



Development: gcc and make

CS 2204

Class meeting 7



Overview of development process

- Creation of source files (.c, .h, .cpp)
- Compilation (e.g. *.c → *.o) and linking
- Running and testing programs



Development tools in UNIX

- Creation of source files (.c, .h, .cpp)
 - Text editors (e.g. `vi`)
 - Revision control systems (e.g. `rcs`)
- Compilation (e.g. `*.c → *.o`) and linking
 - Compilers (e.g. `gcc`)
 - Automatic building tools (e.g. `make`)
- Running and testing programs
 - Debuggers (e.g. `gdb`)
- Integrated development environments (IDEs)



Compiling with gcc

- GNU C compiler
- Performs one or more of the following:
 - C pre-processing
 - Compilation
 - Linking



Basic gcc examples

- `gcc hello.c` (compile `hello.c`, produce executable `a.out`)
- `gcc -o hello hello.c` (compile `hello.c`, produce executable `hello`)
- `gcc -o hello hello.c other_functions.c` (compile `hello.c` and `other_functions.c`, produce executable `hello`)



Using intermediate files

- From any source file, you can produce an object file to be linked in later to an executable

```
gcc -c hello.c
```

```
gcc -c other_functions.c
```

```
gcc -o hello hello.o other_functions.o
```



Other important `gcc` options

- `-g`: include debugging symbols in the output
- `-l<name>`: include a library called `libname.a`



Include and library paths

- There are default directories in which gcc looks for include files and libraries
- `-I<path>`: also look for include files in this directory
- `-L<path>`: also look for library files in this directory



Defines in gcc

- Often programs contain conditional parts based on defines:

```
#ifdef DEBUG
printf("value of var is %d", var);
#endif
```

- You can set preprocessor defines on the command line

```
gcc -DDEBUG -o prog prog.c
```



Using `make` in compilation

- With medium to large software projects containing many files, it's difficult to:
 - Type commands to compile all the files correctly each time
 - Keep track of which files have been changed
 - Keep track of files' dependencies on other files
- The `make` utility automates this process



Basic operation of make

- Reads a file called [Mm]akefile, which contains rules for building a “target”
- If the target depends on a file, then that file is built
- If that file depends on a third file, then the third file is built, and so on...
- Works backward through the chain of dependencies
- Targets only built if they are older than the files they depend on



Basic Makefile example

```
program : main.o iodat.o dorun.o
    gcc -o program main.o iodat.o dorun.o
main.o : main.c
    gcc -c main.c
iodat.o : iodat.c
    gcc -c iodat.c
dorun.o : dorun.c
    gcc -c dorun.c
```



Types of lines in Makefiles

- Dependency or rules lines
- Commands
- Macro assignments
- Comments



Dependency/rules lines

- Specify a target and a list of prerequisites (optional) for that target

```
target : prereq1 prereq2 prereq3 ...
```



Command lines

- Follow dependency lines
- MUST start with a TAB!
- Any command that can be run in the shell can be placed here

```
target : prereq1 prereq2
        command1
        command2
```

- Special variables in commands:
 - `$$` represents the target
 - `$$?` represents prereqs that are newer than target



Macro (variable) assignments

- You can use macros to represent other text in a Makefile
 - Saves typing
 - Allows you to easily change the action of the Makefile
- Assignment:
`MACRONAME = macro value`
- Usage: `#{MACRONAME}`



Comments and other Makefile notes

- Comments begin with a '#'
- Can be placed at the beginning of a line or after a non-comment line
- Lines that are too long can be continued on the next line by placing a '\' at the end of the first line



Invoking make

- Be sure that your description file:
 - is called makefile or Makefile
 - is in the directory with the source files
- `make` (builds the first target in the file)
- `make target(s)` (builds target(s))
- Important options:
 - `-n`: don't run the commands, just list them
 - `-f file`: use file instead of [Mm]akefile



Basic Makefile example

```
program : main.o iodat.o dorun.o
    gcc -o program main.o iodat.o dorun.o
main.o : main.c
    gcc -c main.c
iodat.o : iodat.c
    gcc -c iodat.c
dorun.o : dorun.c
    gcc -c dorun.c
```



Simplifying the example Makefile with macros

```
OBJS = main.o iodat.o dorun.o
CC = /usr/bin/gcc
program : ${OBJS}
    ${CC} -o $@ ${OBJS}
main.o : main.c
    ${CC} -c $?
iodat.o : iodat.c
    ${CC} -c $?
dorun.o : dorun.c
    ${CC} -c $?
```



Suffix rules

- It's still tedious to specifically tell `make` how to build each `.o` file from a `.c` file
- Suffix rules can be used to generalize such situations
- A default suffix rule turns `.c` files into `.o` files by running the command:
$$\$ \{ CC \} \ \$ \{ CFLAGS \} \ -c \ \$ <$$
- `$(C)` refers to the prerequisite (`file.c`)



Simplifying the example Makefile again

```
OBJS = main.o iodat.o dorun.o
```

```
CC = /usr/bin/gcc
```

```
program : ${OBJS}
```

```
    ${CC} -o $@ ${OBJS}
```



Other useful Makefile tips

- Include a way to clean up your mess

clean:

```
/bin/rm -f *.o core
```

- Include a target to build multiple programs

all:

```
program1 program2 program3
```