# Compilers 101

Debuggers

# Previously…

- Preprocessing
- Lexical analysis
- Syntax analysis
- Semantic analysis
- IR Generation
- IR Optimization
- Code generation
- Optimization

Fronten
d

Middle-end

Backend

# Why debugging?

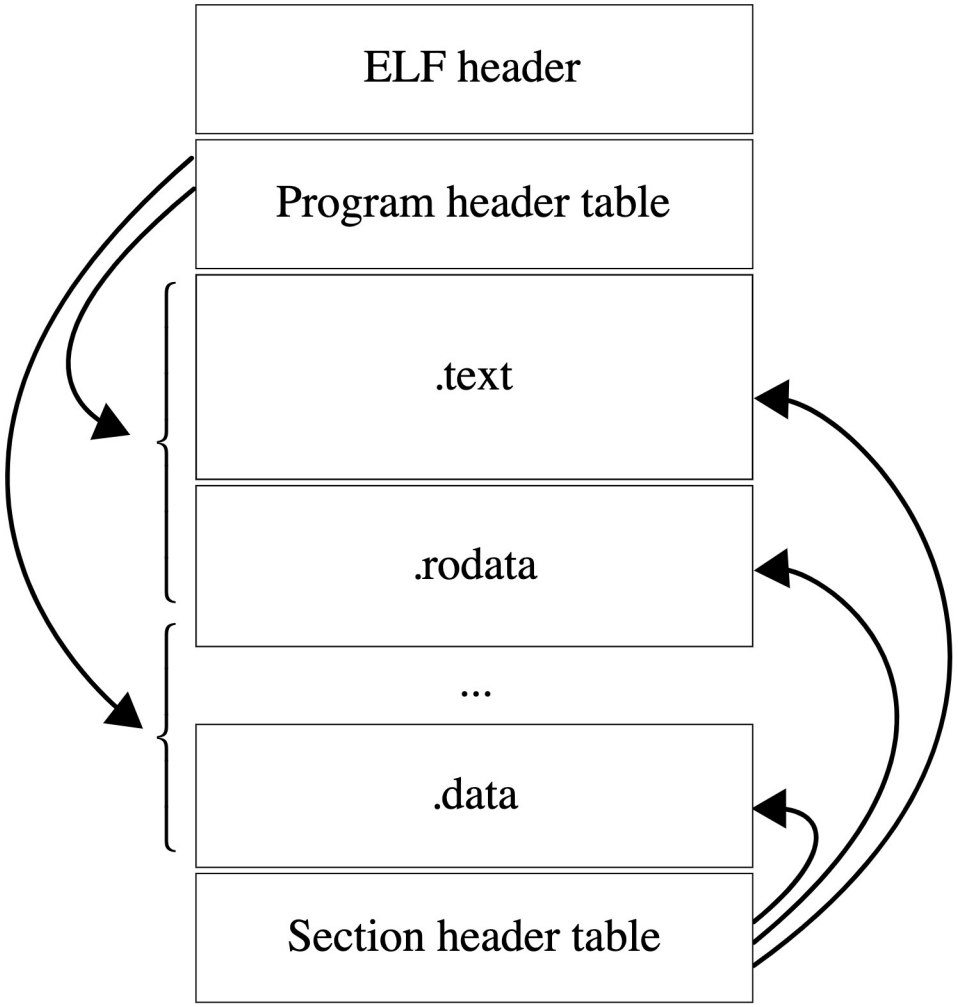
- Static analysis does not discover many kinds of errors (especially, logic errors)
- Retrieve runtime information
- Need some insight into running programs
- Allow to change execution flow without recompilation

# Debuggers in a nutshell

- Ability to control execution
  - Resume after signal/trap
- Ability to read/write memory
  - Registers and RAM
- Mapping from binary code to source

# Executable and linkable format (ELF)

- ELF is a common executable file format for Unix-like systems
- File is divided in multiple sections
- Sections can be read-only and executable




https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

# DWARF

- DWARF is a widely used debugging information format
- DWARF uses Debugging Information Entry (DIE) data structure
  - A DIE has a tag (DW_TAG_variable, DW_TAG_pointer_type, DW_TAG_subprogram)
  - And attributes (key-value pairs)
- DIE attributes can reference other DIEs

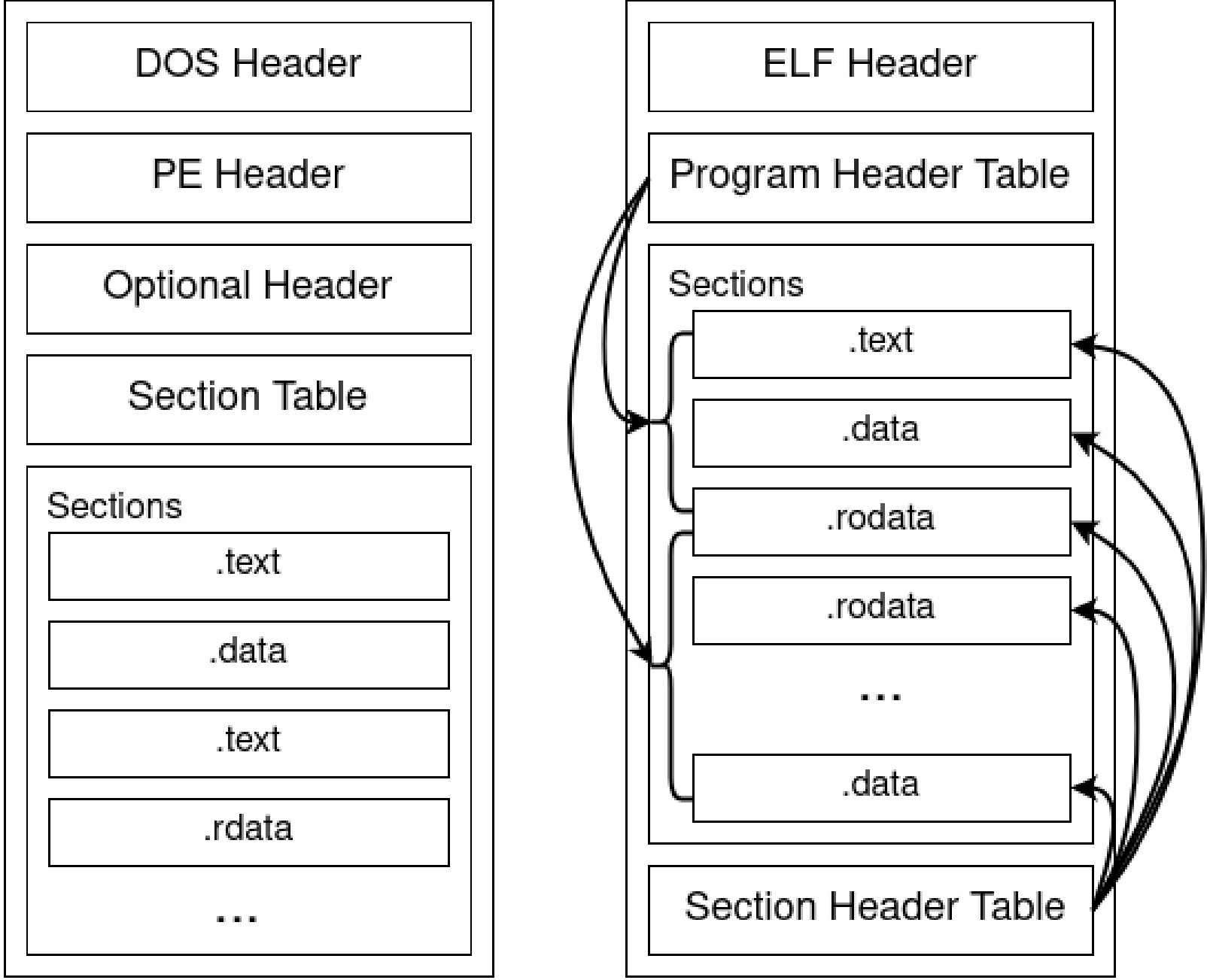
# Working with DWARF

- Use -g flag to enable DWARF in the compiler
- GDB and LLDB are the most used debuggers on Unix-like platfroms
- libdwarf – C library for working with DWARF ( http://www.prevanders.net/dwarf.html)
- dwex – GUI for visualizing DWARF (https://github.com/sevaa/dwex)

# PE and PDB

- Portable Executable (PE) is an executable file format on Windows
- Program database (PDB) is a debug info file format on Windows
- PE is very much like ELF
- Unlike DWARF, PDB is typically stored as an external file

# ELF and PE


DOS Header
PE Header
Optional Header
Section Table
Sections
.text
.data
.text
.rdata
...
(a) WinPE
ELF Header
Program Header Table
Sections
.text
.data
.rodata
.rodata
...
.data
Section Header Table
(b) ELF

# From LLVM IR to DWARF

```
1  // Type your code here, or load an example.
2  int square(int num) {
3      return num * num;
4  }
```

LLVM IR:

https://godbolt.org/z/ecn15d566

Assembler:

https://godbolt.org/z/oW19bY35E

```
!llvm.ident = !{!9}

!0 = distinct !DICompileUnit(language: DW_LANG_C_plus_plus_14, file: !1, producer:
!1 = !DIFile(filename: "/app/example.cpp", directory: "/app", checksumkind: CSK_MD
!2 = !{i32 7, !"Dwarf Version", i32 5}
!3 = !{i32 2, !"Debug Info Version", i32 3}
!4 = !{i32 1, !"wchar_size", i32 4}
!5 = !{i32 7, !"PIC Level", i32 2}
!6 = !{i32 7, !"PIE Level", i32 2}
!7 = !{i32 7, !"uwtable", i32 2}
!8 = !{i32 7, !"frame-pointer", i32 2}
!9 = !{!"clang version 15.0.0 (https://github.com/llvm/llvm-project.git cac19f4141
!10 = distinct !DISubprogram(name: "square", linkageName: "_Z6squarei", scope: !11
!11 = !DIFile(filename: "example.cpp", directory: "/app", checksumkind: CSK_MD5, c
!12 = !DISubroutineType(types: !13)
!13 = !{!14, !14}
!14 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!15 = !{}
!16 = !DILocalVariable(name: "num", arg: 1, scope: !10, file: !11, line: 2, type:
!17 = !DILocation(line: 2, column: 16, scope: !10)
!18 = !DILocation(line: 3, column: 12, scope: !10)
!19 = !DILocation(line: 3, column: 18, scope: !10)
!20 = !DILocation(line: 3, column: 16, scope: !10)
!21 = !DILocation(line: 3, column: 5, scope: !10)
```

# ptrace

**SYNOPSIS** top

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

- Attach to process
- Read/write registers
- Read/write memory
- Signal on traps
- Trace syscalls (emulated capability)

https://man7.org/linux/man-pages/man2/ptrace.2.html

# Debugger Engine

- Debugger Engine provides an interface for examining and manipulating running processes
- Debugger Engine can be used to both write debugger extensions (e.g., for WinDbg) and full-featured debuggers
- Debugger Markup Language is similar to HTML, but for debug info
- Full docs: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-engine-and-extension-apis

# Debugger features

- Breakpoints
- Step-by-step execution
- Local variables overview

and many others…

# Breakpoints

- Essential debugging tool
- Two very different kind of breakpoints
  - Hardware – supported by CPU, limited number of BPs
  - Software – replace instruction at address with halt/trap/interrupt and then replace back with original instruction

```cpp
#include <iostream>

#define N 10

int main() {
    int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    for (int i = 0; i < N / 2; ++i) {
        int t = a[N - 1 - i];
        a[N - 1 - i] = a[i];
        a[i] = t;
    }
    for (int i = 0; i < N; ++i) {
        std::cout << a[i];
    }
    std::cout << '\n';
    return 0;
}
```

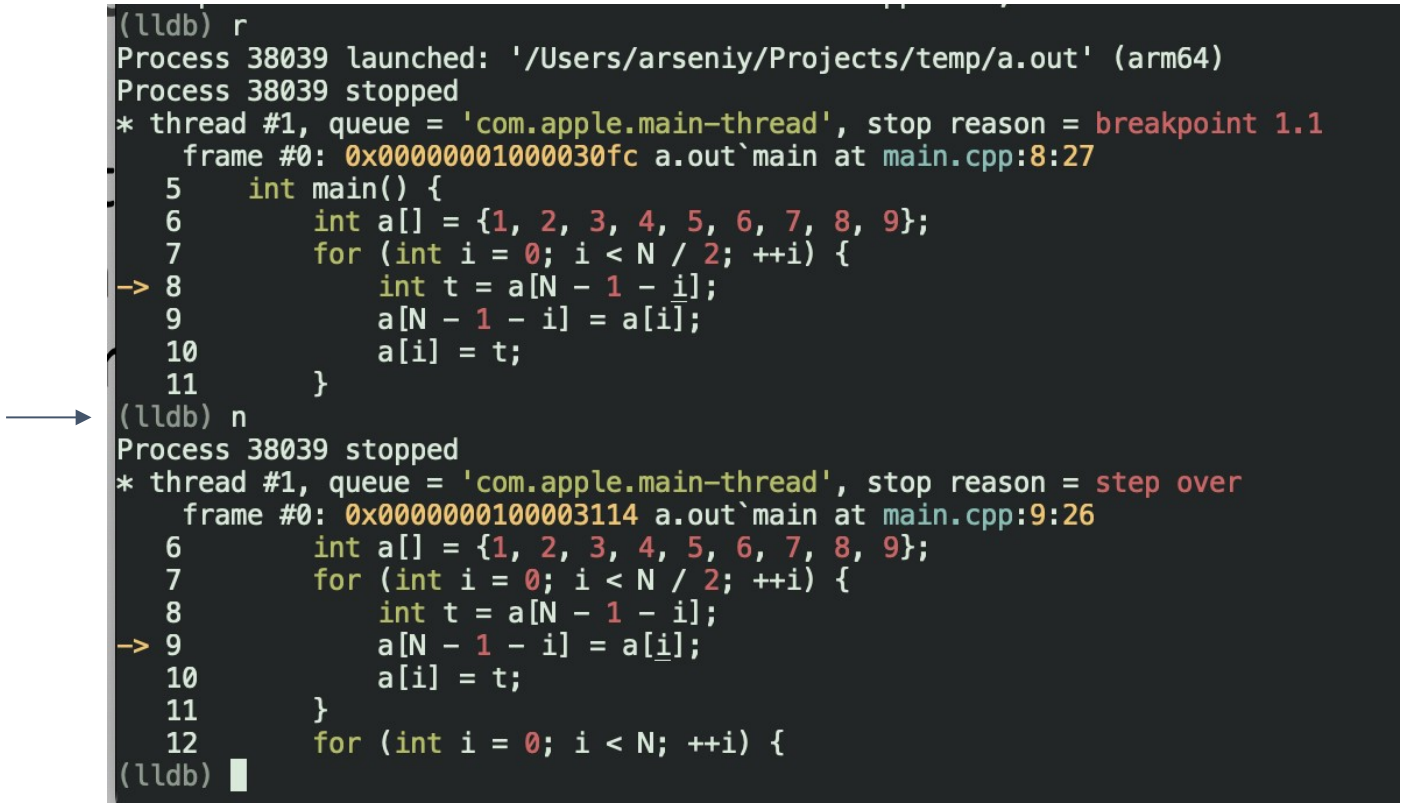
```
(lldb) breakpoint set -l 8
Breakpoint 1: where = a.out`main + 80 at main.cpp:8:26, address = 0x0000000100003118
```

```
GDB: break <file>:<line>
LLDB: breakpoint set -l <line>
```

# Step-by-step execution

Stepping commands let developers execute their program one line or instruction at a time. This helps in closely monitoring the changes in program state and variable values

GDB/LLDB:
`step`
`next`



# Inspect local variables and stack

See current values of the variables

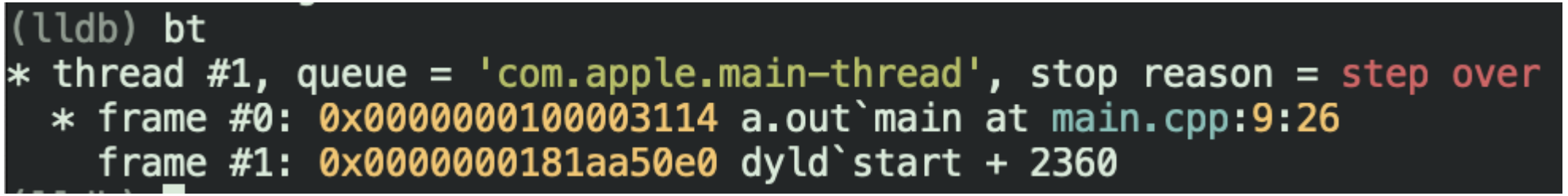
```
Process 38039 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100003114 a.out`main at main.cpp:9:26
   6           int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   7           for (int i = 0; i < N / 2; ++i) {
   8               int t = a[N - 1 - i];
-> 9               a[N - 1 - i] = a[i];
   10              a[i] = t;
   11          }
   12          for (int i = 0; i < N; ++i) {
```

GDB:
`print <variable>`
LLDB:
`frame variable [variable]`

```
(lldb) frame variable
(int[9]) a = ([0] = 1, [1] = 2, [2] = 3, [3] = 4, [4] = 5, [5] = 6, [6] = 7, [7] = 8, [8] = 9)
(int) i = 0
(int) t = 1486422108
```

# Stack trace

Stack tracing provides a look at the function call stack at any point in a program's execution. This is useful for understanding the sequence of function calls leading to the current point.




GDB/LLDB:
backtrace
bt

# Watchpoints

Watchpoints are similar to breakpoints but are triggered by changes in the value of a variable rather than the execution of a specific line of code.

```
GDB:
watch <variable>
LLDB:
watchpoint set variable
<variable>
```

```
arseniy@Arseniys-MacBook-Pro:~/Projects/temp$ lldb ./a.out
(lldb) target create "./a.out"
Current executable set to '/Users/arseniy/Projects/temp/a.out' (arm64).
(lldb) b main
Breakpoint 1: where = a.out`main + 48 at main.cpp:6:9, address = 0x00000001000030dc
(lldb) r
Process 44495 launched: '/Users/arseniy/Projects/temp/a.out' (arm64)
Process 44495 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000030dc a.out`main at main.cpp:6:9
   3    #define N 10
   4
   5    int main() {
-> 6        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   7        for (int i = 0; i < N / 2; ++i) {
   8            int t = a[N - 1 - i];
   9            a[N - 1 - i] = a[i];
(lldb) n
Process 44495 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000030e0 a.out`main at main.cpp:7:14
   4
   5    int main() {
   6        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
-> 7        for (int i = 0; i < N / 2; ++i) {
   8            int t = a[N - 1 - i];
   9            a[N - 1 - i] = a[i];
   10           a[i] = t;
(lldb) watchpoint set variable a[5]
Watchpoint created: Watchpoint 1: addr = 0x16fdfee78 size = 4 state = enabled type = w
    declare @ '/Users/arseniy/Projects/temp/main.cpp:6'
    watchpoint spec = 'a[5]'
    new value: 6
(lldb) c
Process 44495 resuming

Watchpoint 1 hit:
old value: 6
new value: 5
Process 44495 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = watchpoint 1
    frame #0: 0x0000000100003128 a.out`main at main.cpp:10:16
   7        for (int i = 0; i < N / 2; ++i) {
   8            int t = a[N - 1 - i];
   9            a[N - 1 - i] = a[i];
-> 10           a[i] = t;
   11       }
   12       for (int i = 0; i < N; ++i) {
   13           std::cout << a[i];
```

# Conditional breakpoints

These are breakpoints that are triggered only if a specified condition is true

Condition is checked every time when program reaches particular line of code

```
GDB:
break [location] if [condition]
LLDB:
breakpoint set --name [function] --condition
'[condition]'
```

# Modifying Program State

Debuggers often allow altering the state of the program, such as changing variable values or jumping to different points in the code.

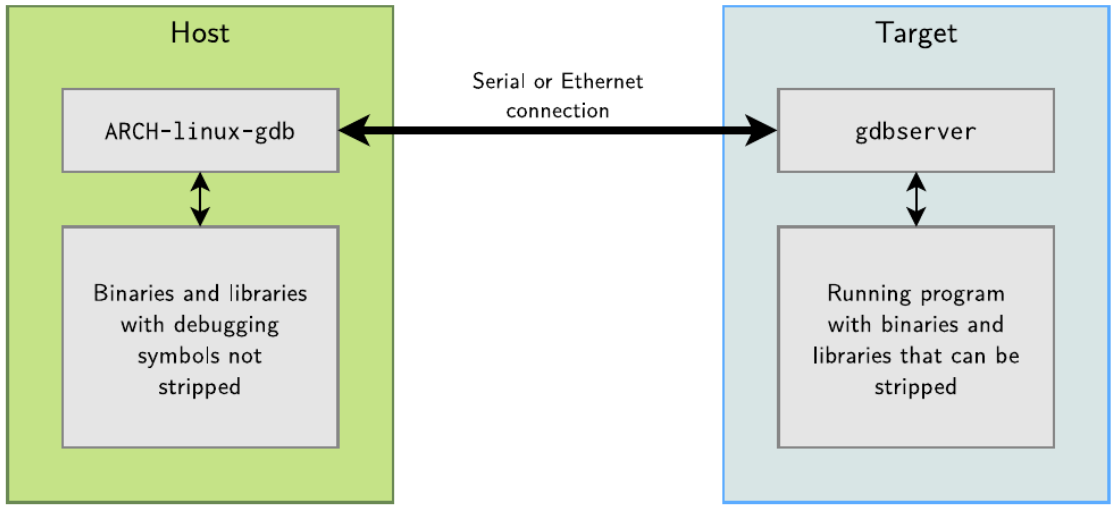
```
GDB:
set var <variable>=<value>
LLDB:
expression <variable> = <value>
```

# Modifying Program State

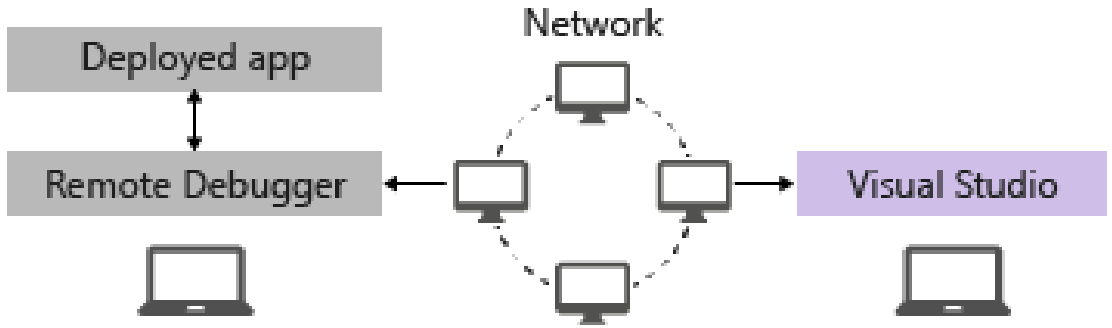
```
(lldb) r
There is a running process, kill it and restart?: [Y/n] y
Process 44495 exited with status = 9 (0x00000009) killed
Process 54529 launched: '/Users/arseniy/Projects/temp/a.out' (arm64)
Process 54529 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000030dc a.out`main at main.cpp:6:9
   3    #define N 10
   4
   5    int main() {
-> 6        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
   7        for (int i = 0; i < N / 2; ++i) {
   8            int t = a[N - 1 - i];
   9            a[N - 1 - i] = a[i];
(lldb) n
Process 54529 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x00000001000030e0 a.out`main at main.cpp:7:14
   4
   5    int main() {
   6        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
-> 7        for (int i = 0; i < N / 2; ++i) {
   8            int t = a[N - 1 - i];
   9            a[N - 1 - i] = a[i];
   10           a[i] = t;
(lldb) frame variable a
(int[9]) a = ([0] = 1, [1] = 2, [2] = 3, [3] = 4, [4] = 5, [5] = 6, [6] = 7, [7] = 8, [8] = 9)
(lldb) expression a[5] += 100500
(int) $0 = 100506
(lldb) frame variable a
(int[9]) a = ([0] = 1, [1] = 2, [2] = 3, [3] = 4, [4] = 5, [5] = 100506, [6] = 7, [7] = 8, [8] = 9)
```

# Remote debugging

This feature enables the debugging of a program running on a different machine than the debugger, which is useful for testing in different environments or on different hardware.

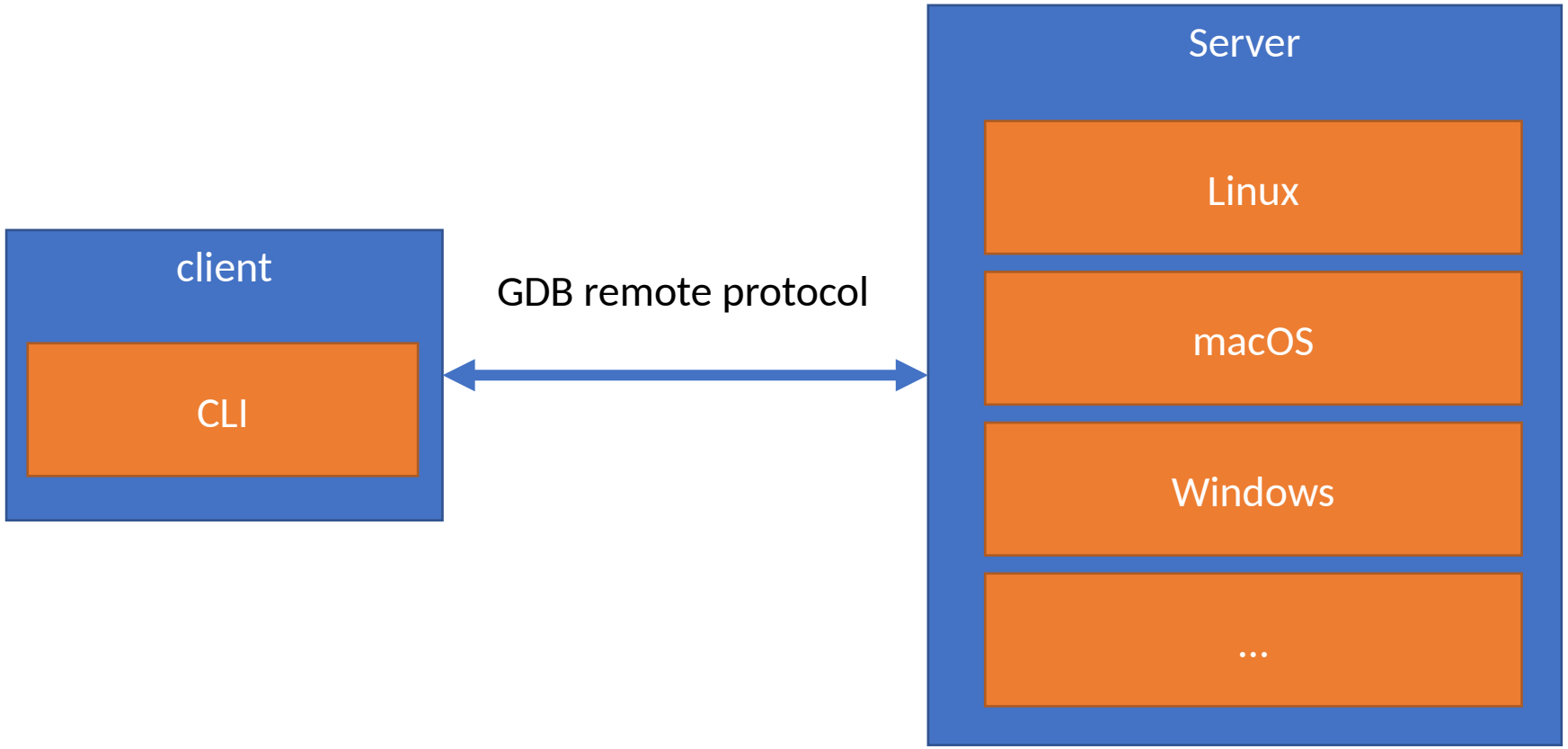



gdb/lldb network debugging




MSVC network debugging

# LLDB network debugging architecture



# Expression evaluation

- Parsing programming languages is still a challenge for debuggers
  - Hard to keep up with all new features
- For C++ LLDB uses a full Clang instance
  - Generate AST for given expression and try to generate a DWARF expression or JIT code

# GDB remote protocol

- Exchange textual messages in the format

  -> $*packet-data#checksum*

  <- +
- Checksum is modulo 256 sum of all characters between $ and #
- Most common packets
  - `? – query reason for halt`
  - `b addr,mode – set breakpoint`
  - `c addr – continue at addr`
  - `g – read general registers`
  - `g XX... – write general registers`
  - `m addr,length - read memory`
  - `m addr,length:XX… - write memory`

https://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html

# Python interface

- LLDB has flexible scripting facilities
  - Interfaces to control entire debugging session
  - Custom debugger commands
  - Pretty printers
- Customize debugger to support your data structures

# Time travel

- Time travel debugging is the ability to step back one or more instructions
- Basic principle: save state in particular points of program execution and restore it
- Typical implementation ideas:
  - Virtual machine, that saves the whole processor state
  - Save state on perf counters change only
  - Use hardware assistance: Intel Processor Trace (PT), ARM CoreSight
- Limitations:
  - Networking, GPUs, other peripherals
  - Multithreading

# More useful materials on LLDB

LLDB tutorial: https://lldb.llvm.org/use/tutorial.html

GDB to LLDB commands mapping: https://lldb.llvm.org/use/map.html

# Test

https://forms.gle/ESvwo5dfw9drBFvk7

Submission time: **10 minutes**

В чем заключается принцип работы отладчика?

Your answer

На каком этапе компиляции генерируется информация, необходимая для правильной работы отладчика?

Your answer

Backup: me@gooddoog.ru

# Extra materials

- Greg Law "Give me 15 minutes & I'll change your view of GDB” - https://www.youtube.com/watch?v=PorfLSr3DDI
- LLVM Developers’ Meeting: R. Isemann “Better C++ debugging using Clang Modules in LLDB” - https://www.youtube.com/watch?v=vuNZLlHhy0k
- 2015 EuroLLVM Developers’ Meeting: “Why should I use LLDB?”- https://www.youtube.com/watch?v=JtpQZw9NpIU