COMP 206 – Introduction to Software Systems

Lecture 14 –Multi-file C Programs and Make

October 22nd, 2018

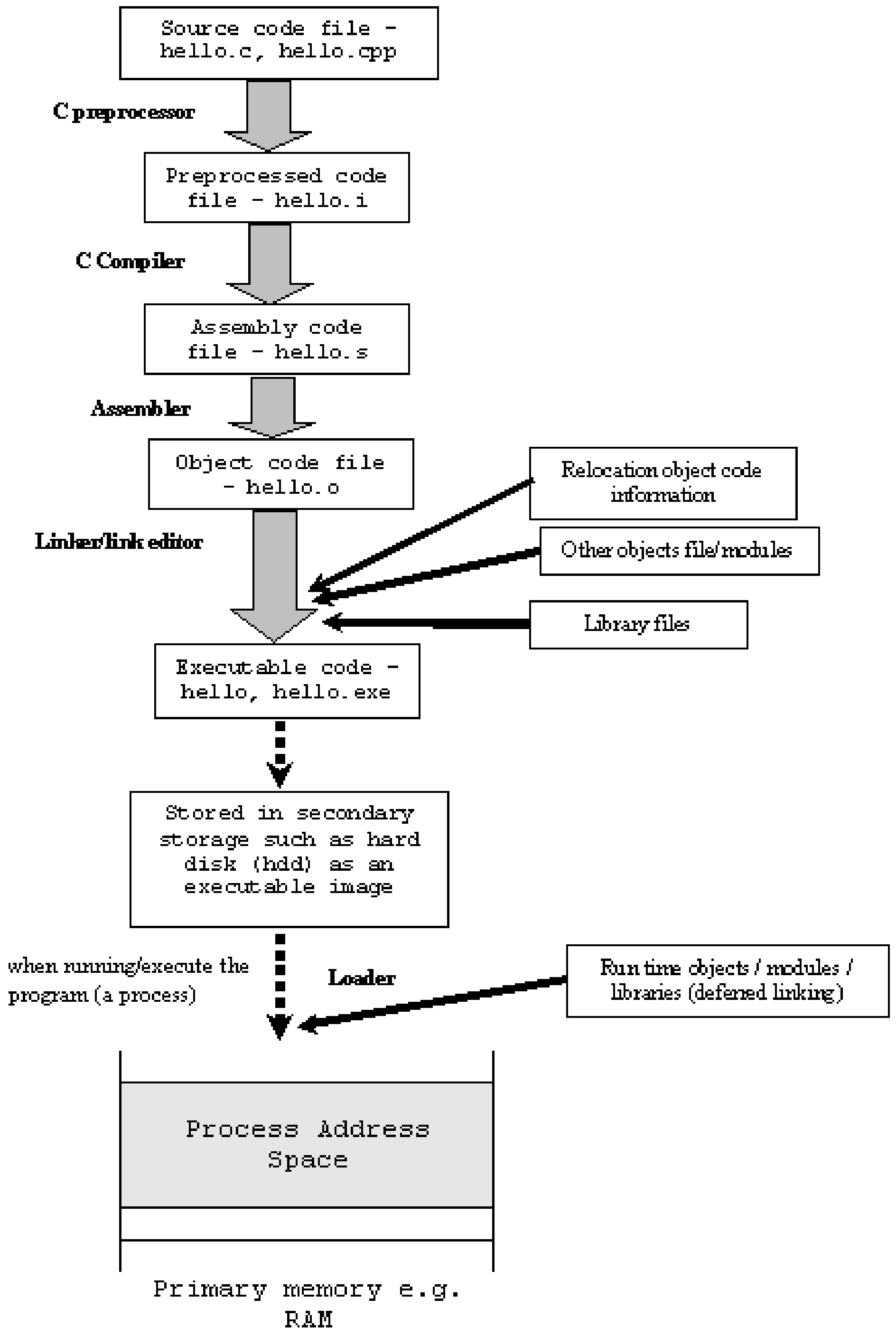
Today’s outline

- Building C code from multiple files

- The make tool and Makefiles

Compiling Aint Easy

- In the next weeks, we will begin to understand much more about the steps involved in gcc turning our C code into a program
- Today, the basics: what if our own C code is split into more than one file?


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Linker/link editor
Relocation object code information
Other objects file/modules
Library files
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

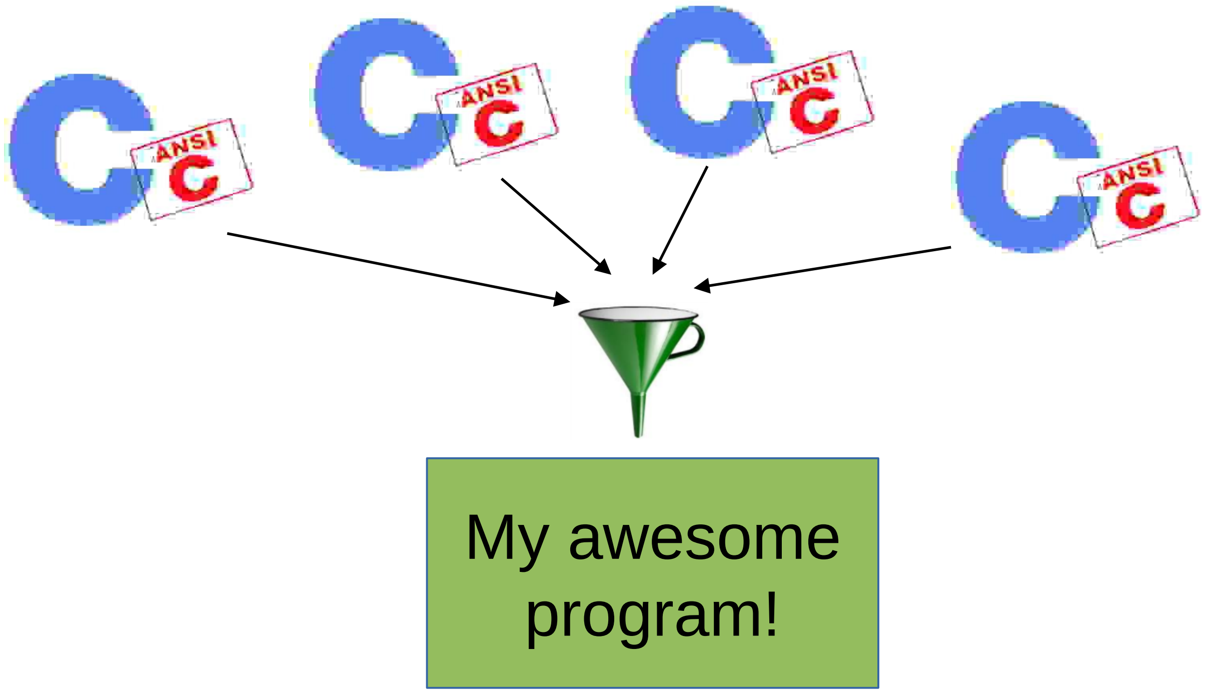
You can see these steps in action

- 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

Creating and Managing Larger C Programs



Multiple-file Projects and Libraries

- As C programs grow, desirable to break your own code into multiple C files to stay organized
- We may also use code written by others, even without getting the C source

- Today:
  - How this can be done with gcc
  - How this fits with what we know about programs
  - First tools that helps work with C projects: make

Start from this C program

- 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

File: main.c

```
void swap( int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}

void main(){
   int a = 5;
   int b = 6;
   swap( &a, &b);
}
```

Suppose you split your C program...

- Try 1: Still try to compile with “gcc main.c”
  - FAILS
  - Of course, since we have not told gcc anything about the swap function!

File: swap.c

```
void swap( int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

File: main.c

```
void main(){
   int a = 5;
   int b = 6;
   swap( &a, &b);
}
```

Suppose you split your 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

File: swap.c

```
void swap( int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

File: main.c

```
void main(){
   int a = 5;
   int b = 6;
   swap( &a, &b);
}
```

Suppose you split your C program...

- Try 1: Compile only the main -> FAILS
- Try 2: List both main and swap for gcc -> Warning
- Fix requires telling C how to find all required functionality **before** it needs to use it

File: swap.c

```
void swap( int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

File: main.c

```
void main(){
   int a = 5;
   int b = 6;
   swap( &a, &b);
}
```

Simple Multi-file Compilation

- Create a header (.h) file for each C file
- #include it in dependent files
- List all “.c” files for gcc

File: swap.h

```
void swap( int *a, int *b);
```

File: main.c

```
#include “swap.h”
void main(){
   int a = 5;
   int b = 6;
   swap( &a, &b);
}
```

File: swap.c

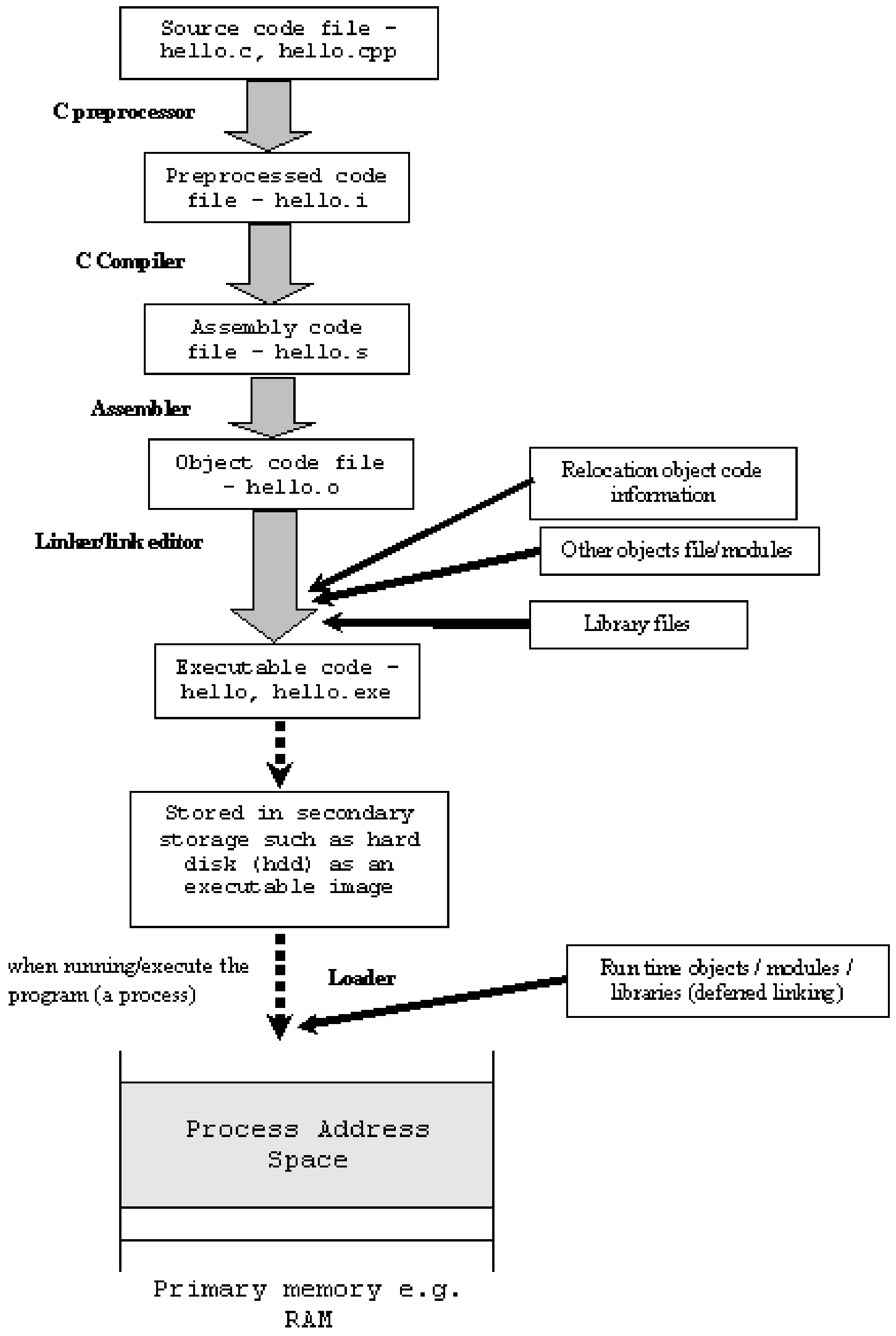
```
void swap( int *a, int *b){
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

Objects and Compiling vs Linking

- When we list a C file for gcc, it parses the code, checks types, optimizes etc.
- If we use the same (perhaps complex) library repeatedly, this is 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 -c swap.c -> produces swap.o
- A linker combines objects to form a program (or library, as we will soon see)

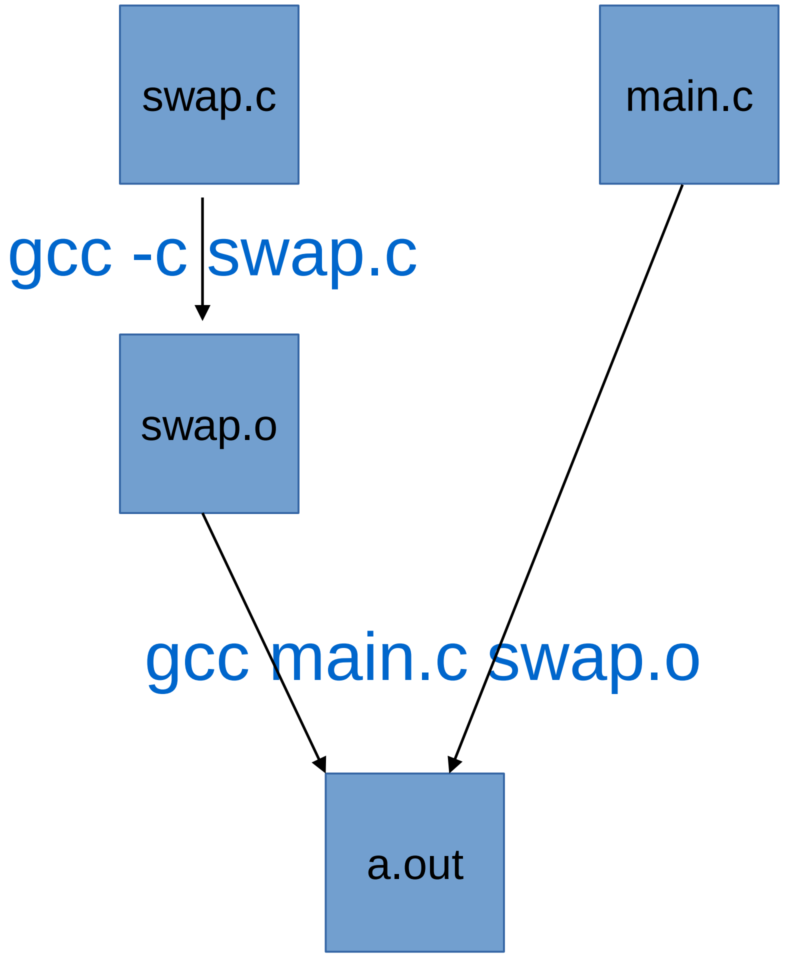
Recall: the C Program Creation Pipeline

- A common use-case is not to go all the way through this process
  - Stop at .o file: you have done most of the work
  - Code is easily re-usable by others… if you wrote a good header ".h"!


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Linker/link editor
Relocation object code information
Other objects file/modules
Library files
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

Objects and Compiling vs Linking

- 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



Exercise

- This is not meant to be a deep concept at all, but if it’s your first time creating a program with many files, it can be hard to relate

- Do try it! Follow the recipe:
    - Make a single main.c program with many functions, ensure it works
    - Cut/paste the functions into separate .c files
    - Create a .h header for each that only includes the spec, not the body
    - Compile them all together with one gcc command, ensure it works
    - Compile one by one into .o files and then create a.out using just main.c listed with the .o for every other function

`make`
a specific software tool

Reference book (not required):

Managing Projects with `make`

Andrew Oram and Steve Talbott

O'Reilly & Associates

Why `make`?

- When a project contains many source files, it can be very time consuming to compile all of the source files & error prone.

- We would like to re-compile only those files which have changed.

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

Makefile

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
```

Our first Makefile

- Format:

  target: dependencies

      commands

File: Makefile

```
all: foo
foo: foo.c
	gcc  -o foo foo.c
```

Our first Makefile

- 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!)

File: Makefile

```
all: foo
foo: foo.c
	gcc  -o foo foo.c
```

Our second Makefile

- Same logic applies. Make sure you can work this through:

File: Makefile

```
all: foo bar
foo: foo.c
    gcc  -o foo foo.c
bar: bar.c
    gcc  -o bar bar.c
clean:
    rm foo bar
```

Macros (similar to shell variable)

macros are specified in `make` as follows:

```
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 compile only, 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

Sample Complete Makefile

```
SRCS=foo.c bar.c
OBJS=foo.o bar.o barbar.o
CFLAGS=-Wall -ansi
LDFLAGS=-lm -lmylib
INCDIR=-I/home/ericb/include
LIBDIR=-L/home/ericb/lib

all: ${OBJS}
   ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
foo: ${OBJS}
   ${CC} -o foo ${OBJS} ${LIBDIR} ${LDFLAGS}
clean:
    /bin/rm -f ${OBJS}
install:  foo
    /bin/cp -f foo /usr/local/bin

.c.o:
    ${CC} ${CFLAGS} ${INCDIR} -c $<
```

Practice

- Look over the Makefile for A3, ensure you use it in your workflow and know what’s going on there

- Try to write you own Makefiles from scratch:
  - First simple with just 2-3 files, type make and ensure the files build, change the files, make again and confirm it all makes sense
  - Try to write one with a macro and a suffix rule: this will prepare you for the final (and it’s a good life skill!)