.cpp
Diag(IdentifierLoc,
    diag::err_expected_class_or_namespace);
Improving Diagnostics
Improving Diagnostics

// DiagnosticSemaKinds.td
def err_not_class_or_namespace
    : Error"%0 is not a class or namespace";
Improving Diagnostics

```cpp
// DiagnosticSemaKinds.td
def err_not_class_or_namespace:
    Error<"%0 is not a class or namespace">;
```

```cpp
// SemaCXXScopeSpec.cpp
if (TypeDecl *TD = Found.getAsSingle<TypeDecl>())
    Diag(IdentifierLoc,
         diag::err_not_class_or_namespace)
    << Context.getTypeDeclType(TD);
```
Improving Diagnostics

typedef int N;
N::string str;

 t.cpp:2:1: error: ‘N’ (aka ‘int’) is not a class or namespace
N::string str;
Improving Diagnostics
Improving Diagnostics

// SemaCXXScopeSpec.cpp
if (TypeDecl *TD = Found.getAsSingle<TypeDecl>()) {
    Diag(IdentifierLoc,
         diag::err_not_class_or_namespace)
        << Context.getTypeDeclType(TD);
    Diag(TD->getLocation(),
         diag::note_declared_at);
}

Wednesday, December 7, 11
Improving Diagnostics

typedef int N;
N::string str;

t.cpp:2:1: error: ‘N’ (aka ‘int’) is not a class or namespace
N::string str;
^
t.cpp:1:13: note: declared here
typedef int N;
^
Attributes & LLVM IR

Lex → Parse → Sema → CodeGen
Attributes & LLVM IR

Lex → Parse → Sema → CodeGen
Feeding Information to IR

“If I could just tell the compiler that some declarations are <adjective>, my new optimization pass would be awesome!”
Feeding Information to IR

“If I could just tell the compiler that some declarations are <adjective>, my new optimization pass would be awesome!”

• Attributes make such experiments easy
• Trivial to parse with few ambiguities
• Easy to introduce into the AST
The annotate Attribute

- Clang supports the annotate attribute with arbitrary strings:
  ```c
  __attribute__((annotate("singleton")))
  Class *object;
  ```
- Annotations are mapped down to LLVM IR annotations
Adding Real Attributes
Adding Real Attributes

```python
// include/clang/Basic/Attr.td
def ReturnsTwice : InheritableAttr {
    let Spellings = ["returns_twice"];
}
```
Adding Real Attributes

```cpp
// include/clang/Basic/Attr.td
def ReturnsTwice : InheritableAttr {
    let Spellings = ["returns_twice"];
}
```

```cpp
// lib/Sema/SemaDeclAttr.cpp
static void handleReturnsTwiceAttr(Sema &S, Decl *D,
    const AttributeList &Attr) {
    if (!isa<FunctionDecl>(D)) {
        // diagnose error
        return;
    }
    D->addAttr(::new (S.Context) ReturnsTwiceAttr(...));
}
```
Adding Real Attributes

```cpp
// include/clang/Basic/Attr.td
def ReturnsTwice : InheritableAttr {
    let Spellings = ["returns_twice"];
}
```

```cpp
// lib/Sema/SemaDeclAttr.cpp
static void handleReturnsTwiceAttr(Sema &S, Decl *D,
const AttributeList &Attr) {
    if (!isa<FunctionDecl>(D)) {
        // diagnose error
        return;
    }
    D->addAttr(::new (S.Context) ReturnsTwiceAttr(...));
}
```

http://clang.llvm.org/docs/InternalsManual.html#AddingAttributes
Source-to-Source Translation
The Rewriter Class

- Rewriter class provides textual rewriting
The Rewriter Class

- Rewriter class provides textual rewriting

```cpp
class Rewriter {
public:
    bool InsertText(SourceLocation Loc, StringRef Text);
    bool RemoveText(SourceRange Range);
    bool ReplaceText(SourceRange Range, StringRef Text);
};
```
Help Wanted
Plugins
Plugins

- Clang allows plugins during normal compilation:

  clang -Xclang -load foo.so -Xclang -plugin foo-plugin <command line arguments>
Plugins

- Clang allows plugins during normal compilation:

  clang -Xclang -load foo.so -Xclang -plugin foo-plugin <command line arguments>

- Numerous problems with plug-in support:
  - ASTConsumers aren’t chained in a natural way
  - Command-line option parsing is too hard
  - Building plugins is too hard
  - Documentation is absent
One-Off Tools
One-Off Tools

- Building one-off tools is possible (but hard):
  - CompilerInstance/CompilerInvocation/Action not simple enough
  - Missing a “quickstart” tutorial
One-Off Tools

- Building one-off tools is possible (but hard):
  - CompilerInstance/CompilerInvocation/Action not simple enough
  - Missing a “quickstart” tutorial
  - Simple source-to-source translation needed
  - Tie together traversal, rewriter, verification
Summary
Summary

• Numerous extension points to Clang
• Picking the best one is important
Summary

- Numerous extension points to Clang
  - Picking the best one is important
- We need to make extension easier
  - Plugins need to be super-easy to write
  - “Your first extension” tutorials
- Make source-to-source translation easy