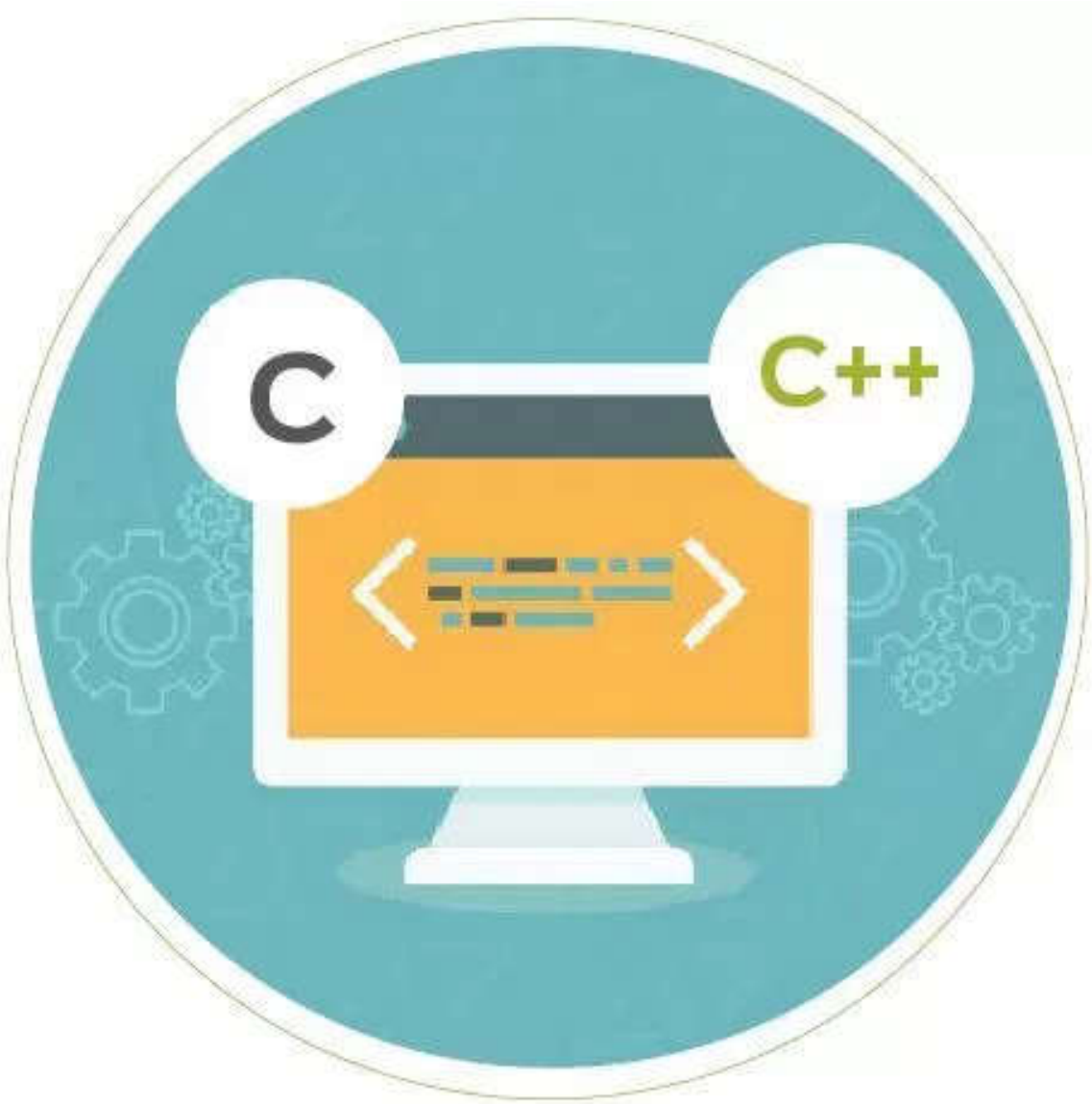
GNU Tools that Help You Develop C/C++ Applications

DEVELOPERS (HTTP://OPENSOURCEFORU.COM/CATEGORY/DEVELOPERS/)



This article explores how various GNU tools are used for the development of C/C++ applications. It also covers the anatomy of the generated intermediate files and the final outcome. These tools are also available for the Windows environment with support from the MingW runtime.

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.

For Windows, similar tools are provided by the MingW suite, originally available from *mingw.org*; but this is limited to 32-bit versions and there have been no consistent updates recently. You may need to download multiple packages and merge them for recent versions. An installer utility is preferred to download all necessary packages in

online mode. MingW generates code for the Windows runtime in the PE format, so UNIX/Linux-specific system calls can't be used with this.
You can get a TDM variant of *gcc*, which comes with the Code::Blocks IDE. Currently *gcc* v4.9.2 is bundled with Code::Blocks v16.01. With this you can also avail an elegant IDE with a *gcc* backend. Download *codeblocks-16.01mingw-nosetup.zip* from *codeblocks.org/downloads*, extract and then update the path to *codeblocks-16.01mingw-nosetup\MingW\bin* at the user level or system level. Now navigate to the directory holding the code in a command prompt, and run any command like *gcc*.

Alternatively, you can use the MingW-w64 Project, a fork of MingW, which supports 64-bit systems also and provides offline archives. You can download the archive from *sourceforge.net/projects/mingw-w64/files/*, then navigate to *Toolchains targetting Win64/Personal Builds/dongsheng-daily* and choose the desired version like 4.8, 4.9 or 5.x. Extract the archive and update the path to the extracted bin directory.

Optionally, the Cygwin Project also provides a complete UNIX-like environment in Windows with GNU tools and an abstraction layer for POSIX APIs. Any UNIX/Linux application can be ported on Cygwin.

A simple program and its analysis
To see various intermediate phases while building, let's look at the following simple example:

```
#simple.c
#include<stdio.h>
#define PI 22.0/7.0
int main()
{
double area,r; #a comment
area=PI*r*r;
printf("area of circle=%lf\n",area);
return 0;
}
```

To build the above program, we use the following command:

```
gcc simple.c -o simple
```

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

```
gcc -E simple.c #output comes on stdout by default
cpp simple.c -o simple.i
#you can use -o option to store output in a file, .i
extension is a convention to store outcome of preprocessor.
```

Optionally, we can provide symbolic constant PI externally using the *-D* option:

```
gcc -DPI=22.0/7.0 simple.c
```

To stop with generation of assembly code equivalent to source code the -S option is used, which internally uses *cc1,cc1plus* commands:

```
gcc -S simple.c #generates simple.s
```

To locate the *cc1* command, use the following command:

```
gcc --print-prog-name=cc1 #path could be libexec/gcc/mingw32/4.9.2/ in MingW
```

To generate assembly using *cc1*, use the command given below:

```
<path-of-cc1-dir>/cc1 simple.c
#cc1 is not located in bin dir of toolchain
$( gcc --print-prog-name) simple.c #Linux specific
```

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

```
as simple.s -o simple.o
```

To stop compilation from the source code, use the code given below:

```
gcc -c simple.c #generates object file simple.o
```

To combine one or more object files with the necessary library files, runtime support, and to generate executables using *collect2*, *ld* internally (e.g., printf is taken from the standard C library in the form of *libc.a* or *libc.so* and math functions are taken from libm.*), use the code given below:

```
gcc simple.o -o simple #generates executable/binary, a.out in absence of -o option
```

To retain all intermediate files while building with gcc, type:

```
gcc --save-temps simple.c #keeps simple.i,simple.s,simple.o
```

gcc supports various optimisation levels by using options like *-O1,-O2,-O3* and *-O0* to turn off optimisation, and *-OS* for space optimising. If you are planning to debug generated code using *gdb*, the *-g<level>(-g1,-g2,-g3)* option can be used.

-g2 is assumed if *-g* is specified and *-g0* turns off debug support. You can use the *-I* option to specify the custom path of additional header files. Compile-specific options like *–I* and *-D* can go into the *CFLAGS* variable, and linker-specific options like *-L*, *-l* and *-static* can go into the *LDFLAGS* variable.

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
#all.exe in case of windows
```

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

# 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
```

Note: *The following section is Linux specific.*

```
First, type:
gcc -shared -Wl,-soname,libsample.so -o ~/dlibs/libsample.so.1.0.1 sum.o sqr.o
ldconfig -n ~/dlibs
```

To run the dynamically linked executable, we need to specify the custom path of the libraries:

```
LD_LIBRARY_PATH=~/dlibs ./d.out
```

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
```

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. Even though MingW generates Windows PE format, the effect of these tools (except *readelf*) will be the same.

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.

Here, you can observe that sum and square are undefined in *test.o* and defined in *sum.o, sqr.o*. A lot of runtime support symbols are added in *all.out*, as follows:

```
nm test.o sum.o sqr.o all.out
```

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

```
objdump -d test.o #disassemble ELF files
objdump -t test.o #displays symbol info
objdump -x test.o #displays all headers
objdump -S test.o #intermixed source and assembly, code has to be compiled with debug support using -g option
```

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

```
readlef -h test.o all.out #Provides ELF header details like magic number, target #architecture,endianness, ABI, e
readelf -a all.out
#Provides all headers and section details
```

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

```
size test.o all.out
```

Last but not the least, the *file* utility mentioned earlier is used to know each type of *file*. Based on the magic number in the header part of file, it can distinguish between object files, executables, static libraries and dynamic libraries.

```
file *.o all.out libsample.a libsample.so
```

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:

```
int d1=10;                          //D
int c;                              //C
static int d2=10;                   //d
static int b1;                      //b
const int r1=10;                    //R
const int r2;                       //r
void foo() { }                      //T
static void bar() { }               //t
```

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 *f2* can 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.

Function overloading and name demangling

Let's look at a few ways in which the tools are used on C++ code, with examples of function overloading.

```
int sum(int x, int y);
float sum(float x, float y);
int sum(int x, int y,int z);
```

Write two simple C++ programs with definitions of the above functions in *fun.cpp*, the main function with a few calls in *test.cpp*, and then generate object files.

```
g++ -c fun.cpp
g++ -c test.cpp
```

Now check the symbols generated for function definitions and calls for the sum and swap functions using *nm*.

```
nm test.o fun.o
```

Here, you can see overloaded function calls. Definitions are decorated according to the number and type of parameters, and the enclosing scope like global, class, namespace, usage of templates, etc. But names are mangled as per internal conventions. To demangle the symbol names in readable format, you can use the option *-C* or *–demangle* with *nm* or *objdump*.

```
nm --demangle test.o fun.o
objdump -t -C test.o fun.o
```

A note about elfutils and cross toolchains

GNU tools target a wide variety of platforms. Ulrich Drepper wrote elfutils purely for Linux and the ELF format. You can download these from the package manager or build them from source available at *fedorahosted.org/releases/e/l/elfutils*. Similar tools are also available for various target platforms with suitable prefixes, e.g., the Linaro family of toolchains for building Linux components, and bare metal toolchains from *launchpad.net/gcc-arm-embedded* for ARM targets provide similar tools with their own prefixes. The options and usage of these tools will be similar to the steps mentioned here.

In this article, only minimal hints on tools have been provided. Please refer to the man pages of each command and other resources for additional inputs.

TAGS: C/C++ (HTTP://OPENSOURCEFORU.COM/TAG/CC/), GNU (HTTP://OPENSOURCEFORU.COM/TAG/GNU/), OPEN SOURCE (HTTP://OPENSOURCEFORU.COM/TAG/OPEN-SOURCE/), TOOLS (HTTP://OPENSOURCEFORU.COM/TAG/TOOLS/)

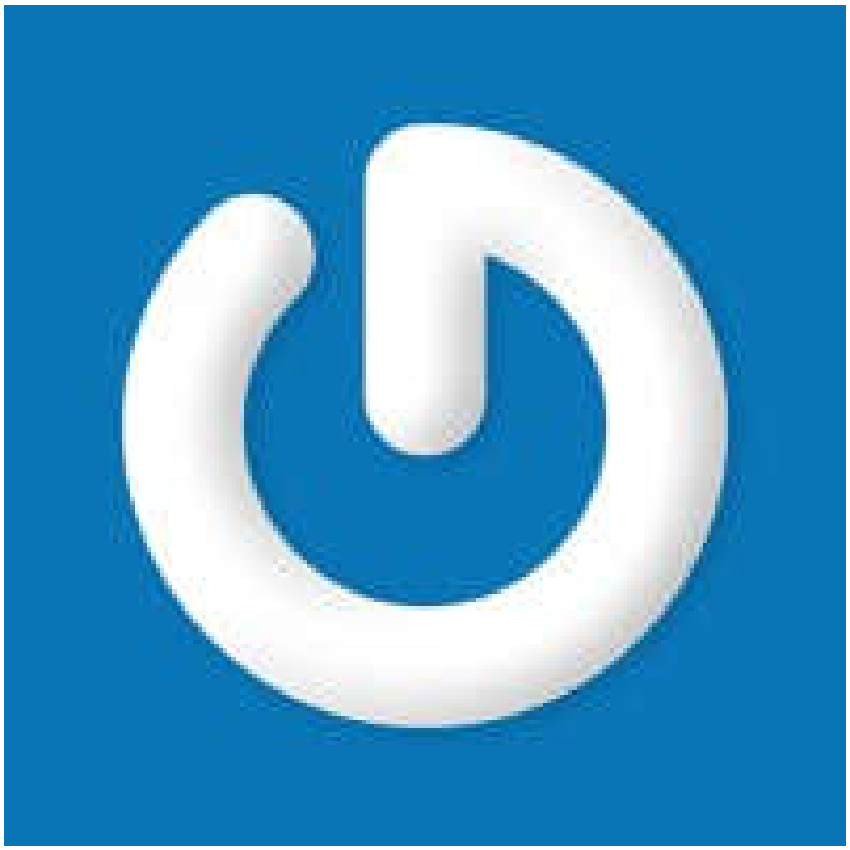
t
(https://www.tumblr.com/widgets/share/tool
shareSource=legacy&canonicalUrl=&url=http%
tools-help-develop-cc-
applications%2F&posttype=link&title=GNU+To

Next Article

Phase Two Delivered! Thanks to Apache XMLBeans
(http://opensourceforu.com/2016/12/phase-two-delivered-thanks-apache-xmlbeans/)

Previous Article

Qt Creator 4.2 brings new SCXML Editor module
(http://opensourceforu.com/2016/12/qt-creator-4-2-released/)



(http://opensourceforu.com/author/rajesh-sola/)

Author

Rajesh Sola (http://opensourceforu.com/author/rajesh-sola/)

The author is a faculty member at C-DAC's advanced computing training school, Pune and an evangelist in the embedded systems and IoT domains. He loves teaching and open source. You can reach him at rajeshsola@gmail.com

RELATED ARTICLES

DR KUMAR GAURAV (HTTP://OPENSOURCEFORU.COM/AUTHOR/DR-GAURAV-KUMAR/), APRIL 4, 2015

(http://opensourceforu.com/2015/04/deploying-infrastructure-as-a-service-using-openstack/)

Deploying Infrastructure-as-a-Service Using OpenStack (http://opensourceforu.com/2015/04/deploying-infrastructure-as-a-service-using-openstack/)

VAIBHAV KAUSHAL (HTTP://OPENSOURCEFORU.COM/AUTHOR/VAIBHAV-KAUSHAL/), DECEMBER 5, 2013

(http://opensourceforu.com/2013/12/build-web-page-qforms/)

Build Your Own Web Page With QForms (http://opensourceforu.com/2013/12/build-web-page-qforms/)

ANKIT MATHUR (HTTP://OPENSOURCEFORU.COM/AUTHOR/ANKIT-MATHUR/), MARCH 1, 2011

(http://opensourceforu.com/2011/03/mapreduce-more-power-less-code-hadoop/)

(Hadoop) MapReduce: More Power, Less Code (http://opensourceforu.com/2011/03/mapreduce-more-power-less-code-hadoop/)

ALSO ON OPEN SOURCE FOR YOU

Google highlights most popular open source projects through report card

1 comment • 2 months ago•

Kunle Adeniyi — What about a link to the google report?

NES Classic Edition uses Linux to enable retro games

1 comment • a month ago•

nuu — pleaaaase disable that annoying scrolling 'feature' !

Intel Joule rivals Raspberry Pi with Atom processors and IoT-centric Linux support

1 comment • 4 months ago•

Alhassan Abdulkadir — How is the $369 device a competitor to a $35 Raspberry Pi?

A quick look at multi-architecture, multi-platform mobile app development frameworks

1 comment • 3 months ago•

HM Sathish — Good info. Lucky me I reach on your website by accident, I bookmarked it.

CASE STUDIES

(http://opensourceforu.com/2016/10/government-leverages-open-source-build-digilocker-indian-citizens/)

Government leverages open source to build DigiLocker for Indian citizens (http://opensourceforu.com/2016/10/government-leverages-open-source-build-digilocker-indian-citizens/)

JAGMEET SINGH , OCTOBER 4, 2016

(http://opensourceforu.com/2016/09/open-source-helps-visually-impaired-thriveni-foundation-become-independent/)

Open source helps visually impaired at Thriveni Foundation become independent (http://opensourceforu.com/2016/09/open-source-helps-visually-impaired-thriveni-foundation-become-independent/)

JAGMEET SINGH , SEPTEMBER 1, 2016

(http://opensourceforu.com/2016/09/jugnoo-uses-open-source-dominate-auto-rickshaw-aggregator-space/)