"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 Support Library

The LLVM `libSupport` library provides many underlying libraries and data-structures, including command line option processing, various containers and a system abstraction layer, which is used for file system access.

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, an English translation associated with it, a :ref:`SourceLocation <SourceLocation>` to "put the caret", and a severity (e.g., `WARNING` or `ERROR`). They can also optionally include a number of arguments to the diagnostic (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 :ref:`rendered in many different ways <DiagnosticConsumer>` depending on how the `DiagnosticConsumer` interface is implemented. A representative example of a diagnostic 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 `Diagnostic*Kinds.td` *files*

Diagnostics are created by adding an entry to one of the `clang/Basic/Diagnostic*Kinds.td` files, depending on what library will be using it. From this file, :program:`tblgen` generates the unique ID of the diagnostic, the severity of the diagnostic 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`, `REMARK`, `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 `REMARK` severity provides generic information about the compilation that is not necessarily related to any dubious code. 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`, `Remark`, `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 :ref:`produces them <internals-producing-diag>`, 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.td` 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 :ref:`translating <internals-diag-translation>` 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 `DiagnosticConsumer` 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. The selected string does undergo formatting.

"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.

"ordinal" format

Example:

```
"ambiguity in %ordinal0 argument"
```

Class:

Integers

Description:

This is a formatter which represents the argument number as an ordinal: the value `1` becomes `1st`, `3` becomes `3rd`, and so on. Values less than `1` are not supported. This formatter is currently hard-coded to use English ordinals.

"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`".

"diff" format

Example:

```
"no known conversion %diff{from $ to $|from argument type to parameter type}1,2"
```

Class:

`QualType`

Description:

This formatter takes two `QualTypes` and attempts to print a template difference between the two. If tree printing is off, the text inside the braces before the pipe is printed, with the formatted text replacing the $. If tree printing is on, the text after the pipe is printed and a type tree is printed after the diagnostic message.

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.

"sub" format

Example:

Given the following record definition of type `TextSubstitution`:

```
def select_ovl_candidate : TextSubstitution<
  "%select{function|constructor}0%select{| template| %2}1">;
```

which can be used as

```
def note_ovl_candidate : Note<
  "candidate %sub{select_ovl_candidate}3,2,1 not viable">;
```

and will act as if it was written `"candidate %select{function|constructor}3%select{| template| %1}2 not viable"`.

Description:

This format specifier is used to avoid repeating strings verbatim in multiple diagnostics. The argument to `%sub` must name a `TextSubstitution` tblgen record. The substitution must specify all arguments used by the substitution, and the modifier indexes in the substitution are re-numbered accordingly. The substituted text must itself be a valid format string before substitution.

Producing the Diagnostic

Now that you've created the diagnostic in the `Diagnostic*Kinds.td` 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: it takes a location (a :ref:`SourceLocation <SourceLocation>` object) and a diagnostic enum value (which matches the name from `Diagnostic*Kinds.td`). 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.

Fix-It 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 hint (referred to as a "fix-it 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 from the C++ front end, where we warn about the right-shift operator changing meaning from C++98 to C++11:

```
test.cpp:3:7: warning: use of right-shift operator ('>>') in template argument
                       will require parentheses in C++11
A<100 >> 2> *a;
      ^
  (       )
```

Here, the fix-it hint is suggesting that parentheses be added, and showing exactly where those parentheses would be inserted into the source code. The fix-it 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. :ref:`Other diagnostic clients <DiagnosticConsumer>` might choose to render the code differently (e.g., as markup inline) or even give the user the ability to automatically fix the problem.

Fix-it hints on errors and warnings need to obey these rules:

- Since they are automatically applied if `-Xclang -fixit` is passed to the driver, they should only be used when it's very likely they match the user's intent.
- Clang must recover from errors as if the fix-it had been applied.
- Fix-it hints on a warning must not change the meaning of the code. However, a hint may clarify the meaning as intentional, for example by adding parentheses when the precedence of operators isn't obvious.

If a fix-it can't obey these rules, put the fix-it on a note. Fix-its on notes are not applied automatically.

All fix-it hints are described by the `FixItHint` 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. Fix-it hints can be created with one of three constructors:

- `FixItHint::CreateInsertion(Loc, Code)`

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

- `FixItHint::CreateRemoval(Range)`

  Specifies that the code in the given source `Range` should be removed.

- `FixItHint::CreateReplacement(Range, Code)`

  Specifies that the code in the given source `Range` should be removed, and replaced with the given `Code` string.

The `DiagnosticConsumer` *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 `DiagnosticConsumer` interface with the information.

It is possible to implement this interface in many different ways. For example, the normal Clang `DiagnosticConsumer` (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 `DiagnosticConsumer` 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. Full documentation for the `-verify` mode can be found in the Clang API documentation for VerifyDiagnosticConsumer.

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: its spelling location and its expansion 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 expansion point or the location of the `_Pragma` itself).

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.

`SourceRange` and `CharSourceRange`

Clang represents most source ranges by [first, last], where "first" and "last" each point to the beginning of their respective tokens. For example consider the `SourceRange` of the following statement:

```
x = foo + bar;
^first    ^last
```

To map from this representation to a character-based representation, the "last" location needs to be adjusted to point to (or past) the end of that token with either `Lexer::MeasureTokenLength()` or `Lexer::getLocForEndOfToken()`. For the rare cases where character-level source ranges information is needed we use the `CharSourceRange` class.

The Driver Library

The clang Driver and library are documented :doc:`here <DriverInternals>`.

Precompiled Headers

Clang supports precompiled headers (:doc:`PCH <PCHInternals>`), which uses a serialized representation of Clang's internal data structures, encoded with the LLVM bitstream format.

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 :ref:`Token <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 :ref:`Lexer <Lexer>` class) and a buffered token stream (provided by the :ref:`TokenLexer <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` matters. On a 32-bit system, `sizeof(Token)` is currently 16 bytes.

Tokens occur in two forms: :ref:`annotation tokens <AnnotationToken>` 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`. The parser merely preserves the information for `Sema` to later interpret based on the annotation token kind.
- **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 the `QualType` returned by `Sema::getTypeName()`, possibly with source location information attached.
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 a `NestedNameSpecifier *` returned by the `Sema::ActOnCXXGlobalScopeSpecifier` and `Sema::ActOnCXXNestedNameSpecifier` callbacks.
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 EOD 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 :ref:`MultipleIncludeOpt <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

This library contains a recursive-descent parser that polls tokens from the preprocessor and notifies a client of the parsing progress.

Historically, the parser used to talk to an abstract `Action` interface that had virtual methods for parse events, for example `ActOnBinOp()`. When Clang grew C++ support, the parser stopped supporting general `Action` clients -- it now always talks to the :ref:`Sema library <Sema>`. However, the Parser still accesses AST objects only through opaque types like `ExprResult` and `StmtResult`. Only :ref:`Sema <Sema>` looks at the AST node contents of these wrappers.

The AST Library

Design philosophy

Immutability

Clang AST nodes (types, declarations, statements, expressions, and so on) are generally designed to be immutable once created. This provides a number of key benefits:

- Canonicalization of the "meaning" of nodes is possible as soon as the nodes are created, and is not invalidated by later addition of more information. For example, we :ref:`canonicalize types <CanonicalType>`, and use a canonicalized representation of expressions when determining whether two function template declarations involving dependent expressions declare the same entity.
- AST nodes can be reused when they have the same meaning. For example, we reuse `Type` nodes when representing the same type (but maintain separate `TypeLocs` for each instance where a type is written), and we reuse non-dependent `Stmt` and `Expr` nodes across instantiations of a template.
- Serialization and deserialization of the AST to/from AST files is simpler: we do not need to track modifications made to AST nodes imported from AST files and serialize separate "update records".

There are unfortunately exceptions to this general approach, such as:

- The first declaration of a redeclarable entity maintains a pointer to the most recent declaration of that entity, which naturally needs to change as more declarations are parsed.
- Name lookup tables in declaration contexts change after the namespace declaration is formed.
- We attempt to maintain only a single declaration for an instantiation of a template, rather than having distinct declarations for an instantiation of the declaration versus the definition, so template instantiation often updates parts of existing declarations.
- Some parts of declarations are required to be instantiated separately (this includes default arguments and exception specifications), and such instantiations update the existing declaration.

These cases tend to be fragile; mutable AST state should be avoided where possible.

As a consequence of this design principle, we typically do not provide setters for AST state. (Some are provided for short-term modifications intended to be used immediately after an AST node is created and before it's "published" as part of the complete AST, or where language semantics require after-the-fact updates.)

Faithfulness

The AST intends to provide a representation of the program that is faithful to the original source. We intend for it to be possible to write refactoring tools using only information stored in, or easily reconstructible from, the Clang AST. This means that the AST representation should either not desugar source-level constructs to simpler forms, or -- where made necessary by language semantics or a clear engineering tradeoff -- should desugar minimally and wrap the result in a construct representing the original source form.

For example, `CXXForRangeStmt` directly represents the syntactic form of a range-based for statement, but also holds a semantic representation of the range declaration and iterator declarations. It does not contain a fully-desugared `ForStmt`, however.

Some AST nodes (for example, `ParenExpr`) represent only syntax, and others (for example, `ImplicitCastExpr`) represent only semantics, but most nodes will represent a combination of syntax

and associated semantics. Inheritance is typically used when representing different (but related) syntaxes for nodes with the same or similar semantics.

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 :ref:`QualType <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 typedefs. 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`/`dyn_cast` 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`/`dyn_cast` 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`, plus some extended qualifiers required by language extensions) separately from the types themselves. `QualType` is conceptually a pair of "`Type*`" and the bits for these 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 (:ref:`Type <Type>` does not even contain qualifiers).

In practice, the two most common type qualifiers (`const` and `restrict`) are stored in the low bits of the pointer to the `Type` object, together with a flag indicating whether extended qualifiers are present (which must be heap-allocated). This means that `QualType` is exactly the same size as a pointer.

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 10 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.

`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 :ref:`type <QualType>` 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 :ref:`type <QualType>` 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`).

`CXXLiteralOperatorName`

> The name is a C++11 user defined literal operator. User defined Literal operators are named according to the suffix they define, e.g., "`_foo`" for "`operator "" _foo`". Use `N.getCXXLiteralIdentifier()` to retrieve the corresponding `IdentifierInfo*` pointing to the identifier.

`CXXUsingDirective`

> The name is a C++ using directive. Using directives are not really NamedDecls, in that they all have the same name, but they are implemented as such in order to store them in DeclContext effectively.

`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 :ref:`Redeclarations and Overloads <Redeclarations>`), 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 :ref:`LexicalAndSemanticContexts` 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".

(Note that because `f` can be redeclared at block scope, or in a friend declaration, etc. it is possible that the declaration of `f` found by name lookup will not be the most recent one.)

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 a `DeclContext::lookup_result` that contains a range of iterators over 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 "f" 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++11 "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++11 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 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 a `DeclContext::lookup_result` that contains a range of iterators over 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 a DeclContext, one can obtain the set of declaration contexts that are semantically connected to this declaration context, in source order, including this context (which will be the only result, for non-namespace contexts) via `DeclContext::collectAllContexts`. 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.

Because the same entity can be defined multiple times in different modules, it is also possible for there to be multiple definitions of (for instance) a `CXXRecordDecl`, all of which describe a definition of the same class. In such a case, only one of those "definitions" is considered by Clang to be the definiition of the class, and the others are treated as non-defining declarations that happen to also contain member declarations. Corresponding members in each definition of such multiply-defined classes are identified either by redeclaration chains (if the members are `Redeclarable`) or by simply a pointer to the canonical declaration (if the declarations are not `Redeclarable` -- in that case, a `Mergeable` base class is used instead).

The ASTImporter

The `ASTImporter` class imports nodes of an `ASTContext` into another `ASTContext`. Please refer to the document :doc:`ASTImporter: Merging Clang ASTs <LibASTImporter>` for an introduction. And please read through the high-level description of the import algorithm, this is essential for understanding further implementation details of the importer.

Abstract Syntax Graph

Despite the name, the Clang AST is not a tree. It is a directed graph with cycles. One example of a cycle is the connection between a `ClassTemplateDecl` and its "templated" `CXXRecordDecl`. The *templated* `CXXRecordDecl` represents all the fields and methods inside the class template, while the `ClassTemplateDecl` holds the information which is related to being a template, i.e. template arguments, etc. We can get the *templated* class (the `CXXRecordDecl`) of a `ClassTemplateDecl` with `ClassTemplateDecl::getTemplatedDecl()`. And we can get back a pointer of the "described" class template from the *templated* class: `CXXRecordDecl::getDescribedTemplate()`. So, this is a cycle between two nodes: between the *templated* and the *described* node. There may be various other kinds of cycles in the AST especially in case of declarations.

Structural Equivalency

Importing one AST node copies that node into the destination `ASTContext`. To copy one node means that we create a new node in the "to" context then we set its properties to be equal to the properties of the source node. Before the copy, we make sure that the source node is not *structurally equivalent* to any existing node in the destination context. If it happens to be equivalent then we skip the copy.

The informal definition of structural equivalency is the following: Two nodes are **structurally equivalent** if they are

- builtin types and refer to the same type, e.g. `int` and `int` are structurally equivalent,
- function types and all their parameters have structurally equivalent types,
- record types and all their fields in order of their definition have the same identifier names and structurally equivalent types,
- variable or function declarations and they have the same identifier name and their types are structurally equivalent.

In C, two types are structurally equivalent if they are *compatible types*. For a formal definition of *compatible types*, please refer to 6.2.7/1 in the C11 standard. However, there is no definition for *compatible types* in the C++ standard. Still, we extend the definition of structural equivalency to templates and their instantiations similarly: besides checking the previously mentioned properties, we have to check for equivalent template parameters/arguments, etc.

The structural equivalent check can be and is used independently from the ASTImporter, e.g. the `clang::Sema` class uses it also.

The equivalence of nodes may depend on the equivalency of other pairs of nodes. Thus, the check is implemented as a parallel graph traversal. We traverse through the nodes of both graphs at the same time. The actual implementation is similar to breadth-first-search. Let's say we start the traverse with the <A,B> pair of nodes. Whenever the traversal reaches a pair <X,Y> then the following statements are true:

- A and X are nodes from the same ASTContext.
- B and Y are nodes from the same ASTContext.
- A and B may or may not be from the same ASTContext.
- if A == X and B == Y (pointer equivalency) then (there is a cycle during the traverse)
    - A and B are structurally equivalent if and only if
        - All dependent nodes on the path from <A,B> to <X,Y> are structurally equivalent.

When we compare two classes or enums and one of them is incomplete or has unloaded external lexical declarations then we cannot descend to compare their contained declarations. So in these cases they are considered equal if they have the same names. This is the way how we compare forward declarations with definitions.

Redeclaration Chains

The early version of the `ASTImporter`'s merge mechanism squashed the declarations, i.e. it aimed to have only one declaration instead of maintaining a whole redeclaration chain. This early approach simply skipped importing a function prototype, but it imported a definition. To demonstrate the problem with this approach let's consider an empty "to" context and the following `virtual` function declarations of `f` in the "from" context:

```
struct B { virtual void f(); };
void B::f() {} // <-- let's import this definition
```

If we imported the definition with the "squashing" approach then we would end-up having one declaration which is indeed a definition, but `isVirtual()` returns `false` for it. The reason is that the definition is indeed not virtual, it is the property of the prototype!

Consequently, we must either set the virtual flag for the definition (but then we create a malformed AST which the parser would never create), or we import the whole redeclaration chain of the function. The most recent version of the `ASTImporter` uses the latter mechanism. We do import all function declarations - regardless if they are definitions or prototypes - in the order as they appear in the "from" context.

If we have an existing definition in the "to" context, then we cannot import another definition, we will use the existing definition. However, we can import prototype(s): we chain the newly imported prototype(s) to the existing definition. Whenever we import a new prototype from a third context, that will be added to the end of the redeclaration chain. This may result in long redeclaration chains in certain cases, e.g. if we import from several translation units which include the same header with the prototype.

To mitigate the problem of long redeclaration chains of free functions, we could compare prototypes to see if they have the same properties and if yes then we could merge these prototypes. The implementation of squashing of prototypes for free functions is future work.

Chaining functions this way ensures that we do copy all information from the source AST. Nonetheless, there is a problem with member functions: While we can have many prototypes for free functions, we must have only one prototype for a member function.

```
void f(); // OK
void f(); // OK

struct X {
  void f(); // OK
  void f(); // ERROR
};
void X::f() {} // OK
```

Thus, prototypes of member functions must be squashed, we cannot just simply attach a new prototype to the existing in-class prototype. Consider the following contexts:

```
// "to" context
struct X {
  void f(); // D0
};
```

```
// "from" context
struct X {
  void f(); // D1
};
void X::f() {} // D2
```

When we import the prototype and the definition of `f` from the "from" context, then the resulting redecl chain will look like this `D0 -> D2'`, where `D2'` is the copy of `D2` in the "to" context.

Generally speaking, when we import declarations (like enums and classes) we do attach the newly imported declaration to the existing redeclaration chain (if there is structural equivalency). We do not import, however, the whole redeclaration chain as we do in case of functions. Up till now, we haven't found any essential property of forward declarations which is similar to the case of the virtual flag in a member function prototype. In the future, this may change, though.

Traversal during the Import

The node specific import mechanisms are implemented in `ASTNodeImporter::VisitNode()` functions, e.g. `VisitFunctionDecl()`. When we import a declaration then first we import everything which is needed to call the constructor of that declaration node. Everything which can be set later is set after the node is created. For example, in case of a `FunctionDecl` we first import the declaration context in which the function is declared, then we create the `FunctionDecl` and only then we import the body of the function. This means there are implicit dependencies between AST nodes. These dependencies determine the order in which we visit nodes in the "from" context. As with the regular graph traversal algorithms like DFS, we keep track which nodes we have already visited in `ASTImporter::ImportedDecls`. Whenever we create a node then we immediately add that to the `ImportedDecls`. We must not start the import of any other declarations before we keep track of the newly created one. This is essential, otherwise, we would not be able to handle circular dependencies. To enforce this, we wrap all constructor calls of all AST nodes in `GetImportedOrCreateDecl()`. This wrapper ensures that all newly created declarations are immediately marked as imported; also, if a declaration is already marked as imported then we just return its counterpart in the "to" context. Consequently, calling a declaration's `::Create()` function directly would lead to errors, please don't do that!

Even with the use of `GetImportedOrCreateDecl()` there is still a probability of having an infinite import recursion if things are imported from each other in wrong way. Imagine that during the import of `A`, the import of `B` is requested before we could create the node for `A` (the constructor needs a reference to `B`). And the same could be true for the import of `B` (`A` is requested to be imported before we could create the node for `B`). In case of the :ref:`templated-described swing <templated>` we take extra attention to break the cyclical dependency: we import and set the described template only after the `CXXRecordDecl` is created. As a best practice, before creating the node in the "to" context, avoid importing of other nodes which are not needed for the constructor of node `A`.

Error Handling

Every import function returns with either an `llvm::Error` or an `llvm::Expected<T>` object. This enforces to check the return value of the import functions. If there was an error during one import then we return with that error. (Exception: when we import the members of a class, we collect the individual errors with each member and we concatenate them in one Error object.) We cache these errors in cases of declarations. During the next import call if there is an existing error we just return with that. So, clients of the library receive an Error object, which they must check.

During import of a specific declaration, it may happen that some AST nodes had already been created before we recognize an error. In this case, we signal back the error to the caller, but the "to" context remains polluted with those nodes which had been created. Ideally, those nodes should not had been created, but that time we did not know about the error, the error happened later. Since the AST is immutable (most of the cases we can't remove existing nodes) we choose to mark these nodes as erroneous.

We cache the errors associated with declarations in the "from" context in `ASTImporter::ImportDeclErrors` and the ones which are associated with the "to" context in `ASTImporterSharedState::ImportErrors`. Note that, there may be several ASTImporter objects which import into the same "to" context but from different "from" contexts; in this case, they have to share the associated errors of the "to" context.

When an error happens, that propagates through the call stack, through all the dependant nodes. However, in case of dependency cycles, this is not enough, because we strive to mark the erroneous nodes so clients can act upon. In those cases, we have to keep track of the errors for those nodes which are intermediate nodes of a cycle.

An **import path** is the list of the AST nodes which we visit during an Import call. If node `A` depends on node `B` then the path contains an `A->B` edge. From the call stack of the import functions, we can read the very same path.

Now imagine the following AST, where the `->` represents dependency in terms of the import (all nodes are declarations).

```
A->B->C->D
   `->E
```

We would like to import A. The import behaves like a DFS, so we will visit the nodes in this order: ABCDE. During the visitation we will have the following import paths:

```
A
AB
ABC
ABCD
ABC
AB
ABE
AB
A
```

If during the visit of E there is an error then we set an error for E, then as the call stack shrinks for B, then for A:

```
A
AB
ABC
ABCD
ABC
```

```
AB
ABE // Error! Set an error to E
AB  // Set an error to B
A   // Set an error to A
```

However, during the import we could import C and D without any error and they are independent of A,B and E. We must not set up an error for C and D. So, at the end of the import we have an entry in `ImportDeclErrors` for A,B,E but not for C,D.

Now, what happens if there is a cycle in the import path? Let's consider this AST:

```
A->B->C->A
   `->E
```

During the visitation, we will have the below import paths and if during the visit of E there is an error then we will set up an error for E,B,A. But what's up with C?

```
A
AB
ABC
ABCA
ABC
AB
ABE // Error! Set an error to E
AB  // Set an error to B
A   // Set an error to A
```

This time we know that both B and C are dependent on A. This means we must set up an error for C too. As the call stack reverses back we get to A and we must set up an error to all nodes which depend on A (this includes C). But C is no longer on the import path, it just had been previously. Such a situation can happen only if during the visitation we had a cycle. If we didn't have any cycle, then the normal way of passing an Error object through the call stack could handle the situation. This is why we must track cycles during the import process for each visited declaration.

Lookup Problems

When we import a declaration from the source context then we check whether we already have a structurally equivalent node with the same name in the "to" context. If the "from" node is a definition and the found one is also a definition, then we do not create a new node, instead, we mark the found node as the imported node. If the found definition and the one we want to import have the same name but they are structurally in-equivalent, then we have an ODR violation in case of C++. If the "from" node is not a definition then we add that to the redeclaration chain of the found node. This behaviour is essential when we merge ASTs from different translation units which include the same header file(s). For example, we want to have only one definition for the class template `std::vector`, even if we included `<vector>` in several translation units.

To find a structurally equivalent node we can use the regular C/C++ lookup functions: `DeclContext::noload_lookup()` and `DeclContext::localUncachedLookup()`. These functions do respect the C/C++ name hiding rules, thus you cannot find certain declarations in a given declaration context. For instance, unnamed declarations (anonymous structs), non-first `friend` declarations and template specializations are hidden. This is a problem, because if we use the regular C/C++ lookup then we create redundant AST nodes during the merge! Also, having two instances of the same node could result in false :ref:`structural in-equivalencies <structural-eq>` of other nodes which depend on the duplicated node. Because of these reasons, we created a lookup class which has the sole purpose to register all declarations, so later they can be looked up by subsequent import requests. This is the `ASTImporterLookupTable` class. This lookup table should be shared amongst the different

`ASTImporter` instances if they happen to import to the very same "to" context. This is why we can use the importer specific lookup only via the `ASTImporterSharedState` class.

ExternalASTSource

The `ExternalASTSource` is an abstract interface associated with the `ASTContext` class. It provides the ability to read the declarations stored within a declaration context either for iteration or for name lookup. A declaration context with an external AST source may load its declarations on-demand. This means that the list of declarations (represented as a linked list, the head is `DeclContext::FirstDecl`) could be empty. However, member functions like `DeclContext::lookup()` may initiate a load.

Usually, external sources are associated with precompiled headers. For example, when we load a class from a PCH then the members are loaded only if we do want to look up something in the class' context.

In case of LLDB, an implementation of the `ExternalASTSource` interface is attached to the AST context which is related to the parsed expression. This implementation of the `ExternalASTSource` interface is realized with the help of the `ASTImporter` class. This way, LLDB can reuse Clang's parsing machinery while synthesizing the underlying AST from the debug data (e.g. from DWARF). From the view of the `ASTImporter` this means both the "to" and the "from" context may have declaration contexts with external lexical storage. If a `DeclContext` in the "to" AST context has external lexical storage then we must take extra attention to work only with the already loaded declarations! Otherwise, we would end up with an uncontrolled import process. For instance, if we used the regular `DeclContext::lookup()` to find the existing declarations in the "to" context then the `lookup()` call itself would initiate a new import while we are in the middle of importing a declaration! (By the time we initiate the lookup we haven't registered yet that we already started to import the node of the "from" context.) This is why we use `DeclContext::noload_lookup()` instead.

Class Template Instantiations

Different translation units may have class template instantiations with the same template arguments, but with a different set of instantiated `MethodDecls` and `FieldDecls`. Consider the following files:

```
// x.h
template <typename T>
struct X {
    int a{0}; // FieldDecl with InitListExpr
    X(char) : a(3) {}     // (1)
    X(int) {}             // (2)
};

// foo.cpp
void foo() {
    // ClassTemplateSpec with ctor (1): FieldDecl without InitlistExpr
    X<char> xc('c');
}

// bar.cpp
void bar() {
    // ClassTemplateSpec with ctor (2): FieldDecl WITH InitlistExpr
    X<char> xc(1);
}
```

In `foo.cpp` we use the constructor with number `(1)`, which explicitly initializes the member a to 3, thus the `InitListExpr {0}` is not used here and the AST node is not instantiated. However, in the case of `bar.cpp` we use the constructor with number `(2)`, which does not explicitly initialize the a member, so the default `InitListExpr` is needed and thus instantiated. When we merge the AST of `foo.cpp` and `bar.cpp` we must create an AST node for the class template instantiation of `X<char>` which has all the required nodes. Therefore, when we find an existing `ClassTemplateSpecializationDecl` then we

merge the fields of the `ClassTemplateSpecializationDecl` in the "from" context in a way that the `InitListExpr` is copied if not existent yet. The same merge mechanism should be done in the cases of instantiated default arguments and exception specifications of functions.

Visibility of Declarations

During import of a global variable with external visibility, the lookup will find variables (with the same name) but with static visibility (linkage). Clearly, we cannot put them into the same redeclaration chain. The same is true the in case of functions. Also, we have to take care of other kinds of declarations like enums, classes, etc. if they are in anonymous namespaces. Therefore, we filter the lookup results and consider only those which have the same visibility as the declaration we currently import.

We consider two declarations in two anonymous namespaces to have the same visibility only if they are imported from the same AST context.

Strategies to Handle Conflicting Names

During the import we lookup existing declarations with the same name. We filter the lookup results based on their :ref:`visibility <visibility>`. If any of the found declarations are not structurally equivalent then we bumped to a name conflict error (ODR violation in C++). In this case, we return with an `Error` and we set up the `Error` object for the declaration. However, some clients of the `ASTImporter` may require a different, perhaps less conservative and more liberal error handling strategy.

E.g. static analysis clients may benefit if the node is created even if there is a name conflict. During the CTU analysis of certain projects, we recognized that there are global declarations which collide with declarations from other translation units, but they are not referenced outside from their translation unit. These declarations should be in an unnamed namespace ideally. If we treat these collisions liberally then CTU analysis can find more results. Note, the feature be able to choose between name conflict handling strategies is still an ongoing work.

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. :ref:`Conditional control-flow <ConditionalControlFlow>` 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 = ...
std::unique_ptr<CFG> FooCFG = CFG::buildCFG(FooBody);
```

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 unfortunate 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 have 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 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 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 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 an integer constant expression) if the operand evaluates to either a numeric value (that is, not a pointer cast to integral type) of integral, enumeration, floating or complex type, or if it evaluates to the address of the first character of a string literal (possibly cast to some other type). 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.
- `__builtin_strlen` and `strlen`: These are constant folded as integer constant expressions if the argument is a string literal.

The Sema Library

This library is called by the :ref:`Parser library <Parser>` during parsing to do semantic analysis of the input. For valid programs, Sema builds an AST for parsed constructs.

The CodeGen Library

CodeGen takes an :ref:`AST <AST>` as input and produces LLVM IR code from it.

How to change Clang

How to add an attribute

Attributes are a form of metadata that can be attached to a program construct, allowing the programmer to pass semantic information along to the compiler for various uses. For example, attributes may be used to alter the code generation for a program construct, or to provide extra semantic information for static analysis. This document explains how to add a custom attribute to Clang. Documentation on existing attributes can be found here.

Attribute Basics

Attributes in Clang are handled in three stages: parsing into a parsed attribute representation, conversion from a parsed attribute into a semantic attribute, and then the semantic handling of the attribute.

Parsing of the attribute is determined by the various syntactic forms attributes can take, such as GNU, C++11, and Microsoft style attributes, as well as other information provided by the table definition of the attribute. Ultimately, the parsed representation of an attribute object is an `ParsedAttr` object. These parsed attributes chain together as a list of parsed attributes attached to a declarator or declaration specifier. The parsing of attributes is handled automatically by Clang, except for attributes spelled as keywords. When implementing a keyword attribute, the parsing of the keyword and creation of the `ParsedAttr` object must be done manually.

Eventually, `Sema::ProcessDeclAttributeList()` is called with a `Decl` and an `ParsedAttr`, at which point the parsed attribute can be transformed into a semantic attribute. The process by which a parsed attribute is converted into a semantic attribute depends on the attribute definition and semantic requirements of the attribute. The end result, however, is that the semantic attribute object is attached to the `Decl` object, and can be obtained by a call to `Decl::getAttr<T>()`.

The structure of the semantic attribute is also governed by the attribute definition given in Attr.td. This definition is used to automatically generate functionality used for the implementation of the attribute, such as a class derived from `clang::Attr`, information for the parser to use, automated semantic checking for some attributes, etc.

include/clang/Basic/Attr.td

The first step to adding a new attribute to Clang is to add its definition to include/clang/Basic/Attr.td. This tablegen definition must derive from the `Attr` (tablegen, not semantic) type, or one of its derivatives. Most attributes will derive from the `InheritableAttr` type, which specifies that the attribute can be inherited by later redeclarations of the `Decl` it is associated with. `InheritableParamAttr` is similar to `InheritableAttr`, except that the attribute is written on a parameter instead of a declaration. If the attribute is intended to apply to a type instead of a declaration, such an attribute should derive from `TypeAttr`, and will generally not be given an AST representation. (Note that this document does not cover the creation of type attributes.) An attribute that inherits from `IgnoredAttr` is parsed, but will generate an ignored attribute diagnostic when used, which may be useful when an attribute is supported by another vendor but not supported by clang.

The definition will specify several key pieces of information, such as the semantic name of the attribute, the spellings the attribute supports, the arguments the attribute expects, and more. Most members of the `Attr` tablegen type do not require definitions in the derived definition as the default suffice. However, every attribute must specify at least a spelling list, a subject list, and a documentation list.

Spellings

All attributes are required to specify a spelling list that denotes the ways in which the attribute can be spelled. For instance, a single semantic attribute may have a keyword spelling, as well as a C++11 spelling and a GNU spelling. An empty spelling list is also permissible and may be useful for attributes which are created implicitly. The following spellings are accepted:

Spelling	Description
`GNU`	Spelled with a GNU-style `__attribute__((attr))` syntax and placement.
`CXX11`	Spelled with a C++-style `[[attr]]` syntax with an optional vendor-specific namespace.
`C2x`	Spelled with a C-style `[[attr]]` syntax with an optional vendor-specific namespace.
`Declspec`	Spelled with a Microsoft-style `__declspec(attr)` syntax.
`Keyword`	The attribute is spelled as a keyword, and required custom parsing.
`GCC`	Specifies two spellings: the first is a GNU-style spelling, and the second is a C++-style spelling with the `gnu` namespace. Attributes should only specify this spelling for attributes supported by GCC.
`Clang`	Specifies two or three spellings: the first is a GNU-style spelling, the second is a C++-style spelling with the `clang` namespace, and the third is an optional C-style spelling with the `clang` namespace. By default, a C-style spelling is provided.
`Pragma`	The attribute is spelled as a `#pragma`, and requires custom processing within the preprocessor. If the attribute is meant to be used by Clang, it should set the namespace to `"clang"`. Note that this spelling is not used for declaration attributes.

Subjects

Attributes appertain to one or more `Decl` subjects. If the attribute attempts to attach to a subject that is not in the subject list, a diagnostic is issued automatically. Whether the diagnostic is a warning or an error depends on how the attribute's `SubjectList` is defined, but the default behavior is to warn. The diagnostics displayed to the user are automatically determined based on the subjects in the list, but a custom diagnostic parameter can also be specified in the `SubjectList`. The diagnostics generated for subject list violations are either `diag::warn_attribute_wrong_decl_type` or `diag::err_attribute_wrong_decl_type`, and the parameter enumeration is found in include/clang/Sema/ParsedAttr.h If a previously unused Decl node is added to the `SubjectList`, the logic used to automatically determine the diagnostic parameter in utils/TableGen/ClangAttrEmitter.cpp may need to be updated.

By default, all subjects in the SubjectList must either be a Decl node defined in `DeclNodes.td`, or a statement node defined in `StmtNodes.td`. However, more complex subjects can be created by creating a `SubsetSubject` object. Each such object has a base subject which it appertains to (which must be a Decl or Stmt node, and not a SubsetSubject node), and some custom code which is called when determining whether an attribute appertains to the subject. For instance, a `NonBitField` SubsetSubject appertains to a `FieldDecl`, and tests whether the given FieldDecl is a bit field. When a SubsetSubject is specified in a SubjectList, a custom diagnostic parameter must also be provided.

Diagnostic checking for attribute subject lists is automated except when `HasCustomParsing` is set to `1`.

Documentation

All attributes must have some form of documentation associated with them. Documentation is table generated on the public web server by a server-side process that runs daily. Generally, the documentation for an attribute is a stand-alone definition in include/clang/Basic/AttrDocs.td that is named after the attribute being documented.

If the attribute is not for public consumption, or is an implicitly-created attribute that has no visible spelling, the documentation list can specify the `Undocumented` object. Otherwise, the attribute should have its documentation added to AttrDocs.td.

Documentation derives from the `Documentation` tablegen type. All derived types must specify a documentation category and the actual documentation itself. Additionally, it can specify a custom heading for the attribute, though a default heading will be chosen when possible.

There are four predefined documentation categories: `DocCatFunction` for attributes that appertain to function-like subjects, `DocCatVariable` for attributes that appertain to variable-like subjects, `DocCatType` for type attributes, and `DocCatStmt` for statement attributes. A custom documentation category should be used for groups of attributes with similar functionality. Custom categories are good for providing overview information for the attributes grouped under it. For instance, the consumed annotation attributes define a custom category, `DocCatConsumed`, that explains what consumed annotations are at a high level.

Documentation content (whether it is for an attribute or a category) is written using reStructuredText (RST) syntax.

After writing the documentation for the attribute, it should be locally tested to ensure that there are no issues generating the documentation on the server. Local testing requires a fresh build of clang-tblgen. To generate the attribute documentation, execute the following command:

```
clang-tblgen -gen-attr-docs -I /path/to/clang/include /path/to/clang/include/clang/Basic
```

When testing locally, *do not* commit changes to `AttributeReference.rst`. This file is generated by the server automatically, and any changes made to this file will be overwritten.

Arguments

Attributes may optionally specify a list of arguments that can be passed to the attribute. Attribute arguments specify both the parsed form and the semantic form of the attribute. For example, if `Args` is `[StringArgument<"Arg1">, IntArgument<"Arg2">]` then `__attribute__((myattribute("Hello", 3)))` will be a valid use; it requires two arguments while parsing, and the Attr subclass' constructor for the semantic attribute will require a string and integer argument.

All arguments have a name and a flag that specifies whether the argument is optional. The associated C++ type of the argument is determined by the argument definition type. If the existing argument types are insufficient, new types can be created, but it requires modifying utils/TableGen/ClangAttrEmitter.cpp to properly support the type.

Other Properties

The `Attr` definition has other members which control the behavior of the attribute. Many of them are special-purpose and beyond the scope of this document, however a few deserve mention.

If the parsed form of the attribute is more complex, or differs from the semantic form, the `HasCustomParsing` bit can be set to `1` for the class, and the parsing code in Parser::ParseGNUAttributeArgs() can be updated for the special case. Note that this only applies to arguments with a GNU spelling -- attributes with a __declspec spelling currently ignore this flag and are handled by `Parser::ParseMicrosoftDeclSpec`.

Note that setting this member to `1` will opt out of common attribute semantic handling, requiring extra implementation efforts to ensure the attribute appertains to the appropriate subject, etc.

If the attribute should not be propagated from a template declaration to an instantiation of the template, set the `Clone` member to 0. By default, all attributes will be cloned to template instantiations.

Attributes that do not require an AST node should set the `ASTNode` field to `0` to avoid polluting the AST. Note that anything inheriting from `TypeAttr` or `IgnoredAttr` automatically do not generate an AST node. All other attributes generate an AST node by default. The AST node is the semantic representation of the attribute.

The `LangOpts` field specifies a list of language options required by the attribute. For instance, all of the CUDA-specific attributes specify `[CUDA]` for the `LangOpts` field, and when the CUDA language option is not enabled, an "attribute ignored" warning diagnostic is emitted. Since language options are not table generated nodes, new language options must be created manually and should specify the spelling used by `LangOptions` class.

Custom accessors can be generated for an attribute based on the spelling list for that attribute. For instance, if an attribute has two different spellings: 'Foo' and 'Bar', accessors can be created: `[Accessor<"isFoo", [GNU<"Foo">]>, Accessor<"isBar", [GNU<"Bar">]>]` These accessors will be generated on the semantic form of the attribute, accepting no arguments and returning a `bool`.

Attributes that do not require custom semantic handling should set the `SemaHandler` field to `0`. Note that anything inheriting from `IgnoredAttr` automatically do not get a semantic handler. All other attributes are assumed to use a semantic handler by default. Attributes without a semantic handler are not given a parsed attribute `Kind` enumerator.

"Simple" attributes, that require no custom semantic processing aside from what is automatically provided, should set the `SimpleHandler` field to `1`.

Target-specific attributes may share a spelling with other attributes in different targets. For instance, the ARM and MSP430 targets both have an attribute spelled `GNU<"interrupt">`, but with different parsing and semantic requirements. To support this feature, an attribute inheriting from `TargetSpecificAttribute` may specify a `ParseKind` field. This field should be the same value between all arguments sharing a spelling, and corresponds to the parsed attribute's `Kind` enumerator. This allows attributes to share a parsed attribute kind, but have distinct semantic attribute classes. For

instance, `ParsedAttr` is the shared parsed attribute kind, but ARMInterruptAttr and MSP430InterruptAttr are the semantic attributes generated.

By default, attribute arguments are parsed in an evaluated context. If the arguments for an attribute should be parsed in an unevaluated context (akin to the way the argument to a `sizeof` expression is parsed), set `ParseArgumentsAsUnevaluated` to `1`.

If additional functionality is desired for the semantic form of the attribute, the `AdditionalMembers` field specifies code to be copied verbatim into the semantic attribute class object, with `public` access.

Boilerplate

All semantic processing of declaration attributes happens in lib/Sema/SemaDeclAttr.cpp, and generally starts in the `ProcessDeclAttribute()` function. If the attribute has the `SimpleHandler` field set to `1` then the function to process the attribute will be automatically generated, and nothing needs to be done here. Otherwise, write a new `handleYourAttr()` function, and add that to the switch statement. Please do not implement handling logic directly in the `case` for the attribute.

Unless otherwise specified by the attribute definition, common semantic checking of the parsed attribute is handled automatically. This includes diagnosing parsed attributes that do not appertain to the given `Decl`, ensuring the correct minimum number of arguments are passed, etc.

If the attribute adds additional warnings, define a `DiagGroup` in include/clang/Basic/DiagnosticGroups.td named after the attribute's `Spelling` with "_"s replaced by "-"s. If there is only a single diagnostic, it is permissible to use `InGroup<DiagGroup<"your-attribute">>` directly in DiagnosticSemaKinds.td

All semantic diagnostics generated for your attribute, including automatically- generated ones (such as subjects and argument counts), should have a corresponding test case.

Semantic handling

Most attributes are implemented to have some effect on the compiler. For instance, to modify the way code is generated, or to add extra semantic checks for an analysis pass, etc. Having added the attribute definition and conversion to the semantic representation for the attribute, what remains is to implement the custom logic requiring use of the attribute.

The `clang::Decl` object can be queried for the presence or absence of an attribute using `hasAttr<T>()`. To obtain a pointer to the semantic representation of the attribute, `getAttr<T>` may be used.

How to add an expression or statement

Expressions and statements are one of the most fundamental constructs within a compiler, because they interact with many different parts of the AST, semantic analysis, and IR generation. Therefore, adding a new expression or statement kind into Clang requires some care. The following list details the various places in Clang where an expression or statement needs to be introduced, along with patterns to follow to ensure that the new expression or statement works well across all of the C languages. We focus on expressions, but statements are similar.

1. Introduce parsing actions into the parser. Recursive-descent parsing is mostly self-explanatory, but there are a few things that are worth keeping in mind:
    - Keep as much source location information as possible! You'll want it later to produce great diagnostics and support Clang's various features that map between source code and the AST.
    - Write tests for all of the "bad" parsing cases, to make sure your recovery is good. If you have matched delimiters (e.g., parentheses, square brackets, etc.), use `Parser::BalancedDelimiterTracker` to give nice diagnostics when things go wrong.
2. Introduce semantic analysis actions into `Sema`. Semantic analysis should always involve two functions: an `ActOnXXX` function that will be called directly from the parser, and a `BuildXXX` function that performs the actual semantic analysis and will (eventually!) build the AST node. It's

fairly common for the `ActOnCXX` function to do very little (often just some minor translation from the parser's representation to `Sema`'s representation of the same thing), but the separation is still important: C++ template instantiation, for example, should always call the `BuildXXX` variant. Several notes on semantic analysis before we get into construction of the AST:

- Your expression probably involves some types and some subexpressions. Make sure to fully check that those types, and the types of those subexpressions, meet your expectations. Add implicit conversions where necessary to make sure that all of the types line up exactly the way you want them. Write extensive tests to check that you're getting good diagnostics for mistakes and that you can use various forms of subexpressions with your expression.
- When type-checking a type or subexpression, make sure to first check whether the type is "dependent" (`Type::isDependentType()`) or whether a subexpression is type-dependent (`Expr::isTypeDependent()`). If any of these return `true`, then you're inside a template and you can't do much type-checking now. That's normal, and your AST node (when you get there) will have to deal with this case. At this point, you can write tests that use your expression within templates, but don't try to instantiate the templates.
- For each subexpression, be sure to call `Sema::CheckPlaceholderExpr()` to deal with "weird" expressions that don't behave well as subexpressions. Then, determine whether you need to perform lvalue-to-rvalue conversions (`Sema::DefaultLvalueConversions`) or the usual unary conversions (`Sema::UsualUnaryConversions`), for places where the subexpression is producing a value you intend to use.
- Your `BuildXXX` function will probably just return `ExprError()` at this point, since you don't have an AST. That's perfectly fine, and shouldn't impact your testing.

3. Introduce an AST node for your new expression. This starts with declaring the node in `include/Basic/StmtNodes.td` and creating a new class for your expression in the appropriate `include/AST/Expr*.h` header. It's best to look at the class for a similar expression to get ideas, and there are some specific things to watch for:
   - If you need to allocate memory, use the `ASTContext` allocator to allocate memory. Never use raw `malloc` or `new`, and never hold any resources in an AST node, because the destructor of an AST node is never called.
   - Make sure that `getSourceRange()` covers the exact source range of your expression. This is needed for diagnostics and for IDE support.
   - Make sure that `children()` visits all of the subexpressions. This is important for a number of features (e.g., IDE support, C++ variadic templates). If you have sub-types, you'll also need to visit those sub-types in `RecursiveASTVisitor`.
   - Add printing support (`StmtPrinter.cpp`) for your expression.
   - Add profiling support (`StmtProfile.cpp`) for your AST node, noting the distinguishing (non-source location) characteristics of an instance of your expression. Omitting this step will lead to hard-to-diagnose failures regarding matching of template declarations.
   - Add serialization support (`ASTReaderStmt.cpp`, `ASTWriterStmt.cpp`) for your AST node.
4. Teach semantic analysis to build your AST node. At this point, you can wire up your `Sema::BuildXXX` function to actually create your AST. A few things to check at this point:
   - If your expression can construct a new C++ class or return a new Objective-C object, be sure to update and then call `Sema::MaybeBindToTemporary` for your just-created AST node to be sure that the object gets properly destructed. An easy way to test this is to return a C++ class with a private destructor: semantic analysis should flag an error here with the attempt to call the destructor.
   - Inspect the generated AST by printing it using `clang -cc1 -ast-print`, to make sure you're capturing all of the important information about how the AST was written.
   - Inspect the generated AST under `clang -cc1 -ast-dump` to verify that all of the types in the generated AST line up the way you want them. Remember that clients of the AST should

never have to "think" to understand what's going on. For example, all implicit conversions should show up explicitly in the AST.

- Write tests that use your expression as a subexpression of other, well-known expressions. Can you call a function using your expression as an argument? Can you use the ternary operator?

5. Teach code generation to create IR to your AST node. This step is the first (and only) that requires knowledge of LLVM IR. There are several things to keep in mind:

   - Code generation is separated into scalar/aggregate/complex and lvalue/rvalue paths, depending on what kind of result your expression produces. On occasion, this requires some careful factoring of code to avoid duplication.
   - `CodeGenFunction` contains functions `ConvertType` and `ConvertTypeForMem` that convert Clang's types (`clang::Type*` or `clang::QualType`) to LLVM types. Use the former for values, and the latter for memory locations: test with the C++ "`bool`" type to check this. If you find that you are having to use LLVM bitcasts to make the subexpressions of your expression have the type that your expression expects, STOP! Go fix semantic analysis and the AST so that you don't need these bitcasts.
   - The `CodeGenFunction` class has a number of helper functions to make certain operations easy, such as generating code to produce an lvalue or an rvalue, or to initialize a memory location with a given value. Prefer to use these functions rather than directly writing loads and stores, because these functions take care of some of the tricky details for you (e.g., for exceptions).
   - If your expression requires some special behavior in the event of an exception, look at the `push*Cleanup` functions in `CodeGenFunction` to introduce a cleanup. You shouldn't have to deal with exception-handling directly.
   - Testing is extremely important in IR generation. Use `clang -cc1 -emit-llvm` and FileCheck to verify that you're generating the right IR.

6. Teach template instantiation how to cope with your AST node, which requires some fairly simple code:

   - Make sure that your expression's constructor properly computes the flags for type dependence (i.e., the type your expression produces can change from one instantiation to the next), value dependence (i.e., the constant value your expression produces can change from one instantiation to the next), instantiation dependence (i.e., a template parameter occurs anywhere in your expression), and whether your expression contains a parameter pack (for variadic templates). Often, computing these flags just means combining the results from the various types and subexpressions.
   - Add `TransformXXX` and `RebuildXXX` functions to the `TreeTransform` class template in `Sema`. `TransformXXX` should (recursively) transform all of the subexpressions and types within your expression, using `getDerived().TransformYYY`. If all of the subexpressions and types transform without error, it will then call the `RebuildXXX` function, which will in turn call `getSema().BuildXXX` to perform semantic analysis and build your expression.
   - To test template instantiation, take those tests you wrote to make sure that you were type checking with type-dependent expressions and dependent types (from step #2) and instantiate those templates with various types, some of which type-check and some that don't, and test the error messages in each case.

7. There are some "extras" that make other features work better. It's worth handling these extras to give your expression complete integration into Clang:

   - Add code completion support for your expression in `SemaCodeComplete.cpp`.
   - If your expression has types in it, or has any "interesting" features other than subexpressions, extend libclang's `CursorVisitor` to provide proper visitation for your expression, enabling various IDE features such as syntax highlighting, cross-referencing, and so on. The `c-index-test` helper program can be used to test these features.