

# CS 162

# Intro to Computer Science II

Lecture 9

Compilation & Makefile

Begin File Input/Output

2/2/24



**Oregon State**  
University

# Odds and Ends

- Due Sunday midnight:
  - Design 2 doc
  - Design 2 ex.
  - Quiz 2 (unlock after today's lecture)

# How do we separate files?

- Different ways to separate files
  - ✓ • By classes ✓ *زبطه*.
  - 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

# 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 & Makefile
- Begin File I/O

# 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 ..  
[TAB] 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++ -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
```

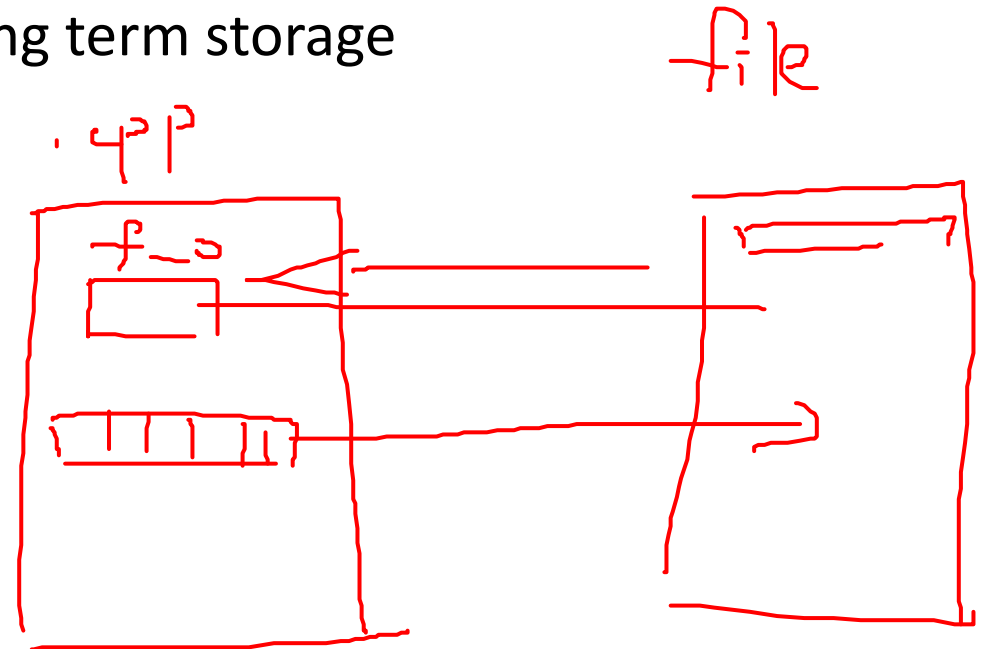
```
clean:  
    rm -f *.o $(exe_file)
```

# Today's Topics:

- File I/O Demo
- Intro to OOP

# File I/O

- File input output
- Allows us to read and write data to files for long term storage
- General algorithm
  1. Create file object
  2. Open the file
  3. Perform action on the file (read/write/etc.)
  4. 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 the 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

# Warning 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()` or with `fail()`

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

# Perform Action on the File

- Reading (Precondition: the file is not empty)

```
int num = 0;
ifstream 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)

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

# Close the file

- Don't forget to do this when you are done with the file  
`my_file_obj.close();`

# File Input – Using “space” as delimiter

```
ifstream fin;
fin.open ("book.txt");
if (!fin.is_open())
    return 1;
while (!fin.eof()){
    string tmp_string;
    int tmp_int;
    // read non-blank characters;
    fin >> tmp_string >> tmp_int;
    cout << "Text: " << tmp_string << endl;
    cout << "Int: " << tmp_int << endl;
}
fin.close();
```

# File Input Strategies

- What if the input file does not delineate text with spaces?
  - E.g. “student\_name,grade,gpa”
- **getline(cin, dest\_string);**
  - Reads an entire line at once
  - Previously used this when accepting user input from the console
- **getline(cin, dest\_string, ',');**
  - Keeps reading text until reaching the specified char
  - Discards the specified char
  - Can be used to handle an alternate delimiter (e.g. comma)



# The Newline Character

- Most user-readable files use newlines
  - Makes the text much easier to read
- Often used to indicate “new entry”
  - Make sure that your code handles these correctly
- Hint: Use [std::istream::ignore\(\)](#)
  - Discards one or more characters from the input stream
  - Useful for discarding newline characters
  - Common usage: `cin.ignore()` → throw away the next char

# File Output

- You control the delimiters, newlines, etc.
- Easier to handle

```
string output_file = "book_stats.txt";
ofstream fout;
fout.open (output_file.c_str(), ios::app);
if (!fout.is_open()){
    cout << "Error, unable to open the file!" << endl;
    return 1;
}
fout << "Hello world!" << endl;
fout.close();
```

# File I/O Demo