Cross-compilation using Clang

Introduction

This document will guide you in choosing the right Clang options for cross-compiling your code to a different architecture. It assumes you already know how to compile the code in question for the host architecture, and that you know how to choose additional include and library paths.

However, this document is *not* a "how to" and won't help you setting your build system or Makefiles, nor choosing the right CMake options, etc. Also, it does not cover all the possible options, nor does it contain specific examples for specific architectures. For a concrete example, the **instructions for cross-compiling LLVM itself** may be of interest.

After reading this document, you should be familiar with the main issues related to cross-compilation, and what main compiler options Clang provides for performing cross-compilation.

Cross compilation issues

In GCC world, every host/target combination has its own set of binaries, headers, libraries, etc. So, it's usually simple to download a package with all files in, unzip to a directory and point the build system to that compiler, that will know about its location and find all it needs to when compiling your code.

On the other hand, Clang/LLVM is natively a cross-compiler, meaning that one set of programs can compile to all targets by setting the `-target` option. That makes it a lot easier for programmers wishing to compile to different platforms and architectures, and for compiler developers that only have to maintain one build system, and for OS distributions, that need only one set of main packages.

But, as is true to any cross-compiler, and given the complexity of different architectures, OS's and options, it's not always easy finding the headers, libraries or binutils to generate target specific code. So you'll need special options to help Clang understand what target you're compiling to, where your tools are, etc.

Another problem is that compilers come with standard libraries only (like `compiler-rt`, `libcxx`, `libgcc`, `libm`, etc), so you'll have to find and make available to the build system, every other library required to build your software, that is specific to your target. It's not enough to have your host's libraries installed.

Finally, not all toolchains are the same, and consequently, not every Clang option will work magically. Some options, like `--sysroot` (which effectively changes the logical root for headers and libraries), assume all your binaries and libraries are in the same directory, which may not true when your cross-compiler was installed by the distribution's package management. So, for each specific case, you may use more than one option, and in most cases, you'll end up setting include paths (`-I`) and library paths (`-L`) manually.

To sum up, different toolchains can:

be host/target specific or more flexible
be in a single directory, or spread out across your system
have different sets of libraries and headers by default
need special options, which your build system won't be able to figure out by itself

General Cross-Compilation Options in Clang

Target Triple

The basic option is to define the target architecture. For that, use `-target <triple>`. If you don't specify the target, CPU names won't match (since Clang assumes the host triple), and the compilation will go ahead, creating code for the host platform, which will break later on when assembling or linking.

The triple has the general format `<arch><sub>-<vendor>-<sys>-<abi>`, where:

`arch` = `x86_64`, `i386`, `arm`, `thumb`, `mips`, etc.
`sub` = for ex. on ARM: `v5`, `v6m`, `v7a`, `v7m`, etc.
`vendor` = `pc`, `apple`, `nvidia`, `ibm`, etc.
`sys` = `none`, `linux`, `win32`, `darwin`, `cuda`, etc.
`abi` = `eabi`, `gnu`, `android`, `macho`, `elf`, etc.

The sub-architecture options are available for their own architectures, of course, so "x86v7a" doesn't make sense. The vendor needs to be specified only if there's a relevant change, for instance between PC and Apple. Most of the time it can be omitted (and Unknown) will be assumed, which sets the defaults for the specified architecture. The system name is generally the OS (linux, darwin), but could be special like the bare-metal "none".

When a parameter is not important, it can be omitted, or you can choose `unknown` and the defaults will be used. If you choose a parameter that Clang doesn't know, like `blerg`, it'll ignore and assume `unknown`, which is not always desired, so be careful.

Finally, the ABI option is something that will pick default CPU/FPU, define the specific behaviour of your code (PCS, extensions), and also choose the correct library calls, etc.

CPU, FPU, ABI

Once your target is specified, it’s time to pick the hardware you’ll be compiling to. For every architecture, a default set of CPU/FPU/ABI will be chosen, so you’ll almost always have to change it via flags.

Typical flags include:

- `-mcpu=<cpu-name>`, like x86-64, swift, cortex-a15
- `-mfpu=<fpu-name>`, like SSE3, NEON, controlling the FP unit available
- `-mfloat-abi=<fabi>`, like soft, hard, controlling which registers to use for floating-point

The default is normally the common denominator, so that Clang doesn’t generate code that breaks. But that also means you won’t get the best code for your specific hardware, which may mean orders of magnitude slower than you expect.

For example, if your target is `arm-none-eabi`, the default CPU will be `arm7tdmi` using soft float, which is extremely slow on modern cores, whereas if your triple is `armv7a-none-eabi`, it’ll be Cortex-A8 with NEON, but still using soft-float, which is much better, but still not great.

Toolchain Options

There are three main options to control access to your cross-compiler: `--sysroot`, `-I`, and `-L`. The two last ones are well known, but they’re particularly important for additional libraries and headers that are specific to your target.

There are two main ways to have a cross-compiler:

1. When you have extracted your cross-compiler from a zip file into a directory, you have to use `--sysroot=<path>`. The path is the root directory where you have unpacked your file, and Clang will look for the directories `bin`, `lib`, `include` in there.

   In this case, your setup should be pretty much done (if no additional headers or libraries are needed), as Clang will find all binaries it needs (assembler, linker, etc) in there.

2. When you have installed via a package manager (modern Linux distributions have cross-compiler packages available), make sure the target triple you set is *also* the prefix of your cross-compiler toolchain.

   In this case, Clang will find the other binaries (assembler, linker), but not always where the target headers and libraries are. People add system-specific clues to Clang often, but as things change, it’s more likely that it won’t find than the other way around.

So, here, you'll be a lot safer if you specify the include/library directories manually (via `-I` and `-L`).

Target-Specific Libraries

All libraries that you compile as part of your build will be cross-compiled to your target, and your build system will probably find them in the right place. But all dependencies that are normally checked against (like `libxml` or `libz` etc) will match against the host platform, not the target.

So, if the build system is not aware that you want to cross-compile your code, it will get every dependency wrong, and your compilation will fail during build time, not configure time.

Also, finding the libraries for your target are not as easy as for your host machine. There aren't many cross-libraries available as packages to most OS's, so you'll have to either cross-compile them from source, or download the package for your target platform, extract the libraries and headers, put them in specific directories and add `-I` and `-L` pointing to them.

Also, some libraries have different dependencies on different targets, so configuration tools to find dependencies in the host can get the list wrong for the target platform. This means that the configuration of your build can get things wrong when setting their own library paths, and you'll have to augment it via additional flags (configure, Make, CMake, etc).

Multilibs

When you want to cross-compile to more than one configuration, for example hard-float-ARM and soft-float-ARM, you'll have to have multiple copies of your libraries and (possibly) headers.

Some Linux distributions have support for Multilib, which handle that for you in an easier way, but if you're not careful and, for instance, forget to specify `-ccc-gcc-name armv7l-linux-gnueabihf-gcc` (which uses hard-float), Clang will pick the `armv7l-linux-gnueabi-ld` (which uses soft-float) and linker errors will happen.

The same is true if you're compiling for different ABIs, like `gnueabi` and `androideabi`, and might even link and run, but produce run-time errors, which are much harder to track down and fix.