

# Make and CMake Tutorial

## 1. Introduction

This tutorial intends to help you understand how to use make for building an application with multiple source files and header files. Using make and its configuration file, i.e., Makefile, programmers can easily automate the build process.

## 2. Source Files

```
#include <stdio.h>
#include <arithmetic.h>

int main(int argc, char *argv[])
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    printf("%d + %d = %d\n", a, b, add(a, b));
    printf("%d - %d = %d\n", a, b, sub(a, b));
    printf("%d * %d = %d\n", a, b, mul(a, b));
    printf("%d / %d = %d\n", a, b, dur(a, b));

    return 0;
}
```

cal\_main.c

```
#ifndef __ARITHMETIC_H__
#define __ARITHMETIC_H__

int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int dur(int a, int b);

#endif
```

arithmetic.h

```
#include <arithmetic.h>

int add(int a, int b)
{
    return a+b;
}
```

add.c

```
#include <arithmetic.h>

int sub(int a, int b)
{
    return a-b;
}
```

sub.c

```
#include <arithmetic.h>

int mul(int a, int b)
{
    return a*b;
}
```

mul.c

```
#include <arithmetic.h>

int dur(int a, int b)
{
    return a/b;
}
```

dur.c

## 3. Build the application manually

You can build the application by manually compiling each source file and linking them into a executable file. However, imagine there are hundreds of source files. As the program size grows up, manual building process certainly won't work.

```
$ gcc -I. -c cal_main.c
$ gcc -I. -c add.c
$ gcc -I. -c sub.c
$ gcc -I. -c mul.c
$ gcc -I. -c dur.c
$ gcc -o prog cal_main.o add.o sub.o mul.o dur.o
```

Then, how about making a simple program that repeatedly calls gcc as above. That program might seemingly automate the build process. Try to make a simple script file as below.

```
#!/bin/sh
gcc -I. -c cal_main.c
gcc -I. -c add.c
gcc -I. -c sub.c
gcc -I. -c mul.c
gcc -I. -c dur.c
gcc -o prog cal_main.o add.o sub.o mul.o dur.o
```

**build.sh**

Then, you can simply execute build.sh, which sequentially executes the given commands.

```
$ chmod a+x build.sh
$ ./build.sh
gcc -I. -c cal_main.c
gcc -I. -c add.c
gcc -I. -c sub.c
gcc -I. -c mul.c
gcc -I. -c dur.c
gcc -o prog cal_main.o add.o sub.o mul.o dur.o
```

※. Note that to execute the script file, you should give it an executable permission first. Just use the chmod command. The above chmod command sequence will give an executable permission to all users.

Do you think this is the best way of building a large software project? If we have hundreds of source files, this building process can take hours. After testing it, you might change some source files. Then, it again takes hours to build and link the whole source files. How can we both automate and shorten the build process?

#### 4. Basic Makefile

Makefile is a simple configuration file that describes the build dependencies among files. Just take a look at the following Makefile:

```

prog: cal_main.o add.o sub.o mul.o dur.o
    gcc -o prog cal_main.o add.o sub.o mul.o dur.o

cal_main.o : cal_main.c arithmetic.h
    gcc -I. -c cal_main.c

add.o : add.c arithmetic.h
    gcc -I. -c add.c

sub.o : sub.c arithmetic.h
    gcc -I. -c sub.c

mul.o : mul.c arithmetic.h
    gcc -I. -c mul.c

dur.o : dur.c arithmetic.h
    gcc -I. -c dur.c

clean:
    rm -rf prog cal_main.o add.o sub.o mul.o dur.o

```

### Makefile (1<sup>st</sup> version)

After making this Makefile as a simple text file in the same directory containing the source files, then you can run make, which in turn reads the Makefile in the current directory and tries to build the program as indicated in the Makefile.

```

$ make
gcc -I. -c cal_main.c
gcc -I. -c add.c
gcc -I. -c sub.c
gcc -I. -c mul.c
gcc -I. -c dur.c
gcc -o prog cal_main.o add.o sub.o mul.o dur.o

```

The nice thing about make is that it tracks which files have been changed since the last make. If you execute make after changing several source files, it will automatically find the necessary files to be compiled and only compile them. Thus the build time can be minimized, which has a huge impact on the productivity of software developers

**Modify add.c and sub.c**

```

$ vi add.c
$ vi sub.c
$ make
gcc -I. -c add.c
gcc -I. -c sub.c
gcc -o prog cal_main.o add.o sub.o mul.o dur.o

```

**Only the modified files are compiled**

Makefile is basically a sequence of the following rule construct, which describes the dependency between files and the command that will produce the target file from the source files. Note that there should be a tab right before the command. If not, make will not recognize the command properly. If you are using vi, try :set noexpandtab to make the tab key produce a real tab instead of generating a sequence of four or eight spaces.

```
target: source
```

```
[tab] command
```

When you start the make program, it first finds the first target in the Makefile, which is prog in our Makefile. If every necessary source file exists, then the command is executed to actually produce the target file. If any of the source files are missing, it investigates other target files to find how to make the missing files. This dependency resolving process eventually automates the build process.

Looking at the example Makefile, you can find a special target “clean” at the bottom. This is a special target with no source, which is specifically designed to clean the intermediate files produced during the build process. Try make clean to initiate the clean process, which will remove all the object files and executable files in the directory. This is useful when you want to make a clean start.

```
$ make clean
rm -rf prog cal_main.o add.o sub.o mul.o dur.o
```

You can use variables to eliminate any redundant information and enhance maintainability. In the second version of Makefile, we use PROG and OBJS variables. Assigning a value to a variable is similar to C syntax. However, when referencing a variable, you should use the form \$(variable\_name) to reference it. \$(CC) is a built-in variable for the compiler's name.

```
PROG = prog
OBJS = cal_main.o add.o sub.o mul.o dur.o

$(PROG): $(OBJS)
    $(CC) -o $(PROG) $(OBJS)

cal_main.o : cal_main.c arithmetic.h
    $(CC) -I. -c cal_main.c

add.o : add.c arithmetic.h
    $(CC) -I. -c add.c

sub.o : sub.c arithmetic.h
    $(CC) -I. -c sub.c

mul.o : mul.c arithmetic.h
    $(CC) -I. -c mul.c

dur.o : dur.c arithmetic.h
    $(CC) -I. -c dur.c

clean:
    rm -rf $(PROG) $(OBJS)
```

#### Makefile (2<sup>nd</sup> version)

Additionally, since there is a fixed rule from c file to o file, modern make allows to remove this implicit command. The implicit rule for compiling a c file is \$(CC) \$(CPPFLAGS) \$(CFLAGS) -c. You can put the CFLAGS variable for additional compiler options.

```


PROG = prog
OBJS = cal_main.o add.o sub.o mul.o dur.o
CFLAGS = -I. -Wall

$(PROG): $(OBJS)
    $(CC) -o $(PROG) $(OBJS)

cal_main.o : cal_main.c arithmetic.h
add.o : add.c arithmetic.h
sub.o : sub.c arithmetic.h
mul.o : mul.c arithmetic.h
dur.o : dur.c arithmetic.h

clean:
    rm -rf $(PROG) $(OBJS)

```

 -Wall option produces every possible warning. You had better always use -Wall option when compiling.

## 5. CMake

CMake is a cross-platform build system generator that is so-called a Makefile maker. Instead of writing down all the dependency information in Makefile, CMake automatically finds the dependencies and generates a Makefile for you. All you need to do is to make a CMakeLists.txt file with the minimum information in it. In the following CMakeLists.txt, only the target executable name “prog” and its source files “cal\_main.c add.c sub.c mul.c dur.c” are specified as well as the include path, which is, the current directory.

```

add_executable(prog cal_main.c add.c sub.c mul.c dur.c)
include_directories(.)

```

CMakeLists.txt (1<sup>st</sup> version)

Then running cmake generates the following files and directories in the working directory:

- CMakeCache.txt
- CMakeFiles/
- Makefile
- cmake\_install.cmake

In the automatically generated files and directories, we have a Makefile and all the dependency information which is extracted from the source files.

```

$ cmake .
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMake Warning (dev) in CMakeLists.txt:
  No cmake_minimum_required command is present.  A line of code such as

      cmake_minimum_required(VERSION 3.10)

  should be added at the top of the file.  The version specified may be lower
  if you wish to support older CMake versions for this project.  For more
  information run "cmake --help-policy CMP0000".
This warning is for project developers.  Use -Wno-dev to suppress it.

-- Configuring done
-- Generating done
-- Build files have been written to: /working

```

Then, simply running make build the executable file using the Makefile.

```

$ make
[ 16%] Building C object CMakeFiles/prog.dir/cal_main.o
[ 33%] Building C object CMakeFiles/prog.dir/add.o
[ 50%] Building C object CMakeFiles/prog.dir/sub.o
[ 66%] Building C object CMakeFiles/prog.dir/mul.o
[ 83%] Building C object CMakeFiles/prog.dir/dur.o
[100%] Linking C executable prog
[100%] Built target prog

```

If you want to build a static library instead of making an executable file, use the following:

```

add_library(arithmetic STATIC add.c sub.c mul.c dur.c)
include_directories(.)

```

CMakeLists.txt (2<sup>nd</sup> version)

If you want to make a shared library, use the following:

```
add_library(arithmetic SHARED add.c sub.c mul.c dur.c)
include_directories(.)
```

CMakeLists.txt (3<sup>rd</sup> version)

In the following, we have more CMake commands. `project()` specifies the name of the project. `cmake_minimum_required()` specifies the minimal version of the cmake which is allowed to build this project. The following configuration first builds the shared library “libarithmetic.so” and link it to the final executable “prog.” When doing that, you have to specify the library’s location using `link_directories()`. You can specify the compiler option using `add_compile_options()`. Note that the include directory is changed to `inc/`. Thus you have to move all the header files in `inc/` directory.

```
project(hello)
cmake_minimum_required(VERSION 3.10)
include_directories(inc)
link_directories(prog PUBLIC .)
add_compile_options(-Wall)

add_library(arithmetic SHARED add.c sub.c mul.c dur.c)

add_executable(prog cal_main.c)
target_link_libraries(prog PUBLIC arithmetic)
```

CMakeLists.txt (4<sup>th</sup> version)

CMake is powerful since it can generate other build systems like MS Visual Studio. You can use the same CMakeLists.txt to generate Linux build system, MS Windows build system, and the Mac OS build system.

Note you can download the docker image as in the following:

```
$ docker run -it jongchank/ubuntu_dev_tutorial:latest
```