Table of Contents

[GNU Tools(Linux) that Help To Develop C/C++ Applications 1](#__RefHeading___Toc356_1493665560)

[FAQs 9](#__RefHeading___Toc358_1493665560)

[How do I generate symbolic information for gdb and warning messages? 9](#__RefHeading___Toc360_1493665560)

[How do I generate optimized code on a Linux machine? 9](#__RefHeading___Toc362_1493665560)

[How do I compile a C program that uses math functions? 9](#__RefHeading___Toc364_1493665560)

[How do I compile a C++ program that uses Xlib graphics functions? 9](#__RefHeading___Toc366_1493665560)

[GNU Binutils 10](#__RefHeading___Toc368_1493665560)

[Obtaining binutils 11](#__RefHeading___Toc370_1493665560)

# GNU Tools(Linux) that Help To Develop C/C++ Applications

In the Linux environment, many GNU tools are installed by default when you select the C/C++ development category during installation, or use the package manager, post installation. Most of these tools are supported by packages like gcc, glibc, binutils etc. You can use the package manager of a specific distribution to install missing packages or, on rare occasions, you can build these from the source code.



To build the above program, test.c, we use the following command:

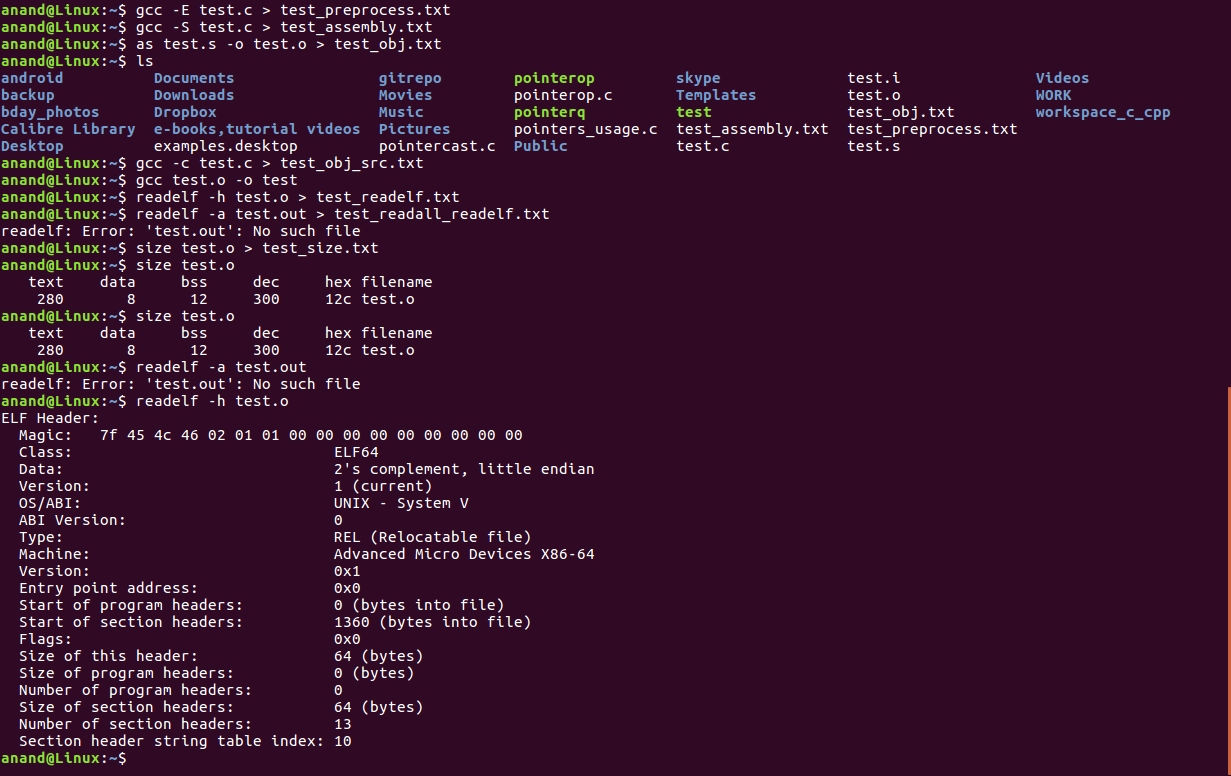
*##>> gcc program-source-code.c -o executable-file-name*

*>> gcc test.c -o test*

The above command produces an ELF executable bin

When you run the above command, have you ever thought of the various underlying phases of development? gcc and g++ (without options) undergo various phases like **preprocessing, assembling, linking, etc,** with the help of tools like **cpp, as, ld, etc.** So gcc and g++ act like wrappers for these tools.

To see each of these phases, let’s us try out some commands.



To see preprocessed output, use,

*>> gcc -E test.c #output comes on stdout by default*

Output as taken from the console:



To see preprocessed output, you can use the -E option of gcc which invokes the cpp command internally. Here, you can see symbolic constants are replaced by their values; comments are removed, macros are expanded and header file contents are included.



A preprocessed test.i file is an output of the C or C++ preprocessor. It is usually this extension which is characteristic of files created as the preprocessor output. The preprocessor performs primary transformations of the source program text using only [lexical analysis](https://www.viva64.com/en/t/0025/).

The preprocessor carries out the following operations:

* Inclusion of header files (the '#include' instruction);
* Macros substitution (the '#define' instruction);
* Conditional compilation (instructions '#if', '#ifdef', '#else', '#elif', '#endif').

The necessity of primary preprocessing of source files is determined by the fact that all the instructions '#include' and '#define' should be expanded to enable good static analysis.

To see Assembly output, use,

*>> gcc -S test.c #generates test.s*



*you can generate the object file from the generated assembly code, as follows:*

*>> as test.s -o test.o*

*>> gcc test.o -o test #generates executable/binary, a.out in absence of -o option*

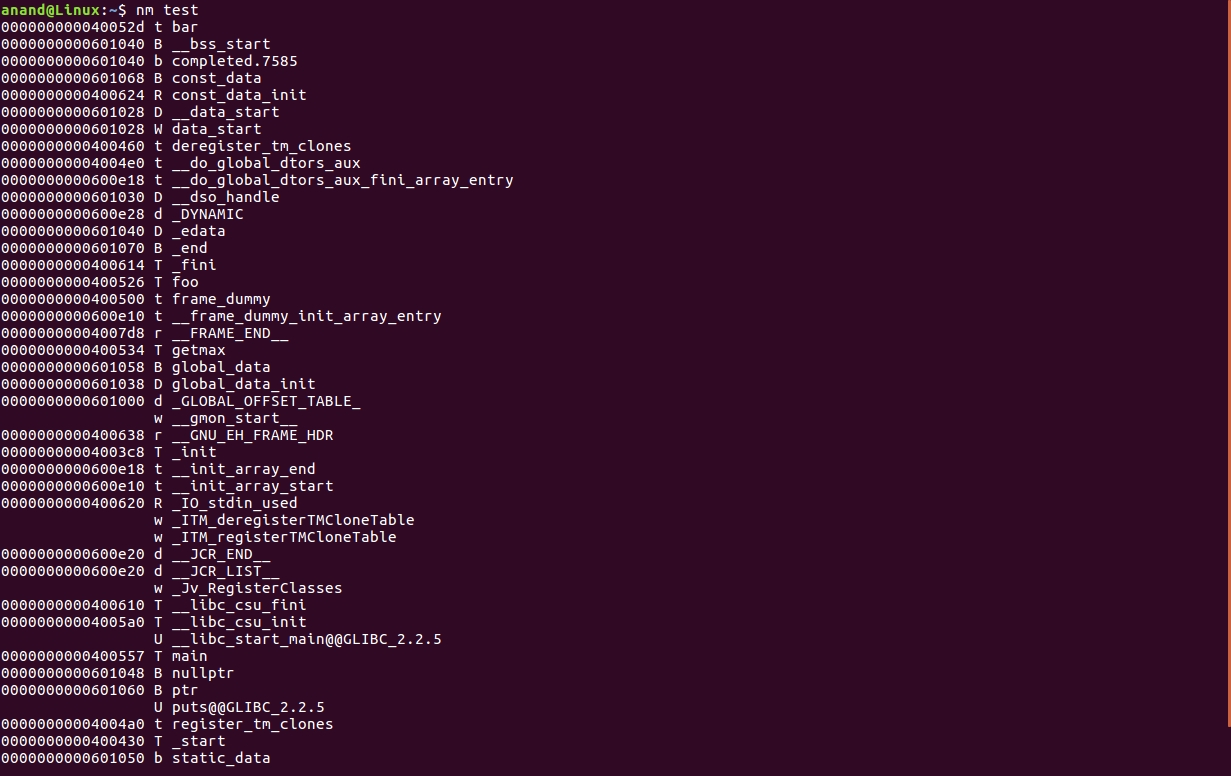
Simply type the program name to run the program:

*>> ./test*

**ELF files and analysis**

Generated object files, executables and libraries in Linux are known as ELF (executable and linkable format) files. To analyse ELF files you can use a few dissection tools listed below.

nm: This is used to print a symbol table from an object file or executable. All the global variable and function names seen by the linker are known as symbols.



readelf: This is used to provide meta information about ELF files (Linux only).



objdump: This is used for the dissection of ELF files in terms of disassembly, symbols, section details, etc.

To know the size of the text, data, bss sections and the total, we can use the size command, as follows:

*>> size test.o*



**Sections of a program/process**

An actively loaded program can have various sections like code (.text), initialised data (.data), uninitialised data (.bss), read-only data (.rodata) and stack. To see the applicable sections of symbols in a code, let’s add a few global variables and functions, as follows. Compile as object file and see the symbol table using nm or objdump. Then check the indicated symbol states:

The explanation of the above code is given below.  
T, t: .text  
D, d: .data  
R, r: .rodata  
C: Common symbols, will be merged into .bss on linking  
B: .bss  
W, w: Weak symbols  
T, R,D: Indicates eligible for external linkage  
t, d, r: Indicates restricted for internal linkage only

You may wonder about the invisibility of local variables, which are not considered as symbols. These are converted into offsets with respect to the stack frame using a stack pointer, frame pointer registers (ESP, EBP in x86) thus not seen by linker, which is clear from disassembly using objdump also.

Weak symbols are almost similar to declarations, but don’t cause a linker error if they are not defined by the user. These are useful for interrupt handlers, exception handlers or any event handlers that can be aliased to default handlers. Users can override these with their own custom handlers. Let’s add this code in test.c to check for weak symbols quickly and check nm output.

*void f1() {*

*#default code for f1*

*}*

*void f1() \_\_attribute((weak)); #f1 can be redefined*

*void f2\_default() {*

*#default code for f2*

*}*

*void f2() \_\_attribute\_\_((weak,alias(“f2\_default”)));*

*#f2 can be redefined*

f1 or f2can be redefined by the user only once as strong symbols in other translation units, or can be redefined as weak symbols multiple times subsequently. f2\_default will be invoked if f2 is not redefined which is aliased to f2. If a weak function is not aliased to a strong function, not having default code in the same translation unit or not redefined further by user causes runtime error.

**An example of a multi-file**

Now let’s try another example where the application is built from multiple source files; each source file with the included header files getting compiled individually is known as the translation unit. In this example, sum and square are invoked from the main function in test.c, and are defined in sum.c, sqr.c respectively. Assume the suitable function declarations and necessary header files.

*#test.c:-*

*int main()*

*{*

*int a,b,c,d;*

*a=10,b=20;*

*c=sum(a,b);*

*d=square(a);*

*printf(“c=%d,d=%d\n”,c,d);*

*return 0;*

*}*

*#sum.c:-*

*int sum(int x,int y)*

*{*

*int z;*

*z=x+y;*

*return z;*

*}*

*#sqr.c:-*

*int square(int x)*

*{*

*return x\*x;*

*}*

To build the above code, i.e., compile individual translation units and link generated object files, you can use the following sequence of commands:

*>>gcc test.c -c #-o test.o*

*>>gcc sum.c -c #-o sum.o*

*>>gcc sqr.c -c #-o sqr.o*

*>>gcc test.o sum.o sqr.o -o all.out*

**Static vs dynamic linking**

Applications may be linked statically or dynamically. In static linking, all necessary code is kept part of executable by the linker, which enables better performance, eliminating runtime overhead. But this poses the problem of a larger footprint of executables. In dynamic linking, library functions are excluded in the executable and loaded on demand, which allows an optimal footprint, but poses the problem of runtime overhead. Dynamic libraries come with the added advantage of sharing among applications and versioning support. A library is a collection of object files. Let’s create libraries for the frequently used functions like sum and square in the above code.

*>>ar rc libsample.a sum.o sqr.o*

*#static libraries come with extension .a, lib prefix is a convention*

*>>gcc -L. -lsample -o p.out*

*#libsample.a is linked statically and other std libraries like libc.so, libm.so are linked dynamically*

*>>gcc -L. -lsample -o s.out -static*

*# all libraries are linked statically, glibc-devel-static package is required in Linux for static linking of std libs like libc.a, libm.a*

*# Assume .exe extension instead of .out in windows,*

*# MingW suite doesn’t have much dynamic libs, most of the linking happens statically here.*

*>>gcc -shared sum.o sqr.o -o libsample.so*

*#On Linux shared object(.so) files support dynamic linking*

*>>gcc -shared sum.o sqr.o -o libsample.dll*

*#On Windows dynamic link libraries(dll) format is used*

*>>gcc -L. test.o -lsample -o d.out*

*#Link with libsample.\* dynamically*

While linking, the -L option is used by gcc to specify the custom path of our own libraries, and -l is used to specify the custom libraries, assuming the lib prefix and the .a or .so extension. In the above commands, a dot (.) is specified with –L, as the necessary libraries are in the current directory; so replace the dot (.) with the concerned path, as applicable. For example, -lc stands for libc.a or libc.so, -lpthread stands for libpthread.\* . We use -lsample to specify libsample.a or libsample.so.  
When both .a and .so are available in a specified directory, gcc opts for .so files by default. In Linux, the -static option is used to enforce static linking.

Compare the footprint of s.out, p.out and d.out using the size command or OS-specific commands like ls, du or dir. You will observe that s.out is heavy with all the library code and d.out is very light with minimal code, while p.out comes inbetween as only our library is statically linked. The strip utility can be applied on executables to reduce the footprint, which removes symbols from underlying object files. Please note that the strip can’t be applied on individual object files or libraries before linking. It is meaningful for executables only, especially statically linked code for constrained environments.

*>> strip s.out #compare the size of s.out before and after strip*

Alternatively, we need to update the LD\_LIBRARY\_PATH environment variable appended with ~/dlibs. It would be preferable if you could add an entry of the custom directory in /etc/ld.so.conf and run ldconfig once, to update the cache.  
To check the shared library dependencies of any executable, we can use ldd, as follows:

*>> ldd p.out d.out*

*#our library is listed in d.out but not in p.out*

*>> ldd s.out*

*#says no dependencies*

Please refer to tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

for more details on shared object files.  
A simple Linux utility file mentions the type of each file, particularly for binaries, providing helpful details like target architecture, whether it is statically or dynamically linked and if it is stripped or not.

*>> file d.out p.out s.out*

## **FAQs**

## How do I generate symbolic information for gdb and warning messages?

The syntax is as follows C compiler:  
*gcc -g -Wall input.c -o executable*   
The syntax is as follows C++ compiler:  
*g++ -g -Wall input.C -o executable*

## How do I generate optimized code on a Linux machine?

The syntax is as follows C compiler:  
*gcc -O input.c -o executable*   
The syntax is as follows C++ compiler:  
*g++ -O -Wall input.C -o executable*

## How do I compile a C program that uses math functions?

The syntax is as follows when need pass the -lm option with gcc to link with the math libraries:  
*gcc myth1.c -o executable -lm*

## How do I compile a C++ program that uses Xlib graphics functions?

The syntax is as follows when need pass the -lX11 option with gcc to link with the Xlib libraries:  
*g++ fireworks.c -o executable -lX11*

# GNU Binutils

The GNU Binutils are a collection of binary tools. The main ones are:

* **ld** - the GNU linker.
* **as** - the GNU assembler.

But they also include:

* **addr2line** - Converts addresses into filenames and line numbers.
* **ar** - A utility for creating, modifying and extracting from archives.
* **c++filt** - Filter to demangle encoded C++ symbols.
* **dlltool** - Creates files for building and using DLLs.
* **gold** - A new, faster, ELF only linker, still in beta test.
* **Gprof** - Displays profiling information.
* **Nlmconv** - Converts object code into an NLM.
* **Nm** - Lists symbols from object files.
* **Objcopy** - Copies and translates object files.
* **Objdump** - Displays information from object files.
* **Ranlib** - Generates an index to the contents of an archive.
* **Readelf** - Displays information from any ELF format object file.
* **Size** - Lists the section sizes of an object or archive file.
* **Strings** - Lists printable strings from files.
* **Strip** - Discards symbols.
* **Windmc** - A Windows compatible message compiler.
* **Windres** - A compiler for Windows resource files.

Most of these programs use **BFD**, the Binary File Descriptor library, to do low-level manipulation. Many of them also use the **opcodes** library to assemble and disassemble machine instructions.

The binutils have been ported to most major Unix variants as well as Wintel systems, and their main reason for existence is to give the [GNU system](http://www.gnu.org/gnu/gnu-history.html) (and [GNU/Linux](http://www.gnu.org/gnu/linux-and-gnu.html)) the facility to compile and link programs.

### Obtaining binutils

The latest release of GNU binutils is 2.28. The various NEWS files ([binutils](https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=blob_plain;f=binutils/NEWS;hb=refs/tags/binutils-2_28), [gas](https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=blob_plain;f=gas/NEWS;hb=refs/tags/binutils-2_28), and [ld](https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=blob_plain;f=ld/NEWS;hb=refs/tags/binutils-2_28)) have details of what has changed in this release.

If you plan to do active work on GNU binutils, you can access the development source tree by anonymous git:

git clone git://sourceware.org/git/binutils-gdb.git

Alternatively, you can use [the gitweb interface](http://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git), or the source snapshots, available as bzipped tar files via anonymous FTP from, <ftp://sourceware.org/pub/binutils/snapshots>.