15 Most Frequently Used GCC Compiler Command Line Options

GCC Compiler is a very powerful and popular C compiler for various Linux distributions. This article explains some of the popular GCC compiler options.

### An Example C Code

The following basic C code (main.c) will used in this article :

#include<stdio.h>

int main(void)

{

printf("\n The Geek Stuff\n");

return 0;

}

### GCC Compiler Options

### 1. Specify the Output Executable Name

In its most basic form, gcc compiler can be used as :

gcc main.c

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

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

gcc main.c -o main

The command above would produce an output file with name ‘main’.

To understand the complete compilation process of a GCC compiler, read our article [Journey of a C Program to Linux Executable in 4 Stages](http://www.thegeekstuff.com/2011/10/c-program-to-an-executable/).

### 2. Enable all warnings set through -Wall option

This option enables all the warnings in GCC.

#include<stdio.h>

int main(void)

{

int i;

printf("\n The Geek Stuff [%d]\n", i);

return 0;

}

If the above code is compiled, the following warning related to uninitialized variable i is produced :

$ gcc -Wall main.c -o main

main.c: In function â€˜mainâ€™:

main.c:6:10: warning: â€˜iâ€™ is used uninitialized in this function [-Wuninitialized]

### 3. Produce only the preprocessor output with -E option

The output of preprocessing stage can be produced using the -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

The assembly level output can be produced using the -S option.

gcc -S main.c > main.s

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.

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

The option -save-temps can do all the work done in example 4,5 and 6 above. Through this option, output at all the stages of compilation is stored in the current directory. Please note that this option produces the executable also.

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 was produced in the output.

### 7. Link with shared libraries using -l option

The option -l can be used to link with shared libraries. For example:

gcc -Wall main.c -o main -lCPPfile

The gcc command mentioned above links the code main.c with the shared library libCPPfile.so to produce the final executable ‘main’.

### 8. Create position independent code using -fPIC option

While creating the shared libraries, position independent code should be produced. This helps the shared library to get loaded as any address instead of some fixed address. For this -fPIC option is used.

For example, the following commands create a shared library libCfile.so from source file Cfile.c:

$ gcc -c -Wall -Werror -fPIC Cfile.c

$ gcc -shared -o libCfile.so Cfile.o

So we see that the option -fPIC was used in creation of a shared library.

### 9. Print all the executed commands using -V option

The option -v can be used to provide verbose information on all the steps gcc takes while compiling a source file.

For example :

$ gcc -Wall -v main.c -o main

Using built-in specs.

COLLECT\_GCC=gcc

COLLECT\_LTO\_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.6/lto-wrapper

Target: i686-linux-gnu

Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.6.3-1ubuntu5' --with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin --enable-objc-gc --enable-targets=all --disable-werror --with-arch-32=i686 --with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu

Thread model: posix

gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)

...

...

...

So we see that detailed information was produced in the output.

### 10. Enable the support of ISO C89 programs using -ansi option

Through -ansi option the support for ISO C89 style is enabled.

Consider the following code :

#include<stdio.h>

int main(void)

{

// Print the string

printf("\n The Geek Stuff\n");

return 0;

}

If the above code is compiled with -ansi option then gcc would produce an error because the C++ comments are not allowed in ISO C89 style.

Here is the output :

$ gcc -Wall -ansi main.c -o main

main.c: In function â€˜mainâ€™:

main.c:5:3: error: expected expression before â€˜/â€™ token

So we see that gcc threw an error related to commenting style.

### 11. Interpret char as unsigned char using -funsigned-char option

Through this option, the char type is treated as unsigned type.

Here is an example :

#include<stdio.h>

int main(void)

{

char c = -10;

// Print the string

printf("\n The Geek Stuff [%d]\n", c);

return 0;

}

When the above code is compiled with funsigned-char option, here is the output :

$ gcc -Wall -funsigned-char main.c -o main

$ ./main

The Geek Stuff [246]

So we see that the char was indeed treated as unsigned.

### 12. Interpret char as signed char using -fsigned-char option

This is the opposite of what we discussed in (12) above. Using this flag, the char variables are treated as signed.

Here is an example :

$ gcc -Wall -fsigned-char main.c -o main

$ ./main

The Geek Stuff [-10]

The output confirms that char was treated as signed.

### 13. 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 The Geek Stuff [%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

The Geek Stuff [-10]

The print related to macro in the output confirms that the macro was defined.

### 14. 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 â€˜mainâ€™:

main.c:7:10: error: â€˜câ€™ is used uninitialized in this function [-Werror=uninitialized]

cc1: all warnings being treated as errors

### 15. Provide gcc options through a file using @ option

The options to 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 options are separated by a white space.

Here is an example :

$ cat opt\_file

-Wall -omain

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.