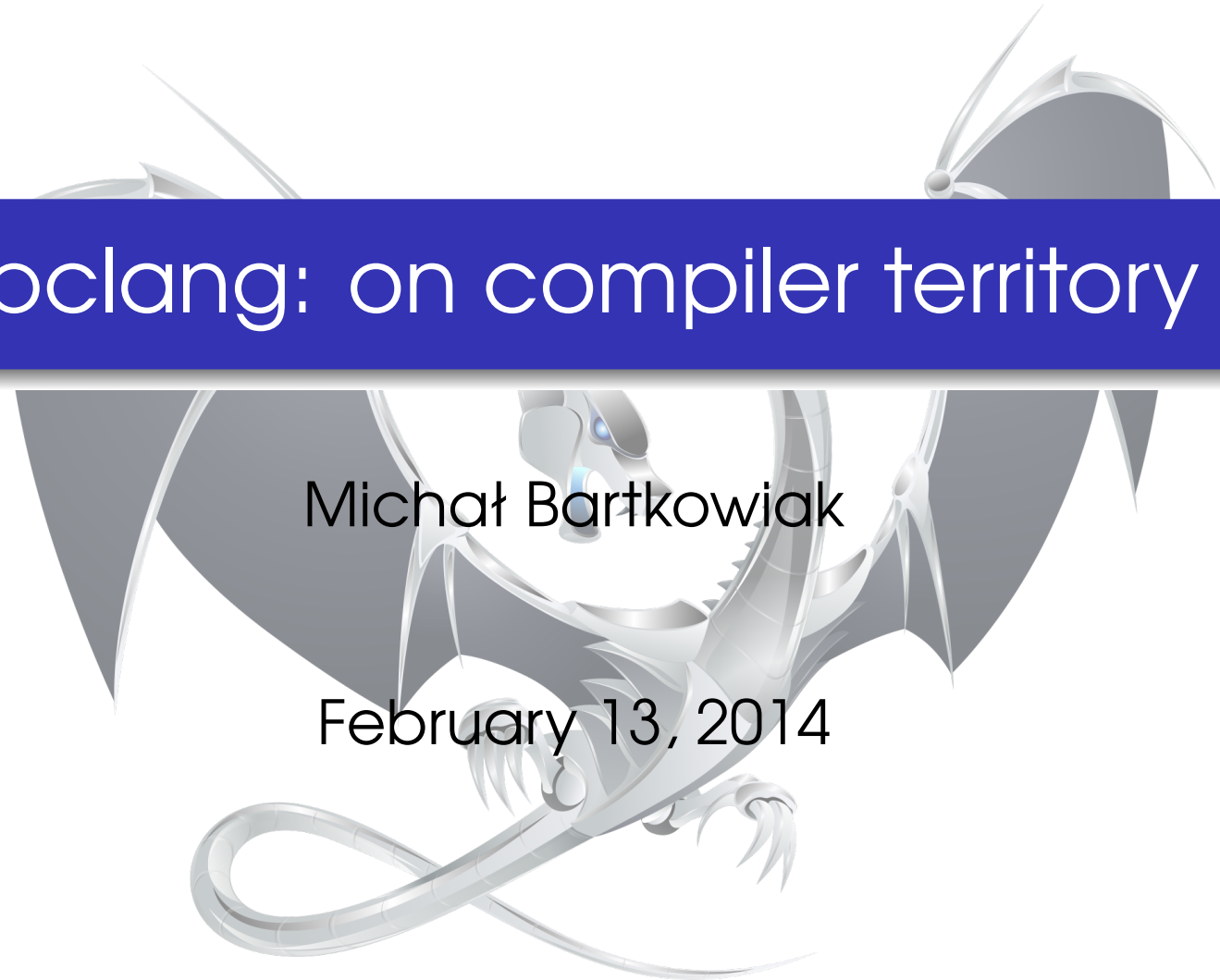


libclang: on compiler territory

Michał Bartkowiak

February 13, 2014

Outline

What is libclang?

libclang is a library for processing source code

- Source code translation to Abstract Syntax Trees (AST)
- Diagnostic reporting
- Traversing AST with cursors
- Mapping between cursors and source code
- Cross-referencing in the AST
- Code completion
- Macro expansion
- Token extraction and manipulation

Why libclang?

- Widely-used and thus verified
- Broadest range of parsing capabilities
- Simple C API
- Detailed information about source code locations at any moment
- libclang is now trendy ;-):
  - XCode
  - YouCompleteMe (ultimate code completion for Vim)

Where shall we begin?

Common header:

```
#include <clang-c/Index.h>
```

Create shared index and translation unit:

```
auto index = clang_createIndex(0, 0);
auto tu = clang_parseTranslationUnit(
              m_index, 0, argv, argc, 0, 0,
              CXTranslationUnit_None);

// ...

clang_disposeTranslationUnit(tu);
clang_disposeIndex(index);
```

Compilation Flags

We would like to have means for generation and storing of compilation flags

Solution:
JSON Compilation Database Format Specification

- Well defined, portable format
- Decouples tools from build systems
- Supported systems:
  - CMake: via `CMAKE_EXPORT_COMPILE_COMMANDS` option
  - Build EAR: via `bear -- make`
- libclang can use these flags

JSON Compilation Database - example

```
[
{
  "directory": "/home/miszak/build/libclang-tools/Apps",
  "command": "/usr/bin/clang++
              -std=c++11 -Wall -Wextra -pedantic -fsanitize=address
              -I/home/miszak/build/libclang-tools/clang+llvm-3.4-x86_64-linux...
              -I/home/miszak/libclang-tools
              -o CMakeFiles/diagnose.dir/Diagnose.cpp.o
              -c /home/miszak/libclang-tools/Apps/Diagnose.cpp",
  "file": "/home/miszak/libclang-tools/Apps/Diagnose.cpp"
},
{
  "directory": "/home/miszak/build/libclang-tools/Apps",
  "command": "/usr/bin/clang++
              -std=c++11 -Wall -Wextra -pedantic -fsanitize=address
              -I/home/miszak/build/libclang-tools/clang+llvm-3.4-x86_64-linux...
              -I/home/miszak/libclang-tools
              -o CMakeFiles/function_name_check.dir/FunctionNameCheck.cpp.o
              -c /home/miszak/libclang-tools/Apps/FunctionNameCheck.cpp",
  "file": "/home/miszak/libclang-tools/Apps/FunctionNameCheck.cpp"
}
]
```

Obtaining Diagnostics

Given the translation unit `tu`:

```
for (auto diagNum : clang_getNumDiagnostics(tu))
{
    auto diag = clang_getDiagnostic(tu, diagNum);
    auto diagStr =
        clang_formatDiagnostic(diag,
            clang_defaultDiagnosticDisplayOptions());

    std::cout << clang_getCString(diagStr) << std::endl;

    clang_disposeString(diagStr);
}
```

Diagnostics - Example

From:

```
1 class X
2 {
3     const int a;
4 }
```

we will get *formatted* output:

- class.cpp:1:7: warning: class ‘X’ does not declare any constructor to initialize its non-modifiable members
- class.cpp:4:2: error: expected ‘;’ after class
- class.cpp:3:15: warning: private field ‘a’ is not used (-Wunused-private-field)

Diagnostics - Details

Each information about diagnostic can be obtained separately:

- clang_getDiagnosticSeverity
- clang_getDiagnosticSpelling
- clang_getDiagnosticLocation and clang_getSpellingLocation
- clang_getDiagnosticNumRanges and clang_getDiagnosticRange

But we want more…

Diagnostics - Fix-its

```
for (auto fixitNum: clang_getDiagnosticNumFixIts(diag))
{
    CXSourceRange range;
    auto fixItStr =
        clang_getDiagnosticFixIt(diag, fixitNum, &range);

    auto rangeStart = clang_getRangeStart(range);
    auto rangeEnd = clang_getRangeEnd(range);

    unsigned lStart, cStart, lEnd, cEnd;
    clang_getSpellingLocation(
            rangeStart, 0, &lStart, &cStart, 0);
    clang_getSpellingLocation(
            rangeEnd, 0, &lEnd, &cEnd, 0);

    std::cout << lStart << ":" << cStart << "␣-␣" <<
              << lEnd   << ":" << lEnd << ":␣" <<
              clang_getCString(fixItStr) << std::endl;

    clang_disposeString(fixItStr);
}
```

Diagnostics - Fix-its - Output

As simple as:

```
4:2 - 4:2:   ;
```

In line **4**,
in column **2**
put **;**

```
1 class X
2 {
3       const int a;
4 }_
```

Walking the AST with CXCursor

CXCursor represents generalised AST node

It can represent e.g.:

- declaration
- definition
- statement
- reference

Provides:

- name
- location and range in source code
- type information
- child(ren)

Learning to Walk

It is simple!

Provide:

```
typedef enum CXChildVisitResult (* CXCursorVisitor)(
        CXCursor cursor,
        CXCursor parent,
        CXClientData client_data)
```

and use:

```
unsigned clang_visitChildren(
        CXCursor parent,
        CXCursorVisitor visitor,
        CXClientData client_data)
```

First Visit: Guest

```
CXChildVisitResult guest(
    CXCursor cursor,
    CXCursor parent,
    CXClientData client_data)
{
    switch (clang_getCursorKind(cursor))
    {
        case CXCursor_FunctionDecl:
            std::cout << "function"; break;
        case CXCursor_CXXMethod:
            std::cout << "cxxmethod"; break;
        default:
            std::cout << "other"; break;
    }
    std::cout << std::endl;
    return CXChildVisit_Recurse;
}
```

First Visit

```
unsigned clang_visitChildren(
    clang_getTranslationUnitCursor(tu),
    guest, 0)
```

Example:

```
1  void f1();
2  namespace A
3  {
4      void f2();
5      class Y
6      {
7          void m1() {};
8      };
9      template<typename T> T ft1();
10 }
```

Output:

```
function
other
function
other
cxxmethod
other
other
other
other
```

When Things Get More Complicated

Example was trivial

What to do when translation unit has (many) includes?

```
auto sourceLoc = clang_getCursorLocation(cursor);
CXFile file;
clang_getFileLocation(sourceLoc, &file, 0, 0, 0);
auto fileName = clang_getFileName(file);

// skip cursors which are not in our file
if (fileName != "/path/to/our/file.cpp")
{
    return CXChildVisit_Continue;
}
```

We can *always* learn CXCursor's detailed location.

What About Parents?

Given the cursor, we can learn about two kinds of parents:

- lexical: `clang_getCursorLexicalParent`
- semantic: `clang_getCursorSemanticParent`

```
1 namespace N
2 {
3     class C
4     {
5         void foo();
6     };
7
8     void C::foo() { /* ... */ }
9 }
```

For declarations: `clang_getCursorDefinition`

Reference Cursors

If the cursor kind is `CXCursor_*Ref` (Type, Variable...), then we can learn about the referenced entity:
`clang_getCursorReferenced`

This way we can find all local references to type, variable... And we are able to e.g.:

- rename them (refactoring)
- colour them (*semantic* highlighting)
- jump between occurences
- jump between reference and declaration

Unified Symbol Resolutions

Each `CXCursor` with external linkage can be uniquely identified by USR:

`clang_getCursorUSR`

This way we can deal with declarations *across* translation units.

Example: `c:@N@A@C@X@F@m1#`
for `A::Y::m1` (method `m1` in class `X` in namespace `A`)

Tokens

Cursors enable us to see the code from AST perspective

Sometimes we just want tokens, e.g. in *syntax highlighting*

```
clang_tokenize(
        CXTranslationUnit TU,
        CXSourceRange Range,
        CXToken **Tokens,
        unsigned *NumTokens)
```

For each token we can obtain:

- kind (`clang_getTokenKind`):
  keyword, identifier, punctuation, literal, comment
- source location and range (`clang_getTokenLocation`)
- spelling (`clang_getTokenSpelling`)
- corresponding cursor (`clang_annotateTokens`)

Code Completion

This is the moment when C-api becomes horryfying...

```
clang_codeCompleteAt(
        CXTranslationUnit tu,
        const char * complete_filename,
        unsigned complete_line,
        unsigned complete_column,
        struct CXUnsavedFile * unsaved_files,
        unsigned num_unsaved_files,
        unsigned options)
```

Code Completion: Example

```
auto compls = clang_codeCompleteAt(
                    tu, "fileName.cpp", 13, 7, 0, 0,
                    clang_defaultCodeCompleteOptions());

for (auto i = 0u; i < compls->NumResults; ++i)
{
  auto &complStr = completionResults->Results[i].CompletionString;

  for (auto j = 0u; j < clang_getNumCompletionChunks(complStr); ++j)
  {
    auto chunkStr = clang_getCompletionChunkText(complStr, j);
    std::cout << toString(chunkStr) << "_";
  }

  std::cout << std::endl;
}

clang_disposeCodeCompleteResults(compls);
```

* A bit of clang_dispose* function calls is omitted...

Code Completion: Example

```
1  class A
2  {
3      void fp() {};
4  public:
5      void f1() {};
6      void f2(int k = 0) {};
7      void f3(int i) {};
8  };
9
10 void foo()
11 {
12     auto a = A();
13     a.
14 }
```

```
void_f2_(__)_
void_f3_(_int i_)_
void_~A_(_)_
A &_operator=_(_const A &_)_
A &_operator=_(_A &&_)_
A_::_
void_fp_(_)_
void_f1_(_)_
```

Code Completion: Algorithm

Client triggers completion procedure at proper place (e.g. at "." when it follows class/struct instance) and presents initial suggestions

The starting place is remembered

Then following procedure is done for each newly typed character:

- trigger code completion
- filter the results basing on contents of token
- present suggestions

Code Completion: Even More

For each completion we can also:

- obtain its priority (`clang_getCompletionPriority`)
- get its context(s) (`clang_codeCompleteGetContexts`)
- for container context get kind of the container (`clang_codeCompleteGetContainerKind`)
- obtain brief comment (`clang_getCompletionBriefComment`)

c-index-test

Use **c-index-test** for experiments

```
usage: c-index-test -code-completion-at=<site> <compiler arguments>
       c-index-test -code-completion-timing=<site> <compiler arguments>
       c-index-test -cursor-at=<site> <compiler arguments>
       [...]

$ c-index-test -code-completion-at=<filename>:13:7 <filename + args>
```

Output:

```
ClassDecl:{TypedText A}{Text ::} (75)
CXXMethod:{ResultType void}{TypedText f1}{LeftParen (}{RightParen )} (34)
CXXMethod:{ResultType void}{TypedText f2}{LeftParen (}{Optional {Placeholder int k}}
    {RightParen )} (34)
[...]
```

libclang in Python

Want to use libclang capabilities in Python?
Not a problem

- clang.cindex module: copy it or set PYTHONPATH
  (warning: Python bindings are part of clang's source)
- clang.cindex needs to be able to find the libclang.so
- ```
  import clang.cindex
  index = clang.cindex.Index.create()
  tu = index.parse(sys.argv[1])
  ```

More on this: `http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang/`
  ```

What's Next?

Create awesome developer tools!

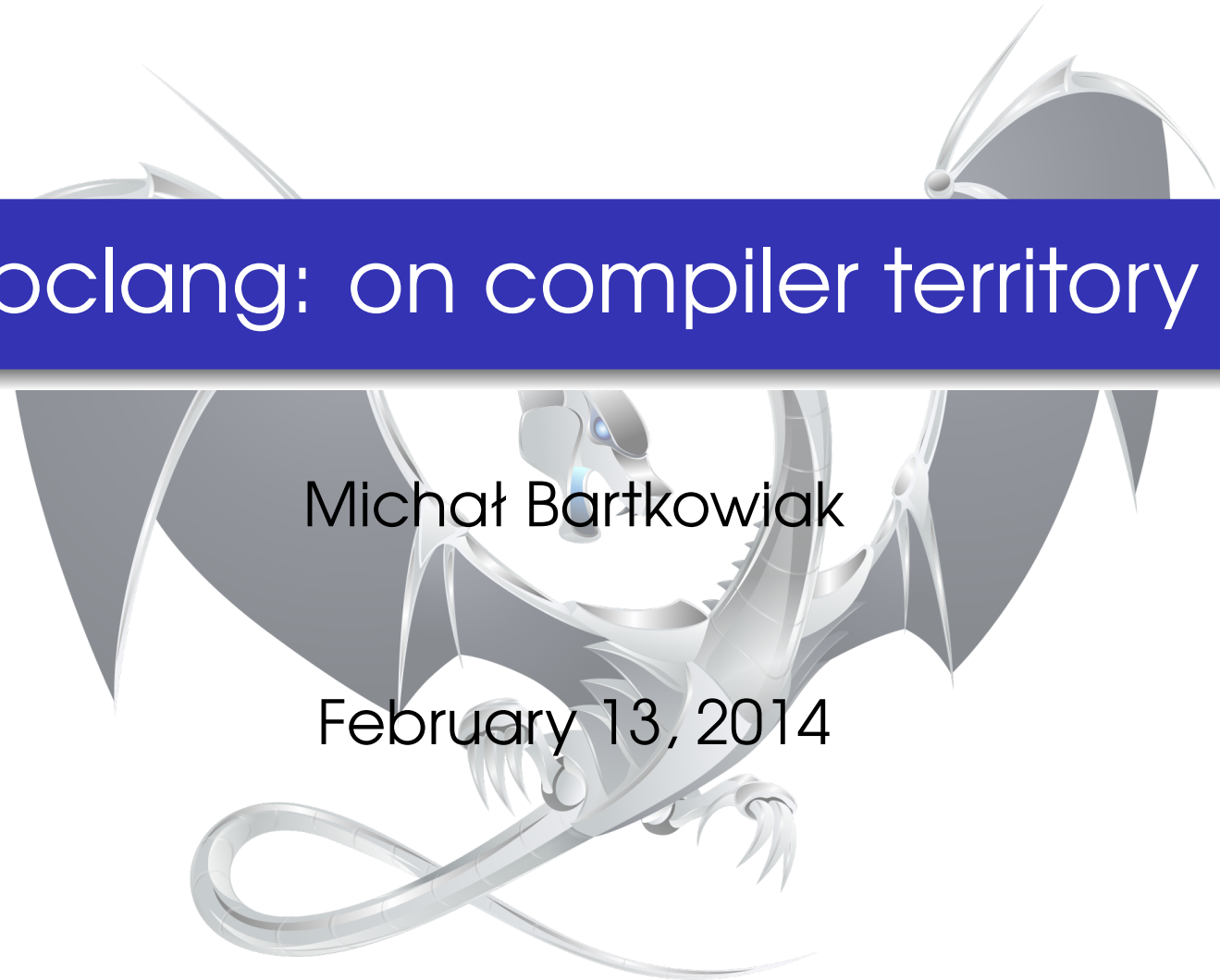
- basis for improvements of IDEs
- code completion and syntax checking available for virtually any text editor (e.g. Vim ;-))
- refactoring tools
- automatic fixing of compile errors
- automatic formatting
- static code analyzers
- migration tools for new features in new language standards

If C api is too clumsy dive directly into clang's C++ interface (and make presentation about it!)

References

- http://clang.llvm.org/doxygen/
- http://clang.llvm.org/docs/Tooling.html
- http://llvm.org/devmtg/2010-11/Gregor-libclang.pdf
- http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang/
- http://llvm.org/devmtg/2011-11/Gregor_ExtendingClang.pdf
- https://github.com/llvm-mirror/clang/tree/master/tools/c-index-test
- https://github.com/miszak/libclang-tools
- https://github.com/Valloric/YouCompleteMe
- https://github.com/axw/cmonster

libclang: on compiler territory



Michał Bartkowiak

February 13, 2014