"Clang" CFE Internals Manual

Introduction

This document describes some of the more important APIs and internal design decisions made in the Clang C front-end. The purpose of this document is to both capture some of this high level information and also describe some of the design decisions behind it. This is meant for people interested in hacking on Clang, not for end-users. The description below is categorized by libraries, and does not describe any of the clients of the libraries.

LLVM System and Support Libraries

The LLVM libsystem library provides the basic Clang system abstraction layer, which is used for file system access. The LLVM libsupport library provides many underlying libraries and data-structures, including command line option processing and various containers.

The Clang 'Basic' Library

This library certainly needs a better name. The 'basic' library contains a number of low-level utilities for tracking and manipulating source buffers, locations within the source buffers, diagnostics, tokens, target abstraction, and information about the subset of the language being compiled for.

Part of this infrastructure is specific to C (such as the TargetInfo class), other parts could be reused for other non-C-based languages (SourceLocation, SourceManager, Diagnostics, FileManager). When and if there is future demand we can figure out if it makes sense to introduce a new library, move the general classes somewhere else, or introduce some other solution.

We describe the roles of these classes in order of their dependencies.

The Diagnostics Subsystem

The Clang Diagnostics subsystem is an important part of how the compiler communicates with the human. Diagnostics are the warnings and errors produced when the code is incorrect or dubious. In Clang, each diagnostic produced has (at the minimum) a unique ID, a SourceLocation to "put the caret", an English translation associated with it, and a severity (e.g. `WARNING` or `ERROR`). They can also optionally include a number of arguments to the dianostic (which fill in "%0"'s in the string) as well as a number of source ranges that related to the diagnostic.

In this section, we'll be giving examples produced by the Clang command line driver, but diagnostics can be rendered in many different ways depending on how the DiagnosticClient interface is implemented. A representative example of a diagonstic is:

```
t.c:38:15: error: invalid operands to binary expression ('int *' and '_Complex float')
   P = (P-42) + Gamma*4;
       ~~~~~~ ^ ~~~~~~~
```

In this example, you can see the English translation, the severity (error), you can see the source location (the caret ("^") and file/line/column info), the source ranges "~~~~", arguments to the diagnostic ("int*" and "_Complex float"). You'll have to believe me that there is a unique ID backing the diagnostic :).

Getting all of this to happen has several steps and involves many moving pieces, this section describes them and talks about best practices when adding a new diagnostic.

The DiagnosticKinds.def file

Diagnostics are created by adding an entry to the `DiagnosticKinds.def` file. This file encodes the unique ID of the diagnostic (as an enum, the first argument), the severity of the diagnostic (second argument) and the English translation + format string.

There is little sanity with the naming of the unique ID's right now. Some start with err_, warn_, ext_ to encode the severity into the name. Since the enum is referenced in the C++ code that produces the diagnostic, it is somewhat useful for it to be reasonably short.

The severity of the diagnostic comes from the set {`NOTE`, `WARNING`, `EXTENSION`, `EXTWARN`, `ERROR`}. The `ERROR` severity is used for diagnostics indicating the program is never acceptable under any circumstances. When an error is emitted, the AST for the input code may not be fully built. The `EXTENSION` and `EXTWARN` severities are used for extensions to the language that Clang accepts. This means that Clang fully understands and can represent them in the AST, but we produce diagnostics to tell the user their code is non-portable. The difference is that the former are ignored by default, and the later warn by default. The `WARNING` severity is used for constructs that are valid in the currently selected source language but that are dubious in some way. The `NOTE` level is used to staple more information onto previous diagnostics.

These *severities* are mapped into a smaller set (the Diagnostic::Level enum, {`Ignored`, `Note`, `Warning`, `Error`, `Fatal` }) of output *levels* by the diagnostics subsystem based on various configuration options. Clang internally supports a fully fine grained mapping mechanism that allows you to map almost any diagnostic to the output level that you want. The only diagnostics that cannot be mapped are `NOTE`s, which always follow the severity of the previously emitted diagnostic and `ERROR`s, which can only be mapped to `Fatal` (it is not possible to turn an error into a warning, for example).

Diagnostic mappings are used in many ways. For example, if the user specifies `-pedantic`, `EXTENSION` maps to `Warning`, if they specify `-pedantic-errors`, it turns into `Error`. This is used to implement options like `-Wunused_macros`, `-Wundef` etc.

Mapping to `Fatal` should only be used for diagnostics that are considered so severe that error recovery won't be able to recover sensibly from them (thus spewing a ton of bogus errors). One example of this class of error are failure to #include a file.

The Format String

The format string for the diagnostic is very simple, but it has some power. It takes the form of a string in English with markers that indicate where and how arguments to the diagnostic are inserted and formatted. For example, here are some simple format strings:

```
"binary integer literals are an extension"
"format string contains '\\0' within the string body"
"more '%%' conversions than data arguments"
"invalid operands to binary expression (%0 and %1)"
"overloaded '%0' must be a %select{unary|binary|unary or binary}2 operator"
     " (has %1 parameter%s1)"
```

These examples show some important points of format strings. You can use any plain ASCII character in the diagnostic string except "%" without a problem, but these are C strings, so you have to use and be aware of all the C escape sequences (as in the second example). If you want to produce a "%" in the output, use the "%%" escape sequence, like the third diagnostic. Finally, Clang uses the "%...[digit]" sequences to specify where and how arguments to the diagnostic are formatted.

Arguments to the diagnostic are numbered according to how they are specified by the C++ code that produces them, and are referenced by `%0` .. `%9`. If you have more than 10 arguments to your diagnostic, you are doing something wrong :). Unlike printf, there is no requirement that arguments to the diagnostic end up in the output in the same order as they are specified, you could have a format string with `"%1 %0"` that swaps them, for example. The text in between the percent and digit are formatting instructions. If there are no instructions, the argument is just turned into a string and substituted in.

Here are some "best practices" for writing the English format string:

- Keep the string short. It should ideally fit in the 80 column limit of the `DiagnosticKinds.def` file. This avoids the diagnostic wrapping when printed, and forces you to think about the important point you are conveying with the diagnostic.
- Take advantage of location information. The user will be able to see the line and location of the caret, so you don't need to tell them that the problem is with the 4th argument to the function: just point to it.
- Do not capitalize the diagnostic string, and do not end it with a period.
- If you need to quote something in the diagnostic string, use single quotes.

Diagnostics should never take random English strings as arguments: you shouldn't use `"you have a problem with %0"` and pass in things like `"your argument"` or `"your return value"` as arguments. Doing this prevents translating the Clang diagnostics to other languages (because they'll get random English words in their otherwise localized diagnostic). The exceptions to this are C/C++ language keywords (e.g. auto, const, mutable, etc) and C/C++ operators (`/=`). Note that things like "pointer" and "reference" are not keywords. On the other hand, you *can* include anything that comes from the user's source code, including variable names, types, labels, etc. The 'select' format can be used to achieve this sort of thing in a localizable way, see below.

Formatting a Diagnostic Argument

Arguments to diagnostics are fully typed internally, and come from a couple different classes: integers, types, names, and random strings. Depending on the class of the argument, it can be optionally formatted in different ways. This gives the DiagnosticClient information about what the argument means without requiring it to use a specific presentation (consider this MVC for Clang :).

Here are the different diagnostic argument formats currently supported by Clang:

"s" format

Example: `"requires %1 parameter%s1"`

Class: Integers

Description: This is a simple formatter for integers that is useful when producing English diagnostics. When the integer is 1, it prints as nothing. When the integer is not 1, it prints as "s". This allows some simple grammatical forms to be to be handled correctly, and eliminates the need to use gross things like `"requires %1 parameter(s)"`.

"select" format

Example: `"must be a %select{unary|binary|unary or binary}2 operator"`

Class: Integers

Description: This format specifier is used to merge multiple related diagnostics together into one common one, without requiring the difference to be specified as an English string argument. Instead of specifying the string, the diagnostic gets an integer argument and the format string selects the numbered option. In this case, the "%2" value must be an integer in the range [0..2]. If it is 0, it prints 'unary', if it is 1 it prints 'binary' if it is 2, it

prints 'unary or binary'. This allows other language translations to substitute reasonable words (or entire phrases) based on the semantics of the diagnostic instead of having to do things textually.

"plural" format

Example: `"you have %1 %plural{1:mouse|:mice}1 connected to your computer"`

Class: Integers

Description: This is a formatter for complex plural forms. It is designed to handle even the requirements of languages with very complex plural forms, as many Baltic languages have. The argument consists of a series of expression/form pairs, separated by ':', where the first form whose expression evaluates to true is the result of the modifier.

An expression can be empty, in which case it is always true. See the example at the top. Otherwise, it is a series of one or more numeric conditions, separated by ','. If any condition matches, the expression matches. Each numeric condition can take one of three forms.

- number: A simple decimal number matches if the argument is the same as the number. Example: `"%plural{1:mouse|:mice}4"`
- range: A range in square brackets matches if the argument is within the range. Then range is inclusive on both ends. Example: `"%plural{0:none|1:one|[2,5]:some|:many}2"`
- modulo: A modulo operator is followed by a number, and equals sign and either a number or a range. The tests are the same as for plain numbers and ranges, but the argument is taken modulo the number first. Example: `"%plural{%100=0:even hundred|%100=[1,50]:lower half|:everything else}1"`

The parser is very unforgiving. A syntax error, even whitespace, will abort, as will a failure to match the argument against any expression.

"objcclass" format

Example: `"method %objcclass0 not found"`

Class: DeclarationName

Description: This is a simple formatter that indicates the DeclarationName corresponds to an Objective-C class method selector. As such, it prints the selector with a leading '+'.

"objcinstance" format

Example: `"method %objcinstance0 not found"`

Class: DeclarationName

Description: This is a simple formatter that indicates the DeclarationName corresponds to an Objective-C instance method selector. As such, it prints the selector with a leading '-'.

"q" format

Example: `"candidate found by name lookup is %q0"`

Class: NamedDecl*

Description This formatter indicates that the fully-qualified name of the declaration should be printed, e.g., "std::vector" rather than "vector".

It is really easy to add format specifiers to the Clang diagnostics system, but they should be discussed before they are added. If you are creating a lot of repetitive diagnostics and/or have an idea for a useful formatter, please bring it up on the cfe-dev mailing list.

Producing the Diagnostic

Now that you've created the diagnostic in the DiagnosticKinds.def file, you need to write the code that detects the condition in question and emits the new diagnostic. Various components of Clang (e.g. the preprocessor, Sema, etc) provide a helper function named "Diag". It creates a diagnostic and accepts the arguments, ranges, and other information that goes along with it.

For example, the binary expression error comes from code like this:

```
if (various things that are bad)
  Diag(Loc, diag::err_typecheck_invalid_operands)
```

```
      << lex->getType() << rex->getType()
      << lex->getSourceRange() << rex->getSourceRange();
```

This shows that use of the Diag method: they take a location (a SourceLocation object) and a diagnostic enum value (which matches the name from DiagnosticKinds.def). If the diagnostic takes arguments, they are specified with the << operator: the first argument becomes %0, the second becomes %1, etc. The diagnostic interface allows you to specify arguments of many different types, including `int` and `unsigned` for integer arguments, `const char*` and `std::string` for string arguments, `DeclarationName` and `const IdentifierInfo*` for names, `QualType` for types, etc. SourceRanges are also specified with the << operator, but do not have a specific ordering requirement.

As you can see, adding and producing a diagnostic is pretty straightforward. The hard part is deciding exactly what you need to say to help the user, picking a suitable wording, and providing the information needed to format it correctly. The good news is that the call site that issues a diagnostic should be completely independent of how the diagnostic is formatted and in what language it is rendered.

Code Modification Hints

In some cases, the front end emits diagnostics when it is clear that some small change to the source code would fix the problem. For example, a missing semicolon at the end of a statement or a use of deprecated syntax that is easily rewritten into a more modern form. Clang tries very hard to emit the diagnostic and recover gracefully in these and other cases.

However, for these cases where the fix is obvious, the diagnostic can be annotated with a code modification "hint" that describes how to change the code referenced by the diagnostic to fix the problem. For example, it might add the missing semicolon at the end of the statement or rewrite the use of a deprecated construct into something more palatable. Here is one such example C++ front end, where we warn about the right-shift operator changing meaning from C++98 to C++0x:

```
test.cpp:3:7: warning: use of right-shift operator ('>>') in template argument will require parentheses in C++0x
A<100 >> 2> *a;
      ^
  (       )
```

Here, the code modification hint is suggesting that parentheses be added, and showing exactly where those parentheses would be inserted into the source code. The code modification hints themselves describe what changes to make to the source code in an abstract manner, which the text diagnostic printer renders as a line of "insertions" below the caret line. Other diagnostic clients might choose to render the code differently (e.g., as markup inline) or even give the user the ability to automatically fix the problem.

All code modification hints are described by the `CodeModificationHint` class, instances of which should be attached to the diagnostic using the << operator in the same way that highlighted source ranges and arguments are passed to the diagnostic. Code modification hints can be created with one of three constructors:

`CodeModificationHint::CreateInsertion(Loc, Code)`
: Specifies that the given `Code` (a string) should be inserted before the source location `Loc`.

`CodeModificationHint::CreateRemoval(Range)`
: Specifies that the code in the given source `Range` should be removed.

`CodeModificationHint::CreateReplacement(Range, Code)`
: Specifies that the code in the given source `Range` should be removed, and replaced with the given `Code` string.

The DiagnosticClient Interface

Once code generates a diagnostic with all of the arguments and the rest of the relevant information, Clang needs to know what to do with it. As previously mentioned, the diagnostic machinery goes through some filtering to map a severity onto a diagnostic level, then (assuming the diagnostic is not mapped to "`Ignore`") it invokes an object that implements the DiagnosticClient interface with the information.

It is possible to implement this interface in many different ways. For example, the normal Clang DiagnosticClient (named 'TextDiagnosticPrinter') turns the arguments into strings (according to the various formatting rules), prints out the file/line/column information and the string, then prints out the line of code, the source ranges, and the caret. However, this behavior isn't required.

Another implementation of the DiagnosticClient interface is the 'TextDiagnosticBuffer' class, which is used when Clang is in -verify mode. Instead of formatting and printing out the diagnostics, this implementation just captures and remembers the diagnostics as they fly by. Then -verify compares the list of produced diagnostics to the list of expected ones. If they disagree, it prints out its own output.

There are many other possible implementations of this interface, and this is why we prefer diagnostics to pass down rich structured information in arguments. For example, an HTML output might want declaration names be linkified to where they come from in the source. Another example is that a GUI might let you click on typedefs to expand them. This application would want to pass significantly more information about types through to the GUI than a simple flat string. The interface allows this to happen.

Adding Translations to Clang

Not possible yet! Diagnostic strings should be written in UTF-8, the client can translate to the relevant code page if needed. Each translation completely replaces the format string for the diagnostic.

The SourceLocation and SourceManager classes

Strangely enough, the SourceLocation class represents a location within the source code of the program. Important design points include:

1. sizeof(SourceLocation) must be extremely small, as these are embedded into many AST nodes and are passed around often. Currently it is 32 bits.
2. SourceLocation must be a simple value object that can be efficiently copied.
3. We should be able to represent a source location for any byte of any input file. This includes in the middle of tokens, in whitespace, in trigraphs, etc.
4. A SourceLocation must encode the current #include stack that was active when the location was processed. For example, if the location corresponds to a token, it should contain the set of #includes active when the token was lexed. This allows us to print the #include stack for a diagnostic.
5. SourceLocation must be able to describe macro expansions, capturing both the ultimate instantiation point and the source of the original character data.

In practice, the SourceLocation works together with the SourceManager class to encode two pieces of information about a location: it's spelling location and it's instantiation location. For most tokens, these will be the same. However, for a macro expansion (or tokens that came from a _Pragma directive) these will describe the location of the characters corresponding to the token and the location where the token was used (i.e. the macro instantiation point or the location of the _Pragma itself).

For efficiency, we only track one level of macro instantiations: if a token was produced by multiple instantiations, we only track the source and ultimate destination. Though we could track the intermediate instantiation points, this would require extra bookkeeping and no known client would benefit substantially from this.

The Clang front-end inherently depends on the location of a token being tracked correctly. If it is ever incorrect, the front-end may get confused and die. The reason for this is that the notion of the 'spelling' of a Token in Clang depends on being able to find the original input characters for the token. This concept maps directly to the "spelling location" for the token.

The Driver Library

The clang Driver and library are documented here.

Precompiled Headers

Clang supports two implementations of precompiled headers. The default implementation, precompiled headers (PCH) uses a serialized representation of Clang's internal data structures, encoded with the LLVM bitstream format. Pretokenized headers (PTH), on the other hand, contain a serialized representation of the tokens encountered when preprocessing a header (and anything that header includes).

The Frontend Library

The Frontend library contains functionality useful for building tools on top of the clang libraries, for example several methods for outputting diagnostics.

The Lexer and Preprocessor Library

The Lexer library contains several tightly-connected classes that are involved with the nasty process of lexing and preprocessing C source code. The main interface to this library for outside clients is the large Preprocessor class. It contains the various pieces of state that are required to coherently read tokens out of a translation unit.

The core interface to the Preprocessor object (once it is set up) is the Preprocessor::Lex method, which returns the next Token from the preprocessor stream. There are two types of token providers that the preprocessor is capable of reading from: a buffer lexer (provided by the Lexer class) and a buffered token stream (provided by the TokenLexer class).

The Token class

The Token class is used to represent a single lexed token. Tokens are intended to be used by the lexer/preprocess and parser libraries, but are not intended to live beyond them (for example, they should not live in the ASTs).

Tokens most often live on the stack (or some other location that is efficient to access) as the parser is running, but occasionally do get buffered up. For example, macro definitions are stored as a series of tokens, and the C++ front-end periodically needs to buffer tokens up for tentative parsing and various pieces of look-ahead. As such, the size of a Token matter. On a 32-bit system, sizeof(Token) is currently 16 bytes.

Tokens occur in two forms: "Annotation Tokens" and normal tokens. Normal tokens are those returned by the lexer, annotation tokens represent semantic information and are produced by the parser, replacing normal tokens in the token stream. Normal tokens contain the following information:

- **A SourceLocation** - This indicates the location of the start of the token.
- **A length** - This stores the length of the token as stored in the SourceBuffer. For tokens that include them, this length includes trigraphs and escaped newlines which are ignored by later phases of the compiler. By pointing into the original source buffer, it is always possible to get the original spelling of a token completely accurately.
- **IdentifierInfo** - If a token takes the form of an identifier, and if identifier lookup was enabled when the token was lexed (e.g. the lexer was not reading in 'raw' mode) this contains a pointer to the unique hash value for the identifier. Because the lookup happens before keyword identification, this field is set even for language keywords like 'for'.
- **TokenKind** - This indicates the kind of token as classified by the lexer. This includes things like `tok::starequal` (for the "*=" operator), `tok::ampamp` for the "&&" token, and keyword values (e.g. `tok::kw_for`) for identifiers that correspond to keywords. Note that some tokens can be spelled multiple ways. For example, C++ supports "operator keywords", where things like "and" are treated exactly like the "&&" operator. In these cases, the kind value is set to `tok::ampamp`, which is good for the parser, which doesn't have to consider both forms. For something that cares about which form is used (e.g. the preprocessor 'stringize' operator) the spelling indicates the original form.
- **Flags** - There are currently four flags tracked by the lexer/preprocessor system on a per-token basis:
    1. **StartOfLine** - This was the first token that occurred on its input source line.
    2. **LeadingSpace** - There was a space character either immediately before the token or transitively before the token as it was expanded through a macro. The definition of this flag is very closely defined by the stringizing requirements of the preprocessor.
    3. **DisableExpand** - This flag is used internally to the preprocessor to represent identifier tokens which have macro expansion disabled. This prevents them from being considered as candidates for macro expansion ever in the future.
    4. **NeedsCleaning** - This flag is set if the original spelling for the token includes a trigraph or escaped newline. Since this is uncommon, many pieces of code can fast-path on tokens that did not need cleaning.

One interesting (and somewhat unusual) aspect of normal tokens is that they don't contain any semantic information about the lexed value. For example, if the token was a pp-number token, we do not represent the value of the number that was lexed (this is left for later pieces of code to decide). Additionally, the lexer library has no notion of typedef names vs variable names: both are returned as identifiers, and the parser is left to decide whether a specific identifier is a typedef or a

variable (tracking this requires scope information among other things). The parser can do this translation by replacing tokens returned by the preprocessor with "Annotation Tokens".

Annotation Tokens

Annotation Tokens are tokens that are synthesized by the parser and injected into the preprocessor's token stream (replacing existing tokens) to record semantic information found by the parser. For example, if "foo" is found to be a typedef, the "foo" `tok::identifier` token is replaced with an `tok::annot_typename`. This is useful for a couple of reasons: 1) this makes it easy to handle qualified type names (e.g. "foo::bar::baz<42>::t") in C++ as a single "token" in the parser. 2) if the parser backtracks, the reparse does not need to redo semantic analysis to determine whether a token sequence is a variable, type, template, etc.

Annotation Tokens are created by the parser and reinjected into the parser's token stream (when backtracking is enabled). Because they can only exist in tokens that the preprocessor-proper is done with, it doesn't need to keep around flags like "start of line" that the preprocessor uses to do its job. Additionally, an annotation token may "cover" a sequence of preprocessor tokens (e.g. `a::b::c` is five preprocessor tokens). As such, the valid fields of an annotation token are different than the fields for a normal token (but they are multiplexed into the normal Token fields):

- **SourceLocation "Location"** - The SourceLocation for the annotation token indicates the first token replaced by the annotation token. In the example above, it would be the location of the "a" identifier.
- **SourceLocation "AnnotationEndLoc"** - This holds the location of the last token replaced with the annotation token. In the example above, it would be the location of the "c" identifier.
- **void* "AnnotationValue"** - This contains an opaque object that the parser gets from Sema through an Actions module, it is passed around and Sema intepretes it, based on the type of annotation token.
- **TokenKind "Kind"** - This indicates the kind of Annotation token this is. See below for the different valid kinds.

Annotation tokens currently come in three kinds:

1. **tok::annot_typename**: This annotation token represents a resolved typename token that is potentially qualified. The AnnotationValue field contains a pointer returned by Action::getTypeName(). In the case of the Sema actions module, this is a `Decl*` for the type.
2. **tok::annot_cxxscope**: This annotation token represents a C++ scope specifier, such as "A::B::". This corresponds to the grammar productions "::" and ":: [opt] nested-name-specifier". The AnnotationValue pointer is returned by the Action::ActOnCXXGlobalScopeSpecifier and Action::ActOnCXXNestedNameSpecifier callbacks. In the case of Sema, this is a `DeclContext*`.
3. **tok::annot_template_id**: This annotation token represents a C++ template-id such as "foo<int, 4>", where "foo" is the name of a template. The AnnotationValue pointer is a pointer to a malloc'd TemplateIdAnnotation object. Depending on the context, a parsed template-id that names a type might become a typename annotation token (if all we care about is the named type, e.g., because it occurs in a type specifier) or might remain a template-id token (if we want to retain more source location information or produce a new type, e.g., in a declaration of a class template specialization). template-id annotation tokens that refer to a type can be "upgraded" to typename annotation tokens by the parser.

As mentioned above, annotation tokens are not returned by the preprocessor, they are formed on demand by the parser. This means that the parser has to be aware of cases where an annotation could occur and form it where appropriate. This is somewhat similar to how the parser handles Translation Phase 6 of C99: String Concatenation (see C99 5.1.1.2). In the case of string concatenation, the preprocessor just returns distinct tok::string_literal and tok::wide_string_literal tokens and the parser eats a sequence of them wherever the grammar indicates that a string literal can occur.

In order to do this, whenever the parser expects a tok::identifier or tok::coloncolon, it should call the TryAnnotateTypeOrScopeToken or TryAnnotateCXXScopeToken methods to form the annotation token. These methods will maximally form the specified annotation tokens and replace the current token with them, if applicable. If the current tokens is not valid for an annotation token, it will remain an identifier or :: token.

The Lexer class

The Lexer class provides the mechanics of lexing tokens out of a source buffer and deciding what they mean. The Lexer is complicated by the fact that it operates on raw buffers that have not had spelling eliminated (this is a necessity to get decent

performance), but this is countered with careful coding as well as standard performance techniques (for example, the comment handling code is vectorized on X86 and PowerPC hosts).

The lexer has a couple of interesting modal features:

- The lexer can operate in 'raw' mode. This mode has several features that make it possible to quickly lex the file (e.g. it stops identifier lookup, doesn't specially handle preprocessor tokens, handles EOF differently, etc). This mode is used for lexing within an "`#if 0`" block, for example.
- The lexer can capture and return comments as tokens. This is required to support the -C preprocessor mode, which passes comments through, and is used by the diagnostic checker to identifier expect-error annotations.
- The lexer can be in ParsingFilename mode, which happens when preprocessing after reading a #include directive. This mode changes the parsing of '<' to return an "angled string" instead of a bunch of tokens for each thing within the filename.
- When parsing a preprocessor directive (after "`#`") the ParsingPreprocessorDirective mode is entered. This changes the parser to return EOM at a newline.
- The Lexer uses a LangOptions object to know whether trigraphs are enabled, whether C++ or ObjC keywords are recognized, etc.

In addition to these modes, the lexer keeps track of a couple of other features that are local to a lexed buffer, which change as the buffer is lexed:

- The Lexer uses BufferPtr to keep track of the current character being lexed.
- The Lexer uses IsAtStartOfLine to keep track of whether the next lexed token will start with its "start of line" bit set.
- The Lexer keeps track of the current #if directives that are active (which can be nested).
- The Lexer keeps track of an MultipleIncludeOpt object, which is used to detect whether the buffer uses the standard "`#ifndef XX` / `#define XX`" idiom to prevent multiple inclusion. If a buffer does, subsequent includes can be ignored if the XX macro is defined.

The TokenLexer class

The TokenLexer class is a token provider that returns tokens from a list of tokens that came from somewhere else. It typically used for two things: 1) returning tokens from a macro definition as it is being expanded 2) returning tokens from an arbitrary buffer of tokens. The later use is used by _Pragma and will most likely be used to handle unbounded look-ahead for the C++ parser.

The MultipleIncludeOpt class

The MultipleIncludeOpt class implements a really simple little state machine that is used to detect the standard "`#ifndef XX` / `#define XX`" idiom that people typically use to prevent multiple inclusion of headers. If a buffer uses this idiom and is subsequently #include'd, the preprocessor can simply check to see whether the guarding condition is defined or not. If so, the preprocessor can completely ignore the include of the header.

The Parser Library

The AST Library

The Type class and its subclasses

The Type class (and its subclasses) are an important part of the AST. Types are accessed through the ASTContext class, which implicitly creates and uniques them as they are needed. Types have a couple of non-obvious features: 1) they do not capture type qualifiers like const or volatile (See QualType), and 2) they implicitly capture typedef information. Once created, types are immutable (unlike decls).

Typedefs in C make semantic analysis a bit more complex than it would be without them. The issue is that we want to capture typedef information and represent it in the AST perfectly, but the semantics of operations need to "see through" typedefs. For example, consider this code:

```
void func() {
  typedef int foo;
  foo X, *Y;
  typedef foo* bar;
  bar Z;
  *X; // error
  **Y; // error
  **Z; // error
}
```

The code above is illegal, and thus we expect there to be diagnostics emitted on the annotated lines. In this example, we expect to get:

```
test.c:6:1: error: indirection requires pointer operand ('foo' invalid)
*X; // error
^~
test.c:7:1: error: indirection requires pointer operand ('foo' invalid)
**Y; // error
^~~
test.c:8:1: error: indirection requires pointer operand ('foo' invalid)
**Z; // error
^~~
```

While this example is somewhat silly, it illustrates the point: we want to retain typedef information where possible, so that we can emit errors about "`std::string`" instead of "`std::basic_string<char, std:...`". Doing this requires properly keeping typedef information (for example, the type of "X" is "foo", not "int"), and requires properly propagating it through the various operators (for example, the type of *Y is "foo", not "int"). In order to retain this information, the type of these expressions is an instance of the TypedefType class, which indicates that the type of these expressions is a typedef for foo.

Representing types like this is great for diagnostics, because the user-specified type is always immediately available. There are two problems with this: first, various semantic checks need to make judgements about the *actual structure* of a type, ignoring typdefs. Second, we need an efficient way to query whether two types are structurally identical to each other, ignoring typedefs. The solution to both of these problems is the idea of canonical types.

Canonical Types

Every instance of the Type class contains a canonical type pointer. For simple types with no typedefs involved (e.g. "`int`", "`int*`", "`int**`"), the type just points to itself. For types that have a typedef somewhere in their structure (e.g. "`foo`", "`foo*`", "`foo**`", "`bar`"), the canonical type pointer points to their structurally equivalent type without any typedefs (e.g. "`int`", "`int*`", "`int**`", and "`int*`" respectively).

This design provides a constant time operation (dereferencing the canonical type pointer) that gives us access to the structure of types. For example, we can trivially tell that "bar" and "foo*" are the same type by dereferencing their canonical type pointers and doing a pointer comparison (they both point to the single "`int*`" type).

Canonical types and typedef types bring up some complexities that must be carefully managed. Specifically, the "isa/cast/dyncast" operators generally shouldn't be used in code that is inspecting the AST. For example, when type checking the indirection operator (unary '*' on a pointer), the type checker must verify that the operand has a pointer type. It would not be correct to check that with "`isa<PointerType>(SubExpr->getType())`", because this predicate would fail if the subexpression had a typedef type.

The solution to this problem are a set of helper methods on Type, used to check their properties. In this case, it would be correct to use "`SubExpr->getType()->isPointerType()`" to do the check. This predicate will return true if the *canonical type is a pointer*, which is true any time the type is structurally a pointer type. The only hard part here is remembering not to use the `isa/cast/dyncast` operations.

The second problem we face is how to get access to the pointer type once we know it exists. To continue the example, the result type of the indirection operator is the pointee type of the subexpression. In order to determine the type, we need to get the instance of PointerType that best captures the typedef information in the program. If the type of the expression is literally a PointerType, we can return that, otherwise we have to dig through the typedefs to find the pointer type. For example, if the subexpression had type "`foo*`", we could return that type as the result. If the subexpression had type "`bar`", we want to return "`foo*`" (note that we do *not* want "`int*`"). In order to provide all of this, Type has a getAsPointerType()

method that checks whether the type is structurally a PointerType and, if so, returns the best one. If not, it returns a null pointer.

This structure is somewhat mystical, but after meditating on it, it will make sense to you :).

The QualType class

The QualType class is designed as a trivial value class that is small, passed by-value and is efficient to query. The idea of QualType is that it stores the type qualifiers (const, volatile, restrict) separately from the types themselves: QualType is conceptually a pair of "Type*" and bits for the type qualifiers.

By storing the type qualifiers as bits in the conceptual pair, it is extremely efficient to get the set of qualifiers on a QualType (just return the field of the pair), add a type qualifier (which is a trivial constant-time operation that sets a bit), and remove one or more type qualifiers (just return a QualType with the bitfield set to empty).

Further, because the bits are stored outside of the type itself, we do not need to create duplicates of types with different sets of qualifiers (i.e. there is only a single heap allocated "int" type: "const int" and "volatile const int" both point to the same heap allocated "int" type). This reduces the heap size used to represent bits and also means we do not have to consider qualifiers when uniquing types (Type does not even contain qualifiers).

In practice, on hosts where it is safe, the 3 type qualifiers are stored in the low bit of the pointer to the Type object. This means that QualType is exactly the same size as a pointer, and this works fine on any system where malloc'd objects are at least 8 byte aligned.

Declaration names

The `DeclarationName` class represents the name of a declaration in Clang. Declarations in the C family of languages can take several different forms. Most declarations are named by simple identifiers, e.g., "`f`" and "`x`" in the function declaration `f(int x)`. In C++, declaration names can also name class constructors ("`Class`" in `struct Class { Class(); }`), class destructors ("`~Class`"), overloaded operator names ("operator+"), and conversion functions ("`operator void const *`"). In Objective-C, declaration names can refer to the names of Objective-C methods, which involve the method name and the parameters, collectively called a *selector*, e.g., "`setWidth:height:`". Since all of these kinds of entities - variables, functions, Objective-C methods, C++ constructors, destructors, and operators - are represented as subclasses of Clang's common `NamedDecl` class, `DeclarationName` is designed to efficiently represent any kind of name.

Given a `DeclarationName` `N`, `N.getNameKind()` will produce a value that describes what kind of name `N` stores. There are 8 options (all of the names are inside the `DeclarationName` class)

Identifier
: The name is a simple identifier. Use `N.getAsIdentifierInfo()` to retrieve the corresponding `IdentifierInfo*` pointing to the actual identifier. Note that C++ overloaded operators (e.g., "`operator+`") are represented as special kinds of identifiers. Use `IdentifierInfo`'s `getOverloadedOperatorID` function to determine whether an identifier is an overloaded operator name.

ObjCZeroArgSelector, ObjCOneArgSelector, ObjCMultiArgSelector
: The name is an Objective-C selector, which can be retrieved as a `Selector` instance via `N.getObjCSelector()`. The three possible name kinds for Objective-C reflect an optimization within the `DeclarationName` class: both zero- and one-argument selectors are stored as a masked `IdentifierInfo` pointer, and therefore require very little space, since zero- and one-argument selectors are far more common than multi-argument selectors (which use a different structure).

CXXConstructorName
: The name is a C++ constructor name. Use `N.getCXXNameType()` to retrieve the type that this constructor is meant to construct. The type is always the canonical type, since all constructors for a given type have the same name.

CXXDestructorName
: The name is a C++ destructor name. Use `N.getCXXNameType()` to retrieve the type whose destructor is being named. This type is always a canonical type.

CXXConversionFunctionName
: The name is a C++ conversion function. Conversion functions are named according to the type they convert to, e.g., "`operator void const *`". Use `N.getCXXNameType()` to retrieve the type that this conversion function converts to. This type is always a canonical type.

CXXOperatorName
: The name is a C++ overloaded operator name. Overloaded operators are named according to their spelling, e.g., "`operator+`" or "`operator new []`". Use `N.getCXXOverloadedOperator()` to retrieve the overloaded operator (a value of type `OverloadedOperatorKind`).

`DeclarationNames` are cheap to create, copy, and compare. They require only a single pointer's worth of storage in the common cases (identifiers, zero- and one-argument Objective-C selectors) and use dense, uniqued storage for the other kinds of names. Two `DeclarationNames` can be compared for equality (`==`, `!=`) using a simple bitwise comparison, can be ordered with `<`, `>`, `<=`, and `>=` (which provide a lexicographical ordering for normal identifiers but an unspecified ordering for other kinds of names), and can be placed into LLVM `DenseMaps` and `DenseSets`.

`DeclarationName` instances can be created in different ways depending on what kind of name the instance will store. Normal identifiers (`IdentifierInfo` pointers) and Objective-C selectors (`Selector`) can be implicitly converted to `DeclarationNames`. Names for C++ constructors, destructors, conversion functions, and overloaded operators can be retrieved from the `DeclarationNameTable`, an instance of which is available as `ASTContext::DeclarationNames`. The member functions `getCXXConstructorName`, `getCXXDestructorName`, `getCXXConversionFunctionName`, and `getCXXOperatorName`, respectively, return `DeclarationName` instances for the four kinds of C++ special function names.

Declaration contexts

Every declaration in a program exists within some *declaration context*, such as a translation unit, namespace, class, or function. Declaration contexts in Clang are represented by the `DeclContext` class, from which the various declaration-context AST nodes (`TranslationUnitDecl`, `NamespaceDecl`, `RecordDecl`, `FunctionDecl`, etc.) will derive. The `DeclContext` class provides several facilities common to each declaration context:

Source-centric vs. Semantics-centric View of Declarations
: `DeclContext` provides two views of the declarations stored within a declaration context. The source-centric view accurately represents the program source code as written, including multiple declarations of entities where present (see the section Redeclarations and Overloads), while the semantics-centric view represents the program semantics. The two views are kept synchronized by semantic analysis while the ASTs are being constructed.

Storage of declarations within that context
: Every declaration context can contain some number of declarations. For example, a C++ class (represented by `RecordDecl`) contains various member functions, fields, nested types, and so on. All of these declarations will be stored within the `DeclContext`, and one can iterate over the declarations via [`DeclContext::decls_begin()`, `DeclContext::decls_end()`). This mechanism provides the source-centric view of declarations in the context.

Lookup of declarations within that context
: The `DeclContext` structure provides efficient name lookup for names within that declaration context. For example, if `N` is a namespace we can look for the name `N::f` using `DeclContext::lookup`. The lookup itself is based on a lazily-constructed array (for declaration contexts with a small number of declarations) or hash table (for declaration contexts with more declarations). The lookup operation provides the semantics-centric view of the declarations in the context.

Ownership of declarations
: The `DeclContext` owns all of the declarations that were declared within its declaration context, and is responsible for the management of their memory as well as their (de-)serialization.

All declarations are stored within a declaration context, and one can query information about the context in which each declaration lives. One can retrieve the `DeclContext` that contains a particular `Decl` using `Decl::getDeclContext`. However, see the section Lexical and Semantic Contexts for more information about how to interpret this context information.

Redeclarations and Overloads

Within a translation unit, it is common for an entity to be declared several times. For example, we might declare a function "f" and then later re-declare it as part of an inlined definition:

```
void f(int x, int y, int z = 1);

inline void f(int x, int y, int z) { /* ... */ }
```

The representation of "f" differs in the source-centric and semantics-centric views of a declaration context. In the source-centric view, all redeclarations will be present, in the order they occurred in the source code, making this view suitable for

clients that wish to see the structure of the source code. In the semantics-centric view, only the most recent "f" will be found by the lookup, since it effectively replaces the first declaration of "f".

In the semantics-centric view, overloading of functions is represented explicitly. For example, given two declarations of a function "g" that are overloaded, e.g.,

```
void g();
void g(int);
```

the `DeclContext::lookup` operation will return an `OverloadedFunctionDecl` that contains both declarations of "g". Clients that perform semantic analysis on a program that is not concerned with the actual source code will primarily use this semantics-centric view.

Lexical and Semantic Contexts

Each declaration has two potentially different declaration contexts: a *lexical* context, which corresponds to the source-centric view of the declaration context, and a *semantic* context, which corresponds to the semantics-centric view. The lexical context is accessible via `Decl::getLexicalDeclContext` while the semantic context is accessible via `Decl::getDeclContext`, both of which return `DeclContext` pointers. For most declarations, the two contexts are identical. For example:

```
class X {
public:
  void f(int x);
};
```

Here, the semantic and lexical contexts of `X::f` are the `DeclContext` associated with the class `X` (itself stored as a `RecordDecl` AST node). However, we can now define `X::f` out-of-line:

```
void X::f(int x = 17) { /* ... */ }
```

This definition of has different lexical and semantic contexts. The lexical context corresponds to the declaration context in which the actual declaration occurred in the source code, e.g., the translation unit containing `X`. Thus, this declaration of `X::f` can be found by traversing the declarations provided by [`decls_begin()`, `decls_end()`) in the translation unit.

The semantic context of `X::f` corresponds to the class `X`, since this member function is (semantically) a member of `X`. Lookup of the name `f` into the `DeclContext` associated with `X` will then return the definition of `X::f` (including information about the default argument).

Transparent Declaration Contexts

In C and C++, there are several contexts in which names that are logically declared inside another declaration will actually "leak" out into the enclosing scope from the perspective of name lookup. The most obvious instance of this behavior is in enumeration types, e.g.,

```
enum Color {
  Red,
  Green,
  Blue
};
```

Here, `Color` is an enumeration, which is a declaration context that contains the enumerators `Red`, `Green`, and `Blue`. Thus, traversing the list of declarations contained in the enumeration `Color` will yield `Red`, `Green`, and `Blue`. However, outside of the scope of `Color` one can name the enumerator `Red` without qualifying the name, e.g.,

```
Color c = Red;
```

There are other entities in C++ that provide similar behavior. For example, linkage specifications that use curly braces:

```
extern "C" {
  void f(int);
  void g(int);
```

```
}
// f and g are visible here
```

For source-level accuracy, we treat the linkage specification and enumeration type as a declaration context in which its enclosed declarations ("Red", "Green", and "Blue"; "f" and "g") are declared. However, these declarations are visible outside of the scope of the declaration context.

These language features (and several others, described below) have roughly the same set of requirements: declarations are declared within a particular lexical context, but the declarations are also found via name lookup in scopes enclosing the declaration itself. This feature is implemented via *transparent* declaration contexts (see `DeclContext::isTransparentContext()`), whose declarations are visible in the nearest enclosing non-transparent declaration context. This means that the lexical context of the declaration (e.g., an enumerator) will be the transparent `DeclContext` itself, as will the semantic context, but the declaration will be visible in every outer context up to and including the first non-transparent declaration context (since transparent declaration contexts can be nested).

The transparent `DeclContexts` are:

- Enumerations (but not C++0x "scoped enumerations"):

  ```
  enum Color {
    Red,
    Green,
    Blue
  };
  // Red, Green, and Blue are in scope
  ```

- C++ linkage specifications:

  ```
  extern "C" {
    void f(int);
    void g(int);
  }
  // f and g are in scope
  ```

- Anonymous unions and structs:

  ```
  struct LookupTable {
    bool IsVector;
    union {
      std::vector<Item> *Vector;
      std::set<Item> *Set;
    };
  };

  LookupTable LT;
  LT.Vector = 0; // Okay: finds Vector inside the unnamed union
  ```

- C++0x inline namespaces:

  ```
  namespace mylib {
    inline namespace debug {
      class X;
    }
  }
  mylib::X *xp; // okay: mylib::X refers to mylib::debug::X
  ```

Multiply-Defined Declaration Contexts

C++ namespaces have the interesting--and, so far, unique--property that the namespace can be defined multiple times, and the declarations provided by each namespace definition are effectively merged (from the semantic point of view). For example, the following two code snippets are semantically indistinguishable:

```
// Snippet #1:
namespace N {
```

```
  void f();
}
namespace N {
  void f(int);
}

// Snippet #2:
namespace N {
  void f();
  void f(int);
}
```

In Clang's representation, the source-centric view of declaration contexts will actually have two separate `NamespaceDecl` nodes in Snippet #1, each of which is a declaration context that contains a single declaration of "f". However, the semantics-centric view provided by name lookup into the namespace `N` for "f" will return an `OverloadedFunctionDecl` that contains both declarations of "f".

`DeclContext` manages multiply-defined declaration contexts internally. The function `DeclContext::getPrimaryContext` retrieves the "primary" context for a given `DeclContext` instance, which is the `DeclContext` responsible for maintaining the lookup table used for the semantics-centric view. Given the primary context, one can follow the chain of `DeclContext` nodes that define additional declarations via `DeclContext::getNextContext`. Note that these functions are used internally within the lookup and insertion methods of the `DeclContext`, so the vast majority of clients can ignore them.

The `CFG` class

The `CFG` class is designed to represent a source-level control-flow graph for a single statement (`Stmt*`). Typically instances of `CFG` are constructed for function bodies (usually an instance of `CompoundStmt`), but can also be instantiated to represent the control-flow of any class that subclasses `Stmt`, which includes simple expressions. Control-flow graphs are especially useful for performing flow- or path-sensitive program analyses on a given function.

Basic Blocks

Concretely, an instance of `CFG` is a collection of basic blocks. Each basic block is an instance of `CFGBlock`, which simply contains an ordered sequence of `Stmt*` (each referring to statements in the AST). The ordering of statements within a block indicates unconditional flow of control from one statement to the next. Conditional control-flow is represented using edges between basic blocks. The statements within a given `CFGBlock` can be traversed using the `CFGBlock::*iterator` interface.

A `CFG` object owns the instances of `CFGBlock` within the control-flow graph it represents. Each `CFGBlock` within a CFG is also uniquely numbered (accessible via `CFGBlock::getBlockID()`). Currently the number is based on the ordering the blocks were created, but no assumptions should be made on how `CFGBlocks` are numbered other than their numbers are unique and that they are numbered from 0..N-1 (where N is the number of basic blocks in the CFG).

Entry and Exit Blocks

Each instance of `CFG` contains two special blocks: an *entry* block (accessible via `CFG::getEntry()`), which has no incoming edges, and an *exit* block (accessible via `CFG::getExit()`), which has no outgoing edges. Neither block contains any statements, and they serve the role of providing a clear entrance and exit for a body of code such as a function body. The presence of these empty blocks greatly simplifies the implementation of many analyses built on top of CFGs.

Conditional Control-Flow

Conditional control-flow (such as those induced by if-statements and loops) is represented as edges between `CFGBlocks`. Because different C language constructs can induce control-flow, each `CFGBlock` also records an extra `Stmt*` that represents the *terminator* of the block. A terminator is simply the statement that caused the control-flow, and is used to identify the nature of the conditional control-flow between blocks. For example, in the case of an if-statement, the terminator refers to the `IfStmt` object in the AST that represented the given branch.

To illustrate, consider the following code example:

```
int foo(int x) {
  x = x + 1;

  if (x > 2) x++;
  else {
    x += 2;
    x *= 2;
  }

  return x;
}
```

After invoking the parser+semantic analyzer on this code fragment, the AST of the body of `foo` is referenced by a single `Stmt*`. We can then construct an instance of `CFG` representing the control-flow graph of this function body by single call to a static class method:

```
  Stmt* FooBody = ...
  CFG* FooCFG = CFG::buildCFG(FooBody);
```

It is the responsibility of the caller of `CFG::buildCFG` to `delete` the returned `CFG*` when the CFG is no longer needed.

Along with providing an interface to iterate over its `CFGBlocks`, the `CFG` class also provides methods that are useful for debugging and visualizing CFGs. For example, the method `CFG::dump()` dumps a pretty-printed version of the CFG to standard error. This is especially useful when one is using a debugger such as gdb. For example, here is the output of `FooCFG->dump()`:

```
 [ B5 (ENTRY) ]
    Predecessors (0):
    Successors (1): B4

 [ B4 ]
    1: x = x + 1
    2: (x > 2)
    T: if [B4.2]
    Predecessors (1): B5
    Successors (2): B3 B2

 [ B3 ]
    1: x++
    Predecessors (1): B4
    Successors (1): B1

 [ B2 ]
    1: x += 2
    2: x *= 2
    Predecessors (1): B4
    Successors (1): B1

 [ B1 ]
    1: return x;
    Predecessors (2): B2 B3
    Successors (1): B0

 [ B0 (EXIT) ]
    Predecessors (1): B1
    Successors (0):
```

For each block, the pretty-printed output displays for each block the number of *predecessor* blocks (blocks that have outgoing control-flow to the given block) and *successor* blocks (blocks that have control-flow that have incoming control-flow from the given block). We can also clearly see the special entry and exit blocks at the beginning and end of the pretty-printed output. For the entry block (block B5), the number of predecessor blocks is 0, while for the exit block (block B0) the number of successor blocks is 0.

The most interesting block here is B4, whose outgoing control-flow represents the branching caused by the sole if-statement in `foo`. Of particular interest is the second statement in the block, **`(x > 2)`**, and the terminator, printed as **`if [B4.2]`**. The second statement represents the evaluation of the condition of the if-statement, which occurs before the actual branching of control-flow. Within the `CFGBlock` for B4, the `Stmt*` for the second statement refers to the actual expression in

the AST for `(x > 2)`. Thus pointers to subclasses of `Expr` can appear in the list of statements in a block, and not just subclasses of `Stmt` that refer to proper C statements.

The terminator of block B4 is a pointer to the `IfStmt` object in the AST. The pretty-printer outputs `if [B4.2]` because the condition expression of the if-statement has an actual place in the basic block, and thus the terminator is essentially *referring* to the expression that is the second statement of block B4 (i.e., B4.2). In this manner, conditions for control-flow (which also includes conditions for loops and switch statements) are hoisted into the actual basic block.

Constant Folding in the Clang AST

There are several places where constants and constant folding matter a lot to the Clang front-end. First, in general, we prefer the AST to retain the source code as close to how the user wrote it as possible. This means that if they wrote "5+4", we want to keep the addition and two constants in the AST, we don't want to fold to "9". This means that constant folding in various ways turns into a tree walk that needs to handle the various cases.

However, there are places in both C and C++ that require constants to be folded. For example, the C standard defines what an "integer constant expression" (i-c-e) is with very precise and specific requirements. The language then requires i-c-e's in a lot of places (for example, the size of a bitfield, the value for a case statement, etc). For these, we have to be able to constant fold the constants, to do semantic checks (e.g. verify bitfield size is non-negative and that case statements aren't duplicated). We aim for Clang to be very pedantic about this, diagnosing cases when the code does not use an i-c-e where one is required, but accepting the code unless running with `-pedantic-errors`.

Things get a little bit more tricky when it comes to compatibility with real-world source code. Specifically, GCC has historically accepted a huge superset of expressions as i-c-e's, and a lot of real world code depends on this unfortuate accident of history (including, e.g., the glibc system headers). GCC accepts anything its "fold" optimizer is capable of reducing to an integer constant, which means that the definition of what it accepts changes as its optimizer does. One example is that GCC accepts things like "case X-X:" even when X is a variable, because it can fold this to 0.

Another issue are how constants interact with the extensions we support, such as __builtin_constant_p, __builtin_inf, __extension__ and many others. C99 obviously does not specify the semantics of any of these extensions, and the definition of i-c-e does not include them. However, these extensions are often used in real code, and we have to have a way to reason about them.

Finally, this is not just a problem for semantic analysis. The code generator and other clients have to be able to fold constants (e.g. to initialize global variables) and has to handle a superset of what C99 allows. Further, these clients can benefit from extended information. For example, we know that "foo()||1" always evaluates to true, but we can't replace the expression with true because it has side effects.

Implementation Approach

After trying several different approaches, we've finally converged on a design (Note, at the time of this writing, not all of this has been implemented, consider this a design goal!). Our basic approach is to define a single recursive method evaluation method (`Expr::Evaluate`), which is implemented in `AST/ExprConstant.cpp`. Given an expression with 'scalar' type (integer, fp, complex, or pointer) this method returns the following information:

- Whether the expression is an integer constant expression, a general constant that was folded but has no side effects, a general constant that was folded but that does have side effects, or an uncomputable/unfoldable value.
- If the expression was computable in any way, this method returns the APValue for the result of the expression.
- If the expression is not evaluatable at all, this method returns information on one of the problems with the expression. This includes a SourceLocation for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should be have ERROR type.
- If the expression is not an integer constant expression, this method returns information on one of the problems with the expression. This includes a SourceLocation for where the problem is, and a diagnostic ID that explains the problem. The diagnostic should be have EXTENSION type.

This information gives various clients the flexibility that they want, and we will eventually have some helper methods for various extensions. For example, Sema should have a `Sema::VerifyIntegerConstantExpression` method, which calls Evaluate on the expression. If the expression is not foldable, the error is emitted, and it would return true. If the expression is not an i-c-e, the EXTENSION diagnostic is emitted. Finally it would return false to indicate that the AST is ok.

Other clients can use the information in other ways, for example, codegen can just use expressions that are foldable in any way.

Extensions

This section describes how some of the various extensions Clang supports interacts with constant evaluation:

- **`__extension__`**: The expression form of this extension causes any evaluatable subexpression to be accepted as an integer constant expression.
- **`__builtin_constant_p`**: This returns true (as a integer constant expression) if the operand is any evaluatable constant. As a special case, if `__builtin_constant_p` is the (potentially parenthesized) condition of a conditional operator expression ("?:"), only the true side of the conditional operator is considered, and it is evaluated with full constant folding.
- **`__builtin_choose_expr`**: The condition is required to be an integer constant expression, but we accept any constant as an "extension of an extension". This only evaluates one operand depending on which way the condition evaluates.
- **`__builtin_classify_type`**: This always returns an integer constant expression.
- **`__builtin_inf,nan,..`**: These are treated just like a floating-point literal.
- **`__builtin_abs,copysign,..`**: These are constant folded as general constant expressions.