Red Hat Developer Tools 2018.4

Using Clang and LLVM Toolset

Installing and Using Clang and LLVM Toolset

Last Updated: 2018-11-29

Red Hat Developer Tools 2018.4 Using Clang and LLVM Toolset

Installing and Using Clang and LLVM Toolset

Robin Owen
kowen@redhat.com

Legal Notice

Abstract

Clang and LLVM Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. The Using Clang and LLVM Toolset provides an overview of this product, explains how to invoke and use the Clang and LLVM Toolset versions of the tools, and links to resources with more in-depth information.

Table of Contents

CHAPTER 1. CLANG AND LLVM TOOLSET

1.1. ABOUT CLANG AND LLVM TOOLSET

Clang and LLVM Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. It provides the **LLVM** compiler infrastructure framework, the **Clang** compiler for the C and C++ languages, the **LLDB** debugger, and related tools for code analysis.

Clang and LLVM Toolset is distributed as a part of Red Hat Developer Tools for Red Hat Enterprise Linux 7.

The following components are available as a part of Clang and LLVM Toolset:

Table 1.1. Clang and LLVM Toolset Components

Name	Version	Description
`clang`	6.0.1	A LLVM compiler front-end for C and C++.
`lldb`	6.0.1	A debugger using portions of LLVM.
`CMake`	3.6.2	A build management system.
`compiler-rt`	6.0.1	Runtime libraries for LLVM.
`llvm`	6.0.1	A collection of modular and reusable compiler and toolchain technologies.
`libomp`	6.0.1	A library for utilization of Open MP API specification for parallel programming.
`python-lit`	0.6	A Software testing tool for LLVM- and Clang-based test suites.

1.2. COMPATIBILITY

Clang and LLVM Toolset is available for Red Hat Enterprise Linux 7 on the following architectures:

- The 64-bit Intel and AMD architectures
- The 64-bit ARM architecture
- The IBM Power Systems architecture
- The little-endian variant of IBM Power Systems architecture
- The IBM Z Systems architecture

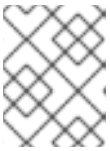
1.3. GETTING ACCESS TO CLANG AND LLVM TOOLSET

Clang and LLVM Toolset is an offering that is distributed as a part of the Red Hat Developer Tools content set, which is available to customers with deployments of Red Hat Enterprise Linux 7. To install Clang and LLVM Toolset, enable the Red Hat Developer Tools and Red Hat Software Collections repositories by using the Red Hat Subscription Management and add the Red Hat Developer Tools key to your system.

1. Enable the **rhel-7-*variant*-devtools-rpms** repository:

   ```
   # subscription-manager repos --enable rhel-7-variant-devtools-rpms
   ```

   Replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

   NOTE

   We recommend developers to use Red Hat Enterprise Linux Server for access to the widest range of development tools.

2. Enable the **rhel-*variant*-rhscl-8-rpms** repository:

   ```
   # subscription-manager repos --enable rhel-variant-rhscl-8-rpms
   ```

   Replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

3. Add the Red Hat Developer Tools key to your system:

   ```
   # cd /etc/pki/rpm-gpg
   # wget -O RPM-GPG-KEY-redhat-devel
   https://www.redhat.com/security/data/a5787476.txt
   # rpm --import RPM-GPG-KEY-redhat-devel
   ```

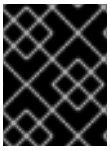
Once the subscription is attached to the system and repositories enabled, you can install Red Hat Clang and LLVM Toolset as described in Section 1.4, “Installing Clang and LLVM Toolset”.

Additional Resources

- For more information on how to register your system using Red Hat Subscription Management and associate it with subscriptions, see the Red Hat Subscription Management collection of guides.
- For detailed instructions on subscription to Red Hat Software Collections, see the *Red Hat Developer Toolset User Guide*, Section 1.4. Getting Access to Red Hat Developer Toolset.

1.4. INSTALLING CLANG AND LLVM TOOLSET

Clang and LLVM Toolset is distributed as a collection of RPM packages that can be installed, updated, uninstalled, and inspected by using the standard package management tools that are included in Red Hat Enterprise Linux. Note that a valid subscription that provides access to the Red Hat Developer Tools content set is required in order to install Clang and LLVM Toolset on your system. For detailed instructions on how to associate your system with an appropriate subscription and get access to Clang and LLVM Toolset, see Section 1.3, “Getting Access to Clang and LLVM Toolset”.



IMPORTANT

Before installing Clang and LLVM Toolset, install all available Red Hat Enterprise Linux updates.

To install all components that are included in Clang and LLVM Toolset, install the **llvm-toolset-6.0** package:

```
# yum install llvm-toolset-6.0
```

This installs all development and debugging tools, and other dependent packages to the system.

1.5. ADDITIONAL RESOURCES

A detailed description of the Clang and LLVM Toolset and all its features is beyond the scope of this book. For more information, see the resources listed below.

Online Documentation

- LLVM documentation overview — The official **LLVM** documentation.

CHAPTER 2. CLANG

clang is a **LLVM** compiler front end for C-based languages: C, C++, Objective C/C++, OpenCL, and Cuda.

Clang and LLVM Toolset is distributed with **clang 6.0.1**.

2.1. INSTALLING CLANG

In Clang and LLVM Toolset, **clang** is provided by the **llvm-toolset-6.0-clang** package and is automatically installed with the **llvm-toolset-6.0** package. See Section 1.4, “Installing Clang and LLVM Toolset”.

2.2. USING THE C COMPILER

To compile a C program on the command line, run the **clang** compiler as follows:

```
$ scl enable llvm-toolset-6.0 'clang -o output_file source_file'
```

This creates a binary file named ***output_file*** in the current working directory. If the **-o** option is omitted, the compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line:

```
$ scl enable llvm-toolset-6.0 'clang -o object_file -c source_file'
```

This creates an object file named ***object_file***. If the **-o** option is omitted, the compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file:

```
$ scl enable llvm-toolset-6.0 'clang -o output_file object_file ...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Clang and LLVM Toolset binaries available. This allows you to run a shell session with Clang and LLVM Toolset **clang** directly available:

```
$ scl enable llvm-toolset-6.0 'bash'
```

IMPORTANT

Certain more recent library features are statically linked into applications built with Clang and LLVM Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk as standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

Because of this additional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Example 2.1. Compiling a C Program on the Command Line

Consider a source file named **hello.c** with the following contents:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

Compile this source code on the command line by using the **clang** compiler from Clang and LLVM Toolset:

```
$ scl enable llvm-toolset-6.0 'clang -o hello hello.c'
```

This creates a new binary file called **hello** in the current working directory.

2.3. RUNNING A C PROGRAM

When **clang** compiles a program, it creates an executable binary file. To run this program on the command line, change to the directory with the executable file and run the program:

```
$ ./file_name
```

Example 2.2. Running a C Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 2.1, “Compiling a C Program on the Command Line”, you can run it by typing the following at a shell prompt:

```
$ ./hello
Hello, World!
```

2.4. USING THE C++ COMPILER

To compile a C++ program on the command line, run the **clang++** compiler as follows:

```
$ scl enable llvm-toolset-6.0 'clang++ -o output_file source_file ...'
```

This creates a binary file named ***output_file*** in the current working directory. If the **-o** option is omitted, the **clang++** compiler creates a file named **a.out** by default.

When you are working on a project that consists of several source files, it is common to compile an object file for each of the source files first and then link these object files together. This way, when you change a single source file, you can recompile only this file without having to compile the entire project. To compile an object file on the command line:

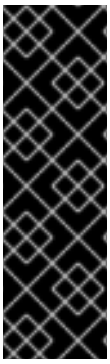
```
$ scl enable llvm-toolset-6.0 'clang++ -o object_file -c source_file'
```

This creates an object file named ***object_file***. If the **-o** option is omitted, the **clang++** compiler creates a file named after the source file with the **.o** file extension. To link object files together and create a binary file:

```
$ scl enable llvm-toolset-6.0 'clang++ -o output_file object_file ...'
```

Note that you can execute any command using the **scl** utility, causing it to be run with the Clang and LLVM Toolset binaries available. This allows you to run a shell session with Clang and LLVM Toolset **clang** directly available:

```
$ scl enable llvm-toolset-6.0 'bash'
```



IMPORTANT

Certain more recent library features are statically linked into applications built with Clang and LLVM Toolset to support execution on multiple versions of Red Hat Enterprise Linux. This creates an additional minor security risk as standard Red Hat Enterprise Linux errata do not change this code. If the need arises for developers to rebuild their applications due to this risk, Red Hat will communicate this using a security erratum.

Because of this accitional security risk, developers are strongly advised not to statically link their entire application for the same reasons.

Example 2.3. Compiling a C++ Program on the Command Line

Consider a source file named **hello.cpp** with the following contents:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv[]) {
  cout << "Hello, World!" << endl;
  return 0;
}
```

Compile this source code on the command line by using the **clang++** compiler from Clang and LLVM Toolset:

```
$ scl enable llvm-toolset-6.0 'clang++ -o hello hello.cpp'
```

This creates a new binary file called **hello** in the current working directory.

2.5. RUNNING A C++ PROGRAM

When **clang++** compiles a program, it creates an executable binary file. Change to the directory with the executable file and run this program:

```
./file_name
```

Example 2.4. Running a C++ Program on the Command Line

Assuming that you have successfully compiled the **hello** binary file as shown in Example 2.3, "Compiling a C++ Program on the Command Line", you can run it by typing the following at a shell prompt:

```
$ ./hello
Hello, World!
```

2.6. USING THE CLANG INTEGRATED ASSEMBLER

To produce an object file from an assembly language program, run the **clang** tool as follows:

```
$ scl enable llvm-toolset-6.0 'clang option... -o object_file source_file'
```

This creates an object file named ***object_file*** in the current working directory.

Note that you can execute any command using the **scl** utility, causing it to be run with the Clang and LLVM Toolset binaries available. This allows you to run a shell session with Red LLVM Developer Toolset:

```
$ scl enable llvm-toolset-6.0 'bash'
```

2.7. ADDITIONAL RESOURCES

A detailed description of the **clang** compiler and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- *clang*(1) — The manual page for the **clang** compiler provides detailed information on its usage; with few exceptions, **clang++** accepts the same command line options as **clang**. To display the manual page for the version included in Clang and LLVM Toolset:

  ```
  $ scl enable llvm-toolset-6.0 'man clang'
  ```

Online Documentation

- clang — The clang compiler documentation provides detailed information on **clang**'s usage.

See Also

- Chapter 1, *Clang and LLVM Toolset* — An overview of Clang and LLVM Toolset and more information on how to install it on your system.

CHAPTER 3. LLDB

lldb is a command line tool you can use to debug programs written in various programming languages. It allows you to inspect memory within the code being debugged, control the execution state of the code, detect the execution of particular sections of code, and much more.

Clang and LLVM Toolset is distributed with **lldb 6.0.1**.

3.1. INSTALLING LLDB

The **lldb** tool is provided by the **llvm-toolset-6.0-lldb** package and is automatically installed with the **llvm-toolset-6.0** package. See Section 1.4, “Installing Clang and LLVM Toolset”.

3.2. PREPARING A PROGRAM FOR DEBUGGING

To compile a C or C++ program with debugging information that **lldb** can read, make sure the compiler you use is instructed to create debug information.

- For instructions on suitably configuring **GCC**, see TODO link to DTS User Guide.
- For instructions on suitably configuring **clang**, see the section Controlling Debug Information in Clang Compiler User's Manual.

3.3. RUNNING LLDB

To run **lldb** on a program you want to debug:

```
$ scl enable llvm-toolset-6.0 'lldb program_file_name'
```

This command starts **`lldb`** in an interactive mode and displays the default prompt, **`(lldb)`**.

To quit the debugging session and return to the shell prompt, run the following command at any time:

```
(lldb) quit
```

Note that you can execute any command using the **`scl`** utility, causing it to be run with the Clang and LLVM Toolset binaries available. This allows you to run a shell session with Clang and LLVM Toolset **`lldb`** directly available:

```
$ scl enable llvm-toolset-6.0 'bash'
```

Example 3.1. Running the lldb Utility on the fibonacci Binary File

This example assumes that you have successfully compiled the **`fibonacci`** binary file as shown in TODO add example and link it.

Start debugging the program with **`lldb`**:

```
$ scl enable llvm-toolset-6.0 'lldb fibonacci'
(lldb) target create "fibonacci"
Current executable set to 'fibonacci' (x86_64).
(lldb)
```

The output indicates that the program **fibonacci** is ready for debugging.

3.4. LISTING SOURCE CODE

To view the source code of the program you are debugging:

```
(lldb) list
```

As a result, the first ten lines of the source code are displayed.

To display the code surrounding a particular line:

```
(lldb) list file_name:line_number
```

Additionally, **lldb** displays source code listing automatically in the following situations:

- Before you start the execution of the program you are debugging, **lldb** displays the first ten lines of the source code.
- Each time the execution of the program is stopped, **lldb** displays the lines that surround the line on which the execution stops.

3.5. USING BREAKPOINTS

Setting a New Breakpoint

To set a new breakpoint at a certain line:

```
(lldb) breakpoint file_name:line_number
```

To set a breakpoint on a certain function:

```
(lldb) breakpoint file_name:function_name
```

Example 3.2. Setting a New Breakpoint

This example assumes that you have successfully compiled the **fibonacci.c** file listed in TODO add example and link it with debugging information.

Set a new breakpoint at line 10 by running either of the following commands:

```
(lldb) b 10
Breakpoint 1: where = fibonacci`main + 33 at fibonacci.c:10, address =
0x000000000040054e
(lldb)  breakpoint set -f fibonacci.c --line 10
Breakpoint 2: where = fibonacci`main + 33 at fibonacci.c:10, address =
0x000000000040054e
```

NOTE

In lldb, the command **b** is not an alias to **breakpoint**. You can use both commands to set breakpoints, but **b** uses a subset of the syntax supported by gdb's **break** command, and **breakpoint** uses **lldb**'s syntax for setting breakpoints.

Listing Breakpoints

To display a list of currently set breakpoints:

```
(lldb) breakpoint list
```

Example 3.3. Listing Breakpoints

This example assumes that you have successfully followed the instructions in Example 3.2, "Setting a New Breakpoint".

Display the list of currently set breakpoints:

```
(lldb) breakpoint list
Current breakpoints:
1: file = 'fibonacci.c', line = 10, exact_match = 0, locations = 1
  1.1: where = fibonacci`main + 33 at fibonacci.c:10, address =
fibonacci[0x000000000040054e], unresolved, hit count = 0

2: file = 'fibonacci.c', line = 10, exact_match = 0, locations = 1
  2.1: where = fibonacci`main + 33 at fibonacci.c:10, address =
fibonacci[0x000000000040054e], unresolved, hit count = 0
```

Deleting Existing Breakpoints

To delete a breakpoint that is set at a certain line:

```
(lldb) breakpoint clear -f file_name -l 10
```

Example 3.4. Deleting an Existing Breakpoint

This example assumes that you have successfully compiled the **fibonacci.c** file.

Set a new breakpoint at line 7:

```
(lldb) b 7
Breakpoint 3: where = fibonacci`main + 31 at fibonacci.c:9, address =
0x000000000040054c
```

Remove this breakpoint:

```
(lldb) breakpoint clear -l 7 -f fibonacci.c
1 breakpoints cleared:
3: file = 'fibonacci.c', line = 7, exact_match = 0, locations = 1
```

3.6. STARTING EXECUTION

To start an execution of the program you are debugging:

```
(lldb) run
```

If the program accepts command-line arguments, you can provide them as arguments to the **run** command:

```
(lldb) run argument …
```

The execution stops when the first breakpoint is reached, when an error occurs, or when the program terminates.

Example 3.5. Executing the fibonacci Binary File

This example assumes that you have successfully followed the instructions in Example 3.2, “Setting a New Breakpoint”.

Execute the **fibonacci** binary file:

```
(lldb) run
Process 21054 launched: 'fibonacci' (x86_64)
Process 21054 stopped
* thread #1, name = 'fibonacci', stop reason = breakpoint 1.1
    frame #0: fibonacci`main(argc=1, argv=0x00007fffffffdeb8) at
fibonacci.c:10
   7       unsigned long int sum;
   8
   9       while (b < LONG_MAX) {
-> 10        printf("%ld ", b);
   11        sum = a + b;
   12        a = b;
   13        b = sum;
```

Execution of the program stops at the breakpoint set in Example 3.2, “Setting a New Breakpoint”.

3.7. DISPLAYING CURRENT VALUES

The **lldb** tool enables you to display many values relevant to the program state, including:

- Variables of any complexity
- Any valid expressions
- Function call return values

The most common task is to display the value of a variable. To display the current value of a certain variable:

```
(lldb) print variable_name
```

Example 3.6. Displaying the Current Values of Variables

This example assumes that you have successfully followed the instructions in Example 3.5, "Executing the fibonacci Binary File". Execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10.

Display the current values of variables **a** and **b**:

```
(lldb) print a
$0 = 0
(lldb) print b
$1 = 1
```

3.8. CONTINUING EXECUTION

To resume the execution of the program you are debugging after it reached a breakpoint:

```
(lldb) continue
```

The execution stops again when it reaches another breakpoint.

To skip a certain number of breakpoints, typically when you are debugging a loop, run the **continue** command in the following form:

```
(lldb) continue -i number_of_breakpoints_to_skip
```

The **lldb** tool also enables you to stop the execution after executing a single line of code:

```
(lldb) step
```

To execute a certain number of lines:

```
(lldb) step -c number
```

Example 3.7. Continuing the Execution of the fibonacci Binary File

This example assumes that you have successfully followed the instructions in Example 3.5, "Executing the fibonacci Binary File". The execution of the **fibonacci** binary stopped after reaching the breakpoint at line 10.

Resume the execution:

```
(lldb) continue
Process 21580 resuming
Process 21580 stopped
* thread #1, name = 'fibonacci', stop reason = breakpoint 1.1
    frame #0: fibonacci`main(argc=1, argv=0x00007fffffffdeb8) at
fibonacci.c:10
   7       unsigned long int sum;
   8
   9       while (b < LONG_MAX) {
```

```
-> 10       printf("%ld ", b);
   11       sum = a + b;
   12       a = b;
   13       b = sum;
```

The execution stops the next time it reaches a breakpoint. (In this case it is the same breakpoint). Execute the next three lines of code:

```
(lldb) step -c 3
Process 21580 stopped
* thread #1, name = 'fibonacci', stop reason = step in
    frame #0: fibonacci`main(argc=1, argv=0x00007fffffffdeb8) at
fibonacci.c:11
   8
   9      while (b < LONG_MAX) {
   10       printf("%ld ", b);
-> 11       sum = a + b;
   12       a = b;
   13       b = sum;
   14     }
```

Verify the current value of the **sum** variable before it is assigned to the **b** variable:

```
(lldb) print sum
$2 = 2
```

3.9. ADDITIONAL RESOURCES

A detailed description of the **lldb** debugger and all its features is beyond the scope of this book. For more information, see the resources listed below.

Online Documentation

- lldb Tutorial — The official **lldb** tutorial.
- gdb to lldb command map — A list of **GDB** commands and their **lldb** equivalents.

See Also

- Chapter 1, *Clang and LLVM Toolset* — An overview of Clang and LLVM Toolset and more information on how to install it.

CHAPTER 4. CONTAINER IMAGE

The Clang and LLVM Toolset is available as a Docker-formatted container image which can be downloaded from Red Hat Container Registry.

4.1. IMAGE CONTENTS

The **devtools/llvm-toolset-6.0-rhel7** image provides content corresponding to the following packages:

Component	Version	Package
`LLVM`	6.0.1	llvm-toolset-8-llvm
`Clang`	6.0.1	llvm-toolset-8-clang
`LLDB`	6.0.1	llvm-toolset-8-lldb
`Runtime libraries`	6.0.1	llvm-toolset-8-compiler-rt
`Open MP library`	6.0.1	llvm-toolset-8-libomp
`CMake`	3.6.2	llvm-toolset-8-cmake
`python-lit`	0.6	llvm-toolset-8-python-lit

4.2. ACCESS TO THE IMAGE

To pull the **devtools/llvm-toolset-6.0-rhel7** image, run the following command as **root**:

```
# docker pull registry.access.redhat.com/devtools/llvm-toolset-6.0-rhel7
```

4.3. ADDITIONAL RESOURCES

- Clang and LLVM Toolset 8 - entry in the Red Hat Container Catalog
- Using Red Hat Software Collections Container Images

CHAPTER 5. CHANGES IN CLANG AND LLVM TOOLSET IN RED HAT DEVELOPER TOOLS 2018.4

This chapter lists some notable changes in Clang and LLVM Toolset since its previous release.

5.1. LLVM

LLVM has been updated from version **5.0.1** to **6.0.1**.

For more information, see the LLVM 6.0.0 Release Notes.

5.2. CLANG

clang has been updated from version **5.0.1** to **6.0.1**.

For more information, see the Clang 6.0.0 Release Notes.