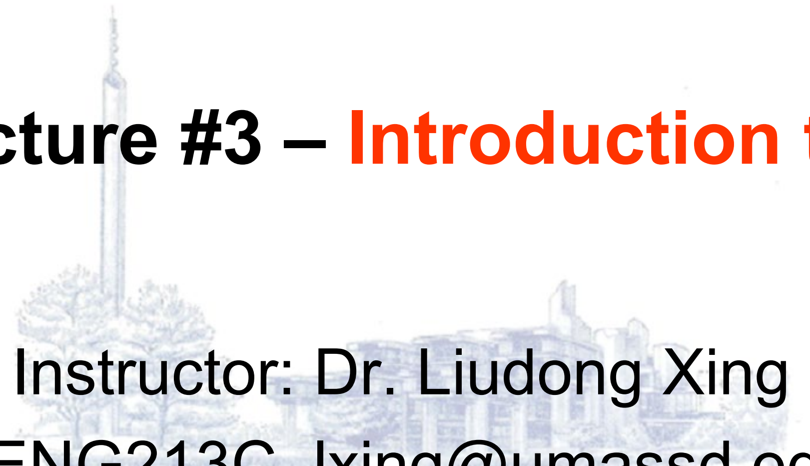
UNIVERSITY OF MASSACHUSETTS
DARTMOUTH

ECE160: Foundations of Computer Engineering I



Lecture #3 – Introduction to C

Instructor: Dr. Liudong Xing
SENG213C, lxing@umassd.edu
ECE Dept.

Administrative Issues

- The first lab assigned
  - Lab L1: Monday (1/23) 10-11:50am
  - Lab L2: Wednesday (1/25) 10-11:50am
  - Due by **5pm, Wednesday, Jan. 25**
  - TA (Lab assistant & Grader): Guixiang Lyu <glv@umassd.edu>
  - Lab assistant: Hailey Williams <hwilliams3@umassd.edu>

- Homework #1 assigned
  - Due **9am, Monday, Jan. 30**

Go to http://xing160.sites.umassd.edu/

Review of Lecture #2

- Basic concepts of number systems
  - Base, positional value, symbol value
  - Binary, decimal, octal, hexadecimal
- Number systems conversions
  - Binary, Octal, Hex ←→ Decimal
  - Binary ←→Hex, Binary ←→ Octal, Hex ←→ Octal

Topics

- Definitions and conventions
- Computer languages
- Your first C program
- Software development lifecycle

Definitions

Term	Definition
bit	0 or 1
byte (B)	a group of 8 bits
nibble (nybble)	half a byte (4 bits)
word (w)	a group of bits that is processed simultaneously. a word may consist of 8/16/32/other number of bits machine dependent (ex: 8086 – 16 bits; 80386/80486/Pentium – 32 bits)
double word	2 words
msb (most significant bit)	the leftmost bit in a word
lsb (least significant bit)	the rightmost bit in a word
Hz (hertz)	reciprocal of second

Conventions

Term	Normal Usage	Usage as a Power of 2
Kilo (K)	10^{3}	2^{10} =1,024
Mega (M)	10^{6}	2^{20} =1,048,576
Giga (G)	10^{9}	2^{30} =1,073,741,824
Tera (T)	10^{12}	2^{40} =1,099,511,627,776
Mili (m)	10^{-3}	
Micro (m)	10^{-6}	
Nano (n)	10^{-9}	
Pico (p)	10^{-12}	

- Powers of 2 are most often used in describing memory capacity.
  - Ex: 1Kilobyte (KB) =1024 bytes= 2^{10} bytes
- Powers of 10 are used to describe the CPU clock frequencies: cycles per second (Hz)
  - Ex: Pentium 4 --1.8GHz = 1.8×10^{9} Hz

Computer Language Evolution

- Machine languages
- Symbolic/assembly languages
- High-level languages

Machine Languages

- Machine dependent
- Binary-based code
  - made of streams of 0s and 1s
- The only language understood by computers
- Example of a machine language instruction:

00000101	**00010000**	**00000000**
ADD operation	Value of 1st operand	Address to store result

Low-Level Programming Languages: Machine Languages

Do you understand what this program does?

A sample machine language program:

```
10111000 00000101 00000000
00000101 00010000 00000000
00000101 00100000 00000000
10100011 00000000 00000001
```

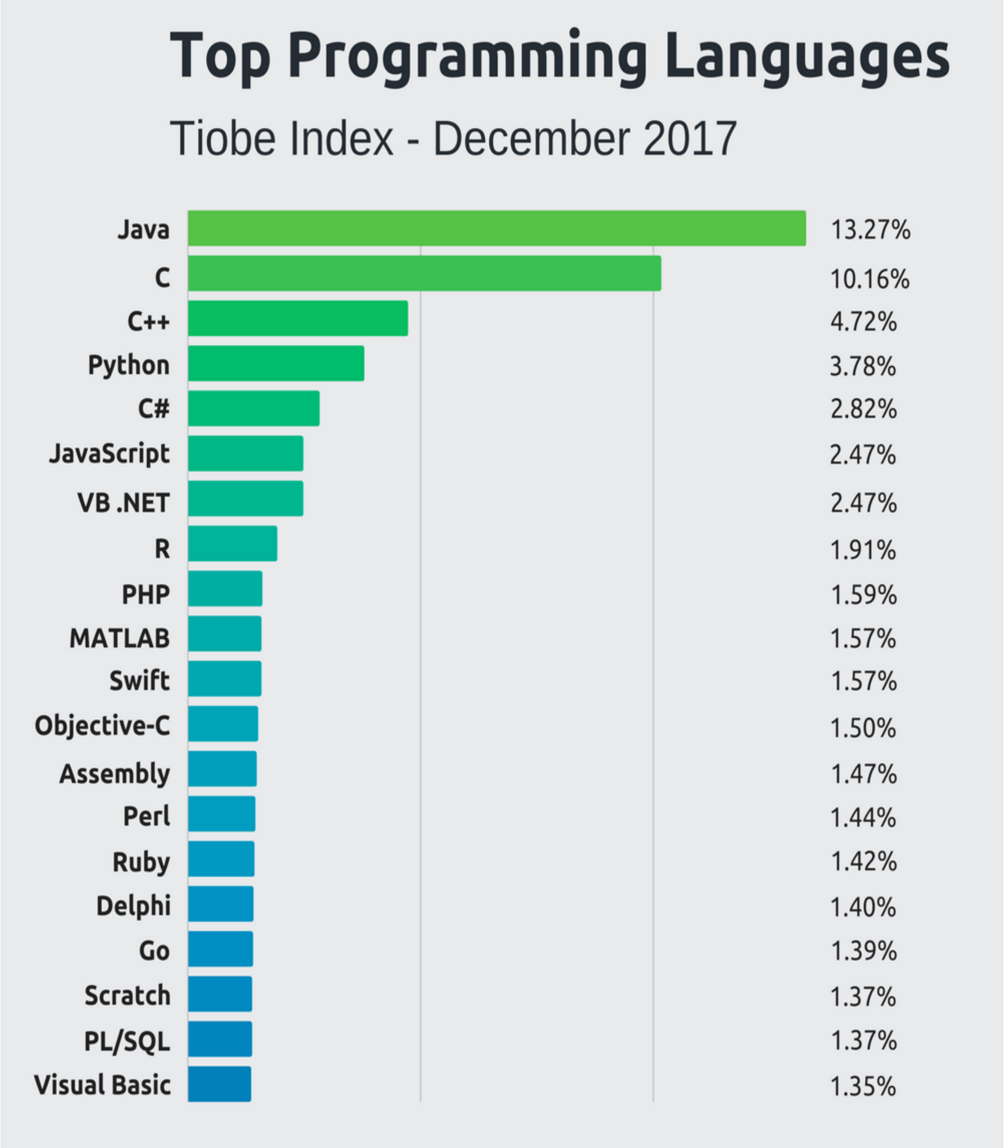
Assembly Languages

- Machine dependent
- Numbers, symbols, and abbreviations are used
- Example of an assembly language instruction:

```
MOV AL, 61h;
// load register AL with 61 in hexadecimal
```

High-Level Programming Languages

- Generally, machine-independent
- Usually, several machine instructions are combined into one high-level instruction.
- Examples:
  - **C**, C++, LISP, JAVA



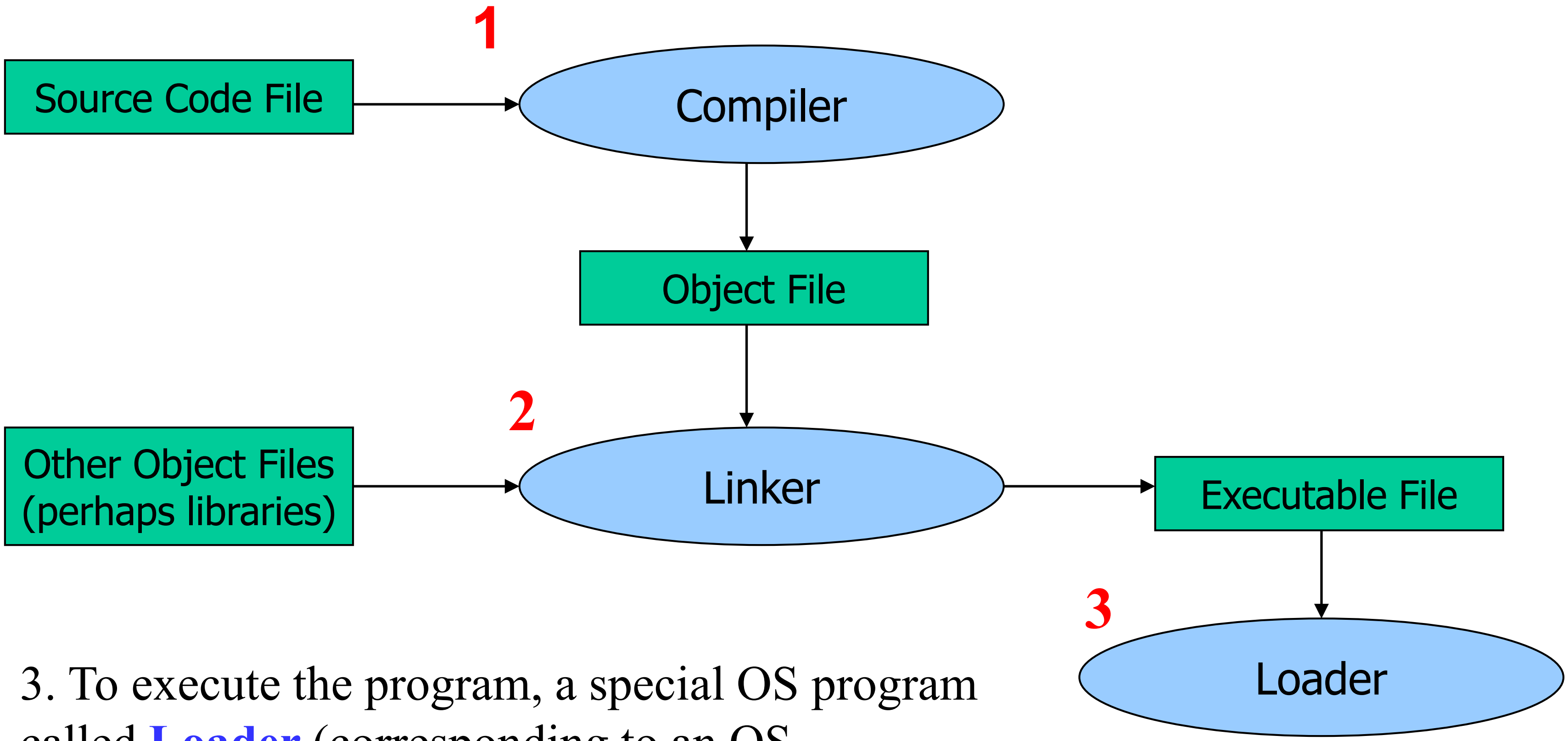
Compilers and Linkers

Starting from C source code, two steps in creating an executable program

1. A program called the **C compiler** translates the C code into an equivalent program (object file) in the processor's machine language (1's and 0's)
2. A program called the **linker** combines this translated program with any library files it references (e.g., printf, scanf) to produce an executable machine language program (.exe file)

Environments like **Visual Studio** do both steps when you "build" the program

Modern Software Development



3. To execute the program, a special OS program called **Loader** (corresponding to an OS command, e.g., run) is used to load the program into the main memory and execute it.

Topics

- ✓ Definitions and conventions
- ✓ Computer languages
- **Your first C program**
- Software development lifecycle

Your First C Program

```
/* The first C program
learned in ECE160 */

#include <stdio.h>

void main(void)

{

        printf("Hello world!");

}
```

- Output:

  when executing it, the statement Hello world! appears on the screen

Comments

```
/* The first C program
learned in ECE160 */
#include <stdio.h>
void main(void)
{
        printf(“Hello world!");
}
```

- /* any text, number, or character */
- /* and */ must form a couple, but need not be on the same line
- No blanks between slash and asterisk
- Comments can appear anywhere in the code.
- Comments can not be nested.
  - /* /* bla bla */ */   !!! Illegal.

Preprocessor Directives

```
/* The first C program
learned in ECE160 */

#include <stdio.h>

void main(void)

{

        printf(“Hello world!");

}
```

- Every C program consists of preprocessor directives (commands to the C preprocessor)
  - Start with #

- Preprocessor: the first phase of a C compilation in which the source statements are prepared for the compilation and any necessary libraries are loaded; preparation prior to the translation of C code into machine language instructions

Preprocessor Directives (Cont'd)

- Come at the beginning of the program, telling the preprocessor how to prepare the program for compilation.

- Most important preprocessor command: include
  - Tell the preprocessor that we need information from selected libraries known as header files
  - All header files end in .h

- `#include <stdio.h>` tells preprocessor to attach the stdio.h file to the source file
  - stdio.h: standard input / output functions, e.g. printf

Functions

```
/* The first C program
learned in ECE160 */

#include <stdio.h>

void main(void)

{

        printf(“Hello world!");

}
```

- Every C program consists of: one or more functions. One and only one of the functions of the program must be called **main()**

- Information can be passed from calling function to function being called and vice versa
  - First void: no information is passed from main() to OS
  - Second void inside the parentheses: no information is passed from OS to main(), or main() does not take any arguments
  - A string constant is passed from main() to printf() (*An output function contained in a library file: stdio.h*)

Functions (Cont’d)

- Function starts with an open bracket { and closes with a close bracket }
- The lines enclosed in a pair of {} are called a block of code
- You can define your own functions (later lectures)

C is case sensitive!

- printf is different from PRINTF, Printf
- Traditionally C is written primarily in lowercase letters: main, printf
- You may use whatever case when naming your self-developed functions

Identifiers

- Identifiers are used to name data and other objects (e.g. functions) in our program.

- The only valid name symbols are the capital letters A-Z, the lowercase letters a-z, the digits 0-9, and the underscore

- C is case sensitive
  - Celsius, celsius, and CELSIUS are three different identifiers.

Identifier Name Rules

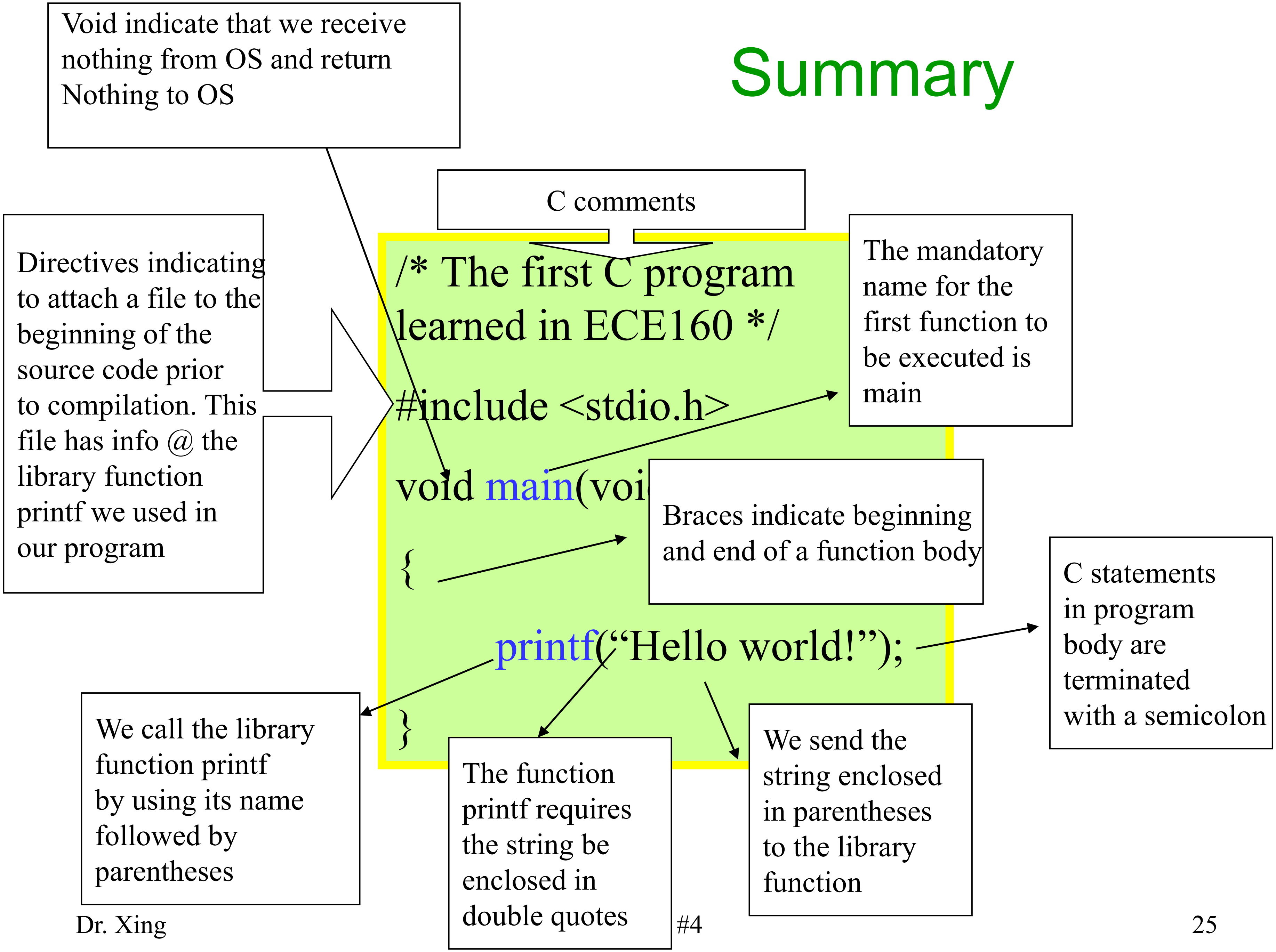
- The first character can not be a digit. It has to be an alphabetic character or underscore.
- The identifier name must consist only of alphabetic characters, digits, or underscores
- First 31 characters of an identifier are significant/used.
- DO NOT use a C reserved word /keywords (e.g., **int**).

Exercises

- Indicate the following names are valid or invalid C names

  Student
  2names
  $sum
  Stud number
  int
  _systemname
  SystemName
  F_3
  f%

Summary



```
/* The first C program
learned in ECE160 */

#include <stdio.h>

void main(voi

{

    printf(“Hello world!”);

}
```

C comments

Void indicate that we receive nothing from OS and return Nothing to OS

Directives indicating to attach a file to the beginning of the source code prior to compilation. This file has info @ the library function printf we used in our program

The mandatory name for the first function to be executed is main

Braces indicate beginning and end of a function body

C statements in program body are terminated with a semicolon

We call the library function printf by using its name followed by parentheses

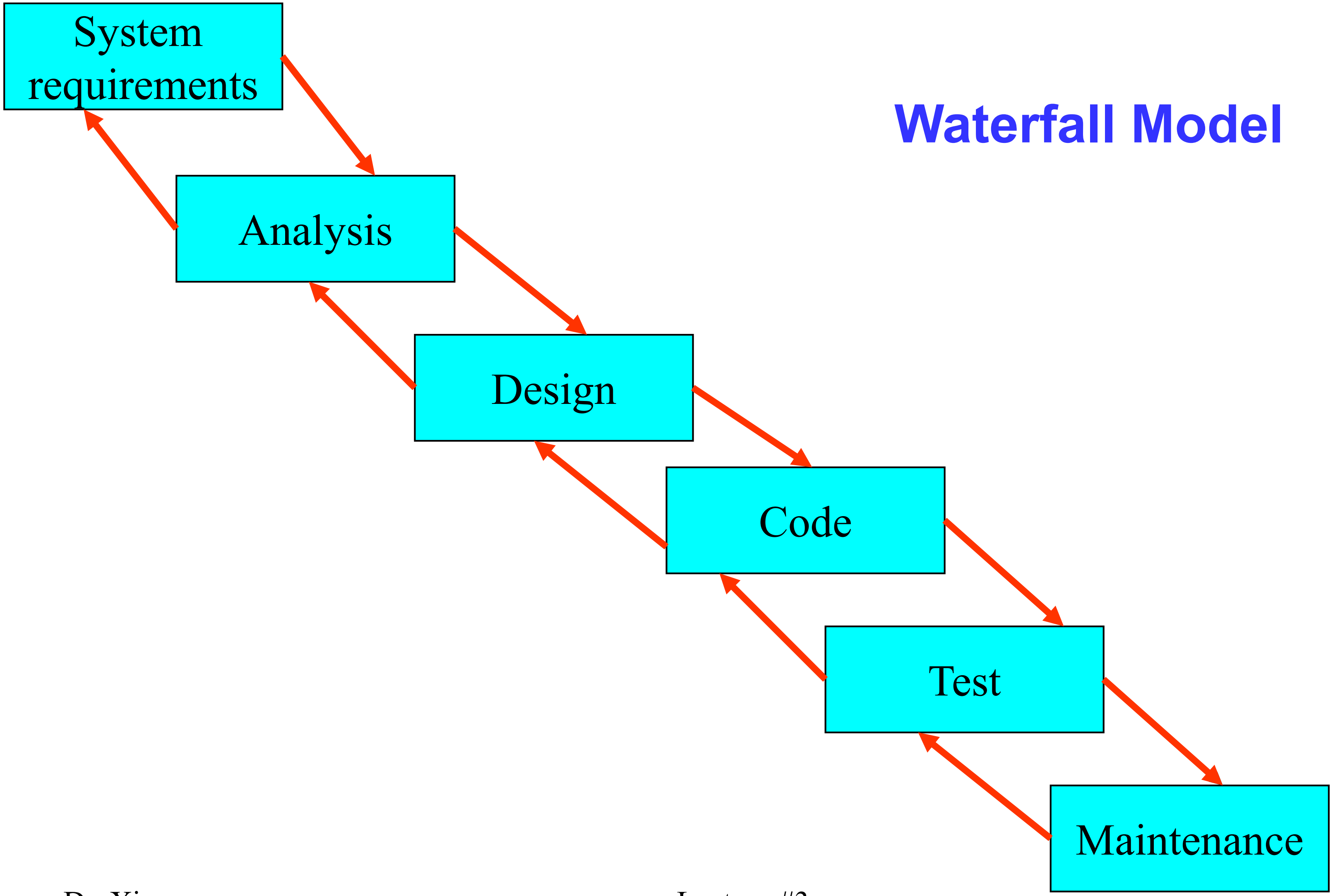
The function printf requires the string be enclosed in double quotes

We send the string enclosed in parentheses to the library function

Topics

- ✓ Definitions and conventions
- ✓ Computer languages
- ✓ Your first C program
- **Software development lifecycle**

Software Development Lifecycle



System requirements

Analysis

Design

Code

Test

Maintenance

Waterfall Model

The Software Development Method

- System requirements
  - Specify the problem, define requirements specifying what the proposed system is to accomplish.
- Analysis
  - Analyze the problem, look at different alternatives from a system point of view
- Design
  - Design an algorithm (a sequence of well-defined computational steps that transform the input into the output) to solve the problem.
- Code
  - Write programs to implement the algorithm.
- Test and verify the program.
- Maintain and update the program.

An Illustrative Example Problem

- Write a program that converts Celsius temperatures to Fahrenheit.

Step 1: System Requirements

- Write a C program that takes as input a Celsius temperature and converts it to Fahrenheit.

Step 2: Analysis

- The input is going to be a real number representing the Celsius temperature.
- The output is going to be a real number representing the Fahrenheit temperature.

Step 3: Design

Natural-Language Algorithm:

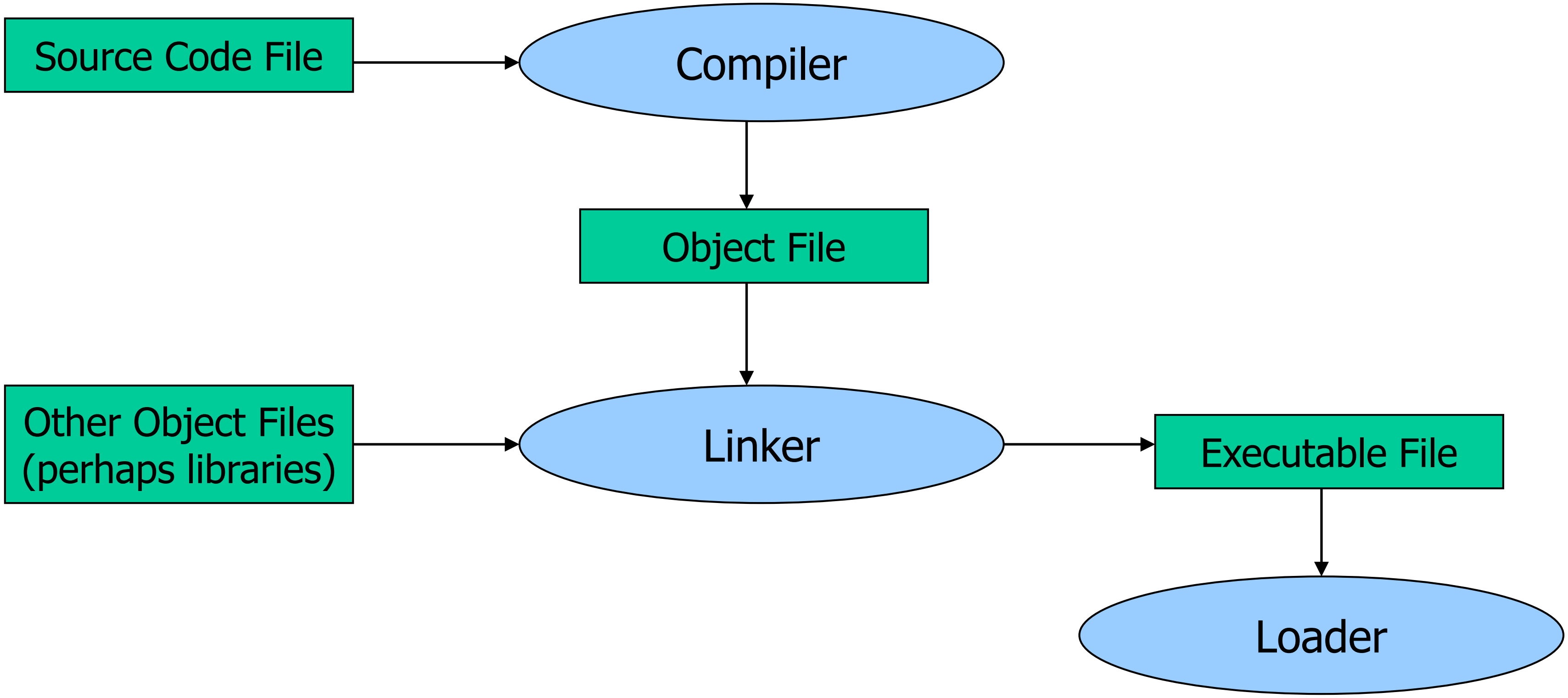
1. Prompt user for the Celsius temperature.
2. Read the Celsius temperature.
3. Store value in storage location called *celsius*.
4. Compute the Fahrenheit temperature by solving the formula “*fahrenheit = (9/5)*celsius+ 32*”
5. Print out the value stored in location *fahrenheit.*

Step 4: Coding using C Programming Language

temperature.c

```
#include <stdio.h>
int main(void) {
float celsius;
float fahrenheit;
printf("This program converts Celsius  to Fahrenheit. \n");
printf("Please enter a Celsius temperature. \n");
scanf("%f", &celsius);
fahrenheit = 9.0/5.0 * celsius + 32;
printf("The temperature in Fahrenheit is: %f\n", fahrenheit);
return 0;}
```

Step 5: Run & Test



Why Testing?

- Many things can go wrong!
  - Things are rarely perfect on the first attempt

- There are two types of errors
  - Syntax: the required form of the program punctuation, keywords (int, float, return, …) etc.
    - **The C compiler always catches these “syntax errors” or “compiler errors”**
  - Semantics (logic): what the program means
    - What you want it to do
    - **The C compiler cannot catch these kinds of errors!**
    - They can be extremely difficult to find

Why Testing? (Cont’d)

- Both the compiler and linker could detect syntax errors
- Even if no errors are detected, logic errors (“**bugs**”) could be lurking in the code
- Getting the logic errors out is a challenge even for professional software developers

Summary of Lecture #3

1. Computer languages evolution: machine → assembly → high-level→ natural (AI)
2. The first C program
    - preprocessor directives
    - main(), printf()
    - comments
3. A popular software development lifecycle – waterfall model
4. Two types of errors: syntax and logic / semantics errors

Things To Do

- The first lab
  - Due by **5pm, Wednesday, Jan. 25**
- Homework #1
  - Due **9am, Monday, Jan. 30**

http://xing160.sites.umassd.edu/

Next Topic

- Data Types and Variables