# Week 6 Lecture 1: Compiling!

# Review and Preview

- Functions
- Assignment 2
- Midterm

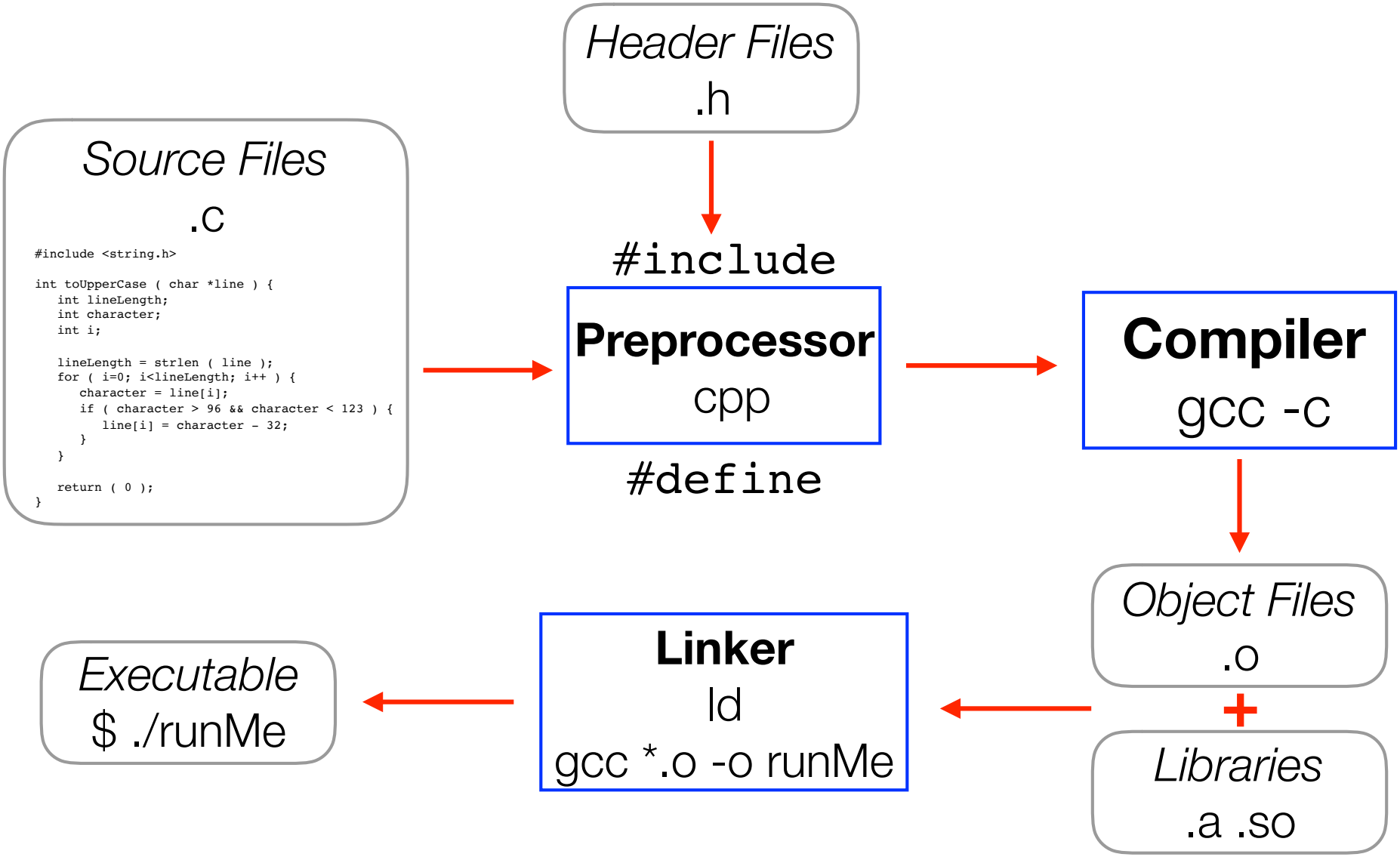
---

- Compiling and Linking
- Compiler Options
- make

**K&R: Chapters 1-5**
**Sections 4.5, 4.11**
**Sections 5.1-5.5**

# Compiling and Linking

What has been going on when you use gcc?




# Compilation and Linking

# Compiling and Linking: The Preprocessor

- The preprocessor is a separate program (`cpp`) that is invoked automatically by the compiler before compilation begins.
- Your source code file is written into an expanded source code file directed by the preprocessor commands.
- Preprocessor **directives** start with the pound sign (#).
- You already know the most important ones: `#include` and `#define`.

# Preprocessor Directives

- `#include` - paste code of a given file into the current file
- `#define` - defines a macro which is a segment of code which is replaced by the value of macro
- `#undef` - cancel the definition of a macro
- `#ifdef` - checks if a macro is defined by `#define`. If yes, it executes the code.
- `#ifndef` - checks if a macro is not defined by `#define`. If yes, it executes the code.

# #define

```
#include <stdio.h>
#define PI 3.141592653589793
int main() {
   printf("%16.15f\n",PI);
}
```

```
$ ./preproc1
3.141592653589793
```

# #undef

```
#include <stdio.h>
#define PI 3.141592653589793
int main() {
   printf("%16.15f\n",PI);
#undef PI
   printf("%16.15f\n",PI);
}
```

```
$ gcc -ansi preproc2.c -o preproc2
preproc2.c:6:23: error: use of undeclared identifier
'PI'
   printf("%16.15f\n",PI);
                      ^
1 error generated.
```

# #ifdef, #ifndef, #endif

```
#include <stdio.h>
#define PI 3.141592653589793
int main() {
#ifdef PI
   printf( "%16.15f\n", PI );
#endif

#ifndef PLANCK
  printf  ( "No Planck's Constant\n" );
#endif
}
```

```
$ ./preproc3
3.141592653589793
No Planck's Constant
```

# Preprocessor Directives

- `#if` - evaluates an expression or condition. If it is true, it executes the code.
- `#else` - if condition of #if is false the the #else code is executed. It can be used with #if, #elif, #ifdef and #ifndef.
- `#elif` - else if construct
- `#endif` - ends an if-else set of statements

# #if, #else, #endif

```
#include <stdio.h>
#define PI 3.141592653589793
#define PFLAG 1
int main() {
#if PFLAG == 1
   printf("Value of pi is: %8.7f\n",PI);
#else
   printf("Value of pi is undefined");
#endif
}
```

```
$ ./preproc4
Value of pi is: 3.1415927
```

# #if, #else, #endif

```
#include <stdio.h>
#define PI 3.141592653589793
#define PFLAG 0
int main() {
#if PFLAG == 1
   printf("Value of pi is: %8.7f\n",PI);
#else
   printf("Value of pi is undefined\n");
#endif
}
```

```
$ ./preproc5
Value of pi is undefined
```

# Preprocessor Directives

- `#error` - causes the preprocessor to report a fatal error. The rest of the line following `#error` are used as the error message.

- `#warning` - causes the preprocessor to report a warning.

- `#pragma` - used to provide additional information to the compiler. The `#pragma` directive is used by the compiler to offer machine or operating-system features.

# #error

```
#include <stdio.h>
#define __OSX 1
int main() {
#ifdef __OSX
#error "Will not work on OSX.  See
documentation.”
#endif
}
```

```
$ gcc -ansi preproc6.c -o preproc6
preproc6.c:7:2: error: "Will not work on OSX. See
documentation."
#error "Will not work on OSX. See documentation."
 ^
1 error generated.
```

# #warning

```
#include <stdio.h>
#define __OSX 1
int main() {
#ifdef __OSX
#warning "Will not work well on OSX.  See documentation."
#endif
printf ( "Just a warning\n" );
}
```

```
$ gcc -ansi preproc7.c -o preproc7
preproc7.c:7:2: warning: "Will not work well on OSX.  See
documentation. [-W#warnings]
#warning "Will not work well on OSX.  See documentation.
 ^
1 warning generated.
$ ./preproc7
Just a warning
```

# #pragma

```
#include <stdio.h>
#pragma GCC poison atoi
int main ( int argc, char *argv[] ) {
    int num;
    if ( argc > 1 ) {
        num = atoi ( argv[1] );
    }
}
```

```
$ gcc -ansi preproc8.c -o preproc8
preproc8.c:8:13: error: attempt to use a poisoned
identifier
        num = atoi ( argv[1] );
              ^
1 error generated.
```

# The Compiler

- The compiler converts the C source code into object code and puts it into a file ending in ".o"
- An object files contains the binary version of the source code.
- Object code is not directly executable.
- To be executable, you have to add code for all of the library functions used by the program.

# The Linker

- The linker *links* together object files (`.o` files) into a binary executable.
- This includes
  - object files created from your source code files
  - object files that have been pre-compiled and collected into library files. These files have names ending in `.a` or `.so`.

# File Types

- Source code files (`.c`) contain function definitions.
- Header files (`.h`) contain function declarations and preprocessor statements. They connect the source code files to externally-defined functions.
- Object files (.o) are produced as the output of the compiler. They are not executable by themselves.
- Binary executables are produced as the output of the linker. They can be directly executed.

# Compiler Options

What else is on the gcc command line?

# Compiler Options

- Specify the Output Executable Name

```
$ gcc main.c -o main
```

- Enable all warnings

```
$ gcc -Wall main.c -o main
```

- Produce only the compiled code

```
$ gcc -c main.c
```

# Compiler Options

- Link with shared libraries

```
$ gcc -Wall main.c -o main -lm
```

- Enable the support of ISO C89 style

```
$ gcc -Wall -ansi main.c -o main
```

# make

Can we make this easier?

# `make`

- make is a UNIX tool to simplify building program executables from many modules.
- make reads in rules (specified as a list of target entries) from a Makefile.
- make will only re-build things that need to be re-built (object or executables that depend on files that have been modified since the last time the objects or executables were built).

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html#C

# Creating a Makefile

- A **Makefile** consists of variable definitions, target entries for building specific targets (typically .o & executable files) or executing a set of commands associated with a target label.
- The following is the generic target entry form:

```
# The <tab> in the command line is necessary

target:  dependencies/prerequisites (.c,.h)

        <tab> recipe/commands
```

# Creating a Makefile

```
#
# target to build executable from main
# and myfunc object files
#
main: main.o myfunc.o
   gcc -Wall -ansi -o main main.o myfunc.o

main.o main.c
   gcc -c -Wall -ansi main.c

myfunc.o myfunc.c
   gcc -c -Wall -ansi myfunc.c
```

# Creating a Makefile

```
# the compiler: gcc for C programs
CC = gcc
# compiler flags:
#  -Wall -ansi
CFLAGS = -ansi -Wall
main: main.c
    $(CC) $(CFLAGS) -o main main.c
    gcc -ansi -Wall -o main main.c
```

# Using `make`

- If you have a **Makefile** then you can build the executables (targets) by typing:

```
$ make
```

- A specific target in the **Makefile** can be built by typing:

```
$ make target_label
$ make findWords
```

https://opensource.com/article/18/8/what-how-makefile

# Makefile

```
textRW: textRW.c
	gcc -ansi -Wall -o textRW textRW.c

testConvert: testConvert.o convertLowerCase.o
	gcc -ansi -Wall -o testConvert \
testConvert.o convertLowerCase.o

testConvert.o: testConvert.c
	gcc -ansi -Wall -c testConvert.c

convertLowerCase.o: convertLowerCase.c
	gcc -ansi -Wall -c convertLowerCase.c
```

```
findWords: findWords.o chop.o convertLowerCase.o \
replaceDigits.o replacePunc.o reduceSpace.o trim.o
        gcc -ansi -Wall -o findWords findWords.o chop.o \
convertLowerCase.o replaceDigits.o replacePunc.o \
reduceSpace.o trim.o

findWords.o: findWords.c
        gcc -ansi -Wall -c findWords.c

chop.o: chop.c
        gcc -ansi -Wall -c chop.c

replaceDigits.o: replaceDigits.c
        gcc -ansi -Wall -c replaceDigits.c

replacePunc.o: replacePunc.c
        gcc -ansi -Wall -c replacePunc.c

reduceSpace.o: reduceSpace.c
        gcc -ansi -Wall -c reduceSpace.c

trim.o: trim.c
        gcc -ansi -Wall -c trim.c
```