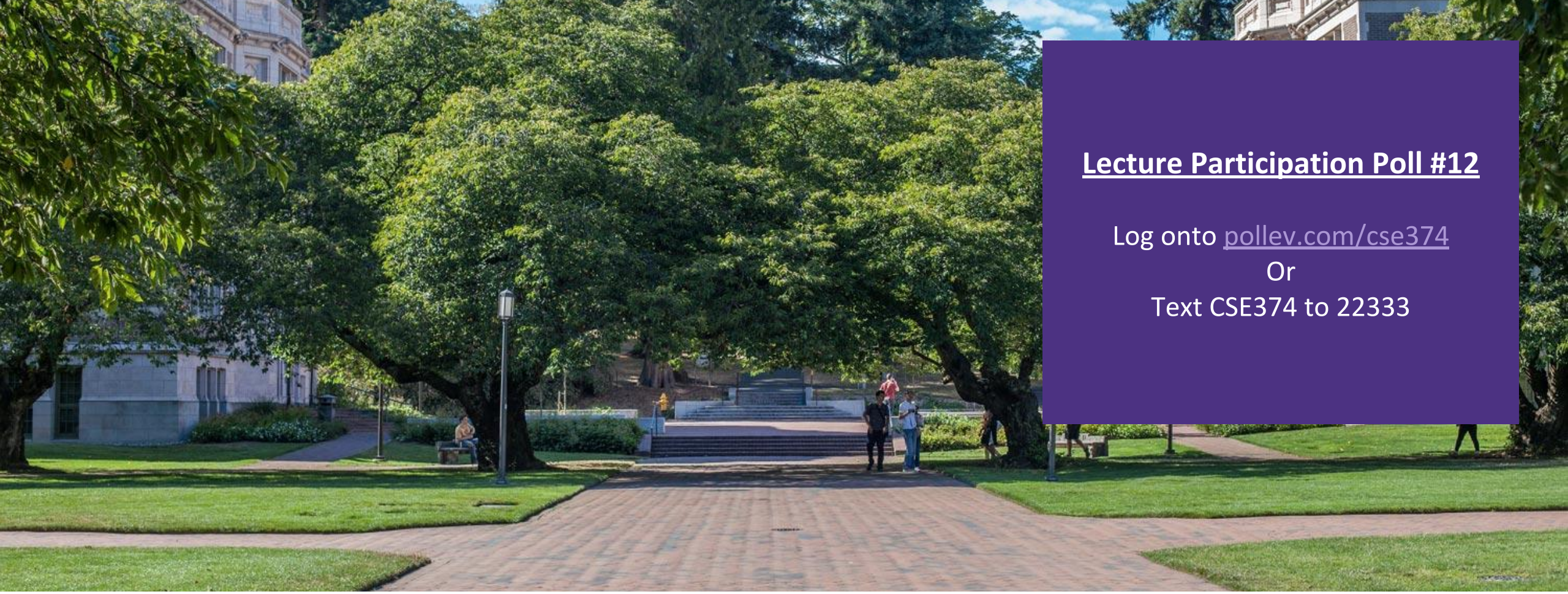


Lecture Participation Poll #12

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 12: Structs and Multi File C

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

HW2 Live - Soft Deadline Thursday October 29th at 9pm PST

- Don't need to zip files
- More hints added!

HW3 coming this week

Reminder: Midpoint Deadline Friday November 6th at 9pm PST

Review Assignment Live – Due Wednesday

- 24 hrs late 20% penalty
- 48 hrs late 50% penalty
- Not accepted more than 48hrs late

Data Types in C

- void – a place holder
- numbers – int, short, long, double, float (signed, unsigned)
- char – a very short int (1 byte) interpreted as a printable character
- pointers (T*) – stores address of where a value is stored in memory
- arrays (T[]) – implicit promotion to pointer when passed as an argument to a function or returned from a function
- booleans – not defined in C so instead we use values, 0 or NULL is interpreted as false, anything else true
- Advanced: Union T, Enum E, Function Pointers, Structs

Typedef

- A function that creates an alias for an existing type

```
typedef <type> <name>;
```

Example: In C, strings are ”char*” but we can rename them to “string”

```
typedef char* string;
int main(int argc, string *argv)
{
   string s = “hello, world”;
   printf(“%s\n”, s);
}
```

Type-casting

- **casting** – converting one type to another

```
(T)E
  * same as Java
```

```
main ()
{
    int sum = 17, count = 15;
    double mean;
    mean = (double) sum / count;
    printf(Value of mean: %f\n”, mean);
}
```

If E is a numeric type and T is a numeric type:

- To wider type, get same value
- To narrower type, may not get same value (employs mod operator)
- From floating point to int, will round (may overflow)
- From int to floating point, may round (int to double is exact on most machines)

Pointer-casting

- If be has type `T1*`, then `(T2*)E` is a (pointer)cast
- Does not alter the address stored, but used to manage types

```
void evil (int **p, int x)
{
   int *q = (int*)p;
   *q = x;
}
void f(int **p)
{
   evil(p, 345);
   **p = 17; // writes 17 to address 345 -
best case crash
}
```

Structs

- **structs** are a method of constructing new datatypes
  - store a collection of values together in memory, fields
  - similar to a Java class, but no methods
  - individual values are referred to using the "." operator
  - can use typedef to rename and turn struct tag into a "type"

```
typedef struct Cat Cat;
```

or

```
typedef struct Cat {
    …
} Cat;
```

Then you don't need keyword "struct"

`Cat mercy;` instead of `struct Cat mercy;`

```
struct Cat
{
    char *name;
    int age;
    char *breed;
}
int main()
{
    struct Cat mercy;
    mercy.name = "Iron Fist No Mercy";
    mercy.age = 6;
    mercy.breed = "Pixie Bob";
}
```

Parameters / Arguments

- Function parameters are initialized with a copy of corresponding argument
  - If the argument is a pointer, the parameter value will point to the same thing (pointer is copied)
  - arrays are passed as pointers
  - Structs are passed as a copy by default, so it is more common to intentionally pass as pointers
    - avoids copying large objects
    - allows manipulation of original struct <- allows creation of methods that manipulate new type, like Java
    - to access members you must dereference the pointer (*) and access the field (.) – use parenthesis to ensure dereference happens first
    - `(*ptr).` has a shortcut: `ptr->`

```
Cat (*ptr) = (Cat*)malloc(sizeof(Cat));
(*ptr).age = 6;
…
(*ptr).age++;
ptr->age;
```

Example: Pointer.c

```
// constructor for a new Point
Point newPoint()
{
   Point p; p.x = 0; p.y = 0; return p;
}
// translateX moves one point horizontally by deltax
void translateX(Point* p, int deltaX)
{
   p->x += deltaX; // OR (*p).x += deltaX;
}
// translateX_wrong won't move the original point
void translateX_wrong(Point p, int deltaX)
{
   p.x += deltaX;
}
// print out the point.
void print(Point* p)
{
   printf("p = (%d, %d)\n", p->x, p->y);
}
// note: here we could pass by value
void print_point(Point p)
{
   printf("p = (%d, %d)\n", p.x, p.y);
}
```

```
// main tests the Point struct
int main(int argc, char **argv)
{
   Point p = newPoint();
   printf ("Show point.\n");
   print(&p); // pass by reference
   translateX(&p, 12);
   print(&p);
   printf ("Show incorrectly translated point.\n");
   translateX_wrong(p, 12);
   print(&p);
   printf ("But pass by value works for print.\n");
   print_point (p);
}
// constructor for a new Point Point newPoint()
{
   Point p;
   p.x = 0;
   p.y = 0;
   return p;
}
```

Linked Lists


```c
#include <stdlib.h>
#include <stdio.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

```c
int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    printf(
        "%d%d%d\n",
        n3->value,
        n3->next->value,
        n3->next->next->value
    );

    free(n3);
    free(n2);
    free(n1);
}
```

Multi-File C Programming

- You can split C into multiple files!
  - What if we wanted to use Linked List code in a different project?
  - If the linked list code is long, it can make files unwieldy
  - What if we want to separate our "main" from the struct definitions
- Pass all ".c" files into gcc:

```
gcc -o try_lists ll.c main.c
```

Must include code header files to enable one file to see the other, otherwise you have linking errors

```
$ gcc -g -Wall -o try_lists ll.c main.c
main.c: In function 'main':
main.c:5:5: error: unknown type name 'Node'
    5 |     Node *n1 = make_node(4, NULL);
      |     ^~~~
main.c:5:16: warning: implicit declaration of function 'make_node' [-Wimplicit-function-declaration]
    5 |     Node *n1 = make_node(4, NULL);
      |                ^~~~~~~~~
```

Sharing code across files

- Must always declare a function or struct in every file it's used in
  - Thank goodness C lets us separate declarations and definitions ;)
  - Include function header as definition

```
Node *make_node (int value, Node *next);
```

  - Include struct type definition

```
typedef struct Node
{
    int value;
    struct Node *next;
} Node;
```

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node
*next);

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

```c
#include <stdlib.h>

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);

Node *make_node(int value, Node *next) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

Header Files

- Copying your function declarations to every file you want to use them is not fun
  - If you forget to make a change to all of them, confusing errors occur!
- A ***header file*** (.h) is a file which contains just *declarations*
- #include inserts the contents of a header file into your .c file
  - Put declarations in a header, then include it in all other files
  - Two types of #include

`#include <stdio.h>`

  - Used to include external libraries. Does not look for other files that you created.

`#include "myfile.h"`

  - Used to include your own headers. Searches in the same folder as the rest of your code.

```c
typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
```

ll.h

```c
#include <stdlib.h>
#include <stdio.h>

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```

ll.c

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```

main.c

Header Guards

- Consider the following header structure:
  - Header A includes header B.
  - Header C includes header B.
  - A source code file includes headers A and C.
  - The code now includes two copies of header B!
  - Solution: "**header guard**"

```c
#ifndef LL_H
#define LL_H

typedef struct Node {
    int value;
    struct Node *next;
} Node;

Node *make_node(int value, Node *next);
#endif
```
ll.h

```c
#include "ll.h"

int main() {
    Node *n1 = make_node(4, NULL);
    Node *n2 = make_node(7, n1);
    Node *n3 = make_node(3, n2);

    // rest of main…
}
```
main.c

```c
#include <stdlib.h>
#include <stdio.h>

#include "ll.h"

Node *make_node(int value, Node *next) {
    Node *node =
(Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = next;
    return node;
}
```
ll.c

Libraries in C

- Remember #include <stdio.h>?
- That tells our .c file what function declarations are in stdio.h, but what about the function definitions? (i.e. the code)
- We don't have access to stdio.c
- Instead, we have a pre-compiled library that we can call functions within
  - The stdio library is included by default with gcc
- In C, these "libraries" are called object files

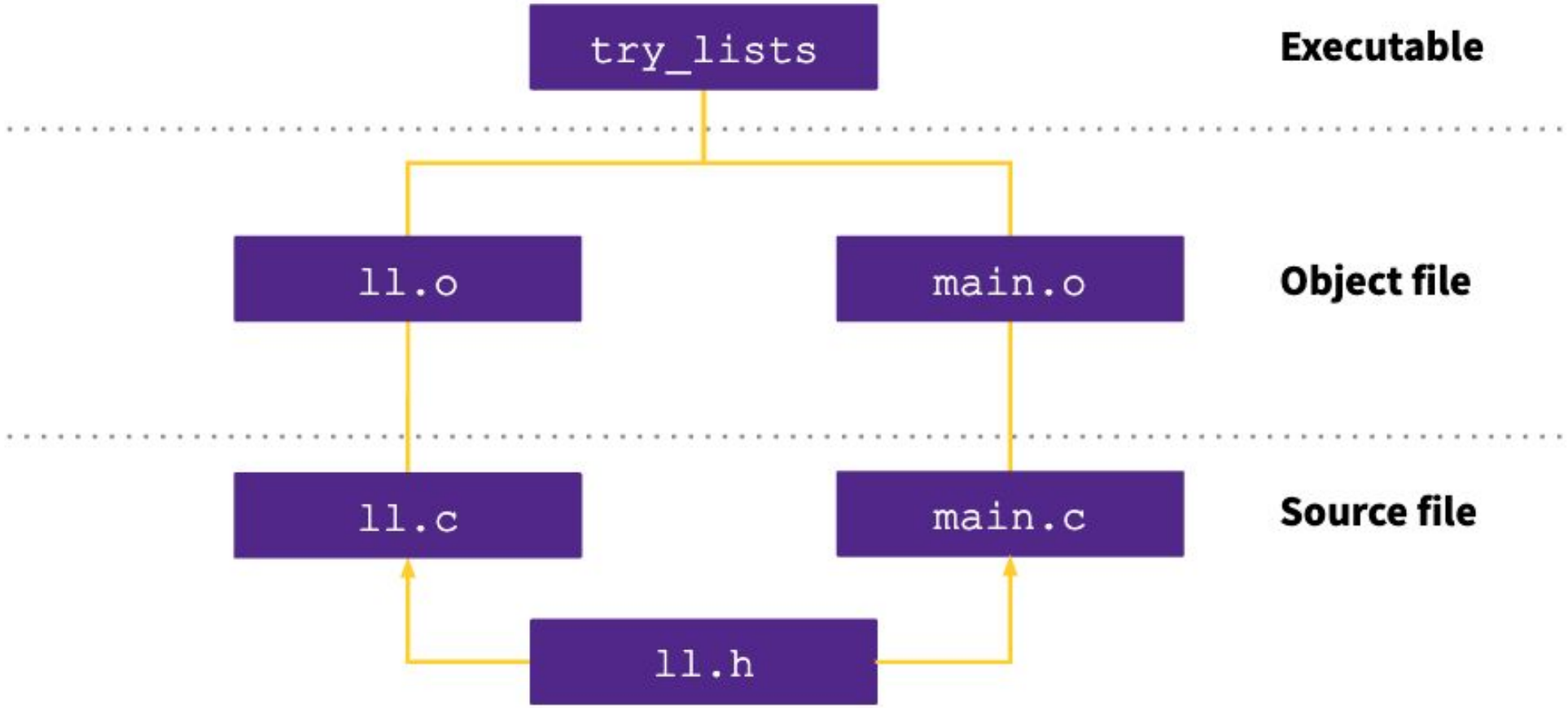
Object Files

- All C code is broken down into functions
- When compiled, a function is turned into "machine code" which the physical CPU electronics can understand
- *Object files* contain the machine code for the functions within
- These define the complete behavior of a function and can be called from your own C code

Linking in C

- Every time you have compiled something with gcc, you have actually been doing *two* things:
  - Compiling
  - Linking
- Compiling: Translating C code (a *single* .c file) into machine code stored in *object files*
- Linking: Combining many object files into one executable
- Building multiple programs which use some of the same source code
  - Compile each object once and re-use it for multiple executables
- Many files
  - Slow-to-compile files which you don't change often don't have to be re-compiled
  - incremental compilation: Huge projects can take hours or days to compile from scratch! We can save time by only re-compiling what has changed.

Dependency Graph: linked list project



Example

Consider this dependency gra
What files (*source* and *objec*
required when building
`program_two`?

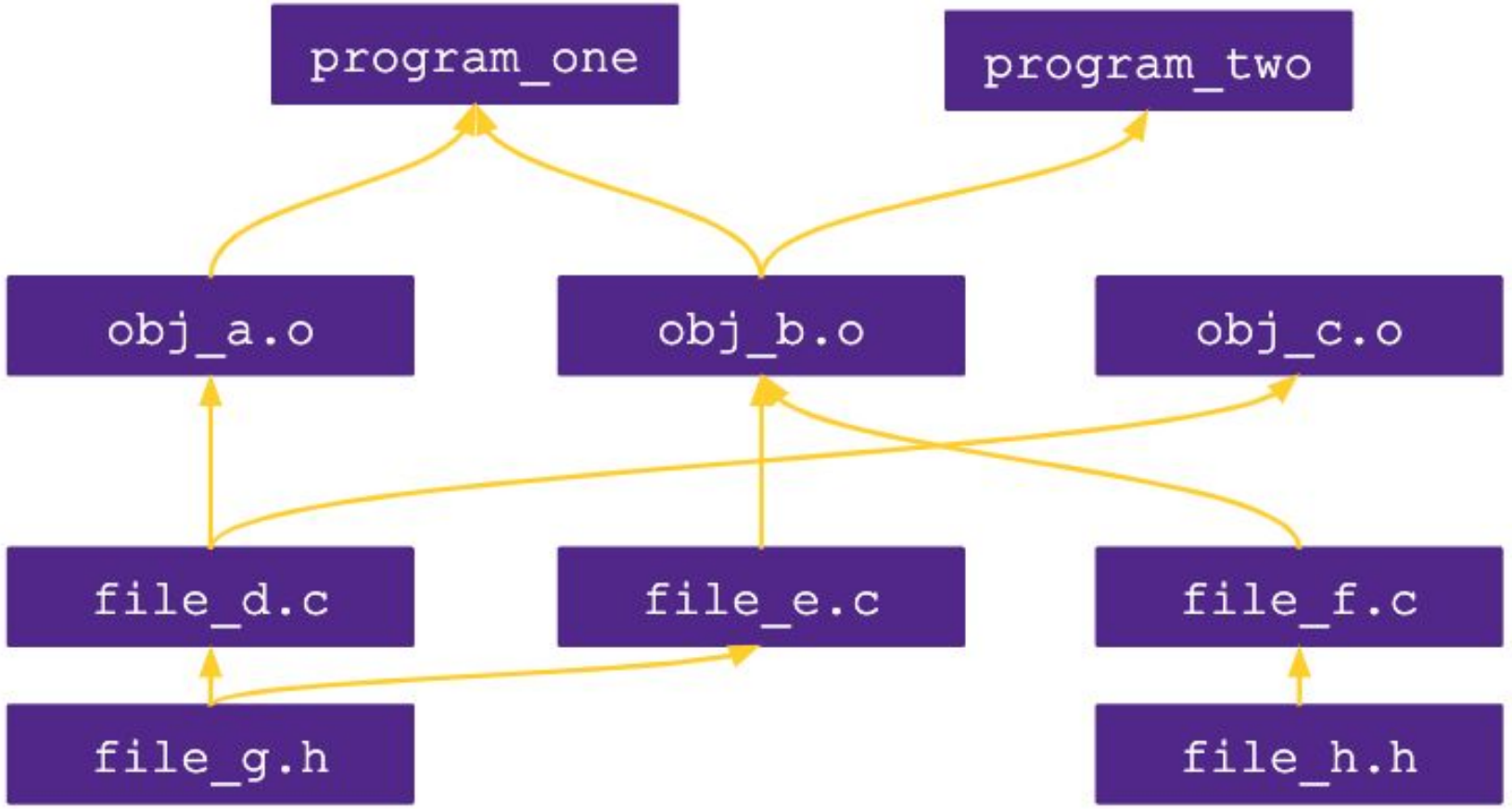
A. b, e

B. b, e, g

C. a, b ,c, e, f

D. b, e, f, g, h

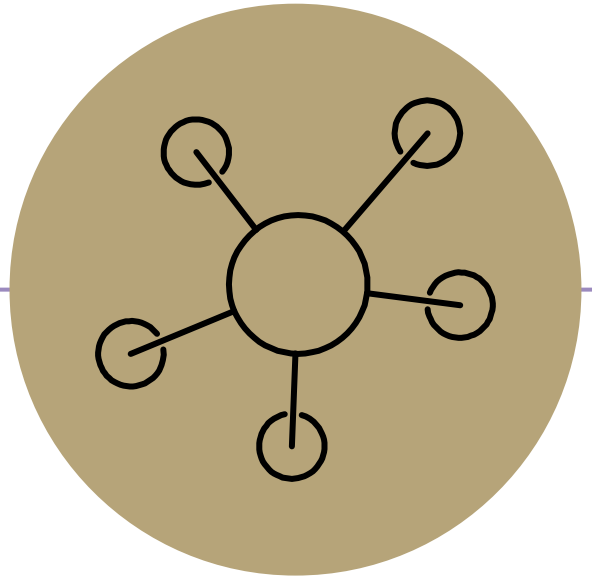
E. b, d, e, f, g, h



Automating Dependency Graphs with Make

- make is a program which automates building *trees* of dependencies
  - List of rules written in a Makefile declares the commands which build each intermediate part
  - Helps you avoid manually typing gcc commands
- single rule specifies:
  - An output file to be generated (also called a target)
  - List of input files (also called sources)
  - List of commands which will turn the input files into the output file
- "To build this target, make sure you have these files available, and then run these commands"
- Make can check when you've last edited each file, and only build what is needed!
  - Files have "last modification date". make can check whether the sources are more recent than the target.
- Rule syntax:

```
ll.o: ll.c ll.h
    gcc -c ll.c
```



Appendix