



LEARN MAKEFILE

unix makefile

tutorialspoint
SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

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 also the knowledge of compiling and executing programs in Unix/Linux environment.

Disclaimer & Copyright

© Copyright 2019 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher. We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com.

Contents

About the Tutorial	i
Audience	i
Prerequisites	i
Disclaimer & Copyright	i
Contents	ii
1. WHY MAKEFILE?	1
2. MACROS.....	3
 Special Macros.....	3
 Conventional Macros.....	4
3. DEPENDENCIES.....	7
4. RULES.....	8
 Makefile Implicit Rules	9
5. SUFFIX RULES	10
6. DIRECTIVES.....	11
 Conditional Directives.....	11
 Syntax of Conditionals Directives.....	11
 The include Directive	12
 The override Directive	13
7. RECOMPILATION	14
 Avoiding Recompilation.....	14
8. OTHER FEATURES	15
 Recursive Use of Make	15
 Communicating Variables to a Sub-Make	15
 The Variable MAKEFILES	16

Including Header File from Different Directories	16
Appending More Text to Variables	16
Continuation Line in Makefile.....	17
Running Makefile from Command Prompt	17
9. EXAMPLE.....	18

1. WHY MAKEFILE?

Compiling the source code files can be tiring, especially when you have to include several source files and type the compiling commands every time. 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

The following is the code for main.cpp source file:

```
#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

The code given below is for hello.cpp source file:

```
#include <iostream.h>

#include "functions.h"

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

factorial.cpp

The code for factorial.cpp is given below:

```
#include "functions.h"

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

functions.h

The following is the code for functions.h:

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

The trivial way to compile the files and obtain an executable is by running the following 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 will 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. An example has been shown below:

```
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 name 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 given below for the construction of .o (object) files out of .cpp (source files).

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

alternatively:

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

End of ebook preview

If you liked what you saw...

Buy it from our store @ **<https://store.tutorialspoint.com>**