Linaro connect
Hong Kong 2018

Using the Clang Developer Tools

Peter Smith

Contents

- Introduction to the Clang tools
- Clang-format
- Clang-tidy
- Exploring the Clang AST
- Libclang and its python bindings

Clang tools

- Clang is designed to be much more than just a C/C++ Compiler.
- Difficulty in parsing C++ code restricts the number of developer tools
  - Refactoring.
  - Static analysis.
  - IDE syntax highlighting and indexing
- Clang makes libraries available for the development of new tools.
- Project maintains several useful C/C++ processing tools.
  - Code formatting via clang-format.
  - Static analysis via clang-tidy.
  - Refactoring tools such as clang-rename.

Clang tools

- clang-format
- clang-static-analyzer
- clang-tidy

clang-format

- A code formatting tool that specializes in 80 column layout.
- Supports many styles
  - LLVM, Google, Chromium, WebKit, GNU.
  - Style can be modified with a simple configuration file.
- Can be integrated with editors
  - clang-format-region with emacs.
- Output good enough that it can be sensibly used as part of a coding standard
  - LLD requires clang-format.

Example on iocc contest entry

```
#include   /*recall-the\    /-good--old-\    /IOCCC-days!\    */<unistd.h>
 typedef  unsigned/*int*/  short U;U(main)  [32768],n,r[8];  __attribute__((
# define  R(x)       A(r[  7-(n       >>x&  7)],       (n>>  x>>3       )%8)
#define   C(x)       (U*)  ((/*             |IO|             -dpd
*/char*)  main       +(x)  )/*|             |CC|             ll*/
# define  A(v,       i)(i  ?i<2             ?C(v             ):i\
-4?v+=2,  C(i-       6?v-  2:v+       *C(v  -2))       :C(v  -=2)       :&v)
/*lian*/  constructor))U(  x)(){for(;;*r+=  2,*r+=!n?_exit(  write(2,"Illeg"
"al ins"   "truction ;-"    "(\n",24)),0:     n>>8==001?(      signed char
```

```
               )n*2   :548==n>>    6&&usleep     /**/(10
               )+n%  64==   4?0*  write  (r[7   /**/],C(
               *C(*  r)),   *C(*  r+2)   )+4:  /**/ n>>9
               ==63   &&--r[7-n/   64%8]?n%+  /**/  64*-
               2:0,         n>>6  ==47   ?*R( 0):n>>12==1?
               *R(0  )=*R   (+6)  :n>>  12==+       14?*
               R(0)   -=*R(2*3)    :0)n=*C(*        r);}
```

// Courtesy of https://www.ioccc.org/2015/endoh3/prog.c

Clang-format output

```
#include /*recall-the\    /-good--old-\    /IOCCC-days!\    */ < unistd.h >
typedef unsigned /*int*/ short U;
U(main)[32768], n, r[8];
__attribute__((
#define R(x) A(r[7 - (n >> x & 7)], (n >> x >> 3) % 8)
#define C(x)                                                                   \
  (U *)((/*             |IO|             -dpd
  */ char *)main +                    \
        (x)) /*|             |CC|             ll*/
#define A(v, i)                                                                \
  (i ? i < 2 ? C(v) : i - 4 ? v += 2,                                          \
   C(i - 6 ? v - 2 : v + *C(v - 2)) : C(v -= 2) : &v)
    /*lian*/ constructor)) U(x)() {
  for (;; *r += 2, *r += !n ? _exit(write(2, "Illeg"
                                             "al ins"
                                             "truction ;-"
                                             "(\n",
                                          24)),
          0 : n >> 8 == 001
                  ? (signed char

                     )n *
                        2
                  : 548 == n >> 6 && usleep /**/ (10) + n % 64 == 4
                        ? 0 * write(r[7 /**/], C(*C(*r)), *C(*r + 2)) + 4
                        : /**/ n >> 9 == 63 && --r[7 - n / 64 % 8]
                              ? n % +/**/ 64 * -2
                              : 0,
          n >> 6 == 47 ? *R(0) : n >> 12 == 1
                                     ? *R(0) = *R(+6)
                                     : n >> 12 == +14 ? *R(0) -= *R(2 * 3) : 0)
    n = *C(*r);
}
```

Compilation Database

- Majority of C/C++ code makes use of the preprocessor
    - `#include`, `#define`, `#ifdef` ...
- Many analyses not possible without the command line options.
- Clang tools rely on a compilation database for these options.
    - Simple JSON file recording filename, options, and directory where the compilation is run.
- Libraries such as libclang can make use of compilation database.
- Several ways to obtain a compilation database from your build
    - `cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1`
    - `ninja -t compdb`
    - Tool called `bear` can be used for other build systems, for example `bear make`

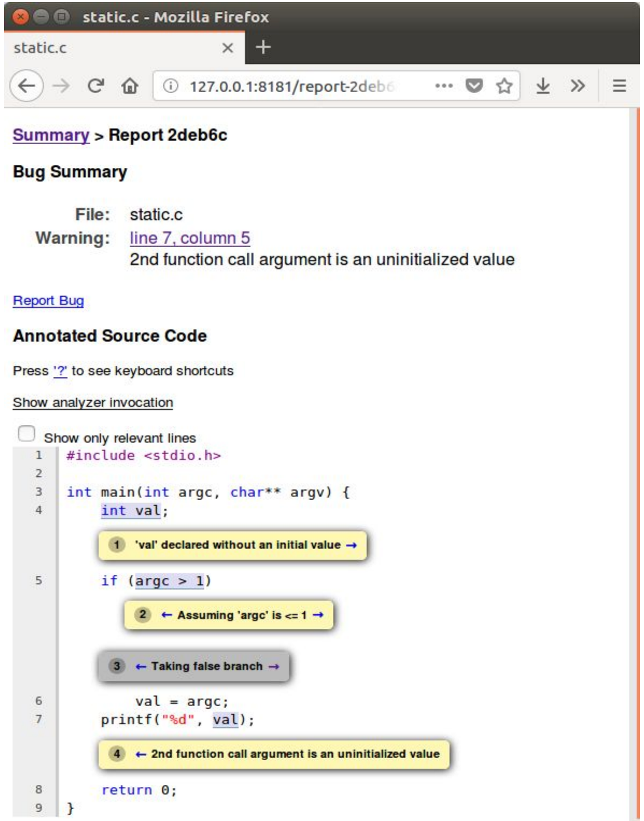
clang static analyzer

- Performs symbolic execution of the program
  - Can find some bugs that would only show up in testing if the relevant path was exercised.
- Limited support for inter-procedural analysis
  - Not done by default.
- Quality of static analysis is highly dependent on codebase
  - False positive rate higher in C++.
  - Results often disjoint from other static analyzers.
- Integrates with build system via `scan-build` tool.
- Produces annotated source code report.

Example clang static analyzer output

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int val;
    if (argc > 1)
        val = argc;
    printf("%d", val);
    return 0;
}
```



clang-tidy

- Lint like tool for checking against a coding style or readability.
- Can offer and apply fixes.
- Can be used as a text-based front-end for the clang static analyzer.
- Helper script run-clang-tidy.py available to run on all files in the compilation database.

clang-tidy example

```
#include <vector>
#include <iostream>

int main(void) {
    std::vector<int> v = { 1, 2, 3, 4, 5 };
    for (std::vector<int>::iterator it = v.begin();
         it != v.end(); ++it)
        std::cout << *it << "\n";
    return 0;
}

// clang-tidy -checks=modernize* modernise.cpp
// --extra-arg=”-std=c++14” --
```

```
modernise.cpp:4:10: warning: redundant void argument list in
function definition [modernize-redundant-void-arg]
int main(void) {
         ^~~~~
modernise.cpp:6:5: warning: use range-based for loop instead
[modernize-loop-convert]
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
^   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    (int & it : v)
modernise.cpp:6:10: warning: use auto when declaring iterators
[modernize-use-auto]
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
         ^
note: this fix will not be applied because it overlaps with another
fix
```

Building your own tools

- libclang
- libtooling

Clang structure

- Clang has a modular structure, with a library based design including:
  - **libbasic** source code abstractions.
  - **libast** classes to represent the AST.
  - **liblex** and **libparse**.
  - **libsema** semantic analysis to build an AST.
  - **librewrite** editing of text buffers.
  - **libanalysis** static analysis.
- These modules are built upon to provide libraries that can build tools
  - **libclang** a stable high-level C interface to clang.
  - **libtooling** a less stable but fully featured C++ interface to clang.
  - **Plugins**, to run during compilation

Clang library options and recommendations

- Libclang
  - High level, stable abstraction makes it the default choice for most tools.
  - C++ IDE support for indexing and code-completion built in.
  - Not all of the underlying AST exposed by design.
  - Python bindings available.
  - Not really suitable for AST modification.
- Plugins
  - Run additional actions on the AST during compilation time.
  - Useful when the build status is dependent on the output of AST action.
  - Uses the same unstable C++ interface to the Clang AST.
- Libtooling
  - Build standalone tools using the full C++ interface to the AST.
  - Includes modification of the AST.
  - Can share code with plugins.

libclang

- Before you start, a word of warning
  - Make sure you have a clear idea of what you want to do before jumping in.
  - Some knowledge of the clang AST structure is necessary.
  - The documentation is sparse, expect to have to look through the libclang API.
  - Examples that you find online can be out of date and simple.
  - Python bindings can have memory/performance problems compared to C.

libclang

- Provides a cursor based interface to the AST
    - A cursor abstracts all the different AST nodes behind a single interface.
    - Source location, Name and symbol resolution, Type, Child nodes
- C API has a visitor based API with callbacks for each child node.
- Python API provides an iterator based interface via get_children.
- Typical program:
    - For each translation unit in compilation database
        - Parse translation unit with libclang
        - Visit each node starting with the root cursor
            - Do some action on each node

Using libclang

- Goal: print a histogram of the number of function parameters in a project
  - Include C functions and C++ member functions.
  - Do not include C++ lambda expressions to keep program simple.
  - Use python bindings for shorter program and development time.
- Shopping list
  - `libclang.so` shared library.
  - Python bindings for cindex in `llvm/tools/clang/bindings/python/clang/cindex.py`
  - Compilation database for our program.
- Environment variables
  - `PYTHONPATH` to find cindex.py.
  - LD_LIBRARY_PATH to find `libclang.so`
    - `$(llvm-config --libdir)`

Iterating through our compilation database

```
from clang.cindex import *
compdb = CompilationDatabase.fromDirectory("path/to/dir/containing/compile_commands.json")
commands = compdb.getAllCompileCommands()
index = Index.create()
for cc in commands:
   arglist = [ar for ar in cc.arguments] # index.parse needs an array not an iterator.
   tu = index.parse(None, arglist) # Pass None for filename as arglist contains it.
   visit_node(tu.cursor) # tu.cursor is root AST node, we provide visit_node.
```

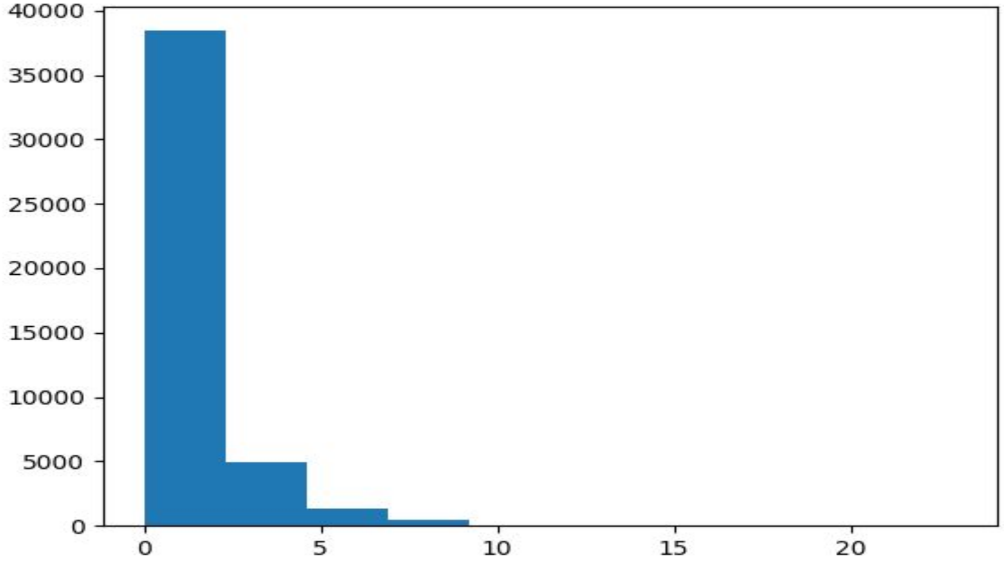
Processing cursor

```
def visit_node(node, parent_fn = None):
    if ((node.kind == CursorKind.FUNCTION_DECL or
         node.kind == CursorKind.CXX_METHOD) and node.isDefinition()):
      # Check if we have processed this function before (Header file)
      # Parameters found will be attributed to this function.
      parent_fn = node
    elif node.kind == CursorKind.PARM_DECL:
      # record parameter for function
    for c in node.getChildren():
        visit_node(c, parent_fn)
```

Complications

- Analysing a whole program rather than translation unit needs care
  - Header files will be seen more than once.
  - I chose to only look at function definitions.
  - Can use `cursor.get_usr()` “Unified Symbol Resolution” to compare across translation units.
- Lambda functions are difficult to handle
  - Requires deeper knowledge of the Clang AST.
- Parsing is slow
  - Index.parse can be passed a flag to skip function bodies, but this means we can’t distinguish between declarations and definitions.
  - Parsing via libclang failed at least once where clang succeeds.
  - Can run out of memory if your program keeps references to information in translation units.

Proportions of first 1000 source files in LLVM



- Advantage of python is that we can use the libraries.
- Histogram courtesy of matplotlib.

Libtooling

- C++ interface to clang.
  - In tree build by adding program to clang/tools/extra simplest way to get started.
  - Out of tree build needs many includes and libraries added.
- Can modify the program with a rewriter or clang-apply-replacements
- Helper functions available to use compilation database.
- Two methods to match the clang AST
  - Recursive AST Visitor.
  - AST Matcher.
- AST matcher is a DSL like language that can concisely describe common matches.
  - `clang-query` tool can be used to interactively work out your matcher.
- C++ AST matcher implementation of python program was of similar size.

References

- Clang static analyzer
    - http://clang-analyzer.llvm.org/
- Clang tidy
    - http://clang.llvm.org/extra/clang-tidy/index.html
- libclang
    - https://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang
    - http://llvm.org/devmtg/2010-11/Gregor-libclang.pdf
    - http://clang.llvm.org/doxygen/group__CINDEX.html
    - https://github.com/llvm-mirror/clang/blob/master/bindings/python/clang/cindex.py
- Compilation Database
    - https://eli.thegreenplace.net/2014/05/21/compilation-databases-for-clang-based-tools

References for libtooling

- Github repo of relatively up to date examples
  - https://github.com/eliben/llvm-clang-samples
- Article about AST matchers
  - https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools
- The same example implemented with AST Visitor and AST Matcher
  - https://jonasdevlieghere.com/understanding-the-clang-ast/

Conclusions

- Clang has many extra tools that you can make use of even if you don't compile your project with clang.
- Building your own tool is practical but non-trivial.
  - Make sure you have a good idea of what you want to build!
- Expect a bit of choice paralysis.
- libclang python bindings are useful for simple analysis programs.
- AST Matchers can be used to write transformations.
  - Expect to need to know much more about clang internals.

Thank You

#HKG18

HKG18 keynotes and videos on: connect.linaro.org

For further information: www.linaro.org