# How to Compile and Run a C/C++ Program in Linux

## Step 1: Install GNU C/C++ compiler and related tools

* If you are using Debian or Ubuntu:

$ sudo apt-get update

$ sudo apt-get install build-essential manpages-dev

**Note**: The [build-essential](http://packages.ubuntu.com/xenial/build-essential) package includes:

* gcc compiler (GNU C Compiler)
* g++ compiler (GNU C++ Compiler)
* make tool
* Other related tools for C/C++ development in Linux

If you are using Fedora, Red Hat, CentOS, or Scientific:

$ sudo yum groupinstall 'Development Tools'

## Step 2: Verify installation

Type the following commands to display the location and version number of the compiler:

$ whereis gcc

$ gcc --version

Sample outputs:



## Step 3: Write a "Hello, World!" program

* If C program: Create a text file called helloworld.c using a text editor such as vi, nano, or gedit:

#include<stdio.h>

int main(void) {

printf("Hello, World!");

return 0;

}

* If C++ program: Create a text file called helloworld.cpp with following content:

#include "iostream"

int main(void) {

std::cout << "Hello, World!";

return 0;

}

## Step 4: Compile the code

* **If C program**: Go to the directory of your helloworld.c and then use any of these:

gcc helloworld.c -o helloworld

Or:

cc helloworld.c -o helloworld

Or:

## assuming that helloworld.c exists in the current directory ##

make helloworld

* **If C++ program**: Go to the directory of your helloworld.cpp and then use any of these:

g++ helloworld.cpp -o helloworld

Or:

## assuming that helloworld.cpp exists in the current directory ##

make helloworld

In both languages, if there is no error in your code, the compiler will successfully create an executable file called helloworld in the current directory, otherwise you need fix the code. To verify this, type:

$ ls -l helloworld\*

## Step 5: Run the executable file

Simply type the the program name:

$ ./helloworld

or

$ /path/to/helloworld

## Q&As

### How to compile a program with multiple source files?

The syntax is as follows if the source code is in several files (such as light.c, sky.c, fireworks.c):

gcc light.c sky.c fireworks.c -o executable-file

**Note**: For C++, apply the same rule as above.

**Tip**: The best way to build C/C++ program in Linux is using a [makefile](#_gjdgxs).

# Makefile

## What Is Makefile? Why Is It Important?

Typing compiling commands every time you need to compile source code files can be tiring, especially when your project consists of many source files. Makefiles are the solution to simplify and automate this task.

A makefile is basically **a script which defines and controls the whole building process automatically**. It works based on a strict set of rules defined and implemented by the [***GNU make***](http://www.gnu.org/software/make/manual/make.html) program.

For example, let’s assume we have the following source files in the same directory:

1. main.c

#include <stdio.h>

#include "functions.h"

int main() {

   print\_hello();

   printf("The factorial of 5 is %d.\n"), factorial(5);

   return 0;

}

2. hello.c

#include <stdio.h>

#include "functions.h"

void print\_hello() {

   prinf("Hello, World!\n");

}

3. factorial.c

#include "functions.h"

int factorial(int n) {

   if (n != 1) {

      return (n \* factorial(n-1));

   } else {

      return 1;

   }

}

4. functions.h

void print\_hello();

int factorial(int n);

The trivial way to compile these files and obtain an executable is by running the command:

$ gcc main.c hello.c factorial.c –I. -o hello

Or:

$ cc main.c hello.c factorial.c –I. -o hello

Or you can reverse the order of output and input files as follows:

$ cc -o hello main.c hello.c factorial.c –I.

For a large project where we might have hundreds of source files, it becomes extremely difficult and time-wasting to maintain the building process. Moreover, you might notice that you usually only work on a small section of the program (e.g. several functions in some files) and much of the remaining program is unchanged; re-compiling all source files is definitively unnecessary.

That's when a makefile comes to play! The following sections describe how to write makefiles steps by steps.

## Makefile Rules

### File Name

A makefile should be named 'Makefile' or 'makefile'.

### Target-Dependency Rule

A target entry form looks like this:

target: dependency-list

    command

    command

    command

Where:

* The target is the output file. When you run 'make', the target serves as an argument ($ make <target>).
* The dependency-list is the list of input files. Multiple dependencies are separated by spaces.
* The commands are a series of steps used to make the target. Each command has to start with a tab character, NOT spaces.

### Flow

**The first target is referred as the default target which is called first when we run make**. The make program then looks at the default target's list of dependencies; if any of them are older, it will run the targets for those dependencies before running itself.

**Example 1**:

some\_file: other\_file

    echo "This will run second, because it depends on other\_file."

other\_file:

    echo "This will run first."

Output:

$ make

This will run first.

This will run second, because it depends on other\_file.

**Example 2**:

blah: blah.o

    cc blah.o -o blah

blah.o: blah.c

    cc -c blah.c -o blah.o

blah.c:

    echo "int main() { return 0; }" > blah.c

clean:

    rm -f blah.o blah.c blah

Output:

$ make

echo "int main() { return 0; }" > blah.c

cc -c blah.c -o blah.o

cc blah.o -o blah

$ make clean

rm -f blah.o blah.c blah

### When Re-compiling?

The *GNU make* program is intelligent and works based on the changes you do in your source files. From the very first example, we know that all main.c, hello.c and factorial.c are dependent on functions.h, and main.c is dependent on both hello.c and factorial.c. If you make any changes in functions.h, then the *make* program will recompile all the source files to generate new object files. However if you make a change in main.c, as this is not dependent of any other file, then only main.c file will be recompiled.

**While compiling a file, the *make* program checks its object file and compares the time stamps. If the source file has a newer time stamp than the object file, it will generate a new object file** assuming that the source file has been changed.

## Makefile Macros

The *make* program allows to use macros, which are similar to variables and can only be strings.

### User-Defined Macros

files = file1 file2

some\_file: $(files)

    echo "Look at this variable: " $(files)

    touch some\_file

file1:

    touch file1

file2:

    touch file2

### Pre-Defined Macros

#### Conventional Macros

There are various pre-defined macros used in implicit rules. You can see them by typing "make -p" to print out the defaults. They fall into two classes:

1. Macros that are names of programs:

* **CC**: Program to compiling C programs. Default is 'cc'.
* **CXX**: Program to compiling C++ programs. Default is 'g++'.
* **CPP**: Program to running the C preprocessor. Default is '$(CC) -E'.
* **LINT**: Program to use to run lint on source code. Default is 'lint'.
* **RM**: Command to remove a file. Default is 'rm -f'.

2. Macros that contain arguments of the programs:

* **ASFLAGS**: Flags to give to the assembler when explicitly invoked on a '.s' or '.S' file.
* **CFLAGS**: Flags to give to the C compiler.
* **CXXFLAGS**: Flags to give to the C++ compiler.
* **CPPFLAGS**: Flags to give to the C preprocessor and programs, which use it (such as C and Fortran compilers).
* **LDFLAGS**: Flags to give to compilers when they are supposed to invoke the linker, 'ld'.
* **LINTFLAGS**: Flags to give to lint.

**Notes**:

* You can cancel all variables used by implicit rules with '-R' or '--no-builtin-variables' option.
* You can define macros at the command line as: $ make CPP = /home/courses/spring02

#### Special Macros

There are some special macros predefined, such as:

* **$@** is the name of the output file being generated (left hand side of :).
* **$?** is the list of names of changed dependencies (
* **$^** is the dependencies list (right hand side of :).
* **$<** is the first item in the dependencies list (first item at the right hand side of :).
* **$\*** is the prefix shared by target and dependencies.

Example 1:

# $@ represents hello and $? picks up all the changed source files.

SRC = main.c factorial.c hello.c

CFLAG = -Wall -g

CC = gcc

hello: ${SRC}

    ${CC} ${CFLAGS} $? $(LDFLAGS) -o $@

Example 2:

# Make of .o files out of .c files

# Way 1:

%.o: %.c:

$(CC) $(CFLAGS) -c $^ -o $@

# Way2:

# Note: This way is old (not recommended). Should use the first way instead.

.c.o:

$(CC) $(CFLAGS) -c $^ -o $@

Alternatively:

.c.o:

$(CC) $(CFLAGS) -c $\*.c

## Examples

The following makefile example is generic, so you can apply it (and customize it) to many projects:

# Run 'make' or 'make all' to build executable file

# Run 'make clean' to remove all object files and executable files

# Run 'make depend' to use makedepend to automatically generate dependencies

# (dependencies are added to end of Makefile)

# Define the compiler to use ('gcc' if C, or 'g++' if C++)

CC = gcc

# Define compiler flags

CFLAGS = -Wall -g

# Define directories containing header files other than /usr/include

INCLUDES = -I/home/triho/include  -I../include

# Define directories containing libraries other than /usr/lib

LFLAGS = -L/home/triho/lib  -L../lib

# Define names of libraries to link into executable:

#   The rule is appending '-l' before the lib name

#   For example, to link in libraries mylib.so or mylib.a, use -lmylib

LIBS = -lmylib1 -lmylib2

# Define the C source files

# Note: Change this list frequently

SRCS =  emitter.c \

        error.c \

        init.c \

        main.c \

        parser.c

# Define the C object files

#   This uses Suffix Replacement within a macro: $(name:oldstr=newstr)

#   For each word in 'name', replace 'oldstr' with 'newstr'

# Below we're replacing the suffix .c of all words in SRCS with the .o suffix

OBJS = $(SRCS:.c=.o)

# Define the executable file

EXECUTABLE = mycc

# Running 'make' will invoke the first target entry in the file

# You can name this target entry anything,

# but "default" or "all" are the most commonly used names by convention

all: $(EXECUTABLE)

    echo "All source files have been compiled"

# To create the executable file, we need the object files

$(EXECUTABLE): $(OBJS)

    $(CC) $(CFLAGS) $(INCLUDES) $(OBJS) $(LFLAGS) $(LIBS) -o $(EXECUTABLE)

# To create the object file, we need the source files (build .o files from .c files)

# Below we're using pre-defined macros:

#    '$^' is the list of names of the dependencies (.c files)

#    '$@' is the name of the target (.o file)

%.o: %.c:

    $(CC) $(CFLAGS) $(INCLUDES) -c $^ -o $@

# Running 'make clean' removes the executable file, all .o files and \*~ backup files

clean:

    $(RM) \*.o \*~ $(EXECUTABLE)

# Running 'make depend' generates dependencies of C source files automatically

depend: $(SRCS)

    makedepend $(INCLUDES) $^

# DO NOT DELETE THIS LINE -- make depend needs it

## .PHONY

Adding .PHONY to a target prevents *make* from confusing the phony target with a file name.

In the below example, if there is a source code file named 'clean.c', then make clean will still run correctly.

.PHONY: clean

clean:

    $(RM) \*.o \*~ $(EXECUTABLE)

## makedepend

<https://en.wikipedia.org/wiki/Makedepend>

<https://linux.die.net/man/1/makedepend>

## CMake

Also, [CMake](http://www.cmake.org/), is a cross platform build system. Among other things, cmake will generate makefiles for you. It is particularly useful for large projects, for builds that use lots of libraries, and for dealing with platform-specific compilation issues. It automatically generates (often hard to read and debug) makefiles for different platforms. For small projects, writing makefiles by hand is likely easier. See [Andy Danner's Cmake documentation](http://www.cs.swarthmore.edu/~adanner/tips/cmake.php) for more information on cmake.

## OTHERS

More features

<https://www.tutorialspoint.com/makefile/makefile_features.htm>

compile all c files at once

<https://stackoverflow.com/questions/170467/makefiles-compile-all-c-files-at-once>

You can include other Makefiles by using the include directive.

You can create conditional syntax in Makefiles, using ifdef, ifeq, ifndef, ifneq.

Debug mode vs release mode

<https://randu.org/tutorials/c/make.php>

GNU autoconf is a tool for automatically generating configure files from a configure.in file. These configure files can automatically setup Makefiles in conjunction with GNU automake and GNU m4. These tools are way beyond the scope of this document. Look in [GNU's manual repository](http://www.gnu.org/manual/) for more information on these tools.

CVS is another tool that may be useful for very large projects. CVS stands for Concurrent Versions System and it allows you to record the history of your source files. CVS stores the base source and then stores the differences for each version. CVS also allows for protecting code pieces of a multi-developer effort from accidental overwriting... in other words, code-insulation. More information on CVS can be found [here](http://www.gnu.org/manual/cvs-1.9/).

# GCC

<https://www.rapidtables.com/code/linux/gcc.html>

# GDB

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjHGWPN_S9NS_Ky1-tC8ZrUI>