CSO-Tutorial

Some useful GCC command-line options.

This tutorial will use the program below called main.c -

```
#include<stdio.h> int main() { printf("\n CSO First tutorial \n"); return 0;
}
```

Use the command below to compile the program

```
gcc main.c
```

The above command executes the complete compilation process and outputs an executable with the name a.out

1. Use option -o, as shown below, to specify the output file name for the executable.

```
gcc main.c -o main
```

2. Enable all warnings set through -Wall option

This option enables all the warnings in GCC

For example for the code below -

```
#include<stdio.h> int main(void) { int i; printf("\n Welcome to CSO Tute-%d
\n",i+1); return 0; }
```

```
$ gcc -Wall main.c -o main main.c: In function ‘int main()’: main.c:6:10:
warning: ‘i’ is used uninitialized in this function [-Wuninitialized] 6 |
printf("\n Welcome to CSO Tute-%d \n",i+1); |
~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

3. Produce only the pre-processor output with -E option

```
$ gcc -E main.c > main.i
```

The gcc command produces the output on stdout so you can redirect the output in any file. In our case (above), the file main.i would contain the preprocessed output.

4. Produce only the assembly code using -S option

```
gcc -S main.c
```

In this case, the file main.s would contain the assembly output.

5. Produce only the compiled code using the -c option

To produce only the compiled code (without any linking), use the -c option.

```
gcc -C main.c
```

The command above would produce a file main.o that would contain machine-level code or the compiled code.

To view the contents of .o file use the command

```
objdump -d filename.o
```

6. Produce all the intermediate files using the -save-temps function

To look at all the intermediary files generated during the compilation processes use the command below -

```
gcc -Wall -save-temps main.c -o main
```

For example :

```
$ gcc -save-temps main.c $ ls a.out main.c main.i main.o main.s
```

So we see that all the intermediate files as well as the final executable were produced in the output.

The option -save-temps can do all the work done in earlier examples above. Through this option, output of all the stages of compilation is stored in the current directory. Please note that this option produces the executable also.

7. Link with shared libraries using the -l option

foo.h

```
#ifndef foo_h__ #define foo_h__ extern void foo(void); #endif // foo_h__
```

foo.c

```
#include <stdio.h> void foo(void) { printf("Hello, I am a shared library");
}
```

main.c

```
#include <stdio.h> #include "foo.h" int main(void) { printf("This is a shared library test..."); foo(); return 0; }
```

foo.h defines the interface to our library, a single function, foo(). foo.c contains the implementation of that function, and main.c is a driver program that uses our library.

For the sake of this example, everything will happen in /home/username/foo

Step 1: Compiling with Position Independent Code

We need to compile our library source code into position-independent code

```
gcc -c -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. We will 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. Let us compile our main.c and link it with libfoo. We will 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).

```
$ gcc -Wall -o test main.c -lfoo /usr/bin/ld: cannot find -lfoo collect2: ld returned 1 exit status
```

Telling GCC where to find the shared library

Uh-oh! The linker does not know where to find libfoo. GCC has a list of places it looks by default, but our directory is not in that list. We need to tell GCC where to find libfoo.so. We will do that with the -L option. In this example, we will assume the directory, /home/username/foo:

```
$ gcc -L/home/username/foo -o test main.c -lfoo
```

Step 4: Making the library available at runtime

Good, no errors. Now let us run our program:

```
$ ./test ./test: error while loading shared libraries: libfoo.so: cannot ope
n shared object file: No such file or directory
```

Oh no! The loader cannot find the shared library. We did not 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. Let us take a look first at LD_LIBRARY_PATH:

Using LD_LIBRARY_PATH

```
$ echo $LD_LIBRARY_PATH
```

There is nothing in there. Let us fix that by prepending our working directory to the existing LD_LIBRARY_PATH:

```
$ export LD_LIBRARY_PATH=/home/username/foo:$LD_LIBRARY_PATH $ ./test This
is a shared library test... Hello, I am a shared library
```

Good, it worked! LD_LIBRARY_PATH is great for quick tests and for systems on which you do not have admin privileges. As a downside, however, exporting the LD_LIBRARY_PATH variable means it may cause problems with other programs you run that also rely on LD_LIBRARY_PATH if you do not reset it to its previous state when you are done.

Using rpath

Now let s try rpath (first we will clear LD_LIBRARY_PATH to ensure it is rpath that is 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/usernam
e/foo -Wall -o test main.c -lfoo $ ./test This is a shared library test... H
ello, I am 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.

8. Use compile-time macros using -D option

The compiler option D can be used to define compile-time macros in code.

Here is an example :

```
#include<stdio.h> int main(void) { #ifdef MY_MACRO printf("\n Macro defined
\n"); #endif char c = -10; // Print the string printf("\n Welcome to CSO
Tute [%d]\n", c); return 0; }
```

The compiler option -D can be used to define the macro MY_MACRO from command line.

```
$ gcc -Wall -DMY_MACRO main.c -o main $ ./main Macro defined Welcome to CSO
Tute [-10]
```

9. Convert warnings into errors with -Werror option

Through this option, any warning that gcc could report gets converted into error.

Here is an example :

```
#include<stdio.h> int main(void) { char c; // Print the string printf("\n The Geek Stuff [%d]\n", c); return 0; }
```

The compilation of above code should generate warning related to undefined variable c and this should get converted into error by using -Werror option.

```
$ gcc -Wall -Werror main.c -o main main.c: In function 'main': main.c:7:4: error: 'c' is used uninitialized in this function [-Werror=uninitialized] printf("\n Welcome to CSO Tute [%d]\n", c); ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ cc1.exe: all warnings being treated as errors
```

10. Provide gcc options through a file using @ option

The options for gcc can also be provided through a file. This can be done using the @option followed by the file name containing the options. More than one option is separated by white space.

Here is an example :

```
$ cat opt_file -Wall -o main
```

The opt_file contains the options.

Now compile the code by providing opt_file along with option @

```
$ gcc main.c @opt_file main.c: In function ‘main’: main.c:6:11: warning: ‘i’ is used uninitialized in this function [-Wuninitialized] $ ls main main
```

The output confirms that file opt_file was parsed to get the options and the compilation was done accordingly.

 *** **The examples shown here are just representative, It is important to understand the use cases of command-line options (especially the ones related to creating custom shared library and compilation process.) in a larger context which you will by experience.** ***

Additional info:

To obtain a brief reminder of various command-line options, GCC provides a help option which displays a summary of the top-level GCC command-line options:

```
$ gcc --help
```

To display a complete list of options for `gcc` and its associated programs, such as the