Writing Large Programs (3)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Makefiles

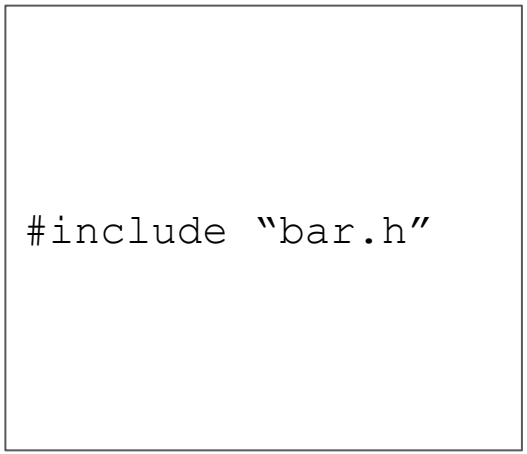
Makefiles

- In the last lecture, we introduced the GCC command that builds `justify` with multiple files.
- However, what if the program we want to build has 10 or even a hundred source files? Do we need to type a long command to build the program every time I revised the codes?

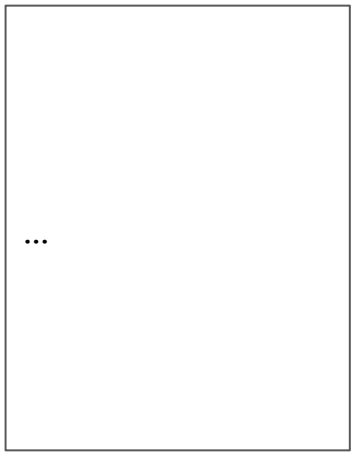
```
gcc -o justify justify.c line.c word.c
```

Makefiles

- To make it easier to build large programs, UNIX originated the concept of the ***makefile.***
- A makefile not only lists the files that are part of the program, but also describes ***dependencies*** among the files.



foo.c



bar.h

We say that `foo.c` "depends" on `bar.h`, because a change to `bar.h` will require us to recompile `foo.c`.

Makefiles

- A UNIX makefile for the `justify` program (use GCC to compile)

```
justify: justify.o word.o line.o
 tab  gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
     gcc -c justify.c

word.o: word.c word.h
     gcc -c word.c

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

Each command in a makefile must be preceded by a tab character

Makefiles

- you can also download it from eCourse2 (`Justify_source_code.zip`)
- I used `clang` to compile. Please type the correct compiler you use!

```
justify: justify.o word.o line.o
    clang -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
    clang -c justify.c

word.o: word.c word.h
    clang -c word.c

line.o: line.c line.h
    clang -c line.c
```

Makefiles

- There are four groups of lines
- Each group is known as a ***rule.***

```
justify: justify.o word.o line.o
    gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
    gcc -c justify.c

word.o: word.c word.h
    gcc -c word.c

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

Makefiles

- The first line in each rule gives a ***target*** file, followed by the files on which it depends.
- The second line is a ***command*** to be executed if the target should need to be rebuilt because of a change to one of its dependent files.

```
justify: justify.o word.o line.o
    gcc -o justify justify.o word.o line.o

justify.o: justify.c word.h line.h
    gcc -c justify.c
```

Makefiles

- In the first rule, `justify` (the executable file) is the target
- The first line states that `justify` depends on the files `justify.o`, `word.o`, and `line.o`.
- If any of these files have changed since the program was last built, `justify` needs to be rebuilt.
- The command on the following line shows how the rebuilding is to be done

```
justify: justify.o word.o line.o
    gcc -o justify justify.o word.o line.o
```

Makefiles

- In the second rule, `justify.o` is the target
- The first line indicates that `justify.o` needs to be rebuilt if there's been a change to `justify.c`, `word.h`, or `line.h`.
- The next line shows how to update `justify.o` (by recompiling `justify.c`).
- The `-c` option tells the compiler to compile `justify.c` but **not attempt to link it.**

```
justify.o: justify.c word.h line.h
    gcc -c justify.c
```

Makefiles

- Now we have seen the structure of a Makefiles, so what exactly is the advantages of using Makefiles?
- Because it can shorten the command to rebuild a program and can also automatically checked which source files need to be recompiled!
- A makefile is normally stored in a file named `Makefile` (or `makefile`).

Makefiles

- Once we've created a makefile for a program, we can use the `make` utility to build (or rebuild) the program.
- By checking the time and date associated with each file in the program, `make` can determine which files are out of date.
- `make` automatically checks the current directory for a file with one of these names.

Makefiles

- To invoke `make`, use the command

  `make` *target*

  where *target* is one of the targets listed in the makefile.

```
fuyincherng@MacBook-Air multi_files % make justify
clang -o justify justify.o word.o line.o
fuyincherng@MacBook-Air multi_files % make word.o
make: `word.o' is up to date.
fuyincherng@MacBook-Air multi_files % make justify
make: `justify' is up to date.
```

Makefiles

- If no target is specified when `make` is invoked, it will build the target of the first rule.
- Except for this special property of the first rule, the order of rules in a makefile is arbitrary.
  - For example, you can switch the order of the rules for `line.o` and `word.o` in the makefile

```
fuyincherng@MacBook-Air multi_files % make justify
make: `justify' is up to date.
fuyincherng@MacBook-Air multi_files % make
make: `justify' is up to date.
fuyincherng@MacBook-Air multi_files %
```

Makefiles

- Real makefiles aren't always easy to understand.
- Actually, `make` is complicated enough that we need another course to introduce it.
- In this course, we just want you to understand the basic concept of makefile and the simplest example
- Alternatives to makefiles include the "project files" supported by some integrated development environments.