CSCE 121

Introduction to Program Design & Concepts

Design – Code in Multiple Files

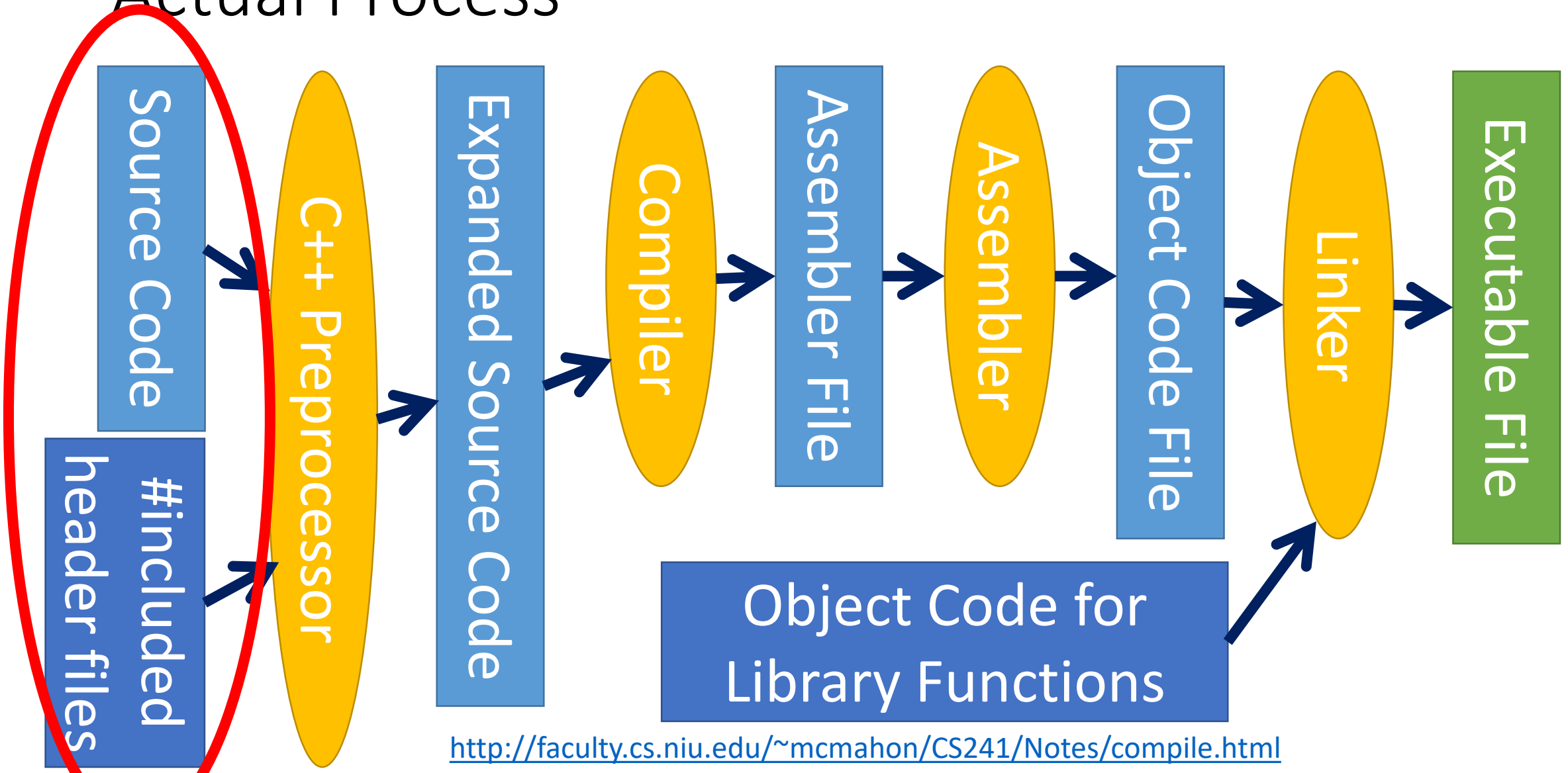
Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

Compilation Process

- Starting with source code (e.g. C++) and converting it into machine code that the computer can run.
- The process looks like this:

Actual Process



http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Functions in Separate External Files

Functions in a Separate C++ File

About

- The program creates a header file, `helper.h`, which contains declarations such as function prototypes, and a C++ file, `helper.cpp`, which contains function definitions.
- Note that it is considered bad programming practice to include implementation details -- such as function definitions -- in header files

main.cpp

```cpp
#include <iostream>
#include "helper.h"
using std::cout, std::endl;

int main() {
    cout << std::boolalpha;
    cout << "2 < 5?  " << lessThan(2, 5) << endl;
    cout << "2 == 5? " << equalTo(2, 5)  << endl;
    cout << "5 < 5?  " << lessThan(5, 5) << endl;
    cout << "5 == 5? " << equalTo(5, 5)  << endl;
    cout << "7 < 5?  " << lessThan(7, 5) << endl;
    cout << "7 == 5? " << equalTo(7, 5)  << endl;
    return 0;
}
```

helper.cpp

```cpp
bool lessThan(int left, int right) {
    return left < right;
}

bool equalTo(int left, int right) {
    return left == right;
}
```

helper.h

```cpp
bool lessThan(int left, int right);
bool equalTo(int left, int right);
```

Functions in a Separate C++ File

main.cpp

Line 2. A new **include** statement is created for including code from the helper.h header file, which is located in the same directory as the current main.cpp file.

- When the program is compiled, this include statement is **replaced** with the entire code from the listed header file helper.h.

Lines 6 to 12. The main method that calls the two functions used in the program multiple times.

main.cpp

```
1  #include <iostream>
2  #include "helper.h"
3  using std::cout, std::endl;
4
5  int main() {
6      cout << std::boolalpha;
7      cout << "2 < 5?  " << lessThan(2, 5) << endl;
8      cout << "2 == 5? " << equalTo(2, 5)  << endl;
9      cout << "5 < 5?  " << lessThan(5, 5) << endl;
10     cout << "5 == 5? " << equalTo(5, 5)  << endl;
11     cout << "7 < 5?  " << lessThan(7, 5) << endl;
12     cout << "7 == 5? " << equalTo(7, 5)  << endl;
13     return 0;
14 }
```

helper.cpp

```
1  bool lessThan(int left, int right) {
2      return left < right;
3  }
4
5  bool equalTo(int left, int right) {
6      return left == right;
7  }
```

helper.h

```
1  bool lessThan(int left, int right);
2  bool equalTo(int left, int right);
```

Functions in a Separate C++ File

helper.cpp

Lines 1 to 7. The function definitions.

- **Note** that since the program from the main file now executes code from this separate C++ file, the compile command in the console must now be updated to also compile this file:
  `g++ -std=c++17 main.cpp helper.cpp`

main.cpp

```
1  #include <iostream>
2  #include "helper.h"
3  using std::cout, std::endl;
4
5  int main() {
6      cout << std::boolalpha;
7      cout << "2 < 5?  " << lessThan(2, 5) << endl;
8      cout << "2 == 5? " << equalTo(2, 5)  << endl;
9      cout << "5 < 5?  " << lessThan(5, 5) << endl;
10     cout << "5 == 5? " << equalTo(5, 5)  << endl;
11     cout << "7 < 5?  " << lessThan(7, 5) << endl;
12     cout << "7 == 5? " << equalTo(7, 5)  << endl;
13     return 0;
14 }
```

helper.cpp

```
1  bool lessThan(int left, int right) {
2      return left < right;
3  }
4
5  bool equalTo(int left, int right) {
6      return left == right;
7  }
```

helper.h

```
1  bool lessThan(int left, int right);
2  bool equalTo(int left, int right);
```

Separate Declaration from Definition

```
// functions.h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

int gcd(int, int);
int factorial(int);
int mod_pow(int, int, int);



#endif
```

```
// functions.cpp
#include “functions.h”

int gcd(int a, int b) {
     …
}

int factorial(int n) {
     …
}


…
```

Group Related Functions Together

```
// math functions
#ifndef MATH_H
#define MATH_H

…
double exp(double);
double log(double);
double pow(double, double);
double sqrt(double);
…

#endif
```

```
// string functions
#ifndef STRING_H
#define STRING_H

…
int find(string, string);
int length(string);
string lower(string);
string upper(string);
string reverse(string);
…

#endif
```

Header Guards

```
#ifndef NAME_OF_FILE_H
#define NAME_OF_FILE_H

…

#endif
```

- Prevents double inclusion: inclusion of same header file multiple times
  - Helps prevent linker error due to multiple definitions (re-definition)
- `#ifndef`
  - pre-processor directive “if not defined”

Including and Compiling

```
// source1.cpp
#include <c++_library>
#include <c_library.h>
#include
“user_defined_library.h”
…
```

```
// source2.cpp
#include
<iostream>
…
int main() {
      …
}
```

```
$ g++ source1.cpp source2.cpp source3.cpp
```

Data Types

Data types in C++

- Built-in / Primitive
- Derived
- User-defined

Built-in / Primitive data types

- Integer types
  - bool, char, short, int, long
- Floating point types
  - float, double, long double
- Void type
  - void

Derived data types

- Pointer
  - memory address and a type
  - void pointer is just address, no type
  - size depends on architecture (e.g. 32- or 64-bit)
- Array
  - contiguous segment of memory storing a single (homogenous) data type
  - represented by a pointer to the first (0th) element
- Function
  - block of reusable code with a name, arguments, and a return value
  - represented by a pointer to the first instruction

User-defined data types

- struct / class
  - composite of heterogenous data types
  - In C++: struct and class only differ on default access level
- union
  - composite of heterogenous data types (similar to struct / class)
  - only has the value of one of it’s members at a time
- enum
  - enumeration mapping identifiers (names) to integer values.

Structs

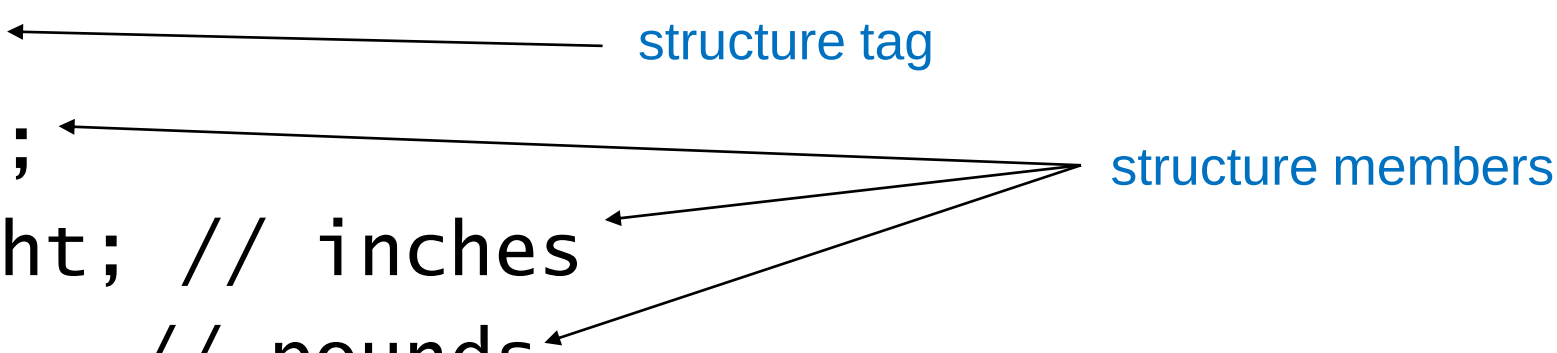
A struct is a heterogeneous aggregate data type

- A “bundle” of different types of data rolled into a new type
  - struct StructName
- Each instance of the struct is stored in a block of memory large enough to hold all the fields

name	string	“Andre the Giant”
height	float	84.3333
weight	int	520

Declaring and defining structs

```
struct Person {
    string name;
    double height; // inches
    int weight;   // pounds
};
```



This declares the `struct Person` and defines it to have 3 members.

It does not allocate any space.

This is just the blueprint for instances of `struct Person`.

`struct` Declaration Notes

- Must have `;` after closing `}`
- **`struct`** names commonly begin with uppercase letter
- Multiple fields of same type can be in comma-separated list:

```
string name, address;
```

- Use the dot `(.)` operator to refer to members of **`struct`** variables
- Member variables can be used in any manner appropriate for their data type

Initializing structs

```
// declaring and initializing in one line
struct Person andre = {“Andre the Giant”,
84+1./3, 520};

struct Person andre;
// initialize fields individually
andre.name = “Andre the Giant”;
andre.height = 84+1./3;
andre.weight = 520;
```

Accessing struct members

```
int weight = andre.weight; // get member value
andre.weight = 520; // set member value


// compiler error
andre.age = 46; // no member named “age”
```

Using typedef to define a type

```
typedef struct Person {
    string name;
    float height; // inches
    int weight;   // pounds
} Person;
```

Renames “struct Person” to “Person”

```
Person andre = {“Andre the Giant”, 84+1./3, 520};
```