Unix Programming

The Makefile Utility

Nguyen Thanh Hung

Hanoi University of Science and Technology

May 2015

Example

main.c

```c
#include <stdio.h>

void print_hello() {
        printf("Hello World!");
}


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

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

```
gcc -o run main.c
```

Motivation

- Many lines of code
- More than one programmer
- Long files are harder to manage (for both **programmers** and **machines**)
- Every change requires long compilation
- Many programmers can not modify the same file simultaneously

Split the code into multiple **files** or **components**.

Bad Solution

main.c

```
#include <stdio.h>
#include "hello.c"
#include "factorial.c"
int main() {
        print_hello();
        printf("\n");
        printf("The factorial of 5 is %d\n", factorial(5));
        return 0;
}
```

factorial.c

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

hello.c

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

```
gcc -o run main.c
# Every change requires long compilation
# Include the same .c file in different files => redefinition
```

Header files

- keep the function declaration separate from function definition.
- Your programs often need to use functions “defined" elsewhere.
- Using the header files, you can “compile" your programs without needing the function definition.
- From your `.c`/`.cpp` file, an `.obj` file is generated. You can now distribute your obj file along with the `.h` file. You now do not have to distribute the actual code.
- Order of functions in the `.c`/`.cpp` files.

Example

You don’t have the source code for the windows functions. Instead you include `windows.h` file in your programs and the **linker** extracts the binary function definitions from the `.obj` files and merges them into your program’s .exe file.

Good Solution

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;
}
```

functions.h

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

factorial.c

```
#include "functions.h"
int factorial(int n) {
        if(n!=1)
                return(n*factorial(n-1));
        else
                return 1;
}
```

hello.c

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

```
gcc -c factorial.c # generates factorial.o
gcc -c main.c # generates main.o
gcc -c hello.c # generates hello.o

gcc -o run main.o factorial.o hello.o # generates run
```

Makefile - Motivation

- If you lose the compile command or switch computers you have to retype it from **scratch**.
- Many source and header files!
- Determining which modules to recompile can be difficult when working on large programs.
- The make utility automates the process

```
target: prerequisite-list
[TAB] construction commands
```

Makefile - Example

`Makefile`

```
run: hello.o main.o factorial.o
        gcc -o run hello.o main.o factorial.o

main.o: main.c functions.h
        gcc -c main.c

factorial.o: factorial.c functions.h
        gcc -c factorial.c

hello.o: hello.c functions.h
        gcc -c hello.c
```

```
make
gcc -c hello.c
gcc -c main.c
gcc -c factorial.c
gcc -o run hello.o main.o factorial.o

make
make: 'run' is up to date.
```

The default goal is the target of the first rule in the first `Makefile`. (run in this case).

A Better Solution

main.c

```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;
}
```

functions.h

```c
#include "hello.h"
#include "factorial.h"
```

hello.h

```c
void print_hello();
```

factorial.h

```c
int factorial(int n);
```

factorial.c

```c
#include "factorial.h"
int factorial(int n) {
        if(n!=1)
                return(n*factorial(n-1));
        else
                return 1;
}
```

hello.c

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

Makefile - Example

Makefile

```
run: hello.o main.o factorial.o
        gcc -o run hello.o main.o factorial.o

main.o: main.c functions.h
        gcc -c main.c

factorial.o: factorial.c factorial.h
        gcc -c factorial.c

hello.o: hello.c hello.h
        gcc -c hello.c
```

```
make
gcc -c hello.c
gcc -c main.c
gcc -c factorial.c
gcc -o run hello.o main.o factorial.o

# Modify hello.h
make
gcc -c hello.c
gcc -c main.c
gcc -o run hello.o main.o factorial.o
```

Makefile - New Rules - clean

Makefile

```
run: hello.o main.o factorial.o
        gcc -o run hello.o main.o factorial.o

main.o: main.c functions.h
        gcc -c main.c

factorial.o: factorial.c factorial.h
        gcc -c factorial.c

hello.o: hello.c hello.h
        gcc -c hello.c
clean:
        rm -rf *.o run
```

```
make clean
rm -rf *.o run
```

wildcard,.PHONY

One use of the wildcard function is to get a list of all the C source files in a directory. We can change the list of C source files into a list of object files by replacing the ‘.o’ suffix with ‘.c’ in the result:

```
$(wildcard ∗.c)
```

```
$(patsubst %.c,%.o,$(wildcard ∗.c))
```

```
OBJECTS = $(patsubst %.c,%.o,$(wildcard ∗.c)) # define variable

run: $(OBJECTS)
        gcc -o run $(OBJECTS)

main.o: main.c functions.h factorial.h hello.h
        gcc -c main.c

factorial.o: factorial.c factorial.h
        gcc -c factorial.c

hello.o: hello.c hello.h
        gcc -c hello.c

.PHONY: clean # solve the problem when you have a file named clean, no dependency!
clean:
        rm -rf ∗.o run
```

Another Solution

```
SOURCES = main.c hello.c factorial.c
OBJECTS = $(SOURCES:.c=.o)

run: $(OBJECTS)
        gcc -o run $(OBJECTS)

main.o: main.c functions.h factorial.h hello.h
        gcc -c main.c

factorial.o: factorial.c factorial.h
        gcc -c factorial.c

hello.o: hello.c hello.h
        gcc -c hello.c

.PHONY: clean # solve the problem when you have a file named clean, no dependency!
clean:
        rm -rf *.o run
```

Static Library

```
library : factorial.o hello.o
        ar rcs libcmps.a factorial.o hello.o # create static library libcmps.a
```

```
run: main.c
        gcc -o run main.o -L. -lcmps

main.o: main.c functions.h factorial.h hello.h
        gcc -c main.c
factorial.o: factorial.c factorial.h
        gcc -c factorial.c
hello.o: hello.c hello.h
        gcc -c hello.c

library : factorial.o hello.o
        ar rcs libcmps.a factorial.o hello.o # create static library libcmps.a

.PHONY: clean # solve the problem when you have a file named clean, no dependency!
clean:
        rm -rf *.o run
```

```
make library # create libcmps.a
make # gcc -o run main.o -L. -lcmps
```

Macro definitions for flexibility

```
CC = g++
DEBUG = -g
CFLAGS = -Wall -c $(DEBUG)
LFLAGS = -Wall $(DEBUG)
```

- `CC` The name of the compiler
- `DEBUG` The debugging flag. This is -g in both g++ and cxx. The purpose of the flag is to include debugging information into the executable, so that you can use utilities such as gdb to debug the code.
- `LFLAGS` The flags used in linking. `-Wall` tells the compiler to print all warnings.
- `CFLAGS` The flags used in compiling and creating object files. The “-c" option is needed to create object files, i.e. .o files.

Makefile - Example - Macro

```
CC = gcc
CFLAGS = -Wall -c
LFLAGS = -Wall -o
OUTPUT = run
LIBS = -L. -lcmps
INCLUDES = -I../path/to/include
run: main.o
        $(CC) $(LFLAGS) $(OUTPUT) main.o $(LIBS)

main.o: main.c functions.h factorial.h hello.h
        $(CC) $(CFLAGS) main.c $(INCLUDES)

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

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

library : factorial.o hello.o
        ar rcs libcmps.a factorial.o hello.o # create static library libcmps.a

.PHONY: clean # solve the problem when you have a file named clean, no dependency!
clean:
        rm -rf *.o libcmps.a run
```

Special Macro

```
‘$^’ => list of all the dependencies
‘$@’ => value of target target
‘$<’ => value of the first dependency
```

```
CC = gcc
CFLAGS = -Wall -c
LFLAGS = -Wall -o
OUTPUT = run
LIBS = -L. -lcmps
INCLUDES = -I../path/to/include
$(OUTPUT): main.o
        $(CC) $(LFLAGS) $@ $^ $(LIBS)

main.o: main.c functions.h factorial.h hello.h
        $(CC) $(CFLAGS) $< $(INCLUDES)

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

hello.o: hello.c hello.h
        $(CC) $(CFLAGS) $<
...
```

Makefile

```
SOURCES=main.c hello.c factorial.c
OBJECTS=$(SOURCES:.c=.o)

CC = g++
CFLAGS = -Wall -c
LFLAGS = -Wall -o
OUTPUT = run

$(OUTPUT): $(OBJECTS)
        $(CC) $(LFLAGS) $@ $^

.c.o:
        $(CC) $(CFLAGS) $<
        @echo "Compilation of $< done..."

.PHONY: clean # solve the problem when you have a file named clean, no dependency!
clean:
        rm -rf *.o libcmps.a run
```