Adding Index-While-Building support to Clang

This document details the Clang enhancements behind the new index-while-building functionality introduced in Xcode 9.

Motivation

Indexing typically happens continuously in the background, and works as outlined in Figure 1. When a source file is first registered or changed, the IDE informs an indexer service, which in turn calls out to libclang to provide updated indexing data for the affected files. The indexer service then uses the returned information to update the index data store, so that later queries to it – that support a range of IDE features like jump-to-definition, find usages, and global rename – provide up-to-date results.

With this setup, indexing typically runs at a lower priority than building, and in the case of Xcode, is paused until the build finishes. Most of the work during indexing, though, is in creating type-checked ASTs to gather the indexing data from, which the build also creates as part of the regular compilation process. By adding support to Clang to output indexing data as part of that process, there's no need to parse and type-check the same source files again in a separate indexing phase.

Figure 1. Background indexing with libClang

Design

Index-while-building support in Clang is enabled through the new `-index-store-path` option. When specified, Clang writes out the indexing data for the compilation unit to the supplied path, in addition to its usual outputs. By passing `-index-store-path` as an additional compiler option during a build, updated indexing data is written out to the store for all re-compiled source files. Background indexing still occurs with this setup, but instead of being based on a call to libclang, is achieved by invoking Clang with both the `-index-store-path` option and `-fsyntax-only`. This change provides process isolation for background indexing, but more importantly matches the behavior of indexing during the build – indexing data is written to the store directly. The means the indexer service can simply subscribe to changes in the store in order to read in the updated index data, regardless of how it was produced.

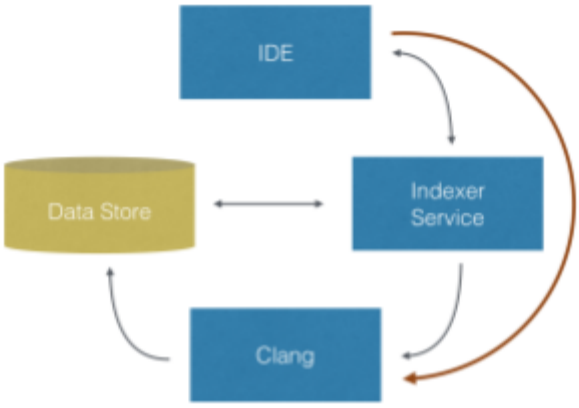

Figure 2. Generating indexing data during a build

Build overhead

While creating type-checked ASTs was always part of compilation, index-while-building support does incur extra overhead in walking these ASTs to collect the indexing data and in writing that data out to the store. From recent measurements with a range of open source projects, however, this overhead is minimal: 2–5% of base build times. When building LLVM+Clang

itself, for example, the time overhead of producing the indexing data averaged 2.5% of the base build time. Similarly, the overhead for WebKit was around 3.1% of the base build time.

Populating the index store

Index-while-building is enabled by a new frontend action, `IndexRecordAction`, that is added when the `-index-store-path` option is present. Like the `IndexAction` frontend action, used when indexing via libclang, `IndexRecordAction` uses an `IndexASTConsumer` to collecting symbol occurrences from the type-checked AST. In addition to this, though, it also uses an `IndexDependencyProvider` to collect the source files and modules involved in the compilation, and any associated file inclusion and module import relations. When the frontend action is done (i.e. `IndexRecordAction`'s `finished()` method is called) the collected indexing data is written out to the index data store.

Writing the indexing data to the store is handled by the `IndexRecordWriter` and `IndexUnitWriter` classes. As their names imply, these classes write out **record** and **unit** files to the directory supplied via `-index-store-path` in the LLVM Bitstream file format. Record files represent and model the symbolic content of a particular source file *as seen during compilation*, while unit files correspond to compilation units, tracking the the set of source file paths and matching record files that comprise them, as well as the unit files of any other compilation units they depend on. To better understand this, consider the example in Figure 3.

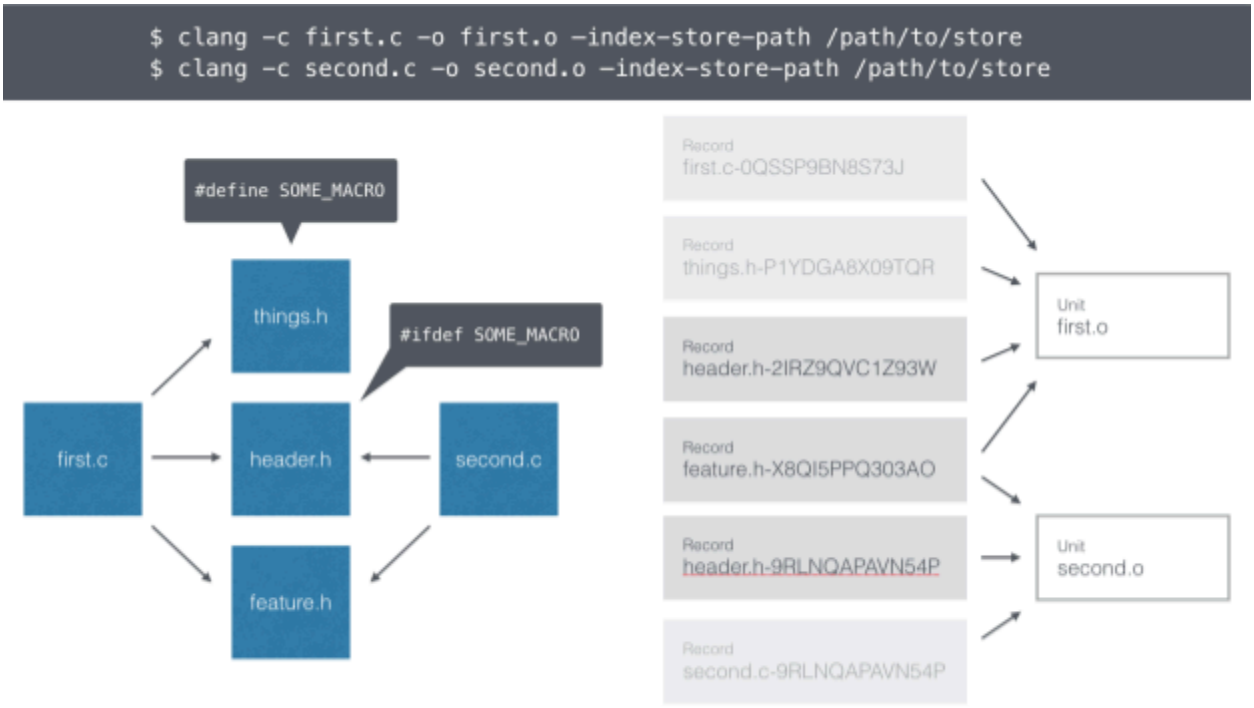

Figure 3. Record and unit files produced from two example invocations

In this example, there are two Clang invocations to compile `first.c` and `second.c`, both with the new `-index-store-path` option specified. The two main files both include `header.h` and `feature.h`, but `first.c` also includes `things.h` – that defines `SOME_MACRO` – prior to its other inclusions. Because of an `#ifdef` block in `header.h` that checks if `SOME_MACRO` is defined, the symbolic content of `header.h` is different in the compilations of `first.c` and `second.c`. Because of this, each compilation produces a distinct record file for `header.h` in the index store that is then referenced in their corresponding unit files. With `feature.h`, on the other hand, the symbolic content is identical across both invocations, so a single record file is produced and referenced in both unit files.

This outcome is achieved by checking if the index store already has a record file with the collected symbolic content before a new record file is written out. Comparing the symbolic content directly, though, would be too expensive, so this is instead

done by including a hash of the symbolic content in each record file as part of its name. The check then becomes just a matter of:

1. computing a hash of the data to be written
2. deriving the corresponding record file name – the source file name + hash, and
3. checking if that file already exists or not.

This approach avoids propagating the duplication introduced by textual inclusion in the preprocessor down to the index store, while still accounting for the effects of differing preprocessor contexts. It also means that through the course of re-indexing compilation units following source changes, stale record files – i.e. those that are no longer referenced in any unit file – may be left in the index data store. There is enough information in the store to detect and remove such records, but this hasn't yet been implemented (records are always accessed via units so these are not visible to clients of the store).

Data model

As well as units and records, covered above, the indexing data collected from the AST and stored within each record file is further organized into entries for the **symbols**, **occurrences** and **relations** libIndex already produces. Symbol entries include all the information about a symbol that doesn't change based on where in a source file it appears: its USR, source language, name, kind and subkind (e.g. `Constructor` and `CXXMoveConstructor`), and a bit set encoding other useful properties, like whether the symbol is locally scoped, or templated. Occurrence entries, on the other hand, include the information unique to each occurrence of a symbol in a source file – its position in the file, its roles in that position (e.g. `Definition`, `Reference`, `Call`, `Read`, `Write`), and its relations to other symbols. Each Relation includes a reference to the related symbol entry along with the set of roles that describe the nature of the occurrence's relationship to that symbol (e.g. `RelationBaseOf`, `RelationCalledBy`, or `RelationChildOf`). Figure 4 shows a subset of the symbols, occurrences and relations produced from an example source file.

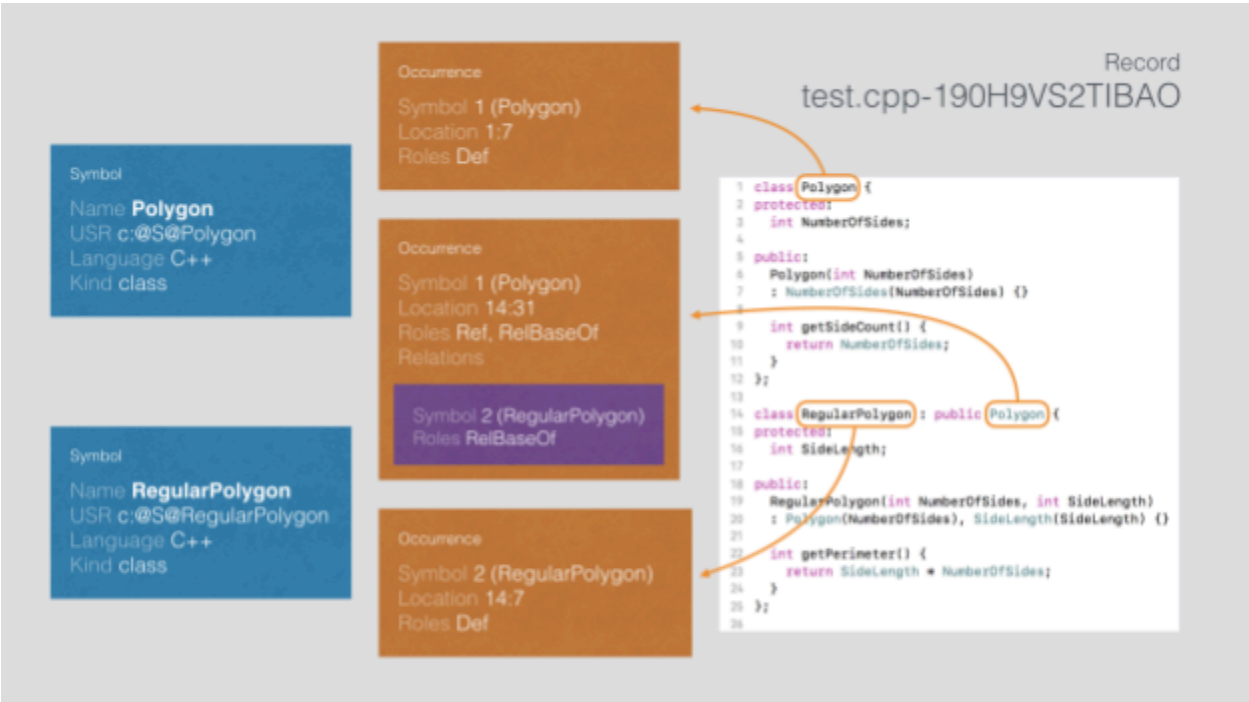

Figure 4. A subset of the symbols, occurrences and relations from a sample source file.

Using the index store

Along with the new frontend action to populate the index data store, there is also a new library that provides APIs for reading and managing it: libIndexStore. This library abstracts away the file structure and format of the store so that clients, like an indexer service, can focus on the units, records, symbols, occurrences and relations it contains.

It is important to note that the index store contains only raw indexing data, and the APIs provided by libIndexStore are simply for iterating over its contents. Most of the queries typically provided by an indexer service would be too slow using libIndexStore APIs alone. As an example, consider the steps required to find all subclasses of `Polygon` (from Figure 4) using the the store:

1. Iterate through the symbol entries in every record in the store to find those with `Polygon`'s USR
2. Iterate through all the occurrences in the records with such a symbol entry to find all occurrences of that symbol with the `RelationBaseOf` Role (indicating it appears as the base of another type).
3. Iterate through the relations of these occurrences to find those with the `RelationBaseOf` role, and note the corresponding symbol entry (the entry of the subclass)

To speed these queries up, most clients will need to maintain at least one in-memory or persistent mapping on top of the index store. Figure 5 is an example of a particularly useful mapping from a symbol's USR to the records it appears in and the set of roles its occurrences have in each of those records. Using this mapping, the first step of the example above – finding the subclasses of `Polygon` – becomes much more performant. Instead of iterating through every symbol in every record to find which ones are relevant, the mapping gives enough information to only access the records where `Polygon` occurs as a base class.

Figure 5. Mapping to find relevant record files.

The same mapping is useful for a range of other queries too. Finding the callers of a particular function or method, for example, works identically to finding subclasses, except with `RelationCalledBy` role, rather than `RelationBaseOf`. For Jump-to-definition, the mapping can be used to quickly determine which records contain an occurrence of the queried USR's symbol with a `Definition` role, and for find references, to simply find all records the queried USR appears in, regardless of role. Additional mappings may be necessary to speed up other queries. A mapping from symbol name to the list of USRs of symbols with that name, for example, is useful for fast symbol lookup, while a mapping from a header file to its dependent units is useful in determining what to re-index following a header file change.

The index data store is constantly being updated via background indexing as users make changes, and also whenever they build, so any mappings clients produce need to be maintained to stay in sync. To assist with this, libIndexStore also allows clients to subscribe to receive notifications whenever unit files are updated, added or removed.

Check out the implementation

The implementation is open source and currently available at https://github.com/apple/swift-clang. Building this repository provides a `clang` with the new `-index-store-path` option and a `c-index-test` with several new options for dumping the index store data both in a pipe separated format and as JSON. To help get

```
struct Vector2D {
    double x, y;

    Vector2D scaled(double xFactor, double yFactor) {
        return {x * xFactor, y * yFactor};
    }
    Vector2D scaled(double factor) {
        return scaled(factor, factor);
    }
};
```

started, try out the example source file, `Vector2D.cpp`, on the right.

Generate the indexing data for it by compiling with the `-index-store-path` option:

```
$ clang -index-store-path /tmp/example-store -c /tmp/Vector2D.cpp -o Vector2D.o -std=c++14
```

To dump information on the units in the store run:

```
$ c-index-test core -print-unit /tmp/example-store
  Vector2D.o-2JTIP5SXAGSWQ
  --------
  provider: clang-900.0.35
  is-system: 0
  is-module: 0
  module-name: <none>
  has-main: 1
  main-path: /tmp/Vector2D.cpp
  work-dir: /tmp
  out-file: Vector2D.o
  target: x86_64-apple-macosx10.13.0
  is-debug: 1
  DEPEND START
  Record | user | /tmp/Vector2D.cpp | Vector2D.cpp-2LLOT9HCYLS10 | 1503945622
  DEPEND END (1)
  INCLUDE START
  INCLUDE END (0)
```

To see the information on the records in the store, run:

```
$ c-index-test core -print-record /tmp/example-store
  Vector2D.cpp
  ------------
  struct/C++ | Vector2D | c:@S@Vector2D | <no-cgname> | Def,Ref,RelCont - RelChild
  field/C++ | x | c:@S@Vector2D@FI@x | <no-cgname> | Def,Ref,Read,RelChild,RelCont -
  field/C++ | y | c:@S@Vector2D@FI@y | <no-cgname> | Def,Ref,Read,RelChild,RelCont -
  instance-method/C++ | scaled | c:@S@Vector2D@F@scaled#d#d# | <no-cgname> | Def,Ref,Call,RelChild,RelCall,RelCont -
RelCont
  instance-method/C++ | scaled | c:@S@Vector2D@F@scaled#d# | <no-cgname> | Def,RelChild - RelCall,RelCont
  ------------
  1:8 | struct/C++ | c:@S@Vector2D | Def | rel: 0
  2:10 | field/C++ | c:@S@Vector2D@FI@x | Def,RelChild | rel: 1
    RelChild | c:@S@Vector2D
  2:13 | field/C++ | c:@S@Vector2D@FI@y | Def,RelChild | rel: 1
    RelChild | c:@S@Vector2D
  4:3 | struct/C++ | c:@S@Vector2D | Ref,RelCont | rel: 1
    RelCont | c:@S@Vector2D@F@scaled#d#d#
  4:12 | instance-method/C++ | c:@S@Vector2D@F@scaled#d#d# | Def,RelChild | rel: 1
    RelChild | c:@S@Vector2D
  5:13 | field/C++ | c:@S@Vector2D@FI@x | Ref,Read,RelCont | rel: 1
    RelCont | c:@S@Vector2D@F@scaled#d#d#
  5:26 | field/C++ | c:@S@Vector2D@FI@y | Ref,Read,RelCont | rel: 1
    RelCont | c:@S@Vector2D@F@scaled#d#d#
  7:3 | struct/C++ | c:@S@Vector2D | Ref,RelCont | rel: 1
    RelCont | c:@S@Vector2D@F@scaled#d#
  7:12 | instance-method/C++ | c:@S@Vector2D@F@scaled#d# | Def,RelChild | rel: 1
    RelChild | c:@S@Vector2D
  8:12 | instance-method/C++ | c:@S@Vector2D@F@scaled#d#d# | Ref,Call,RelCall,RelCont | rel: 1
    RelCall,RelCont | c:@S@Vector2D@F@scaled#d#
```

For both units and records you can also specify a path to the specific file you're interested in:

```
$ c-index-test core -print-unit /tmp/example-store/v5/units/Vector2D.o-2JTIP5SXAGSWQ
$ c-index-test core -print-record /tmp/example-store/v5/records/10/Vector2D.cpp-2LLOT9HCYLS10
```

To aggregate all the unit and record data in the store into a single JSON file, run:

```
$ c-index-test core -aggregate-json /tmp/example-store
```

For more example output see the tests in `test/Index/Core` and `test/Index/Store`.