CS 162
Intro to Computer Science II

Lecture 7

Structs (cont.)

File Separation & Compilation

Makefile

1/29/24

Oregon State University

Odds and Ends

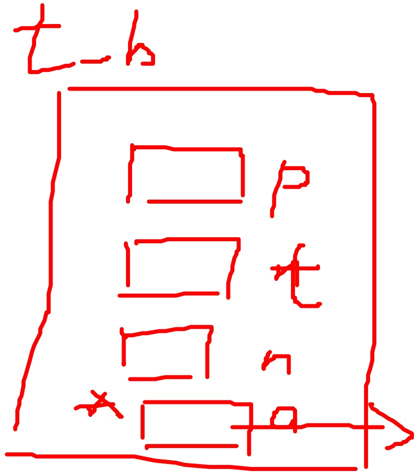
- Lab 4 posted
- Assignment 2 will be posted by today
- Sign up for demos if you haven’t already!

Lecture Topics:

- Structs (cont.)

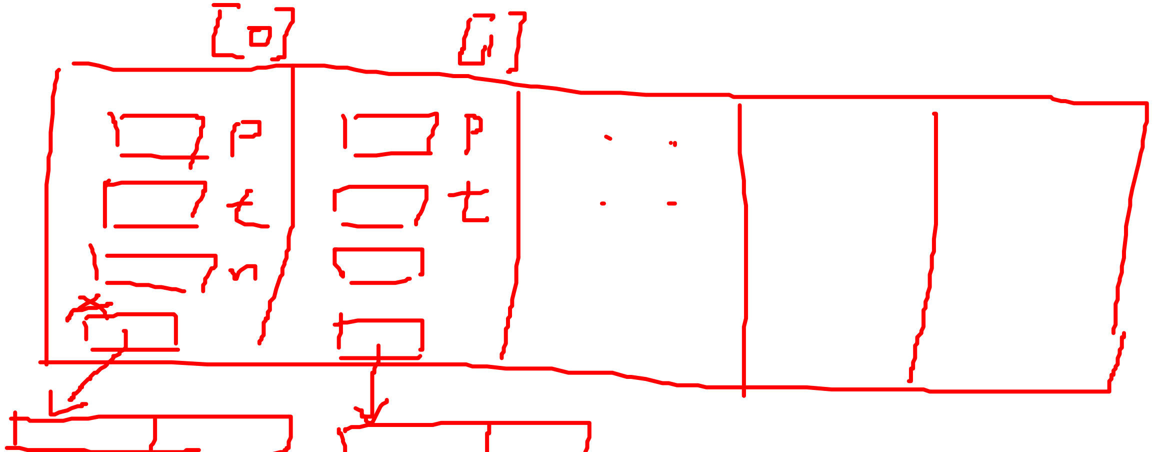
Recap: How to define a struct?

```
// definition of a Book struct
struct Book {
        int pages;
        string title; // a string inside the struct
        int num_authors;
        string* authors; // a pointer to a string
};

// declare a Book object (item)
Book text_book;

// declare and initialize at the same time
Book b1 = {.pages = 150, .title = "Harry Potter", .num_authors = 2};
//or
Book b1 = {150, "Harry Potter", 2};
```



Note: in order, non-skip

Working with structs



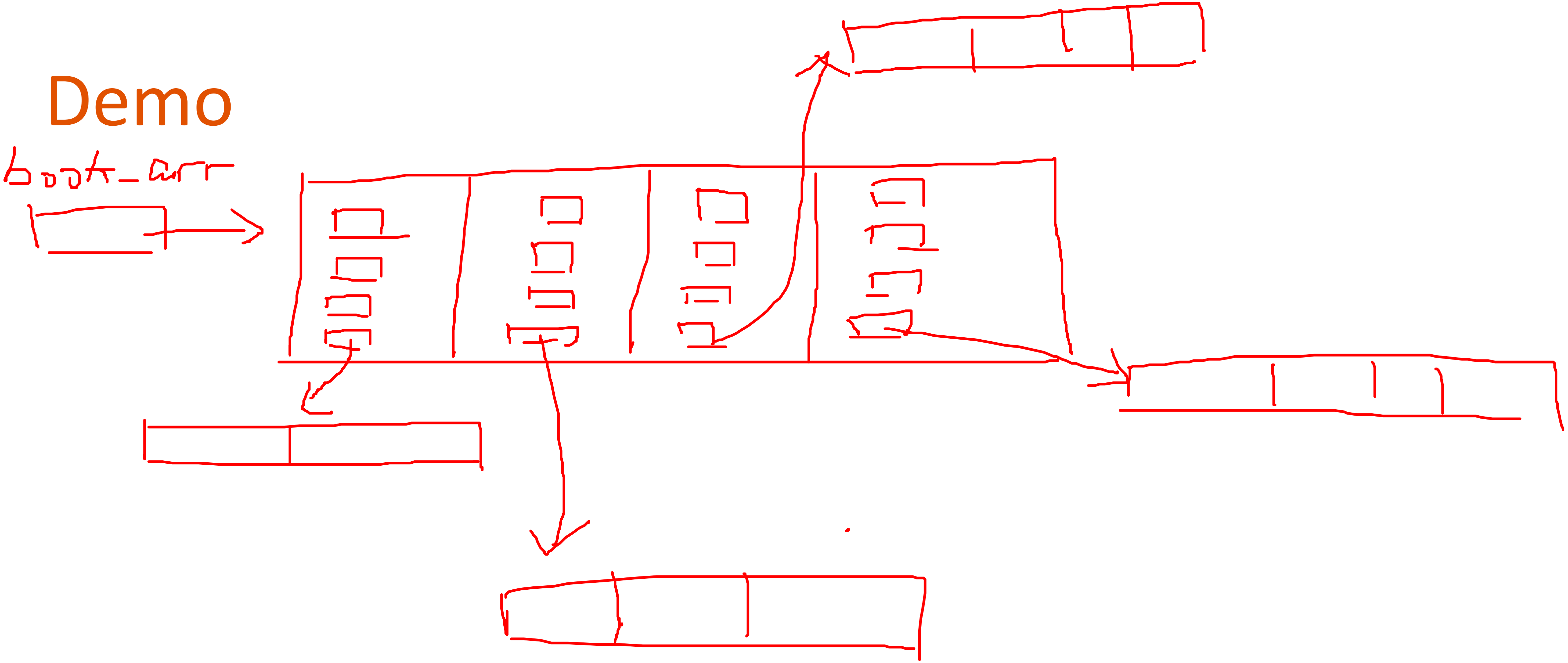
- Can use the same way as any other type
- The dot operator(.) allows us to access the member variables

```
Book bookshelf[10];
for (int i = 0; i < 10; ++i){
      bookshelf[i].num_pages = 100;
      bookshelf[i].title = "Harry Potter";
      bookshelf[i].num_authors = 2;
      bookshelf[i].authors = new string[2];
}
```

Using pointers with structs

```
Book bk1; //statically allocated
Book* bk_ptr = &bk1;

//dereference the pointer and access the data member
(*bk_ptr).title = “Harry Potter”;

//a shortcut to dereference the pointer to the struct
// the arrow (->) operator
bk_ptr -> title = “The Cars”;
bk_ptr -> num_pages = 259;

//this works for objects on the heap as well
Book* bk_ptr2 = new Book;
bk_ptr2 -> title = “Transformers”;
```

Demo



Today's Topics:

- File Separation
    - Header Guard
- Compilation and Makefile
- Begin File I/O

Why do we separate files?

- Programs can get very large, making them difficult to navigate
  - Real-life code bases can be millions of lines. Imagine how hard writing code in a 1 million-line file would be…
- Reuse functions in many applications
- Compiling code can take a long time
  - The time increases as the code grows

How do we separate files?

- Different ways to separate files
  - By classes
  - By common functionality
- Different file types
  - Interface file (.h): description of all reusable parts
    - Prototypes for reusable functions
    - Struct (and later, class) definitions
    - Important constant values
  - Implementation file (.cpp): actual implementation of the interface
    - Definitions of functions (function body) for all prototypes in corresponding .h
  - Driver file (.cpp): the part that you execute to accomplish some specific goal
    - Where `main()` lives with all relevant libraries included

File Separation Demo

Food for thought …

- What happens if you try to define the same variable or struct more than once?

```
int global_var;
char global_var;

int main() {
        // do something
        return 0;
}
```

```
./duplicate.cpp:2:6: error: conflicting
declaration 'char global_var'
 char global_var;
      ^
./duplicate.cpp:1:5: error: 'global_var' has
a previous declaration as 'int global_var'
 int global_var;
     ^
```

When could this happen?

- Suppose that the `book` structure is defined inside a header file: `book.h`
  - Imagine that the `book.h` file is included in the main file
  - Now suppose we include another file `collections.h` which in turn includes `book.h`

```
// book.h
struct book {
    int pages;
    string title;
    int num_authors;
    string* authors;
};
```

```
// main.cpp
#include "book.h"
#include "collections.h"

int main() {
  return 0;
}
```

```
// collections.h
#include "book.h"
#include "movie.h"

// other code...
// perhaps something
// that relies on the
// book struct
```

When could this happen?

- Suppose that the `book` structure is defined inside a header file: `book.h`
  - Imagine that the `book.h` file is included in the main file
  - Now suppose we include another file `collections.h` which in turn includes `book.h`

```
$ g++ ./main.cpp

In file included from ./collections.h:2:0,
                 from ./main.cpp:6:
./book.h:1:8: error: redefinition of 'struct book'
 struct book {
        ^
In file included from ./main.cpp:5:0:
./book.h:1:8: error: previous definition of 'struct book'
 struct book {
        ^
```

How to avoid this problem?

- Use Header Guards
  - Conditional preprocessor directives
    - Recall that these lines starting with “#”
  - This strategy is standard in header files (.h)

```
// book.h
#ifndef BOOK_H
#define BOOK_H
struct book {
    int pages;
    string title;
    int num_authors;
    string* authors;
};
#endif
```

Today's Topics:

- Compilation and Makefile

Compilation

- Process of compilation
  - ***Preprocessing***: expands all preprocessors like `#include`, `#define`, `#ifndef`, etc. into pure C++ code
  - ***Compilation***: parses the pure C++ code into assembly code
  - ***Assembly***: translates the assembly code into machine code
    - Object files produced
  - ***Linking***: link all of the object files produced by the assembler and produce the final output of compilation, which is often an executable file

*Happen behind the scene when you run `g++`

Compilation – can be interrupted

- Very useful when interrupting after assembly but before linking
  - Produce one or more object files but no executable
  - How? Add `–c` option, e.g:

    ```
    g++ -c book.cpp
    ```

    - This would produce an object file, `book.o`, if no syntax errors in book.cpp

- Benefits of stopping before linking
  - Only compile a subset of your program (files that have changed)
    - The rest of your program doesn't need to be re-compiled
  - Greatly speed up the whole compilation process
  - Help debugging
    - Tell if that is a linking issue or a syntax error

In real practice…

- Suppose we have a program that's factored into the following files:
  - Interface/implementation:
    - `book.h, book.cpp`
    - `bookshelf.h, bookshelf.cpp`
    - `library.h, library.cpp`
  - Driver:
    - `prog.cpp`
- Preprocess, compile, and assemble all implementation files into object files

```
g++ -c book.cpp
g++ -c bookshelf.cpp
g++ -c library.cpp
```

- Produce executable by compiling the driver and linking it together with the object files produced by the previous step:

```
g++ prog.cpp book.o bookshelf.o library.o -o prog
```

In real practice… (cont.)

- Find a bug in `book.cpp`. Make changes to that file and recompile it, stopping before linking:

```
g++ -c book.cpp
```

- Recompile the driver and link it with the new `book.o` and all of the old object files:

```
g++ prog.cpp book.o bookshelf.o library.o –o prog
```

- This ends up skipping the compilation process on the rest of our implementation files → SAVES TIME!!!

- But need a lot of different `g++` commands to compile our program…

Makefile

- Make – A Unix utility helps automate the entire compilation process
  - Relies on a specification file: makefile
- A makefile may have multiple rules/commands, each of which consists of 3 things:
  - ***Target***: the output file it is producing
  - ***Dependencies***: components (files or other targets) this particular target depends
    - Optional
  - ***Commands***: specify how to transform the dependencies into the target (e.g. g++ calls)
- General structure:

```
target: dependency dependency …
        command
```

- Note: The commands for a target are only run if one (or more) of the dependencies has been modified
  - Files that haven't changed won't be recompiled

Makefile (cont.)

- A basic `makefile` for our project above might look like this:

```
prog: prog.cpp book.o bookshelf.o library.o
      g++ prog.cpp book.o bookshelf.o library.o –o prog
book.o: book.cpp book.h
      g++ -c book.cpp
bookshelf.o: bookshelf.cpp bookshelf.h
      g++ -c bookshelf.cpp
library.o: library.cpp library.h
      g++ -c library.cpp
```

To run the whole compilation, simply type: **`make`**

More makefile

- Other things we can do in makefile:
  - Use variables to make it easier to control
  - Add a target to clean up our working directory

```
CC=g++
exe_file=prog
$(exe_file): prog.cpp book.o bookshelf.o library.o
		$(CC) prog.cpp book.o bookshelf.o library.o -o $(exe_file)
book.o: book.cpp book.h
		$(CC) -c book.cpp
bookshelf.o: bookshelf.cpp bookshelf.h
		$(CC) -c bookshelf.cpp
library.o: library.cpp library.h
		$(CC) -c library.cpp

clean:
		rm -f *.o $(exe_file)
```

Makefile Demo…

Advanced makefile:

- Recall: How to compile our code with GDB (GNU Debugger)?
  - Add **–g** flag, i.e. `g++ -c struct.cpp -g`

- How to incorporate this into our makefile?

```
CC = g++
exe_file = prog

$(exe_file): prog.cpp struct.o
    $(CC) prog.cpp struct.o -o $(exe_file)
struct.o: struct.cpp struct.h
    $(CC) -c struct.cpp

clean:
    rm -f *.o $(exe_file)
```