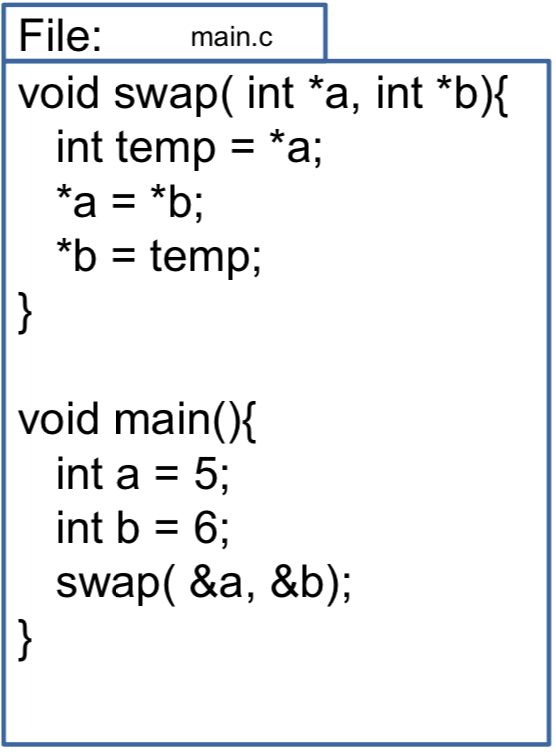
COMP206-Multi-file C programs & Make

To only pre-compile:

$ gcc –E macro\_debugging.c -o macro\_debugging.i

To see the assembly code (extra flags to be more readable):

$ gcc -S -fverbose-asm -g -O2 macro\_debugging.c -o macro\_debugging.s

To see the “object” file:

$ gcc -c -g macro\_debugging.c -o macro\_debugging.o

Compile with “gcc main.c”

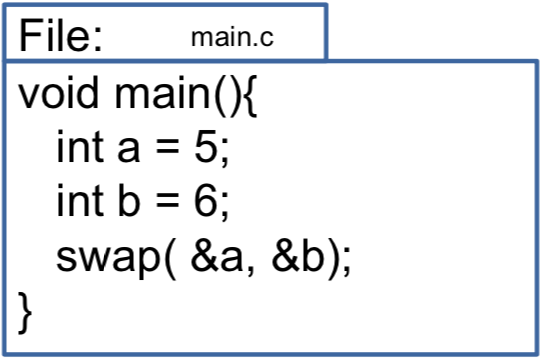
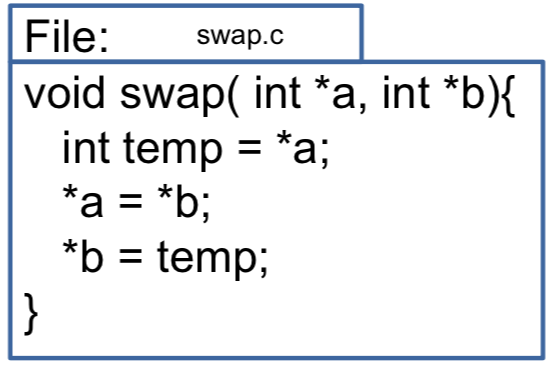
Pros:

It always works, it swaps the values, all is well

Cons:

As we continue to add functions, file gets large.

All code builds every time, even if only one function changes

Spilt C program

Try 1: Compile only the main

FAILS

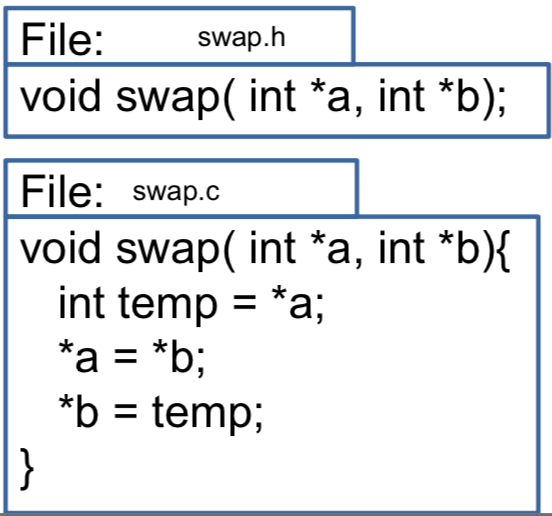
Try 2: List both main and swap with

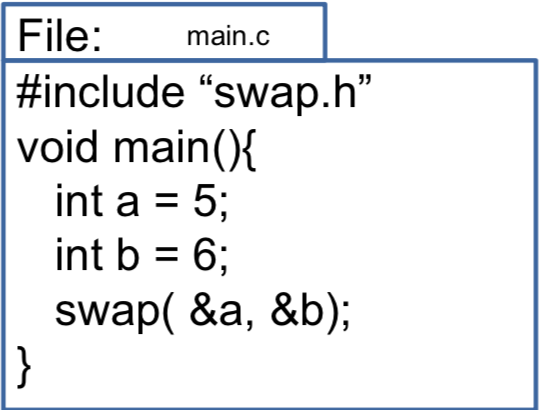
“gcc main.c swap.c”

Warning about implicit declaration,

but does compile

Fix requires telling C how to find all required functionality before it needs to use it



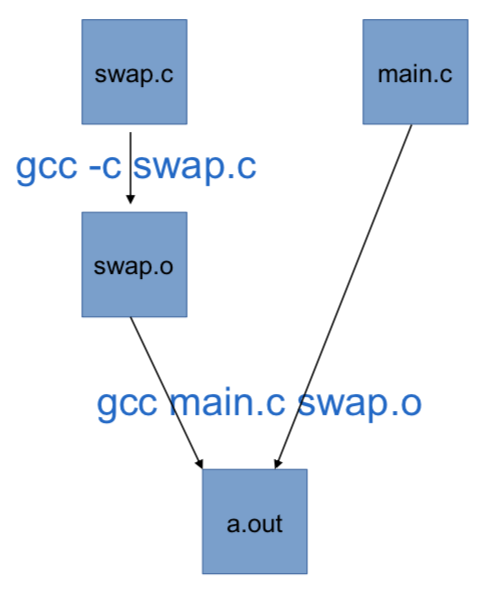
Simple Multi-file Compilation

Create a header (.h) file for each C file

#include it in dependent files

List all “.c” files for gcc

**Objects & Compiling**

When we list a C file for gcc, it parses the code, checks types, optimizes etc.

If we use the same library repeatedly, it’s a big waste of time

As we saw, the first stages of compilation allow us to save this time by creating **“object”** files(.o) that store the temporary result.

− gcc-cswap.c->producesswap.o

Objects allow us to throw-away the C file

CAUTION: This means changes to swap.c are not used unless we explicitly re-create the object

Make

*make is a utility which allows us to specify dependencies, and to rebuild only the necessary files according to the dependencies and modification times.*

In order to use make, we place all of our macro definitions, dependencies, commands, and targets into a file which must be called **Makefile**

We then run make with a target (default is all)

make

make all

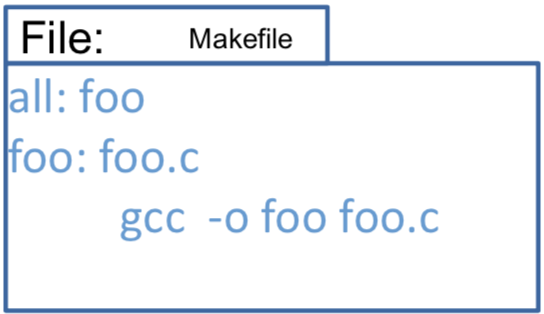
make clean

make install

Format:

target: dependencies

commands



Make checks all targets:

If the target filename doesn’t exist or it exists but dependencies are newer:

* Recursively build any dependencies that are also listed as targets, using the same logic
* Execute the commands listed
* Note this logic means that it’s important now to use “-o” flag for gcc (output to a different filename than a.out

Else (the filename exists and is newer than all deps):

* Nothing to do for this target (we saved wasted effort!)



Macros (similar to shell variable)

macros are specified in *make* : name=text\_string

macro expansion: $(name) OR ${name}

example: SRC=foo.c

${SRC}

[Common Macros]

SRCS=foo.c bar.c

CFLAGS=-Wall -ansi

LDFLAGS=-lm -lmylib

INCDIR=-I/home/ericb/include

LIBDIR=-L/home/ericb/lib

Example command in *make*:

gcc ${CFLAGS} ${INCDIR} -o foo ${SRCS}\ ${LIBDIR} ${LDFLAGS}

[Macro String Substitution]

*make* has a powerful string substitution operator for macros:

SRCS=defs.c redraw.c calc.c

OBJS=${SRCS:.c=.o}

Same as:

OBJS=defs.o redraw.o calc.o

[Suffix Rules]

suffix rules tell make how files are inter-dependent:

.c.o:

${CC} ${CFLAGS} ${INCDIR} -c $<

The above tells make how to create any needed ".o"

file from its matching ".c" file.

* $< is set to the current dependency
* recall that -c to gcc means to compileonly,not to link (i.e., to produce a .o file)

NOTE: For 206 this is the only suffix rule we need you to know. Just memorize it, you don’t have to apply to new cases

