/unc/comp211

Systems Fundamentals

C "Modules"!

Modularizing Programs

Downsides to single file programs:

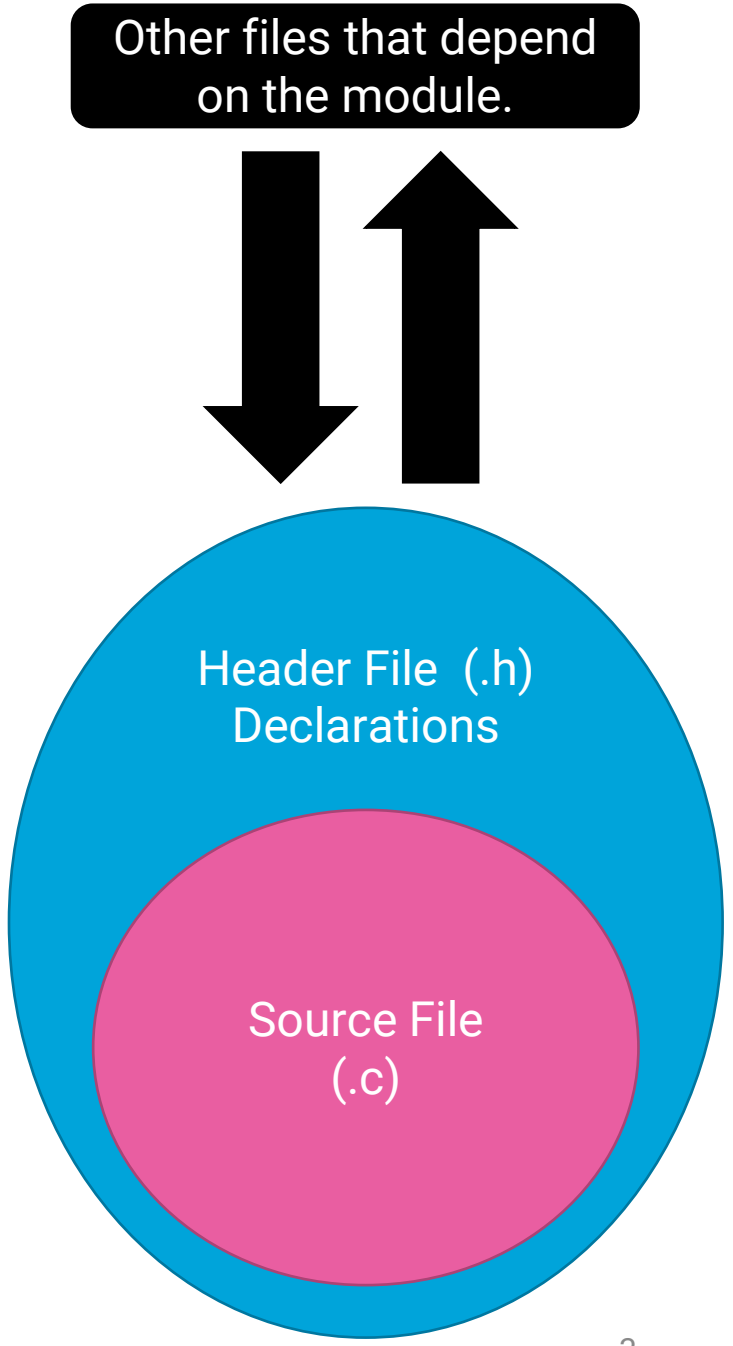
1. Single namespace for static global identifiers (such as variable/function names)
   - Static global identifiers have file-level namespaces (hidden from other files)
2. Recompiling large projects can be slow, even though changes are small.
   - Incremental compilation on a per-file basis should help avoid this downside
3. Sharing code between multiple programs requires duplication of code.
   - Duplication of code is generally best avoided!
4. Long files are more difficult to reason through separate of concerns.

"Modules" in C

- There are not *formal* modules in the C language, but the combination of a header and source file offer benefits of a module

- Header file (.h) contains function and type declarations
- Source file (.c) contains concrete implementations

- *Header file* provides the *interface* to a module.
  - The contents of a well-defined module should treat the *source file* as a black box.



Defining a Header File (1/2)

- Header filenames end in .h
- The filename conventionally matches its corresponding .c file
  - Ex: Point.h / Point.c
- Header files are surrounded in macro "include guards" (shown right, discussed on next slide)
- Declarations of functions, structs, and shared global variables are made inside the include guard.
- Documentation for end users of a module is written in the header files.
  - This should be the *only* file a programmer needs to look at in order to make use of a module!

Point.h

Open Include Guard

```
1  #ifndef POINT_H
2  #define POINT_H
3
4  typedef struct Point {
5      double x;
6      double y;
7  } Point;
8
9  /**
10  * Initialize a new Point value.
11  */
12 Point Point_value(double x, double y);
13
14 /**
15  * Compute the distance between two Points.
16  */
17 double Point_distance(const Point *self, const Point *other);
18
19 /**
20  * Print a nice representation of a Point.
21  */
22 void Point_print(const Point *self);
23
24 #endif
```

Close Include Guard

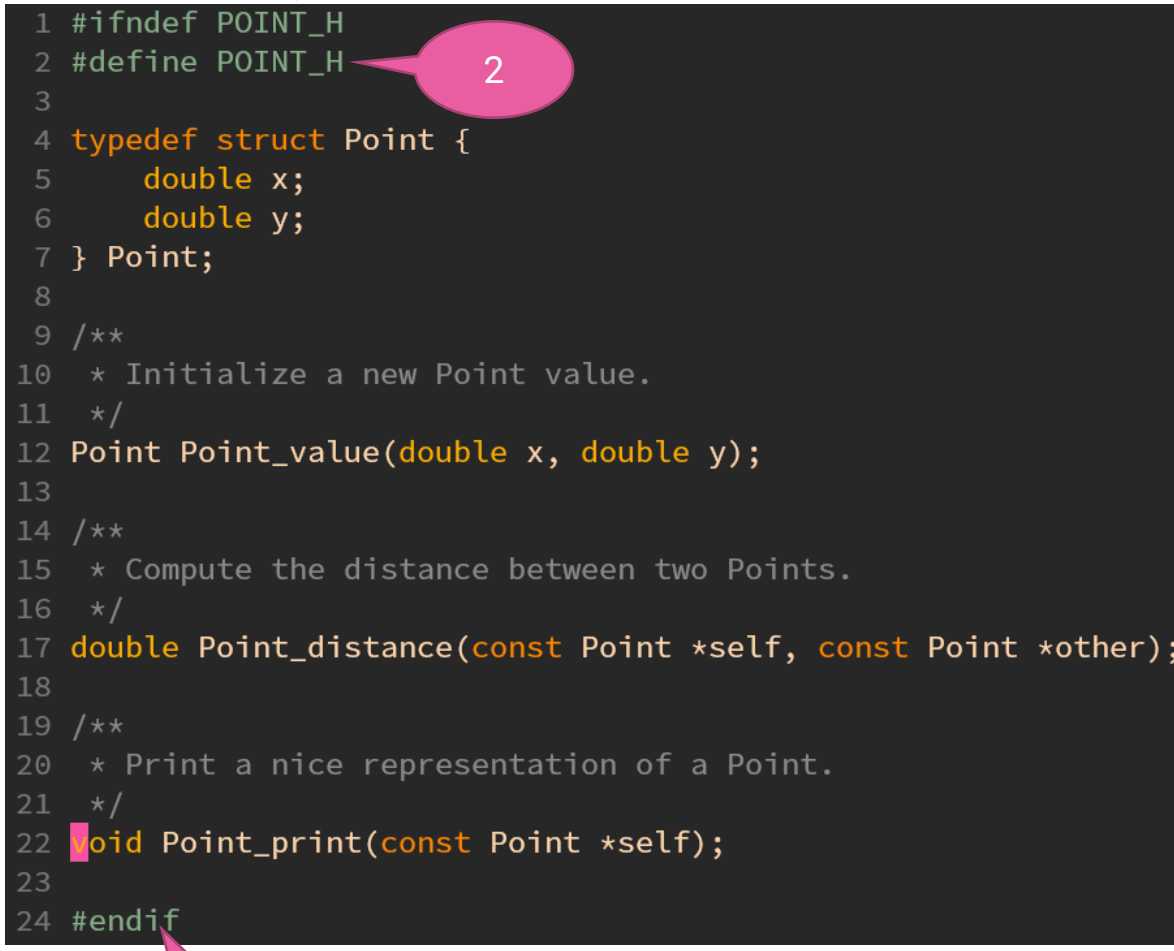
Defining a Header File (2/2) - Include Guards

- When the C preprocessor reaches an #include macro, it literally replaces the include line with the contents of the file.
- If many C files include the same header file, the declarations of the header files would be repeated which is invalid C
- To address this, **include guards** are a pattern of macro if-then statements as shown right:

1. #ifndef (if ***not*** defined) checks to see if a macro symbol has been defined
2. #define defines a macro symbol
3. #endif ends the conditional

- Convention: name the macro symbol the same as the file name with underscores replacing non-alphanumerics: ALL_CAPS_H
- Effect: The first time a file includes a header file, its declarations are loaded. Subsequent times it is skipped over. This makes including a header file *idempotent*.

Point.h




```h
#ifndef POINT_H
#define POINT_H

typedef struct Point {
    double x;
    double y;
} Point;

/**
 * Initialize a new Point value.
 */
Point Point_value(double x, double y);

/**
 * Compute the distance between two Points.
 */
double Point_distance(const Point *self, const Point *other);

/**
 * Print a nice representation of a Point.
 */
void Point_print(const Point *self);

#endif
```

Including a Header File

- So far, you've included *system library* header files:
  - #include <stdint.h>
  - #include <stdlib.h>
  - #include <stdio.h>

- The <header.h> syntax tells the compiler to *look in system include paths*
  - Some cryptic gcc flags (shown left) will show you the defaults on a system. You can also override these per project with other gcc flags.

```
learncli$ echo "" | gcc -E -Wp,-v -
#include "..." search starts here:
#include <...> search starts here:
/usr/lib/gcc/x86_64-linux-gnu/7/include
/usr/local/include
/usr/include/x86_64-linux-gnu
/usr/include
End of search list.
```

- To include a header file from a module in your project, you surround it with "s instead:
  - #include "Guards.h"
  - #include "Rational.h"

- The "s tell the compiler to look in local project director(y/ies) and can also be customized (later).

Compiling a "Module" into an Object Code file (1/2)

- C compilers have facilities to build single modules at a time
- A compiled C module is called an Object Code file (.o extension)
- An Object Code file contains machine code and a symbol table
- Symbols are global identifiers like function names and variables
  - Symbols defined in the module are mapped to their locations in the obj file
  - External symbols (imported into the module) are undefined
- Object files are not executable on their own. They must be linked with the other object files they depend on in order to be executable programs.

Incremental Compilation with Object Code files (2/2)

- Adding the **-c** flag to the compiler flags produces **object code files**

- Ex: `gcc -Wextra -Wall -std=c11 -g -c Point.c`
  - This produces the object code file Point.o

- To **link** object code files together into an **executable** file, exactly one of the object code files must have a main function symbol defined.

- Ex: **gcc Point.o main.o**
  - This produces a.out (though, with the -o option you could change the filename)

- Notice that changing one file means recompiling only its module and relinking. Next lecture we'll look at how to use the **make** build tool to automate these steps away.

Aside: Inspecting the Object Files

- The objdump utility "displays information from object files"
  - man objdump
- The -t flag outputs the symbol table (shown right)
  - Notice Point_distance is defined
  - pow, sqrt, and printf symbols are undefined (need to be filled in when linked with system libraries)
- objdump -d shows disassembled assembly code
  - Converts machine code back to assembly code

```
learncli$ objdump -t Point.o

Point.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 Point.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     F .text  0000000000000042 Point_value
0000000000000042 g     F .text  0000000000000085 Point_distance
0000000000000000         *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000         *UND*  0000000000000000 pow
0000000000000000         *UND*  0000000000000000 sqrt
00000000000000c7 g     F .text  000000000000003d Point_print
0000000000000000         *UND*  0000000000000000 printf
```