SOURCE-LEVEL DEBUGGING OF C++ CODES WITH THE LLDB SOFTWARE DEBUGGER

Till Ehrengruber

Department of Mathematics
ETH Zürich
Zürich, Switzerland

1. ABSTRACT

A short, practice oriented guide to debugging C++ programs with the LLDB debugger is given. Key functionalities of a source-level debugger for debugging single-threaded C++ applications are explained briefly with their corresponding commands in LLDB, and supplemented with various examples.

2. INTRODUCTION

LLDB is a free and open-source software debugger developed as part of the LLVM project. It uses various LLVM libraries (e.g. Clang) to provide a reusable set of components for software debugging. We will use the command-line frontend of LLDB throughout the tutorial to explore some of the available functionality. Pre-built binaries for LLDB are available for most modern operating systems. We also recommend to install the C++ compiler and standard library implementation of the LLVM project, namely Clang and libc++, since they generate binaries that show slightly better compatibility with LLDB.

Debian (Ubuntu, Linux Mint)

```
sudo apt-get install LLDB

# optional
sudo apt-get install clang libc++-dev
```

RedHat (Fedora, CentOS)

```
sudo yum install LLDB
```

Mac OS X
Install the `command-line tools` package in Xcode.

Fig. 1: Installation instructions for Debian and RedHat based Linux distributions

Please note, that several IDEs[1] exist that provide graphical frontends for LLDB, GDB (a debugger developed by the GNU Project) or other debuggers.

– *CLion*[2] (commercial, but free for open-source projects and students, cross-platform, LLDB, GDB)
– *XCode*[3] (free, Mac OS X, LLDB)
– *Eclipse*[4] (open-source, cross-platform, GDB)
– *QtCreator*[5] (free, cross-platform, GDB, LLDB)
– *Visual Studio*[6] (free, Windows, integrated debugger)

Fig. 2: Incomplete list of IDEs with graphical debugger frontends

3. GETTING STARTED

To start debugging, additional information for the target program must be generated at compile time. The additional information allows the debugger to match single source lines with its assembly code. Please note that enabling compiler optimization does in most cases break the integrity of the information, since program order is not always preserved, and as such only a subset of the available functionality can be used. The author therefore recommends to disable all optimizations for debugging purposes.

Default C++ compiler

```
$ c++ -g ex1.cpp -o ex1
```

Clang with libc++

```
$ clang++ -stdlib=libc++ -g ex1.cpp -o ex1
```

Fig. 3: Compiler invocation with debug information included

[1] integrated development environment

[2] `https://www.jetbrains.com/clion/`
[3] `https://developer.apple.com/xcode/`
[4] `https://eclipse.org/`
[5] `https://www.qt.io/ide/`
[6] `https://www.visualstudio.com/`

The required debug information is generated and included in the compiled binary by appending `-g` to the compiler invocation or, for programs that use the CMake build system, by appending `-DCMAKE_BUILD_TYPE=Debug` to the invocation of `cmake`.

Every time LLDB is launched a command prompt appears at which LLDB's internal commands can be used. Help for these commands is obtained by prefixing the command with `help`. Furthermore, recently used commands are accessible through the ↑ and ↓ keys. Pressing *Enter* evaluates the last command again.

```
$ LLDB
(LLDB) help quit
   Quit out of the LLDB debugger.

Syntax: quit
(LLDB)
```

4. COMMAND REFERENCE

4.1. Stepping Through the Source Code

This section covers the operations used to pause and resume the flow of execution of the target program.

Breakpoints mark source locations where the debugger will pause execution. Note that additional constraints are possible.

`breakpoint set ...`
Set a breakpoint.

```
# break at the beginning of function main
breakpoint set -n main
# break at line 13 of file ex1.cpp
breakpoint set --file ex1.cpp --line 13
# break at the beginning of function main if
    ↳ foo.bar() evaluates to true
breakpoint set -n main -c foo.bar()
```

`breakpoint delete <id>`
Delete the specified breakpoint.

```
# break at the beginning of function main
breakpoint delete 1
```

`breakpoint enable <id>`
Enable the specified breakpoint.
`breakpoint disable <id>`
Disable the specified breakpoint.
`breakpoint clear`
Delete all breakpoints.
`breakpoint list`
List all breakpoints with their ids.

Single-stepping. A typical debugging pattern is as follows: after a breakpoint is hit, one moves through the code line-by-line while observing the program's behaviour. This so-called single stepping is accomplished through one of the following commands.

`thread step-over`
Execute the next statement, then pause.
`thread step-in`
If the statement is (or contains) a function call, resume execution and pause at the beginning of the function. Otherwise execute the next statement and then pause.
`thread step-out`
Finish execution of the current procedure, then pause at the return address of the current stack frame.
`thread until <num_lines>`
Execute until the current stack frame reaches the specified line number, then pause.

Continue execution.

`thread continue`
Continue execution until the next breakpoint is hit or the program terminates.

4.2. Accessing the Call Stack

`thread backtrace`
Print all stack frames on the call stack.
`frame select <frame-index>`
Select the specified frame. The current frame is indexed by 0.
`frame variables`
Print all local variables.
`up`
Go up the specified number of frames.
`down`
Go up the specified number of frames.

4.3. Evaluating Expressions, Inspecting Variables

LLDB's usage of the Clang compiler for JIT[7] code generation enables evaluation of complex C++ commands [8] while debugging.

[7]Just-in-time

[8]Templated code is not possible yet.

`expression`

Evaluate the specified C++ expression in the current context and output the result.

```
# define integer var with value 5
expression int var=5
# increment var by 1
expression var++
# call function func
expression func(10)
# print integer values of vec[2:4]
expression for (int i=2; i<4; ++i) (int) printf
    ↳ ("%i\n", vec[i])
```

`print`

Print the result of the specified expression (essentially just an alias of `expression`).

```
# print contents of var
print var
```

5. INTRODUCTORY DEBUGGING SESSION

We will now start with a complete debugging session of Example 1. The code is fairly simple and self-explanatory. Our goal is to use LLDB to locate and fix the bug on line 16.

We start by compiling the code.

```
$ c++ -g -o ex1 ex1.cpp
```

After compilation has finished, we start LLDB with the target executable as a parameter.

```
$ LLDB ex1
(LLDB) target create "ex1"
Current executable set to 'ex1' (x86_64).
```

Before we start debugging, we launch the program once to see the incorrect output.

```
(LLDB) run
Process 2086 launching
Process 2086 launched: '/debugging/example/ex1' (
    ↳ x86_64)
1
Process 2086 exited with status = 0 (0x00000000)
```

The process terminates normally and prints `1` to the console, which is obviously not the mean value of `a` in double precision arithmetic. Under the assumption that we do not know the location of the error, a good start is to set a breakpoint at the beginning of the main function

```
(LLDB) break set -n main
```

and rerun the program.

```
(LLDB) run
Process 2193 launching
```

```
1   #include <iostream>
2   #include <vector>
3
4   // calculate the sum of vec
5   int sum(const std::vector<int>& vec) {
6     int result = 0;
7     for (unsigned i=0; i<vec.size(); ++i) {
8       result += vec[i];
9     }
10    return result;
11  };
12
13  // calculate the mean of vec in double precision
      ↳ arithmetic
14  double mean(const std::vector<int>& vec) {
15    return sum(vec)/vec.size();
16              // bug: unintended round off
17  }
18
19  int main () {
20    // init vector filled with ones
21    std::vector<int> a(10, 1);
22    // set first element of a to 10
23    a[0]=10;
24    // output result
25    std::cout << mean(a)
26              << std::endl;
27  }
```

Example 1: Calculating the mean value of a vector of integers

```
Process 2193 launched: '/debugging/example/ex1' (
    ↳ x86_64)
Process 2193 stopped
* thread #1: tid = 2193, 0x0000000000400c2c ex1`
    ↳ main + 12 at ex1.cpp:21, name = 'ex1',
    ↳ stop reason = breakpoint 1.1
    frame #0: 0x0000000000400c2c ex1`main + 12 at
        ↳  ex1.cpp:21
   18
   19   int main () {
   20     // init vector filled with ones
-> 21     std::vector<int> a(10, 1);
   22     // set first element of a to 10
   23     a[0]=10;
   24     // output result
```

Line 21 marks the first statement of the `main` function and thus, execution is paused there as desired. Curious about the value of `a` we step over the line 21 and 23

```
(LLDB) thread step-over
Process 2379 stopped
* thread #1: tid = 2379, 0x0000000000400c5d ex1`
    ↳ main + 61 at ex1.cpp:23, name = 'ex1',
    ↳ stop reason = step over
    frame #0: 0x0000000000400c5d ex1`main + 61 at
        ↳  ex1.cpp:23
   20     // init vector filled with ones
   21     std::vector<int> a(10, 1);
   22     // set first element of a to 10
-> 23     a[0]=10;
   24     // output result
   25     std::cout << mean(a)
   26               << std::endl;
(LLDB) thread step-over
Process 2379 stopped
* thread #1: tid = 2379, 0x0000000000400c7d ex1`
    ↳ main + 93 at ex1.cpp:25, name = 'ex1',
    ↳ stop reason = step over
    frame #0: 0x0000000000400c7d ex1`main + 93 at
        ↳  ex1.cpp:25
   22     // set first element of a to 10
   23     a[0]=10;
   24     // output result
-> 25     std::cout << mean(a)
   26               << std::endl;
   27   }
```

and print its contents[9].

```
(LLDB) print a
(std::__1::vector<int, std::__1::allocator<int>
    ↳ >) $0 = size=10 {
  [0] = 10
  [1] = 1
  [2] = 1
  [3] = 1
  [4] = 1
  [5] = 1
  [6] = 1
  [7] = 1
  [8] = 1
  [9] = 1
}
```

The contents of `a` are as expected, so we move on with inspecting the inner workings of the `mean` function.

```
(LLDB) thread step-in
Process 2518 stopped
* thread #1: tid = 2518, 0x000000000040116c ex1`
    ↳ mean(vec=0x00007fffffffe148) + 12 at ex1.
    ↳ cpp:15, name = 'ex1', stop reason = step
    ↳ in
    frame #0: 0x000000000040116c ex1`mean(vec=0
        ↳ x00007fffffffe148) + 12 at ex1.cpp:15
   12
   13   // calculate the mean of vec in double
      ↳ precision arithmetic
   14   double mean(const std::vector<int>& vec)
      ↳ {
-> 15     return sum(vec)/vec.size();
   16              // bug: unintended round off
   17   }
   18
```

The function `mean` only contains a division, so let us see whether the operands are correct by evaluating them directly.

```
(LLDB) expression sum(vec)
(int) $0 = 19
(LLDB) expr vec.size()
error: call to a function 'std::__1::vector<int,
    ↳ std::__1::allocator<int> >::size() const'
    ↳ ('
    ↳ _ZNKSt3__16vectorIiNS_9allocatorIiEEE4sizeEv
    ↳ ') that is not present in the target
error: The expression could not be prepared to
    ↳ run in the target
```

We see the sum of all values in `vec` is 19 and thus correct, but for the evaluation of the divisor an error is raised[10]. The error already points to the location of the problem, namely that there is no entry in the symbol table for the called function. The reason for this is that the function `std::vector<int>::size()` is implemented as an inline function and thus not directly callable. To the knowledge of the author, there is currently no reliable way to circumvent this. Since the `size` function is simple and part of the C++ standard library, we can safely continue our examination. For functions where this is not the case we either have to remove the inline declaration or step into the function and observe the result there. As both operands are fine, we conclude that the bug must be located in the division itself. Indeed we see that we unintentionally performed an integer division. Fixing the bug by replacing line 15 with

```
return double(sum(vec))/vec.size();
```

gives the correct result. Without needing to close LLDB, we compile the corrected program again (e.g. in a different terminal window) and launch it in LLDB.

6. INSPECT CRASHES

With an attached debugger, abnormal program termination can be analysed in great detail. When an exception occurs the processor halts execution and hands over control to the operating system. The operating system then generates a signal corresponding to this type of exception[11] and sends it to LLDB.

– Memory access violation
– Integer overflow
– Floating-point overflow or underflow
– Integer divide-by-zero
– Floating-point divide by zero
– Illegal instruction
– Data misalignment
– Access violation

Fig. 4: List of common exceptions

The information about the received signal in combination with a backtrace (the most recent frames of the C++ standard library can be ignored) can give valuable information about the origin of the problem. If the information obtained is not sufficient to fix the error, or more detailed

[9]LLDB abstracts away the inner structure of STL containers with data formatters written in Python and prints only the values of the container's elements.

[10]Note that this error will not occur if compiled with GNU GCC's implementation of the C++ standard library, but the general problem persists.

[11]The full list of signals and their meaning are specified by the operating system. For Linux see `http://man7.org/linux/man-pages/man7/signal.7.html`.

analysis is desired, rerunning the program with additional breakpoints at locations can give further insight.

Note that the probability of introducing bugs that trigger undefined behaviour in locations unrelated to the functionality they are embedded in should be minimized by good program design principles, assertion of invariants, and rigorous testing.

We continue with debugging Example 2. The code does not serve any particular purpose, but illustrates the general behaviour, when an exception occurs. In this example the null-pointer is dereferenced, which leads to a `SEGFAULT` and subsequent termination of the program.

```
int main() {
    int *ptr = nullptr;
    *ptr = 0;

    return 0;
}
```

Example 2: Dereferencing of a null-pointer

We compile the source code and launch the program first without an attached debugger.

```
$ c++ -g ex2.cpp -o ex2_nullptr
$ ./ex2_nullptr
[1]    23120 segmentation fault (core dumped)  ./
    ↳ ex2_nullptr
```

The program crashes and the type of the exception is printed. While the type of the exception gives a general hint, the lack of detailed information about the origin makes it hard to locate the error in more complex programs. We therefore start the program again, but this time with LLDB attached.

```
(lldb) target create "ex2_nullptr"
Current executable set to 'ex2_nullptr' (x86_64).
(lldb) run
Process 23195 launching
Process 23195 launched: '/debugging/example/
    ↳ ex2_nullptr' (x86_64)
Process 23195 stopped
* thread #1: tid = 23195, 0x00000000004005d9 ex2`
    ↳ main + 25 at ex2_nullptr.cpp:3, name = '
    ↳ ex2', stop reason = invalid address (fault
    ↳  address: 0x0)
    frame #0: 0x00000000004005d9 ex2`main + 25 at
        ↳  ex2.cpp:3
   1    int main() {
   2        int *ptr = nullptr;
-> 3        *ptr = 0;
   4
   5        return 0;
   6    }
```

Now LLDB will print the exact position where the exception occurred and gives additional information.

7. LOCATING INFINITE LOOPS

Unintended infinite loops are a common type of bug that occurs frequently if complex loop conditions are used. Since this type of bug can quickly become really hard to find without a debugger, this debugging session was added. Example 3 shows a manufactured example for this type of bug.

```
int main () {
  int result;
  for (int i=1; i!=10; i+=2) {
        result = i;
  }
}
```

Example 3: Manufactured example of an infinite loop

We will start as before by compiling the code and launching LLDB.

```
$ LLDB ex2_endless_loop
(LLDB) target create "ex3_infinite_loop"
Current executable set to 'ex3_infinite_loop' (
    ↳ x86_64).
```

However, due to the faulty loop the program does not terminate and thus we are not dropped to LLDB's internal command prompt as in the previous examples.

```
(LLDB) run
Process 3840 launching
Process 3840 launched: '/debugging/example/
    ↳ ex3_infinite_loop' (x86_64)
```

After waiting for a moment we expect that the program reached the endless loop. We therefore interrupt the running process by pressing *Ctrl + C*, causing the current terminal to send a `SIGSTOP` signal to LLDB, which then interrupts the currently running process. Since this is a very simple example, execution will be paused directly inside the loop and thus we already found the problem. Note that in a more complex case, execution might be paused in some subroutine called inside the endless loop and additional insight about the call stack must be collected.

8. INSPECT MATRICES AND VECTORS OF THE EIGEN C++ TEMPLATE LIBRARY

Eigen is a C++ template library for linear algebra. Its usage of intelligent compile-time mechanisms like expression templates allows for efficient linear algebra operations while retaining code readability.

In this section we will use LLDB to inspect the contents of dense matrices and vectors of the Eigen library. The code for this debugging session can be found in Example 4. The example is contrary to previous ones free of bugs and performs a simple matrix vector product of the form $Ax = b$

```cpp
1  #include <iostream>
2  #include <Eigen/Dense>
3
4  int main() {
5    // A = [ 1 0 0;
6    //       0 2 0;
7    //       0 0 3 ]
8    Eigen::Matrix<double, 3, 3> A;
9    A << 1, 0, 0,
10        0, 2, 0,
11        0, 0, 3;
12   // x = [1 1 1]
13   Eigen::Matrix<double, 3, 1> x;
14   x.setConstant(1);
15
16   // compute A*x
17   Eigen::Matrix<double, 3, 1> b1 = A * x;
18   auto b2 = A * x;
19
20   return 0;
21 }
```

Example 4: Calculation of a matrix vector product

with $A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$ and $x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$.

Our aim is to inspect the result of the matrix vector product performed on line 17 as well as on line 18 and to verify that it indeed gives the correct result of $b = (1, 2, 3)^T$. Note that this is a manufactured example, where the result could also be verified by just printing the contents of `b1` and `b2`. However we explicitly want to avoid this to illustrate the interactivity that the usage of a debugger gives, since adding trace code in more complex program can be very time- consuming.

We start as before by compiling the code, launching LLDB, setting a breakpoint at the beginning of the main function,

```
$ c++ -g -std=c++11 -I/usr/include/eigen3 main.
    ↳ cpp -o ex4_eigen
$ lldb ex4_eigen
(lldb) target create "ex4_eigen"
Current executable set to 'ex4_eigen' (x86_64).
(lldb) breakpoint set -n main
Breakpoint 1: where = main`main + 22 at main.cpp
    ↳ :12, address = 0x0000000000400946
```

and starting execution.

```
(lldb) run
Process 17245 launching
Process 17245 launched: '/home/tehrengruber/
    ↳ polybox/Uni/ETHZ/Semester 4/Numerical
    ↳ Methods for Partial Differential Equations
    ↳ /Debugging/example/data_formatter/main' (
    ↳ x86_64)
Process 17245 stopped
* thread #1: tid = 17245, 0x0000000000400946 main
    ↳ `main + 22 at main.cpp:12, name = 'main',
    ↳ stop reason = breakpoint 1.1
    frame #0: 0x0000000000400946 main`main + 22
        ↳ at main.cpp:12
   9      // A = [ 1 0 0;
   10     //       0 2 0;
   11     //       0 0 3 ]
-> 12     Eigen::Matrix<double, 3, 3> A;
   13     A << 1, 0, 0,
   14          0, 2, 0,
   15          0, 0, 3;
```

Printing the value of `A` reveals the inner structure of the `Eigen::Matrix` instance.

```
(lldb) print A
(Eigen::Matrix<double, 3, 3, 0, 3, 3>) $2 = {
  Eigen::PlainObjectBase<Eigen::Matrix<double, 3,
      ↳ 3, 0, 3, 3> > = {
    m_storage = {
      m_data = {
        array = ([0] = 6.9533558074437043E-310,
            ↳ [1] = 2.0732936177486861E-317, [2]
            ↳ = 9.8813129168249309E-324, [3] =
            ↳ 2.0796433494290378E-317, [4] =
            ↳ 2.073297076208207E-317, [5] = 0,
            ↳ [6] = 2.079605306374308E-317, [7]
            ↳ = 2.073297076208207E-317, [8] =
            ↳ 6.9533558074579334E-310)
      }
    }
  }
}
```

We see that the matrix is not yet initialized and is as such filled with arbitrary values. We now continue execution until line 13 is reached and print the contents of `A` again.

```
(lldb) thread until 13
Process 17382 resuming
Process 17382 stopped
* thread #1: tid = 17382, 0x0000000000400ae7 main
    ↳ `main + 439 at main.cpp:13, name = 'main',
    ↳ stop reason = step until
    frame #0: 0x0000000000400ae7 main`main + 439
        ↳ at main.cpp:13
   10          0, 2, 0,
   11          0, 0, 3;
   12     // x = [1 1 1]
-> 13     Eigen::Matrix<double, 3, 1> x;
   14     x.setConstant(1);
   15
   16     // compute A*x
(lldb) print A
(Eigen::Matrix<double, 3, 3, 0, 3, 3>) $0 = {
  Eigen::PlainObjectBase<Eigen::Matrix<double, 3,
      ↳ 3, 0, 3, 3> > = {
    m_storage = {
      m_data = {
        array = ([0] = 1, [1] = 0, [2] = 0, [3] =
            ↳ 0, [4] = 2, [5] = 0, [6] = 0, [7]
            ↳ = 0, [8] = 3)
      }
    }
  }
}
```

Now A is initialized and we see that the contents are stored in column-major order. While printing and inspecting the inner structure of the `Eigen::Matrix` object reveals all information, its readability is rather poor. LLDB addresses this by providing the possibility to add custom data formatters using its Python API[12] to print variables in a more human-readable format. We therefore install such a data formatter for matrices and vectors of the Eigen library[13].

```
$ bash -c "$(curl -fsSL https://raw.
    ↳ githubusercontent.com/tehrengruber/LLDB-
    ↳ Eigen-Data-Formatter/master/tools/install.
    ↳ sh)"
```

Fig. 5: Instructions for installing the LLDB data formatter for matrices and vectors of the Eigen library.

The output of the `print` command now looks much cleaner.

```
(lldb) p A
(Eigen::Matrix<double, 3, 3, 0, 3, 3>) $138 =
    ↳ rows: 3, cols: 3
[ 1 0 0;
  0 2 0;
  0 0 3 ]
```

We step forward to the end of the main function and print the contents of $b1$ and $b2$.

```
(lldb) p b1
(Eigen::Matrix<double, 3, 1, 0, 3, 1>) $143 =
    ↳ rows: 3, cols: 1
[ 1;
  2;
  3 ]
(lldb) p b2
(Eigen::CoeffBasedProduct<const Eigen::Matrix<
    ↳ double, 3, 3, 0, 3, 3> &, const Eigen::
    ↳ Matrix<double, 3, 1, 0, 3, 1> &, 6>) $148
    ↳ = {
  m_lhs = 0x00007fffffffe100 rows: 3, cols: 3
[ 1 0 0;
  0 2 0;
  0 0 3 ]

  m_rhs = 0x00007fffffffe070 rows: 3, cols: 1
[ 1;
  1;
  1 ]

  m_result = rows: 3, cols: 1
[                        0;
   6.953349068253761E-310;
  4.9406564584124654E-324 ]

}
```

For $b1$ the result is as expected $(1, 2, 3)^T$, but $b2$ shows different behaviour. As the type of $b2$ is not explicitly specified in the source code by using the `auto` keyword, its type[14] is not `Eigen::Matrix` as for $b1$, but `Eigen::CoeffBasedProduct`. This is an expression template of Eigen and its evaluation is delayed until the result is required. As such we don't see the final values in the `m_result` attribute. Sadly a generally applicable method to see intermediate results of Eigens expression templates does not exist yet, but in some cases the result can be computed by calling the `eval` method prior to printing.

9. CONCLUSIONS

While only a very brief introduction to debugging with LLDB was given, it already illustrates that interactivity in the debugging process is a powerful way to find and eliminate bugs. LLDB is especially useful for this by offering a modern, powerful, and extensible interface to the user. Additionally it has been shown that inspection of a program and analysis of the code flow even in the absence of bugs can give the user valuable understanding of the inner workings of their code. Advanced topics like watchpoints, core dumps, low-level insight, remote targets, and sophisticated debugging practices were skipped explicitly because their usage is either complicated and required rarely, or because the desired effects can be imitated with the already available functionality.

10. ACKNOWLEDGEMENTS

I would like to thank Nicolas Hafner, Baranidharan Mohan, and Eric Sinner for their feedback and corrections on this guide.

11. REFERENCES

Linux Tutorial - GNU GDB Debugger Command Cheat Sheet

[12]application programming interface

[13]For further information and detailed installation instructions, see `https://github.com/tehrengruber/LLDB-Eigen-Data-Formatter`

[14]Template parameters are omitted for readability