CS 162:
File Separation, Makefiles, and I/O

Sometimes it’s helpful to separate files

- Programs can get very large making them difficult to navigate
- Makes aspects of the program more portable to other programs
- Different ways to separate files
  - By classes
  - By common functionality
- Different file types
  - Interface file (.h): contains prototypes for all functions
  - Implementation file (.cpp): contains function body for all prototypes in corresponding .h
  - Driver file (.cpp): where main lives with all relevant libraries included

Food for thought…

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

```
int global_var;
char global_var;

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

Food for thought…

- What happens if you try to declare 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**

```
// main.cpp
#include “book.h”
#include “collections.h”

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

```
// collections.cpp
#include “book.h”
#include “video.h”
#include “board_game.h”

int count_items() {
   // ...
   return sum;
}
```

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

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**

```
goinsj$ 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 {
        ^
```

Header Guards

- Uses conditional preprocessor directives to avoid the problem
  - Recall that these are lines starting with "**#**"
- This strategy is standard in header files

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

Makefile

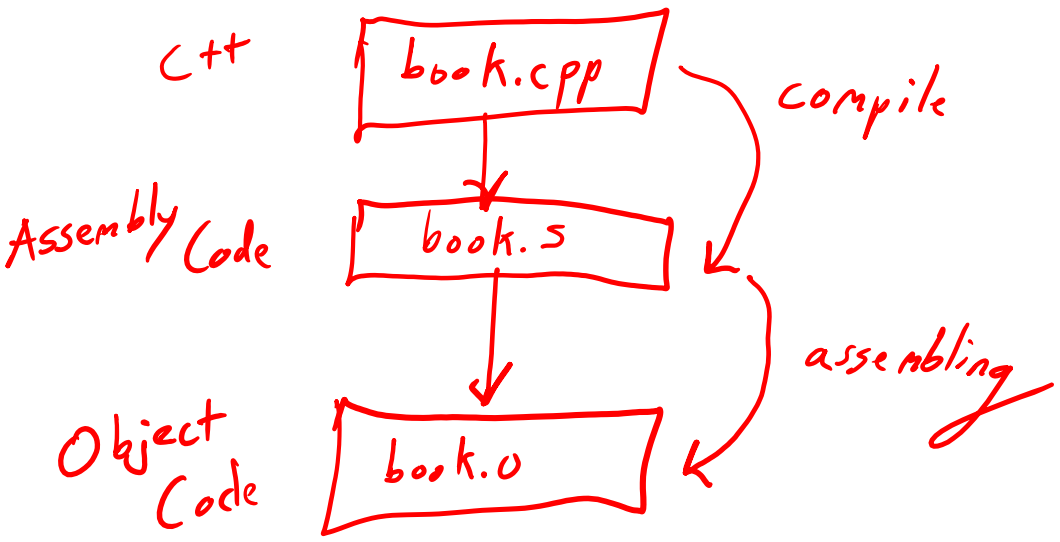
- A special Unix utility… **make**
- Executes the shell commands in the Makefile
- Can have multiple rules/commands
- Divided into sections that look like this:

```
targetfile: prerequisite files
	Commands to generate the target file
	Another command…
```

- More info: https://opensource.com/article/18/8/what-how-makefile

Compilation Process

- Preprocessor/Expand Macros
- Compile C/Translate to Assembly
- Run Assembler/Translate to Machine
- Run Linker/Translate to Executables



Perks to using a Makefile

- All parts of the project can be compiled by typing a single command
  - You could do this by hand, but it involves too much typing

- We can save time by only compiling updated files
  - Imagine our project includes 10 files, but we only modified 1 of them
  - There’s no need to recompile all 10 files!

- Very common to see Makefiles included with professional code
  - i.e. Download a C++ project from GitHub, it probably includes some sort of Makefile

File I/O

- File input output
- Allows us to read and write data to a variety of files for long term storage
- General algorithm
  - Create file object
  - Open the file
  - Perform action on the file (read/write/etc)
  - Close the file

File Stream Objects

```
#include <fstream> // input output file stream class
using namespace std;
int main() {
     fstream f; // create a file stream object
     ifstream fin; // create an input-only file stream
     ofstream fout; // create an output-only file stream

     return 0;
}
```

Open the file

```
int main() {
        fstream f; //create the object
        f.open("file.txt", ios::app); // open(const char* filename, mode)
        return 0;
}
```

- Modes (default is input & output for fstream):
  - ios::in -> input: file open for reading
  - ios::out -> output: file open for writing
  - ios::binary -> binary: operations are performed in binary mode
  - ios::ate -> at end: output position starts at the end of the file
  - ios::app -> append: all output operations happen at the end of the file, appending to existing contents
  - ios::trunc -> truncate: existing file contents are discarded

Open the file

```
int main() {
        fstream f; //create the object
        f.open("file.txt", ios::app); // open(const char* filename, mode)
        return 0;
}
```

- Modes can be combined using the bitwise OR operator

  f.open("file.txt", ios::out | ios::app)

- Not all combination of modes are valid

  e.g. append and truncate

Warnings about opening files

- If there is already a file open in the stream it will not open another file
  - Check if the stream has a file open using is_open()

```
f.open("some_file.txt");
if (f.is_open()) {
    // perform operations
} else {
    cout << "Error opening file" << endl;
}
```

- The file may fail to open if you request an invalid combination of modes

Perform Action on the File

- Reading from a file…

```
int num = 0;
fstream f;
f.open("numbers.txt");
f >> num;
// can read the entire file by doing a while(!f.eof())
// (eof == end of file)
// read a single character with get(), read a line with getline()
```

- Writing (Caution: know where the cursor is in the file)

```
fstream f;
f.open("an_awesome_story.txt");
f << "Once upon a time…" << endl;
```

Close the file

- Good practice to close the file when you are done:

```
my_file_obj.close();
```