Getting Started: Building and Running Clang

This page gives you the shortest path to checking out Clang and demos a few options. This should get you up and running with the minimum of muss and fuss. If you like what you see, please consider getting involved with the Clang community. If you run into problems, please file bugs in LLVM Bugzilla.

Release Clang Versions

Clang is released as part of regular LLVM releases. You can download the release versions from http://llvm.org/releases/.

Clang is also provided in all major BSD or GNU/Linux distributions as part of their respective packaging systems. From Xcode 4.2, Clang is the default compiler for Mac OS X.

Building Clang and Working with the Code

On Unix-like Systems

Note: as an experimental setup, you can use a **single checkout** with all the projects, and an **easy CMake invocation**, see the LLVM Doc "For developers to work with a git monorepo"

If you would like to check out and build Clang, the current procedure is as follows:

1. Get the required tools.
    - See Getting Started with the LLVM System - Requirements.
    - Note also that Python is needed for running the test suite. Get it at: http://www.python.org/download
    - Standard build process uses CMake. Get it at: http://www.cmake.org/download
2. Check out LLVM:
    - Change directory to where you want the llvm directory placed.
    - `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
3. Check out Clang:
    - `cd llvm/tools`

   - `svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`
   - `cd ../..`
4. Check out extra Clang tools: (optional)
   - `cd llvm/tools/clang/tools`
   - `svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra`
   - `cd ../../../..`
5. Check out Compiler-RT (optional):
   - `cd llvm/projects`
   - `svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt`
   - `cd ../..`
6. Check out libcxx: (only required to build and run Compiler-RT tests on OS X, optional otherwise)
   - `cd llvm/projects`
   - `svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx`
   - `cd ../..`
7. Build LLVM and Clang:
   - `mkdir build` (in-tree build is not supported)
   - `cd build`
   - `cmake -G "Unix Makefiles" ../llvm`
   - `make`
   - This builds both LLVM and Clang for debug mode.
   - Note: For subsequent Clang development, you can just run `make clang`.
   - CMake allows you to generate project files for several IDEs: Xcode, Eclipse CDT4, CodeBlocks, Qt-Creator (use the CodeBlocks generator), KDevelop3. For more details see Building LLVM with CMake page.
8. If you intend to use Clang's C++ support, you may need to tell it how to find your C++ standard library headers. In general, Clang will detect the best version of libstdc++ headers available and use them - it will look both for system installations of libstdc++ as well as installations adjacent to Clang itself. If your configuration fits neither of these scenarios, you can use the `-DGCC_INSTALL_PREFIX` cmake option to tell Clang where the gcc containing the desired libstdc++ is installed.

9. Try it out (assuming you add llvm/build/bin to your path):
    - `clang --help`
    - `clang file.c -fsyntax-only` (check for correctness)
    - `clang file.c -S -emit-llvm -o -` (print out unoptimized llvm code)
    - `clang file.c -S -emit-llvm -o - -O3`
    - `clang file.c -S -O3 -o -` (output native machine code)
10. Run the testsuite:
    - `make check-clang`

If you encounter problems while building Clang, make sure that your LLVM checkout is at the same revision as your Clang checkout. LLVM's interfaces change over time, and mismatched revisions are not expected to work together.

Simultaneously Building Clang and LLVM:

Once you have checked out Clang into the llvm source tree it will build along with the rest of `llvm`. To build all of LLVM and Clang together all at once simply run `make` from the root LLVM directory.

Note: Observe that Clang is technically part of a separate Subversion repository. As mentioned above, the latest Clang sources are tied to the latest sources in the LLVM tree. You can update your toplevel LLVM project and all (possibly unrelated) projects inside it with **`make update`**. This will run `svn update` on all subdirectories related to subversion.

Using Visual Studio

The following details setting up for and building Clang on Windows using Visual Studio:

1. Get the required tools:
    - **Subversion**. Source code control program. Get it from: http://subversion.apache.org/packages.html
    - **CMake**. This is used for generating Visual Studio solution and project files. Get it from: http://www.cmake.org/cmake/resources/software.html
    - **Visual Studio 2013 or later**
    - **Python**. This is needed only if you will be running the tests (which is essential, if you will be developing for clang). Get it from: http://www.python.org/download/

- **GnuWin32 tools** These are also necessary for running the tests. (Note that the grep from MSYS or Cygwin doesn't work with the tests because of embedded double-quotes in the search strings. The GNU grep does work in this case.) Get them from http://getgnuwin32.sourceforge.net/.

2. Check out LLVM:
   - `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
3. Check out Clang:
   - `cd llvm\tools`
   - `svn co http://llvm.org/svn/llvm-project/cfe/trunk clang`

   Note: Some Clang tests are sensitive to the line endings. Ensure that checking out the files does not convert LF line endings to CR+LF. If you use git-svn, make sure your `core.autocrlf` setting is false.

4. Run CMake to generate the Visual Studio solution and project files:
   - `cd ..\..` (back to where you started)
   - `mkdir build` (for building without polluting the source dir)
   - `cd build`
   - If you are using Visual Studio 2013: `cmake -G "Visual Studio 12" ..\llvm`
   - By default, the Visual Studio project files generated by CMake use the 32-bit toolset. If you are developing on a 64-bit version of Windows and want to use the 64-bit toolset, pass the ``-Thost=x64`` flag when generating the Visual Studio solution. This requires CMake 3.8.0 or later.
   - See the LLVM CMake guide for more information on other configuration options for CMake.
   - The above, if successful, will have created an LLVM.sln file in the `build` directory.
5. Build Clang:
   - Open LLVM.sln in Visual Studio.
   - Build the "clang" project for just the compiler driver and front end, or the "ALL_BUILD" project to build everything, including tools.
6. Try it out (assuming you added llvm/debug/bin to your path). (See the running examples from above.)
7. See Hacking on clang - Testing using Visual Studio on Windows for

information on running regression tests on Windows.

Note that once you have checked out both llvm and clang, to synchronize to the latest code base, use the `svn update` command in both the llvm and llvm\tools\clang directories, as they are separate repositories.

Clang Compiler Driver (Drop-in Substitute for GCC)

The `clang` tool is the compiler driver and front-end, which is designed to be a drop-in replacement for the `gcc` command. Here are some examples of how to use the high-level driver:

```
$ cat t.c
#include <stdio.h>
int main(int argc, char **argv) { printf("hello world\n"); }
$ clang t.c
$ ./a.out
hello world
```

The 'clang' driver is designed to work as closely to GCC as possible to maximize portability. The only major difference between the two is that Clang defaults to gnu99 mode while GCC defaults to gnu89 mode. If you see weird link-time errors relating to inline functions, try passing -std=gnu89 to clang.

Examples of using Clang

```
$ cat ~/t.c
typedef float V __attribute__((vector_size(16)));
V foo(V a, V b) { return a+b*a; }
```

Preprocessing:

```
$ clang ~/t.c -E
# 1 "/Users/sabre/t.c" 1

typedef float V __attribute__((vector_size(16)));

V foo(V a, V b) { return a+b*a; }
```

Type checking:

```
$ clang -fsyntax-only ~/t.c
```

GCC options:

```
$ clang -fsyntax-only ~/t.c -pedantic
/Users/sabre/t.c:2:17: warning: extension used
```

```
typedef float V __attribute__((vector_size(16)));
                ^
1 diagnostic generated.
```

Pretty printing from the AST:

Note, the `-cc1` argument indicates the compiler front-end, and not the driver, should be run. The compiler front-end has several additional Clang specific features which are not exposed through the GCC compatible driver interface.

```
$ clang -cc1 ~/t.c -ast-print
typedef float V __attribute__(( vector_size(16) ));
V foo(V a, V b) {
   return a + b * a;
}
```

Code generation with LLVM:

```
$ clang ~/t.c -S -emit-llvm -o -
define <4 x float> @foo(<4 x float> %a, <4 x float> %b) {
entry:
         %mul = mul <4 x float> %b, %a
         %add = add <4 x float> %mul, %a
         ret <4 x float> %add
}
$ clang -fomit-frame-pointer -O3 -S -o - t.c # On x86_64
...
_foo:
Leh_func_begin1:
        mulps   %xmm0, %xmm1
        addps   %xmm1, %xmm0
        ret
Leh_func_end1:
```