



# Programming with C I

Fangtian Zhong  
CSCI 112

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

```
/* main.cpp */
#include <iostream>
#include "functions.h"

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

```
/* factorial.cpp */
#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.cpp */  
#include <iostream>  
#include "functions.h"  
  
using namespace std;  
void print_hello()  
{  
    cout << "Hello World!";  
}
```

```
/* functions.h */  
#if !defined(_FUNC_H_)  
#define _FUNC_H_  
  
void print_hello();  
int factorial(int n);  
  
#endif /* if !define(_FUNC_H_) */
```

# 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.cpp
    $(CC) $(CFLAGS) main.c

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

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

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

# 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 arenttheses 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.
  - Double-suffix is defined by the source suffix and the target suffix. E.g. .c .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 .c 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.



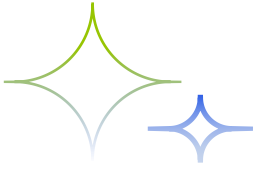
# A New Makefile

```
# This is a comment line
CC=gcc
# 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: %.c
    $(CC) $(CFLAGS) $<

clean:
    rm -rf *.o
```



# THE END

Fangtian Zhong  
CSCI 112

2024.10.25

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