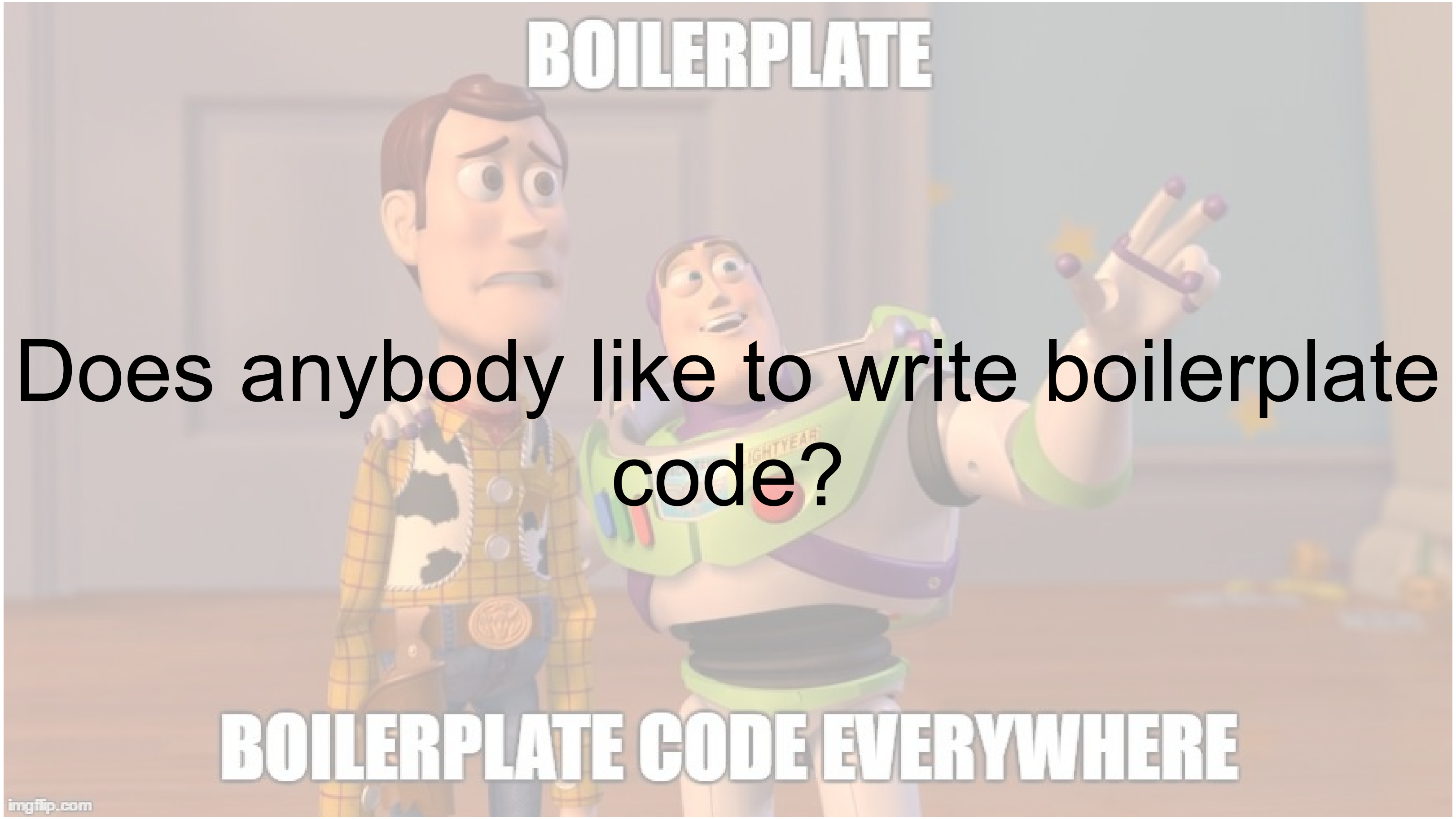
Using Clang for source code generation

Sergei Sadovnikov, “Kaspersky Lab”, ACCU 2017



Does anybody like to write boilerplate code?

Introduction

- Enum to string conversion (and vice versa)
- Serialization/deserialization
- Object RPC and proxy/stubs
- ORM
- GUI controls binding and models
- <your own case>

Introduction

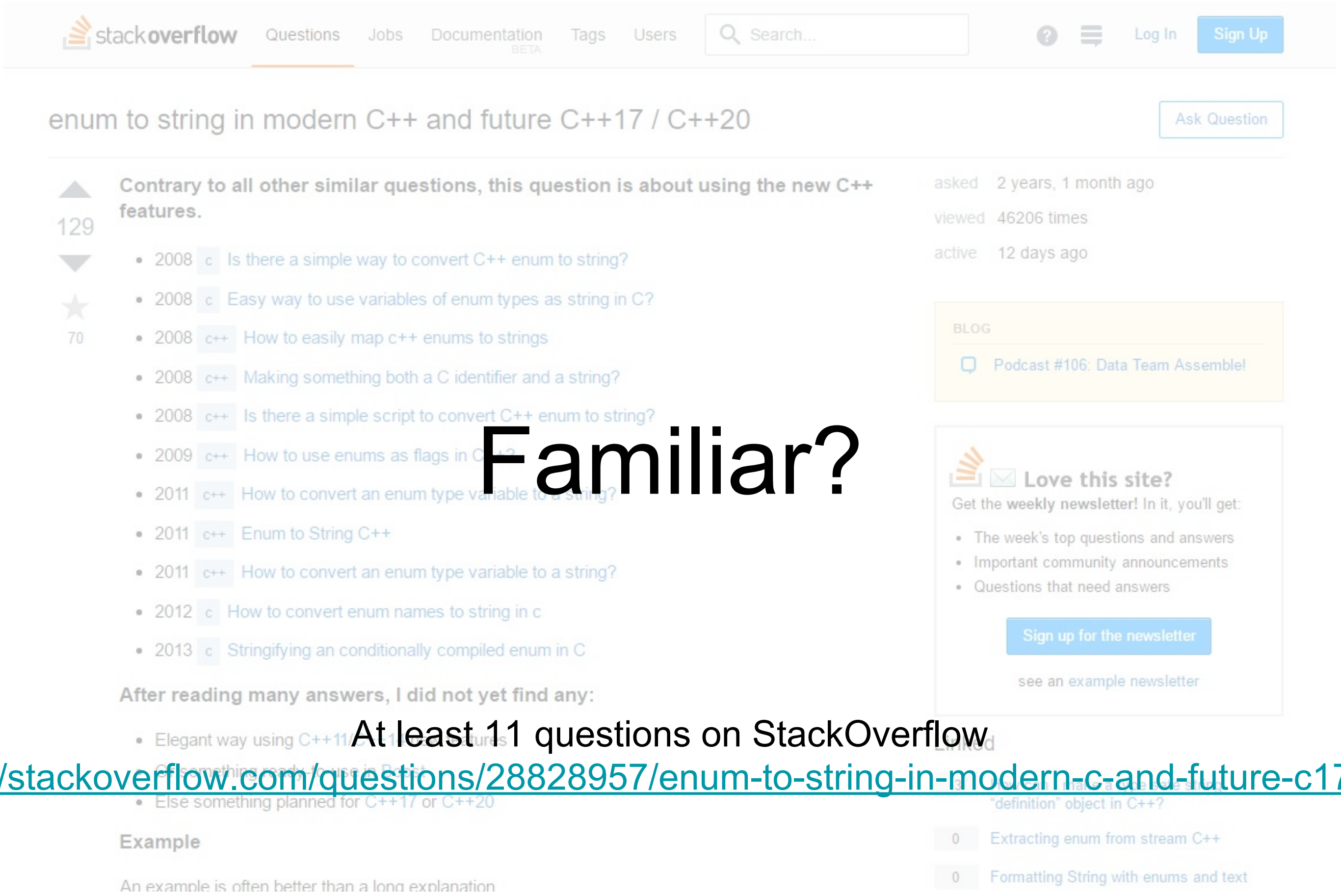
Simple well-known case:

```
enum SomeEnum
{
  Item1 = 10,
  Item2 = 20,
  Item3 = 30
};
```

```
const char* SomeEnumToString(SomeEnum e)
{
  /* ... */
}

SomeEnum StringToSomeEnum(const char* itemName)
{
  /* ... */
}
```

*Yes, I know about std::string_view



enum to string in modern C++ and future C++17 / C++20

Contrary to all other similar questions, this question is about using the new C++ features.

- 2008 c Is there a simple way to convert C++ enum to string?
- 2008 c Easy way to use variables of enum types as string in C?
- 2008 c++ How to easily map c++ enums to strings
- 2008 c++ Making something both a C identifier and a string?
- 2008 c++ Is there a simple script to convert C++ enum to string?
- 2009 c++ How to use enums as flags in C++?
- 2011 c++ How to convert an enum type variable to a string?
- 2011 c++ Enum to String C++
- 2011 c++ How to convert an enum type variable to a string?
- 2012 c How to convert enum names to string in c
- 2013 c Stringifying an conditionally compiled enum in C

After reading many answers, I did not yet find any:

- Elegant way using C++11/C++14 features
- Or something ready-to-use in Boost
- Else something planned for C++17 or C++20

Example

An example is often better than a long explanation

asked 2 years, 1 month ago
viewed 46206 times
active 12 days ago

BLOG
Podcast #106: Data Team Assemble!

Love this site?
Get the **weekly newsletter!** In it, you'll get:
- The week's top questions and answers
- Important community announcements
- Questions that need answers

Sign up for the newsletter
see an example newsletter

Linked
3 How can I make a type-safe string "definition" object in C++?
0 Extracting enum from stream C++
0 Formatting String with enums and text

Familiar?

At least 11 questions on StackOverflow

http://stackoverflow.com/questions/28828957/enum-to-string-in-modern-c-and-future-c17-c20

Introduction

Once upon a time in the bright future with in-language static reflection…

```
template<class Enum>
std::string to_string(Enum e) {
  using namespace std::reflect;
  using e_m = $reflect(Enum);
  static_assert(std::reflect::Enum<e_m>());
  std::string result;
  for_each<get_enumerators_t<e_m>>([&](auto m) {
     using en_m = decltype(m);
     if (get_constant_v<en_m> == e)
        result = get_base_name_v<en_m>;
  });
  return result;
}
```

“Reflection in C++Next”, Anton Bikineev, C++ Russia 2017
https://www.slideshare.net/SergeyPlatonov/reflection-in-cnext?ref=http://2017.cppconf.ru/talks/anton-bikineev

Introduction

Copy-paste. A lot of copy-paste:

```
const char* SomeEnum1ToString(SomeEnum1 e)
{ /* ... */}
SomeEnum1 StringToSomeEnum1(const char* itemName)
{/* ... */}
const char* SomeEnum2ToString(SomeEnum2 e)
{/* ... */}
SomeEnum2 StringToSomeEnum2(const char* itemName)
{/* ... */}
const char* SomeEnum3ToString(SomeEnum3 e)
{/* ... */}
SomeEnum3 StringToSomeEnum3(const char* itemName)
{/* ... */}
const char* SomeEnum4ToString(SomeEnum4 e)
{/* ... */}
SomeEnum4 StringToSomeEnum4(const char* itemName)
{/* ... */}
const char* SomeEnum5ToString(SomeEnum5 e)
{/* ... */}
SomeEnum5 StringToSomeEnum5(const char* itemName)
{/* ... */}
```

Introduction

Template or preprocessor metaprogramming...

```
#define FL_STRINGIZED_ENUM(EnumName, EnumItems) \
        typedef char EnumName ## _StringMap_CharType; \
        FL_DECLARE_ENUM(EnumName, EnumItems) \
        FL_DECLARE_ENUM_STRINGS(EnumName, FL_MAKE_STRING_ENUM_NAME, EnumItems) \

#define FL_WSTRINGIZED_ENUM(EnumName, EnumItems) \
        typedef wchar_t EnumName ## _StringMap_CharType; \
        FL_DECLARE_ENUM(EnumName, EnumItems) \
        FL_DECLARE_ENUM_STRINGS(EnumName, FL_MAKE_WSTRING_ENUM_NAME, EnumItems) \

#define FL_ENUM_ENTRY(EnumEntry) ((EnumEntry, BOOST_PP_EMPTY(), #EnumEntry))
#define FL_ENUM_NAMED_ENTRY(EnumEntry, EnumEntryName) ((EnumEntry, BOOST_PP_EMPTY(), EnumEntryName))
#define FL_ENUM_SPEC_ENTRY(EnumEntry, Id) ((EnumEntry, = Id, #EnumEntry))
```

Introduction

Ugly preprocessor metaprogramming...

```
#define FL_DECLARE_STRING2ENUM_ENTRY_IMPL(STRING_EXPANDER, EntryId, EntryName) result[STRING_EXPANDER(EntryName)] = EntryId;

#define FL_DECLARE_ENUM2STRING_ENTRY(_, STRING_EXPANDER, Entry) FL_DECLARE_ENUM2STRING_ENTRY_IMPL(STRING_EXPANDER, BOOST_PP_TUPLE_ELEM(3, 0, Entry), BOOST_PP_TUPLE_ELEM(3, 2, Entry))
#define FL_DECLARE_STRING2ENUM_ENTRY(_, STRING_EXPANDER, Entry) FL_DECLARE_STRING2ENUM_ENTRY_IMPL(STRING_EXPANDER, BOOST_PP_TUPLE_ELEM(3, 0, Entry), BOOST_PP_TUPLE_ELEM(3, 2, Entry))
#define FL_DECLARE_ENUM_STRINGS(EnumName, MACRO_NAME, S) \
        inline EnumName ## _StringMap_CharType const* EnumName##ToString(EnumName e) \
        { \
                EnumName ## _StringMap_CharType const* result = NULL; \
                switch (e) \
                { \
                BOOST_PP_SEQ_FOR_EACH(FL_DECLARE_ENUM2STRING_ENTRY, MACRO_NAME, S) \
                } \
                \
                return result; \
        }
        inline EnumName StringTo##EnumName(EnumName ## _StringMap_CharType const* str) \
        { \
                static std::map<std::basic_string<EnumName ## _StringMap_CharType>, EnumName> strings_map = []() -> std::map<std::basic_string<EnumName ## _StringMap_CharType>, EnumName> { \
                        std::map<std::basic_string<EnumName ## _StringMap_CharType>, EnumName> result; \
                        BOOST_PP_SEQ_FOR_EACH(FL_DECLARE_STRING2ENUM_ENTRY, MACRO_NAME, S) \
                        return result; \
                }(); \
                \
                return flex_lib::detail::FindEnumEntryForString(strings_map, str); \
        } \
```

Introduction

Very ugly preprocessor metaprogramming...

```
template<typename ImplClass, typename RetTy=void>
class TypeVisitor {
public:
 /// \brief Performs the operation associated with this visitor object.
 RetTy Visit(const Type *T) {
   // Top switch stmt: dispatch to VisitFooType for each FooType.
   switch (T->getTypeClass()) {
#define ABSTRACT_TYPE(CLASS, PARENT)
#define TYPE(CLASS, PARENT) case Type::CLASS: DISPATCH(CLASS##Type);
#include "Clang/AST/TypeNodes.def"
   }
   llvm_unreachable("Unknown type class!");
 }
 // If the implementation chooses not to implement a certain visit method, fall
 // back on superclass.
#define TYPE(CLASS, PARENT) RetTy Visit##CLASS##Type(const CLASS##Type *T) { \
 DISPATCH(PARENT);                                                          \
}
#include "Clang/AST/TypeNodes.def"
 /// \brief Method called if \c ImpClass doesn't provide specific handler
 /// for some type class.
 RetTy VisitType(const Type*) { return RetTy(); }
};
#undef DISPATCH
```

$llvm.src/tools/clang/include/clang/AST/TypeVisitor.h

Introduction

Python/perl scripting

```
import re
import sys
import os
fileName = sys.argv[1]
enumName = os.path.basename(os.path.splitext(fileName)[0])
with open(fileName, 'r') as f:
   content = f.read().replace('\n', '')

searchResult = re.search('enum(.*)\{(.*?)\};', content)
tokens = searchResult.group(2)
tokens = tokens.split(',')
tokens = map(str.strip, tokens)
tokens = map(lambda token: re.search('([a-zA-Z0-9_]*)', token).group(1), tokens)

textOut = ''
textOut += '\n#include "' + enumName + '.hpp"\n\n'
textOut += 'namespace myns\n'
textOut += '{\n'
textOut += '    std::string ToString(ErrorCode errorCode)\n'
textOut += '    {\n'
textOut += '        switch (errorCode)\n'
textOut += '        {\n'
# ...
```

```
#include "ErrorCode.hpp"
namespace myns
{
   std::string ToString(ErrorCode errorCode)
   {
       switch (errorCode)
       {
       case ErrorCode::OK:
           return "OK";
       case ErrorCode::OutOfSpace:
           return "OutOfSpace";
       case ErrorCode::ConnectionFailure:
           return "ConnectionFailure";
       case ErrorCode::InvalidJson:
           return "InvalidJson";
       case ErrorCode::DatabaseFailure:
           return "DatabaseFailure";
       case ErrorCode::HttpError:
           return "HttpError";
       case ErrorCode::FileSystemError:
           return "FileSystemError";
       case ErrorCode::FailedToEncrypt:
           return "FailedToEncrypt";
```

http://stackoverflow.com/questions/201593/is-there-a-simple-way-to-convert-c-enum-to-string

Introduction

Ruby + preprocessor + templates

```
#define BOOST_HANA_DEFINE_STRUCT(...) \

BOOST_HANA_DEFINE_STRUCT_IMPL(BOOST_HANA_PP_NARG(__VA_ARGS__), __VA_ARGS__)

#define BOOST_HANA_DEFINE_STRUCT_IMPL(N, ...) \
  BOOST_HANA_PP_CONCAT(BOOST_HANA_DEFINE_STRUCT_IMPL_, N)(__VA_ARGS__)

<% (0..MAX_NUMBER_OF_MEMBERS).each do |n| %>
#define BOOST_HANA_DEFINE_STRUCT_IMPL_<%= n+1 %>(TYPE <%= (1..n).map { |i| ", m#{i}" }.join %>) \
 <%= (1..n).map { |i| "BOOST_HANA_PP_DROP_BACK m#{i} BOOST_HANA_PP_BACK m#{i};" }.join(' ') %> \
                                                                      \
 struct hana_accessors_impl {                                                   \
```

http://jackieokay.com/2017/04/13/reflection1.html

```
  static constexpr auto apply() {                                        \
    struct member_names {                                                 \
      static constexpr auto get() {                                       \
        return ::boost::hana::make_tuple(                                 \
          <%= (1..n).map { |i|                                            \
            "BOOST_HANA_PP_STRINGIZE(BOOST_HANA_PP_BACK m#{i})" }.join(', ') %>     \
        );                                                                \
      }                                                                   \
    };                                                                    \
    return ::boost::hana::make_tuple(                                     \
      <%= (1..n).map { |i| "::boost::hana::make_pair( \
        ::boost::hana::struct_detail::prepare_member_name<#{i-1}, member_names>(),         \
        ::boost::hana::struct_detail::member_ptr< \
          decltype(&TYPE::BOOST_HANA_PP_BACK m#{i}), \
          &TYPE::BOOST_HANA_PP_BACK m#{i}>{})" }.join(', ') %> \
    );                                                                    \
  }
}
```

Introduction

DSL + translators

```
//===----------------------------------------------------------------------===//
// BlockCommand
//===----------------------------------------------------------------------===//

def Brief : BlockCommand<"brief"> { let IsBriefCommand = 1; }
def Short : BlockCommand<"short"> { let IsBriefCommand = 1; }

// Opposite of \brief, it is the default in our implementation.
def Details : BlockCommand<"details">;

def Returns : BlockCommand<"returns"> { let IsReturnsCommand = 1; }
def Return  : BlockCommand<"return"> { let IsReturnsCommand = 1; }
def Result  : BlockCommand<"result"> { let IsReturnsCommand = 1; }

def Param : BlockCommand<"param"> { let IsParamCommand = 1; }

// Doxygen command for template parameter documentation.
def Tparam : BlockCommand<"tparam"> { let IsTParamCommand = 1; }
```

```
/*===- TableGen'erated file -------------------------------------*- C++ -*-===*\
|* A list of commands useable in documentation comments
|*
|* Automatically generated file, do not edit!
|*
\*===----------------------------------------------------------------------===*/

namespace {
const CommandInfo Commands[] = {
 { "a", "", 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "abstract", "", 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "addtogroup", "", 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0 },
 { "arg", "", 3, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "attention", "", 4, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "author", "", 5, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "authors", "", 6, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "b", "", 7, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
 { "brief", "", 8, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
```

$llvm.src/tools/clang/include/clang/AST/CommentCommands.td

Introduction

What are the reasons to write all these stuff?

- No in-language introspection and reflection (either compile-time or runtime)
  - P0194R3: Static reflection
    - 7th revision
    - date: 2017-02-06
  - P0590R0: A design for static reflection
    - 1st revision
    - date: 2017-02-05
- Too complicated language grammar, so it’s difficult to write C++ parser ‘from scratch’

But things have been changing...

Introduction

We’ve got extraordinary teammate:

- He is able to work on 24/7 schedule
- He is able to write megabytes of boilerplate code without any complains
- He is always in row
- He earns no salary!
- Everybody loves him…
- … but he’s not a human.

We call him tenderly ‘autoprogrammer’

Introduction

- It takes a piece of handwritten C++ code
- It parses this code with the Clang frontend
- It analyses the parsing result and produces another piece of C++ code

No more copypaste and other genius tricks. C++ in, C++ out. Nothing more.

You can achieve a lot of automation with Clang tooling infrastructure

Introduction

Pros:

- High level of code generalization
- High level of code customization (without macros hell)
- Possible errors are quite easy to fix
- Minimal impact of the possible ‘human factor’ during code generation
- Generation results are easy to read and understand
- Generation tool code is easier to understand than tons of macroses and templates
- Production code remains simple

Introduction

Cons:

- High level of entrance - need to understand Clang internals
- No ready-to-use solutions
- Clang libraries need to be built manually before usage
- Generation tool code needs support and update
- And you have to integrate ‘yet another tool’ into your build toolchain

- Introduction
- **Brief tour to Clang C++ API**
- Generation tool implementation
- Generation tool usage

Brief tour to Clang C++ API

Before we start…

- This talk describes a C++ version of the Clang API (current is 4.0). Not pure C (libClang), which is more stable but more difficult to use for deep AST analysis (https://clang.llvm.org/docs/Tooling.html)
- Clang 3.3 or newer is implied

Brief tour to Clang C++ API

- **Compilation infrastructure classes**
- Abstract syntax tree classes
- Support and utility classes

Brief tour to Clang C++ API - Infrastructure

A typical Clang compiler invocation procedure:

1. Prepare vector of the command line options (in text representation)
2. Prepare the diagnostic engine
3. Create the compiler invocation object (from the command line options)
4. Create the frontend action which accepts compilation result
5. Make additional preparations of the source files (if needed)
6. Invoke the compiler and analyze invocation result

The invocation result is represented by **clang::ASTUnit** (if everything is OK), which contains the root of the AST tree.

Brief tour to Clang C++ API - Infrastructure

Main parts of the Clang infrastructure classes:

- Compiler invocation facilities
- Compilation frontend action classes
- Source code management tools
- Diagnostic support engine

Brief tour to Clang C++ API - Infrastructure

The compiler invocation facilities (**clang::CompilerInvocation**) help to invoke compiler:

- Parse command line options
- Combine options into the groups for simpler analysis
- Fill omitted options with the default values

Brief tour to Clang C++ API - Infrastructure

The compilation frontend action (**clang::FrontendAction**) - an abstract class which defines interface from the Clang compiler internals to the compilation frontend (your code). This interface allows:

- Make some preparation before the source file is processed
- Execute the action when all preparations have done
- Perform some post-processing after the compilation has done

Clang provides a lot of default frontend actions and allows you to define your own.

Brief tour to Clang C++ API - Infrastructure

The source code management tools (**clang::SourceManager** and related) provide access to the source code, which is being compiled:

- Hold the main source file and all it’s includes
- Contain mapping of the source locations to the appropriate files
- Contain mapping of the file handles to the real files or internal buffers

Brief tour to Clang C++ API - Infrastructure

The diagnostic engine (**clang::DiagnosticsConsumer**) provides the diagnostic output interface for the compiler internals. This is also an abstract class with number of default implementations. This engine allows:

- Accept, analyze and output diagnostic messages from the compiler
- Accept and analyze special ‘FixIt Hints’, which describe the possible way of fixing the found problems

Brief tour to Clang C++ API

- Compilation infrastructure classes
- **Abstract syntax tree classes**
- Support and utility classes

Brief tour to Clang C++ API - AST

- Provides information about source code structure (declaration, types, statements etc.)
- Provides binding between AST elements and source code
- Describes source code in the form, suitable for analysis

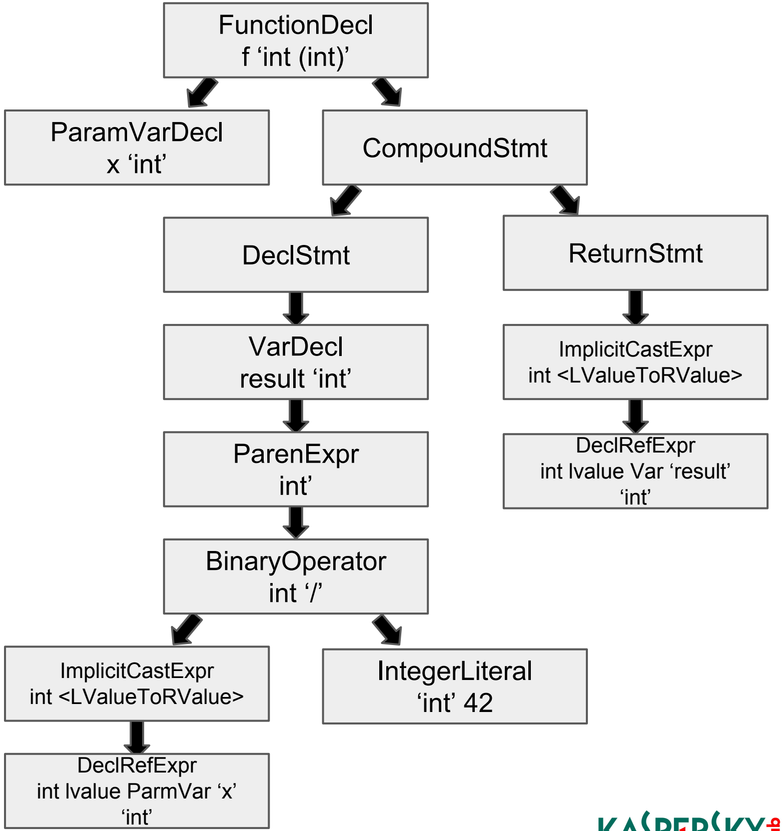
Brief tour to Clang C++ API - AST

```
int f(int x)
{
 int result = (x / 42);
 return result;
}
```

could be transformed into...

Brief tour to Clang C++ API - AST

```
`-FunctionDecl f 'int (int)'
  |-ParmVarDecl x 'int'
  `-CompoundStmt
    |-DeclStmt
    | `-VarDecl result 'int'
    |   `-ParenExpr 'int'
    |     `-BinaryOperator 'int' '/'
    |       |-ImplicitCastExpr 'int' <LValueToRValue>
    |       | `-DeclRefExpr 'int' lvalue ParmVar 'x' 'int'
    |       `-IntegerLiteral 'int' 42
    `-ReturnStmt
      `-ImplicitCastExpr 'int' <LValueToRValue>
        `-DeclRefExpr 'int' lvalue Var 'result' 'int'
```



FunctionDecl
f ‘int (int)’

ParamVarDecl
x ‘int’

CompoundStmt

DeclStmt

ReturnStmt

VarDecl
result ‘int’

ImplicitCastExpr
int <LValueToRValue>

ParenExpr
int’

DeclRefExpr
int lvalue Var ‘result’
‘int’

BinaryOperator
int ‘/’

ImplicitCastExpr
int <LValueToRValue>

IntegerLiteral
‘int’ 42

DeclRefExpr
int lvalue ParmVar ‘x’
‘int’

https://clang.llvm.org/docs/IntroductionToTheClangAST.html

Brief tour to Clang C++ API - AST

Clang AST consists of the four big clusters:

- Declarations
- Types
- Statements and expressions
- Number of support classes, such as QualType, comments reflection and so on

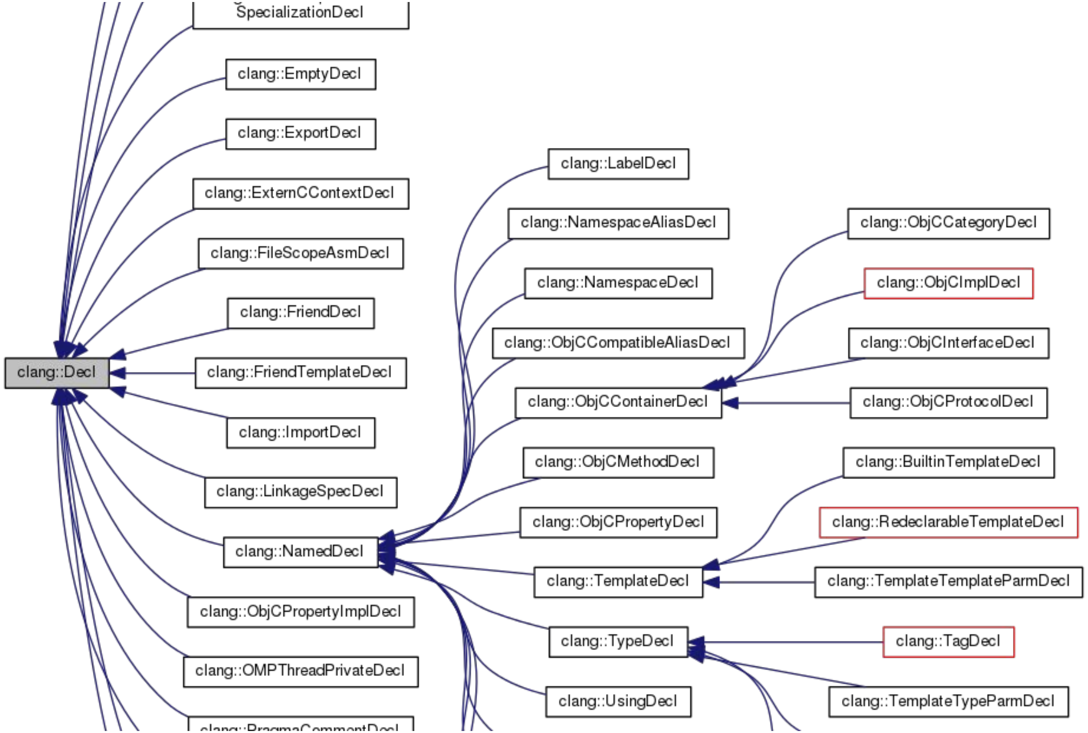
Brief tour to Clang C++ API - AST

‘Declaration’ cluster describes everything can be declared in code:

- Variables
- Functions
- Classes and structures
- Namespaces
- Templates
- etc.

The root of any AST, provided by Clang, is **clang::TranslationUnitDecl**, which is also a part of the declaration cluster.

Brief tour to Clang C++ API - AST



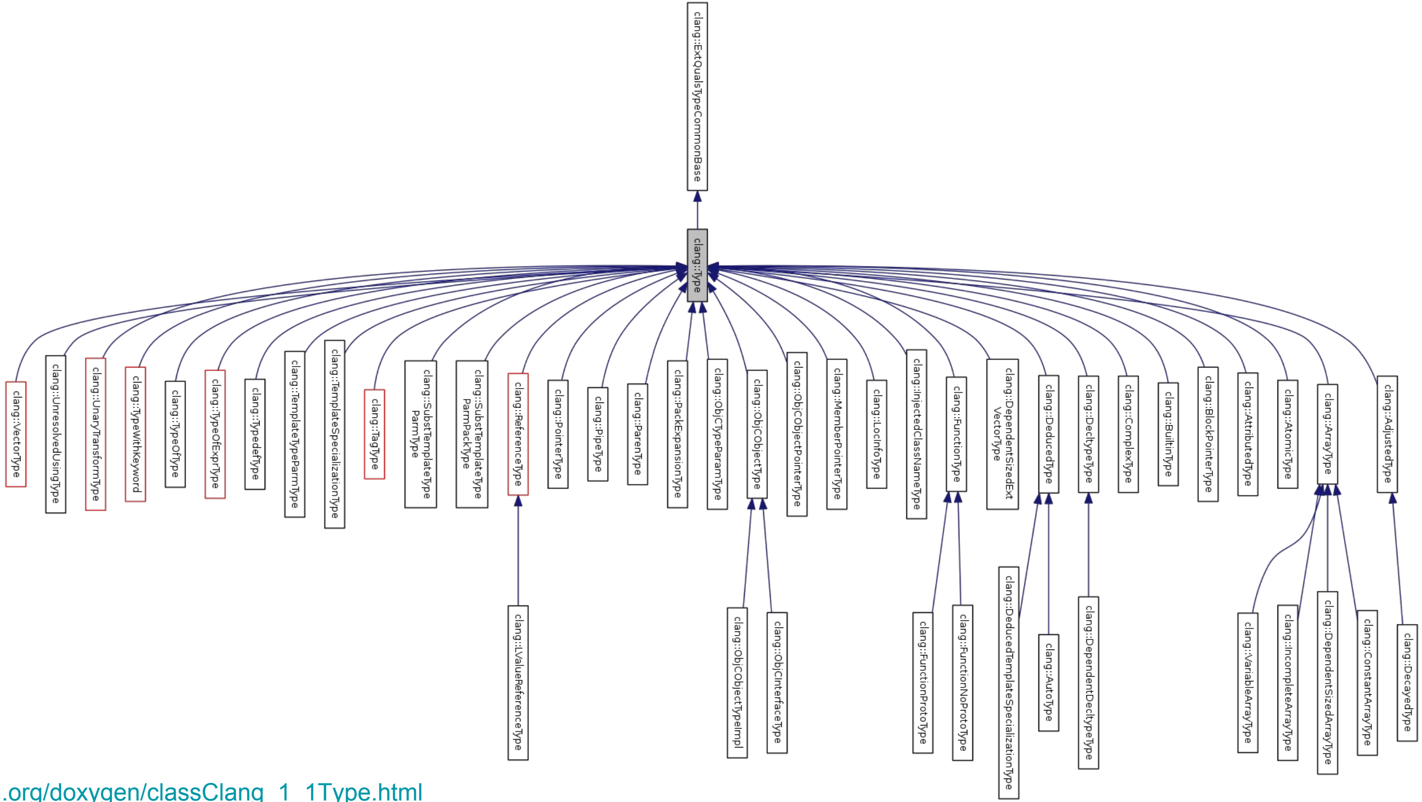
http://clang.llvm.org/doxygen/classClang_1_1Decl.html

Brief tour to Clang C++ API - AST

‘Types’ cluster reflects the C/C++ type system:

- Fundamental types
- Pointers/references/arrays
- Template instantiations
- User-defined types
- And so on

Brief tour to Clang C++ API - AST



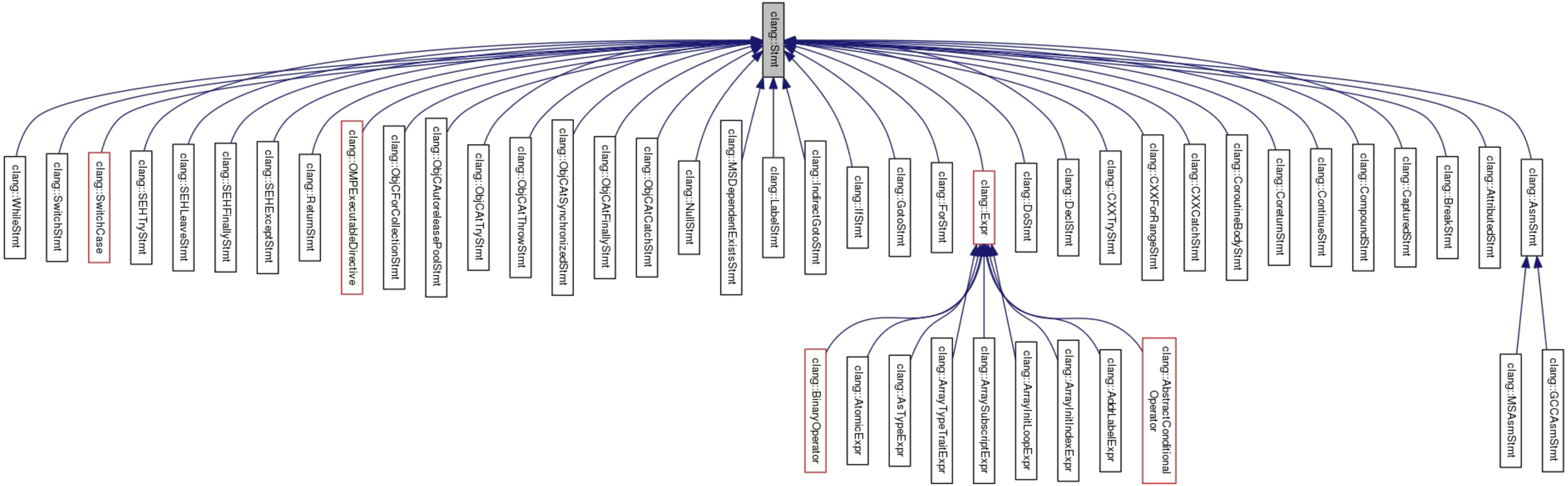
http://clang.llvm.org/doxygen/classClang_1_1Type.html

Brief tour to Clang C++ API - AST

‘Statements and expressions’ cluster describes the imperative part of C/C++:

- Statements
- Expressions
- Operators
- Function calls
- Declaration references
- etc.

Brief tour to Clang C++ API - AST



http://clang.llvm.org/doxygen/classClang_1_1Stmt.html

Brief tour to Clang C++ API - AST

Support classes provide the additional services to main AST such as:

- Qualified types description (QualType)
- AST context holder (ASTContext)
- Binding of AST nodes to source code (SourceLocation)
- Doxygen comments parsing result
- Preprocessor results (#include files tree, macro expansion and evaluation and so on)

Brief tour to Clang C++ API - AST

Methods of AST traversal:

1. SAX-like - via visitors
2. DOM-like - direct AST nodes enumeration
3. xpath-like - via ASTMatcher Clang facility

Brief tour to Clang C++ API

- Compilation infrastructure classes
- Abstract syntax tree classes
- **Support and utility classes**

Brief tour to Clang C++ API - Utilities

The AST visitors (**clang::RecursiveASTVisitor<>** and others) - the most convenient way for AST analysis:

- Use static polymorphism according to CRTP pattern
- Implements three-way visitation:
    - *traverse* methods (e.g. **TraverseNamespaceDecl**) - initiate visitation of the current node and all its subnodes, if any
    - *walkup* methods (e.g. **WalkUpNamespaceDecl**) - dispatch visitation across the AST classes hierarchy and call VisitXXX method
    - *visitation* methods - handles the current AST node according to its type (e.g. **VisitNamespaceDecl**)
- Cover the whole AST classes set

CRTP - ‘Curiously Recurring Template Pattern’, provides polymorphic behaviour without virtuality

Brief tour to Clang C++ API - Utilities

Special visitors for:

- Types (**clang::TypeVisitor**)
- Declarations (**clang::DeclVisitor**)
- Statements (**clang::StmtVisitor**)
- Comments (**clang::comments::CommentsVisitor**)

Brief tour to Clang C++ API - Utilities

The AST matching support (**clang::ast_matchers** namespace) classes:

- Provide xpath-like way of AST analysis
- Provide intuitive way of description of the AST nodes for extraction
- Easy to use
- … but don’t support the whole AST node type and filter combinations

Brief tour to Clang C++ API - Utilities

The tool creation support (**clang::tooling** namespace) classes:

- Provide sophisticated command line options parser (**clang::tooling::CommonOptionsParser**)
- Implement small framework for easy Clang compiler invocation (**clang::tooling::ClangTool**)
- Provide support for compilation database (**clang::tooling::CompilationDatabase**)
- Provide special diagnostic facilities (**clang::tooling::Diagnostic**)

Combining it all together

- Introduction
- Brief tour to Clang C++ API
- **Generation tool implementation**
- Generation tool usage

Generation tool implementation - Enum2String

Let's return to the enum to string conversion sample:

```
enum SomeEnum
{
  Item1 = 10,
  Item2 = 20,
  Item3 = 30
};
```

```
const char* SomeEnumToString(SomeEnum e)
{
  /* ... */
}

SomeEnum StringToSomeEnum(const char* itemName)
{
  /* ... */
}
```

---

*Yes, I know about std::string_view

Generation tool implementation - Enum2String

```
const char* SomeEnumToString(SomeEnum e)
{
   switch (e)
   {
   case Item1:
       return "Item1";
   case Item2:
       return "Item2";
   case Item3:
       return "Item3";
   }

   return "Unknown Item";
}
```

Generation tool implementation - Enum2String

```
SomeEnum StringToSomeEnum(const char* itemName)
{
  static std::pair<const char*, SomeEnum> items[] = {
     {"Item1", Item1},
     {"Item2", Item2},
     {"Item3", Item3},
  };
  auto p = std::lower_bound(begin(items), end(items), itemName,
                             [](auto&& i, auto&& v) {return strcmp(i.first, v) < 0;});
  if (p == end(items) || strcmp(p->first, itemName) != 0)
     throw std::bad_cast();

  return p->second;
}
```

Generation tool implementation - Preparations

The command line parser should be able to parse:

1. Tool-specific options (working mode, output files, diagnostic and so on)
2. Clang-specific options (include paths, command line definitions and others)

What is possible with help of the LLVM command line parser (**llvm::cl** namespace) and the libtooling **CommonOptionsParser** (from **clang::tooling**), which are working together.

Generation tool implementation - Preparations

Command line options description:

```
using namespace llvm;

// Define code generation tool option category
static cl::OptionCategory CodeGenCategory("Code generator options");
// Define option for output file name
cl::opt<std::string> OutputFilename("o", cl::desc("Specify output filename"), cl::value_desc("filename"));

// Define common help message printer
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);
// Define specific help message printer
static cl::extrahelp MoreHelp("\nCode generation tool help text...");
```

Generation tool implementation - Preparations

Command line parsing:

```
int main(int argc, const char **argv)
{
  using namespace clang::tooling;
  CommonOptionsParser optionsParser(argc, argv, CodeGenCategory);

  // ...
}
```

Generation tool implementation - Preparations

A few notes about command line parsing:

- Full set of the Clang options for the file can be omitted if you use the CMake build system and its compilation database
- Tool-specific and Clang-specific options in the command line should be divided with the “--” (double dash) pseudo-option
- By default, input file(s) treated as a positional arguments of the tool-specific part of the options

Generation tool implementation - Preparations

Before the Clang compiler runs it nees to:

1. Prepare the compiler invocation with **clang::tooling::ClangTool**
2. Prepare the AST match finder (**clang::ast_matcher::MatchFinder** class)
3. Think about a strategy of the multiple input files processing

Generation tool implementation - Preparations

AST MatchFinder instantiation:

```
int main(int argc, const char **argv)
{
   using namespace clang::tooling;
   CommonOptionsParser optionsParser(argc, argv, CodeGenCategory);
   ClangTool tool(optionsParser.getCompilations(), optionsParser.getSourcePathList());

   // ...
   MatchFinder finder;

   // ...
}
```

Generation tool implementation - Preparations

Multiple input files processing strategies:

1. Run the Clang compiler for the each input file separately (one input file - one output file). This is default **ClangTool** behaviour
2. Do one compiler invocation for the all input files (many input file - one output file). In this case you can:
    a. Create a temporary file
    b. ‘#include’ into it all input files
    c. Run the compiler on this temporary file
    d. Remove temporary file before the tool finishes

Generation tool implementation - Preparations

AST matcher and finder setup:

- Prepare the matcher with the content of the **clang::ast_matcher** namespace
- Implement and instantiate the ‘MatchCallback’ - a callback class, derived from the **clang::ast_matcher::MatchFinder::MatchCallback**
- Pass it to the the **clang::ast_matcher::MatchFinder** class

Generation tool implementation - Preparations

AST matcher setup:

```
using namespace clang::ast_matchers;

// Matcher declaration
DeclarationMatcher enumMatcher =
    enumDecl(isExpansionInMainFile()).bind("enum");
```

Generation tool implementation - Preparations

MatchCallback implementation sceleton:

```
class EnumHandler : public MatchFinder::MatchCallback
{
public:
   void run(const MatchFinder::MatchResult& result) override
   {
       if (const clang::EnumDecl* decl = result.Nodes.getNodeAs<clang::EnumDecl>("enum"))
       {
           // do something useful with the found enum declaration
       }
   }
};
```

Generation tool implementation - Preparations

Binding MatchCallback and AST matcher to the MatchFinder:

```
int main(int argc, const char **argv)
{
  using namespace clang::tooling;
  CommonOptionsParser optionsParser(argc, argv, CodeGenCategory);
  ClangTool tool(optionsParser.getCompilations(), optionsParser.getSourcePathList());

  EnumHandler handler;
  MatchFinder finder;
  finder.addMatcher(enumMatcher, &handler);

  // ...
}
```

Generation tool implementation - Preparations

Clang compiler invocation:

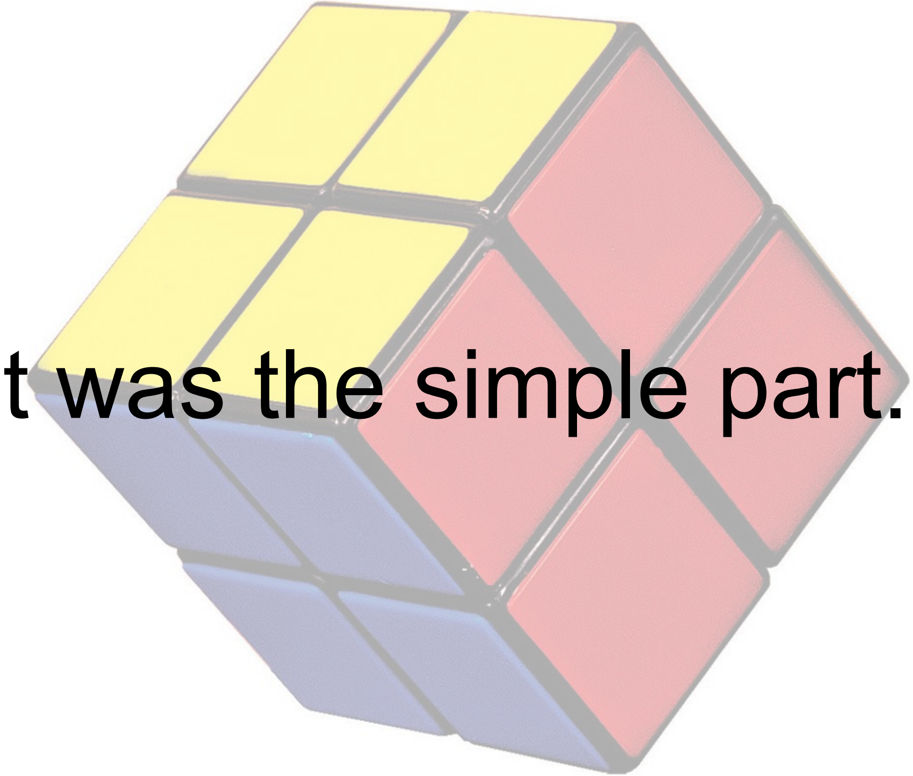
1. **ClangTool::run()** method runs the compiler
2. After the AST is successfully built, the AST matcher (and the AST matcher callback) will be invoked

Generation tool implementation - Preparations

Compiler invocation:

```
int main(int argc, const char **argv)
{
  // ...
  EnumHandler handler;
  MatchFinder finder;
  finder.addMatcher(enumMatcher, &handler);

  auto result = tool.run(newFrontendActionFactory(&finder).get());
  // …
  return result;
}
```

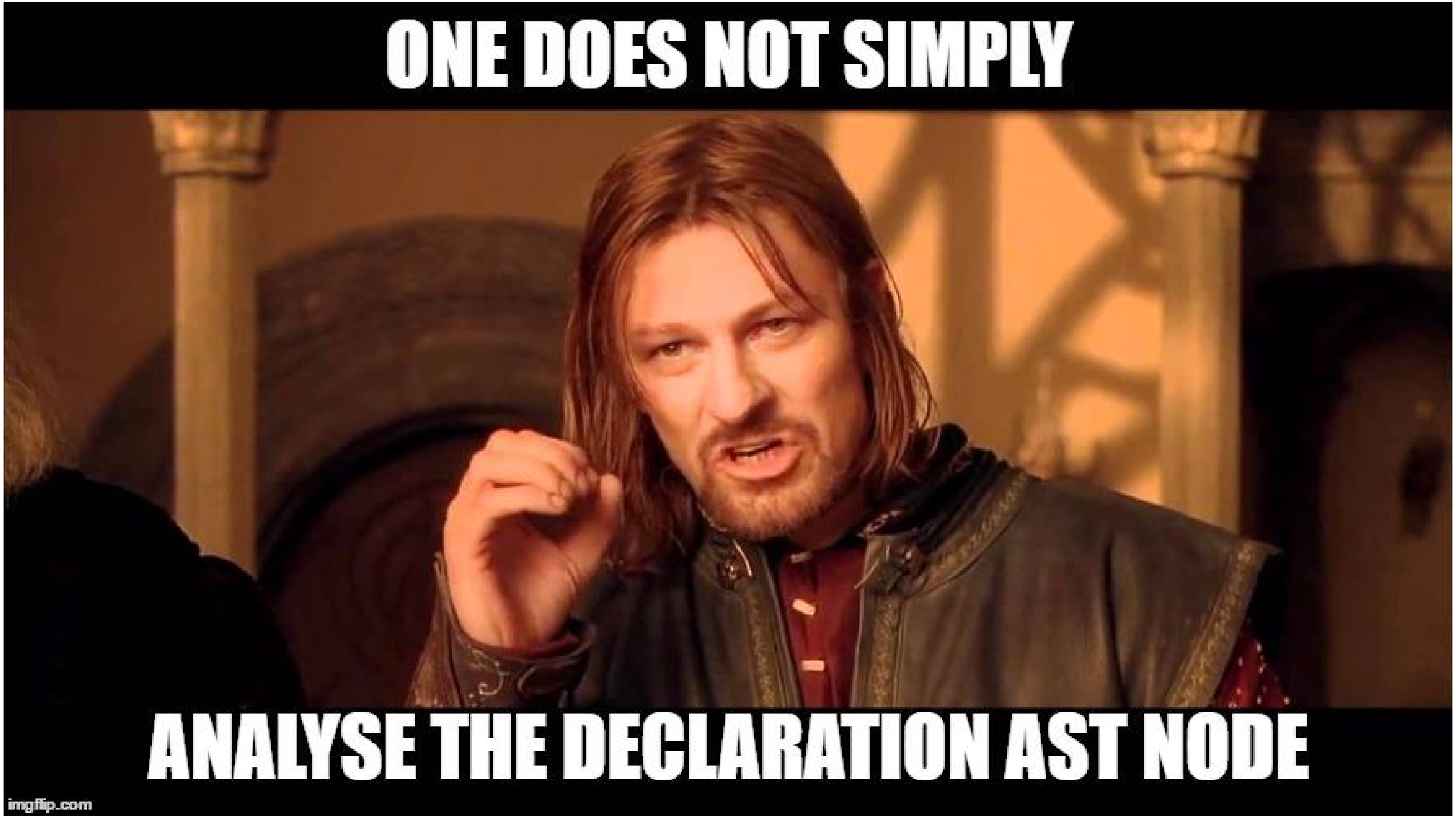


It was the simple part…

Generation tool implementation - AST analysis

In AST matcher callback ‘run’ method:

1. Analyse ‘MatchResult’ input argument (matching result). It contains one of the named AST match
2. Extract the AST node from the matching result
3. Analyse this node according to the desired output result


ONE DOES NOT SIMPLY
ANALYSE THE DECLARATION AST NODE
imgflip.com

Generation tool implementation - AST analysis

Main problems of the declarations AST node analysis:

- Clang AST is extremely detailed and fine-grained
- ‘type’ part of the declaration is hard to analyse due to type erasure
- inline and anonymous namespaces
- etc.

For example, real full-qualified type of std::string can look like std::**__cxx11::**basic_string<>, real full-qualified name within anonymous namespace is “**(anonymous namespace)**::SomeDecl”

Generation tool implementation - AST analysis

Simple cases of analysis:

- Enumerate class/enum/namespace members and do something without deep analysis
- Do simple transformations with the functions/methods/fields with help of the magnificent ‘print’ method

In some cases that’s enough, but...

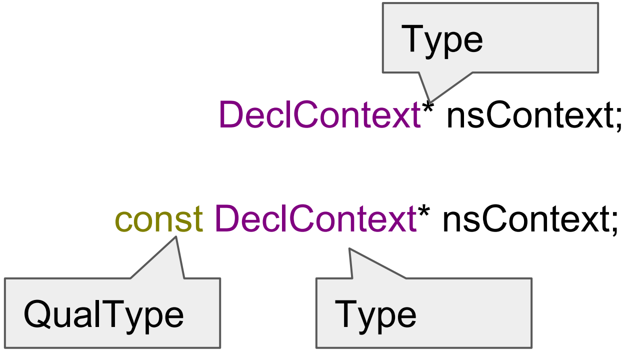
Generation tool implementation - AST analysis

‘A bit’ harder cases:

- Analysis based on the relations between declarations (inheritance, template instantiations, etc.)
- Analysis based on the declaration types (detailed knowledge of data member types, argument types, variable types etc.)

Generation tool implementation - AST analysis

Pain of the Clang ‘types’ AST nodes:



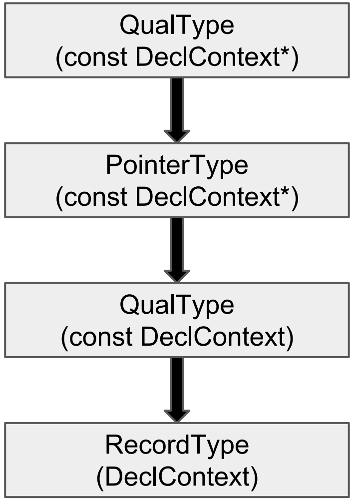
Type

`DeclContext* nsContext;`

`const DeclContext* nsContext;`

QualType

Type



QualType (const DeclContext*)

PointerType (const DeclContext*)

QualType (const DeclContext)

RecordType (DeclContext)

Generation tool implementation - AST analysis

class SomeClass : public SomeTemplate<SomeOtherClass*> {}

Generation tool implementation - AST analysis

class SomeClass : public SomeTemplate<SomeOtherClass*> {}

1. Enumerate base classes of the ‘SomeClass’ -> **CXXBaseSpecifier**
2. Get cv-qualified type from the selected base specifier -> **QualType**
3. Get pointer to the base class type from qualified type -> **Type**
4. Cast it to the **TemplateSpecializationType**
5. Enumerate template instantiation arguments -> **TemplateArgument**
6. Ensure that this is type argument
7. Get qualified type from it -> **QualType**
8. Get pointer to type -> **Type**
9. Cast it to PointerType -> **PointerType**
10. Get pointee type -> **QualType**
11. Get pointer to pointee type -> **Type**
12. Cast it to **RecordType**
13. Get record declaration from it -> **RecordDecl**
14. BINGO!

Generation tool implementation - AST analysis

Special set of helper classes can be created:

- They can describe type ‘in general’ (cv-qualifiers, type of references, number of pointer indirections, short name, full-qualified name etc.)
- They can describe the particular sort of the type (built-in type, record type, template instantiation type, enum and so on)
- Can transform the generic **clang::QualType** instance to the particular type descriptor via type visitor

Generation tool implementation - AST analysis

Declarations/statements attributes problem:

- The C++ attributes system is closed for extension - you can’t add your own project-specific attributes (until C++17)
- Standard C++ attributes are not enough - e.g. you can’t mark some structure as ‘serializable’

But...

Generation tool implementation - AST analysis

… you can write something like this:

```
/*!
 * \serializable
 */
struct SomeSerializableStruct {};
```

And get this attribute during AST analysis.

Generation tool implementation - AST analysis

Doxygen comments allow:

- Add any additional attributes to the declarations
- Get them during AST analysis and **clang::comments::CommentVisitor**
- Change behaviour of the generation tool according to attributes

Generation tool implementation - AST analysis

Summing up:

1. Extract declarations/statements for analysis via the appropriate matchers and MatchFinder
2. Filter the MatchFinder results which belongs to the input files
3. Perform the deeper analysis via direct access to the Clang AST nodes
4. Extract attributes from the Doxygen comments via comment visitors
5. Transform the found types to the simpler representation via type visitors
6. Save the intermediate (simpler) form of the found AST nodes for further generation purpose

Generation tool implementation - AST analysis

‘enum’ declaration reflection:

```
struct EnumDescriptor
{
  // Enumeration name
  std::string enumName;
  // Is enum item needs scope specifier or not
  bool isScoped = false;
  // Collection of enum items
  std::vector<std::string> enumItems;
};
```

Generation tool implementation - AST analysis

Processing of the found AST nodes:

```
class EnumHandler : public MatchFinder::MatchCallback
{
public:
   // ...

   auto& GetFoundEnums() const {return m_foundEnums;}

private:
   std::vector<EnumDescriptor> m_foundEnums;
   void ProcessEnum(const clang::EnumDecl* decl) // ...
};
```

Generation tool implementation - AST analysis

‘enum’ reflection:

```
void ProcessEnum(const clang::EnumDecl* decl)
{
    EnumDescriptor descriptor;
    descriptor.enumName = decl->getName();
    descriptor.isScoped = decl->isScoped();

    for (auto itemDecl : decl->enumerators())
        descriptor.enumItems.push_back(itemDecl->getName());

    std::sort(descriptor.enumItems.begin(), descriptor.enumItems.end());
    m_foundEnums.push_back(std::move(descriptor));
}
```

Generation tool implementation - AST analysis

Processing of the found AST nodes:

```
void run(const MatchFinder::MatchResult& result) override
{
   if (const clang::EnumDecl* decl = Result.Nodes.getNodeAs<clang::EnumDecl>("enum"))
   {
      ProcessEnum(decl);
   }
}
```

Generation tool implementation - Generation

Three quite simple steps:

1. Validate the intermediate form produced by the AST analyser
2. Output tool-specific diagnostic (if needed)
3. Write generation artefacts

Generation tool implementation - Generation

Output file creation:

```
int main(int argc, const char **argv)
{
  // ...
  // Run the tool
  auto result = tool.run(newFrontendActionFactory(&finder).get());
  if (result != 0)
    return result;
  // Open output file
  std::ofstream outFile(OutputFilename.c_str());
  if (!outFile.good())
  {
    std::cerr << "Can't open output file for writing: " << OutputFilename << std::endl;
    return -1;
  }
  // ...
```

Generation tool implementation - Generation

Generation results writing:

```
int main(int argc, const char **argv)
{
    // ...
    // Write conversion functions to output file
    for (auto& descr : handler.GetFoundEnums())
    {
        WriteToStringConversion(outFile, descr);
        WriteFromStringConversion(outFile, descr);
    }

    return 0;
}
```

Generation tool implementation - Generation

Conversion functions writer:

```
void WriteEnumToStringConversion(std::ostream& os, const EnumDescriptor& enumDescr)
{
   auto& enumName = enumDescr.enumName;
   os << "inline const char* " << enumName << "ToString(" << enumName << " e)\n{\n";
   os << "    switch (e)\n";
   os << "    {\n";
   auto scopePrefix = enumDescr.isScoped ? enumName + "::" : std::string();
   for (auto& i : enumDescr.enumItems)
   {
       os << "    case " << scopePrefix << i << ":\n";
       os << "        return \"" << i << "\";\n";
   }
   os << "    }\n";
   os << "    return \"Unknown Item\";\n";
   os << "}\n\n";
}
```

Generation tool implementation - Generation

Conversion functions writer:

```
void WriteEnumFromStringConversion(std::ostream& os, const EnumDescriptor& enumDescr)
{
   auto& enumName = enumDescr.enumName;
   auto scopePrefix = enumDescr.isScoped ? enumName + "::" : std::string();
   os << "inline " << enumName << " StringTo" << enumName << "(const char* itemName)\n{\n";
   os << "    static std::pair<const char*, " << enumName << "> items[] = {\n";
   for (auto& i : enumDescr.enumItems)
       os << "        {\"" << i << "\", " << scopePrefix << i << "},\n";
   os <<R"(    };
   auto p = std::lower_bound(begin(items), end(items), itemName,
                  [](auto&& i, auto&& v) {return strcmp(i.first, v) < 0;});
   if (p == end(items) || strcmp(p->first, itemName) != 0)
       throw std::bad_cast();
   return p->second;
})";
}
```

Generation tool implementation - Generation

Some tips and tricks about the generation:

- You can’t fully rely on the ‘**NamedDecl::getQualifiedNameAsString**’ method - it can return an invalid C++ identifier
- Therefore, in some cases it’s better to manually build full-qualified name of the declaration
- **clang::Decl::print** method works perfectly for clang::ValueDecl children…
- … if you don’t care about formatting
- Be careful with ‘bool’ type. By default Clang names it as ‘_Bool’

Generation tool implementation - Generation

A couple of words about the diagnostic output:

- Format of the tool-specific diagnostic should correspond to the target compiler format (i. e. MSVC format should be used if the generation result will be compiled with MSVC)
- Clang diagnostic should be suppressed
- Tool-specific diagnostic should contain the source location of the incorrect construction. This location can be obtained for almost any AST node and converted via **clang::SourceManager** to the human-readable format

The code is written and needs to be built!

Generation tool implementation - Build

Preconditions:

- You have to build llvm and Clang manually in order to get Clang C++ libraries
- You have to use CMake build system in order to integrate with Clang smoothly
- You have to build Clang and your tool with the same toolchain (obviously)

Generation tool implementation - Build

Easy way:

1. Follow the instruction from the official Clang documentation: https://clang.llvm.org/docs/LibASTMatchersTutorial.html
2. Build your tool as a part of Clang source tree (in should be placed in <llvm-root>/tools/clang/tools/extra directory)

This is the best way if you don’t familiar with CMake.

Generation tool implementation - Build

Harder way:

1. Create CMakeLists.txt for your tool
2. Include CMake modules from the llvm installation
3. Declare your target with the ‘add_llvm_executable’ command
4. Declare linkage with the required Clang libraries

Generation tool implementation - Build

```
project (enum2string)

set(LLVM_INSTALL_PREFIX $ENV{LLVM_INSTALL_PREFIX} CACHE PATH "Path to LLVM installation root directory")

list (APPEND CMAKE_MODULE_PATH ${LLVM_INSTALL_PREFIX}/lib/cmake/llvm)
list (APPEND CMAKE_MODULE_PATH ${LLVM_INSTALL_PREFIX}/lib/cmake/clang)

include(AddLLVM)
include(ClangConfig)

include_directories(${CLANG_INCLUDE_DIRS} ${LLVM_INCLUDE_DIRS})

add_llvm_executable(enum2string main.cpp)

target_link_libraries(enum2string ClangAST ClangBasic ClangDriver ClangFrontend ClangRewriteFrontend
   ClangStaticAnalyzerFrontend ClangTooling)
```

The tool is completed. Let’s use it!

- Introduction
- Brief tour to Clang C++ API
- Generation tool implementation
- **Generation tool usage**

Generation tool usage

Basic way: run the tool from the command line:

```
> enum2string -o test_enums_gen.h ./test_enums.h -- -x c++
```

Generation tool usage

A few nuances:

- For the header files you have to manually specify the language (if they’ve got ‘.h’ extension)
- You have to pass all the include file paths, defines and some other options to the generator
- In some cases you can't rely on the compilation database, produced by CMake

Generation tool usage

Easy way: integration with CMake build system:

1. Add tool invocation via ‘add_custom_command’ command into CMakeLists.txt
2. Pass to this command input files, output file name and invocation options
3. Specify dependencies
4. Add generation result to the target ‘add_xxx’ command

Generation tool usage

Generation tool invocation command:

```
set (CODEGEN_DIR ${CMAKE_CURRENT_BINARY_DIR}/codegen)
file (MAKE_DIRECTORY ${CODEGEN_DIR}/generated)
set (ENUM_CONV_FILE ${CODEGEN_DIR}/generated/enum_conv_gen.h)

add_custom_command(OUTPUT ${ENUM_CONV_FILE}
   COMMAND ${CODEGEN_BIN_NAME} -o ${ENUM_CONV_FILE}
       ${CMAKE_CURRENT_SOURCE_DIR}/test_enums.h -- Clang-cl -std=c++14 -x c++
${CMAKE_CXX_FLAGS}
   MAIN_DEPENDENCY ${CMAKE_CURRENT_SOURCE_DIR}/test_enums.h
   COMMENT "Generating enum2string converters for
${CMAKE_CURRENT_SOURCE_DIR}/test_enums.h"
   )
```

Generation tool usage

Use generation results:

```
include_directories(
  ${CODEGEN_DIR}
  )

add_executable(${PROJECT_NAME}
  ${Sources}
  ${Headers}
  ${ENUM_CONV_FILE}
  )
```

Generation tool usage

Integration with other build system (qmake/Makefile/MSBuild/bjam/...)

- Depends on abilities of the particular build system
- The proper way of the external tools usage should be described in the build system documentation

Generation tool usage

```
#include <gtest/gtest.h>

#include <generated/enum_conv_gen.h>
#include "test_enums.h"

TEST(Enum2String, ConvertToString_Successfull)
{
  EXPECT_STREQ("Item1", Enum1ToString(Item1));
  EXPECT_STREQ("Item1", Enum2ToString(Enum2::Item1));
}

TEST(Enum2String, ConvertFromString_Successfull)
{
  EXPECT_EQ(Item1, StringToEnum1("Item1"));
  EXPECT_EQ(Enum2::Item1, StringToEnum2("Item1"));
}
```

Bingo!

Real-life usage - In Our Project

- Enum<->String conversion functions
- Three types of data serialization/deserialization
- Object RPC proxy/stubs
- Default components implementations
- Google mocks for interfaces
- Configuration data validators

References

- Clang up-to-date documentation: https://clang.llvm.org/docs/index.html
- Clang doxygen documentation: http://clang.llvm.org/doxygen/
- enum2string generation tool example: https://github.com/flexferrum/flex_lib/tree/accu2017/tools/codegen
- slides about Clang AST: http://llvm.org/devmtg/2013-04/klimek-slides.pdf

Thank you.
Questions?