LEARN MAKEFILE

unix makefile

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

About the Tutorial

Makefile is a program building tool which runs on Unix, Linux, and their flavors. It aids in simplifying building program executables that may need various modules. To determine how the modules need to be compiled or recompiled together, make takes the help of user-defined makefiles. This tutorial should enhance your knowledge about the structure and utility of makefile.

Audience

Makefile guides the **make** utility while compiling and linking program modules. Anyone who wants to compile their programs using the **make** utility and wants to gain knowledge on makefile should read this tutorial.

Prerequisites

This tutorial expects good understanding of programming language such as C and C++. The reader is expected to have knowledge of linking, loading concepts, and how to compile and execute programs in Unix/Linux environment.

Disclaimer & Copyright

Contents

1. WHY MAKEFILE?

Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling command everytime you need to compile. Makefiles are the solution to simplify this task.

Makefiles are special format files that help build and manage the projects automatically.

For example, let's assume we have the following source files.

- main.cpp
- hello.cpp
- factorial.cpp
- functions.h

main.cpp:

```
#include <iostream.h>

#include "functions.h"

int main(){
    print_hello();
    cout << endl;
    cout << "The factorial of 5 is " << factorial(5) << endl;
    return 0;
}
```

hello.cpp:

```
#include <iostream.h>

#include "functions.h"

void print_hello(){
   cout << "Hello World!";
}
```

factorial.cpp:

```
#include "functions.h"


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

functions.h:

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

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

```
CC  main.cpp hello.cpp factorial.cpp -o hello
```

This command generates *hello* binary. In this example, we have only four files and we know the sequence of the function calls. Hence, it is feasible to type the above command and prepare a final binary.

However, for a large project where we have thousands of source code files, it becomes difficult to maintain the binary builds.

The **make** command allows you to manage large programs or groups of programs. As you begin to write large programs, you notice that re-compiling large programs takes longer time than re-compiling short programs. Moreover, you notice that you usually only work on a small section of the program such as a single function, and much of the remaining program is unchanged.

In the subsequent section, we see how to prepare a makefile for our project.

2. MACROS

The **make** program allows you to use macros, which are similar to variables. Macros are defined in a Makefile as = pairs. For example,

```
MACROS=  -me
PSROFF=  groff -Tps
DITROFF= groff -Tdvi
CFLAGS= -O -systype bsd43
LIBS = "-lncurses -lm -lsdl"
MYFACE = ":*)"
```

Special Macros

Before issuing any command in a target rule set, there are certain special macros predefined:

- $@ is the name of the file to be made.
- $? is the names of the changed dependents.

For example, we could use a rule as follows:

```
hello: main.cpp hello.cpp factorial.cpp
     $(CC) $(CFLAGS) $? $(LDFLAGS) -o $@

alternatively:

hello: main.cpp hello.cpp factorial.cpp
        $(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@
```

In this example, $@ represents *hello* and $? or $@.cpp picks up all the changed source files.

There are two more special macros used in the implicit rules. They are:

- $< the name of the related file that caused the action.
- $* the prefix shared by target and dependent files.

Common implicit rule is for the construction of .o (object) files out of .cpp (source files).

```
.o.cpp:
        $(CC) $(CFLAGS) -c $<

alternatively:

.o.cpp:
        $(CC) $(CFLAGS) -c $*.c
```

Conventional Macros

There are various default macros. You can see them by typing "make -p" to print out the defaults. Most are pretty obvious from the rules in which they are used.

These predefined variables, i.e., macros used in implicit rules fall into two classes:

1. Macros that are names of programs (such as CC)
2. Macros that contain arguments of the programs (such as CFLAGS).

Here is a table of some of the common variables used as names of programs in built-in rules of makefiles.

AR	Archive-maintaining program; default is 'ar'.
AS	Program for compiling assembly files; default is 'as'.
CC	Program for compiling C programs; default is 'cc'.
CO	Program for checking out files from RCS; default is 'co'.
CXX	Program for compiling C++ programs; default is 'g++'.
CPP	Program for running the C preprocessor, with results to standard output; default is '$(CC) -E'.
FC	Program for compiling or preprocessing Fortran and Ratfor programs; default is 'f77'.
GET	Program for extracting a file from SCCS; default is 'get'.
LEX	Program to use to turn Lex grammars into source code; default is 'lex'.

YACC	Program to use to turn Yacc grammars into source code; default is 'yacc'.
LINT	Program to use to run lint on source code; default is 'lint'.
M2C	Program to use to compile Modula-2 source code; default is 'm2c'.
PC	Program for compiling Pascal programs; default is 'pc'.
MAKEINFO	Program to convert a Texinfo source file into an Info file; default is 'makeinfo'.
TEX	Program to make TeX dvi files from TeX source; default is 'tex'.
TEXI2DVI	Program to make TeX dvi files from Texinfo source; default is 'texi2dvi'.
WEAVE	Program to translate Web into TeX; default is 'weave'.
CWEAVE	Program to translate C Web into TeX; default is 'cweave'.
TANGLE	Program to translate Web into Pascal; default is 'tangle'.
CTANGLE	Program to translate C Web into C; default is 'ctangle'.
RM	Command to remove a file; default is 'rm -f'.

Here is a table of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

ARFLAGS	Flags to give the archive-maintaining program; default is 'rv'.
ASFLAGS	Extra flags to give to the assembler when explicitly invoked on a '.s' or '.S' file.
CFLAGS	Extra flags to give to the C compiler.
CXXFLAGS	Extra flags to give to the C compiler.

COFLAGS	Extra flags to give to the RCS co program.
CPPFLAGS	Extra flags to give to the C preprocessor and programs, which use it (such as C and Fortran compilers).
FFLAGS	Extra flags to give to the Fortran compiler.
GFLAGS	Extra flags to give to the SCCS get program.
LDFLAGS	Extra flags to give to compilers when they are supposed to invoke the linker, 'ld'.
LFLAGS	Extra flags to give to Lex.
YFLAGS	Extra flags to give to Yacc.
PFLAGS	Extra flags to give to the Pascal compiler.
RFLAGS	Extra flags to give to the Fortran compiler for Ratfor programs.
LINTFLAGS	Extra flags to give to lint.

NOTE: You can cancel all variables used by implicit rules with the '-R' or '--no-builtin-variables' option.

You can also define macros at the command line as shown below:

```
make CPP = /home/courses/cop4530/spring02
```

3. DEPENDENCIES

It is very common that a final binary will be dependent on various source code and source header files. Dependencies are important because they let the **make** know about the source for any target. Consider the following example:

```
hello: main.o factorial.o hello.o
     $(CC) main.o factorial.o hello.o -o hello
```

Here, we tell the **make** that hello is dependent on main.o, factorial.o, and hello.o files. Hence, whenever there is a change in any of these object files, **make** will take action.

At the same time, we need to tell the **make** how to prepare .o files. Hence, we need to define those dependencies also as follows:

```
main.o: main.cpp functions.h
     $(CC) -c main.cpp

factorial.o: factorial.cpp functions.h
     $(CC) -c factorial.cpp

hello.o: hello.cpp functions.h
     $(CC) -c hello.cpp
```

4. RULES

The general syntax of a Makefile target rule is:

```
target [target...] : [dependent ....]
[ command ...]
```

Arguments in brackets are optional, ellipsis means one or more. Note the tab to preface each command is required.

A simple example is given below where you define a rule to make your target hello from three other files.

```
hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello
```

NOTE: In this example, you would have to give rules to make all object files from the source files.

The semantics is pretty simple. When you say "make target", the **make** finds the target rule that applies and if any of the dependents are newer than the target, the **make** executes the commands one at a time (after macro substitution). If any dependents have to be made, that happens first (so you have a recursion.

A **make** will terminate if any command returns a failure status. That's why you see rules like:

```
clean:
        -rm *.o *~ core paper
```

Make ignores the returned status on command lines that begin with a dash. For example, who cares if there is no core file?

Make echoes the commands, after macro substitution to show you what is happening as it happens. Sometimes you might want to turn that off. For example:

```
install:
        @echo You must be root to install
```

People have come to expect certain targets in Makefiles. You should always browse first. However, it is reasonable to expect that the targets all (or just make), install, and clean is found.

- **make all** – It compiles everything so that you can do local testing before installing applications.
- **make install** – It installs applications at right places.

- **make clean** – It cleans applications up, gets rid of the executables, any temporary files, object files, etc.

Makefile Implicit Rules

The command is one that ought to work in all cases where we build an executable x out of the source code x.cpp. This can be stated as an implicit rule:

```
.cpp:
        $(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@
```

This Implicit rule says how to make x out of x.c -- run cc on x.c and call the output x. The rule is implicit because no particular target is mentioned. It can be used in all cases.

Another common implicit rule is for the construction of .o (object) files out of .cpp (source files).

```
.o.cpp:
        $(CC) $(CFLAGS) -c $<

Alternatively:

.o.cpp:
        $(CC) $(CFLAGS) -c $*.cpp
```

5. SUFFIX RULES

By itself, **make** already knows that in order to create a *.o* file, it must use *cc -c* on the corresponding *.c* file. These rules are built into the **make**, and you can take their advantage to shorten your Makefile. If you indicate just the *.h* files in the dependency line of the Makefile that the current target is dependent on, **make** will know that the corresponding *.c*file is already required. You do not even need to include the command for the compiler.

This reduces the Makefile further, as shown:

```
OBJECTS = main.o hello.o factorial.o
hello: $(OBJECTS)
        cc $(OBJECTS) -o hello
hellp.o: functions.h
main.o: functions.h
factorial.o: functions.h
```

Make uses a special target, named *.SUFFIXES* to allow you to define your own suffixes. For example, refer the dependency line:

```
.SUFFIXES: .foo .bar
```

It informs the **make** that you will be using these special suffixes to make your own rules.

Similar to how **make** already knows how to make a *.o* file from a *.c* file, you can define rules in the following manner:

```
.foo.bar:
        tr '[A-Z][a-z]' '[N-Z][A-M][n-z][a-m]' < $< > $@
.c.o:
        $(CC) $(CFLAGS) -c $<
```

The first rule allows you to create a *.bar* file from a *.foo* file. It basically scrambles the file. The second rule is the default rule used by **make** to create a*.o* file from a *.c* file.

6. DIRECTIVES

There are numerous directives available in various forms. The **make** program on your system may not support all the directives. So please check if your **make** supports the directives we are explaining here. **GNU make** supports these directives.

Conditional Directives

The conditional directives are:

- The **ifeq** directive begins the conditional, and specifies the condition. It contains two arguments, separated by a comma and surrounded by parentheses. Variable substitution is performed on both arguments and then they are compared. The lines of the makefile following the ifeq are obeyed if the two arguments match; otherwise they are ignored.
- The **ifneq** directive begins the conditional, and specifies the condition. It contains two arguments, separated by a comma and surrounded by parentheses. Variable substitution is performed on both arguments and then they are compared. The lines of the makefile following the ifneq are obeyed if the two arguments do not match; otherwise they are ignored.
- The **ifdef** directive begins the conditional, and specifies the condition. It contains single argument. If the given argument is true then condition becomes true.
- The **ifndef** directive begins the conditional, and specifies the condition. It contains single argument. If the given argument is false then condition becomes true.
- The **else** directive causes the following lines to be obeyed if the previous conditional failed. In the example above this means, the second alternative linking command is used whenever the first alternative is not used. It is optional to have an else in a conditional.
- The **endif** directive ends the conditional. Every conditional must end with an endif.

Syntax of Conditionals Directives

The syntax of a simple conditional with no else is as follows:

```
conditional-directive
text-if-true
endif
```

The text-if-true may be any lines of text, to be considered as part of the makefile if the condition is true. If the condition is false, no text is used instead.

The syntax of a complex conditional is as follows:

```
conditional-directive
text-if-true
else
text-if-false
endif
```

If the condition is true, text-if-true is used; otherwise, text-if-false is used. The text-if-false can be any number of lines of text.

The syntax of the conditional-directive is the same whether the conditional is simple or complex. There are four different directives that test various conditions. They are as given:

```
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"
```

Opposite directives of the above conditions are are follows:

```
ifneq (arg1, arg2)
ifneq 'arg1' 'arg2'
ifneq "arg1" "arg2"
ifneq "arg1" 'arg2'
ifneq 'arg1' "arg2"
```

Example of Conditionals Directives

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

The include Directive

The include directive tells the **make** to suspend reading the current makefile and read one or more other makefiles before continuing. The directive is a line in the makefile that looks as follows:

```
include filenames...
```

The filenames can contain shell file name patterns. Extra spaces are allowed and ignored at the beginning of the line, but a tab is not allowed. For example, if you have three '.mk' files namely, 'a.mk', 'b.mk', and 'c.mk', and $(bar), then it expands to bish bash, and then the following expression.

```
include foo *.mk $(bar)


is equivalent to


include foo a.mk b.mk c.mk bish bash
```

When the **make** processes an include directive, it suspends reading of the containing makefile and reads from each listed file in turn. When that is finished, **make** resumes reading the makefile in which the directive appears.

The override Directive

If a variable has been set with a command argument, then ordinary assignments in the makefile are ignored. If you want to set the variable in the makefile even though it was set with a command argument, you can use an override directive, which is a line that looks as follows:

```
override variable = value


or


override variable := value
```

7. RECOMPILATION

The **make** program is an intelligent utility and works based on the changes you do in your source files. If you have four files main.cpp, hello.cpp, factorial.cpp, and functions.h, then all the remaining files are dependent on functions.h, and main.cpp is dependent on both hello.cpp and factorical.cpp. Hence if you make any changes in functions.h, then the **make** recompiles all the source files to generate new object files. However, if you make any change in main.cpp, as this is not dependent of any other file, then only main.cpp file is recompiled, and hellp.cpp and factorial.cpp are not.

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

Avoiding Recompilation

There may be a project consisting of thousands of files. Sometimes you may have changed a source file but you may not want to recompile all the files that depend on it. For example, suppose you add a macro or a declaration to a header file, on which the other files depend. Being conservative, the **make** assumes that any change in the header file requires recompilation of all dependent files, but you know that they do not need recompilation and you would rather not waste your time waiting for them to compile.

If you anticipate the problem before changing the header file, you can use the `-t' flag. This flag tells **make** not to run the commands in the rules, but rather to mark the target up to date by changing its last-modification date. You need to follow this procedure:

1. Use the command 'make' to recompile the source files that really need recompilation.
2. Make the changes in the header files.
3. Use the command 'make -t' to mark all the object files as up to date. The next time you run make, the changes in the header files do not cause any recompilation.

If you have already changed the header file at a time when some files do need recompilation, it is too late to do this. Instead, you can use the '-o file' flag, which marks a specified file as "old". This means, the file itself will not be remade, and nothing else will be remade on its account. You need to follow this procedure:

1. Recompile the source files that need compilation for reasons independent of the particular header file, with 'make -o headerfile'. If several header files are involved, use a separate '-o' option for each header file.
2. Update all the object files with 'make -t'.

8. OTHER FEATURES

Recursive Use of Make

Recursive use of **make** means using **make** as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory named 'subdir' which has its own makefile, and you would like the containing directory's makefile to run **make** on the subdirectory. You can do it by writing this:

```
subsystem:
        cd subdir && $(MAKE)


or, equivalently


subsystem:
        $(MAKE) -C subdir
```

You can write recursive **make** commands just by copying this example, but you need to know about how they work and why, and about how the sub-make relates to the top-level make.

Communicating Variables to a Sub-Make

Variable values of the top-level **make** can be passed to the sub-make through the environment by explicit request. These variables are defined in the sub-make as defaults. You cannot override what is specified in the makefile used by the sub-make makefile unless you use the '-e' switch.

To pass down or export a variable, **make** adds the variable and its value to the environment for running each command. The sub-make, in turn, uses the environment to initialize its table of variable values.

The special variables SHELL and MAKEFLAGS are always exported (unless you unexport them). MAKEFILES is exported if you set it to anything.

If you want to export specific variables to a sub-make, use the export directive, as shown:

```
export variable ...
```

If you want to prevent a variable from being exported, use the unexport directive, as shown:

```
unexport variable ...
```

The Variable MAKEFILES

If the environment variable MAKEFILES is defined, **make** considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the include directive: various directories are searched for those files.

The main use of MAKEFILES is in communication between recursive invocations of the **make**.

Including Header File from Different Directories

If you have put the header files in different directories and you are running **make** in a different directory, then it is required to provide the path of header files. This can be done using -I option in makefile. Assuming that functions.h file is available in /home/tutorialspoint/header folder and rest of the files are available in /home/tutorialspoint/src/ folder, then the makefile would be written as follows.

```
INCLUDES = -I "/home/tutorialspoint/header"
CC = gcc
LIBS =  -lm
CFLAGS = -g -Wall
OBJ =  main.o factorial.o hello.o

hello: ${OBJ}
   ${CC} ${CFLAGS} ${INCLUDES} -o $@ ${OBJS} ${LIBS}
.cpp.o:
   ${CC} ${CFLAGS} ${INCLUDES} -c $<
```

Appending More Text to Variables

Often it is useful to add more text to the value of a variable already defined. You do this with a line containing `+=', as shown:

```
objects += another.o
```

It takes the value of the variable objects, and adds the text 'another.o' to it, preceded by a single space. Thus:

```
objects = main.o hello.o factorial.o
objects += another.o
```

sets objects to 'main.o hello.o factorial.o another.o'.

Using '+=' is similar to:

```
objects = main.o hello.o factorial.o
objects := $(objects) another.o
```

Continution Line in Makefile

If you do not like too big lines in your Makefile, then you can break your line using a back-slash "\" as shown below:

```
OBJ =  main.o factorial.o \
     hello.o


is equivalent to


OBJ =  main.o factorial.o hello.o
```

Running Makefile from Command Prompt

If you have prepared the Makefile with name "Makefile", then simply write make at command prompt and it will run the Makefile file. But if you have given any other name to the Makefile, then use the following command:

```
make -f your-makefile-name
```

9. EXAMPLE

This is an example of the Makefile for compiling the hello program. This program consists of three files *main.cpp*, *factorial.cpp,* and *hello.cpp*.

```
# Define required macros here
SHELL = /bin/sh

OBJS =  main.o factorial.o hello.o
CFLAG = -Wall -g
CC = gcc
INCLUDE =
LIBS = -lm

hello:${OBJ}
     ${CC} ${CFLAGS} ${INCLUDES} -o $@ ${OBJS} ${LIBS}

clean:
     -rm -f *.o core *.core

.cpp.o:
     ${CC} ${CFLAGS} ${INCLUDES} -c $<
```

Now you can build the program hello using the **make**. If you issue a command 'make clean' then it removes all the object files and core files present in the current directory.