# CSE 374 Programming Concepts and Tools

Lecture 12 - Struct, Compiler, Makefile

# This Week

**Today: Struct, Compiler, Makefile**

Wednesday: C Data Structures

Friday: Testing

# Today

Structs

Modularization with `gcc`

Compilation

- Dependencies

**`make`** and Makefiles

# Review: Debugging Segfaults

If we get a segmentation fault:

1. Compile with debugging symbols using
   `gcc -g -Wall -std=c11 -o executable_name c_file.c`
1. **`gdb ./executable_name`**
2. Type "**`run`**" into GDB
3. When you get a segfault, type "**`backtrace`**" or "**`bt`**"
4. Look at the line numbers from the backtrace, starting from the top

OR: **`valgrind --leak-check=full ./executable_name`**

# Review: Datatypes in C

- `void`: A placeholder and/or no value
- 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 pointer when passed as an argument to a function or returned from a function
- booleans: not defined in C so instead we use integers, `0` or `NULL` is interpreted as false, anything else true
- Advanced:  Union T, Enum E, Function pointers, Structs

# Struct

# struct

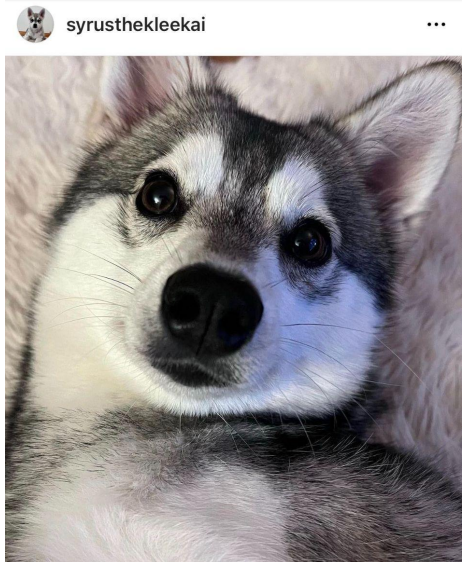
A struct is a group of variables which acts as a single value

Contains **fields**, each with their own name and type

- Like Java class, except *no* methods
- Like Javascript objects, or Typescript record types
- Fields are stored *contiguously* in memory

The struct itself is its own type

- Referred to as: **`struct <typename>`**




```
struct Dog {
  int age;
  char* name;
  char* breed;
};
```

```
struct Dog syrus;
syrus.age = 2;
syrus.name = "Syrus";
syrus.breed = "Alaskan Klee Kai";
```

# struct example

```
void print_dog(struct Dog dog) {
  printf("%s is a %d-year-old %s.\n",
         dog.name, dog.age, dog.breed);
}
int main() {
  struct Dog syrus;
  syrus.age = 2;
  syrus.name = "Syrus";
  syrus.breed = "Alaskan Klee Kai";
  print_dog(syrus);
}
```

```
struct Dog {
  int age;
  char* name;
  char* breed;
};
```

# Trick: avoid having to type "`struct`"!

Keyword "**`typedef`**" can be used to make the struct usable as its own name, without "**`struct`**" as a qualifier

How to use: use the **`typedef`** keyword and add the struct's name after the closing "}"

```
typedef struct Dog {
  int age;
  char* name;
  char* breed;
} Dog;
```

=

```
struct Dog {
  int age;
  char* name;
  char* breed;
};
typedef struct Dog Dog;
```

typedef char* string;

# Aside: `typedef`

Not really a new type – just creating an alias for an existing type

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

In C, strings are "`char*`", but if we wanted to actually provide the name "`string`", we could!

```
typedef char* string;
int main(int argc, string argv[]) {
    string s = "hello, world!";
    printf("%s\n", s);
}
```

# Working with `struct`

Java

```
Object o = null;
o = new Object();
```

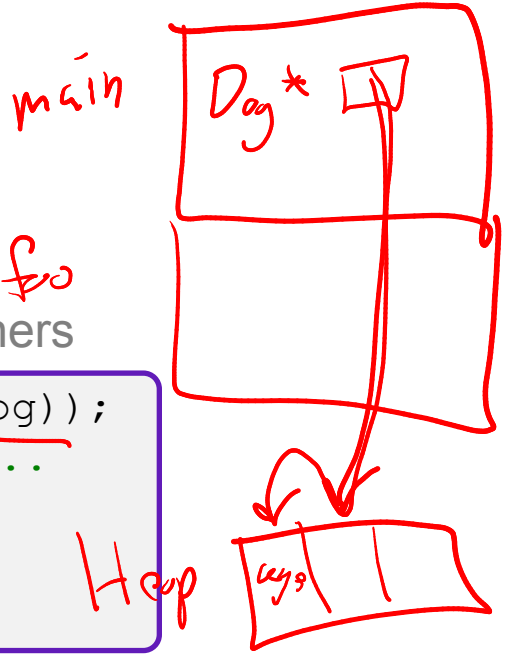
Function parameters are initialized with a copy of corresponding argument

- If the argument is a pointer, the parameter value will point to the same thing
- Arrays are passed as pointers
- Structs are passed as a **copy** by default

But, is often more convenient to use **pointers**

- Avoids copying large objects
- Enables allocating with `malloc`, then passing it to others

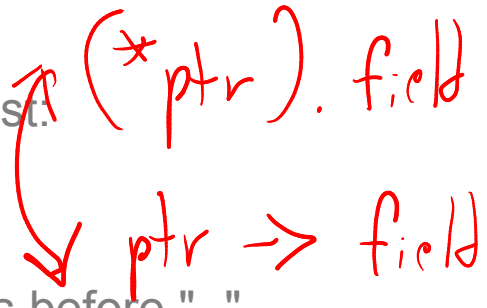
```
Dog* ptr = (Dog*) malloc(sizeof(Dog));
// not shown: initialize the dog...

(*ptr).age++;
char* name = (*ptr).name;
```



# Struct pointers

To access members, of a struct **pointer**, you must:

- Dereference the pointer (`*`)
- Access a particular field (`.`)
- *Use parentheses* to ensure that "`*`" happens before "`.`"




Writing `(*ptr).` is tedious. Instead, we have special syntax **`ptr->`**

```
Dog* ptr = (Dog*) malloc(sizeof(Dog));
// not shown: initialize…

ptr->age++;
char* name = ptr->name;
```

# Aside: `Option struct`

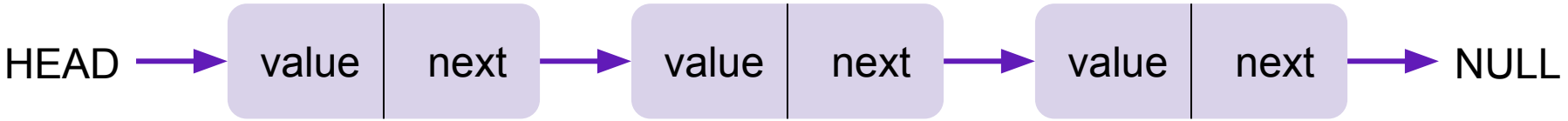
A good example for a struct would be the **option argument** from HW3!

- Takes up more space than just a single `int`, so it's not *optimal* in terms of space, but it can be more clear in *readability*!
- There are many ways to structure your code effectively
  - All have tradeoffs, pros and cons :)

```
// Option represents the option
// specified by the user, if any.
typedef struct Option {
  int bytes; // -c
  int words; // -w
  int lines; // -l
} Option;
```

# Data structure example: Linked Lists

# Linked Lists




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

# Polling Time!

Please answer in the LP12 assignment on Gradescope!

Given the definition of a `Node`, which of the following statements are *syntactically valid*?

Select *all* that apply:

- [x] `n1.next = &n2;`
- [ ] `n2->value = 373;`
- [x] `n1.next->value = 374;`
- [ ] `(&n2)->next->next->value = 474;`

```
typedef struct Node {
    int value;
    struct Node* next;
} Node;
Node n1, n2;
```

# Linked Lists: `make_node`

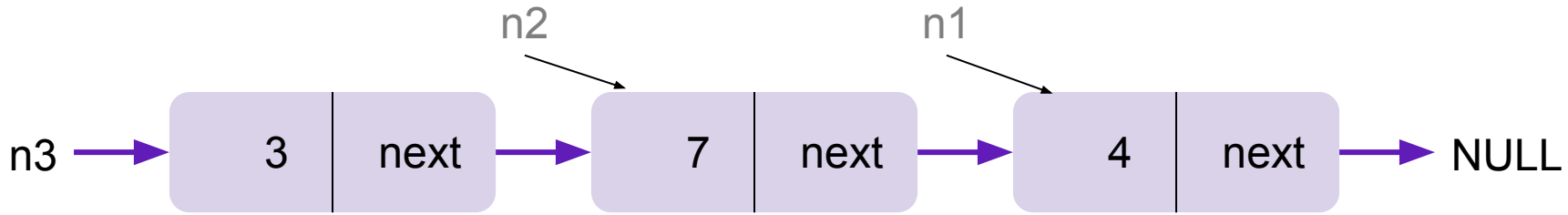
The `make_node` function creates and initializes a new node for a linked list.

- Dynamically allocates memory for a new `Node` structure on the heap
- Initializes the fields in the node
- Returns a pointer to the newly allocated and initialized node

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

Heap

# Linked Lists: build a linked list




```
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);
```

# Linked Lists: structs, pointers, and `malloc`!

```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);
}
```

# Modularization

# Multi-File C Programming

Wouldn't it be nice if our "main" could be separate from the linked list code?

- What if you want to use linked list 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?
- Improves *readability* and *discoverability*!

We can split our project into **multiple files**

- Simply pass all the "`.c`" file names to `gcc`:

```
gcc -g -Wall -std=c11 -o try_lists ll.c main.c
```

But… There's a catch!

# Multi-File C Programming

```
gcc -g -Wall -std=c11 -o try_lists ll.c main.c
```

How does the code in one file know about functions/structs in the other?

- It doesn't, by default!

```
main.c:5:5: error: use of undeclared identifier 'Node'
    Node* n1 = make_node(4, NULL);
    ^
main.c:5:11: error: use of undeclared identifier 'n1'
    Node* n1 = make_node(4, NULL);
          ^
main.c:5:16: error: implicit declaration of function 'make_node'
    Node* n1 = make_node(4, NULL);
               ^
```

# Multi-File C Programming

We must always **declare** a function or struct in **every file** it's used in

For structs, a **declaration** is simply another copy of the struct type definition.

For functions, a **declaration** is the header of the function

- Tells the compiler: "I will later define a function which looks like this."

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

# A multi-file C program

ll.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;
}
```

main.c

```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…
}
```

# Header Files

Manually copying your function declarations to every file is not fun

- If you forget to make a change to all of them, confusing errors occur!
- Don't repeat yourself! (DRY)

A header file (`.h`) is a file that just contains **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
- `#include <foo.h> // standard libraries`
  - searches for `foo.h` in "system include" directories
- `#include "foo.h" // developer files`
  - searches current directory, let developers break project into smaller files

# A multi-file C program

*Using header files* ⭐⭐⭐⭐⭐

ll.h

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

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

ll.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;
}
```

main.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…
}
```

# Header guards

Consider the following header structure:

- Header A includes header B.
- Header C also includes header B.
- A source code file includes headers A and C.

The code now includes two copies of header B!

Solution: "**header guard**"

- Uses **`ifndef`** to check if header is already defined
  - Only runs code if the macro **`LL_H`** wasn't already defined, i.e., included!

ll.h

```
#ifndef LL_H
#define LL_H

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

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

# Demo: Linked List

# C Compilation Process and Libraries

# 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** contain the machine code for the functions within
- When compiled, a function is turned into "machine code" which the physical CPU electronics can understand

# Linking in C

Every time you have compiled something with `gcc`, you have actually been doing *two* things: **Compiling** and **Linking**

- Compiling: Translating C code (in a single `.c` file) into machine code stored in **object files**
- Linking: Combining many object files into one executable


Programmer code – source file

Compiled machine code - object file

# Why compile & link separately?

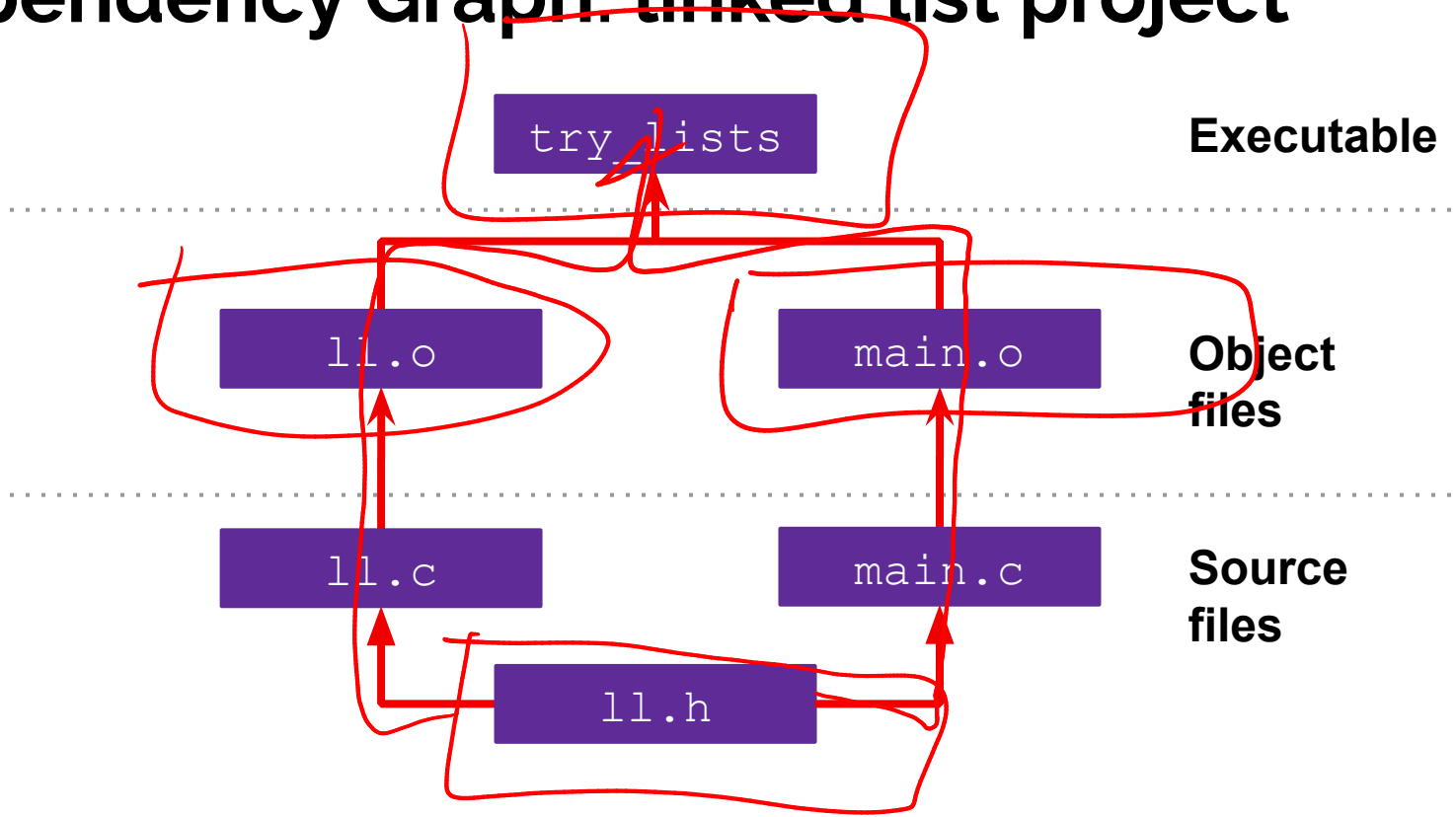
Maybe you're building multiple programs that use some of the same source code

- Compile each .c file *once* and *reuse* the object file for multiple executables
- Example: linked list
  - `gcc -g -Wall -std=c11 -c ll.c`
    - `-c` option makes it compile *without linking*, and doesn't require a main function
    - now we can use the resulting ll.o for multiple future executables


Maybe you have many files

- Slow-to-compile files 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



# Compiling and Linking with gcc

One command:

```
gcc -g -Wall -std=c11 -o try_lists ll.c main.c
```

Separate steps:

Compile flag

```
gcc -g -Wall -std=c11 -c ll.c
```

→ ll.o

```
gcc -g -Wall -std=c11 -c main.c
```

→ main.o

Linking

```
gcc -g -Wall -std=c11 -o try_lists ll.o main.o
```

# Demo: Updating linked list `main.c`

# Polling Time!

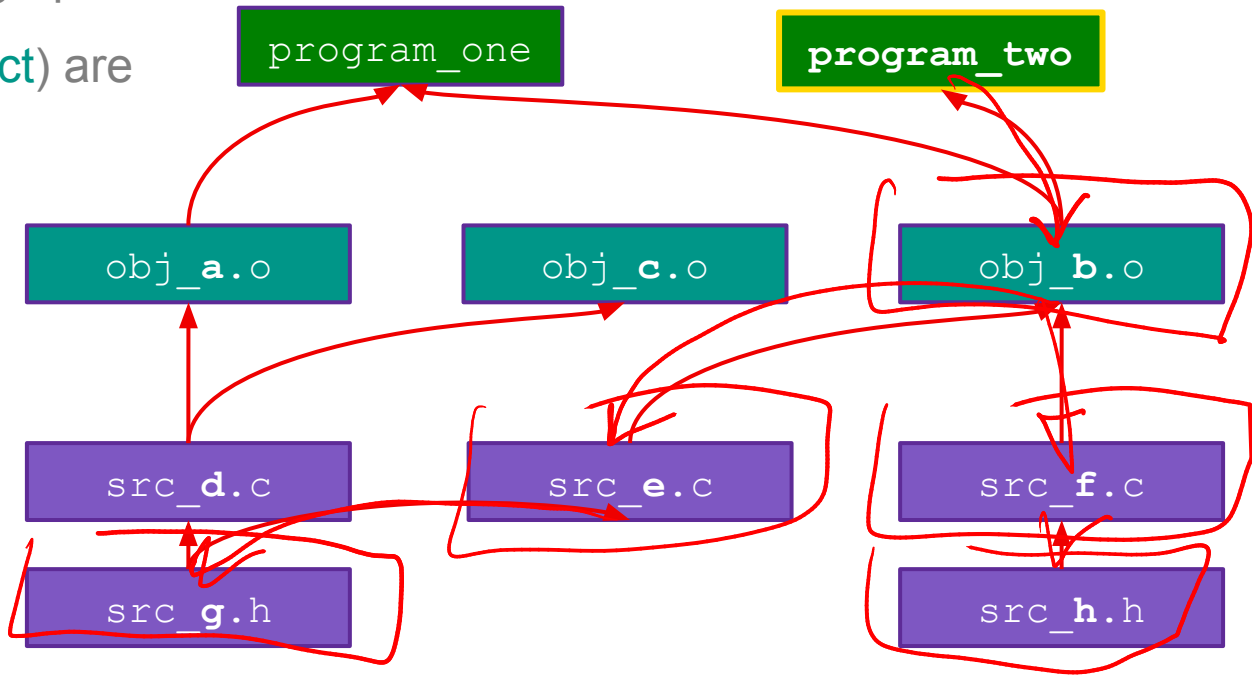
Please answer in the LP12 assignment on Gradescope!

Consider this dependency graph:

What files (source and object) are *required* when building **`program_two`**?

Select *all* that apply:

- [x] a
- [x] b
- [ ] c
- [ ] d
- [x] e
- [x] f
- [ ] g
- [ ] h



# Automating builds with `make`

# Automating Dependency Graphs with make

**`make`** is a tool that automates building dependency graphs

- List of **rules** written in a `Makefile` declares the commands which build each intermediate part
- Helps you avoid manually typing `gcc` commands, easier and less prone to typos
- Automate build process (a **build system**).

Makefiles are a list of **make rules** which consist of:

- Target: an output file to be generated, dependent on one or more sources
- Sources: input source code to be built
- Recipe: list of commands to generate target

# Makefile logic

**make** builds based on **structural organization** (dependencies between code)

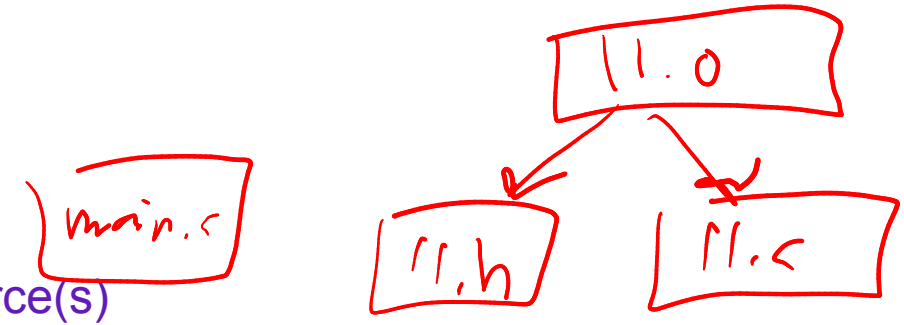
**Recursive** – if a source is also a target for other sources, must also evaluate its dependencies and remake as required

Make can check when you've last edited each file, and *only* rebuild what is needed!

- Files have "last modified" date.
- `make` can check whether the sources are more recent than the target.

Make isn't language specific: recipe can be any valid shell command

# Rule Syntax




target

source(s)

```
ll.o: ll.c ll.h
	gcc -g -Wall -std=c11 -c ll.c
```

Tab

recipe (any shell command)

# Rule Syntax, with variables

`make` variables help reduce repetitive typing and make alterations easier

- Can change variables from command line
- Enables us to reuse Makefiles on new projects
- Can use conditionals to choose variable settings

```
CC = gcc
CFLAGS = -g -Wall -std=c11

ll.o: ll.c ll.h
	$(CC) $(CFLAGS) -c ll.c
```

# Important notes on syntax

Makefile variables are similar to Bash

- Doesn’t need quotes
- Requires parentheses: `$(CC)` is valid, `$CC` is not!

Commands within a rule are run as bash commands

- Remember to list **all** input files, including header files, even if they're only used via **`#include`**
- Commands must be indented with tabs, *not* spaces!

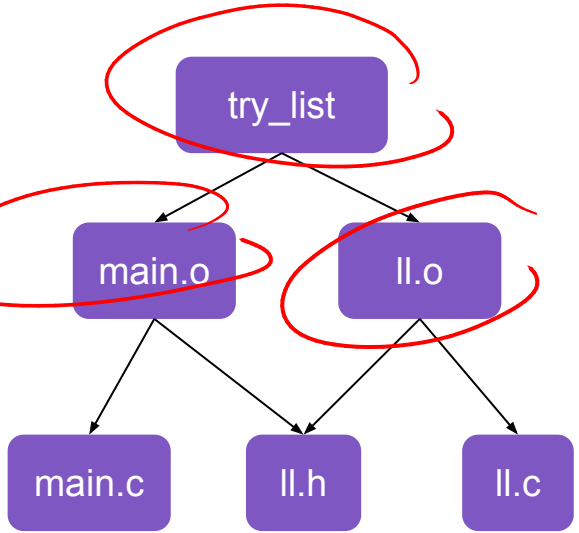
# Example Makefile

Makefile

```
CC = gcc
CFLAGS = -g -Wall -std=c11

try_lists: main.o ll.o
    $(CC) $(CFLAGS) -o try_lists main.o ll.o

main.o: main.c ll.h
    $(CC) $(CFLAGS) -c main.c

ll.o: ll.c ll.h
    $(CC) $(CFLAGS) -c ll.c
```

variable definitions

(must include rules for each file)




Rules define dependency hierarchy

# Using a `Makefile`

Run **`make`** command from within same folder

**`make [ -f makefile] [ options ] … [ targets ] …`**

- `-f` specifies the makefile name, if not provided, will default to “Makefile”

By default, **`make`** will starts with the **first rule** in file

- You can specify one or more targets on the command line instead (e.g. `make ll.o`)

**`make`** will **automatically** follow the dependencies, and run each rule needed

- Think: dependencies-of-dependencies, aka *transitive* dependencies

# "Phony" Targets

A target which doesn't actually create the listed output file

A way to force run commands regardless of dependency tree

Common uses:

- **`all`**: specifies what to make in a complete build.
- **`clean`**: cleans up generated files
- **`test`**: specifies test functionality
- Printing messages or info

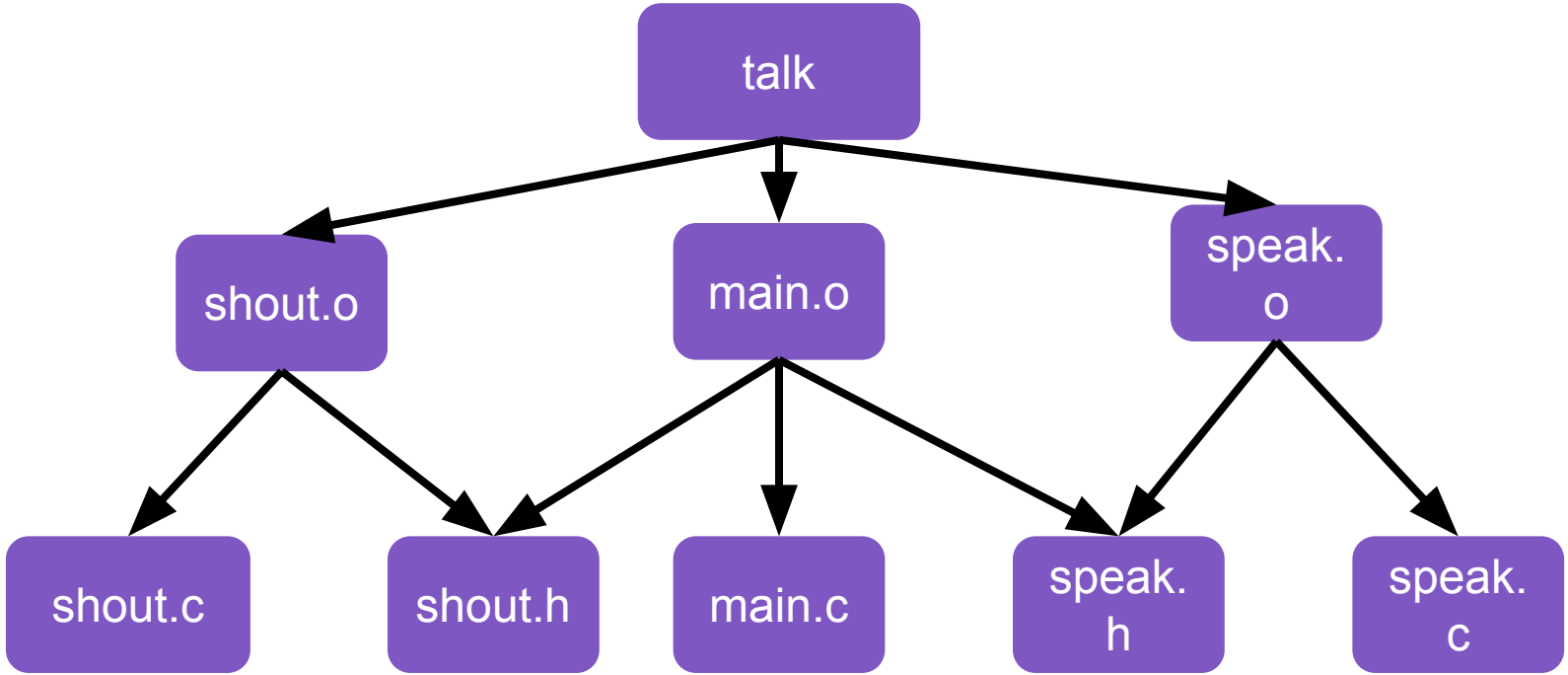
```
all: try_lists test_suite
clean:
  rm ll.o main.o try_lists
test: test_suite
  ./test_suite
```

# Example: Multiple final outputs, phony targets

Makefile

```
CC = gcc
CFLAGS = -g -Wall -std=c11

all: my_program your_program

my_program: foo.o bar.o
    $(CC) $(CFLAGS) -o my_program foo.o bar.o

your_program: bar.o baz.o
    $(CC) $(CFLAGS) -o your_program bar.o baz.o

# not shown: foo.o, bar.o, baz.o targets

clean:
    rm *.o my_program your_program
```

# Dependencies of speak/shout



# Demo: Makefile in "speak/shout"

# Assignment Reminders

EX10 due Fri 7/24 **@10:49am**

Mid-Course Survey due on **Monday 7/27 @11:59pm**

- Let us know how this course is going so far!

Make sure to submit LP12 before the next class as well.

All of these assignments can be found on our course's Gradescope page.

# Bonus Slides 

# Aside: “special” automatic variables

`make` provides automatically-set variables for the current rule

Among them:

- Current target
- First source file
- All source files

Can simplify writing, copying rules

For further research:

- https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

# More Make Tools

`ifdef` checks if a given variable is defined for conditional execution

- `ifndef` checks if a given variable is NOT defined

Special characters

- `$@` for target
- `$^` for all sources
- `$<` for left-most source
- `\` enables multiline recipies
- `*` functions as wildcard (use carefully)
- `%` enables implicit rule definition by using % as a make specific wildcard

# Extra Characters

In commands (short list):

- `$@` for target
- `$^` for all sources
- `$<` for left-most source

Examples:

- ```
  widget$(EXE): foo.o bar.o
      $(CC) $(CFLAGS) -o $@ $^
  ```
- ```
  foo.o: foo.c foo.h bar.h
      $(CC) $(CFLAGS) -c $<
  ```

Also use wildcards ( ex. `*.o` ), but you need to be careful.

Use the ‘wildcard’ function for precision.

```
$(wildcard *.o)
```

https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function

# Example Makefile

Makefile

```
CC = gcc
CGLAGS = -Wall

foo.o: foo.c foo.h bar.h
     $(CC) $(CFLAGS) -c foo.c -o foo.o

make CFLAGS=-g

EXE=
ifdef WINDIR #defined on Windows
     EXE=.exe
endif
```

```
widget$(EXE): foo.o bar.o
     $(CC) $(CFLAGS) -o widget$(EXE)\
          foo.o bar.o

OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
     gcc -o widget $(OBJFILES)

%.o: %.c
     $(CC) -c $(CFLAGS) $< -o $@

clean:
     rm *.o widget
```

# Dependency Generation

Make has no knowledge of dependency trees. If you make a mistake in your source list make can’t fix it.

Consider auto-generation:
In C:

```
$gcc -MM target.c
```

Can ‘make depend’:

```
depend: $(PROGRAM_C_FILES)
    gcc -M $^
```

# Problem of multiple ‘main’ functions

```
//sample.c
#ifdef WIN32
int main() {
    //in this case only this main()
      will be compiled.
}
#endif

#ifdef LINUX
int main() {
  //another main for linux platform
}
#endif

# sample Makefile
ifdef WINDIR # defined on Windows
   CFLAGS += -D WIN32
endif
```

You would not use two ‘main’ functions, because main is always the single entry point.

- It works in Java because we can define one ‘main’ for each class namespace.  We don’t have the same concept in C

Your code could define two mains, and choose one at pre-process time.

You could also include code that was chosen with a compiler flag (such as