

GCC Cross-Compiler

From OSDev Wiki

In this tutorial we will create a GCC cross-compiler for your own operating system. This compiler is specially made to target exactly your operating system and is what allows you to leave the current operating system behind. You *need* a cross-compiler for operating systems development, unless you are developing on your own operating system.

Difficulty level



Beginner

Contents

- 1 Introduction
 - 1.1 Why do I need a Cross Compiler?
 - 1.2 Which compiler version do I want?
 - 1.3 Which binutils version do I want?
 - 1.4 Deciding on the target platform
- 2 Preparing for the build
 - 2.1 Downloading the Source Code
 - 2.2 Linux Users
 - 2.3 Mac OS X Users
 - 2.4 Windows Users
- 3 The Build
 - 3.1 Preparation
 - 3.2 Binutils
 - 3.3 GCC
- 4 Using the new Compiler
- 5 Troubleshooting
- 6 See Also
 - 6.1 Articles
 - 6.2 External Links
 - 6.3 Prebuilt Toolchains

Introduction

Generally speaking, a cross-compiler is a compiler that runs on platform A (the **host**), but generates executables for platform B (the **target**). These two platforms may (but do not need to) differ in CPU, operating system, and/or executable

format. In our case, the host platform is your current operating system, and the target platform is the operating system you are about to make. It is important to realize that these two platforms are not the same; your operating system is always going to be different from your current operating system. This is why we need to build a cross-compiler first, you will most certainly run into trouble otherwise.

Why do I need a Cross Compiler?

Main article: [Why do I need a Cross Compiler?](#)

You need to use a cross-compiler *unless* you are developing on your own operating system. The compiler must know the correct target platform (CPU, operating system), otherwise you will run into trouble. If you use the compiler that comes with your system, then the compiler won't know it is compiling something else entirely. Some tutorials suggest using your system compiler and passing a lot of problematic options to the compiler. This will certainly give you a lot of problems in the future and the solution is build a cross-compiler. If you have already attempted to make an operating system without using a cross-compiler, please read the article [Why do I need a Cross Compiler?](#).

Which compiler version do I want?

Main article: [Building GCC](#)

The newest GCC is recommended as it is the latest and greatest release. However, it is recommended that you use the same major compiler version to build your cross-compiler. For instance, you may run into trouble if you use gcc 4.6.3 to build a gcc 4.8.0 cross-compiler. If you are not using the latest major GCC release for your system compiler, we recommend that you build the newest GCC as your system compiler.

You can also use older releases as they are usually reasonably good. If your local system compiler isn't too terribly old (at least gcc 4.6.0), you may wish to save yourself the trouble and just pick the latest minor release (such as 4.6.3 if your system compiler is 4.6.1) for your cross-compiler.

You can view your current compiler version by invoking:

```
gcc --version
```

You may be able to use an older major GCC release to build a cross-compiler of a newer major GCC releaser. For instance, gcc 4.7.3 may be able to build a gcc 4.8.0 cross-compiler. However, if you want to use the latest and greatest gcc version for your cross-compiler, we recommend that you bootstrap the newest gcc as your

system compiler first. Individuals using Mac OS X 10.7 or earlier might want to invest in either building a system GCC (that outputs native Mach-O), or upgrading the local llvm/clang installation. Users with 10.8 and above should install the Command Line Tools from Apple's developer website and use clang to cross-compile gcc.

Which binutils version do I want?

Main article: Cross-Compiler Successful Builds

We recommend that you use the latest and greatest binutils release. Note, however, that not all combinations of GCC and binutils work. If you run into trouble, use a binutils that was released at roughly the same time as your desired compiler version. You probably need at least binutils 2.22, or preferably the latest 2.23.2 release. It doesn't matter what binutils version you have installed on your current operating system.

Deciding on the target platform

Main article: Target Triplet

You should already know this. If you are following the Bare Bones tutorial, you wish to build a cross-compiler for `i686-elf`.

Preparing for the build

The GNU Compiler Collection is an advanced piece of software with dependencies. You need to install certain dependencies in order to build gcc. You need to install GNU make, GNU bison, flex, and of course an existing system compiler you wish to replace. In addition, you also *need* the packages GNU GMP, GNU MPFR, and MPC that are used by GCC for floating point support.

You need a host system with a working GCC installation, and enough memory as well as hard drive space. How much qualifies as "enough" is depending on the versions of the software involved, but GCC is a big piece of software, so don't be surprised when 128 or 256 MByte are not sufficient.

In short you need the following that you can install manually or through package management:

- An Unix-like environment (Windows users)
- GCC (existing release you wish to replace)
- G++ (if building a version of GCC \geq 4.8.0)
- GNU Make
- GNU Bison

- Flex
- GNU libiconv (Mac OS X users)
- GNU GMP
- GNU MPFR
- GNU MPC
- ISL (optional)
- CLoog (optional)

Downloading the Source Code

You can download the desired binutils release by visiting the binutils website (<https://gnu.org/software/binutils/>) or directly accessing the GNU main ftp mirror (<ftp://ftp.gnu.org/gnu/binutils/>) .

You can download the desired gcc release by visiting the GCC website (<https://gnu.org/software/gcc/>) or directly accessing the GNU main ftp mirror (<ftp://ftp.gnu.org/gnu/gcc/>) .

In addition, to build GCC you need to have installed GNU GMP, GNU MPFR, GNU MPC and the ISL library. Additionally, Mac OS X users need a replacement libiconv because the system libiconv is seriously out of date. You may already have these libraries and the development files installed, but this tutorial builds them as part of GCC. If you don't need this, simply don't build them as part of GCC. Note that not all GMP, MPFR and MPC combinations are compatible with a given GCC release.

You can download GNU GMP from its website (<http://gmplib.org/>) . (libgmp3-dev on apt-based systems, dev-libs/gmp on Gentoo, gmp-devel on Fedora, libgmp-devel on Cygwin)

You can download GNU MPFR from its website (<http://mpfr.org/>) . (libmpfr-dev on apt-based systems, dev-libs/mpfr on Gentoo, mpfr-devel on Fedora, libmpfr-devel on Cygwin)

You can download ISL from its website (<http://isl.gforge.inria.fr/>) (optional). (libisl-dev on apt-based systems)

You can download CLoog from its website (<http://www.cloog.org/>) (optional). (libcloog-isl-dev on apt-based systems)

You can download GNU MPC from its website (<http://multiprecision.org/>) . (libmpc-dev on apt-based systems, dev-libs/mpc on Gentoo, libmpc-devel on Fedora, libmpc-devel on Cygwin)

Mac OS X users can download the latest libiconv release by visiting the libiconv website (<https://gnu.org/software/libiconv/>) or directly accessing the GNU main ftp mirror (<ftp://ftp.gnu.org/gnu/libiconv/>) .

In my experience, texinfo seems to be a required package for building some of the binutils (in particular, "as"). The cross-compiler may still build, but you might get weird errors when the compiler calls "as" and gets the system one instead. Even worse, you might not. The same problem was also discovered here (<https://github.com/Overv/MineAssemble/pull/5>) .

Download the needed source code into a suitable directory such as `$HOME/src`.

Note: The versioning scheme used is that each fullstop separates a full number, i.e. binutils 2.20.0 is newer than 2.9.0. This may be confusing, if you have not encountered this (quite common) versioning scheme yet, when looking at an alphanumerically sorted list of tarballs: The file at the bottom of the list is not the latest version! An easy way of getting the latest version is to sort by the last modified date and scrolling to the bottom.

Note: Version 5.x (or later) of texinfo is known to be incompatible with the current binutils 2.23.2 release (and older). You can check your current version using `makeinfo --version`. If your version is too new and you encounter problems during the build, you will need to either use binutils 2.24 release (or newer) or install an older version of texinfo - perhaps through building from source - and add it to your `PATH` prior and during the binutils build.

Linux Users

Your distribution may ship its own patched GCC and Binutils that is customized to work on your particular Linux distribution. You may not be able to build a functional system compiler using the upstream sources you downloaded above. In that case, try a newer GCC release or get the patched source code. For instance, some GCC releases are known to not understand the new Debian multiarch directory structure. However, if the compiler we are about to build is a cross-compiler targetting another operating system (such as your new one), then this is much less a worry.

Mac OS X Users

When compiling GCC 4.3 or higher on OS X 10.4 and 10.5, you may get unresolved symbol errors related to libiconv. This is because the version shipped with OS X is seriously out of date. Install a new version (compile it yourself or use macports) and add `--with-libiconv-prefix=/opt/local` (or `/usr/local` if you compiled it yourself) to GCC's `./configure` line. Alternatively you may place the libiconv source as `gcc-x.y.z/libiconv` and it will be compiled as part of the GCC compilation process. (This trick also works for mpfr, gmp, and mpc).

The makefiles of binutils and GCC use the `$(CC)` variable to invoke the compiler. On OS X, this resolves to `gcc` by default, which is actually not the "real thing", but

clang. Note that since at least OS X 10.8, Xcode's Command Line Tools package comes with clang, and this version of clang does indeed work to compile a working version of GCC, unlike what these instructions previously reflected.

Note that users running OS X 10.7 may need to find and install gcc, either from homebrew (<http://brew.sh>) , or from somewhere on Apple's website. Thus, the instructions below are really only relevant for these users, but your mileage may vary.

```
# This is only necessary for OS X users running 10.7 or below.  
export CC=/usr/bin/gcc-4.2  
export CXX=/usr/bin/g++-4.2  
export CPP=/usr/bin/cpp-4.2  
export LD=/usr/bin/gcc-4.2
```

You might want to unset these exports once you compiled and installed the cross compiler, as it might confuse other builds. **Do not** make these permanent!

Note for Lion users: If you're on Lion (or above) chances are that you don't have the "real" gcc since Apple removed it from the Xcode package, but you can still install it. You can do it via Homebrew or by compiling from source, both are perfectly described on a StackExchange answer (<http://apple.stackexchange.com/a/38247>) .

Note for Maverick users: You can build binutils-2.24 and gcc-4.8.3 (possible other version) with xcode 5.1.1. Note that building gcc with llvm is not officially supported and may cause interesting bugs, if you are willing to take this risk and save time building host-gcc just to compile a cross-gcc, follow this. Install GMP, MPFR, MPC with macport (<http://http://www.macports.org/>) .

```
sudo port install gmp mpfr libmpc
```

```
./binutils-2.24/configure --prefix=$PREFIX \  
--target=$TARGET \  
--enable-interwork --enable-multilib \  
--disable-nls --disable-werror
```

```
./gcc-4.8.3/configure --prefix=$PREFIX \  
--target=$TARGET \  
--disable-nls \  
--enable-language=c,c++ --without-headers \  
--enable-interwork --enable-multilib \  
--with-gmp=/usr --with-mpc=/opt/local --with-mpfr=/opt/local
```

- Note that there is issue with port's gmp, we use the version from mac os x

from /usr instead.

Windows Users

Windows users need to set up a Unix-like environment such as MinGW or Cygwin. It may well be worth looking into systems such as Linux and see if they fit your needs, as you commonly use a lot of Unix-like tools in operating systems development and this is much easier from a Unix-like operating system. If you have just installed the basic Cygwin package, you have to run the setup.exe again and install the following packages: GCC, Make, Flex, Bison, and Diffutils.

MinGW + MSYS is an option, and as it addresses the native Windows API instead of a POSIX emulation layer, results in a slightly faster toolchain. Some software packages will not build properly under MSYS as they were not designed for use with Windows. As far as this tutorial is concerned, everything that applies to Cygwin also applies to MSYS unless otherwise specified. Make sure you install the C and C++ compilers, and the MSYS Basic System.

Cygwin note: Cygwin includes your Windows %PATH% in its bash \$PATH. If you were using DJGPP before, this could result in confusion as e.g. calling gcc on the Cygwin bash command line would still call the DJGPP compiler. After uninstalling DJGPP, you should delete the DJGPP environment variable and clear the c:\djgpp entry (or wherever you installed it) from your %PATH%. Likewise, it might be a bad idea to mix build environments in your system PATH variable.

MinGW note: Some MinGW-specific information on building a cross-toolchain can be found on the hosted cross-compiler how-to page (<http://www.mingw.org/wiki/HostedCrossCompilerHOWTO>) on the MinGW homepage.

The Build

We build a toolset running on your host that can turn source code into object files for your target system.

You need to decide where to install your new compiler. It is dangerous and a very bad idea to install it into system directories. You also need to decide whether the new compiler should be installed globally or just for you. If you want to install it just for you (recommended), installing into \$HOME/opt/cross is normally a good idea. If you want to install it globally, installing it into /usr/local/cross is normally a good idea.

Please note that we build everything out of the source directory tree, as is considered good practice. Some packages only support building outside, some only inside and some both (but may not offer extensive checking with make). Building GCC inside the source directory tree fails miserably, at least for older

versions.

Preparation

```
export PREFIX="$HOME/opt/cross"  
export TARGET=i686-elf  
export PATH="$PREFIX/bin:$PATH"
```

We add the installation prefix to the `PATH` of the current shell session. This ensures that the compiler build is able to detect our new binutils once we have built them.

The prefix will configure the build process so that all the files of your cross-compiler environment end up in `$HOME/opt/cross`. You can change that prefix to whatever you like (e.g., `/opt/cross` or `$HOME/cross` would be options). If you have administrator access and wish to make the cross-compiler toolchain available to all users, you can install it into the `/usr/local` prefix - or perhaps a `/usr/local/cross` prefix if you are willing to change the system configuration such that this directory is in the search paths for all users. Technically, you could even install directly to `/usr`, so that your cross-compiler would reside alongside your system compiler, but that is not recommended for several reasons (like risking to overwrite your system compiler if you get `TARGET` wrong, or getting into conflict with your system's package management).

Binutils

```
cd $HOME/src  
mkdir build-binutils  
cd build-binutils  
../binutils-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --v  
make  
make install
```

This compiles the binutils (assembler, disassembler, and various other useful stuff), runnable on your system but handling code in the format specified by `$TARGET`.

--disable-nls tells binutils not to include native language support. This is basically optional, but reduces dependencies and compile time. It will also result in English-language diagnostics, which the people on the Forum (<http://forum.osdev.org/>) understand when you ask your questions. ;-)

--with-sysroot tells binutils to enable sysroot support in the cross-compiler by

pointing it to a default empty directory. By default the linker refuses to use sysroots for no good technical reason, while gcc is able to handle both cases at runtime. This will be useful later on.

GCC

See also the official instructions for configuring gcc (<http://gcc.gnu.org/install/configure.html>) .

Now, you can build GCC.

```
cd $HOME/src

# If you wish to build these packages as part of gcc:
mv libiconv-x.y.z gcc-x.y.z/libiconv # Mac OS X users
mv gmp-x.y.z gcc-x.y.z/gmp
mv mpfr-x.y.z gcc-x.y.z/mpfr
mv mpc-x.y.z gcc-x.y.z/mpc

mkdir build-gcc
cd build-gcc
../gcc-x.y.z/configure --target=$TARGET --prefix="$PREFIX" --disabl
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

We build libgcc, a low-level support library that the compiler expects available at compile time. Linking against libgcc provides integer, floating point, decimal, stack unwinding (useful for exception handling) and other support functions. Note how we are *not* simply running `make && make install` as that would build way too much, not all components of gcc is ready to target your unfinished operating system.

--disable-nls is the same as for binutils above.

--without-headers tells GCC not to rely on any C library (standard or runtime) being present for the target.

--enable-languages tells GCC not to compile all the other language frontends it supports, but only C (and optionally C++).

It will take a while to build your cross-compiler.

Using the new Compiler

Now you have a "naked" cross-compiler. It does not have access to a C library or C runtime yet, so you cannot use any of the standard includes or create runnable binaries. But it is quite sufficient to compile the kernel you will be making shortly. Your toolset resides in `$HOME/opt/cross` (or what you set `$PREFIX` to). For example, you have a `gcc` executable installed as `$HOME/opt/cross/bin/$TARGET-gcc`, which creates programs for your `TARGET`.

You can now run your new compiler by invoking something like:

```
$HOME/opt/cross/bin/$TARGET-gcc --version
```

Note how this compiler is *not* able to compile normal C programs. The cross-compiler will spit errors whenever you want to `#include` any of the standard headers (except for a select few that actually are platform-independent, and generated by the compiler itself). This is quite correct - you don't have a standard library for the target system yet!

The C standard defines two different kinds of executing environments - "freestanding" and "hosted". While the definition might be rather fuzzy for the average application programmer, it is pretty clear-cut when you're doing OS development: A kernel is "freestanding", everything you do in user space is "hosted". A "freestanding" environment needs to provide only a subset of the C library: `float.h`, `iso646.h`, `limits.h`, `stdarg.h`, `stdbool.h`, `stddef.h`, and `stdint.h` (as of C99). All of these consist of typedefs and `#define`s "only", so you can implement them without a single `.c` file in sight.

To use your new compiler simply by invoking `$TARGET-gcc`, add `$HOME/opt/cross/bin` to your `$PATH` by typing:

```
export PATH="$HOME/opt/cross/bin:$PATH"
```

This command will add your new compiler to your `PATH` for this shell session. If you wish to use it permanently, add the `PATH` command to your `~/.profile` configuration shell script or similar. Consult your shell documentation for more information.

You can now move on to complete the Bare Bones tutorial variant that lead you here and complete it using your new cross-compiler. If you built a new GCC version as your system compiler and used it to build the cross-compiler, you can now safely uninstall it unless you wish to continue using it.

Troubleshooting

In general, **verify** that you typed the "make" commands precisely. If your shell crashes during a "make" *you need to re-enter any PATH export* before running "make" again. If a compilation seems to have gotten really messed up, type "make clean", and then start the make process over again (but you probably do not need to rerun "configure").

line 11: \$'\r': command not found

You unpacked the source with WinZIP or something else that converts all text files to the Windows CRLF format. Use 7-Zip or the Cygwin tar program (tar xjf packagename.tar.bz2) to do your unpacking.

Note: On some browsers, when you download a "tarball" that ends in ".tar.bz2", your browser may rename it to ".tar.tar" -- don't worry, it will still work, just rename it back to ".tar.bz2".

i686-elf-ar not found

You forgot to set the executable path (\$PATH) to include \$PREFIX/bin.

Error: junk at end of line, first unrecognized character is ','

This, in combination with lots of other assembly-level error messages (like, *Warning: .type pseudo-op used outside of .def/.endef ignored*, or *Error: unknown pseudo-op: '.local'*) results when you did not correctly set the --prefix=\$PREFIX during the binutils configure.

Another possibility is that you did configure, compile and install your cross-compiler correctly, but don't actually use it. Check the "Usage" section above.

If you try compiling in 64-bit windows, you will receive a "Unknown host machine type" error when running configure. To fix this, scroll up in your shell until right after you entered the configure command and you will see a website which will show you where to download updated files to guess host type. Put them in the root directory of where your source files are located. With GCC version 3.4.0 you will have to override the host environment though, as it does not support being compiled with x86_64-unknown-cygwin . Add the command line argument --host=i686-unknown-cygwin to the configure line for GCC. --CjMovie (Too many edits to get this right...)

Configure: error: invalid feature name: nls

If you are using Cygwin, it must be set to use Unix/binary as the file mode. You can set this by running setup.exe and selecting the appropriate mode.

*** Multiple patterns ...

This error can occur when you're using Cygwin (it could also occur on other instances). The most common cause is the cross-usage of UNIX-style paths and DOS-like paths (e.g. /usr/src and C:\MyFolder). If you were using a custom directory (e.g. C:\Cross-Compiler) to build the cross-compiler instead of the /usr/src directory, usually using the default folders (/usr/src, /usr/...) fixes these problems.

But I'm using Cygwin on Windows and I don't have any \$HOME/src folder on my hard-drive!

If you didn't notice, Cygwin is a tool that works on Windows but acts UNIX-like, so if you run the 'Cygwin.bat' file in your Cygwin folder, you will get the BASH shell. If you snoop around in your Cygwin folder, you'll see a **usr** folder with a **src** folder inside (these folders will be used by Cygwin if you follow this tutorial).

C++ preprocessor "/lib/cpp" fails sanity check in Cygwin

In Cygwin's installer, you need to separately select the gcc4-core and gcc4-g++ packages for proper configuration of the host compiler.

ld: cannot find -lgcc

You specified that you want to link the GCC low-level runtime library into your executable through the *-lgcc* switch, but forgot build and properly install the library.

Configuring for a x86_64-unknown-linux-gnu host.

If you'r running a 64bit version of linux you may get this error

Configuring for a x86_64-unknown-linux-gnu host

Invalid configuration `x86_64-unknown-linux-gnu': machine `x86_64-unknown' not recognized

just use linux32 before all of your commands, example : "linux32 ./configure" , "linux32 make" , and so on.

See Also

Articles

- Canadian Cross - making things yet more complicated.
- Cross-Compiler Successful Builds - combinations of GCC and Binutils which have been shown to work with this tutorial by OSDev.org members.

- GCC Cross-Compiler on Linux - some distros make things much easier.
- Target Triplet - on target triplets and their use
- OS Specific Toolchain - going a step further and adding your own target.
- LLVM Cross-Compiler - some compilers make things much easier.

External Links

- <http://kegel.com/crosstool> has a popular example of a script that automatically downloads, patches, and builds binutils, gcc, and glibc for known platforms.
- <http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html> - Summary of the support functions you get when you link with libgcc.
- <http://forums.gentoo.org/viewtopic.php?t=66125> - Compiling Windows applications under Linux
- <http://www.libsdl.org/extras/win32/cross/README.txt> - dito
- <https://github.com/travisg/toolchains> - Another script for building simple cross compilers

Prebuilt Toolchains

These were built by people in the OSdev community for their own building needs and shared at will, without guaranteeing any support or that it will even work on your setup. YMMV.

For Linux x86_64 host

- i386-elf 4.9.1 target (http://newos.org/toolchains/i386-elf-4.9.1-Linux-x86_64.tar.xz)
- i686-elf 4.9.1 target (http://newos.org/toolchains/i686-elf-4.9.1-Linux-x86_64.tar.xz)
- x86_64-elf 4.9.1 target (http://newos.org/toolchains/x86_64-elf-4.9.1-Linux-x86_64.tar.xz)
- aarch64-elf 4.9.1 target (http://newos.org/toolchains/aarch64-elf-4.9.1-Linux-x86_64.tar.xz)
- arm-eabi 4.9.1 target (http://newos.org/toolchains/arm-eabi-4.9.1-Linux-x86_64.tar.xz)
- m68k-elf 4.9.1 target (http://newos.org/toolchains/m68k-elf-4.9.1-Linux-x86_64.tar.xz)
- powerpc-elf 4.9.1 target (http://newos.org/toolchains/powerpc-elf-4.9.1-Linux-x86_64.tar.xz)
- sparc-elf 4.9.1 target (http://newos.org/toolchains/sparc-elf-4.9.1-Linux-x86_64.tar.xz)
- sh-elf 4.9.1 target (http://newos.org/toolchains/sh-elf-4.9.1-Linux-x86_64.tar.xz)

For Linux i686 host

- i386-elf-binutils 2.24 (<http://phillid.tk/r/i686/i386-elf-binutils-2.24-4-i686.pkg.tar.xz>)
- i386-elf-gcc 4.9.1 (<http://phillid.tk/r/i686/i386-elf-gcc-4.9.1-4-i686.pkg.tar.xz>)
- i686-elf-binutils 2.24 (<http://phillid.tk/r/i686/i686-elf-binutils-2.24-4-i686.pkg.tar.xz>)
- i686-elf-gcc 4.9.1 (<http://phillid.tk/r/i686/i686-elf-gcc-4.9.1-4-i686.pkg.tar.xz>)
- arm-eabi-binutils 2.24 (<http://phillid.tk/r/i686/arm-eabi-binutils-2.24-4-i686.pkg.tar.xz>)
- arm-eabi-gcc 4.9.1 (<http://phillid.tk/r/i686/arm-eabi-gcc-4.9.1-4-i686.pkg.tar.xz>)

The packages from phillid.tk have been shrunk to about 10 MiB for each pair of packages (GCC & Binutils). Please note that this has been achieved by enabling only the C front-end for GCC. If you're going to write your OS in any language but C or assembly, these packages aren't for you. These are actually Pacman packages, but untarring them to / and rm-ing /.MTREE and other clutter dotfiles contained in the package will work the same.

For Windows host

- i686-elf 4.8.2 target (https://drive.google.com/file/d/0B85K_c7mx3QjUnZuaFRPWlBIcXM/edit?usp=sharing)

For OSX host

- i686-pc-elf 4.6.1 target with clang for x86_64 OSX host (http://downloads.exquance.com/toolchain-x86_64-darwin.tar.bz2)
- x86_64-pc-elf Cross Compiler setup with GCC and Binutils for x86_64 OSX (<https://docs.google.com/file/d/0BxDNp6DGu6SZcmlHVWpNblRnWWs/edit?usp=sharing>)

Retrieved from "http://wiki.osdev.org/index.php?title=GCC_Cross-Compiler&oldid=17035"

Categories: Level 1 Tutorials | Compilers | Tutorials

- This page was last modified on 12 November 2014, at 17:46.
- This page has been accessed 350,427 times.