Lecture Slides for the Clang Libraries (LLVM/Clang 15)

Edition 0.0

Michael D. Adams

Department of Electrical and Computer Engineering
University of Victoria
Victoria, British Columbia, Canada

To obtain the most recent version of these lecture slides (with functional hyperlinks) or for additional information and resources related to these slides (including errata), please visit:

`https://www.ece.uvic.ca/~mdadams/cppbook`

If you like these lecture slides, **please consider posting a review** of them at:

`https://play.google.com/store/search?q=ISBN:9781990707049&c=books` or
`https://books.google.com/books?vid=ISBN9781990707049`

youtube.com/iamcanadian1867 github.com/mdadams @mdadams16

The author has taken care in the preparation of this document, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This document was typeset with LaTeX.

ISBN 978-1-990707-04-9 (PDF)

License I

Creative Commons Legal Code

Attribution-NonCommercial-NoDerivs 3.0 Unported

> CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed

License II

in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. "Distribute" means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this

License III

License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to 8(f), all rights not expressly granted by Licensor

License IV

are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.

b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

c. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice,

License V

terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. For the avoidance of doubt:

    i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

    ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

    iii. Voluntary License Schemes. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted

License VI

under Section 4(b).

e. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this

License VII

License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

License VIII

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of this License.

Creative Commons may be contacted at https://creativecommons.org/.

Other Textbooks and Lecture Slides by the Author I

1. M. D. Adams, *Exercises for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-5-5 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/cppbook.
2. M. D. Adams, *Lecture Slides for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-4-8 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/cppbook.
3. M. D. Adams, *Multiresolution Signal and Geometry Processing: Filter Banks, Wavelets, and Subdivision (Version 2013-09-26)*, University of Victoria, Victoria, BC, Canada, Sept. 2013, ISBN 978-1-55058-507-0 (print), ISBN 978-1-55058-508-7 (PDF). Available from Google Books, Google Play Books, and author's web site https://www.ece.uvic.ca/~mdadams/waveletbook.

Other Textbooks and Lecture Slides by the Author II

4. M. D. Adams, *Lecture Slides for Multiresolution Signal and Geometry Processing (Version 2015-02-03)*, University of Victoria, Victoria, BC, Canada, Feb. 2015, ISBN 978-1-55058-535-3 (print), ISBN 978-1-55058-536-0 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/waveletbook`.
5. M. D. Adams, *Signals and Systems*, Edition 5.0, Dec. 2022, ISBN 978-1-990707-00-1 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/sigsysbook`.
6. M. D. Adams, *Lecture Slides for Signals and Systems*, Edition 5.0, Dec. 2022, ISBN 978-1-990707-02-5 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/sigsysbook`.

Other Textbooks and Lecture Slides by the Author III

7. M. D. Adams, *Lecture Slides for Linux System Programming*, Edition 0.0, Dec. 2022, ISBN 978-1-990707-03-2 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/cppbook`.

Part 0

Preface

About These Lecture Slides

- This document constitutes a set of lecture slides that are intended to be used to provide a detailed introduction to the Clang libraries in ***version 15*** of LLVM/Clang.
- Although this document specifically targets version 15 of LLVM/Clang, the information presented herein is still likely to be at least partially relevant to other versions of LLVM/Clang, especially versions that are close to 15.
- This document represents a work in progress and should be considered an ***alpha release***.
- In spite of this, it is believed that this document will be of benefit to some people. So, it is being made available in its current form.
- This document is intended to supplement the following slide deck:
  - M. D. Adams, *Lecture Slides for Programming in C++ (Version 2021-04-01)*, Apr. 2021, ISBN 978-0-9879197-4-8 (PDF). Available from Google Books, Google Play Books, and author's web site `https://www.ece.uvic.ca/~mdadams/cppbook`.

Typesetting Conventions

- In a definition, the term being defined is often typeset in a font **like this**.
- To emphasize particular words, the words are typeset in a font ***like this***.
- To show that particular text is associated with a hyperlink to an internal target, the text is typeset like this.
- To show that particular text is associated with a hyperlink to an external document, the text is typeset like this.
- URLs are typeset like `https://www.ece.uvic.ca/~mdadams`.

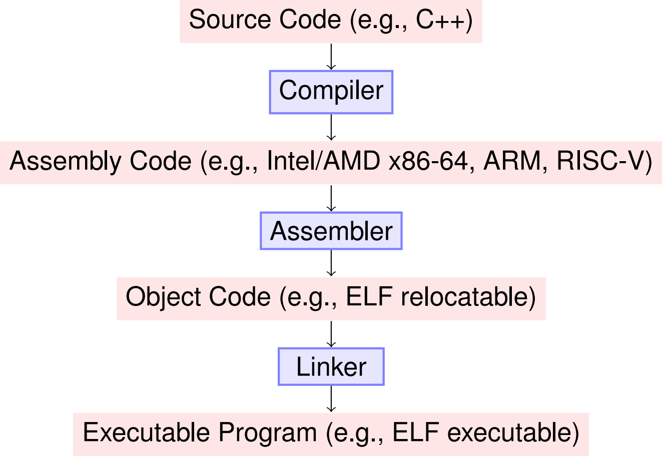
Companion Git Repository

- These lecture slides have a companion Git repository.
- Numerous code examples are available from this repository.
- This repository is hosted by GitHub.
- The URL of the main repository page on GitHub is:
  - `https://github.com/mdadams/clang_libraries_companion`
- The URL of the actual repository itself is:
  - `https://github.com/mdadams/clang_libraries_companion.git`

Part 1

Compilers

Building Code: Compiling, Assembling, and Linking



Source Code (e.g., C++)

Compiler

Assembly Code (e.g., Intel/AMD x86-64, ARM, RISC-V)

Assembler

Object Code (e.g., ELF relocatable)

Linker

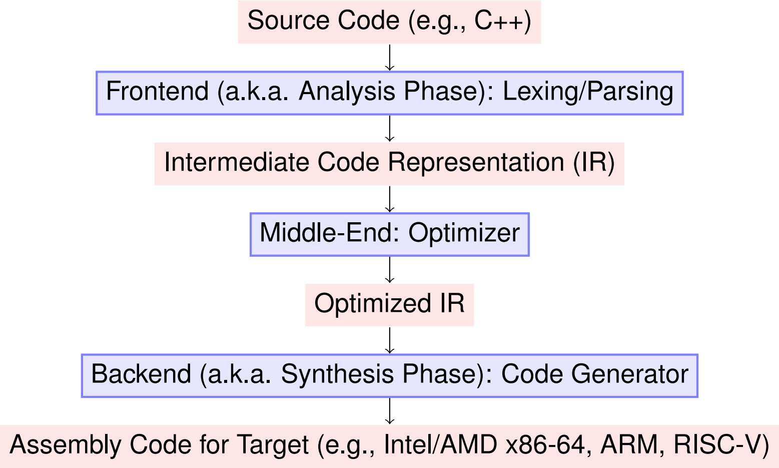
Executable Program (e.g., ELF executable)

- in context of this discussion, interested only in compiler

Section 1.1

Structure of Compiler

Structure of Compiler



Source Code (e.g., C++)

Frontend (a.k.a. Analysis Phase): Lexing/Parsing

Intermediate Code Representation (IR)

Middle-End: Optimizer

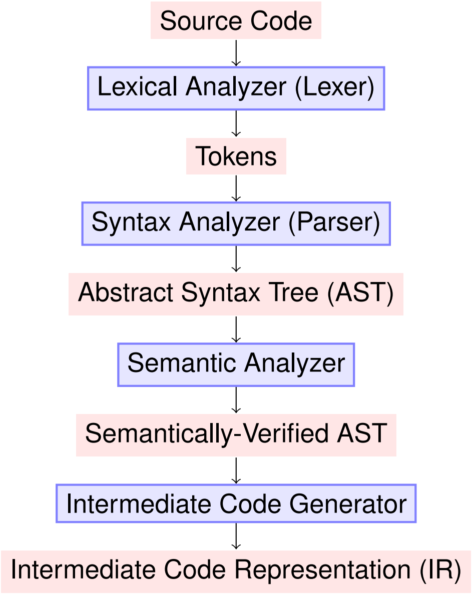
Optimized IR

Backend (a.k.a. Synthesis Phase): Code Generator

Assembly Code for Target (e.g., Intel/AMD x86-64, ARM, RISC-V)

- same IR used by all frontends and backends (avoids $M \times N$ problem)
- in context of this discussion, primary focus is compiler frontend

Structure of Compiler Frontend (Analysis Phase)


Source Code
Lexical Analyzer (Lexer)
Tokens
Syntax Analyzer (Parser)
Abstract Syntax Tree (AST)
Semantic Analyzer
Semantically-Verified AST
Intermediate Code Generator
Intermediate Code Representation (IR)

- frontend is language dependent but machine independent
- in context of this discussion, interest lies with tokens and AST, not IR

Lexical Analyzer (Lexer)

- **lexical analyzer** (also known as **lexer**) reads source code as sequence of characters and groups them into tokens
- tokens often defined using regular expressions
- token might correspond to entity such as: identifier, keyword, separator, operator, literal, comment
- example:

Code Fragment

```
foo = bar + 42;
```

Token Stream

Token Type	Token Value
identifier	`foo`
operator	assignment
identifier	`bar`
operator	addition
literal	`42`
semicolon	—

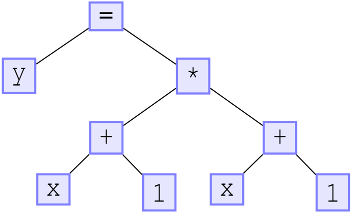
Syntax Analyzer (Parser)

- **syntax analyzer** (also known as **parser**) reads sequence of tokens and uses them to construct abstract syntax tree (AST)
- **AST** is tree-based data structure used to represent semantics of source code
- much of work of compiler frontend performed using AST
- unlike concrete syntax tree (also known as parse tree), AST typically (but not always) omits information that is not necessary for characterizing code structure (e.g., braces and parentheses)
- example:

Code Fragment

y = (x + 1) * (x + 1)

AST for Code Fragment



Semantic Analyzer

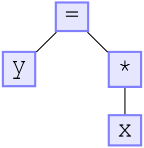
- checks if AST is valid (i.e., corresponds to well-formed code)
- semantic analysis would typically include checks for such things as:
  - invalid types
  - invalid operands
  - invalid arguments in function calls
  - undeclared identifiers/variables
- example:

Code Fragment

`y = *x`

where `x` and `y` of type **`int`**
(code invalid, cannot
dereference `x`)

AST for Code Fragment
(Not Semantically Valid)



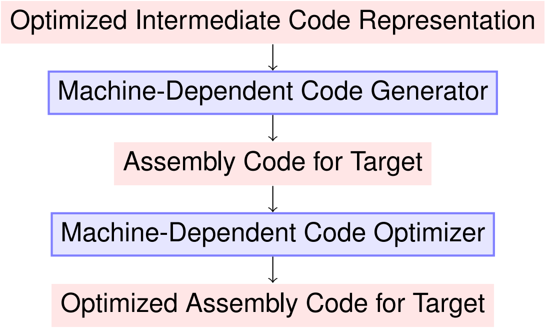
Intermediate Code Generator

- **intermediate code generator** produces intermediate code representation (IR) from AST
- particular IR used highly dependent on compiler
- IR provides way to describe code in manner than is independent of language and target architecture
- can think of IR as very generic assembly-like language for some idealized/fictitious processor architecture
- in case of LLVM, for example, IR can be thought of as assembly-code for processor with infinite number of registers

Optimizer

- **IR optimizer** performs transformations to IR in attempt to produce more efficient code
- since working with IR, optimizations are ***machine independent***
- some common types of optimizations might include:
  - eliminate unreachable code
  - eliminate dead stores
  - eliminate unused variables
- IR optimizer not to be confused with machine-dependent optimizer in compiler backend

Structure of Compiler Backend (Synthesis Phase)


Optimized Intermediate Code Representation
Machine-Dependent Code Generator
Assembly Code for Target
Machine-Dependent Code Optimizer
Optimized Assembly Code for Target

- backend is machine dependent but language independent

Machine-Dependent Code Generator and Optimizer

- **machine-dependent code generator** produces assembly code for target architecture from IR
- must map operations in IR onto instructions of target architecture
- must map registers and storage used by operations in IR onto memory and register set of target architecture
- assembly code produced may then be further optimized to yield final assembly-code output

Example: Source Code

simple_1.cpp

```
int add(int x, int y) {
    return x + y;
}
```

Example: Tokens

simple_1.cpp

```
int add(int x, int y) {
    return x + y;
}
```

Command

```
clang -std=c++20 -Xclang -dump-tokens -fsyntax-only simple_1.cpp
```

Output (Standard Error)

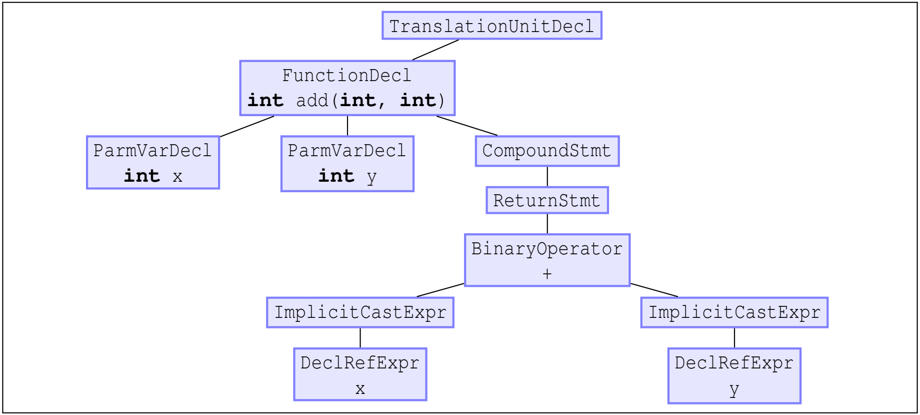
```
int 'int'   [StartOfLine] Loc=<simple_1.cpp:1:1>
identifier 'add'   [LeadingSpace] Loc=<simple_1.cpp:1:5>
l_paren '('    Loc=<simple_1.cpp:1:8>
int 'int'      Loc=<simple_1.cpp:1:9>
identifier 'x'  [LeadingSpace] Loc=<simple_1.cpp:1:13>
comma ','      Loc=<simple_1.cpp:1:14>
int 'int'   [LeadingSpace] Loc=<simple_1.cpp:1:16>
identifier 'y'  [LeadingSpace] Loc=<simple_1.cpp:1:20>
r_paren ')'    Loc=<simple_1.cpp:1:21>
l_brace '{' [LeadingSpace] Loc=<simple_1.cpp:1:23>
return 'return' [StartOfLine] [LeadingSpace] Loc=<simple_1.cpp:2:2>
identifier 'x'  [LeadingSpace] Loc=<simple_1.cpp:2:9>
plus '+'    [LeadingSpace] Loc=<simple_1.cpp:2:11>
identifier 'y'  [LeadingSpace] Loc=<simple_1.cpp:2:13>
semi ';'       Loc=<simple_1.cpp:2:14>
r_brace '}' [StartOfLine] Loc=<simple_1.cpp:3:1>
eof ''     Loc=<simple_1.cpp:3:2>
```

Example: AST [Graphical]

simple_1.cpp

```
int add(int x, int y) {
    return x + y;
}
```

AST (Clang 14; Slightly Abridged)



TranslationUnitDecl

FunctionDecl
int add(int, int)

ParmVarDecl
int x

ParmVarDecl
int y

CompoundStmt

ReturnStmt

BinaryOperator
+

ImplicitCastExpr

DeclRefExpr
x

ImplicitCastExpr

DeclRefExpr
y

Example: AST [clang-check]

simple_1.cpp

```
int add(int x, int y) {
    return x + y;
}
```

Command

```
clang-check -ast-dump -ast-dump-filter=add simple_1.cpp -- \
  -fno-color-diagnostics -std=c++20
```

Output (Clang 14; Standard Output)

```
Dumping add:
FunctionDecl 0x74c4e98 </home/jdoe/simple/simple_1.cpp:1:1, line:3:1> line:1:5 add ’int (int, int)’
|-ParmVarDecl 0x74c4cf8 <col:9, col:13> col:13 used x ’int’
|-ParmVarDecl 0x74c4d80 <col:16, col:20> col:20 used y ’int’
`-CompoundStmt 0x74c5050 <col:23, line:3:1>
  `-ReturnStmt 0x74c5040 <line:2:2, col:13>
    `-BinaryOperator 0x74c5020 <col:9, col:13> ’int’ ’+’
      |-ImplicitCastExpr 0x74c4ff0 <col:9> ’int’ <LValueToRValue>
      | `-DeclRefExpr 0x74c4fb0 <col:9> ’int’ lvalue ParmVar 0x74c4cf8 ’x’ ’int’
      `-ImplicitCastExpr 0x74c5008 <col:13> ’int’ <LValueToRValue>
        `-DeclRefExpr 0x74c4fd0 <col:13> ’int’ lvalue ParmVar 0x74c4d80 ’y’ ’int’
```

Example: AST [clang]

simple_1.cpp

```
int add(int x, int y) {
    return x + y;
}
```

Command

```
clang -std=c++20 -Xclang -ast-dump -fsyntax-only -fno-color-diagnostics \
  simple_1.cpp
```

Output (Clang 14; Standard Output; Abridged)

```
TranslationUnitDecl 0x93dfe78 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x93e0700 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x93e0440 '__int128'
|-TypedefDecl 0x93e0778 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x93e0460 'unsigned __int128'
|-TypedefDecl 0x93e0b30 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_tag'
| `-RecordType 0x93e0880 '__NSConstantString_tag'
|   `-CXXRecord 0x93e07d8 '__NSConstantString_tag'
|-TypedefDecl 0x93e0bd8 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x93e0b90 'char *'
|   `-BuiltinType 0x93dff20 'char'
|-TypedefDecl 0x942a930 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag[1]'
| `-ConstantArrayType 0x942a8d0 '__va_list_tag[1]' 1
|   `-RecordType 0x93e0ce0 '__va_list_tag'
|     `-CXXRecord 0x93e0c38 '__va_list_tag'
`-FunctionDecl 0x942ab48 <simple_1.cpp:1:1, line:3:1> line:1:5 add 'int (int, int)'
  |-ParmVarDecl 0x942a9a8 <col:9, col:13> col:13 used x 'int'
  |-ParmVarDecl 0x942aa30 <col:16, col:20> col:20 used y 'int'
  `-CompoundStmt 0x942ad00 <col:23, line:3:1>
    `-ReturnStmt 0x942acf0 <line:2:2, col:13>
[deleted text]
```

Example: IR

Command

```
clang++ -S -emit-llvm simple_1.cpp
```

LLVM IR (simple_1.ll)

```
; ModuleID = 'simple_1.cpp'
source_filename = "simple_1.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: mustprogress noinline nounwind optnone uwtable
define dso_local noundef i32 @_Z3addii(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

attributes #0 = { mustprogress noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"uwtable", i32 1}
!2 = !{i32 7, !"frame-pointer", i32 2}
!3 = !{!"clang version 14.0.0"}
```

Example: Optimized IR

Command

```
clang++ -O3 -S -emit-llvm -o simple_1-opt.ll simple_1.cpp
```

LLVM Optimized IR (`simple_1-opt.ll`)

```
; ModuleID = 'simple_1.cpp'
source_filename = "simple_1.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn
define dso_local noundef i32 @_Z3addii(i32 noundef %0, i32 noundef %1) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  ret i32 %3
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn "frame-pointer"="none" "min-legal-vector-width"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"uwtable", i32 1}
!2 = !{!"clang version 14.0.0"}
```

Example: x86-64 Assembly Code

Command

```
clang++ -S simple_1.cpp
```

x86-64 Assembly Code (simple_1.s)

```
    .text
    .file  "simple_1.cpp"
    .globl _Z3addii                    # -- Begin function _Z3addii
    .p2align   4, 0x90
    .type  _Z3addii,@function
_Z3addii:                          # @_Z3addii
    .cfi_startproc
# %bb.0:
    pushq  %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq   %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl   %edi, -4(%rbp)
    movl   %esi, -8(%rbp)
    movl   -4(%rbp), %eax
    addl   -8(%rbp), %eax
    popq   %rbp
    .cfi_def_cfa %rsp, 8
    retq
.Lfunc_end0:
    .size  _Z3addii, .Lfunc_end0-_Z3addii
    .cfi_endproc
                                   # -- End function
    .ident "clang version 14.0.0"
    .section   ".note.GNU-stack","",@progbits
    .addrsig
```

Example: Optimized x86-64 Assembly Code

Command

```
clang++ -O3 -S -o simple_1-opt.s simple_1.cpp
```

Optimized x86-64 Assembly Code (simple_1-opt.s)

```
    .text
    .file  "simple_1.cpp"
    .globl _Z3addii                         # -- Begin function _Z3addii
    .p2align   4, 0x90
    .type  _Z3addii,@function
_Z3addii:                               # @_Z3addii
    .cfi_startproc
# %bb.0:
                                        # kill: def $esi killed $esi def $rsi
                                        # kill: def $edi killed $edi def $rdi
    leal   (%rdi,%rsi), %eax
    retq
.Lfunc_end0:
    .size  _Z3addii, .Lfunc_end0-_Z3addii
    .cfi_endproc
                                        # -- End function
    .ident "clang version 14.0.0"
    .section   ".note.GNU-stack","",@progbits
    .addrsig
```

Section 1.2

Name Mangling

Example: Name Demangling

Command (With Mangled Names as Arguments to `llvm-cxxfilt`)

```
llvm-cxxfilt _Z3addii
```

Output

```
add(int, int)
```

Command (Run `llvm-cxxfilt` as Filter)

```
echo "_Z3addii _Z3addii _Z3addii" | llvm-cxxfilt
```

Output

```
add(int, int) add(int, int) add(int, int)
```

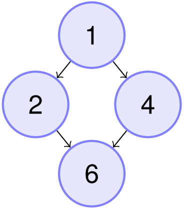
Section 1.3

Control-Flow Graphs (CFGs)

Control-Flow Graphs

- **control-flow graph (CFG)** is directed graph that shows paths of execution in code
- nodes correspond to statement fragments or statements
- given two nodes i and j in CFG, edge from i to j exists if and only if statement fragment corresponding to node j can be executed immediately after statement fragment corresponding to node i
- code can be viewed in terms of CFG
- CFGs are particularly useful for performing certain types of code analysis

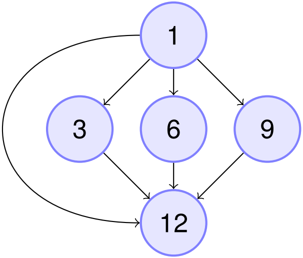
CFG Examples: If-Else and Switch



declarations:

```
bool c; double x; double y;
```

```
1   if (c) {
2       y = x * x;
3   } else {
4       y = x;
5   }
6   // ...
```

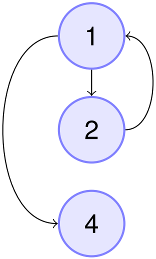


declarations:

```
int n; double x; double y;
```

```
1   switch (n) {
2   case 0:
3       y = 0.0;
4       break;
5   case 1:
6       y = 2.0 * x;
7       break;
8   case 2:
9       y = 0.5 * x * x;
10      break;
11  }
12  // ...
```

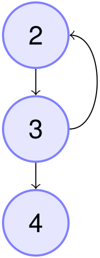
CFG Examples: While and Do-While Loops



declarations:

```
    int n;
```

```
1   while (n > 0) {
2       --n;
3   }
4   // ...
```

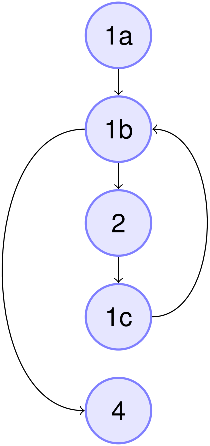


declarations:

```
    int n;
```

```
1   do {
2       --n;
3   while (n > 0);
4   // ...
```

CFG Examples: For Loop



declarations:

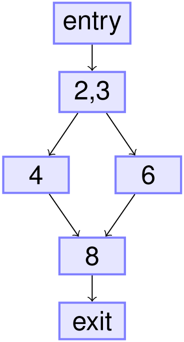
```
    int a[1024];
```

```
1   for (int i = 0; i < 1024; ++i) {
2       a[i] = 0;
3   }
4   // ...
```

Live-Variable Analysis

- variable is said to be **live** at some point in code if it holds value that may be needed in future (or equivalently, if its value may be read before next time variable is written)
- at each point in code execution, each variable is either live or dead
- **live-variable analysis** (also called **liveness analysis**) is classic data-flow analysis to calculate variables that are live at each point in code
- some uses for liveness analysis include:
  - detecting dead stores (i.e., value written to variable that is never read)
  - detecting use of uninitialized variables
  - register allocation (i.e., deciding which variables should be allocated to registers)

Liveness Example



```
1  int foo(int x, int y) {
2      int t = x * y; // t is dead
3      if ((x + 1) * (x - 1) == y) {
4          t = 1;
5      } else {
6          t = 2;
7      }
8      return t;
9  }
```

Performing Live-Variable Analysis

- CFG used to perform liveness analysis
- liveness analysis flows backwards through code considering each statement/expression
- variable becomes live when statement/expression reads value of variable
- variable becomes dead when statement/expression assigns value to variable

Part 2

LLVM and Clang

LLVM

- LLVM is collection of modular compiler and toolchain components
- much of LLVM code is organized as collection of libraries
- frontends for numerous languages, including:
  - Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, and Swift
- modern source- and target-independent optimizer
- backends for numerous targets, including:
  - IA-32 (i.e., 32-bit x86), x86-64, ARM, Qualcomm Hexagon, MIPS, PowerPC, RISC-V, SPARC, z/Architecture
- numerous tools and libraries, including:
  - debugger (LLDB), linker (LLD)
  - C++ standard library implementation (libc++)
- open-source (licensed under Apache 2.0 Licence with LLVM exceptions)
- originally written by Vikram Adve and Chris Lattner
- initial release in 2003

Clang

- Clang is compiler frontend for:
  - C, C++, Objective-C, and Objective-C++ programming languages
- has support for:
  - OpenMP, OpenCL, CUDA, and HIP
- Clang very modular and organized as collection of libraries along with some programs that use these libraries
- library for each of lexer, parser, semantic analyzer, and so on
- Clang originally developed by Apple and then later released under open-source license in 2007
- Clang compiler frontend not to be confused with compiler-driver programs `clang` and `clang++`, which can do more than just compile (e.g., optimize, assemble, and link)

Section 2.1

Obtaining and Building LLVM/Clang

Obtaining LLVM/Clang

- LLVM/Clang source code can be obtained from LLVM monorepo at:
  - `https://github.com/llvm/llvm-project.git`
- repository is quite large
- can also obtain individual release tarballs from GitHub at:
  - `https://github.com/llvm/llvm-project/releases`
- for example, tarball for LLVM/Clang 15.0.6 available from:
  - `https://github.com/llvm/llvm-project/releases/download/llvmorg-15.0.6/llvm-project-15.0.6.src.tar.xz`
- in most cases, probably preferable to obtain LLVM/Clang from Git repository

LLVM/Clang, RTTI, and EH

- by default, LLVM/Clang is built with run-time type-identification (RTTI) and exception handling (EH) disabled
- if LLVM/Clang libraries and code using these libraries built with different RTTI/EH settings, care must be taken to avoid accidentally relying on RTTI/EH in code where these features have been disabled
- some problematic things would include:
  - throwing exceptions that must propagate through code for which exception handling disabled (e.g., Clang libraries)
  - using **`typeid`** or **`dynamic_cast`** with types for which RTTI disabled
- probably advisable to build LLVM/Clang with RTTI/EH enabled if any other code is to be built with RTTI/EH enabled

Building Clang With CMake

- LLVM/Clang employs CMake-based build
- basic build and install of Clang can be performed using command sequence similar to:

```
cmake -H$src_dir/llvm -B$build_dir \
  -DCMAKE_BUILD_TYPE=$build_type \
  -DCMAKE_INSTALL_PREFIX=$install_dir \
  -DLLVM_ENABLE_PROJECTS="clang;openmp"
cmake --build $build_dir
cmake --build $build_dir --target install
```

- usually some additional CMake options are needed or desirable
- for more information, see:
  https://llvm.org/docs/GettingStarted.html

CMake Settings for Building Clang (1)

- `CMAKE_BUILD_TYPE`: specify type of build (e.g., `Debug`, `RelWithDebInfo`, or `Release`)
- `LLVM_ENABLE_PROJECTS`: semicolon-separated list of projects to be built (must at least include `clang`); some projects include:
  - `clang`: frontend for C/C++ family of languages
  - `clang-tools-extras`: various utilities such as `clang-format`, `clang-tidy`, and `clangd`
  - `lld`: LLVM linker
  - `lldb`: LLVM debugger
- `LLVM_ENABLE_RUNTIMES`: semicolon-separated list of runtimes to be built; run-times built using just-built compiler; some runtimes include:
  - `libcxx`: libc++ C++ standard library
  - `libcxxabi`: low-level support for libc++ C++ standard library
  - `libunwind`: stack-unwinding library
  - `compiler-rt`: compiler run-time support libraries (e.g., run-time libraries for code sanitizers such as ASan, MSan, and TSan)
- some components can be built as project or runtime (e.g., `openmp`)

CMake Settings for Building Clang (2)

- LLVM_BUILD_LLVM_DYLIB: boolean flag indicating if LLVM should be built as single shared library
- LLVM_ENABLE_ASSERTIONS: boolean flag indicating if assertions should be enabled
- LLVM_ENABLE_RTTI: boolean flag indicating if RTTI should be enabled
- LLVM_ENABLE_EH: boolean flag indicating if exception handling (EH) should be enabled
- LVM_PARALLEL_COMPILE_JOBS: specifies maximum number of parallel jobs for compiling
- LVM_PARALLEL_LINK_JOBS: specifies maximum number of parallel jobs for linking
- for more details, see:
  - https://llvm.org/docs/CMake.html

Building Clang Distribution

- use `install-distribution` target instead of `install`
- `LLVM_DISTRIBUTION_COMPONENTS`: specifies which software components to be included in distribution (e.g., `cmake-exports`, `LLVM`, `llvm-headers`)

sde_install_clang Script

- can use sde_install_clang script to facilitate easier installation of LLVM/Clang
- script will obtain specified release of LLVM/Clang from either Git repository or tarball and then build and install software
- sde_install_clang script can be obtained from following Git repository:
  - https://github.com/mdadams/sde.git
- invoking sde_install_clang with no command-line arguments will cause script to print help information and exit

llvm-config Program

- can query how LLVM was configured at time it was built using `llvm-config` program
- can query such things as:
  - LLVM build mode (e.g., Debug or Release)
  - whether LLVM was built with RTTI enabled
  - whether LLVM was built with assertions enabled
  - libraries needed to link against various LLVM components
  - installation directory for LLVM headers
- for example, to print build mode, RTTI setting, and assertion mode, use:

```
llvm-config --build-mode --has-rtti --assertion-mode
```

- for more information, see:
  - https://llvm.org/docs/CommandGuide/llvm-config.html

Section 2.2

Compiler Driver and Stages of Compilation

Clang Compiler Driver

- compiler-driver programs, `clang` and `clang++`, responsible for performing various tasks associated with compilation (e.g., compiling, assembling, and linking)
- may perform task directly or by running another program
- determines what tasks (i.e., phases of processing) required
- runs programs needed to perform those tasks
- compiler driver can perform many tasks directly itself, which helps to avoid need to write data to file in order to pass it to next stage in compilation process

Example: Printing Compiler Phases

Command

```
clang++ -ccc-print-phases -c hello.cpp
```

Output (Standard Error)

```
         +- 0: input, "hello.cpp", c++
      +- 1: preprocessor, {0}, c++-cpp-output
   +- 2: compiler, {1}, ir
+- 3: backend, {2}, assembler
4: assembler, {3}, object
```

Command

```
clang++ -ccc-print-phases hello.cpp
```

Output (Standard Error)

```
            +- 0: input, "hello.cpp", c++
         +- 1: preprocessor, {0}, c++-cpp-output
      +- 2: compiler, {1}, ir
   +- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
5: linker, {4}, image
```

Example: Printing Tool Executions

Command

```
clang++ -### hello.cpp
```

Output (Standard Error)

```
clang version 13.0.0 (Fedora 13.0.0-3.fc35)
Target: x86_64-redhat-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
 "/usr/bin/clang-13" "-cc1" "-triple" "x86_64-redhat-linux-gnu" "-emit-obj" "-mrelax-all" "--mrelax-
       ↪ relocations" "-disable-free" "-disable-llvm-verifier" "-discard-value-names" "-main-file-name" "
       ↪ hello.cpp" "-mrelocation-model" "static" "-mframe-pointer=all" "-fmath-errno" "-fno-rounding-math"
       ↪ "-mconstructor-aliases" "-munwind-tables" "-target-cpu" "x86-64" "-tune-cpu" "generic" "-debugger-
       ↪ tuning=gdb" "-fcoverage-compilation-dir=/home/mdadams/work/git/clang_slides/software/
       ↪ llvm_clang_usage" "-resource-dir" "/usr/lib64/clang/13.0.0" "-internal-isystem" "/usr/bin/../lib/
       ↪ gcc/x86_64-redhat-linux/11/../../../../include/c++/11" "-internal-isystem" "/usr/bin/../lib/gcc/
       ↪ x86_64-redhat-linux/11/../../../../include/c++/11/x86_64-redhat-linux" "-internal-isystem" "/usr/
       ↪ bin/../lib/gcc/x86_64-redhat-linux/11/../../../../include/c++/11/backward" "-internal-isystem" "/
       ↪ usr/lib64/clang/13.0.0/include" "-internal-isystem" "/usr/local/include" "-internal-isystem" "/usr/
       ↪ bin/../lib/gcc/x86_64-redhat-linux/11/../../../../x86_64-redhat-linux/include" "-internal-externc-
       ↪ isystem" "/include" "-internal-externc-isystem" "/usr/include" "-fdeprecated-macro" "-fdebug-
       ↪ compilation-dir=/home/mdadams/work/git/clang_slides/software/llvm_clang_usage" "-ferror-limit" "19"
       ↪  "-fgnuc-version=4.2.1" "-fcxx-exceptions" "-fexceptions" "-faddrsig" "-D__GCC_HAVE_DWARF2_CFI_ASM
       ↪ =1" "-o" "/tmp/hello-96987e.o" "-x" "c++" "hello.cpp"
 "/usr/bin/ld" "--hash-style=gnu" "--build-id" "--eh-frame-hdr" "-m" "elf_x86_64" "-dynamic-linker" "/lib64/
       ↪ ld-linux-x86-64.so.2" "-o" "a.out" "/usr/bin/../lib/gcc/x86_64-redhat-linux/11/../../../../lib64/
       ↪ crt1.o" "/usr/bin/../lib/gcc/x86_64-redhat-linux/11/../../../../lib64/crti.o" "/usr/bin/../lib/gcc/
       ↪ x86_64-redhat-linux/11/crtbegin.o" "-L/usr/bin/../lib/gcc/x86_64-redhat-linux/11" "-L/usr/bin/../
       ↪ lib/gcc/x86_64-redhat-linux/11/../../../../lib64" "-L/lib/../lib64" "-L/usr/lib/../lib64" "-L/usr/
       ↪ bin/../lib" "-L/lib" "-L/usr/lib" "/tmp/hello-96987e.o" "-lstdc++" "-lm" "-lgcc_s" "-lgcc" "-lc" "-
       ↪ lgcc_s" "-lgcc" "/usr/bin/../lib/gcc/x86_64-redhat-linux/11/crtend.o" "/usr/bin/../lib/gcc/x86_64-
       ↪ redhat-linux/11/../../../../lib64/crtn.o"
```

Example: Source Code

simple_2.cpp

```cpp
int factorial(int n) {
    int result = 1;
    while (n >= 2) {result *= n--;}
    return result;
}
```

Example: Tokens

simple_2.cpp

```
int factorial(int n) {
   int result = 1;
   while (n >= 2) {result *= n--;}
   return result;
}
```

Command

```
clang -std=c++20 -Xclang -dump-tokens -fsyntax-only simple_2.cpp
```

Output (Standard Error; Abridged)

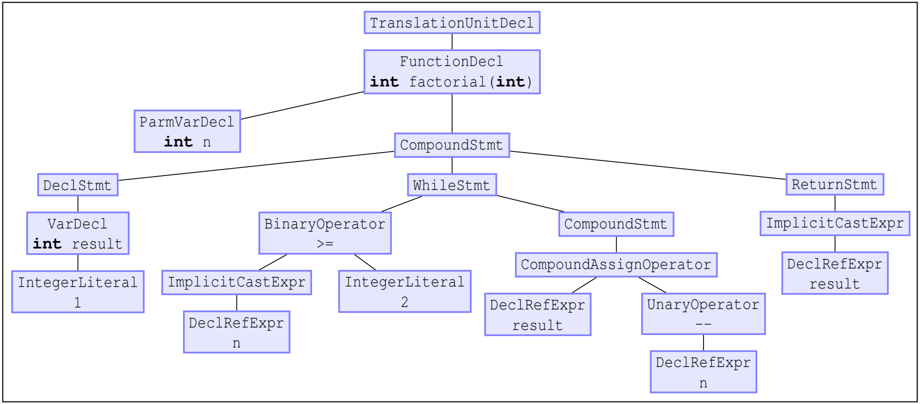
```
int 'int'   [StartOfLine] Loc=<simple_2.cpp:1:1>
identifier 'factorial' [LeadingSpace] Loc=<simple_2.cpp:1:5>
l_paren '('   Loc=<simple_2.cpp:1:14>
int 'int'     Loc=<simple_2.cpp:1:15>
identifier 'n' [LeadingSpace] Loc=<simple_2.cpp:1:19>
r_paren ')'   Loc=<simple_2.cpp:1:20>
l_brace '{' [LeadingSpace] Loc=<simple_2.cpp:1:22>
int 'int'   [StartOfLine] [LeadingSpace] Loc=<simple_2.cpp:2:2>
identifier 'result' [LeadingSpace] Loc=<simple_2.cpp:2:6>
equal '='   [LeadingSpace] Loc=<simple_2.cpp:2:13>
numeric_constant '1'  [LeadingSpace] Loc=<simple_2.cpp:2:15>
semi ';'      Loc=<simple_2.cpp:2:16>
while 'while' [StartOfLine] [LeadingSpace] Loc=<simple_2.cpp:3:2>
l_paren '(' [LeadingSpace] Loc=<simple_2.cpp:3:8>
identifier 'n'    Loc=<simple_2.cpp:3:9>
greaterequal '>='  [LeadingSpace] Loc=<simple_2.cpp:3:11>
numeric_constant '2'  [LeadingSpace] Loc=<simple_2.cpp:3:14>
[text deleted]
eof ''     Loc=<simple_2.cpp:5:2>
```

Example: AST [Graphical]

simple_2.cpp

```
int factorial(int n) {
    int result = 1;
    while (n >= 2) {result *= n--;}
    return result;
}
```

AST (Clang 14; Slightly Abridged)



TranslationUnitDecl

FunctionDecl
int factorial(int)

ParmVarDecl
int n

CompoundStmt

DeclStmt

WhileStmt

ReturnStmt

VarDecl
int result

BinaryOperator
>=

CompoundStmt

ImplicitCastExpr

DeclRefExpr
result

IntegerLiteral
1

ImplicitCastExpr

IntegerLiteral
2

CompoundAssignOperator

DeclRefExpr
n

DeclRefExpr
result

UnaryOperator
--

DeclRefExpr
n

Example: AST [clang-check]

simple_2.cpp

```
int factorial(int n) {
    int result = 1;
    while (n >= 2) {result *= n--;}
    return result;
}
```

Command

```
clang-check -ast-dump -ast-dump-filter=factorial simple_2.cpp -- -fno-color-diagnostics -std=c++20
```

Output (Clang 14; Standard Output)

```
Dumping factorial:
FunctionDecl 0x7825ea8 </home/mdadams/work/git/clang_slides/software/llvm_clang_usage/simple_2.cpp:1:1, line:5:1>
|-ParmVarDecl 0x7825db8 <col:15, col:19> col:19 used n 'int'
`-CompoundStmt 0x7826210 <col:22, line:5:1>
  |-DeclStmt 0x7826078 <line:2:2, col:16>
  | `-VarDecl 0x7825fd8 <col:2, col:15> col:6 used result 'int' cinit
  |   `-IntegerLiteral 0x7826040 <col:15> 'int' 1
  |-WhileStmt 0x78261a8 <line:3:2, col:32>
  | |-BinaryOperator 0x78260e8 <col:9, col:14> 'bool' '>='
  | | |-ImplicitCastExpr 0x78260d0 <col:9> 'int' <LValueToRValue>
  | | | `-DeclRefExpr 0x7826090 <col:9> 'int' lvalue ParmVar 0x7825db8 'n' 'int'
  | | `-IntegerLiteral 0x78260b0 <col:14> 'int' 2
  | `-CompoundStmt 0x7826190 <col:17, col:32>
  |   `-CompoundAssignOperator 0x7826160 <col:18, col:29> 'int' lvalue '*=' ComputeLHSTy='int' ComputeResultTy='int
  |     |-DeclRefExpr 0x7826108 <col:18> 'int' lvalue Var 0x7825fd8 'result' 'int'
  |     `-UnaryOperator 0x7826148 <col:28, col:29> 'int' postfix '--'
  |       `-DeclRefExpr 0x7826128 <col:28> 'int' lvalue ParmVar 0x7825db8 'n' 'int'
  `-ReturnStmt 0x7826200 <line:4:2, col:9>
    `-ImplicitCastExpr 0x78261e8 <col:9> 'int' <LValueToRValue>
      `-DeclRefExpr 0x78261c8 <col:9> 'int' lvalue Var 0x7825fd8 'result' 'int'
```

Example: AST [clang]

simple_2.cpp

```
int factorial(int n) {
    int result = 1;
    while (n >= 2) {result *= n--;}
    return result;
}
```

Command

```
clang -std=c++20 -Xclang -ast-dump -fsyntax-only -fno-color-diagnostics simple_2.cpp
```

Output (Clang 14; Standard Output; Abridged)

```
TranslationUnitDecl 0x9d53738 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x9d53fc0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x9d53d00 '__int128'
|-TypedefDecl 0x9d54038 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x9d53d20 'unsigned __int128'
|-TypedefDecl 0x9d543f0 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_tag'
| `-RecordType 0x9d54140 '__NSConstantString_tag'
|   `-CXXRecord 0x9d54098 '__NSConstantString_tag'
|-TypedefDecl 0x9d54498 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x9d54450 'char *'
|   `-BuiltinType 0x9d537e0 'char'
|-TypedefDecl 0x9d9e7b0 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag[1]'
| `-ConstantArrayType 0x9d9e750 '__va_list_tag[1]' 1
|   `-RecordType 0x9d545a0 '__va_list_tag'
|     `-CXXRecord 0x9d544f8 '__va_list_tag'
`-FunctionDecl 0x9d9e918 <simple_2.cpp:1:1, line:5:1> line:1:5 factorial 'int (int)'
  |-ParmVarDecl 0x9d9e828 <col:15, col:19> col:19 used n 'int'
  `-CompoundStmt 0x9d9ec80 <col:22, line:5:1>
    |-DeclStmt 0x9d9eae8 <line:2:2, col:16>
    | `-VarDecl 0x9d9ea48 <col:2, col:15> col:6 used result 'int' cinit
[text deleted]
```

Example: IR

Command

```
clang++ -O0 -S -emit-llvm -o simple_2.ll -Xclang -disable-O0-optnone simple_2.cpp
```

LLVM IR (`simple_2.ll`)

```
; ModuleID = ’simple_2.cpp’
source_filename = "simple_2.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: mustprogress noinline nounwind uwtable
define dso_local noundef i32 @_Z9factoriali(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  store i32 1, i32* %3, align 4
  br label %4
4:                                          ; preds = %7, %1
  %5 = load i32, i32* %2, align 4
  %6 = icmp sge i32 %5, 2
  br i1 %6, label %7, label %12
7:                                          ; preds = %4
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 %8, -1
  store i32 %9, i32* %2, align 4
  %10 = load i32, i32* %3, align 4
  %11 = mul nsw i32 %10, %8
  store i32 %11, i32* %3, align 4
  br label %4, !llvm.loop !4
12:                                         ; preds = %4
  %13 = load i32, i32* %3, align 4
  ret i32 %13
}

; [text deleted]
```

Example: Optimized IR

Command

```
clang++ -O1 -S -emit-llvm -o simple_2-opt.ll simple_2.cpp
```

LLVM Optimized IR (`simple_2-opt.ll`)

```
; ModuleID = 'simple_2.cpp'
source_filename = "simple_2.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn
define dso_local noundef i32 @_Z9factoriali(i32 noundef %0) local_unnamed_addr #0 {
  %2 = icmp sgt i32 %0, 1
  br i1 %2, label %3, label %9
3:                                                ; preds = %1, %3
  %4 = phi i32 [ %7, %3 ], [ 1, %1 ]
  %5 = phi i32 [ %6, %3 ], [ %0, %1 ]
  %6 = add nsw i32 %5, -1
  %7 = mul nsw i32 %4, %5
  %8 = icmp sgt i32 %5, 2
  br i1 %8, label %3, label %9, !llvm.loop !3
9:                                                ; preds = %3, %1
  %10 = phi i32 [ 1, %1 ], [ %7, %3 ]
  ret i32 %10
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind readnone uwtable willreturn "frame-pointer"="

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"uwtable", i32 1}
!2 = !{!"clang version 14.0.1"}
!3 = distinct !{!3, !4, !5}
!4 = !{!"llvm.loop.mustprogress"}
!5 = !{!"llvm.loop.unroll.disable"}
```

Example: x86-64 Assembly Code

Command

```
clang++ -S simple_2.cpp
```

x86-64 Assembly Code (Abridged; `simple_2.s`)

```
    .text
    .file      "simple_2.cpp"
    .globl     _Z9factoriali                # -- Begin function _Z9factoriali
    .p2align      4, 0x90
    .type      _Z9factoriali,@function
_Z9factoriali:                           # @_Z9factoriali
    .cfi_startproc
# %bb.0:
    pushq      %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq       %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl       %edi, -4(%rbp)
    movl       $1, -8(%rbp)
.LBB0_1:                                 # =>This Inner Loop Header: Depth=1
    cmpl       $2, -4(%rbp)
    jl  .LBB0_3
# %bb.2:                                 #   in Loop: Header=BB0_1 Depth=1
    movl       -4(%rbp), %eax
    movl       %eax, %ecx
    addl       $-1, %ecx
    movl       %ecx, -4(%rbp)
    imull      -8(%rbp), %eax
    movl       %eax, -8(%rbp)
    jmp .LBB0_1
.LBB0_3:
    movl       -8(%rbp), %eax
    popq       %rbp
    .cfi_def_cfa %rsp, 8
    retq
# [text deleted]
```

Example: Optimized x86-64 Assembly Code

Command

```
clang++ -O1 -S -o simple_2-opt.s simple_2.cpp
```

Optimized x86-64 Assembly Code (simple_2-opt.s)

```
    .text
    .file  "simple_2.cpp"
    .globl _Z9factoriali               # -- Begin function _Z9factoriali
    .p2align   4, 0x90
    .type  _Z9factoriali,@function
_Z9factoriali:                     # @_Z9factoriali
    .cfi_startproc
# %bb.0:
                                   # kill: def $edi killed $edi def $rdi
    movl   $1, %eax
    cmpl   $2, %edi
    jl  .LBB0_3
# %bb.1:
    movl   $1, %eax
    .p2align   4, 0x90
.LBB0_2:                           # =>This Inner Loop Header: Depth=1
    imull  %edi, %eax
    leal   -1(%rdi), %ecx
    cmpl   $2, %edi
    movl   %ecx, %edi
    jg  .LBB0_2
.LBB0_3:
    retq
.Lfunc_end0:
    .size  _Z9factoriali, .Lfunc_end0-_Z9factoriali
    .cfi_endproc
                                   # -- End function
    .ident "clang version 14.0.1"
    .section   ".note.GNU-stack","",@progbits
    .addrsig
```

Example: Name Demangling

Command (With Mangled Names as Arguments to `llvm-cxxfilt`)

```
llvm-cxxfilt _Z9factoriali
```

Output

```
factorial(int)
```

Command (Run `llvm-cxxfilt` as Filter)

```
echo "_Z9factoriali _Z9factoriali" | llvm-cxxfilt
```

Output

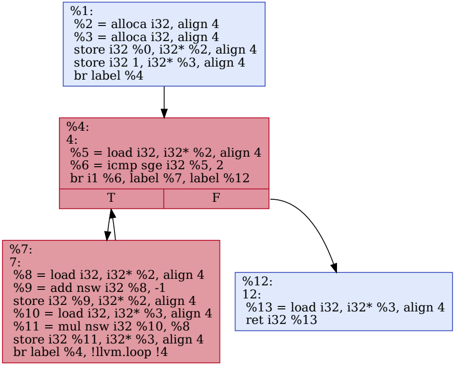
```
factorial(int) factorial(int)
```

Example: CFG for Unoptimized Code

Command

```
clang++ -O0 -S -emit-llvm -o simple_2.ll -Xclang -disable-O0-optnone simple_2.cpp
opt -dot-cfg -cfg-func-name=_Z9factoriali -cfg-dot-filename-prefix \
  dot-simple_2- simple_2.ll > /dev/null
```

CFG



```
%1:
 %2 = alloca i32, align 4
 %3 = alloca i32, align 4
 store i32 %0, i32* %2, align 4
 store i32 1, i32* %3, align 4
 br label %4
```

```
%4:
4:
 %5 = load i32, i32* %2, align 4
 %6 = icmp sge i32 %5, 2
 br i1 %6, label %7, label %12
```

```
%7:
7:
 %8 = load i32, i32* %2, align 4
 %9 = add nsw i32 %8, -1
 store i32 %9, i32* %2, align 4
 %10 = load i32, i32* %3, align 4
 %11 = mul nsw i32 %10, %8
 store i32 %11, i32* %3, align 4
 br label %4, !llvm.loop !4
```

```
%12:
12:
 %13 = load i32, i32* %3, align 4
 ret i32 %13
```

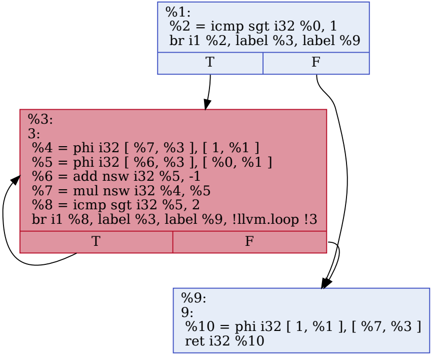
CFG for '_Z9factoriali' function

Example: CFG for Optimized Code

Command

```
clang++ -O1 -S -emit-llvm -o simple_2-opt.ll simple_2.cpp
opt -dot-cfg -cfg-func-name=_Z9factoriali -cfg-dot-filename-prefix \
  dot-simple_2-opt- simple_2-opt.ll > /dev/null
```

CFG



```
%1:
 %2 = icmp sgt i32 %0, 1
 br i1 %2, label %3, label %9
```

```
%3:
3:
 %4 = phi i32 [ %7, %3 ], [ 1, %1 ]
 %5 = phi i32 [ %6, %3 ], [ %0, %1 ]
 %6 = add nsw i32 %5, -1
 %7 = mul nsw i32 %4, %5
 %8 = icmp sgt i32 %5, 2
 br i1 %8, label %3, label %9, !llvm.loop !3
```

```
%9:
9:
 %10 = phi i32 [ 1, %1 ], [ %7, %3 ]
 ret i32 %10
```

CFG for '_Z9factoriali' function

Part 3

LLVM and Clang Libraries

Core LLVM Libraries

- core LLVM libraries provide base-level functionality (i.e., framework) with which frontends, middle-ends, backends, and other tools can be constructed
- written in C++
- code in `llvm` namespace and child namespaces

Comments on LLVM Base Library Functionality

- `llvm::raw_ostream`: fully-buffered output stream (which does not support seeking)
- `llvm::outs`: instance of `llvm::raw_ostream` for normal output from program (similar to `std::cout`)
- `llvm::errs`: instance of `llvm::raw_ostream` for error output from program (similar to `std::cerr`)
- `llvm::StringRef`: immutable non-owning reference to string (similar in spirit to `std::string_view`)
- `llvm::Expected`: tagged union holding either some expected type or error in form of `llvm::Error`
- `llvm::ErrorOr`: holds either error in form of `std::error_code` or some other type

RTTI

- `llvm::isa<>` returns bool indicating if reference/pointer refers to instance of specified class
- `llvm::cast<>` operator converts pointer/reference from base to derived class causing assertion if object not instance of derived type
- `llvm::dyn_cast<>` operator checks if operand of specified type and if so returns pointer to it (with that type); otherwise returns null pointer
- for more details, see:
  - `https://llvm.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates`

Clang Libraries

- Clang is very modular and mostly structured as collection of libraries
- Clang libraries (mostly) written in C++
- for most practical purposes, can simply think of Clang as one large C++ library
- when linking, however, reality is Clang code split across number of libraries
- LibClang provides C library API to access limited subset of Clang functionality

Clang C API: LibClang Library

- C library
- functionality provided includes:
  - parsing source code into AST
  - loading already-parsed ASTs
  - traversing ASTs
  - annotating source locations with elements within AST
- does not provide access to all information in Clang AST
- API intended to be relatively stable from one release to next
- intended to provide only basic functionality needed to support development tools
- data types prefixed with “`CX`” and functions prefixed with “`clang_`”
- access to AST through high-level abstractions
- for details on API, see:
  - `https://clang.llvm.org/doxygen/group__CINDEX.html`

LibTooling

- C++ library
- provides much richer set of functionality relative to LibClang
- Clang library that provides functionality for utilizing parts of Clang in standalone tools or Clang compiler plugins
- provides convenient way to invoke compiler frontend on source code
- provides support for compilation databases
- easily integrates with code using CommandLine Library for processing of command-line arguments
- code in `clang::tooling` namespace
- does not have stable API

Clang Include Directory (1)

- Clang include directory contains numerous low-level headers such as `stddef.h`, `stdint.h`, and `limits.h`
- pathname of Clang include directory depends on how Clang installed, but typically looks something like `$INSTALL_PREFIX/lib/clang/$VERSION/include`, where `$INSTALL_PREFIX` install directory for Clang and `$VERSION` is version of Clang (e.g., `15.0.0`)
- compiler driver programs (i.e., `clang` and `clang++`) configured with header search path that includes Clang include directory
- Clang libraries themselves ***not preconfigured*** with Clang include directory in header search path
- if Clang include directory not included in header search path used by Clang libraries, often ***many compiler errors will arise*** due to missing headers

Clang Include Directory (2)

- Clang libraries assume Clang include directory has pathname `../$lib/clang/$VERSION/include` relative to pathname of running program where `$lib` is typically either `lib` or `lib64`
- this will only result in correct path if program installed in `bin` directory of Clang installation
- typically, user will not place program in this directory so extra directory must be added to header search path when compiling using Clang library (via compiler option)
- programs that employ standard Clang tool command-line interface (via `clang::tooling::CommonOptionsParser`) have `-extra-arg` option that can be used to pass extra options to compiler
- for example, use option `-extra-arg=-I$DIR` to add `$DIR` to header search path

Section 3.1

Command-Line Processing

Command-Line Processing

- LLVM provides CommandLine Library for processing command-line options and arguments
- uses namespace `llvm::cl`
- handles options as well as positional arguments
- option may have arguments
- parses and extracts arguments into variables
- can specify help information for each option as well as general help information
- LLVM CommandLine Library may be more convenient to use than other alternatives such as `getopt` or Boost Program Options Library
- for more details, see:
  - https://llvm.org/docs/CommandLine.html

Command-Line Processing Example: Summary

- in `slides/examples/command_line` directory in companion repository
- processes command-line arguments and prints result obtained
- uses LLVM CommandLine Library and Clang Tooling Library `clang::tooling::CommonOptionsParser`
- several options are supported:
  - `-verbose`
  - `-v`
  - `-o`
  - `-foobar`
- one positional parameter is required
- zero or more additional positional parameters are permitted
- example program output:

```
verbose false
foobar false
operation test
output file (null)
number of compilation database entries: 1
source paths:
    a.cpp
```

Command-Line Processing Example (1)

```
#include <format>
#include <string>
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/raw_ostream.h"

using namespace std::literals;

static llvm::cl::OptionCategory toolOptionCat("Tool Options");
static llvm::cl::extrahelp
  CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
static llvm::cl::extrahelp MoreHelp(
    "This tool does not actually do anything useful.\n"
    "Life is full of disappointments. Get over it.\n"
);
llvm::cl::opt<std::string> outFile(
  "o", llvm::cl::desc("Output file"), llvm::cl::value_desc("output_file"),
  llvm::cl::cat(toolOptionCat));
llvm::cl::opt<bool> verbose("verbose",
  llvm::cl::desc("Enable verbose output."), llvm::cl::cat(toolOptionCat));
llvm::cl::alias verbose2("v", llvm::cl::desc("Alias for -verbose"),
  llvm::cl::aliasopt(verbose));
llvm::cl::opt<bool> foobar("foobar",
  llvm::cl::desc("Enable experimental features."), llvm::cl::Hidden);
llvm::cl::opt<std::string> opName(llvm::cl::Positional, llvm::cl::Required,
  llvm::cl::desc("Operation to perform."),
  llvm::cl::value_desc("op_name"), llvm::cl::cat(toolOptionCat));
```

Command-Line Processing Example (2)

```
int main(int argc, const char **argv) {
    llvm::Expected<clang::tooling::CommonOptionsParser> expectedOptionsParser(
      clang::tooling::CommonOptionsParser::create(argc, argv, toolOptionCat));
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(std::move(expectedOptionsParser.takeError())));
        return 1;
    }
    clang::tooling::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    llvm::outs()
      << std::format("verbose {}\n", verbose)
      << std::format("foobar {}\n", foobar)
      << std::format("operation {}\n",
        !opName.empty() ? opName : "(null)"s)
      << std::format("output file {}\n",
        !outFile.empty() ? outFile : "(null)"s);
    llvm::outs() << std::format("number of compilation database entries: {}\n",
      optionsParser.getCompilations().getAllCompileCommands().size());
    llvm::outs() << "source paths:\n";
    for (auto path : optionsParser.getSourcePathList()) {
        llvm::outs() << std::format("    {}\n", path);
    }
    return 0;
}
```

Section 3.2

Compilation Databases

Compilation Databases

- when running compiler frontend, must specify appropriate options to use for compiler invocations
- **compilation database** specifies which options are used for each source file being built
- Clang provides API for compilation database through base class `clang::tooling::CompilationDatabase`
- several classes derive from this base class in order to provide various kinds of compilation databases
- also provides adapter class that can be used to transform information stored in compilation databases in various ways (e.g., by applying transformations to compiler flags)
- compilation database can have more than one entry for particular source file, since source file could be compiled more than once with different options each time (e.g., non-production and production builds)

CompilationDatabase Class

- clang::tooling::CompilationDatabase base class provides basic API for compilation databases
- some key methods:
  - getCompileCommands: returns all compile commands in which specified source file is compiled
  - getAllFiles: return list of all source files in compilation database
  - getAllCompileCommands: returns all compile commands for all source files in compilation database
- several factory functions for CompilationDatabase objects:
  - loadFromDirectory: load compilation database from build directory (currently, only looks for compile_commands.json)
  - autoDetectFromSource: looks for compilation database in directory containing specified source file and each of its successive parents and loads first one found
  - autoDetectFromDirectory: looks for compilation database in specified directory and each of its successive parents and loads first one found
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1CompilationDatabase.html

JSON Compilation Database

- compilation databases often represented using JavaScript Object Notation (JSON) format
- root JSON object is array
- each array element can have following fields:
  - `directory`: working directory for running compilation command
  - `file`: source file processed
  - `arguments`: compile command argument vector
  - `command`: compile command string (single shell-escaped string)
  - `output`: name of output created
- either `arguments` or `command` is required
- some build tools can generate JSON compilation databases (e.g., CMake)
- for more information, see:
  - https://clang.llvm.org/docs/JSONCompilationDatabase.html

Generating JSON Compilation Database With CMake

- CMake can generate JSON compilation database
- CMake variable `CMAKE_EXPORT_COMPILE_COMMANDS` has boolean value that specifies if compilation database should be generated
- compilation-database file named `compile_commands.json`
- placed in (top-level) binary directory for CMake build
- for example, use command like:

```
cmake -H$src_dir -B$bin_dir -DCMAKE_EXPORT_COMPILE_COMMANDS=1
```

JSON Compilation Database Example

```
[
  {
    "arguments" : [
      "/usr/bin/clang++",
      "-Irelative",
      "-DGREET=Hello, World!\\n",
      "-c",
      "-o",
      "file.o",
      "file.cpp"
    ],
    "directory" : "/home/user/llvm/build",
    "file" : "file.cpp"
  },
  {
    "command" : "/usr/bin/clang++ -Irelative -DGREET=\"Hello, World!\\\\n\" -c -o file.o file.cpp",
    "directory" : "/home/user/llvm/build",
    "file" : "file2.cpp"
  }
]
```

JSONCompilationDatabase Class

- clang::tooling::JSONCompilationDatabase inherits from CompilationDatabase base class
- provides flavor of compilation database that is associated with JSON compilation database
- provides overrides for virtual functions associated with CompilationDatabase appropriate for JSON compilation database
- factory functions:
  - loadFromFile: loads JSON compilation database from specified file
  - loadFromBuffer: loads JSON compilation database from buffer
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1JSONCompilationDatabase.html

JSON Compilation Database Example: Summary

- in slides/examples/compilation_database directory in companion repository
- loads JSON compilation database from specified file
- queries compilation database for each source file specified
- example program output:

```
/home/user/llvm/build/file2.cpp
/home/user/llvm/build/file.cpp
command:
  filename: file.cpp
  directory: /home/user/llvm/build
  command line: /usr/bin/clang++ -Irelative -DGREET=Hello, World!\n -c -o file.o file.
       ↪ cpp
  no output
  no heuristic
command:
  filename: file2.cpp
  directory: /home/user/llvm/build
  command line: /usr/bin/clang++ -Irelative -DGREET=Hello, World!\n -c -o file.o file.
       ↪ cpp
  no output
  no heuristic
```

JSON Compilation Database Example: `json.cpp` (1)

```cpp
#include <format>
#include <utility>
#include "clang/Tooling/CompilationDatabase.h"
#include "clang/Tooling/JSONCompilationDatabase.h"
#include "utility.hpp"

namespace ct = clang::tooling;

int main(int argc, char** argv) {
    std::string errString;
    if (argc < 2) {
        llvm::errs() << "no JSON compilation database specified\n";
        return 1;
    }
    std::string pathname = argv[1];
    std::unique_ptr<ct::CompilationDatabase> compDatabase;
    compDatabase = ct::JSONCompilationDatabase::loadFromFile(pathname,
      errString, ct::JSONCommandLineSyntax::AutoDetect);
    if (!compDatabase) {
        llvm::errs() << std::format("ERROR: {}\n", errString);
        return 1;
    }
    std::vector<std::string> sourcePaths = compDatabase->getAllFiles();
    for (const auto& sourcePath : sourcePaths) {
        llvm::outs() << std::format("{}\n", sourcePath);
    }
```

JSON Compilation Database Example: `json.cpp` (2)

```
    std::vector<ct::CompileCommand> compCommands =
      compDatabase->getAllCompileCommands();
    printCompCommands(llvm::outs(), compCommands);
    for (int i = 2; i < argc; ++i) {
        std::vector<ct::CompileCommand> compCommands =
          compDatabase->getCompileCommands(argv[i]);
        printCompCommands(llvm::outs(), compCommands);
    }
    return 0;
}
```

JSON Compilation Database Example: utility.hpp

```
#include <vector>
#include "llvm/Support/raw_ostream.h"

bool printCompCommands(llvm::raw_fd_ostream& out,
  const std::vector<clang::tooling::CompileCommand>& compCommands);
```

JSON Compilation Database Example: utility.cpp

```cpp
#include <format>
#include <vector>
#include "clang/Tooling/CompilationDatabase.h"
#include "llvm/Support/raw_ostream.h"
#include "utility.hpp"

namespace ct = clang::tooling;

bool printCompCommands(llvm::raw_fd_ostream& out,
  const std::vector<ct::CompileCommand>& compCommands) {
    for (auto compCommand = compCommands.begin();
      compCommand != compCommands.end(); ++compCommand) {
        out << "command:\n"
          << std::format(" filename: {}\n", compCommand->Filename)
          << std::format(" directory: {}\n", compCommand->Directory);
        out << " command line:";
        for (auto word : compCommand->CommandLine) {out << " " << word;}
        out << '\n';
        if (!compCommand->Output.empty()) {
            out << " output: " << compCommand->Output << '\n';
        } else {
            out << " no output\n";
        }
        if (!compCommand->Heuristic.empty()) {
            out << " heuristic: " << compCommand->Heuristic << '\n';
        } else {
            out << " no heuristic\n";
        }
    }
    return !out.has_error();
}
```

Fixed Compilation Database

- Clang supports notion of fixed compilation database
- every source file compiled with identical flags
- command-line arguments specified one per line
- intended for use with very simple projects
- does not identify source files to be built
- only specifies compiler options
- not possible to query source files
- not possible to query all compile commands for all source files

Fixed Compilation Database Example

compile_flags.txt

```
    -DGREETING="Hello, World!"
    -DANSWER=42
    -g
    -O2
    -I
    /usr/local/libfoo/include
```

FixedCompilationDatabase Class

- clang::tooling::FixedCompilationDatabase class inherits from CompilationDatabase base class
- provides flavor of compilation database that is associated with fixed compilation database
- overrides virtual functions from CompilationDatabase base class
- some functions have trivial behavior since fixed database cannot provide full functionality of CompilationDatabase API (e.g., getAllFiles and getAllCompileCommands)
- factory functions:
  - loadFromCommandLine: parses command-line arguments and generates fixed database based on those arguments
  - loadFromFile: reads flags one per line from file
  - loadFromBuffer: reads flags one per line from buffer
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1FixedCompilationDatabase.html

Fixed Compilation Database Example: Summary

- in `slides/examples/compilation_database` directory in companion repository
- loads fixed compilation database from specified file
- queries compilation database for each source file specified
- example program output:

```
command:
  filename: hello.cpp
  directory: ./data
  command line: /home/jdoe/build/clang-tool -DGREETING="Hello, World!" -DANSWER=42 -g
        ↪ -O2 -I /usr/local/libfoo/include hello.cpp
  no output
  no heuristic
command:
  filename: goodbye.cpp
  directory: ./data
  command line: /home/jdoe/build/clang-tool -DGREETING="Hello, World!" -DANSWER=42 -g
        ↪ -O2 -I /usr/local/libfoo/include goodbye.cpp
  no output
  no heuristic
```

Fixed Compilation Database Example: `fixed.cpp` (1)

```
#include <format>
#include <utility>
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/CompilationDatabase.h"
#include "llvm/Config/llvm-config.h"
#include "utility.hpp"

namespace ct = clang::tooling;

int main(int argc, char** argv) {
    if (argc < 2) {
        llvm::errs() << "no fixed database specified\n";
        return 1;
    }
    std::string pathname(argv[1]);
    std::string errString;
    std::unique_ptr<ct::CompilationDatabase> compDatabase;
    compDatabase = ct::FixedCompilationDatabase::loadFromFile(pathname,
      errString);
    if (!compDatabase) {
        llvm::errs() << std::format("ERROR: {}\n", errString);
        return 1;
    }
    std::vector<std::string> sourcePaths = compDatabase->getAllFiles();
    for (const auto& sourcePath : sourcePaths) {
        llvm::outs() << std::format("{}\n", sourcePath);
    }
```

Fixed Compilation Database Example: `fixed.cpp` (2)

```
    std::vector<ct::CompileCommand> compCommands =
      compDatabase->getAllCompileCommands();
    printCompCommands(llvm::outs(), compCommands);
    for (int i = 2; i < argc; ++i) {
        std::vector<ct::CompileCommand> compCommands =
          compDatabase->getCompileCommands(argv[i]);
        printCompCommands(llvm::outs(), compCommands);
    }
    return 0;
}
```

clang::tooling::ArgumentsAdjuster Callable

- clang::tooling::ArgumentsAdjuster is type of callable (i.e., function/functor) that manipulates command-line arguments
- intended to be used with compilation databases to apply transformations to compiler flags
- for more information, see:
  - https://clang.llvm.org/doxygen/namespaceclang_1_1tooling.html#a8dcb3e0419f4f8de952b46ad1c627f68

ArgumentsAdjustingCompilations Class

- clang::tooling::ArgumentsAdjustingCompilations class inherits from CompilationDatabase base class
- ArgumentsAdjustingCompilations provides wrapper around existing compilation database that allow transformations to be applied to compiler flags
- provides method appendArgumentsAdjuster that allows ArgumentsAdjuster to be applied to commands in compilation database
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1ArgumentsAdjustingCompilations.html

Argument Adjuster Example: Summary

- in `slides/examples/compilation_database` directory in companion repository
- loads compilation database from specified source
- optionally applies arguments adjuster to compilation database
- queries compilation database for each source file specified
- example program output:

```
command:
  filename: hello.cpp
  directory: ./data
  command line: /home/jdoe/build/clang-tool -DGREETING="Hello, World!" -DANSWER=42 -g
        ↪ -O2 -I /usr/local/libfoo/include hello.cpp
  no output
  no heuristic
```

Argument Adjuster Example: `adjuster.cpp` (1)

```cpp
#include <format>
#include <utility>
#include <unistd.h>
#include "clang/Tooling/ArgumentsAdjusters.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/CompilationDatabase.h"
#include "clang/Tooling/JSONCompilationDatabase.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Config/llvm-config.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Signals.h"
#include "utility.hpp"

namespace ct = clang::tooling;

int main(int argc, char** argv) {
    int json = -1;
    std::string pathname;
    int adjust = 0;
    for (int c; (c = getopt(argc, argv, "a:j:f:")) >= 0;) {
        switch (c) {
        case 'a':
            adjust = std::atoi(optarg);
            break;
        case 'j':
            pathname = optarg;
            json = 1;
            break;
        case 'f':
            pathname = optarg;
            json = 0;
            break;
        }
    }
    if (json != 0 && json != 1) {
        llvm::errs() << "ERROR: no compilation database specified\n";
        return 1;
    }
```

Argument Adjuster Example: `adjuster.cpp` (2)

```
    std::string errString;
    std::unique_ptr<ct::CompilationDatabase> compDatabase;
    switch (json) {
    case 0:
        compDatabase = ct::FixedCompilationDatabase::loadFromFile(pathname,
          errString);
        break;
    case 1:
        compDatabase = ct::JSONCompilationDatabase::loadFromFile(pathname,
          errString, ct::JSONCommandLineSyntax::AutoDetect);
        break;
    }
    if (!compDatabase) {
        llvm::errs() << std::format("ERROR: {}\n", errString);
        return 1;
    }
    compDatabase = std::make_unique<ct::ArgumentsAdjustingCompilations>(
      std::move(compDatabase));
    auto aac = static_cast<ct::ArgumentsAdjustingCompilations*>(
      compDatabase.get());
    switch (adjust) {
    case 1:
        aac->appendArgumentsAdjuster(ct::getClangSyntaxOnlyAdjuster());
        break;
    case 2:
        aac->appendArgumentsAdjuster(ct::getInsertArgumentAdjuster("-DFOO",
          ct::ArgumentInsertPosition::BEGIN));
        break;
    }
```

Argument Adjuster Example: `adjuster.cpp` (3)

```cpp
    std::vector<std::string> sourcePaths = compDatabase->getAllFiles();
    for (const auto& sourcePath : sourcePaths) {
        llvm::outs() << std::format("{}\n", sourcePath);
    }
    std::vector<ct::CompileCommand> compCommands =
      compDatabase->getAllCompileCommands();
    printCompCommands(llvm::outs(), compCommands);
    for (; optind < argc; ++optind) {
        std::vector<ct::CompileCommand> compCommands =
          compDatabase->getCompileCommands(argv[optind]);
        printCompCommands(llvm::outs(), compCommands);
    }
    return 0;
}
```

Section 3.3

ASTs

ASTs

- four kinds of AST nodes:
  1. `Type`: used to represent type (https://clang.llvm.org/doxygen/classclang_1_1Type.html)
  2. `Decl`: used to represent declaration (https://clang.llvm.org/doxygen/classclang_1_1Decl.html)
  3. `Stmt`: used to represent statement (https://clang.llvm.org/doxygen/classclang_1_1Stmt.html)
  4. `Expr`: used to represent expression (https://clang.llvm.org/doxygen/classclang_1_1Expr.html)
- `Expr` derives from `Stmt` (since expression can be viewed as statement that yields value)
- each of `Type`, `Decl`, `Stmt`, and `Expr` has many subclasses
- AST node types in `clang` namespace (e.g., `clang::Decl` and `clang::Stmt`)
- AST does not have common base node type
- this make tree traversal more difficult

`Type` Nodes

- nodes with type `Type` or types derived therefrom used to represent types
- some examples of types derived from `Type`:
  - □ `BuiltinType`
  - □ `PointerType`
  - □ `ArrayType`
  - □ `RecordType`
  - □ `FunctionType`

Decl Nodes

- Decl type nodes used to represent various kinds of declarations, including declarations of:
  - variables and data members
  - (member and non-member) functions
  - parameters of (member and non-member) functions and templates
  - classes and structs
- different kinds of declarations represented by various subclasses of Decl
- some examples of types derived from Decl include:
  - TranslationUnitDecl: translation unit
  - NamedDecl: declaration that may have name
  - VarDecl: variable declaration
  - ParmVarDecl: declaration of function parameter
  - FunctionDecl: function declaration
  - CxxMethodDecl: declaration of static or non-static method of struct/union/class
  - CxxRecordDecl: declaration of C++ struct/union/class
  - DeclaratorDecl: declaration that uses declarator (e.g., type-and-qualifier name)
  - ValueDecl: declaration for which declared instance can be value

Stmt and Expr Nodes

- most directives in program that correspond to actions can be classified as statements
- statements represented by Stmt class and subclasses thereof
- some examples of subclasses of Stmt class include:
  - CompoundStmt used to represent compound statements
  - IfStmt used to represent **if** statements
  - SwitchStmt used to represent **switch** statements
  - ForStmt used to represent **for** loops
  - WhileStmt used to represent **while** statements
- compound statement represents collection of multiple statements
- expressions are special kinds of statements that generate values (e.g., result of operator, such as function call operator)
- expressions represented by Expr class and subclasses thereof
- some examples of subclasses of Expr class include:
  - DeclRefExpr used to represent reference to symbol
  - CallExpr used to represent function call operator

Section 3.4

Frontend Actions

Frontend Actions

- **frontend action**: task to be performed with help of compiler frontend
- create frontend-action class that embodies particular task to be performed by compiler frontend
- frontend-action class needs to derive from some appropriate class in Clang libraries (namely, subclass of `clang::FrontendAction` to be introduced shortly)
- create frontend-action factory class that can generate frontend action instances
- herein, "frontend-action class" refers to any frontend-action type, whereas "`FrontendAction`" refers to specific type in Clang libraries

Using Frontend Actions

- construct `ClangTool` object, specifying desired compilation database and list of source files to process
- invoke `run` member function of `ClangTool` object to specify frontend-action factory associated with frontend action to be performed
- for each source file to be processed, `run` member function will create compiler instance and use frontend-action factory to create corresponding frontend-action object
- at appropriate points in time, member functions of frontend-action class instance will be invoked to perform various types of processing associated with frontend action

clang::tooling::ClangTool Class

- clang::tooling::ClangTool class provides convenient way to run frontend action over set of source files
- constructed by specifying compilation database and list of source paths
- run method executes frontend action on each source file associated with ClangTool object
- by default modifies command-line arguments (via ClangSyntaxOnlyAdjuster) to syntax-check-only variant (i.e., no generation of IR)
- can supply additional command-line arguments adjusters by using ClangTool::appendArgumentsAdjuster
- can control how diagnostics handled via diagnostics-consumer type by using ClangTool::setDiagnosticConsumer
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1ClangTool.html

clang::FrontendAction Class

- clang::FrontendAction abstract base class provides means to perform user-specified processing in compiler frontend via callbacks
- FrontendAction instance can be thought of as single task running inside compiler frontend
- provides callbacks to be invoked at specific points in processing performed by frontend
- some callbacks provided include:
  - BeginSourceFileAction: invoked just before starting processing of source file
  - EndSourceFileAction: invoked just after finishing processing of source file
  - ExecuteAction: performs main task for frontend action
  - CreateASTConsumer: factory function used to create instance of ASTConsumer, which provides callbacks to be invoked at particular points in processing of AST
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1FrontendAction.html

Frontend Actions

- in order to perform frontend action, library user defines frontend-action class, which derives from appropriate frontend-action class in library
- need to provide override of `CreateASTConsumer`
- this function creates AST consumer for particular translation unit to be processed
- invoked for each translation unit encountered
- implementation of `CreateASTConsumer` constructs AST-consumer object owned by `unique_ptr` and then returns this `unique_ptr`

clang::FrontendActionFactory Class

- clang::tooling::FrontendActionFactory class used to create instances of frontend-action type
- invokes compiler with frontend action
- provides interface for generating frontend action instances
- provides pure virtual function create for obtaining new frontend action instance
- can be used by ClangTool to generate frontend action instance for each translation unit to be processed
- if frontend-action class is default constructible, can use clang::tooling::newFrontendActionFactory function as factory
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1FrontendActionFactory.html

ClangTool Example: Summary

- in slides/examples/frontend_action directory in companion repository
- runs compiler frontend on each source file specified on command line
- performs only syntax checking (i.e., generates AST and then semantically verifies it)
- output resembles something like:

```
/home/jdoe/invalid_1.cpp:3:9: error: use of undeclared identifier 'forty_two'
        return forty_two;
               ^
1 error generated.
Error while processing /home/jdoe/invalid_1.cpp.
error detected
```

ClangTool Example

```
#include "clang/Frontend/FrontendAction.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;

static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << llvm::toString(expectedOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

clang::CompilerInstance Class

- clang::CompilerInstance class used to manage single instance of Clang compiler
- manages various objects which are necessary to run compiler (e.g., preprocessor, target information, and AST context)
- provides utility routines for constructing and manipulating common Clang objects
- associated with FileManager, SourceManager, DiagnosticsEngine, Preprocessor, ASTContext, and ASTConsumer
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1CompilerInstance.html

Section 3.5

Preprocessor-Related Processing

clang::Preprocessor Class

- clang::Preprocessor class works in conjunction with lexer to efficiently preprocess tokens
- custom processing can be performed during preprocessing via callbacks registered via addPPCallbacks member function
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1Preprocessor.html

clang::PPCallbacks Class

- clang::PPCallbacks class provides interface for supplying callbacks to be invoked at various stages during preprocessing
- some events for which callbacks can be registered include:
  - include directive encountered
  - pragma directive encountered
  - macro has been defined
  - macro has been undefined
  - if directive has been encountered
  - else directive has been encountered
  - endif directive has been encountered
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html

Preprocessor Callbacks Example: Summary

- in slides/examples/preprocessor directory in companion repository
- runs preprocessor on each source file specified on command line
- for each preprocessor include directive in main source file, program prints:
  - location of include directive
  - header specified in include directive (with angle brackets or double quotes)
  - full pathname of included file
- output resembles something like:

```
include directive:
    location: /home/jdoe/test_2.cpp:1:1
    header: <cstdint>
    pathname: /usr/lib/gcc/x86_64-redhat-linux/11/../../../../include/c++/11/cstdint
include directive:
    location: /home/jdoe/test_2.cpp:2:1
    header: "test_2.hpp"
    pathname: /home/jdoe/test_2.hpp
```

Preprocessor Callbacks Example: `main.cpp` (1)

```cpp
#include <format>
#include <iostream>
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Lex/PPCallbacks.h"
#include "clang/Lex/Preprocessor.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;
using namespace std::literals;

std::string locationToString(const clang::SourceManager& sourceManager,
  clang::SourceLocation sourceLoc) {
    return std::format("{}:{}:{}", sourceManager.getFilename(sourceLoc),
      sourceManager.getSpellingLineNumber(sourceLoc),
      sourceManager.getSpellingColumnNumber(sourceLoc));
}
```

Preprocessor Callbacks Example: main.cpp (2)

```
class FindIncludes : public clang::PPCallbacks {
public:
    FindIncludes(clang::SourceManager& sourceManager) :
      sourceManager_(&sourceManager) {}
    void InclusionDirective(clang::SourceLocation hashLoc,
      const clang::Token&, llvm::StringRef fileName, bool isAngled,
      clang::CharSourceRange, llvm::Optional<clang::FileEntryRef> file,
      llvm::StringRef, llvm::StringRef, const clang::Module *,
      clang::SrcMgr::CharacteristicKind) override {
        std::string actualFileName;
        if (!sourceManager_->isInMainFile(hashLoc)) {return;}
        if (file) {actualFileName = file->getName();}
        std::string headerName = isAngled ?
          ("<"s + std::string(fileName) + ">"s) :
          ("\""s + std::string(fileName) + "\""s);
        llvm::outs() << std::format("include directive:\n location: {}\n"
          "    header: {}\n  pathname: {}\n",
          locationToString(*sourceManager_, hashLoc), headerName,
          actualFileName);
    }
private:
    clang::SourceManager* sourceManager_;
};
```

Preprocessor Callbacks Example: `main.cpp` (3)

```cpp
class IncludeFinderAction : public clang::PreprocessOnlyAction {
   bool BeginSourceFileAction(clang::CompilerInstance& ci) override {
      std::unique_ptr<FindIncludes> findIncludes(
        new FindIncludes(ci.getSourceManager()));
      clang::Preprocessor& pp = ci.getPreprocessor();
      pp.addPPCallbacks(std::move(findIncludes));
      return true;
   }
};

static llvm::cl::OptionCategory toolCategory("Tool Options");

int main(int argc, char **argv) {
   auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
     const_cast<const char**>(argv), toolCategory);
   if (!expectedOptionsParser) {
      llvm::errs() << llvm::toString(expectedOptionsParser.takeError());
      return 1;
   }
   ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
   ct::ClangTool tool(optionsParser.getCompilations(),
     optionsParser.getSourcePathList());
   return tool.run(
     ct::newFrontendActionFactory<IncludeFinderAction>().get());
}
```

Section 3.6

AST Frontend Actions

clang::ASTFrontendAction Class

- clang::ASTFrontendAction abstract base class provides interface for frontend actions that consume AST
- derives from clang::FrontendAction
- provides override of ExecuteAction function that runs semantic analysis and builds AST
- in some cases, may be desirable to override BeginSourceFileAction and EndSourceFileAction methods
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1ASTFrontendAction.html

clang::ASTConsumer Class

- clang::ASTConsumer class provides interface for consuming AST
- allows code consuming AST to be decoupled from code producing AST
- callbacks for certain types of events can be provided by overriding virtual methods
- some virtual methods include:
  - Initialize: called to perform any initialization of consumer
  - HandleTranslationUnit: called after entire translation unit has been parsed
  - HandleCXXImplicitFunctionInstantiation: called when function implicitly instantiated
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html

AST-Consumer Classes

- AST-consumer classes should be derived from `clang::ASTConsumer`
- provide overrides of functions appropriate for types of processing needed
- for example, common to override `HandleTranslationUnit`

clang::ASTContext Class

- clang::ASTContext class holds long-lived AST nodes (such as types and decls) that can be referred to throughout semantic analysis of file
- some information about AST not stored in AST nodes themselves but rather in AST context (ASTContext) and associated source manager (SourceManager)
- such information includes source locations and global identifier information
- provides getTranslationUnitDecl method for obtaining AST node for translation unit (which often serves as root node for AST traversal)
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1ASTContext.html

AST Consumer Example: Summary

- in slides/examples/ast_consumer_1 directory in companion repository
- runs compiler frontend on each source file specified on command line in order to produce AST
- for each source file processed, prints amount of memory used for AST
- output resembles something like:

```
input file: /home/jdoe/hello.cpp
AST size: 7667712
input file: /home/jdoe/simple_1.cpp
AST size: 8847360
```

AST Consumer Example: main.cpp (1)

```cpp
#include <format>
#include "clang/AST/ASTConsumer.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/raw_ostream.h"

namespace ct = clang::tooling;

class MyAstConsumer : public clang::ASTConsumer {
public:
    MyAstConsumer(const std::string& fileName) : fileName_(fileName) {}
    void HandleTranslationUnit(clang::ASTContext& astContext) override {
        llvm::outs() << std::format("input file: {}\nAST size: {}\n",
          fileName_, astContext.getASTAllocatedMemory());
    }
private:
    std::string fileName_;
};

struct MyAstFrontendAction : public clang::ASTFrontendAction {
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance&, clang::StringRef inFile) override {
        return std::make_unique<MyAstConsumer>(std::string(inFile));
    }
};
```

AST Consumer Example: main.cpp (2)

```
static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(expectedOptionsParser.takeError()));
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<MyAstFrontendAction>().get());
    if (status) {llvm::errs() << "error occurred\n";}
    return !status ? 0 : 1;
}
```

Section 3.7

Traversing the AST With AST Visitors

clang::RecursiveASTVisitor Class Template (1)

- clang::RecursiveASTVisitor class template used for traversing AST and performing specific actions at appropriate nodes
- class template employs CRTP
- templated on AST-visitor class type
- hooks are not virtual (due to use of CRTP)
- be careful to employ correct function signatures for hook methods; otherwise, code will never be called
- can perform preorder or postorder depth-first traversal
- default implementations of hook methods inherited from RecursiveASTVisitor
- only need to implement hook methods for which custom behavior required
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html

`clang::RecursiveASTVisitor` Class Template (2)

- `shouldTraversePostOrder` method:
  - specifies if postorder traversal should be employed (as opposed to preorder)
  - defaults to **`false`** (i.e., preorder)
- `shouldVisitTemplateInstantiations` method:
  - specifies if nodes corresponding to template instantiations should be visited
  - defaults to **`false`** (i.e., not visited)
- nodes corresponding to explicit and partial specializations are visited
- `shouldVisitImplicitCode` method:
  - specifies if nodes corresponding to code implicitly generated by compiler should be visited
  - defaults to **`false`** (i.e., not visited)

Traverse, Walk-Up, and Visit Methods

- traversal functionality provided by three types of methods:
  1. traverse method (named as `Traverse`*Type*): initiates visitation of node and its descendants (e.g., `TraverseDecl`)
  2. walk-up method (`WalkUpFrom`*Type*): dispatches visitation across AST class hierarchy (from node's dynamic type to top-most class) for single node and then call visit method for that node (e.g., `WalkUpFromCXXConstructorDecl`)
  3. visit method (`Visit`*Type*): handles visitation of single node based on its type by calling user-specified function (e.g., `VisitFunctionDecl`, `VisitVarDecl`)
- traverse, walk-up, and visit methods have **`bool`** return type which indicates if traversal should continue

Traverse Methods

- traverse method invoked when node being traversed
- for traverse methods, hooks of following form provided for most AST nodes of type *NodeType*:
  - **`bool`** `Traverse`*NodeType*`(`*NodeType* `*)`
- returning false terminates traversal early
- by default, `Traverse`*Type* invokes `WalkUpFrom`*Type* to visit node for direct base class of *Type* (and ultimately each of its other base classes in inheritance hierarchy)
- for node of type `T`, `Traverse`*Type* method invoked only if `T` is same as *Type*
- traverse methods can call traverse and walk-up methods, but not visit methods

Walk-Up Methods

- walk-up method invoked as part of process of climbing inheritance hierarchy (which visits node at each level of inheritance hierarchy)
- for walk-up methods, hooks of following form provided for most AST nodes of type *NodeType*:
  - **`bool`** `WalkUpFrom`*NodeType*`(`*NodeType* `*)`
- returning false terminates traversal early
- by default, `WalkUpFrom`*Type* invokes `WalkUpFrom`*ParentType* (where *ParentType* is direct base class of *Type*) and then invokes `Visit`*Type*
- since `WalkUpFrom`*ParentType* called before `Visit`*Type*, inheritance hierarchy is visited in top-down order (i.e., from least to most derived)
- for node of type `T`, `WalkUpFrom`*Type* method will be called if `T` same as *Type* or `T` is (directly or indirectly) derived from *Type*
- walk-up methods can call walk-up and visit methods, but not traverse methods

Visit Methods

- visit method invoked to visit node at particular level in inheritance hierarchy
- in terms of visit methods, hooks of following form provided for most AST nodes of type *NodeType* (exception being `TypeLoc` nodes, which are passed by value):
  - `bool Visit`*NodeType*`(`*NodeType*` *)`
- returning false terminates traversal early
- all calls to visit methods for same node grouped together (i.e., not interleaved with calls to visit methods for other nodes)
- by default, `Visit`*Type* is no-op
- visit methods can call visit methods, but not traverse or walk-up methods
- for node of type `T`, `Visit`*Type* method will be called if `T` same as *Type* or `T` is (directly or indirectly) derived from *Type*

Node Handling Example

- consider AST node of type `NamespaceDecl`
- inheritance hierarchy involved:
  - `Decl` ⟵ `NamedDecl` ⟵ `NamespaceDecl`
- when node of type `NamespaceDecl` encountered during AST traversal, callbacks (i.e., traverse, walk-up, and visit methods) invoked in following order:
  1. `TraverseNamespaceDecl`
  2. `WalkUpFromNamespaceDecl`
  3. `WalkUpFromNamedDecl`
  4. `WalkUpFromDecl`
  5. `VisitDecl`
  6. `VisitNamedDecl`
  7. `VisitNamespaceDecl`

AST Visitor Example: Summary

- in slides/examples/ast_visitor_1 directory in companion repository
- runs compiler frontend on each source file specified on command line in order to produce AST
- traverses AST, printing fully-qualified name of each function declared
- only considers function declarations in main source file (not those in headers)
- output resembles something like:

```
main
foo::max
foo::abs
get_values
```

AST Visitor Example: main.cpp (1)

```
#include <format>
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Config/llvm-config.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;

class MyAstVisitor : public clang::RecursiveASTVisitor<MyAstVisitor> {
public:
    MyAstVisitor(clang::ASTContext& astContext) : astContext_(&astContext) {}
    bool VisitFunctionDecl(clang::FunctionDecl* funcDecl) {
        const auto& fileId = astContext_->getSourceManager().getFileID(
          funcDecl->getLocation());
        if (fileId == astContext_->getSourceManager().getMainFileID()) {
            llvm::outs() << std::format("{}\n",
              funcDecl->getQualifiedNameAsString());
        }
        return true;
    }
private:
    clang::ASTContext* astContext_;
};
```

AST Visitor Example: main.cpp (2)

```
class MyAstConsumer : public clang::ASTConsumer {
public:
    void HandleTranslationUnit(clang::ASTContext& astContext) final {
        clang::TranslationUnitDecl* tuDecl =
          astContext.getTranslationUnitDecl();
        MyAstVisitor visitor(astContext);
        visitor.TraverseDecl(tuDecl);
    }
};

class MyFrontendAction : public clang::ASTFrontendAction {
public:
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance&, clang::StringRef) final {
        return std::unique_ptr<clang::ASTConsumer>{new MyAstConsumer};
    }
};
```

AST Visitor Example: `main.cpp` (3)

```cpp
static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(expectedOptionsParser.takeError()));
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<MyFrontendAction>().get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

Class Hierarchy Example: Summary

- in slides/examples/ast_visitor_3 directory in companion repository
- runs compiler frontend on each source file specified on command line in order to produce AST
- traverses AST for each translation unit
- maintains stack to track nesting hierarchy of struct/union declarations
- for each struct/union declaration, prints corresponding class hierarchy
- only considers declarations in main source file (not those in headers)
- output resembles something like:

```
A1 -> A2 -> A3
A1 -> A2
A1
Something -> Wazzit
Something -> (anonymous)
Something
```

Class Hierarchy Example: `main.cpp` (1)

```
#include <format>
#include <vector>
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;

class MyAstVisitor : public clang::RecursiveASTVisitor<MyAstVisitor> {
public:
    MyAstVisitor(clang::ASTContext& astContext) : astContext_(&astContext),
      stack_() {}
    bool TraverseCXXRecordDecl(clang::CXXRecordDecl* recDecl);
private:
    using Base = clang::RecursiveASTVisitor<MyAstVisitor>;
    void printStack() const;
    clang::ASTContext* astContext_;
    std::vector<const clang::CXXRecordDecl*> stack_;
};
```

Class Hierarchy Example: main.cpp (2)

```
bool MyAstVisitor::TraverseCXXRecordDecl(clang::CXXRecordDecl* recDecl) {
   clang::SourceManager& sourceManager = astContext_->getSourceManager();
   stack_.push_back(recDecl);
   bool result = Base::TraverseCXXRecordDecl(recDecl);
   if (sourceManager.getFileID(recDecl->getLocation()) ==
     sourceManager.getMainFileID()) {printStack();}
   stack_.pop_back();
   return result;
}

void MyAstVisitor::printStack() const {
   std::string s;
   for (auto i = stack_.begin(); i != stack_.end(); ++i) {
      std::string name((*i)->getName());
      s += std::format("{}{}", i != stack_.begin() ? " -> " : "",
        name.size() ? name : "(anonymous)");
   }
   llvm::outs() << s << ’\n’;
}

class MyAstConsumer : public clang::ASTConsumer {
public:
   void HandleTranslationUnit(clang::ASTContext& astContext) final {
      clang::TranslationUnitDecl* tuDecl =
        astContext.getTranslationUnitDecl();
      MyAstVisitor astVisitor(astContext);
      astVisitor.TraverseDecl(tuDecl);
   }
};
```

Class Hierarchy Example: main.cpp (3)

```
class MyFrontendAction : public clang::ASTFrontendAction {
public:
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance& compInstance, clang::StringRef) final {
        return std::unique_ptr<clang::ASTConsumer>{new MyAstConsumer};
    }
};

static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(expectedOptionsParser.takeError()));
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<MyFrontendAction>().get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

Section 3.8

Source Manager and Source Locations

clang::SourceManager Class

- clang::SourceManager manages all source files stored in memory and provides interface to access them
- provides APIs to deal with SourceLocation instances
- for more information, see:
    - □ https://clang.llvm.org/doxygen/classclang_1_1SourceManager.html

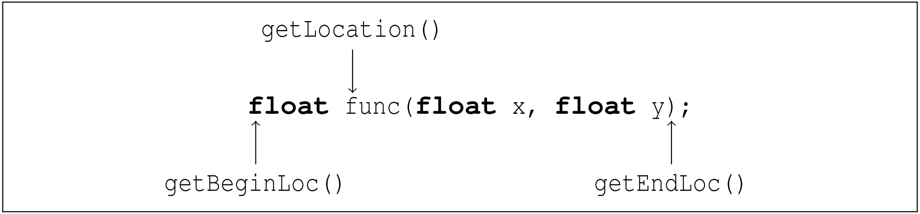
clang::SourceLocation Type

- clang::SourceLocation represents location of specific position in source code
- SourceLocation capable of representing position in source code with character granularity but often used to refer to tokens (by referring to location of first character of token)
- SourceLocation made very lightweight for efficiency
- just reference/handle to specific part of source code
- underlying data being referenced stored in SourceManager instance
- SourceLocation has special invalid value that can used to indicate no corresponding location in source code exists
- can check for invalid value with isValid member function
- can use SourceManager to query file name, line number, column number associated with SourceLocation
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1SourceLocation.html

clang::SourceRange Type

- clang::SourceRange represents contiguous part of source code
- SourceRange represents range of tokens
- essentially pair of SourceLocation objects (i.e., one SourceLocation object for each of begin and end locations)
- range is symmetric (i.e., both begin and end refer to elements in range)
- begin location specifies location of first character of first token in range (obtained via getBegin)
- end location specifies location of first character of last token in range (obtained via getEnd)
- can check for invalid value with isValid member function
- some AST node types have getSourceRange member function to obtain range of tokens related to AST node (e.g., FunctionDecl and VarDecl)
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1SourceRange.html

FunctionDecl and Source Locations (Declaration Only)

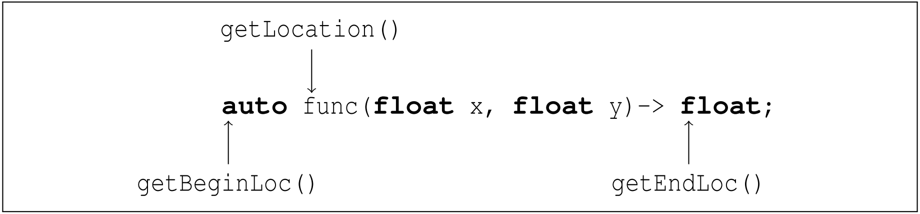


getLocation()

```
float func(float x, float y);
```

getBeginLoc()

getEndLoc()



getLocation()

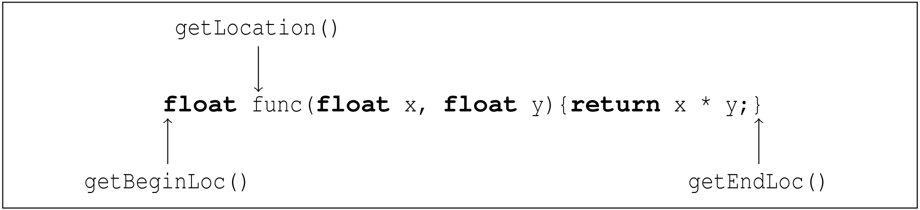
```
auto func(float x, float y)-> float;
```

getBeginLoc()

getEndLoc()

- source range: functionDecl->getBeginLoc(), functionDecl->getEndLoc()
- source location: functionDecl->getLocation()

FunctionDecl and Source Locations (Definition)

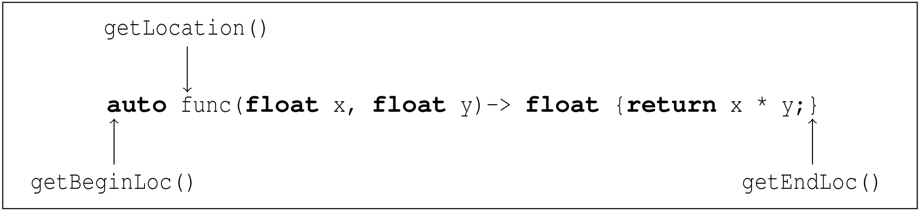


getLocation()

```
float func(float x, float y){return x * y;}
```

getBeginLoc()

getEndLoc()



getLocation()

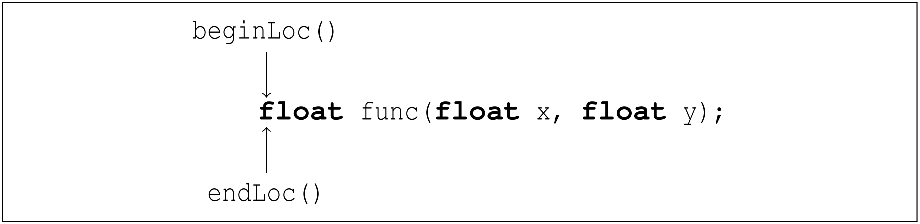
```
auto func(float x, float y)-> float {return x * y;}
```

getBeginLoc()

getEndLoc()

- source range: `functionDecl->getBeginLoc()`, `functionDecl->getEndLoc()`
- source location: `functionDecl->getLocation()`

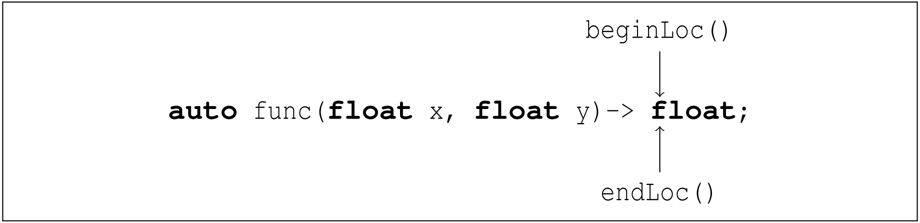
FunctionDecl::getReturnTypeSourceRange



beginLoc()

```
float func(float x, float y);
```

endLoc()



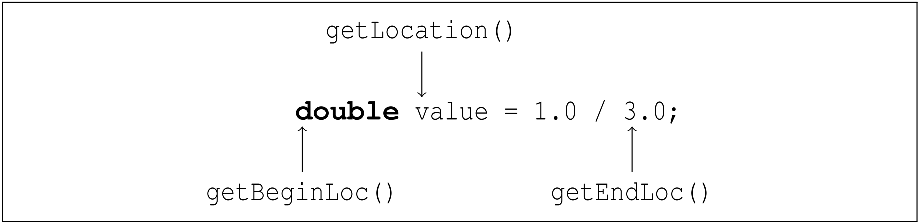
beginLoc()

```
auto func(float x, float y)-> float;
```

endLoc()

- source range:
  `functionDecl->getReturnTypeSourceRange().beginLoc()`,
  `functionDecl->getReturnTypeSourceRange().endLoc()`

VarDecl and Source Locations

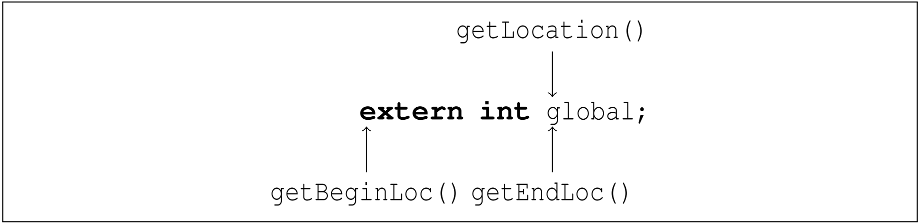


getLocation()

```
double value = 1.0 / 3.0;
```

getBeginLoc()

getEndLoc()



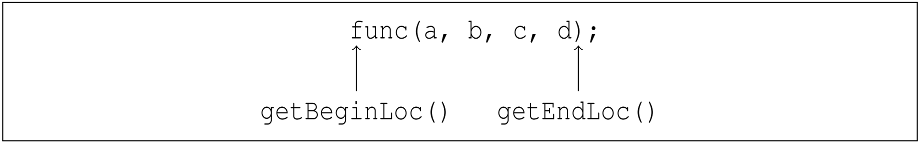
getLocation()

```
extern int global;
```

getBeginLoc() getEndLoc()

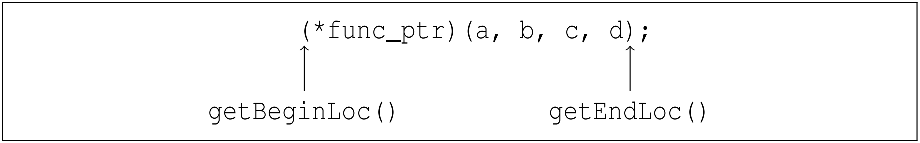
- source range: `varDecl->getBeginLoc()`, `varDecl->getEndLoc()`
- source location: `varDecl->getLocation()`

`CallExpr` and Source Locations



`func(a, b, c, d);`

`getBeginLoc()` `getEndLoc()`



`(*func_ptr)(a, b, c, d);`

`getBeginLoc()` `getEndLoc()`

- source range: `callExpr->getBeginLoc()`, `callExpr->getEndLoc()`

clang::CharSourceRange Type

- clang::CharSourceRange represents contiguous part of source code with character granularity
- CharSourceRange can be used to specify:
  - range of characters
  - range of tokens
- for range of characters:
  - begin and end specify location of first and last characters of range
- for range of tokens:
  - begin specifies location of first character of first token in range
  - end specifies location of first character of last token in range
- can determine type of range (i.e., character versus token) via isCharRange and isTokenRange member functions
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1CharSourceRange.html

clang::FullSourceLocation Type

- SourceLocation instance not useful in isolation, since references information in SourceManager instance
- FullSourceLocation is SourceLocation along with associated SourceManager

Spelling Versus Expansion Locations

- each source location associated with spelling location and expansion location
- spelling location specifies where characters corresponding to token originated
- expansion location specifies where characters for token appear to be from user's point of view
- in case of macro expansion:
  - spelling location specifies where token in macro expansion originated (i.e., in macro definition)
  - expansion location specifies where macro expansion took place (i.e., point where macro invoked)
- `clang::SourceManager` class provides methods for querying information about source locations, including:
  - `getFilename`
  - `getSpellingLineNumber`
  - `getSpellingColumnNumber`
  - `getExpansionRange`

Example: Spellings Versus Expansions

Source File `example_16.cpp`

```
            11111111112222222
   12345678901234567890123456
1  #define␣foo1(x)␣foo(x)
2  int␣foo(int␣x)␣{return␣x;}
3  int␣main()␣{
4  ␣return␣foo1(42);
5  }
```

Preprocessor Output

```
            11111111112222222
   12345678901234567890123456
2  int␣foo(int␣x)␣{return␣x;}
3  int␣main()␣{
4  ␣return␣foo(42);
5  }
```

- consider `CallExpr` AST node associated with call to `foo` on line 4 of source file
- begin spelling location: `example_16.cpp:1:17`
- end spelling location: `example_16.cpp:1:22`
- expansion range: `example_16.cpp:4:9` to `example_16.cpp:4:16`

Example: More Macro Strangeness

Source File `example_17.cpp`

```
            11111111112222222
   12345678901234567890123456
1  #define␣FORTY_TWO␣42
2  #define␣X␣x
3  #define␣INT␣int
4  INT␣X␣=␣FORTY_TWO;
```

Preprocessor Output

```
            11111111112222222
   12345678901234567890123456
4  int␣x␣=␣42;
```

- consider `VarDecl` AST node associated with declaration of variable `x` on line 4 of source file
- begin spelling location: `example_17.cpp:3:13`
- end spelling location: `example_17.cpp:1:19`
- note that end spelling location precedes begin spelling location in source file
- expansion range: `example_17.cpp:4:1` to `example_17.cpp:4:9` (i.e., first character of token `FORTY_TWO`)

Obtaining Source Code for Source Range

- to obtain text of source corresponding to source range, use `getSourceText` member of `clang::Lexer` class
- source range specified as `CharSourceRange` instance
- allows source range to be token or character range
- if token range, gets source text for all tokens covered by range
- if character range, gets source text for all characters covered by range
- may fail in cases where source range contains macro expansions
- failure indicator returned via `bool*` parameter

Source Location Example: Summary

- in slides/examples/clang_utilities directory in companion repository
- provides several functions for printing information about source code from SourceLocation and SourceRange objects:
  - associated source file, line number, and column number
  - corresponding character sequence in source code

Source Location Example: `utilities.cpp` (1)

```
#include <format>

#include "clang/Basic/SourceManager.h"
#include "clang/Basic/SourceLocation.h"
#include "clang/Lex/Lexer.h"

std::string locationToString(const clang::SourceManager& sourceManager,
  clang::SourceLocation sourceLoc) {
    return std::format("{}:{}({})", sourceManager.getFilename(sourceLoc),
      sourceManager.getSpellingLineNumber(sourceLoc),
      sourceManager.getSpellingColumnNumber(sourceLoc));
}

std::string rangeToString(const clang::SourceManager& sourceManager,
  clang::SourceRange sourceRange) {
    std::string beginFilename(sourceManager.getFilename(
      sourceRange.getBegin()));
    std::string endFilename(sourceManager.getFilename(sourceRange.getEnd()));
    return std::format("{}:{}({})-{}{}({})", beginFilename,
      sourceManager.getSpellingLineNumber(sourceRange.getBegin()),
      sourceManager.getSpellingColumnNumber(sourceRange.getBegin()),
      endFilename != beginFilename ? endFilename + ":" : "",
      sourceManager.getSpellingLineNumber(sourceRange.getEnd()),
      sourceManager.getSpellingColumnNumber(sourceRange.getEnd()));
}
```

Source Location Example: `utilities.cpp` (2)

```
std::string getSourceText(const clang::SourceManager& sourceManager,
  clang::SourceRange range) {
    return std::string(clang::Lexer::getSourceText(
      clang::CharSourceRange::getTokenRange(range), sourceManager,
      clang::LangOptions()));
}

std::string addLineNumbers(const std::string& source, unsigned int start) {
    std::string result;
    result += std::format("{:4d}: ", start);
    for (auto c : source) {
        if (c == ’\n’) {
            ++start;
            result += std::format("\n{:4d}: ", start);
        } else {result += c;}
    }
    return result;
}
```

Source Printing Example: Summary

- in slides/examples/ast_visitor_2 directory in companion repository
- for each function definition in main source file, prints:
  - fully-qualified name of function
  - source file containing function definition
  - starting and ending of line number and column number of function definition
  - lines of source code comprising function definition (with line numbers)
- output for function definition resembles something like:

```
h2g2::get_status
/home/jdoe/example_1.cpp:3(1)-5(1)
----------
   3: int get_status() {
   4:   return 42;
   5: }
----------

main
/home/jdoe/example_1.cpp:9(1)-11(1)
----------
   9: int main() {
  10:   return h2g2::get_status();
  11: }
----------
```

Source Printing Example: `main.cpp` (1)

```cpp
#include <format>
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "utilities.hpp" // header for utilities.cpp

namespace ct = clang::tooling;

class MyAstVisitor : public clang::RecursiveASTVisitor<MyAstVisitor> {
public:
    MyAstVisitor(clang::ASTContext& astContext) : astContext_(&astContext) {}
    bool VisitFunctionDecl(clang::FunctionDecl* funcDecl) {
        clang::SourceManager& sm = astContext_->getSourceManager();
        const auto& fileId = sm.getFileID(funcDecl->getLocation());
        if (funcDecl->hasBody() && fileId == sm.getMainFileID()) {
            clang::SourceRange sourceRange = funcDecl->getSourceRange();
            std::string delim("----------\n");
            llvm::outs() << std::format("{}\n{}\n{}{}\n{}\n",
              funcDecl->getQualifiedNameAsString(), rangeToString(sm,
              sourceRange), delim, addLineNumbers(getSourceText(sm,
              sourceRange), sm.getSpellingLineNumber(sourceRange.getBegin())),
              delim);
        }
        return true;
    }
private:
    clang::ASTContext* astContext_;
};
```

Source Printing Example: main.cpp (2)

```
class MyAstConsumer : public clang::ASTConsumer {
public:
    void HandleTranslationUnit(clang::ASTContext& astContext) final {
        clang::TranslationUnitDecl* tuDecl =
          astContext.getTranslationUnitDecl();
        MyAstVisitor astVisitor(astContext);
        astVisitor.TraverseDecl(tuDecl);
    }
};

class MyFrontendAction : public clang::ASTFrontendAction {
public:
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance& compInstance, clang::StringRef) final {
        return std::unique_ptr<clang::ASTConsumer>{new MyAstConsumer};
    }
};

static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(expectedOptionsParser.takeError()));
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<MyFrontendAction>().get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

Section 3.9

Diagnostics

clang::DiagnosticConsumer Class

- clang::DiagnosticConsumer class provides interface for receiving diagnostic information (i.e., warnings, errors, and so on) from compiler frontend
- class provides numerous virtual functions that serve as callbacks
- some classes derived from DiagnosticConsumer provided by library to cover some common use cases (e.g., clang::IgnoringDiagConsumer)
- can also create other diagnostic-consumer classes by deriving from DiagnosticConsumer and providing desired overrides
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1DiagnosticConsumer.html

Diagnostic Consumer Example: Summary

- in slides/examples/diagnostic_consumer directory in companion repository
- runs compiler frontend on specified source files
- for each diagnostic with severity level of (regular) error or fatal error, program prints filename and line/column number associated with diagnostic
- other diagnostic information discarded
- upon completion, program prints count of number of regular/fatal errors
- output resembles something like:

```
error at /home/mdadams/jdoe/invalid_1.cpp:3:9
1 error(s) occurred
```

Diagnostic Consumer Example: `main.cpp` (1)

```
#include <format>
#include <map>
#include <string>
#include "clang/Basic/Diagnostic.h"
#include "clang/Basic/SourceLocation.h"
#include "clang/Basic/SourceManager.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;

std::string locationToString(const clang::SourceManager& sourceManager,
  clang::SourceLocation sourceLoc) {
    return std::format("{}:{}:{}", sourceManager.getFilename(sourceLoc),
      sourceManager.getSpellingLineNumber(sourceLoc),
      sourceManager.getSpellingColumnNumber(sourceLoc));
}

std::string levelToString(clang::DiagnosticsEngine::Level level) {
    const std::map<clang::DiagnosticsEngine::Level, std::string> lut{
      {clang::DiagnosticsEngine::Level::Error, "error"},
      {clang::DiagnosticsEngine::Level::Fatal, "fatal error"},
    };
    auto i = lut.find(level);
    return i != lut.end() ? i->second : "unknown";
}
```

Diagnostic Consumer Example: `main.cpp` (2)

```cpp
class MyDiagnosticConsumer : public clang::DiagnosticConsumer {
public:
    MyDiagnosticConsumer() : errCount_(0) {}
    void HandleDiagnostic(clang::DiagnosticsEngine::Level diagLevel,
      const clang::Diagnostic& info) override {
        clang::SourceManager* sm = info.hasSourceManager() ?
          &info.getSourceManager() : nullptr;
        if (diagLevel == clang::DiagnosticsEngine::Level::Error ||
          diagLevel == clang::DiagnosticsEngine::Level::Fatal) {
            if (sm) {
                llvm::errs() << std::format("{} at {}\n",
                  levelToString(diagLevel), locationToString(*sm,
                  info.getLocation()));
                ++errCount_;
            } else {
                llvm::errs() << std::format("{}\n", levelToString(diagLevel));
            }
        }
    }
    unsigned long getErrCount() const {return errCount_;}
private:
    unsigned long errCount_;
};
```

Diagnostic Consumer Example: main.cpp (3)

```cpp
static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << llvm::toString(expectedOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    MyDiagnosticConsumer diagnosticConsumer;
    tool.setDiagnosticConsumer(&diagnosticConsumer);
    int status = tool.run(
      ct::newFrontendActionFactory<clang::SyntaxOnlyAction>().get());
    unsigned long errCount = diagnosticConsumer.getErrCount();
    if (errCount) {
        llvm::errs() << std::format("{} error(s) occurred\n", errCount);
    }
    return (!status && !errCount) ? 0 : 1;
}
```

Section 3.10

Finding AST Nodes With AST Matchers

AST Matchers

- Clang libraries provide mechanism for finding AST nodes that match specific criteria as determined by some matching predicate
- predicate embodied by AST matcher type
- can match nodes that correspond to declarations, statements, expressions, and types (amongst other things)
- AST matcher API designed such that expressions involving matchers have very natural syntax
- this syntax can be thought of as domain-specific language for AST node matching
- as will be seen later, `clang-query` tool supports similar syntax

Matcher Classes

- numerous matcher types provided (in `clang::ast_matchers` namespace) to allow matching various types of AST nodes

Matcher Type	Type of Node Matched
`DeclarationMatcher`	`Decl`
`StatementMatcher`	`Stmt` (which includes `Expr`)
`TypeMatcher`	`QualType`
`TypeLocMatcher`	`TypeLoc`
`NestedNameSpecifierMatcher`	`NestedNameSpecifier`
`NestedNameSpecifierLocMatcher`	`NestedNameSpecifierLoc`
`CXXBaseSpecifierMatcher`	`CXXBaseSpecifier`
`CXXCtorInitializerMatcher`	`CXXCtorInitializer`
`TemplateArgumentMatcher`	`TemplateArgument`
`TemplateArgumentLocMatcher`	`TemplateArgumentLoc`
`LambdaCaptureMatcher`	`LambdaCapture`
`AttrMatcher`	`Attr`

Trivial Matcher Example

```
#include <string>
#include "clang/ASTMatchers/ASTMatchers.h"

namespace cam = clang::ast_matchers;

cam::DeclarationMatcher matchFuncDef() {
    using namespace cam;
    return functionDecl(isDefinition()).bind("func");
}

cam::DeclarationMatcher matchFuncDeclOf(const std::string& funcName) {
    using namespace cam;
    return functionDecl(hasName(funcName)).bind("func");
}

cam::StatementMatcher matchCallTo(const std::string& funcName) {
    using namespace cam;
    return callExpr(callee(
      functionDecl(hasName(funcName)).bind("func"))).bind("call");
}

cam::TypeMatcher matchPointerType() {
    using namespace cam;
    return qualType(isAnyPointer());
}
```

Section 3.10.1

AST Matchers

AST Matchers

- **AST matcher** is class that holds predicate used to test for match
- three categories of AST matchers:
  1. node matchers: match specific type of AST node
  2. narrowing matchers: match attributes on AST nodes
  3. traversal matchers: allow traversal between AST nodes
- library provides very rich set of predefined AST matchers
- predefined AST matchers in `clang::ast_matchers` namespace
- library also allows custom AST matchers to be defined by user
- for list of AST matchers provided by library, see:
  - `https://clang.llvm.org/docs/LibASTMatchersReference.html`

  [click on name of matcher for more detailed information on that matcher]

Node Matchers

- **node matchers** allow nodes to be matched on basis of their type
- every matcher expression must start with node matcher (or `traverse` matcher to be discussed shortly)
- match expression can be further refined with narrowing or traversal matchers
- all node matchers take arbitrary number of matchers as arguments and perform logical AND of all of these matchers
- matcher expression that matches every `FunctionDecl` node in AST:

  ```
  functionDecl()
  ```
- node matchers support bind operation that allows matched node to be associated with string, which can be later used by callback to gain access to that node
- matcher expression that matches every `FunctionDecl` node and binds matched node to name `"x"`:

  ```
  functionDecl().bind("x")
  ```

Some Predefined Node Matchers

Name	Description
`functionDecl`	matches `FunctionDecl` node (function declaration)
`cxxMethodDecl`	matches `CXXMethodDecl` node (class/union/struct method declaration)
`cxxRecordDecl`	matches `CXXRecordDecl` node (C++ class/union/struct declaration)
`varDecl`	matches `VarDecl` node (variable declaration)
`callExpr`	matches `CallExpr` node (call expression)
`declRefExpr`	matches `DeclRefExpr` node (expression referring to declared entity)

Narrowing Matchers

- **narrowing matchers** match certain attributes on current node
- narrowing matchers allow number of matches to be reduced by only keeping matches with specific attributes
- special logical narrowing matchers provide AND, OR, and NOT logical operations (i.e., `allOf`, `anyOf`, `unless`)
- matcher expression that matches node corresponding to declaration of function whose name is `foo`:

  ```
  functionDecl(hasName("foo"))
  ```
- matcher expression that matches node corresponding to declaration of function whose name is either `foo` or `bar`:

  ```
  functionDecl(anyOf(hasName("foo"), hasName("bar")))
  ```
- matcher expression that matches node corresponding to declaration of function whose name is neither `foo` nor `bar`:

  ```
  functionDecl(unless(anyOf(hasName("foo"), hasName("bar"))))
  ```

Some Predefined Narrowing Matchers

Logical Narrowing Matchers

Name	Description
`allOf`	logical AND
`anyOf`	logical OR
`unless`	logical NOT

Miscellaneous Narrowing Matchers

Name	Description
`hasName`	matches if has specified name
`matchesName`	matches if name matches regular expression
`isExpansionInMainFile`	matches if from main source file (i.e., not included header)
`isImplicit`	matches if implicitly generated by compiler (e.g., implicit default constructor)
`equalsBoundNode`	matches if same as node bound to specified name

Traversal Matchers

- **traversal matchers** specify relationship between current node and other nodes reachable from current node
- special `traverse` matcher allows control over how traversal performed (e.g., all implicit nodes can be ignored)
- matcher expression that matches declaration of function that contains at least one **`if`** statement:

  ```
  functionDecl(hasDescendant(ifStmt()))
  ```

- matcher expression that matches call to function whose name is `foo`:

  ```
  callExpr(callee(functionDecl(hasName("foo"))))
  ```

Some Predefined Traversal Matchers

Parent/Child Traversal Matchers

Name	Description
`hasParent`	parent matches specified matcher
`has`	child matches specified matcher
`hasAncestor`	at least one ancestor matches specified matcher
`hasDescendant`	at least one descendant matches specified matcher

Expr to Decl Traversal

Expr	Matcher	Decl
`CallExpr`	`callee`	`FunctionDecl`
`DeclRefExpr`	`to`	`VarDecl`
`MemberExpr`	`member`	`FieldDecl`

Ignoring Implicit AST Nodes

- compiler sometimes generates AST nodes that do not correspond to constructs explicitly spelled in source code
- such nodes are referred to as implicit nodes
- often implicit nodes can complicate AST matching process (by creating many additional special cases to handle during matching)
- may wish to ignore all implicit nodes
- can be accomplished by using special `traverse` matcher with `clang::TK_IgnoreUnlessSpelledInSource` for first argument
- that is, all implicit AST nodes can be ignored by wrapping matcher expression *e* with `traverse` as follows:

```
clang::ast_matchers::traverse(
  clang::TK_IgnoreUnlessSpelledInSource, e)
```

Examples of AST Matcher Expressions (1)

- match node that corresponds to function declaration in main source file:

  ```
  functionDecl(isExpansionInMainFile())
  ```

- match node that corresponds to inline function declaration:

  ```
  functionDecl(isInline())
  ```

- match node that corresponds to call to function whose name matches regular expression “`^::foo_`”:

  ```
  functionDecl(matchesName("^::foo_"))
  ```

- match node that corresponds to function declaration whose 0th parameter has name `x`:

  ```
  functionDecl(hasParameter(0, parmVarDecl(hasName("x"))))
  ```

Examples of AST Matcher Expressions (2)

- match statement (i.e., `Stmt`) node that corresponds to either normal or range-based **`for`** statement, and bind name "**`for`**" to matching `Stmt` node:

```
stmt(anyOf(forStmt(), cxxForRangeStmt())).bind("for")
```

- match **`if`**-statement (i.e., `IfStmt`) node that is not part of **`else if`**, and bind name "**`if`**" to matched node:

```
ifStmt(stmt().bind("if"), unless(hasParent(ifStmt(hasElse(
  ifStmt(equalsBoundNode("if")))))))
```

- match node corresponding to **`if`**-statement (i.e., `IfStmt` node) whose then or else clause is not compound statement:

```
ifStmt(unless(allOf(hasThen(compoundStmt()), anyOf(unless(
  hasElse(anything())), hasElse(compoundStmt())))))
```

Examples of AST Matcher Expressions (3)

- match call to `make_unique` with template argument `Widget`:

  ```
  callExpr(callee(functionDecl(hasName("make_unique"),
    hasAnyTemplateArgument(refersToType(hasDeclaration(
    namedDecl(hasName("Widget"))))))))
  ```

- match call to function where function has any parameters of rvalue reference type:

  ```
  callExpr(callee(functionDecl(hasAnyParameter(parmVarDecl(
    hasType(rValueReferenceType()))))))
  ```

- match overriding virtual method that does not use **`override`** or **`final`** keyword (where `attr::Override` and `attr::Final` from `clang` namespace):

  ```
  cxxMethodDecl(unless(cxxDestructorDecl()), isOverride(),
    unless(anyOf(hasAttr(attr::Override),
    hasAttr(attr::Final))))
  ```

Section 3.10.2

Using clang-query to Facilitate AST Matcher Development

`clang-query` Program (1)

- `clang-query` is program that facilitates easier development of AST matchers
- reads sequence of commands as input that can be used to find nodes in AST corresponding to given source code using AST matchers
- source files to be considered specified as command-line arguments
- commands for specifying AST matchers and controlling how results generated and presented
- program reads commands from standard input and query results and diagnostics sent to standard output/error
- syntax for specifying matchers mostly compatible with syntax that would be used in C++ source code
- only matchers provided by Clang library supported (i.e., no support for custom matchers)

clang-query Program (2)

- ASTs for source files only generated at program startup (so any subsequent changes to source files not considered)
- source code can be found in clang-tools-extra/clang-query directory of LLVM Git repository
- completion functionality requires LLVM/Clang built to use Editline library (a.k.a. libedit)?
- for more information, see:
  - https://firefox-source-docs.mozilla.org/code-quality/static-analysis/writing-new/clang-query.html

clang-query Example

example_1.cpp

```
#include <iostream>
int square(int x) {return x * x;}
int cube(int x) {return x * x * x;}
int main() {
    std::cout << square(2) + square(3) << '\n';
    std::cout << cube(42) << '\n';
}
```

Command Line Invocation of clang-query Program

```
clang-query example_1.cpp
```

Input to clang-query Program

```
set bind-root false
m callExpr(callee(functionDecl(hasName("square")))).bind("call")
```

Output from clang-query Program

```
Match #1:

/home/jdoe/example_1.cpp:5:15: note: "call" binds here
        std::cout << square(2) + square(3) << '\n';
                     ^~~~~~~~~

Match #2:

/home/jdoe/example_1.cpp:5:27: note: "call" binds here
        std::cout << square(2) + square(3) << '\n';
                                 ^~~~~~~~~
2 matches.
```

Another `clang-query` Example

example_13.cpp

```
int g = 42;
int foo(int n) {
    static int c = 0;
    return n * ++g * ++c;
}
```

Input to `clang-query` Program

```
set bind-root false
m declRefExpr(to(varDecl(unless(isStaticStorageClass())).bind("d"))).bind("r")
```

Output from `clang-query` Program

```
Match #1:
/home/jdoe/example_13.cpp:2:9: note: "d" binds here
int foo(int n) {
        ^~~~~
/home/jdoe/example_13.cpp:4:9: note: "r" binds here
        return n * ++g * ++c;
               ^

Match #2:
/home/jdoe/example_13.cpp:1:1: note: "d" binds here
int g = 42;
^~~~~~~~~~
/home/jdoe/example_13.cpp:4:15: note: "r" binds here
        return n * ++g * ++c;
                     ^

2 matches.
```

Ignoring Implicit AST Nodes

- Clang generates many AST nodes for constructs not spelled explicitly in source
- in `clang-query`, all implicit AST nodes can be ignored when matching by using:

```
    set traversal IgnoreUnlessSpelledInSource
```

Implicit AST Nodes Example

- consider following AST matcher expression, which matches all type declarations:

```
type()
```

- consider running above matcher on empty (i.e., zero-length) C++ source file
- if all AST nodes considered, several matches occur due to nodes corresponding to some special pre-defined types, such as:
  - `__int128`
  - `unsigned __int128`
- if `IgnoreUnlessSpelledInSource` specified, no matches found (as expected)

Implicit AST Nodes Example: AsIs

clang-query program invoked to process empty C++ source file

Input commands for clang-query program

```
    set output print
    set traversal AsIs
    match type()
```

Output from clang-query program

```
    Match #1:
    Binding for "root":
    __int128
    Match #2:
    Binding for "root":
    unsigned __int128
    Match #3:
    Binding for "root":
    __NSConstantString_tag
    Match #4:
    Binding for "root":
    char *
    Match #5:
    Binding for "root":
    char
    Match #6:
    Binding for "root":
    __va_list_tag [1]
    Match #7:
    Binding for "root":
    __va_list_tag
    7 matches.
```

Implicit AST Nodes Example: `IgnoreUnlessSpelledInSource`

`clang-query` program invoked to process empty C++ source file

Input commands for `clang-query` program

```
    set output print
    set traversal IgnoreUnlessSpelledInSource
    match type()
```

Output from `clang-query` program

```
    0 matches.
```

Determining AST Matcher Expression

- dump AST for code of interest to determine particular structure to be matched
- use `clang-query` program to assist in selection of appropriate matcher expression
- use expression in source code of Clang tool being developed

`clang-query` Demonstration

- consider matching all **`for`** statements in C++ source code
- use `clang-query` to help guide process

Section 3.10.3

Finding AST Nodes With AST Matchers

Key Classes Associated with AST Matchers

- `clang::ast_matchers::MatchFinder` class:
  - provides mechanism for traversing AST in order to find matching nodes
- matcher class, normally chosen from one of numerous matcher classes in `clang::ast_matchers`:
  - holds predicate used to determine if node is match
- match-callback class, derived from `clang::ast_matchers::MatchCallback`:
  - specifies actions to be taken when match found
- `clang::ast_matchers::MatchFinder::MatchResult` class:
  - holds match result

clang::ast_matchers::MatchFinder Class

- clang::ast_matchers::MatchFinder class provides mechanism for finding matches over AST
- after creation, can add one or more matchers via calls to addMatcher
- addMatcher has several overloads with signature of form:

  void addMatcher(*MatcherType*&, MatchCallback*)
- newASTConsumer method returns AST consumer that will trigger specified callbacks at appropriate points in matching process
- can generate frontend-action factory for MatchFinder using overload of clang::tooling::newFrontendActionFactory function (see https://clang.llvm.org/doxygen/namespaceclang_1_1tooling.html#a2e8ce7afec3d75043d937692e393fe7f)
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1ast__matchers_1_1MatchFinder.html

clang::...::MatchFinder::MatchResult Class

- clang::ast_matchers::MatchFinder::MatchResult class holds all information for match found
- public data members of class include:
  - Nodes: collection of nodes bound on current match, represented by BoundNodes class
  - Context: ASTContext instance associated with match
  - SourceManager: SourceManager instance associated with match
- BoundNodes type provides getNodeAs template member function which can be used to obtain pointer to matched node bound to particular name
- for more information, see:
  - https://clang.llvm.org/doxygen/structclang_1_1ast__matchers_1_1MatchFinder_1_1MatchResult.html

`clang::...::MatchCallback` Class

- `clang::ast_matchers::MatchFinder::MatchCallback` class provides abstract interface for specifying callbacks that are invoked at particular stages of matching process
- each callback is virtual function
- library user inherits from `MatchCallback` class and provides desired behavior by overriding appropriate virtual functions
- some callbacks include:
  - `run`: called for each match
  - `onStartOfTranslationUnit`: called at start of each translation unit
  - `onEndOfTranslationUnit`: called at end of each translation unit
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1ast__matchers_1_1MatchFinder_1_1MatchCallback.html

Writing a Simple AST Matcher Program

- at code development time:
  - create match-callback type that derives from `clang::ast_matchers::MatchFinder::MatchCallback`
    - override `run` method to handle each match result found
  - for each pattern of interest in AST, write matcher (of appropriate matcher type) that matches pattern
- at run time:
  1. create instance of match-callback type
  2. create `clang::ast_matchers::MatchFinder` instance
  3. add each matcher to `MatchFinder` instance via `addMatcher` method
  4. use `MatchFinder` instance to generate AST frontend-action factory (e.g., by using `clang::tooling::newFrontendActionFactory`)
  5. create `clang::tooling::ClangTool` instance
  6. invoke `run` method of `ClangTool` instance with (above) frontend-action factory to generate AST for each source file and perform AST matching

AST Matcher Example: Summary

- in slides/examples/ast_matcher_1 directory in companion repository
- runs compiler frontend on specified source files
- uses AST matcher to find each instance of call to function specified on command line
- for each function call found, program prints lines of source code containing function call
- when looking for calls to function foo, output might resemble something like:

```
match at /home/jdoe/example_1.cpp:14(10)-18(2):
  14:   int i = foo(
  15:     1,
  16:     2,
  17:     3
  18:   );
match at /home/jdoe/example_1.cpp:19(10)-19(21):
  19:   int j = foo(4, 5, 6);
```

AST Matcher Example: main.cpp (1)

```cpp
#include <format>
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "utilities.hpp"

namespace ct = clang::tooling;
namespace cam = clang::ast_matchers;

clang::SourceLocation getLineStart(const clang::SourceManager& sourceManager,
  clang::SourceLocation loc) {
    return sourceManager.translateLineCol(sourceManager.getFileID(loc),
      sourceManager.getSpellingLineNumber(loc), 1);
}

clang::SourceLocation getLineEnd(const clang::SourceManager& sourceManager,
  clang::SourceLocation loc) {
    return sourceManager.translateLineCol(sourceManager.getFileID(loc),
      sourceManager.getSpellingLineNumber(loc), ~0);
}
```

AST Matcher Example: `main.cpp` (2)

```
struct MyMatchCallback : public cam::MatchFinder::MatchCallback {
    void run(const cam::MatchFinder::MatchResult& result) override {
        clang::SourceManager& sourceManager = *result.SourceManager;
        if (auto p = result.Nodes.getNodeAs<clang::CallExpr>("call")) {
            clang::SourceLocation startLoc = p->getBeginLoc();
            clang::SourceLocation endLoc = p->getEndLoc();
            llvm::outs() << std::format("match at {}:\n", rangeToString(
              sourceManager, clang::SourceRange(startLoc, endLoc)));
            clang::SourceLocation lineStartLoc = getLineStart(sourceManager,
              startLoc);
            clang::SourceLocation lineEndLoc = getLineEnd(sourceManager,
              endLoc);
            unsigned int startLineNo = sourceManager.getSpellingLineNumber(
              lineStartLoc);
            std::string text = getSourceText(sourceManager,
              clang::SourceRange(lineStartLoc, lineEndLoc));
            llvm::outs() << addLineNumbers(text, startLineNo) << "\n";
        }
    }
};

cam::StatementMatcher getMatcher(const std::string& funcName) {
    using namespace cam;
    return callExpr(callee(functionDecl(hasName(funcName)))).bind("call");
}
```

AST Matcher Example: `main.cpp` (3)

```cpp
static llvm::cl::OptionCategory optionCategory("Tool options");
static llvm::cl::opt<std::string> clFuncName(
  "f", llvm::cl::desc("Function name"), llvm::cl::value_desc("function_name"),
  llvm::cl::cat(optionCategory), llvm::cl::Required);
;

int main(int argc, const char **argv) {
    auto expectedParser = ct::CommonOptionsParser::create(argc, argv,
      optionCategory);
    if (!expectedParser) {
        llvm::errs() << llvm::toString(expectedParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = expectedParser.get();
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    MyMatchCallback matchCallback;
    cam::StatementMatcher matcher = getMatcher(clFuncName);
    cam::MatchFinder matchFinder;
    matchFinder.addMatcher(matcher, &matchCallback);
    return tool.run(ct::newFrontendActionFactory(&matchFinder).get());
}
```

Section 3.10.4

Custom AST Matchers

Macros for Defining AST Matchers

- numerous macros provided for defining new AST matchers
- family of macros for defining AST matcher by specifying ***predicate***, including:
  - AST_MATCHER, AST_MATCHER_P, AST_MATCHER_P_OVERLOAD, AST_MATCHER_P2, AST_MATCHER_P2_OVERLOAD
  - AST_POLYMORPHIC_MATCHER, AST_POLYMORPHIC_MATCHER_P, AST_POLYMORPHIC_MATCHER_P_OVERLOAD, AST_POLYMORPHIC_MATCHER_P2, AST_POLYMORPHIC_MATCHER_P2_OVERLOAD
- family of macros for defining AST matcher by specifying ***matcher factory function***, including:
  - AST_MATCHER_FUNCTION, AST_MATCHER_FUNCTION_P, AST_MATCHER_FUNCTION_P_OVERLOAD
- numerous other macros for defining AST matchers (e.g., for handling traverse matchers and regex parameters)
- for more information, see:
  - https://clang.llvm.org/doxygen/ASTMatchersMacros_8h.html

`AST_MATCHER` Macro

- syntax:

  ```
  AST_MATCHER(Type, DefineMatcher)
  ```
- defines ***zero-parameter predicate*** on nodes of type `Type` invoked via function named `DefineMatcher`
- predicate returns **`bool`** indicating if node matches
- provides variables:
  - `Node`: AST node being matched (of type **`const`** `Type&`)
  - `Finder`: AST match finder (of type `clang::ast_matchers::internal::ASTMatchFinder*`)
  - `Builder`: builder (of type `clang::ast_matchers::internal::BoundNodesTreeBuilder*`)

`AST_MATCHER_P` and `AST_MATCHER_P_OVERLOAD` Macros

- syntax:

  ```
  AST_MATCHER_P(Type, DefineMatcher, ParamType, Param)
  ```

- defines ***single-parameter predicate*** on nodes of type `Type` invoked via function named `DefineMatcher`
- predicate returns **`bool`** indicating if node matches
- parameter named `Param1` (of type `ParamType1`)
- provides variables:
  - `Node`: AST node being matched (of type **`const`** `Type&`)
  - `Finder`: AST match finder (of type `clang::ast_matchers::internal::ASTMatchFinder*`)
  - `Builder`: builder (of type `clang::ast_matchers::internal::BoundNodesTreeBuilder*`)
- `AST_MATCHER_P_OVERLOAD` macro similar to `AST_MATCHER_P` macro, except adds extra ID parameter used to disambiguate overloads of overloaded predicate

`AST_MATCHER_P2` and `AST_MATCHER_P2_OVERLOAD` Macros

- syntax:

```
AST_MATCHER_P2(Type, DefineMatcher, ParamType1, Param1,
  ParamType2, Param2)
```

- defines ***two-parameter predicate*** on nodes of type `Type` invoked via function named `DefineMatcher`
- parameters named `Param1` (of type `ParamType1`) and `Param2` (of type `ParamType2`)
- predicate returns **`bool`** indicating if node matches
- provides variables:
  - `Node`: AST node being matched (of type **`const`** `Type&`)
  - `Finder`: AST match finder (of type `clang::ast_matchers::internal::ASTMatchFinder*`)
  - `Builder`: builder (of type `clang::ast_matchers::internal::BoundNodesTreeBuilder*`)
- `AST_MATCHER_P2_OVERLOAD` macro similar to `AST_MATCHER_P2`, except adds extra ID parameter used to disambiguate overloads of overloaded predicate

Macros for Defining AST Matchers via Factory Functions

- several macros provided for defining custom AST matchers by specifying factory function for matcher instances
- `AST_MATCHER_FUNCTION` macro
  - syntax:

    ```
    AST_MATCHER_FUNCTION(ReturnType, DefineMatcher)
    ```

  - defines ***zero-parameter*** function named `DefineMatcher` that returns ***matcher instance*** (of type `ReturnType`)
- `AST_MATCHER_FUNCTION_P` macro
  - syntax:

    ```
    AST_MATCHER_FUNCTION_P(ReturnType, DefineMatcher,
      ParamType, Param)
    ```

  - defines ***single-parameter*** function named `DefineMatcher` that returns ***matcher instance*** (of type `ReturnType`)
  - variable `Param` used for parameter (of type `ParamType`) passed to function

AST Matcher Example: Summary

- in slides/examples/ast_matcher_2 directory in companion repository
- runs compiler frontend on specified source files
- uses AST matcher to find each instance of node that matches criteria selected on command line
- for each match found, program prints source code associated with match
- demonstrates use of macros like AST_MATCHER, AST_MATCHER_P, and AST_MATCHER_P2
- demonstrates use of clang::ast_matchers::traverse to control if implicit nodes should be ignored

AST Matcher Example: `main.cpp` (1)

```cpp
#include <format>
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "utilities2.hpp"

namespace ct = clang::tooling;
namespace cam = clang::ast_matchers;

static llvm::cl::OptionCategory optionCategory("Tool options");
static llvm::cl::opt<int> clMatcherId("m", llvm::cl::desc("Matcher ID"),
  llvm::cl::value_desc("matcher_id"), llvm::cl::cat(optionCategory),
  llvm::cl::init(0));
static llvm::cl::opt<bool> clAllNodes("a", llvm::cl::desc("all nodes"),
  llvm::cl::cat(optionCategory), llvm::cl::init(false));

AST_MATCHER(clang::CXXMethodDecl, isSpecialMember) {
   if (auto p = llvm::dyn_cast<clang::CXXConstructorDecl>(&Node)) {
      return p->isDefaultConstructor() || p->isCopyConstructor() ||
        p->isMoveConstructor();
   } else if (auto p = llvm::dyn_cast<clang::CXXDestructorDecl>(&Node)) {
      return true;
   } else {
      return Node.isCopyAssignmentOperator() ||
        Node.isMoveAssignmentOperator();
   }
}
```

AST Matcher Example: `main.cpp` (2)

```cpp
AST_MATCHER_P(clang::CXXMethodDecl, paramCountAtLeast, unsigned, threshold) {
    return Node.param_size() >= threshold;
}

AST_MATCHER_P2(clang::NamedDecl, nameLengthBetween, unsigned, low, unsigned,
  high) {
    return Node.getIdentifier() && Node.getName().size() >= low &&
      Node.getName().size() <= high;
}

cam::DeclarationMatcher getMatcher(int id) {
    using namespace cam;
    switch (id) {
    default:
    case 0:
        return cxxMethodDecl(isDefinition(), isSpecialMember()).bind("x");
    case 1:
        return cxxMethodDecl(paramCountAtLeast(4)).bind("x");
    case 2:
        return namedDecl(nameLengthBetween(3, 4)).bind("x");
    }
}
```

AST Matcher Example: `main.cpp` (3)

```cpp
class MyMatchCallback : public cam::MatchFinder::MatchCallback {
public:
    MyMatchCallback() : count_(0) {}
    void run(const cam::MatchFinder::MatchResult& result) override {
        const clang::SourceManager& sourceManager = *result.SourceManager;
        clang::SourceRange sourceRange;
        std::string nodeType;
        if (auto p = result.Nodes.getNodeAs<clang::CXXMethodDecl>("x")) {
            nodeType = "CXXMethodDecl";
            sourceRange = p->getSourceRange();
        } else if (auto p = result.Nodes.getNodeAs<clang::FunctionDecl>("x")) {
            nodeType = "FunctionDecl";
            sourceRange = p->getSourceRange();
        }
        if (sourceRange.isValid()) {
            llvm::outs() << std::format("found matching {} at {}\n", nodeType,
              locationToString(sourceManager, sourceRange.getBegin(), true));
            sourceRange.setBegin(sourceManager.getSpellingLoc(sourceRange.getBegin()));
            sourceRange.setEnd(sourceManager.getSpellingLoc(sourceRange.getEnd()));
            sourceRange.setEnd(getEndOfToken(sourceManager,
              sourceRange.getEnd()));
            if (sourceRange.isValid()) {
                llvm::outs() << getSourceTextRaw(sourceManager, sourceRange) << '\n';
            }
        }
        ++count_;
    }
    unsigned getCount() const {return count_;}
private:
    unsigned count_;
};
```

AST Matcher Example: `main.cpp` (4)

```
int main(int argc, const char **argv) {
    auto expectedParser = ct::CommonOptionsParser::create(argc, argv,
      optionCategory);
    if (!expectedParser) {
        llvm::errs() << llvm::toString(expectedParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = expectedParser.get();
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    cam::DeclarationMatcher matcher = getMatcher(clMatcherId);
    if (!clAllNodes) {
        matcher = clang::ast_matchers::traverse(
          clang::TK_IgnoreUnlessSpelledInSource, matcher);
    }
    MyMatchCallback matchCallback;
    cam::MatchFinder matchFinder;
    matchFinder.addMatcher(matcher, &matchCallback);
    int status = tool.run(ct::newFrontendActionFactory(&matchFinder).get());
    llvm::outs() << std::format("number of matches: {}\n",
      matchCallback.getCount());
    return !status ? 0 : 1;
}
```

Section 3.10.5

AST Visitors Versus AST Matchers

AST Visitors Versus AST Matchers

- use of AST matchers can often result in more concise code (relative to AST visitors) by eliminating boilerplate
- for example, if pattern involves descendants/ancestors, approach based on AST visitor would need additional boilerplate to locate those descendants/ancestors, whereas in AST matcher case, library itself provides boilerplate to locate and bind to relevant nodes
- AST visitors can often be better suited to searching when patterns involve variable number of nodes and/or complex relationships between nodes
- use whichever approach best suited for task at hand

For-Statement Example: Summary

- in `slides/examples/ast_visitor_matcher_1` directory in companion repository
- run compiler frontend on specified source files
- for each non-member and member function containing at least one **`for`** statement (including range-based **`for`** statements), print maximum number of levels of nested **`for`** statements in function
- example output:

```
foo1c ... 1
foo2a ... 2
foo3a ... 3
(anonymous class)::operator() ... 1
main()::(anonymous class)::operator() ... 2
```

Visitor-Based For-Statement Example: `visitor.cpp` (1)

```cpp
#include <format>
#include <stack>
#include <type_traits>
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;
```

Visitor-Based For-Statement Example: visitor.cpp (2)

```
class MyAstVisitor : public clang::RecursiveASTVisitor<MyAstVisitor> {
public:
    using Base = clang::RecursiveASTVisitor<MyAstVisitor>;
    MyAstVisitor(clang::ASTContext& astContext) : astContext_(&astContext) {}
    bool shouldVisitImplicitCode() const {return true;}
    bool shouldVisitTemplateInstantiations() const {return true;}
    bool TraverseFunctionDecl(clang::FunctionDecl* funcDecl)
      {return handleFunc<clang::FunctionDecl>(funcDecl);}
    bool TraverseCXXMethodDecl(clang::CXXMethodDecl* funcDecl)
      {return handleFunc<clang::CXXMethodDecl>(funcDecl);}
    bool TraverseForStmt(clang::ForStmt* forStmt)
      {return handleFor<clang::ForStmt>(forStmt);}
    bool TraverseCXXForRangeStmt(clang::CXXForRangeStmt* forStmt)
      {return handleFor<clang::CXXForRangeStmt>(forStmt);}
private:
    struct StackEntry {
        const clang::FunctionDecl* funcDecl;
        unsigned forDepth;
        unsigned maxForDepth;
    };
    template<class NodeType> bool handleFunc(NodeType* funcDecl);
    template<class NodeType> bool handleFor(NodeType* forStmt);
    clang::ASTContext* astContext_;
    std::stack<StackEntry> stack_;
};
```

Visitor-Based For-Statement Example: visitor.cpp (3)

```
template<class NodeType> bool MyAstVisitor::handleFunc(NodeType* funcDecl) {
    const clang::SourceManager& sourceManager =
      astContext_->getSourceManager();
    if (sourceManager.getFileID(funcDecl->getLocation()) !=
      sourceManager.getMainFileID()) {return true;}
    stack_.push({funcDecl, 0, 0});
    bool result;
    if constexpr (std::is_same_v<NodeType, clang::CXXMethodDecl>)
      {result = Base::TraverseCXXMethodDecl(funcDecl);}
    else {result = Base::TraverseFunctionDecl(funcDecl);}
    if (stack_.top().maxForDepth > 0) {
        llvm::outs() << std::format("{} ... {}\n",
          stack_.top().funcDecl->getQualifiedNameAsString(),
          stack_.top().maxForDepth);
    }
    stack_.pop();
    return result;
}

template<class NodeType> bool MyAstVisitor::handleFor(NodeType* forStmt) {
    if (stack_.empty()) {return true;}
    StackEntry& top = stack_.top();
    ++top.forDepth;
    top.maxForDepth = std::max(top.maxForDepth, top.forDepth);
    bool result;
    if constexpr(std::is_same_v<NodeType, clang::CXXForRangeStmt>)
      {result = Base::TraverseCXXForRangeStmt(forStmt);}
    else {result = Base::TraverseForStmt(forStmt);}
    --top.forDepth;
    return result;
}
```

Visitor-Based For-Statement Example: visitor.cpp (4)

```
struct MyAstConsumer : public clang::ASTConsumer {
    void HandleTranslationUnit(clang::ASTContext& astContext) final {
        MyAstVisitor visitor(astContext);
        visitor.TraverseDecl(astContext.getTranslationUnitDecl());
    }
};

struct MyFrontendAction : public clang::ASTFrontendAction {
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance&, clang::StringRef fileName) final {
        llvm::outs() << std::format("PROCESSING SOURCE FILE {}\n", fileName);
        return std::unique_ptr<clang::ASTConsumer>{new MyAstConsumer};
    }
};

static llvm::cl::OptionCategory toolOptions("Tool Options");

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolOptions);
    if (!expectedOptionsParser) {
        llvm::errs() << std::format("Unable to create option parser ({}).\n",
          llvm::toString(expectedOptionsParser.takeError()));
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    int status = tool.run(
      ct::newFrontendActionFactory<MyFrontendAction>().get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

Matcher-Based For-Statement Example: `matcher.cpp` (1)

```
#include <cassert>
#include <format>
#include <map>
#include "clang/AST/ASTContext.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/AST/ParentMapContext.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace ct = clang::tooling;
namespace cam = clang::ast_matchers;
```

Matcher-Based For-Statement Example: `matcher.cpp` (2)

```cpp
template<class NodeType>
const NodeType* getParentOfStmt(clang::ASTContext& astContext,
  const clang::Stmt* stmt) {
    auto parents = astContext.getParents(*stmt);
    const clang::Stmt* curStmt = nullptr;
    const NodeType* parent = nullptr;
    for (auto&& node : parents) {
        if (auto p = node.get<NodeType>()) {
            assert(!parent);
            parent = p;
        }
    }
    return parent;
}

unsigned getForDepth(clang::ASTContext& astContext,
  const clang::Stmt* forStmt) {
    assert(llvm::isa<clang::ForStmt>(forStmt) ||
      llvm::isa<clang::CXXForRangeStmt>(forStmt));
    unsigned count = 1;
    const clang::Stmt* curStmt = forStmt;
    while ((curStmt = getParentOfStmt<clang::Stmt>(astContext, curStmt))) {
        if (llvm::isa<clang::ForStmt>(curStmt) ||
          llvm::isa<clang::CXXForRangeStmt>(curStmt)) {++count;}
    }
    return count;
}
```

Matcher-Based For-Statement Example: `matcher.cpp` (3)

```
class MyMatchCallback : public cam::MatchFinder::MatchCallback {
public:
    void run(const cam::MatchFinder::MatchResult& result) final;
    void onStartOfTranslationUnit() final {funcTab_.clear();}
    void onEndOfTranslationUnit() final;
private:
    using FuncTab = std::map<const clang::FunctionDecl*, unsigned>;
    FuncTab funcTab_;
};

void MyMatchCallback::onEndOfTranslationUnit() {
    for (auto [funcDecl, maxForDepth] : funcTab_) {
        llvm::outs() << std::format("{} ... {}\n",
          funcDecl->getQualifiedNameAsString(), maxForDepth);
    }
    funcTab_.clear();
}

void MyMatchCallback::run(const cam::MatchFinder::MatchResult& result) {
    const clang::SourceManager& sourceManager = *result.SourceManager;
    auto forStmt = result.Nodes.getNodeAs<clang::Stmt>("for");
    auto funcDecl = result.Nodes.getNodeAs<clang::FunctionDecl>("func");
    if (funcDecl && forStmt) {
        auto iter = funcTab_.find(funcDecl);
        if (iter == funcTab_.end()) {
            iter = funcTab_.insert(std::make_pair(funcDecl, 0)).first;
        }
        unsigned depth = getForDepth(*result.Context, forStmt);
        iter->second = std::max(iter->second, depth);
    }
}
```

Matcher-Based For-Statement Example: matcher.cpp (4)

```
cam::StatementMatcher getMatcher() {
    using namespace cam;
    auto f = anyOf(forStmt(), cxxForRangeStmt());
    return stmt(f, hasAncestor(functionDecl(isExpansionInMainFile()).bind(
      "func")), unless(hasDescendant(stmt(f)))).bind("for");
}

struct MyAstConsumer : public clang::ASTConsumer {
    void HandleTranslationUnit(clang::ASTContext& astContext) final {
        MyMatchCallback matchCallback;
        cam::StatementMatcher matcher = getMatcher();
        cam::MatchFinder matchFinder;
        matchFinder.addMatcher(matcher, &matchCallback);
        matchFinder.matchAST(astContext);
    }
};

struct MyFrontendAction : public clang::ASTFrontendAction {
    std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance&, clang::StringRef fileName) final {
        llvm::outs() << std::format("PROCESSING SOURCE FILE {}\n", fileName);
        return std::unique_ptr<clang::ASTConsumer>{new MyAstConsumer};
    }
};
```

Matcher-Based For-Statement Example: matcher.cpp (5)

```
static llvm::cl::OptionCategory optionCategory("Tool options");

int main(int argc, const char **argv) {
   auto expectedParser = ct::CommonOptionsParser::create(argc, argv,
     optionCategory);
   if (!expectedParser) {
      llvm::errs() << llvm::toString(expectedParser.takeError());
      return 1;
   }
   ct::CommonOptionsParser& optionsParser = expectedParser.get();
   ct::ClangTool tool(optionsParser.getCompilations(),
     optionsParser.getSourcePathList());
   int status = tool.run(ct::newFrontendActionFactory<MyFrontendAction>().get());
   if (status) {llvm::errs() << "error detected\n";}
   return !status ? 0 : 1;
}
```

Section 3.10.6

References

References

- various talks and articles by Stephen Kelly (as well as talks/articles by Eli Bendersky and others) listed in References section
- search for “`[clang-ast-matchers]`” on StackOverflow
- some examples of AST matchers can be found in `https://github.com/lanl/CoARCT`

Section 3.11

Control-Flow Graphs (CFGs)

clang::CFG Class

- clang::CFG class represents control-flow graph (CFG) corresponding to source-level intra-procedural control-flow of Stmt (i.e., statement)
- CFG object is essentially collection of CFGBlock elements, which represent basic blocks in CFG
- CFG object always has two dummy blocks, designating entry and exit points of CFG
- some nonstatic methods provided by class include:
  - begin and end: return range corresponding to blocks in CFG
  - size: get number of blocks in CFG
  - isLinear: return true if CFG has no branches
- clang::CFG::BuildCFG factory function provided for building CFG corresponding to Stmt node in AST (such as compound statement comprising function body)
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1CFG.html

CFG Pretty Printer Example: Summary

- in `slides/examples/dump_cfg` directory in companion repository
- runs compiler frontend on specified source files to produce AST
- uses AST matcher to find functions whose name matches regular expression
- for each function found, CFG is generated
- information about CFG is output using pretty-print functionality of Clang libraries

CFG Pretty Printer Example: Sample Program Output

Input Source File

```
int abs(int x) {
    if (x < 0) {return -x;}
    return x;
}
```

Program Output

```
FUNCTION: abs

 [B4 (ENTRY)]
   Succs (1): B3

 [B1]
   1: x (ImplicitCastExpr, LValueToRValue, int)
   2: return [B1.1];
   Preds (1): B3
   Succs (1): B0

 [B2]
   1: -x
   2: return [B2.1];
   Preds (1): B3
   Succs (1): B0

 [B3]
   1: x < 0
   T: if [B3.1]
   Preds (1): B4
   Succs (2): B2 B1

 [B0 (EXIT)]
   Preds (2): B1 B2
```

CFG Pretty Printer Example: main.cpp (1)

```
#include <format>
#include <string>
#include "clang/Analysis/CFG.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Basic/LangOptions.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"

namespace cam = clang::ast_matchers;
namespace ct = clang::tooling;
namespace lc = llvm::cl;

static lc::OptionCategory toolCategory("Tool Options");
static lc::opt<std::string> clFuncNamePattern("f", lc::cat(toolCategory),
  lc::init(".*"));
static lc::opt<bool> clUseColor("c", lc::cat(toolCategory), lc::init(false));
```

CFG Pretty Printer Example: main.cpp (2)

```
cam::DeclarationMatcher getFuncMatcher(const std::string& namePattern)
 {return cam::functionDecl(cam::matchesName(namePattern)).bind("func");}

struct MyMatchCallback : public cam::MatchFinder::MatchCallback {
  virtual void run(const cam::MatchFinder::MatchResult& result) final {
    if (const auto* funcDecl =
     result.Nodes.getNodeAs<clang::FunctionDecl>("func")) {
      clang::ASTContext *astContext = result.Context;
      clang::Stmt *funcBody = funcDecl->getBody();
      if (!funcBody) {return;}
      llvm::outs() << std::format("FUNCTION: {}\n",
       funcDecl->getQualifiedNameAsString());
      std::unique_ptr<clang::CFG> cfg = clang::CFG::buildCFG(
       funcDecl, funcBody, astContext, clang::CFG::BuildOptions());
      if (!cfg) {
        llvm::outs() << "unable to generate CFG\n";
        return;
      }
      auto langOpts = astContext->getLangOpts();
      cfg->print(llvm::outs(), langOpts, clUseColor);
    }
  }
};
```

CFG Pretty Printer Example: main.cpp (3)

```
int main(int argc, const char **argv) {
    llvm::Expected<ct::CommonOptionsParser> expOptionsParser =
    ct::CommonOptionsParser::create(argc, argv, toolCategory);
    if (!expOptionsParser) {
        llvm::errs() << llvm::toString(expOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    cam::DeclarationMatcher funcMatcher = getFuncMatcher(clFuncNamePattern);
    MyMatchCallback matchCallback;
    cam::MatchFinder finder;
    finder.addMatcher(funcMatcher, &matchCallback);
    int status = tool.run(ct::newFrontendActionFactory(&finder).get());
    if (status) {llvm::errs() << "error occurred\n";}
    return !status ? 0 : 1;
}
```

clang::CFGBlock Class

- clang::CFGBlock class represents single basic block in CFG
- CFGBlock object consists of:
  - set of statements/expressions (which may contain subexpressions)
  - terminator statement (not in set of statements), which represents type of control-flow that occurs at end of basic block
  - list of successors (where order is not arbitrary)
  - list of predecessors (where order is arbitrary)
- some methods provided by class include:
  - begin and end: return range corresponding to elements in block (e.g., statements)
  - size: get number of elements in block
  - succ_begin and succ_end: return range corresponding to blocks that are successors to block
  - succ_size: get number of successor blocks
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1CFGBlock.html

`clang::CFGElement` Class

- `clang::CFGElement` class represents top-level expression in basic block
- some methods provided by class include:
  - `dumpToStream`: outputs information about element to stream in human-readable format
  - `getKind`: get kind of element (e.g., statement, constructor)
  - `getAs`: get as specified type or return empty optional if does not have specified type
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1CFGElement.html

Cyclomatic-Complexity Example: Summary

- in `slides/examples/cyclomatic_complexity` directory in companion repository
- cyclomatic (a.k.a., McCabe) complexity is measure of code complexity
- cyclomatic complexity $M = E - N + 2P$, where E is number of edges, N is number of nodes, P is number of connected components
- runs compiler frontend on specified source files to produce AST
- uses AST matcher to find function definitions
- for each function definition, program does following:
  - constructs CFG for function
  - computes cyclomatic complexity of function
  - if complexity not less than specified threshold, prints complexity information for function
- complexity threshold can be specified as command-line option
- output resembles something like:

```
identity 1
abs 2
foo 6
```

Cyclomatic-Complexity Example: `matcher.cpp` (1)

```cpp
#include <format>
#include "clang/Analysis/CFG.h"
#include "clang/AST/ASTContext.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/raw_ostream.h"

namespace ct = clang::tooling;
namespace cam = clang::ast_matchers;

static llvm::cl::OptionCategory toolCategory("Tool Options");
static llvm::cl::opt<unsigned int> thresholdOption("t",
  llvm::cl::init(0), llvm::cl::desc("Set complexity threshold."),
  llvm::cl::cat(toolCategory));
```

Cyclomatic-Complexity Example: matcher.cpp (2)

```
int cyclomaticComplexity(const clang::FunctionDecl& funcDecl,
  clang::ASTContext& astContext) {
    const auto cfg = clang::CFG::buildCFG(&funcDecl, funcDecl.getBody(),
      &astContext, clang::CFG::BuildOptions());
    if (!cfg) {return -1;}
    const int numNodes = cfg->size() - 2;
    int numEdges = 0;
    for (const auto* block : *cfg) {numEdges += block->succ_size();}
    numEdges -= 2;
    return numEdges - numNodes + (2 * 1);
}

struct MyMatchCallback : public cam::MatchFinder::MatchCallback {
    using MatchResult = cam::MatchFinder::MatchResult;
    void run(const MatchResult& result) override {
        const auto* function =
          result.Nodes.getNodeAs<clang::FunctionDecl>("f");
        std::string s = function->getQualifiedNameAsString();
        int complexity = cyclomaticComplexity(*function,
          *result.Context);
        if (complexity >= 0 && complexity >= thresholdOption) {
            llvm::outs() << std::format("{} {}\n", s, complexity);
        }
    }
};
```

Cyclomatic-Complexity Example: `matcher.cpp` (3)

```cpp
int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
    const_cast<const char**>(argv), toolCategory);
    if (!expectedOptionsParser) {
        llvm::errs() << llvm::toString(expectedOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    auto matcher =
      cam::functionDecl(cam::isExpansionInMainFile()).bind("f");
    MyMatchCallback matchCallback;
    cam::MatchFinder matchFinder;
    matchFinder.addMatcher(matcher, &matchCallback);
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    auto status =
      tool.run(ct::newFrontendActionFactory(&matchFinder).get());
    if (status) {llvm::errs() << "error detected\n";}
    return !status ? 0 : 1;
}
```

CFG Example: Summary

- in slides/examples/cfg_1 directory in companion repository
- runs compiler frontend on specified source files to produce AST
- uses AST matcher to find functions (or functions with specified name)
- for each function found, program generates CFG for function and prints information about CFG blocks and elements
- output resembles something like:

```
FUNCTION: main
block: 0 (exit)
block: 1
successors: 0
statement: 0
statement: return 0;
block: 2
successors: 0
statement: 1
statement: return 1;
block: 3
successors: 2 1
statement: argc != 1
block: 4 (entry)
successors: 3
```

CFG Example: `cfg.cpp` (1)

```cpp
#include <format>
#include <map>
#include <string>
#include "clang/Analysis/CFG.h"
#include "clang/AST/ASTContext.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Frontend/FrontendAction.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/raw_ostream.h"

namespace lc = llvm::cl;
namespace ct = clang::tooling;
namespace cam = clang::ast_matchers;

static lc::OptionCategory toolCategory("Tool Options");
static lc::opt<std::string> clFuncName("f", lc::cat(toolCategory));

std::string toString(clang::CFGElement::Kind kind) {
    const std::map<clang::CFGElement::Kind, std::string> lut{
      {clang::CFGElement::Kind::Statement, "statement"},
      {clang::CFGElement::Kind::Constructor, "constructor"},
      {clang::CFGElement::Kind::CXXRecordTypedCall, "recordTypedCall"},
    };
    auto i = lut.find(kind);
    return std::format("{}", (i != lut.end() ? i->second : "unknown"));
}
```

CFG Example: cfg.cpp (2)

```
void printBlock(llvm::raw_ostream& out, const clang::CFG& cfg,
  const clang::CFGBlock& block) {
    out << std::format("block: {}", block.BlockID);
    if (&block == &cfg.getEntry()) {out << " (entry)";}
    if (&block == &cfg.getExit()) {out << " (exit)";}
    if (block.hasNoReturnElement()) {out << " (noreturn)";}
    out << '\n';
    if (block.succ_size()) {
        out << "successors:";
        for (auto succBlockIter = block.succ_begin(); succBlockIter !=
          block.succ_end(); ++succBlockIter) {
            out << std::format(" {}", (*succBlockIter) ? std::format("{}",
              (*succBlockIter)->BlockID) : "invalid");
        }
        out << '\n';
    }
    for (auto elemIter = block.begin(); elemIter != block.end(); ++elemIter) {
        out << std::format("{}: ", toString(elemIter->getKind()));
        elemIter->dumpToStream(out);
    }
}

void processFunc(const clang::FunctionDecl& funcDecl, clang::ASTContext&
  astContext) {
    llvm::outs() << std::format("FUNCTION: {}\n",
      funcDecl.getQualifiedNameAsString());
    const auto cfg = clang::CFG::buildCFG(&funcDecl, funcDecl.getBody(),
      &astContext, clang::CFG::BuildOptions());
    if (!cfg) {return;}
    for (auto blockIter = cfg->nodes_begin(); blockIter != cfg->nodes_end();
      ++blockIter) {printBlock(llvm::outs(), *cfg, **blockIter);}
}
```

CFG Example: cfg.cpp (3)

```cpp
cam::DeclarationMatcher getFuncMatcher(const std::string& name) {
    return (name.size() ? cam::functionDecl(cam::hasName(name)) :
      cam::functionDecl()).bind("func");
}

struct MyMatchCallback : public cam::MatchFinder::MatchCallback {
    virtual void run(const cam::MatchFinder::MatchResult& result) final {
        if (const auto* funcDecl =
          result.Nodes.getNodeAs<clang::FunctionDecl>("func")) {
            if (const clang::Stmt *funcBody = funcDecl->getBody())
              {processFunc(*funcDecl, *result.Context);}
        }
    }
};

int main(int argc, char** argv) {
    auto expectedOptionsParser = ct::CommonOptionsParser::create(argc,
      const_cast<const char**>(argv), toolCategory);
    if (!expectedOptionsParser) {
        llvm::errs() << llvm::toString(expectedOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expectedOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    cam::DeclarationMatcher funcMatcher = getFuncMatcher(clFuncName);
    MyMatchCallback matchCallback;
    cam::MatchFinder finder;
    finder.addMatcher(funcMatcher, &matchCallback);
    int status = tool.run(ct::newFrontendActionFactory(&finder).get());
    if (status) {llvm::errs() << "error occurred\n";}
    return !status ? 0 : 1;
}
```

Section 3.11.1

Code Analysis

clang::AnalysisDeclContextManager Class

- clang::AnalysisDeclContextManager class provides means to create/manage clang::AnalysisDeclContext instances
- AnalysisDeclContext class (to be discussed shortly) used to hold state needed for some types of code analysis
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1AnalysisDeclContextManager.html

clang::AnalysisDeclContext Class

- clang::AnalysisDeclContext class contains context data for function, method, or block under analysis
- holds CFG and CFG-related information
- can be used to perform various kinds of code analysis
- getAnalysis method:
  - factory function for generating code-analysis objects of various types
  - kind of analysis to perform specified via type template parameter
  - returns specified analysis object, lazily running analysis if necessary or nullptr if analysis could not run
  - example of analysis type: clang::LiveVariables
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1AnalysisDeclContext.html#details

clang::LiveVariables Class

- clang::LiveVariables class provides mechanism for performing live-variable analysis
- provides methods for querying if variable is live at various points (e.g., at end of specified block or at beginning of statement)
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1LiveVariables.html

Liveness-Analysis Example: Summary

- in `slides/examples/liveness_analysis` directory in companion repository
- runs compiler frontend on specified source files to produce AST
- uses AST matcher to find functions (or functions with specified name)
- for each function found, CFG for function is generated and liveness analysis is performed, and then results output
- sample output shown on next slide

Liveness-Analysis Example: Sample Program Output

Input Source File

```
1   int foo(int x, int y) { // B5 (entry)
2       int t = x * y; // B4.1
3       if ((x + 1) * (x - 1) == y) { // B4.2
4           t = 1; // B3
5       } else {
6           t = 2; // B2
7       }
8       return t; // B1
9   } // B0 (exit)
```

Program Output

```
FUNCTION: foo

[ B0 (live variables at block exit) ]

[ B1 (live variables at block exit) ]

[ B2 (live variables at block exit) ]
 t </home/jdoe/example_3_b.cpp:2:6>

[ B3 (live variables at block exit) ]
 t </home/jdoe/example_3_b.cpp:2:6>

[ B4 (live variables at block exit) ]

[ B5 (live variables at block exit) ]
 x </home/jdoe/example_3_b.cpp:1:13>
 y </home/jdoe/example_3_b.cpp:1:20>
```

Liveness-Analysis Example: `analyze.hpp`

```
#include "clang/AST/ASTContext.h"
void analyzeFunc(clang::ASTContext& astContext, const clang::FunctionDecl*
  funcDecl, bool printCfg);
```

Liveness-Analysis Example: `analyze.cpp`

```cpp
#include "clang/AST/ASTContext.h"
#include "clang/Analysis/CFG.h"
#include "clang/Analysis/AnalysisDeclContext.h"
#include "clang/Analysis/Analyses/LiveVariables.h"

void analyzeFunc(clang::ASTContext& astContext, const clang::FunctionDecl*
  funcDecl, bool printCfg) {
    clang::AnalysisDeclContextManager adcm(astContext);
    clang::AnalysisDeclContext *adc = adcm.getContext(
      llvm::cast<clang::Decl>(funcDecl));
    assert(adc);
    adc->getCFGBuildOptions().setAllAlwaysAdd();
    const clang::CFG& cfg = *adc->getCFG();
    if (printCfg)
      {cfg.print(llvm::outs(), astContext.getLangOpts(), false);}
    clang::LiveVariables *lv = adc->getAnalysis<clang::LiveVariables>();
    if (!lv) {return;}
    auto observer = std::make_unique<clang::LiveVariables::Observer>();
    assert(observer);
    lv->runOnAllBlocks(*observer);
    lv->dumpBlockLiveness((funcDecl->getASTContext()).getSourceManager());
}
```

Liveness-Analysis Example: `main.cpp` (1)

```cpp
#include <format>
#include <string>
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h"
#include "analyze.hpp"

namespace cam = clang::ast_matchers;
namespace ct = clang::tooling;
namespace lc = llvm::cl;

static lc::OptionCategory toolCategory("Tool Options");
static lc::opt<std::string> clFuncNamePattern("f", lc::cat(toolCategory),
  lc::init(".*"));
static lc::opt<bool> clPrintCfg("c", lc::cat(toolCategory), lc::init(false));

struct MyMatchCallback : public cam::MatchFinder::MatchCallback {
    virtual void run(const cam::MatchFinder::MatchResult& result) final {
        if (auto funcDecl =
          result.Nodes.getNodeAs<clang::FunctionDecl>("func")) {
            clang::ASTContext *astContext = result.Context;
            clang::Stmt *funcBody = funcDecl->getBody();
            if (!funcBody) {return;}
            llvm::outs() << std::format("FUNCTION: {}\n",
              funcDecl->getQualifiedNameAsString());
            analyzeFunc(*astContext, funcDecl, clPrintCfg);
        }
    }
};
```

Liveness-Analysis Example: `main.cpp` (2)

```
cam::DeclarationMatcher getFuncMatcher(const std::string& namePattern)
  {return cam::functionDecl(cam::matchesName(namePattern)).bind("func");}

int main(int argc, const char **argv) {
    llvm::Expected<ct::CommonOptionsParser> expOptionsParser =
    ct::CommonOptionsParser::create(argc, argv, toolCategory);
    if (!expOptionsParser) {
        llvm::errs() << llvm::toString(expOptionsParser.takeError());
        return 1;
    }
    ct::CommonOptionsParser& optionsParser = *expOptionsParser;
    ct::ClangTool tool(optionsParser.getCompilations(),
      optionsParser.getSourcePathList());
    cam::DeclarationMatcher funcMatcher = getFuncMatcher(clFuncNamePattern);
    MyMatchCallback matchCallback;
    cam::MatchFinder finder;
    finder.addMatcher(funcMatcher, &matchCallback);
    int status = tool.run(ct::newFrontendActionFactory(&finder).get());
    if (status) {llvm::errs() << "error occurred\n";}
    return !status ? 0 : 1;
}
```

Section 3.12

Miscellany

A Few More Types

- clang::DynTypedNode
  - used to represent generic AST node
  - https://clang.llvm.org/doxygen/classclang_1_1DynTypedNode.html
- clang::DynTypedNodeList
  - used to represent list of generic AST nodes
  - https://clang.llvm.org/doxygen/classclang_1_1DynTypedNodeList.html
- clang::CFGReverseBlockReachabilityAnalysis
  - check if one block reachable from another block in CFG
  - https://clang.llvm.org/doxygen/classclang_1_1CFGReverseBlockReachabilityAnalysis.html#a73cec1b9cbbc6e2461470906e6a0720a
- clang::CallGraph
  - used for constructing AST-based call graph
  - https://clang.llvm.org/doxygen/classclang_1_1CallGraph.html

Source-Code Comments and AST

- normally, comments discarded before AST is built
- including compiler option `-fparse-all-comments` will cause comments to be captured in AST
- some AST node types associated with comments (in `clang::comments` namespace) include (amongst many):
  - `BlockContentComment`
  - `FullComment`
  - `InlineContentComment`
  - `VerbatimBlockLineComment`
- HTML tags in comments represented explicitly in AST
- comments can be accessed via `Comments` member of `ASTContext`
- can get comments for `Decl` via `getCommentForDecl` member of `ASTContext`
- for more information, see:
  - https://clang.llvm.org/doxygen/classclang_1_1comments_1_1Comment.html

Part 4

References

LLVM/Clang Quick Links I

1. LLVM Project, https://llvm.org.
2. LLVM Programmer's Manual, https://llvm.org/docs/ProgrammersManual.html.
3. LLVM Tutorial: Table of Contents, https://llvm.org/docs/tutorial/index.html.
4. CommandLine 2.0 Library Manual, https://llvm.org/docs/CommandLine.html.
5. Clang Documentation, https://clang.llvm.org/docs.
6. Clang Frontend (CFE) Internals Manual, https://clang.llvm.org/docs/InternalsManual.html.
7. Clang Driver Design and Internals, https://clang.llvm.org/docs/DriverInternals.html.
8. Introduction to the Clang AST, https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

LLVM/Clang Quick Links II

9. Using Clang as a Library https://clang.llvm.org/docs/#using-clang-as-a-library.
10. LLVM Developers' Meetings, https://llvm.org/devmtg.
11. Checker Developer Manual, https://clang-analyzer.llvm.org/checker_dev_manual.html.
12. Getting Started: Building and Running Clang, https://clang.llvm.org/get_started.html.
13. Hacking on Clang, https://clang.llvm.org/hacking.html.
14. LLVM Office Hours, https://llvm.org/docs/GettingInvolved.html#office-hours.
15. How to Write RecursiveASTVisitor Based ASTFrontendActions, https://clang.llvm.org/docs/RAVFrontendAction.html.
16. Tutorial for Building Tools Using LibTooling and LibASTMatchers https://clang.llvm.org/docs/LibASTMatchersTutorial.html.
17. LLVM Discourse Site. https://discourse.llvm.org.

LLVM/Clang Quick Links III

18. Clang Frontend Section of LLVM Discourse Site. `https://discourse.llvm.org/c/clang/6`.
19. Data Flow Analysis: An Informal Introduction, `https://clang.llvm.org/docs/DataFlowAnalysisIntro.html`.
20. Clang Plugins, `https://clang.llvm.org/docs/ClangPlugins.html`.
21. LLVM Community Calendar, `https://calendar.google.com/calendar/u/0/embed?src=calendar@llvm.org`.

Books I

1. Min-Yih Hsu. LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries. Packt Publishing, Dec. 2021, https://isbnsearch.org/isbn/9781838824952. [Source code available from https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clang-and-Middle-End-Libraries.] [Chapters 5–8 discuss various aspects of the Clang frontend in some detail.]
2. Suyog Sarda and Mayur Pandey. LLVM Essentials. Packt Publishing, Dec. 2015, https://isbnsearch.org/isbn/9781785280801. [This book focuses on LLVM as opposed to Clang.]
3. Kai Nacke. Learn LLVM 12. Packt Publishing, May 2021, https://www.packtpub.com/product/cloud_and_networking/9781839213502.
4. Kai Nacke. Learn LLVM 11: A beginner's guide to learning LLVM compiler tools and core libraries with C++. Packt Publishing, Dec. 2021, https://isbnsearch.org/isbn/9781839213502.

Books II

5. Bruno Cardoso Lopes and Rafael Auler. Getting Started with LLVM Core Libraries. Packt Publishing, Aug. 2014, https://isbnsearch.org/isbn/9781782166924.
6. Mayur Pandey and Suyog Sarda. LLVM Cookbook. Packt Publishing, May 2015, https://isbnsearch.org/isbn/9781785285981. [This book does not take a systematic approach to teaching LLVM/Clang. Rather, it teaches by presenting recipes/examples.]

Articles I

1. Xin Huang. Clang Tutorial: Finding Declarations. Oct. 19, 2014, https://xinhuang.github.io/posts/2014-10-19-clang-tutorial-finding-declarations.html.
2. Xin Huang. Clang Tutorial: The AST Matcher. Feb. 8, 2015, https://xinhuang.github.io/posts/2015-02-08-clang-tutorial-the-ast-matcher.html.
3. Eli Bendersky. AST matchers and Clang refactoring tools. July 29, 2014, https://eli.thegreenplace.net/2014/07/29/ast-matchers-and-clang-refactoring-tools.
4. Eli Bendersky. Compilation databases for Clang-based tools. May 21, 2014, https://eli.thegreenplace.net/2014/05/21/compilation-databases-for-clang-based-tools.
5. Eli Bendersky. Modern source-to-source transformation with Clang and libTooling. May 1, 2014, https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling.

Articles II

[6] Jonas Devlieghere. Understanding the Clang AST. Dec. 31, 2015, https://jonasdevlieghere.com/understanding-the-clang-ast/.

[7] Stephen Kelly. Exploring Clang Tooling, Part 0: Building Your Code with Clang. Sept. 18, 2018, https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-0-building-your-code-with-clang/.

[8] Stephen Kelly. Exploring Clang Tooling Part 1: Extending Clang-Tidy. Oct. 19, 2018, https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-1-extending-clang-tidy/.

[9] Stephen Kelly. Exploring Clang Tooling Part 2: Examining the Clang AST with clang-query. Oct. 23, 2018, https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/.

Articles III

[10] Stephen Kelly. Exploring Clang Tooling Part 3: Rewriting Code with clang-tidy. Nov. 6, 2018, https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-3-rewriting-code-with-clang-tidy/.

[11] Stephen Kelly. Composing AST Matchers in `clang-tidy`. Nov. 11, 2018, https://steveire.wordpress.com/2018/11/20/composing-ast-matchers-in-clang-tidy.

[12] Stephen Kelly. Debugging Clang AST Matchers. Apr. 16, 2019, https://steveire.wordpress.com/2019/04/16/debugging-clang-ast-matchers.

[13] Stephen Kelly. AST Matchmaking Made Easy. Feb. 14, 2021, https://steveire.wordpress.com/2021/02/14/ast-matchmaking-made-easy.

[14] Stephen Kelly. Location, Location, Location. Apr. 27, 2021, https://steveire.wordpress.com/2021/04/27/location-location-location.

Articles IV

15 Ehsan Akhgari. C++ Static Analysis Using Clang. Dec. 7, 2015, https://ehsanakhgari.org/blog/2015-12-07/c-static-analysis-using-clang.

Talks I

1. Peter Goldsborough. Clang-useful: Building Useful Tools with LLVM and Clang for Fun and Profit. C++ Now, Aspen, CO, USA, May 15–20, 2017. Available online at https://youtu.be/E6i8jmiy8MY. [Code examples available at https://github.com/peter-can-talk/cppnow-2017.]
2. Sven van Haastregt and Anastasia Stulova. An Overview of Clang. LLVM Developers' Meeting, San Jose, CA, USA, Oct. 22–23, 2019. Available online at https://youtu.be/5kkMpJpIGYU.
3. Manuel Klimek. The Clang AST — A Tutorial. LLVM Euro Conference, Paris, France, Apr. 29–30, 2013. Available online at https://youtu.be/VqCkCDFLSsc.
4. Vince Bridgers and Felipe de Azevedo Piovezan. LLVM IR Tutorial — Phis, GEPs and Other Things, Oh My!. EuroLLVM Developers' Meeting, Brussels, Belgium, Apr. 8–9, 2019. Available online at https://youtu.be/m8G_S5LwlTo. [An excellent talk that well explains various aspects of LLVM IR, including phis and GEPs.]

Talks II

5. Mike Shah. Introduction to LLVM — Building Simple Program Analysis Tools and Instrumentation. FOSDEM, Brussels, Belgium, Feb. 4, 2018. Available online at https://youtu.be/VKIv_Bkp4pk.
6. Sergei Sadovnikov. Automatic C++ Source Code Generation with Clang. ACCU, Bristol, UK, Apr. 26–29, 2017. Available online at https://youtu.be/aPTyatTI42k.
7. Stephen Kelly. Refactor Your Codebase with Clang Tooling. code::dive, Wroclaw, Poland, Nov. 7–8, 2018. Available online at https://youtu.be/_T-5pWQVxeE. [Discusses how to develop refactoring tools based on `clang-tidy` and covers topics such as AST matchers, and how to use `clang-query` to facilitate the development of matchers.]
8. Meike Baumgartner and Dmitri Gribenko. My First Clang Warning. LLVM Developers' Meeting, San Jose, CA, USA, Oct. 22–23, 2019. Available online at https://youtu.be/FNnKMSkaLkY. [An excellent talk that walks through the various steps of adding a new compiler warning to Clang.]

Talks III

9. Peter Smith. YVR18-223: How to Build a C++ Processing Tool Using the Clang Libraries. Linaro Connect 2018 — YVR18, Vancouver, BC, Canada, Sept. 17–21, 2018. Available online at https://youtu.be/8QvLVEaxzC8. Slides and video available at https://resources.linaro.org/en/resource/Bi4FpRDmERUuU5ei7nry9h. [A fast-paced talk that covers an example of using the Clang libraries to apply a simple source-code transformation.]

10. Stephan Bergman. Plug Yourself In: Learn How to Write a Clang Compiler Plugin. LibreOffice Conference, Aarhus, Denmark, Sept. 24, 2015. Available online at https://youtu.be/pdxlmM477KY. [A simple Clang plugin is developed in a step-by-step fashion.]

Talks IV

11. Toby Ho. Live Code: LLVM Tutorial Walkthrough. Available online at https://youtube.com/playlist?list=PLSq9OFrD2Q3ChEc_ejnBcO5u9JeT0ufkg. [This series of videos performs a walkthrough of the LLVM tutorial titled “My First Language Frontend with LLVM Tutorial”, which can be found at https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/.]

Miscellany I

1. Victor Ciura. Better Tools in Your Clang Toolbox: Extending clang-tidy With Your Custom Checks. C++ on Sea, Folkestone, UK, Feb. 4, 2019. Available online at https://youtu.be/7CnFrn0-2TQ.
2. Jeremy Demeule. Adding a New clang-tidy Check by the Practice. CPPP, Paris, France, June 15, 2019. Available online at https://youtu.be/K-WhaEUEZWc.
3. Vince Bridgers. Using Clang-tidy for Customized Checkers and Large Scale Source Refactoring. LLVM Developers’ Meeting, Online, Oct. 7, 2020. Available online at https://youtu.be/UfLH7dORav8.
4. Vince Bridgers. Using the Clang Static Analyzer to Find Bugs. LLVM Developers’ Meeting, Online, Oct. 8, 2020. Available online at https://youtu.be/nTslG8HtKeA.
5. Stephen Kelly. Extending clang-tidy in the Present and in the Future. ACCU, Bristol, UK, Apr. 13, 2019. Available online at https://youtu.be/38tYYrnfNrs.

Miscellany II

6 Artem Dergachev. Developing the Clang Static Analyzer. LLVM Developers' Meeting, San Jose, CA, USA, Oct. 23, 2019. Available online at https://youtu.be/g0Mqx1niUi0.

7 Chris Bieneman and Kit Barton. How to Contribute to LLVM. LLVM Developers' Meeting, San Jose, CA, USA, Oct. 22, 2019. Available online at https://youtu.be/C5Y977rLqpw.

8 Andrzej Warzynski. Writing an LLVM Pass: 101. LLVM Developers' Meeting, San Jose, CA, USA, Oct. 22, 2019. Available online at https://youtu.be/ar7cJl2aBuU

9 Chandler Carruth. Understanding Compiler Optimization. Meeting C++, Berlin, Germany, Dec. 4–5, 2015. Available online at https://youtu.be/FnGCDLhaxKU.

10 Chandler Carruth. Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!. CppCon, Bellevue, WA, USA, Sept. 24, 2015. Available online at https://youtu.be/nXaxk27zwlk.

Miscellany III

[11] Dmitri Gribenko. Parsing Documentation Comments in Clang. LLVM Developers' Meeting, San Jose, CA, USA, Nov. 7–8, 2012. Video available at https://youtu.be/DzRq9Dy0b9c. Slides available at https://llvm.org/devmtg/2012-11/Gribenko_CommentParsing.pdf.