

GCC (Gnu C Compiler)

In Linux, the GCC stands for **GNU Compiler Collection**. It is a compiler system for the various programming languages. It is mainly used to compile the C and C++ programs. It takes the name of the source program as a necessary argument; rest arguments are optional such as debugging, warning, object file, and linking libraries.

GCC is a core component of the GNU toolchain. Various open-source projects are compiled using the GCC, such as Linux kernel and GNU tools.

It is distributed under the GPL (General Public License). The first version, **GCC 1.0**, was released in 1987. It was only for the C programming language, but in the same year, it was extended for the [C++ programming language](#). Later, it was developed for other programming languages such as Objective-C, Objective-C++, Fortran, [Java](#), Ada, [Go](#), and more. Its latest version has the much-improved implementation of the programming languages.

It is the official partner of GNU OS; therefore, it has been adopted as the standard compiler of the Linux based systems.

Installation of GCC on Linux

By default, it comes with the most Linux distributions. We can verify it by executing the below command:

```
gcc --version
```

The above command will display the installed version of the GCC tool. If it is not installed, follow the below steps to install it:

Step1: Update the package list.

To update the package list, execute the following command:

```
sudo apt update
```

It will ask for the system administrative password, enter the password. It will start updating the system package. Consider the below snap of output:

```
javatpoint@javatpoint-Inspiron-3542:~$ sudo apt update
[sudo] password for javatpoint:
Get:1 http://packages.microsoft.com/repos/vscode stable InRelease [3,959 B]
Get:2 http://packages.microsoft.com/repos/vscode stable/main amd64 Packages [174 kB]
Hit:3 http://in.archive.ubuntu.com/ubuntu bionic InRelease
Hit:4 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu bionic InRelease
Get:5 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Get:6 http://in.archive.ubuntu.com/ubuntu bionic-updates InRelease [88.7 kB]
Hit:7 http://ppa.launchpad.net/nilarimogard/webupd8/ubuntu bionic InRelease
Ign:8 http://dl.google.com/linux/chrome/deb stable InRelease
Get:9 http://dl.google.com/linux/chrome/deb stable Release [943 B]
Get:10 http://dl.google.com/linux/chrome/deb stable Release.gpg [819 B]
Get:11 http://dl.google.com/linux/chrome/deb stable/main amd64 Packages [1,128 B]
Get:12 http://security.ubuntu.com/ubuntu bionic-security/main i386 Packages [459 kB]
Get:13 http://in.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Get:14 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [692 kB]
Get:15 http://security.ubuntu.com/ubuntu bionic-security/main amd64 DEP-11 Metadata [38
.7 kB]
Get:16 http://security.ubuntu.com/ubuntu bionic-security/main DEP-11 48x48 Icons [17.6
kB]
Get:17 http://security.ubuntu.com/ubuntu bionic-security/main DEP-11 64x64 Icons [46.0
kB]
Get:18 http://security.ubuntu.com/ubuntu bionic-security/universe i386 Packages [618 kB
]
Get:19 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 Packages [916 kB]
```

Step2: Install the build-essential package.

It contains various packages such as **gcc, g++, and make** utility. Execute the below command to install it:

```
sudo apt install build-essential
```

The above command will install the required packages for the GCC utility. Now, we can use the GCC utility in our machine. Consider the below snap of output:

```
javatpoint@javatpoint-Inspiron-3542:~$ sudo apt install build-essential
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  dpkg-dev fakeroot libalgorithm-diff-perl libalgorithm-diff-xs-perl
  libalgorithm-merge-perl libdpkg-perl libfakeroot
Suggested packages:
  debian-keyring git bzr
The following NEW packages will be installed:
  build-essential dpkg-dev fakeroot libalgorithm-diff-perl
  libalgorithm-diff-xs-perl libalgorithm-merge-perl libfakeroot
The following packages will be upgraded:
  libdpkg-perl
1 upgraded, 7 newly installed, 0 to remove and 318 not upgraded.
Need to get 982 kB of archives.
After this operation, 2,592 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 libdpkg-perl all 1.19.0.5ubuntu2.3 [211 kB]
Get:2 http://in.archive.ubuntu.com/ubuntu bionic-updates/main amd64 dpkg-dev all 1.19.0.5ubuntu2.3 [607 kB]
Get:3 http://in.archive.ubuntu.com/ubuntu bionic/main amd64 build-essential amd64 12.4ubuntu1 [4,758 B]
Get:4 http://in.archive.ubuntu.com/ubuntu bionic/main amd64 libfakeroot amd64 1.22-2ubuntu1 [25.9 kB]
```

Step3: Verify the installation.

To verify the installation, execute the `gcc -version` command as follows:

```
gcc --version
```

It will display the installed version of GCC utility. To display the more specific details about the version, use the `'-v'` option. Consider the below output:


```

javatpoint@javatpoint-Inspiron-3542:~$ gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

javatpoint@javatpoint-Inspiron-3542:~$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 7.5.0-3ubuntu1~18.04' --
with-bugurl=file:///usr/share/doc/gcc-7/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++ --prefix=/usr --with-gcc-major-version-only --program-suffix=-7 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)

```

Here, we have successfully installed the GCC utility. Let's understand to use it. We will create and execute some c programs using GCC.

Run first C program by gcc

Create a basic **c program** "Hello world!". Create a file 'hello.c' and put below code in it:

```

#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}

```

Now, compile the hello.c as follows:

```
gcc hello.c
```

If we directly run the hello.c, it will throw the error. Make it executable, the default executable file for the Linux system is a.out. To execute the file, execute the chmod command as follows:

```
chmod a+x a.out
```

Now, run the c program as:

```
./a.out
```

Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~$ gcc hello.c
javatpoint@javatpoint-Inspiron-3542:~$ chmod a+x a.out
javatpoint@javatpoint-Inspiron-3542:~$ ./a.out
Hello, world!
```

GCC command Examples

Some useful examples of gcc command are as following:

- **Specify the object file name**

By default, the gcc command creates the object file as '**a.out**.' If you want to change the default output file name, use the '**-o**' option.

Let's execute the basic gcc command:

```
gcc hello.c
```

The above command will generate the object file 'a.out.' To specify the object file name, execute the command as follows:

```
gcc hello.c -o hello
```

It will generate the output file 'hello.' Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~/project$ gcc hello.c
javatpoint@javatpoint-Inspiron-3542:~/project$ ls
a.out  hello.c  Makefile~
javatpoint@javatpoint-Inspiron-3542:~/project$ gcc hello.c -o hello
javatpoint@javatpoint-Inspiron-3542:~/project$ ls
a.out  hello  hello.c  Makefile~
javatpoint@javatpoint-Inspiron-3542:~/project$
```

- **Enable all Warnings**

To enable all warnings in the output, use the '**-Wall**' option with the gcc command. Let's create a variable in the main function of hello.c. Consider the below code:

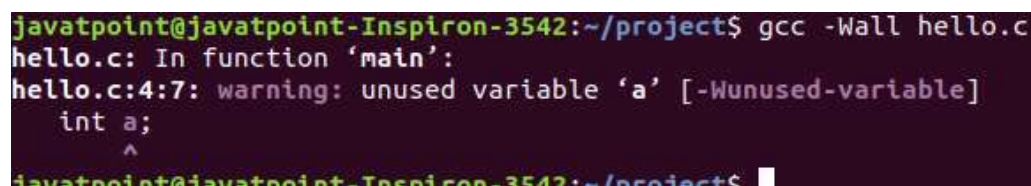
hello.c:

```
#include <stdio.h>
int main() {
int a;
    printf("Hello, world!\n");
    return 0;
}
```

If we compile the above code using -Wall option. It will throw the warnings. Execute the below command to compile the file:

```
gcc -Wall hello.c
```

The above command will display the warnings. Consider the below output:



```
javatpoint@javatpoint-Inspiron-3542:~/project$ gcc -Wall hello.c
hello.c: In function 'main':
hello.c:4:7: warning: unused variable 'a' [-Wunused-variable]
    int a;
    ^
javatpoint@javatpoint-Inspiron-3542:~/project$
```

Linux make command

The Linux **make** command is used to build and maintain groups of programs and files from the source code. In Linux, it is one of the most frequently used commands by the developers. It assists developers to install and compile many utilities from the terminal. Further, it handles the compilation process of the sizeable projects. It saves the compilation time.

The main motive of the make command is to determine a large program into parts and to check whether it needs to be recompiled or not. Also, it issues the necessary orders to recompile them.

In this section, we will use C++ programs since the C++ programming language is an object-oriented language, but you can use any language installed on your machine. It is not just limited to programs; we can use it to describe other tasks also.

How make command works?

The make command takes targets as arguments. These arguments are specified in 'Makefile.' The makefile contains the targets as well as associated actions related to these targets.

When we execute the make command, it searches for the makefile and scans it to find the target and access its dependencies. If dependencies are not specified, it will search for the dependency and will build it. It will build the main target after the dependencies are built.

For example, if we want to change only one source file and we execute the make command; so, this will compile only the object file that is connected with that source file. It will save a lot of time in the final compilation of the project.

Use of the make command

Let's create a C++ project having files main.cpp, function1.cpp, function2.cpp and a dependency file function.h.

The code of the files is as following:

main.cpp:

```
#include <iostream>
#include "functions.h"
int main()
{
    print_hello();
    std::cout<< std::endl;
    std::cout<< "The factorial of 5 is " << factorial(5) << std:: endl;
    return 0;
}
```

function1.cpp:

```
#include "functions.h"

int factorial(int n)
{
    if(n!=1)
    {
        return (n * factorial(n-1));
    }
    else return 1;
}
```

function2.cpp:

```
#include <iostream>
#include "functions.h"
void print_hello()
{
    std::cout << "Hello World";
}
```

functions.h:

```
void print_hello();
int factorial (int n);
```

Now create an executable file of the above project by executing the below command:

```
g++ main.cpp function1.cpp function2.cpp -o hello
```

The above command will create an executable file **'hello'** of the files main.cpp, function1.cpp and function2.cpp.

Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~/sample$ g++ main.cpp function1.cpp function2.cpp -o hello
javatpoint@javatpoint-Inspiron-3542:~/sample$
```

From the above output, if it is successfully executed, it will not give any output.

Let's perform the same task by using the makefile.

Create a file as **Makefile** and put the below code in it.

```
all:
    g++ main.cpp function1.cpp function2.cpp -o hello
```

The all keyword is used for target and in newline put the same command with a TAB as above to specify the operation. Save the file. Consider the below file:

```
all:
    g++ main.cpp function1.cpp function2.cpp -o hello
```

To operate, execute the command as follows:

```
make
```

The above command will create an executable file 'hello' of the specified files. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~/sample$ make
g++ main.cpp function1.cpp function2.cpp -o hello
```

Let's add some more tasks to Makefile. Add a task '**compile**' as follows:

```
all:

compile:
    g++ main.cpp function1.cpp function2.cpp -o hello
```

To execute the task **compile**, execute the below command:

```
make compile
```

The above command will execute the compile task. Consider the below output:

```
javatpoint@javatpoint-Inspiron-3542:~/sample$ make compile
g++ main.cpp function1.cpp function2.cpp -o hello
```

Linux System Calls with C

Linux currently provides about 200 different system calls. A listing of system calls for your version of the Linux kernel is in `/usr/include/unistd.h`. Some of these are for internal use by the system, and others are used only in implementing specialized library functions. In this sample chapter, we present a selection of system calls that are likely to be the most useful to application and system programmers.

access: Testing File Permissions

The `access` system call determines whether the calling process has Access permission to a file. It can check any combination of read, write, and execute permission, and it can also check for a file's existence. The `access` call takes two arguments. The first is the path to the file to check. The second is a bitwise or of `R_OK`, `W_OK`, and `X_OK`, corresponding to read, write, and execute permission. The return value is 0 if the process has all the specified permissions. If the file exists but the calling process does not have the specified permissions, `access` returns `-1` and sets `errno` to `EACCES` (or `EROFS`, if write permission was requested for a file on a read-only file system).

If the second argument is `F_OK`, `access` simply checks for the file's existence. If the file exists, the return value is 0; if not, the return value is `-1` and `errno` is set to `ENOENT`. Note that `errno` may instead be set to `EACCES` if a directory in the file path is inaccessible.

The program shown in Listing 8.1 uses `access` to check for a file's existence and to determine read and write permissions. Specify the name of the file to check on the command line.

Example (check-access.c) Check File Access Permissions

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    char* path = argv[1];
    int rval;

    /* Check file existence. */
    rval = access (path, F_OK);
    if (rval == 0)
        printf ("%s exists\n", path);
    else {
        if (errno == ENOENT)
```

```

    printf ("%s does not exist\n", path);
else if (errno == EACCES)
    printf ("%s is not accessible\n", path);
return 0;
}

/* Check read access. */
rval = access (path, R_OK);
if (rval == 0)
    printf ("%s is readable\n", path);
else
    printf ("%s is not readable (access denied)\n", path);

/* Check write access. */
rval = access (path, W_OK);
if (rval == 0)
    printf ("%s is writable\n", path);
else if (errno == EACCES)
    printf ("%s is not writable (access denied)\n", path);
else if (errno == EROFS)
    printf ("%s is not writable (read-only filesystem)\n", path);
return 0;
}

```

For example, to check access permissions for a file named README on a CD-ROM, invoke it like this:

```

% ./check-access /mnt/cdrom/README
/mnt/cdrom/README exists
/mnt/cdrom/README is readable
/mnt/cdrom/README is not writable (read-only filesystem)

```