Makefiles & Project

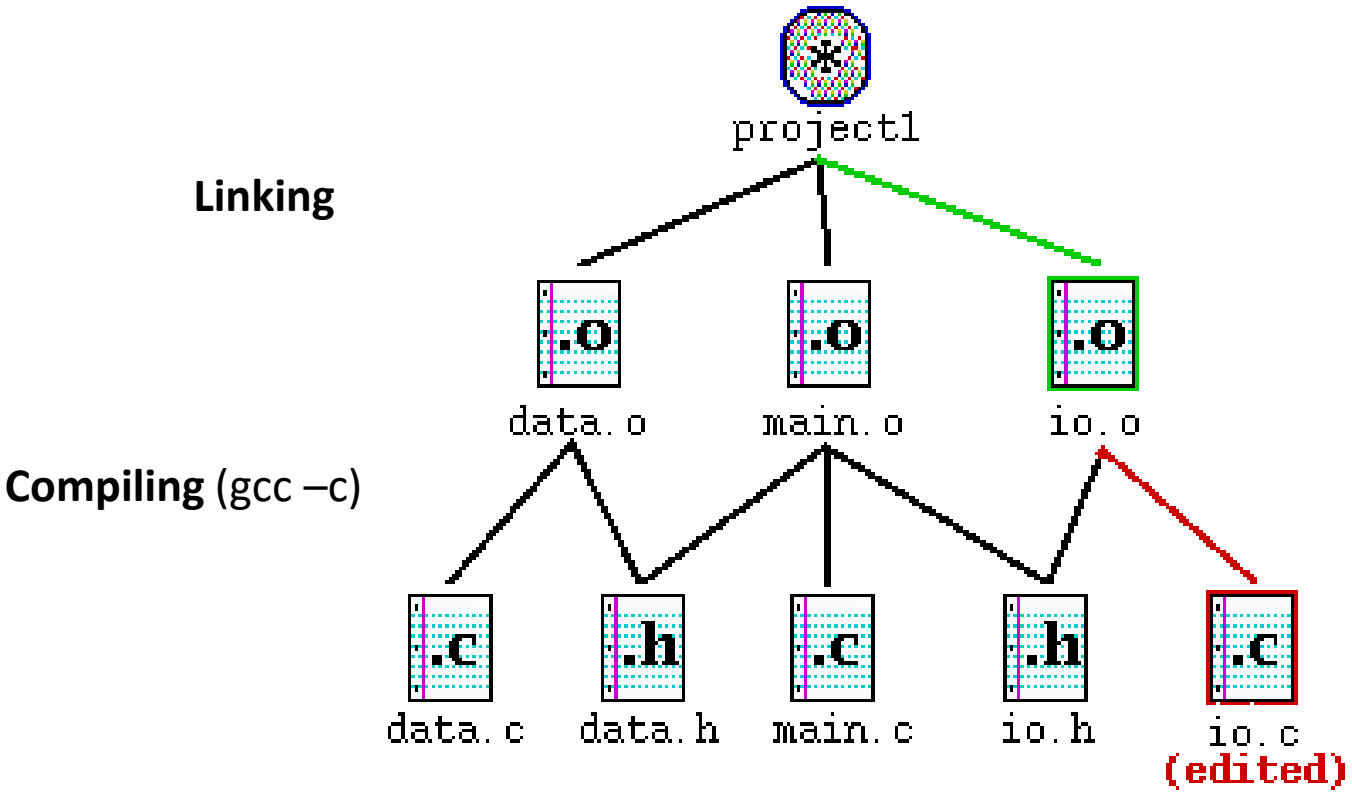
Outline

- Motivation
- gcc
- make and Makefile
- Useful commands
- Project 1 Q&A

Motivation

- Single-file programs do not work well when code gets large
  - compilation can be slow
  - hard to collaborate between multiple programmers
  - more cumbersome to edit
- Larger programs are split into multiple files
  - each file represents a partial program or module
  - modules can be compiled separately or together
  - a module can be shared between multiple programs
- But now we have to deal with all these files just to build our program…

Simple project example



Simple gcc

If we have files:

- prog.c: The main program file, include lib.h and call the function in lib.c
- lib.c: Library .c file
- lib.h: Library header file

% gcc -c prog.c -o prog.o

% gcc -c lib.c -o lib.o

% gcc lib.o prog.o -o binary

gcc flags

- Useful flags
  1. -g: debugging hook
  2. -Wall: all warning
  3. -Werror: treat warning as errors
  4. -O2, -O3: optimization
  5. -DDEBUG: macro for DEBUG (#define DEBUG)

Examples

```
% gcc -g -Wall -Werror -c prog.c -o prog.o
% gcc -g -Wall -Werror -c lib.c -o lib.o
% gcc -g -Wall -Werror lib.o prog.o -o binary
```

But Don’t Repeat Yourself!

Makefile

```
% gcc -g -Wall -Werror -c prog.c -o prog.o
% gcc -g -Wall -Werror -c lib.c -o lib.o
% gcc -g -Wall -Werror lib.o prog.o -o binary
```

```
CC = gcc
CFLAGS = -g -Wall -Werror
OUTPUT = binary
```

Makefile

target: dependency1 dependency2 ...

    unix command (start line with TAB)

    unix command

    ...

% gcc lib.o prog.o -o binary

In the file of “makefile”

binary: lib.o prog.o

    gcc lib.o prog.o -o binary

```
binary: lib.o prog.o
	gcc -g -Wall lib.o prog.o -o binary

lib.o: lib.c
	gcc -g -Wall -c lib.c -o lib.o

prog.o: prog.c
	gcc -g -Wall -c prog.c -o prog.o

clean:
	rm *.o binary
```

```
binary: lib.o prog.o
	gcc -g -Wall lib.o prog.o -o binary

lib.o: lib.c
	gcc -g -Wall -c lib.c -o lib.o

prog.o: prog.c
	gcc -g -Wall -c prog.c -o prog.o

clean:
	rm *.o binary
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary

$(OUTPUT): lib.o prog.o
	$(CC) $(CFLAGS) lib.o prog.o -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary

$(OUTPUT): lib.o prog.o
	$(CC) $(CFLAGS) lib.o prog.o -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

lib.o: lib.c
	$(CC) $(CFLAGS) -c lib.c -o lib.o

prog.o: prog.c
	$(CC) $(CFLAGS) -c prog.c -o prog.o

clean:
	rm *.o $(OUTPUT)
```

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

%.o: %.c
	注释# $<: dependency (%.c)
	注释# $@: target (%.o)
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm *.o $(OUTPUT)
```

$@ to represent the full target name of the current target

$? returns the dependencies that are newer than the current target

$* returns the text that corresponds to % in the target

$< returns the name of the first dependency

$^ returns the names of all the dependencies with space as the delimiter

$% The target member name, when the target is an archive member

Reference:

https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

```
CC = gcc
CFLAGS = -g -Wall
OUTPUT = binary
OBJFILES = lib.o prog.o

all: $(OUTPUT)  test

$(OUTPUT): $(OBJFILES)
	$(CC) $(CFLAGS) $(OBJFILES) -o binary

%.o: %.c
	# $<: dependencies (%.c)
	# $@: target (%.o)
	$(CC) $(CFLAGS) -c $< -o $@

test: $(OUTPUT)
	sh ./testscript.sh

clean:
	rm *.o $(OUTPUT)
```

$@ to represent the full target name of the current target

$? returns the dependencies that are newer than the current target

$* returns the text that corresponds to % in the target

$< returns the name of the first dependency

$^ returns the names of all the dependencies with space as the delimiter

Use Makefile

% make

% make test

% make clean

Google

- “makefile example”
- “makefile template”
- “make tutorial”

Useful Unix Commands

- find “func_name” in files

  % grep -r func_name .

- replace “bad_func_name” to “good_func_name”

  % sed -e “s/bad_func_name/good_func_name/g”\

  prog.c > prog.c.new

Useful Unix Commands

- find a file named “prog.c”

  % find -name prog.c

- download files from Internet

  % wget http://address/to/file.tar.gz

- untar and unzip the file

  % tar xzvf file.tar.gz

What about java or python project

- Similar idea to Make
- Ant/Maven/bazel use a build file instead of a Makefile
    - <project>
      <target name="name">
      tasks
      </target>
      <target name="name">
      tasks
      </target>
    - </project>
- Tasks can be things like:
    - <javac … />
    - <mkdir … />
    - <delete … />

Example build.xml file

```
<!-- Example build.xml file -->
<project>
  <target name="clean">
    <delete dir="build"/>
  </target>
  <target name="compile">
    <mkdir dir="build/classes"/>
    <javac srcdir="src" destdir="build/classes"/>
  </target>
</project>
```