CC compiler command

Tongust

April 9, 2017

Summary

1 The environment variables for GCC/G++

Each variables value is a list of directories separated by a special character, much like PATH, in which to look for header files. The special character, PATH_SEPARATOR, is target-dependent and determined at GCC build time. For Microsoft Windows-based targets it is a semicolon, and for almost all other targets it is a colon.

1.1 Search xx.h file

CPATH specifies a list of directories to be searched as if specified with $-I$. This environment variable is used regardless of which language is being preprocessed. Other characters used as ENV: **C_INCLUDE_PATH CPLUS_INCLUDE_PATH**.
Keep in maind: pls use the **-I** instead of modifing the ENV of **CPATH**.

For example, we have 4 ways to modify this **CPATH**.

- Set ENV **only** for current shell:
  CPATH=/your/include/path/:$CPATH

- Set ENV for current shell and **all processes** started from current shell:
  export CPATH=/your/include/path/:$CPATH
- Set ENV **permanently** for all future bash sessions:
  export CPATH=/your/include/path/:$CPATH $\gg$ */home/your/.bashrc*
- Set ENV permanently and **system wide**(all users and processes), just add to the /etc/environment:
  # gksudo gedit /etc/environment

You can check the default CPATH with cmd: cpp -v

```
#include "..." search starts here:
#include <...> search starts here:
/usr/lib/gcc/x86_64-linux-gnu/5/include
/usr/local/include
/usr/lib/gcc/x86_64-linux-gnu/5/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
End of search list.
```

Ref: gnu.org

1.2 *-iquote* Compiler Flag

GCC actually does allow you to modify this search path as well, with the -iquote compiler flag.

```
$ cpp -iquote hdr1 -v
...
#include "..." search starts here:
 hdr1
#include <...> search starts here:
```

```
/usr/local/include
/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include
/usr/lib/gcc/x86_64-linux-gnu/4.4.5/include-fixed
/usr/include/x86_64-linux-gnu
/usr/include
```

1.3 Understanding the "extern" keyword in C

extern is used for variable and function to just declare them.
As for function: the synatex "int foo(int a);" is treated as by the compiler as "extern int foo(int a);" However the situation is different with variable. You must explicitly prepend the variable with "extern": extern int var;. And the int var; means that the var has been declared as well as defined.
Using the variable that is only declared is undefined.

```
#include <iostrem>
extern int var;
/* Error */
int main() {
    var = 10;
    return 0;
}
/* Success */
int var  = 0; // Only define once.
int main() {
    var = 10;
    return 0;
}
```

1.4 Shared libraries with GCC on Linux

- Step 1: Compiling with Position Independent Code
  We need to compile our library source code into position-independent code (PIC)
  $ gcc -c -Wall -Werror -fpic foo.c

- Step 2: Creating a shared library from an object file
  Now we need to actually turn this object file into a shared library. Well

call it libfoo.so:
gcc -shared -o libfoo.so foo.o

- Step 3: Linking with a shared library
  As you can see, that was actually pretty easy. We have a shared library. Lets compile our main.c and link it with libfoo. Well call our final program test. Note that the -lfoo option is not looking for foo.o, but libfoo.so. GCC assumes that all libraries start with lib and end with .so or .a (**.so is for shared object or shared libraries, and .a is for archive, or statically linked libraries**).
  Telling GCC where to find the shared library
  $ gcc -L/home/username/foo -Wall -o test main.c -lfoo;

- Step 4: Making the library available at runtime
  Good, no errors. Now lets run our program:
  $ ./test ./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory.
  Oh no! The loader cant find the shared library.3 We didnt install it in a standard location, so we need to give the loader a little help. We have a couple of options: we can use the environment variable LD_LIBRARY_PATH for this, or rpath. Lets take a look first at LD_LIBRARY_PATH:

- **Using LD_LIBRARY_PATH**
  $ LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
  $ ./test ./test: error while loading shared libraries: libfoo.so: cannot open shared object file: No such file or directory
  What happened? Our directory is in LD_LIBRARY_PATH, but we didnt export it. In Linux, if you dont export the changes to an environment variable, they wont be inherited by the child processes. The loader and our test program didnt inherit the changes we made. Thankfully, the fix is easy:
  $ export LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH
  $ ./test

- **Using rpath**
  Now lets try rpath (first well clear LD_LIBRARY_PATH to ensure its rpath thats finding our library). Rpath, or the run path, is a way of embedding the location of shared libraries in the executable itself, instead of relying on default locations or environment variables. We do this during the linking stage. Notice the lengthy -Wl,-rpath=/home/username/foo

option. The -Wl portion sends comma-separated options to the linker, so we tell it to send the -rpath option to the linker with our working directory.

```
$ unset LD_LIBRARY_PATH
$ gcc -L/home/username/foo -Wl,-rpath=/home/username/foo -Wall
                                                -o test main.c -lfoo
$ ./test
This is a shared library test...
Hello, I'm a shared library
------
Excellent, it worked. The rpath method is
great because each program gets to list its
shared library locations independently,
so there are no issues with different
programs looking in the wrong paths like
there were for LD_LIBRARY_PATH.
```

- **Using ldconfig to modify ld.so** What if we want to install our library so everybody on the system can use it? For that, you will need admin privileges. You will need this for two reasons: first, to put the library in a standard location, probably /usr/lib or /usr/local/lib, which normal users dont have write access to. Second, you will need to modify the ld.so config file and cache. As root, do the following:
  $ cp /home/username/foo/libfoo.so /usr/lib $ chmod 0755 /usr/lib/libfoo.so
  Now the file is in a standard location, with correct permissions, readable by everybody. We need to tell the loader its available for use, so lets update the cache:
  $ ldconfig That should create a link to our shared library and update the cache so its available for immediate use. Lets double check:
  $ ldconfig -p — grep foo libfoo.so (libc6) =¿ /usr/lib/libfoo.so

- Node
  1. What is position independent code? PIC is code that works no matter where in memory it is placed. Because several different programs can all use one instance of your shared library, the library cannot store things at fixed addresses, since the location of that library in memory will vary from program to program.

2. GCC first searches for libraries in /usr/local/lib, then in /usr/lib. Following that, it searches for libraries in the directories specified by the -L parameter, in the order specified on the command line.
3. The default GNU loader, ld.so, looks for libraries in the following order:
3.1 It looks in the DT_RPATH section of the executable, unless there is a DT_RUNPATH section.
3.2 It looks in LD_LIBRARY_PATH. This is skipped if the executable is setuid/setgid for security reasons.
3.3 It looks in the DT_RUNPATH section of the executable unless the setuid/setgid bits are set (for security reasons).
3.4 It looks in the cache file /etc/ld/so/cache (disabled with the -z nodeflib linker option).
3.5 It looks in the default directories /lib then /usr/lib (disabled with the -z nodeflib linker option).

1.5 Flags for GCC

-I: Include header files xx.h directive
-L: Library directive (1. shared library/objective .so; 2. archive file .a).
-l: Link the library file (-lmylib). The compiler will search the libmylib.so under the path indicated by -I.

1.6 Create, View, Extract and Modify C Archive Files (*.a)

ar is an archive tool used to combine objects to create an archive file with .a extension, also known as library.
1. Compile the Programs and Get Object Codes. Use -c option to compile both the c program. Using option -c will create the corresponding .o files.
2. Create the C Program Static Library using ar utility. Now create the static library libarith.a with the addition object file and multiplication object file as follows
$ ar cr libmylib.a one.o two.o
3. Write C program to Use the Library libmylib.a. You just declear it and use its defination from libmylib.a without include it.
4. Complier it.
Option 1: $ gcc -Wall example.c -L/your/lib/pathname/ -lmylib -o example
Option 2: $ gcc -Wall example.c /pathname/libarith.a -o example. Just explicitly put together the pathname in which the *.a files exist;

List of Listings