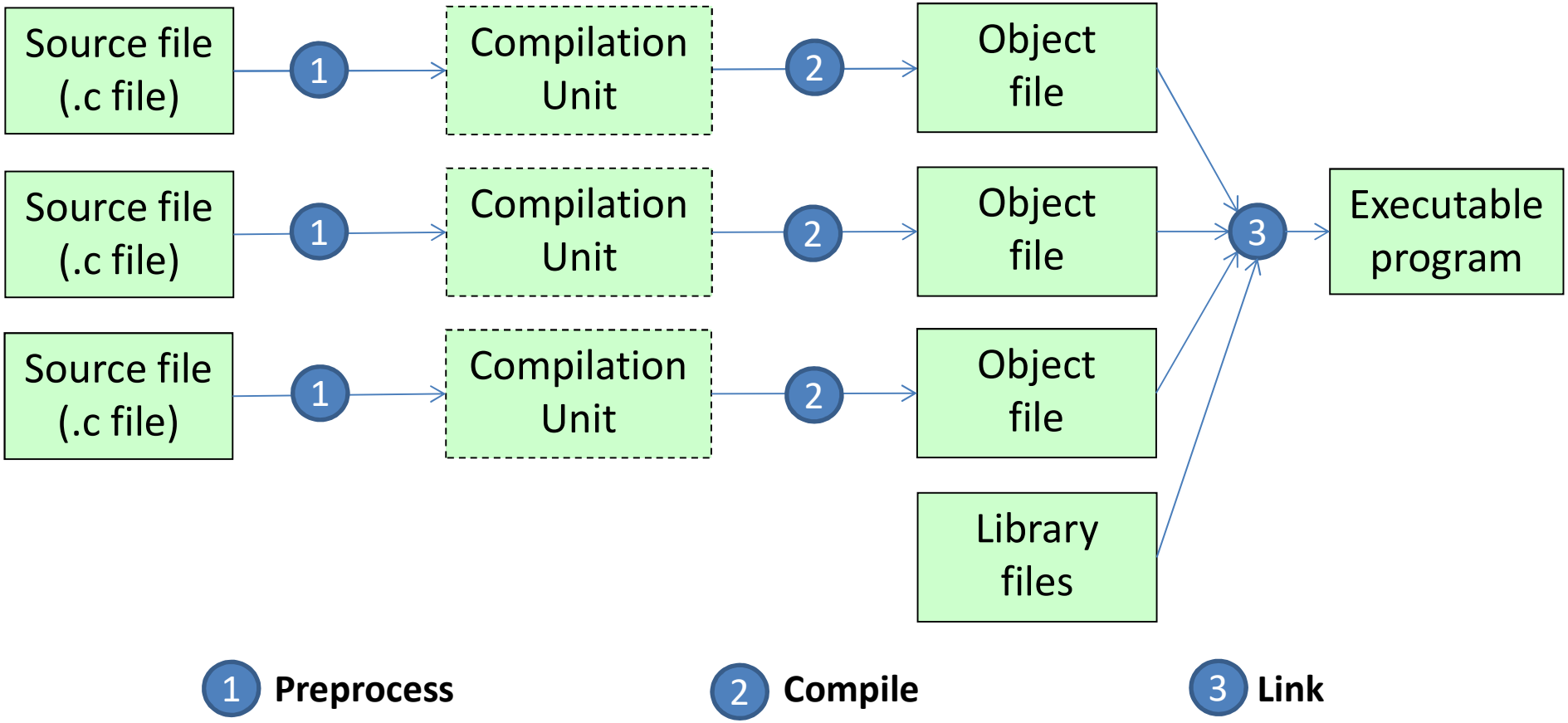
# Building a program from multiple source files

**Pay Attention**: You have to be able to do multi-file compilation for the project.
I will first talk about the concepts and finally do a demo.

# Outline

1. Understand how C compilation works
2. Preprocessor directives
3. Why organizing code into multiple files?
4. How to organize code into multiple files?

# 1. What happen when you "build" a program?

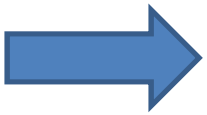



**Note**: In lab exercises, you only have one .c file. However, in a typical software project, the code is usually organized into multiple files.

# Step (1) Preprocess

your_program.c

```
#include <stdio.h>

#define PI 3.1416

int main()
{
  printf( "%lf" , PI );
  ...
}
```



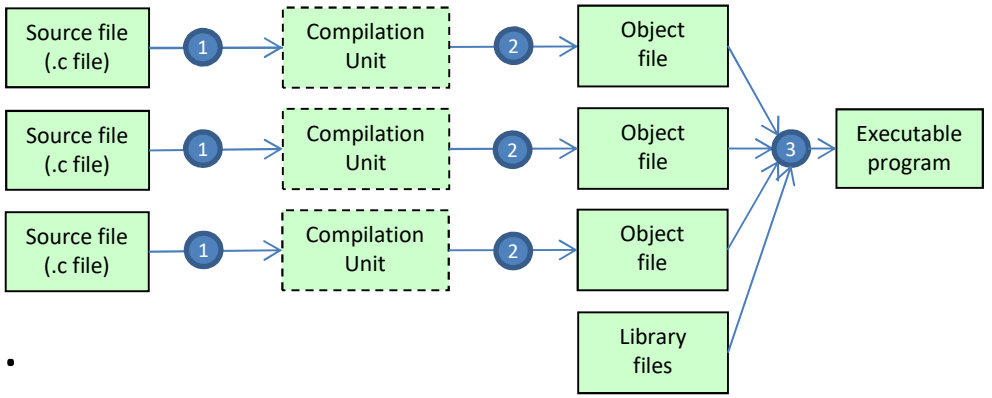
Compilation unit

```
File Content of
"stdio.h" inserted here


int main()
{
  printf( "%lf" , 3.1416 );
  ...
}
```

- A simple, fast but very useful step, before compilation
- The *compiler* goes through the code once, processes all the *preprocessor directives* (lines begin with **#...**) and produces an "intermediate C source file" called the *compilation unit*.

# Step (2) Compile & Step (3) Link

- Compile
  - The *compiler* further translates the compilation unit into machine code, and stores it in an object file.




- Link
  - A program called *linker* produces an executable program by putting together all functions (in machine-code form) from the related *object files* and *library files*.
    - A library file is a collection of object files, with an index to allow rapid searching (e.g., math library).

**EXTRA CONCEPT**: this is known as **statically-linked library.** There's something **dynamically-linked library**, or **DLL in Microsoft**, e.g., for making plugin, etc.
https://en.wikipedia.org/wiki/Dynamic-link_library

# Outline

1. ~~Understand how C compilation works~~

2. Preprocessor directives

3. Why organizing code into multiple files?

4. How to organize code into multiple files?

# 2. Preprocessor Directives (#include, #define)

- What is Preprocessor directives?
  - Lines included in a C program that **begin with #**, which make them different from a typical text in code.
  - They are consumed by the preprocessor to translate your program code into a **compilation unit**

# Header files use lots of “Preprocessor directives”

Browse the source code of include/math.h

```
18
19  /*
20   *        ISO C99 Standard: 7.12 Mathematics        <math.h>
21   */
22
23  #ifndef        _MATH_H
24  #define        _MATH_H        1
25
26  #define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
27  #include <bits/libc-header-start.h>
28
29  __BEGIN_DECLS
30
31  /* Get definitions of __intmax_t and __uintmax_t.  */
32  #include <bits/types.h>
33
34  /* Get machine-dependent vector math functions declarations.
35  #include <bits/math-vector.h>
36
37  /* Gather machine dependent type support.  */
38  #include <bits/floatn.h>
39
40  /* Get machine-dependent HUGE_VAL value (returned on overflow).
41     On all IEEE754 machines, this is +Infinity.  */
42  #include <bits/huge_val.h>
43
44  #if __HAVE_FLOAT128 && __GLIBC_USE (IEC_60559_TYPES_EXT)
45  # include <bits/huge_val_flt128.h>
46  #endif
```

This is JUST “math.h” in C

It’s okay, if you can’t understand it now. But after the next 10-15 slides, you should be able to do so!!!

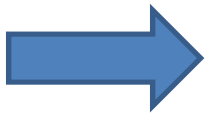
From: https://code.woboq.org/gcc/include/math.h.html

# 2. Preprocessor Directives: #include

- `#include <system_header.h>`
- `#include "user_defined_header.h"`
  - Insert in place “the contents” of the specified file
  - Use <…> to enclose the filename if the file is located in the "**designated system folder**"
  - Use "…" to enclose the filename if the file is in the **same folder** as the .c files with the #include

your_program.c

```
#include <stdio.h>

int main()
{
   ...
```



Compilation unit

```
File Content of
"stdio.h" inserted here

int main()
{
```

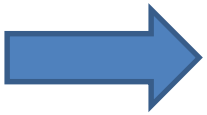
# 2. Preprocessor Directives: #define

- `#define NAME VALUE`
  - Replace all instances of NAME in program code below by VALUE
  - `Usually, NAME is all uppercase`
  - Does not affect string literals, e.g., "PI=%f"

```
#define PI 3.1416
```

your_program.c

```
...

#define PI 3.1416

int main()
{
  printf( "%lf" , PI );
  ...
}
```



Compilation unit

```
...



int main()
{
  printf( "%lf" , 3.1416 );
  ...
}
```

# 2. Preprocessor Directives - macro

- A macro is a **#define** directive that can be parameterized.

```
1  #define PRINT(X)  printf("|%d|",X)
2
3  int main( void )
4  {
5      int number = 10 ;
6      PRINT(number) ;
7      return 0 ;
8  }
```

```
#define P(X)  printf("|%d|", X)

int main( void )
{
    int number = 10 ;
    printf( "|%d|" , number ) ;
    return 0 ;
}
```

Note. the substitution **does not involve the trailing semi-colon**.

# 2. Preprocessor Directives - macro

- More example:

```
1  #define SWAP(A,B)  { int tmp = A ; A = B ; B = tmp ; }
2
3  int main( void )
4  {
5      int a = 10 , b = 20 ;
6      printf( "%d %d\n" , a , b );
7      SWAP( a , b );
8      printf( "%d %d\n" , a , b );
9  }
```

Q1: Can the program perform swapping?

Q2: is the **semi-colon at the end of Line 7** necessary?

Q3: under what circumstances, will the **pair of curly-braces in Line 1** be necessary?

# 2. Preprocessor Directives - macro

- More example:

```
1   #define PRINT(X)   printf( "%s = %d\n" , #X , X )
2
3   int main( void )
4   {
5       int a = 10 , b = 20 ;
6       PRINT(a) ;
7       PRINT(b) ;
8       PRINT(a+b) ;
9       return 0 ;
10  }
```

```
printf("%s = %d", "a", a);
printf("%s = %d", "b", b);
printf("%s = %d", "a+b", a+b);
```

The directive "`#[parameter]`" is super handy if you want to set up **debugging** macro.

This derivative transforms the input parameter **into a string**.

# 2. Preprocessor Directives - macro

- More example:

```
1   #define myMax1(X,Y)    ( X > Y ? X : Y )
2   #define myMax2(X,Y)    ( (X) > (Y) ? (X) : (Y) )
3
4   int main( void )
5   {
6       int a = 10 , b = 20 , c = 30 ;
7       int t1 = myMax1(a,b) ;
8       int t2 = myMax2(a,myMax2(b,c)) ;
9       return 0 ;
10  }
```

Any difference between "**myMax1**" and "**myMax2**"?

Which one is more safe to use?

If we only substitute X and Y by single variables (see line 7) it is fine… but… (next page)

# 2. Preprocessor Directives - macro

- More example:

```
1   #define myMax1(X,Y)    ( X > Y ? X : Y )
2   #define myMax2(X,Y)    ( (X) > (Y) ? (X) : (Y) )
3
4   int main( void )
5   {
6       int a = 10 , b = 20 , c = 30 ;
7       int t1 = myMax1(a,b) ;
8       int t2 = myMax1(a,myMax1(b,c)) ;
9       return 0 ;
10  }
```

```
( a > b ? a : b ) ;
```

```
( a > b > c ? b : c ? a : b > c ? b : c ) ;
```

If we use myMax1 for both Lines 7 and 8… then, see above ☹

# 2. Preprocessor Directives - macro

- More useful directives for debugging:

| Directives | Description |
|---|---|
| __FUNCTION__ | **A string**: the function at which the directive is expanded |
| __LINE__ | **A number**: the line number at which the directive is expanded |
| __FILE__ | A string: the file where the directive is expanded |
| __DATE__ | A string: the date when the source file is compiled |
| __TIME__ | A string: the time when the source file is compiled |

# 2. Preprocessor Directives - macro

- More Examples:

A macro **must be written in one line**.
'\' is the line continuation symbol, asking the compiler to treat the two consecutive line as a single line

```
1   #define DEBUG(X)  printf("%s (%d): %s = %d\n", \
2                       __FUNCTION__, __LINE__, #X, X)
3
4   int main( void )
5   {
6       int a = 10 ;
7       printf( "Filename = %s\n" , __FILE__ ) ;
8       printf( "Compiled:  %s, %s\n" , __TIME__ , __DATE__ ) ;
9       DEBUG(a) ;
10      return 0 ;
11  }
```

```
printf("%s (%d): %s = %d\n", "main", 9, "a", a);
```

# "Conditional Compilation": #ifdef and #endif

```
1   #define DEBUG
2   ...
3
4   // #ifdef means "If defined"
5   int main( void )
6   {
7       // Code to print debug message
8     #ifdef DEBUG
9       printf( "Debug: ... \n" ) ;
10      ...
11    #endif
12  }
```

Mark the name DEBUG as "being defined". It does not need to have a value.

Lines 9-10 are included into the compilation unit **only if** the name DEBUG has been defined earlier using #define (or use debug mode…)

- We can conditionally include/exclude a segment of code in a program in the compilation
- Useful for debugging (you should learn to use debugger during the semester break)

See also https://msdn.microsoft.com/en-us/library/5bb575z2.aspx

# 2. Preprocessor Directives (#ifdef, #endif)

```
1   #define DEBUG
2   #undef DEBUG
3   ...
4
5   int main( void )
6   {
7       // Code to print debug message
8     #ifdef DEBUG
9       printf( "Debug: ... \n" ) ;
10      ...
11    #endif
12  }
```

Mark the name DEBUG as "being undefined". OR you may simply comment out line 1

In this example, if we want to include the debugging code (lines 9-10) into the program, we can simply remove or comment line 2.

Debugger: https://msdn.microsoft.com/en-us/library/sc65sadd.aspx

# 2. Preprocessor Directives (#ifndef)

```
1  // ifndef means "If not defined"
2  #ifndef PI
3  #define PI 3.14159
4  #endif
```

Define PI only if PI has not yet been defined.

This approach prevents a name being redefined by accident.

```
1  #ifndef FOO
2  #define FOO
3
4  // Some code here ...
5
6  #endif
```

The same approach can prevent a segment of code being included twice in a program by accident.

# Let's see "math.h" again

Browse the source code of include/math.h

```
18
19  /*
20   *        ISO C99 Standard: 7.12 Mathematics        <math.h>
21   */
22
23  #ifndef        _MATH_H
24  #define        _MATH_H        1
25
26  #define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
27  #include <bits/libc-header-start.h>
28
29  __BEGIN_DECLS
30
31  /* Get definitions of __intmax_t and __uintmax_t.  */
32  #include <bits/types.h>
33
34  /* Get machine-dependent vector math functions declarations.  */
35  #include <bits/math-vector.h>
36
37  /* Gather machine dependent type support.  */
38  #include <bits/floatn.h>
39
40  /* Get machine-dependent HUGE_VAL value (returned on overflow).
41     On all IEEE754 machines, this is +Infinity.  */
42  #include <bits/huge_val.h>
43
44  #if __HAVE_FLOAT128 && __GLIBC_USE (IEC_60559_TYPES_EXT)
45  # include <bits/huge_val_flt128.h>
46  #endif
```

```
23  #ifndef        _MATH_H
24  #define        _MATH_H        1
```

```
855  #endif /* math.h  */
856
```

Usually, we have #ifndef and #define at the beginning to avoid re-definition

From: https://code.woboq.org/gcc/include/math.h.html

# Any error?

```
1  // Can we include math.h again?
2  #include <math.h>
3  #include <stdio.h>
4  #include <math.h>
5  ...
```

Is there any problem when compiling this piece of code?

```
1  // If math.h is included in mylib.h?
2  #include <math.h>
3  #include <mylib.h>  // your code
4  ...
```

Is there any problem when compiling this piece of code?

```
1  // How about this?
2  #include <mylib.h>
3  #include <mylib2.h>  // has mylib.h
4  ...
```

How to allow this?

# 2. More Preprocessor Directives: “Portability”

```
1  // Mac-only
2  #ifdef __APPLE__
3  ...
4
5  // Linux-only
6  #elif defined __linux__
7  ...
8
9  // Windows-only stuff
10 #elif defined _WIN32 // or _WIN64
11 #include <windows.h>
12 ...
13 #endif
```

Different platforms may have slightly different behavior, e.g., system-related function calls such as reading CPU times

How can we customize a single C program for different platforms?

More information here:

- https://en.wikipedia.org/wiki/C_preprocessor
- https://sourceforge.net/p/predef/wiki/OperatingSystems/

# Outline

1. ~~Understand how C compilation works~~

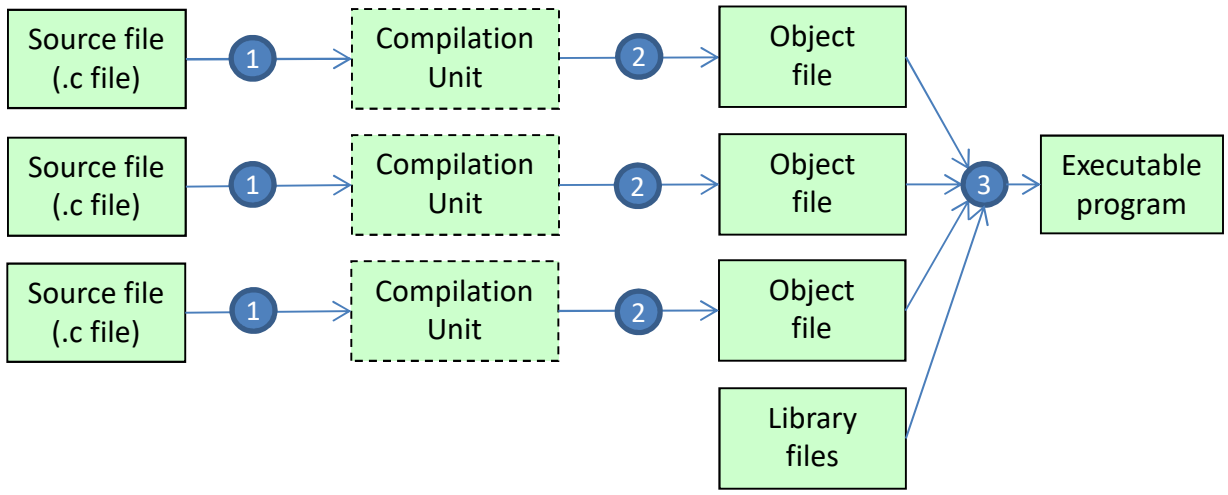
2. ~~Preprocessor directives~~

3. Why organizing code into multiple files?

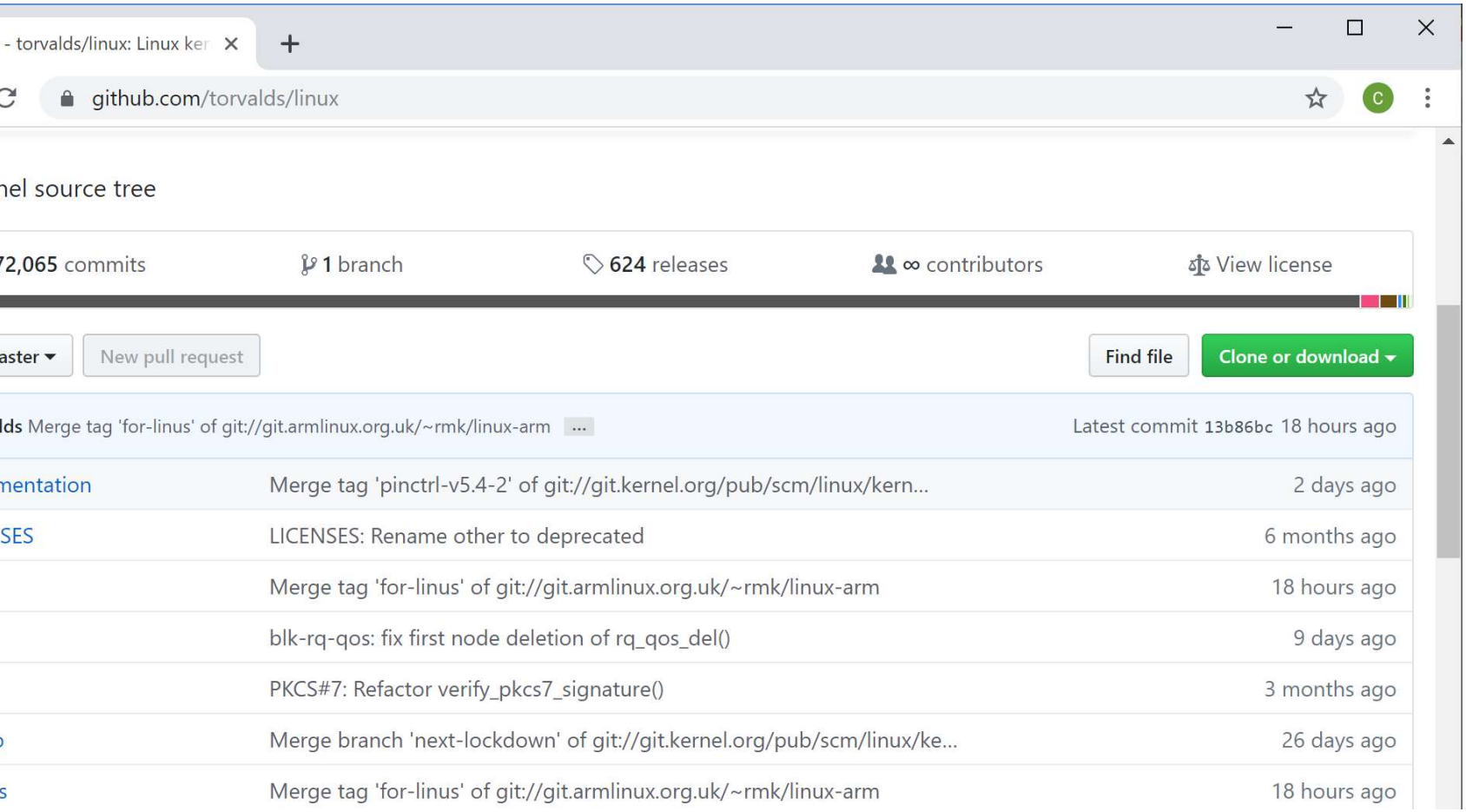
4. How to organize code into multiple files?

# 3. Why organizing code into multiple files?

- Modularity
  - This is “essential” for building large software
- Ease of maintenance
  - Update and change your (large) code



# Example: GitHub project of Linux Kernel




From: https://github.com/torvalds/linux & https://en.m.wikipedia.org/wiki/Linux_kernel
As of June 2015, over 19.5 million lines of code by almost 14,000 programmers

# When all source code are in one single file …

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  bar() ;
  …
}
int bar( void )
{
  …
}
int main()
{
  foo() ;
  bar() ;
  …
}
```

- Whenever you make a small change, the compiler need to recompile the whole file.
  - Not efficient for large software project
  - Also, not efficient for several programmers to work together on the same project

Note: nowadays, even better tools such as version control, e.g., GitHub

# Distribute code in diff. files with function prototype

file1.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  bar() ;
  …
}
```

file2.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int bar( void )
{
  …
}
```

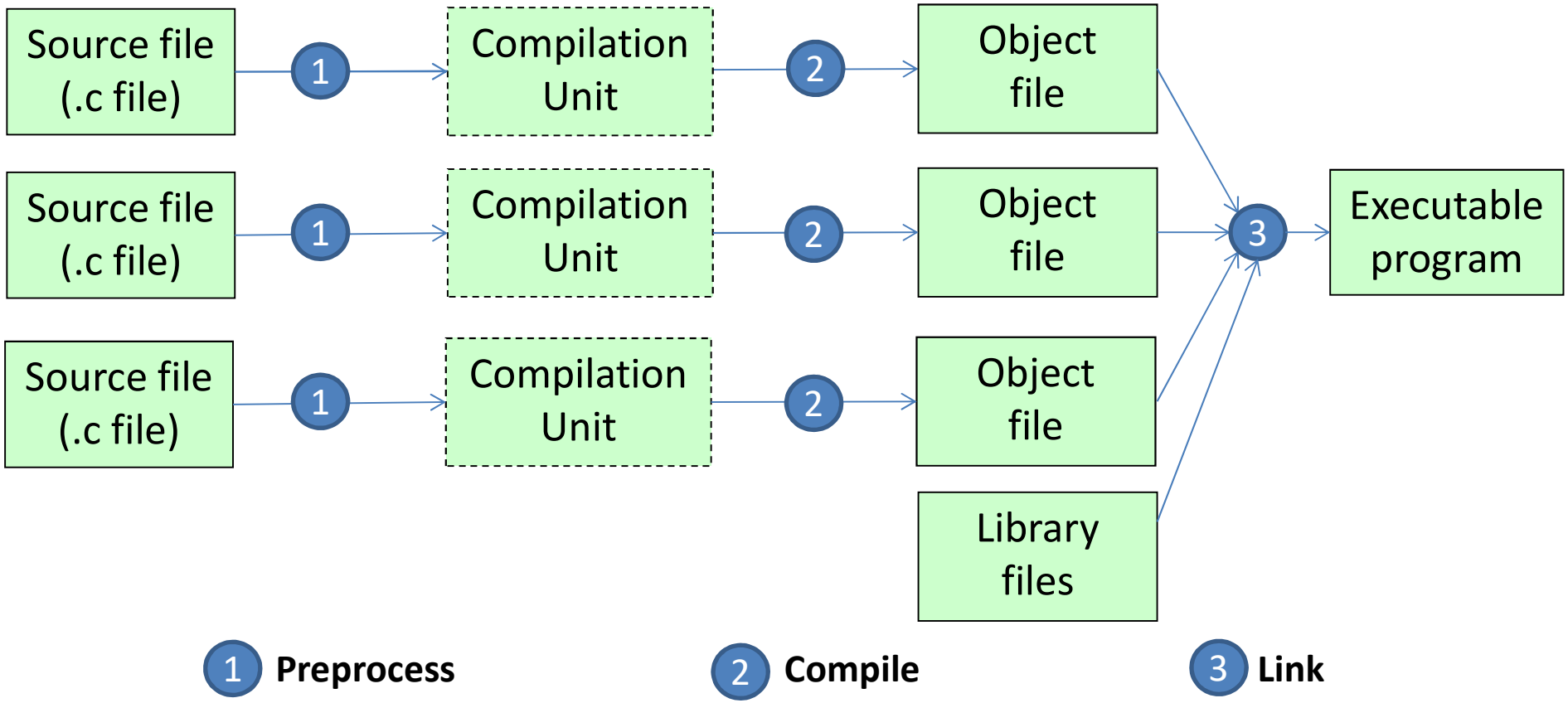
file3.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int main( void )
{
  foo() ; bar() ;
  …
}
```

- Initially, we still need to **compile each file once**, but…
- After we change the code, we only need to **recompile** the **modified file**

**Note:** The prototypes do not produce any code in the program; they only provide some info (API – application programming interface) about the functions to the compiler. As a result, they may appear in every .c file.

# 3. Why organizing code into multiple files?



# Advantage #1: change implementation

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  // Another
  // implementation
}
```

file4.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int bar( void )
{
  …
}
```

file2.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int main( void )
{
  foo() ; bar() ;
  …
}
```

file3.c

- We could replace a function (with different implementation) by replacing a file.
- **Note:** The replacement needs to have exactly the same function name, parameters, and return type.

# Advantage #2: reuse functions!!

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  // Another
  // implementation
}
```

file**4**.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int bar( void )
{
  …
}
```

file2.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int main( void ) {
  // Another program
  int x = foo() + bar() ;
  …
}
```

file**5**.c

- We could easily reuse functions in another program.

# 3. Note

- In practice, we place related functions in the same file (instead of one function per file).
  - E.g., your source code for the computer player

- The previous example does not illustrate how code is typically divided into separate files.
  - Why? Still missing header files…
  - See next part in this lecture

# Outline

1. ~~Understand how C compilation works~~

2. ~~Preprocessor directives~~

3. ~~Why organizing code into multiple files?~~

4. How to organize code into multiple files?

# 4. How to organize code into multiple files?

file1.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  bar() ;
  …
}
```

file2.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int bar( void )
{
  …
}
```

file3.c

```
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int main( void ) {
   foo() ;
   bar() ;
   …
}
```

This is what we have seen before…

Anything in common between the three files?

# Step 1: Create Headers

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int foo( void )
{
  bar() ;
  …
}
```

file1.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int bar( void )
{
  …
}
```

file2.c

```c
#include <stdio.h>

// Function prototypes
int foo( void );
int bar( void );

// Function implementation
int main( void ) {
   foo() ;
   bar() ;
   …
}
```

file3.c

```c
#ifndef _FILE1_H_
#define _FILE1_H_  1

// Function prototypes
// of the functions in
// file1.c
int foo( void );

#endif
```

file1.h

```c
#ifndef _FILE2_H_
#define _FILE2_H_  1

// Function prototypes
// of the functions in
// file2.c
int bar( void );

#endif
```

file2.h

"header files" store the prototype of the functions for use in other files!!

# Step 2: #include the header files

```
#include <stdio.h>

#include "file1.h"
#include "file2.h"


// Function implementation
int foo( void )
{
  bar() ;
  …
}
```

file1.c

```
#include <stdio.h>

#include "file2.h"


// Function implementation
int bar( void )
{
  …
}
```

file2.c

```
#include <stdio.h>

#include "file1.h"
#include "file2.h"


// Function implementation
int main( void ) {
   foo() ;
   bar() ;
   …
}
```

file3.c

```
#ifndef _FILE1_H_
#define _FILE1_H_  1

// Function prototypes
// of the functions in
// file1.c
int foo( void );

#endif
```

file1.h

```
#ifndef _FILE2_H_
#define _FILE2_H_  1

// Function prototypes
// of the functions in
// file2.c
int bar( void );

#endif
```

file2.h

"header files" store the prototype of the functions for use in other files!!

# Let's revisit math.h

code.woboq.org/gcc/include/math.h.html

Search a file or function

Browse the source code of include/math.h

```
18
19 /*
20  *        ISO C99 Standard: 7.12 Mathematics        <math.h>
21  */
22
23 #ifndef        _MATH_H
24 #define        _MATH_H        1
25
26 #define __GLIBC_INTERNAL_STARTING_HEADER_IMPLEMENTATION
27 #include <bits/libc-header-start.h>
28
29 __BEGIN_DECLS
30
31 /* Get definitions of __intmax_t and __uintmax_t.  */
32 #include <bits/types.h>
33
34 /* Get machine-dependent vector math functions declarations.  */
35 #include <bits/math-vector.h>
36
37 /* Gather machine dependent type support.  */
38 #include <bits/floatn.h>
39
40 /* Get machine-dependent HUGE_VAL value (returned on overflow).
41    On all IEEE754 machines, this is +Infinity.  */
42 #include <bits/huge_val.h>
43
44 #if __HAVE_FLOAT128 && __GLIBC_USE (IEC_60559_TYPES_EXT)
45 # include <bits/huge_val_flt128.h>
46 #endif
```

```
23 #ifndef        _MATH_H
24 #define        _MATH_H        1
```

Usually, we have #ifndef and #define at the beginning to avoid re-definition

From: https://code.woboq.org/gcc/include/math.h.html

# 4. Note

- A header file (.h file) typically contains
  - Named constants (defined using #define)
  - Function prototypes of the functions defined in the corresponding .c file

- Typically, main() is not called by other function. In such case, we don't usually need to prepare a header file for the file containing main().

- `_FILE1_H_`, `_FILE2_H_` -- These are just names (valid identifiers). Such "weird" names are typically chosen for their uniqueness to prevent name conflict -> use meaningful names!!!

- In this example, bar() does not need to call foo(), so we do not need to include file.h in file2.c.

# Next … Demo

- For a more elaborated example, please refer to "Sample Project: multifile_compile" (you may download it from course webpage: project folder)
- My suggestion on project:
  - At least two .c files, one for the overall program with main() and the other for anything on the AI player
  - Then, you can submit both for basic part and submit the AI player file for the AI part (see project spec.)
  - Of course, you may have more .c file for other purposes, e.g., anything on user interface, etc.