

# **A Short Introduction to Makefile**

Fangtian Zhong  
CSCI 112

2024.04.01

Gianforte School of Computing  
Norm Asbjornson College of Engineering  
E-mail: [fangtian.zhong@montana.edu](mailto:fangtian.zhong@montana.edu)

# Example Makefile

```
# This is a comment line
CC=gcc
# CFLAGS will be the options passed to the compiler.
CFLAGS= -c -Wall

all: prog

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

main.o: main.c
    $(CC) $(CFLAGS) main.c

factorial.o: factorial.c
    $(CC) $(CFLAGS) factorial.c

hello.o: hello.c
    $(CC) $(CFLAGS) hello.c

clean:
    rm -rf *.o prog
```

```
/* main.c */
#include <stdio.h>
#include "functions.h"

int main() {
    print_hello();
    printf("\n");
    printf("The factorial of 5 is %d\n",
factorial(5));
    return 0;
}
```

```
/* factorial.c */
#include "functions.h"

int factorial(int n)
{
    int i, fac = 1;
    if(n!=1){
        for(i=1; i<= n; i++)
            fac *= i;
        return fac;
    }
    else return 1;
}
```

```
/* hello.c */  
#include <stdio.h>  
#include "functions.h"  
  
void print_hello() {  
    printf("Hello World!");  
}
```

```
/* functions.h */  
void print_hello();  
int factorial(int n);
```

# Basic Makefile Structure

## Macros

- By using macros, we can avoid repeating text entries and makefile is easy to modify.
- Macro definitions have the form:
  - NAME = text string
  - e.g. we have: CC=gcc
- Macros are referred to by placing the name in either parentheses or curly braces and preceding it with \$ sign.
  - E.g. \$(CC) main.o factorial.o hello.o -o prog

# Basic Makefile Structure

## Internal macros

- Internal macros are predefined in *make*.
- “*make -p*” to display a listing of all the macros, suffix rules and targets in effect for the current build.




## Special macros

- The macro `@` evaluates to the name of the current target.
  - E.g.  
prog1 : \$(objs)  
    \$(CC) -o `$@` \$(objs)  
is equivalent to  
prog1 : \$(objs)  
    \$(CC) -o prog1 \$(objs)

## Suffix rules

- A way to define default rules or implicit rules that *make* can use to build a program. There are *double-suffix* and *single-suffix*.
  - Suffix rules are obsolete and are supported for compatibility. Use pattern rules (a rule contains character ‘%’) if possible.
  - Doubles-suffix is defined by the source suffix and the target suffix . E.g. .cpp.o:  
\$(CC) \$(CFLAGS) -c \$<
    - This rule tells make that .o files are made from .c files.
    - \$< is a special macro which in this case stands for a .cpp file that is used to produce a .o file.

## How Does Make Work?

-  The make utility compares the modification time of the target file with the modification times of the dependency files. Any dependency file that has a more recent modification time than its target file forces the target file to be recreated.
-  By default, the first target file is the one that is built. Other targets are checked only if they are dependencies for the first target.
-  Except for the first target, the order of the targets does not matter. The make utility will build them in the order required.



# A New Makefile

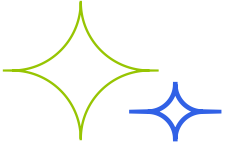
```
# This is a comment line
CC=g++
# CFLAGS will be the options passed to the compiler.
CFLAGS=-c -Wall
OBJECTS = main.o hello.o factorial.o

all: prog

prog: $(OBJECTS)
    $(CC) $(OBJECTS) -o prog

%.o: %.cpp
    $(CC) $(CFLAGS) $<

clean:
    rm -rf *.o
```



# THE END

Fangtian Zhong  
CSCI 112

2024.04.01

Gianforte School of Computing  
Norm Asbjornson College of Engineering  
E-mail: [fangtian.zhong@montana.edu](mailto:fangtian.zhong@montana.edu)