NAME

llvm-symbolizer - convert addresses into source code locations

SYNOPSIS

llvm-symbolizer [*options*] [*addresses...*]

DESCRIPTION

llvm-symbolizer reads input names and addresses from the command-line and prints corresponding source code locations to standard output. It can also symbolize logs containing *Symbolizer Markup* via *--filter-markup*.

If no address is specified on the command-line, it reads the addresses from standard input. If no input name is specified on the command-line, but addresses are, or if at any time an input value is not recognized, the input is simply echoed to the output.

Input names can be specified together with the addresses either on standard input or as positional arguments on the command-line. By default, input names are interpreted as object file paths. However, prefixing a name with **BUILDID:** states that it is a hex build ID rather than a path. This will look up the corresponding debug binary. For consistency, prefixing a name with **FILE:** explicitly states that it is an object file path (the default).

A positional argument or standard input value can be preceded by "DATA" or "CODE" to indicate that the address should be symbolized as data or executable code respectively. If neither is specified, "CODE" is assumed. DATA is symbolized as address and symbol size rather than line number.

llvm-symbolizer parses options from the environment variable **LLVM_SYMBOLIZER_OPTS** after parsing options from the command line. **LLVM_SYMBOLIZER_OPTS** is primarily useful for supplementing the command-line options when **llvm-symbolizer** is invoked by another program or runtime.

EXAMPLES

All of the following examples use the following two source files as input. They use a mixture of C-style and C++-style linkage to illustrate how these names are printed differently (see *--demangle*).

```
// test.h
extern "C" inline int foz() {
  return 1234;
}

// test.cpp
```

```
#include "test.h"
int bar=42;

int foo() {
  return bar;
}

int baz() {
  volatile int k = 42;
  return foz() + k;
}

int main() {
  return foo() + baz();
}
```

These files are built as follows:

```
$ clang -g test.cpp -o test.elf
$ clang -g -O2 test.cpp -o inlined.elf
```

Example 1 - addresses and object on command-line:

```
$ llvm-symbolizer --obj=test.elf 0x4004d0 0x400490
foz
/tmp/test.h:1:0

baz()
/tmp/test.cpp:11:0
```

Example 2 - addresses on standard input:

```
$ cat addr.txt
0x4004a0
0x400490
0x4004d0
$ llvm-symbolizer --obj=test.elf < addr.txt
main
/tmp/test.cpp:15:0
```

```
baz()
/tmp/test.cpp:11:0

foz
/tmp/./test.h:1:0
```

Example 3 - object specified with address:

```
$ llvm-symbolizer "test.elf 0x400490" "FILE:inlined.elf 0x400480"
baz()
/tmp/test.cpp:11:0

foo()
/tmp/test.cpp:8:10

$ cat addr2.txt
FILE:test.elf 0x4004a0
inlined.elf 0x400480

$ llvm-symbolizer < addr2.txt
main
/tmp/test.cpp:15:0

foo()
/tmp/test.cpp:8:10
```

Example 4 - BUILDID and FILE prefixes:

```
$ llvm-symbolizer "FILE:test.elf 0x400490" "DATA BUILDID:123456789abcdef 0x601028"
baz()
/tmp/test.cpp:11:0

bar
6295592 4

$ cat addr3.txt
FILE:test.elf 0x400490
DATA BUILDID:123456789abcdef 0x601028

$ llvm-symbolizer < addr3.txt
```

```
baz()
/tmp/test.cpp:11:0

bar
6295592 4
```

Example 5 - CODE and DATA prefixes:

```
$ llvm-symbolizer --obj=test.elf "CODE 0x400490" "DATA 0x601028"
baz()
/tmp/test.cpp:11:0

bar
6295592 4

$ cat addr4.txt
CODE test.elf 0x4004a0
DATA inlined.elf 0x601028

$ llvm-symbolizer < addr4.txt
main
/tmp/test.cpp:15:0

bar
6295592 4
```

Example 6 - path-style options:

This example uses the same source file as above, but the source file’s full path is /tmp/foo/test.cpp and is compiled as follows. The first case shows the default absolute path, the second --basenames, and the third shows --relativenames.

```
$ pwd
/tmp
$ clang -g foo/test.cpp -o test.elf
$ llvm-symbolizer --obj=test.elf 0x4004a0
main
/tmp/foo/test.cpp:15:0
$ llvm-symbolizer --obj=test.elf 0x4004a0 --basenames
main
```

test.cpp:15:0
$ llvm-symbolizer --obj=test.elf 0x4004a0 --relativenames
main
foo/test.cpp:15:0

OPTIONS

--adjust-vma <offset>
Add the specified offset to object file addresses when performing lookups. This can be used to perform lookups as if the object were relocated by the offset.

--basenames, -s
Print just the file's name without any directories, instead of the absolute path.

--build-id
Look up the object using the given build ID, specified as a hexadecimal string. Mutually exclusive with *--obj*.

--color [=<always|auto|never>]
Specify whether to use color in *--filter-markup* mode. Defaults to **auto**, which detects whether standard output supports color. Specifying **--color** alone is equivalent to **--color=always**.

--debug-file-directory <path>
Provide a path to a directory with a *.build-id* subdirectory to search for debug information for stripped binaries. Multiple instances of this argument are searched in the order given.

--debuginfod, --no-debuginfod
Whether or not to try debuginfod lookups for debug binaries. Unless specified, debuginfod is only enabled if libcurl was compiled in (**LLVM_ENABLE_CURL**) and at least one server URL was provided by the environment variable **DEBUGINFOD_URLS**.

--demangle, -C
Print demangled function names, if the names are mangled (e.g. the mangled name *_Z3bazv* becomes *baz()*, whilst the non-mangled name *foz* is printed as is). Defaults to true.

--dwp <path>
Use the specified DWP file at **<path>** for any CUs that have split DWARF debug data.

--fallback-debug-path <path>
When a separate file contains debug data, and is referenced by a GNU debug link section, use the

specified path as a basis for locating the debug data if it cannot be found relative to the object.

--filter-markup
Reads from standard input, converts contained *Symbolizer Markup* into human-readable form, and prints the results to standard output. The following markup elements are not yet supported:

- **{{{hexdict}}}**
- **{{{dumpfile}}}**

The **{{{bt}}}** backtrace element reports frames using the following syntax:

#<number>[.<inline>] <address> <function> <file>:<line>:<col> (<module>+<relative address>)

<inline> provides frame numbers for calls inlined into the caller coresponding to **<number>**. The inlined call numbers start at 1 and increase from callee to caller.

<address> is an address inside the call instruction to the function. The address may not be the start of the instruction. **<relative address>** is the corresponding virtual offset in the **<module>** loaded at that address.

--functions [=<none|short|linkage>], -f
Specify the way function names are printed (omit function name, print short function name, or print full linkage name, respectively). Defaults to **linkage**.

--help, -h
Show help and usage for this command.

--inlining, --inlines, -i
If a source code location is in an inlined function, prints all the inlined frames. This is the default.

--no-inlines
Don’t print inlined frames.

--no-demangle
Don’t print demangled function names.

--obj <path>, --exe, -e
Path to object file to be symbolized. If **-** is specified, read the object directly from the standard

input stream. Mutually exclusive with *--build-id*.

--output-style <LLVM|GNU|JSON>

Specify the preferred output style. Defaults to **LLVM**. When the output style is set to **GNU**, the tool follows the style of GNU's **addr2line**. The differences from the **LLVM** style are:

- Does not print the column of a source code location.
- Does not add an empty line after the report for an address.
- Does not replace the name of an inlined function with the name of the topmost caller when inlined frames are not shown.
- Prints an address's debug-data discriminator when it is non-zero. One way to produce discriminators is to compile with clang's -fdebug-info-for-profiling.

JSON style provides a machine readable output in JSON. If addresses are
supplied via stdin, the output JSON will be a series of individual objects. Otherwise, all results will be contained in a single array.

```
$ llvm-symbolizer --obj=inlined.elf 0x4004be 0x400486 -p
baz() at /tmp/test.cpp:11:18
 (inlined by) main at /tmp/test.cpp:15:0

foo() at /tmp/test.cpp:6:3

$ llvm-symbolizer --output-style=LLVM --obj=inlined.elf 0x4004be 0x400486 -p --no-inlines
main at /tmp/test.cpp:11:18

foo() at /tmp/test.cpp:6:3

$ llvm-symbolizer --output-style=GNU --obj=inlined.elf 0x4004be 0x400486 -p --no-inlines
baz() at /tmp/test.cpp:11
foo() at /tmp/test.cpp:6

$ clang -g -fdebug-info-for-profiling test.cpp -o profiling.elf
$ llvm-symbolizer --output-style=GNU --obj=profiling.elf 0x401167 -p --no-inlines
main at /tmp/test.cpp:15 (discriminator 2)

$ llvm-symbolizer --output-style=JSON --obj=inlined.elf 0x4004be 0x400486 -p
```

```
[
  {
    "Address": "0x4004be",
    "ModuleName": "inlined.elf",
    "Symbol": [
      {
        "Column": 18,
        "Discriminator": 0,
        "FileName": "/tmp/test.cpp",
        "FunctionName": "baz()",
        "Line": 11,
        "StartAddress": "0x4004be",
        "StartFileName": "/tmp/test.cpp",
        "StartLine": 9
      },
      {
        "Column": 0,
        "Discriminator": 0,
        "FileName": "/tmp/test.cpp",
        "FunctionName": "main",
        "Line": 15,
        "StartAddress": "0x4004be",
        "StartFileName": "/tmp/test.cpp",
        "StartLine": 14
      }
    ]
  },
  {
    "Address": "0x400486",
    "ModuleName": "inlined.elf",
    "Symbol": [
      {
        "Column": 3,
        "Discriminator": 0,
        "FileName": "/tmp/test.cpp",
        "FunctionName": "foo()",
        "Line": 6,
        "StartAddress": "0x400486",
        "StartFileName": "/tmp/test.cpp",
        "StartLine": 5
```

```
    }
   ]
  }
 ]
```

--pretty-print, -p

Print human readable output. If *--inlining* is specified, the enclosing scope is prefixed by (inlined by). For JSON output, the option will cause JSON to be indented and split over new lines. Otherwise, the JSON output will be printed in a compact form.

```
$ llvm-symbolizer --obj=inlined.elf 0x4004be --inlining --pretty-print
baz() at /tmp/test.cpp:11:18
 (inlined by) main at /tmp/test.cpp:15:0
```

--print-address, --addresses, -a

Print address before the source code location. Defaults to false.

```
$ llvm-symbolizer --obj=inlined.elf --print-address 0x4004be
0x4004be
baz()
/tmp/test.cpp:11:18
main
/tmp/test.cpp:15:0

$ llvm-symbolizer --obj=inlined.elf 0x4004be --pretty-print --print-address
0x4004be: baz() at /tmp/test.cpp:11:18
 (inlined by) main at /tmp/test.cpp:15:0
```

--print-source-context-lines <N>

Print **N** lines of source context for each symbolized address.

```
$ llvm-symbolizer --obj=test.elf 0x400490 --print-source-context-lines=3
baz()
/tmp/test.cpp:11:0
10  :   volatile int k = 42;
11 >:   return foz() + k;
12  : }
```

--relativenames

Print the file’s path relative to the compilation directory, instead of the absolute path. If the

command-line to the compiler included the full path, this will be the same as the default.

--verbose

Print verbose address, line and column information.

```
$ llvm-symbolizer --obj=inlined.elf --verbose 0x4004be
baz()
  Filename: /tmp/test.cpp
  Function start filename: /tmp/test.cpp
  Function start line: 9
  Function start address: 0x4004b6
  Line: 11
  Column: 18
main
  Filename: /tmp/test.cpp
  Function start filename: /tmp/test.cpp
  Function start line: 14
  Function start address: 0x4004b0
  Line: 15
  Column: 18
```

--version, -v

Print version information for the tool.

@<FILE>

Read command-line options from response file <*FILE*>.

WINDOWS/PDB SPECIFIC OPTIONS

--dia

Use the Windows DIA SDK for symbolization. If the DIA SDK is not found, llvm-symbolizer will fall back to the native implementation.

MACH-O SPECIFIC OPTIONS

--default-arch <arch>

If a binary contains object files for multiple architectures (e.g. it is a Mach-O universal binary), symbolize the object file for a given architecture. You can also specify the architecture by writing **binary_name:arch_name** in the input (see example below). If the architecture is not specified in either way, the address will not be symbolized. Defaults to empty string.

```
$ cat addr.txt
/tmp/mach_universal_binary:i386 0x1f84
/tmp/mach_universal_binary:x86_64 0x100000f24

$ llvm-symbolizer < addr.txt
_main
/tmp/source_i386.cc:8

_main
/tmp/source_x86_64.cc:8
```

--dsym-hint <path/to/file.dSYM>

If the debug info for a binary isn't present in the default location, look for the debug info at the .dSYM path provided via this option. This flag can be used multiple times.

EXIT STATUS

llvm-symbolizer returns 0. Other exit codes imply an internal program error.

SEE ALSO

llvm-addr2line(1)

AUTHOR

Maintained by the LLVM Team (https://llvm.org/).

COPYRIGHT