

Version 1.0

# UCSD CSE 30

Computer Organization and Systems Programming

Class Overview, PA 1 and Introduction to C

Keith Muller

DEC PDP 11/45 - 1973

# CSE 30 Fall 2022 – Staff Introduction

**Instructor:** Keith Muller

- **Office Hours In Person Only: CSE 3109**
  - **Tue, Thu:** 2:00 PM to 3:00 PM
- **Additional Office Hours – Zoom Only**
  - **Wed:** 3:00 PM to 4:30 PM
  - Zoom: <https://ucsd.zoom.us/j/94331007124>
- **Additional Office Hours by Appointment**
  - Email Request to [muller@eng.ucsd.edu](mailto:muller@eng.ucsd.edu)
  - **Do not use canvas messages to contact me!**
- **Lecture is required!**
  - A lot of material is only available in lecture
  - If you miss lecture, see the podcasts

## TA's

- Rasika Bhave
- Jinhao Liu
- Sananya Majumder
- Jerry Yu

## Tutors

Nitya Agarwal; Alexander G Arias; James Bao; Shubham Bhargava; Adrian Botvinik; Tianyi Irene Chen; Melina Kapsogeorgou; Thuan Quang Do; Ruilin Hu; William Hu; Jinya Jiang; Mihir Kekkar; Austin Li; Binghong Li; Frank Li; Gerui Li; Meihui Liu; Rana Lulla; Saransh Malik; Hyunseo Park; Annie Phan; Prashanth Rajan; Adrian Rosing; Jordan Ruggles; Andrew Russell; Jose Salazar; Arjun Sampath; Michael Shao; Xiyan Shao; Catherine Shen; Marcelo Shen; Christian Sulaiman; Timothy Wu; Benjamin Xia; Zenas Zhu

# Quick Overview of Syllabus – See Canvas

## PAs

- 9 assignments – 50 pts each (**450** total)
- Weekly PA's due **Wed night (except Thanksgiving week) starting Sept 28**
- Late submission up to **Saturday** night (-5, -10, -15)
- 8 Slip days for the entire quarter (applied at the end when grades are calculated) to offset Late penalties
- No work accepted after late submission date
- Special circumstances (e.g., extended absence, contact me right away)

Textbooks and References – See canvas

3

All free via library link

## Weekly Canvas Reading quiz

- **50** pts total
- **Due Sunday night**

## Midterm – In Person

- **150** pts
- Oct. 27<sup>th</sup> evening in-person

## Final – In Person

- **350** pts
- Dec. 9<sup>th</sup> 11:30 AM – 2:29 PM

x

# My View of CSE30....

- A little about me...
  - Started as a (BSEE, MSEE) microelectronics worked at Bell Labs in R&D – **got bored**
  - **Changed career/retrained** in systems programming/architecture
    - MS, PhD Computer Science UCSD
  - 45 years as a system (OS) software developer and platform architect
    - At various start ups
    - Ran a consulting firm – Next, Apple, General Dynamics, Loral Space Systems, Citibank, ...
    - Worked at several large Fortune 500 Tech companies in Product R&D
    - Specialized in enterprise storage systems
    - Last 10+ years as a CTO at a Database company – **got bored and retired**
  - Adjunct Professor at UCSD
- My Goals for the Class
  - Complete the Academic Requirements for class but with a focus on making the course more relevant to industry
  - Practice the skills important to a having a successful career
    - PA's better representative of real problems
  - Learn and practice **creativity** by understanding both how and why software/hardware was designed the way it was

## Survival Tips

- Keep up!
  - CSE30 gets hard real fast! Do not expect you can do an assignment in 1-2 days
- Do not be Shy! Learn to ask questions in lecture and discussion sections!
  - Raise your hand, send a chat message or call out if I do not see you
  - I love questions! There are no dumb questions.
- I will go too fast in class! You must tell me to slow down!!!
- Come to office hours: get to know me, give me a chance to help
  - I will spend as much time as necessary to help you understand the material
- Go to Discussion Sessions, do the readings, ask the TA's and Tutors, post to piazza - DO THIS PROMPTLY, DO NOT WAIT....
- I highly encourage feedback: See me for all Complaints, requests, etc.
  - I will never penalize anyone for speaking up, bring your class issues to me I will fix it

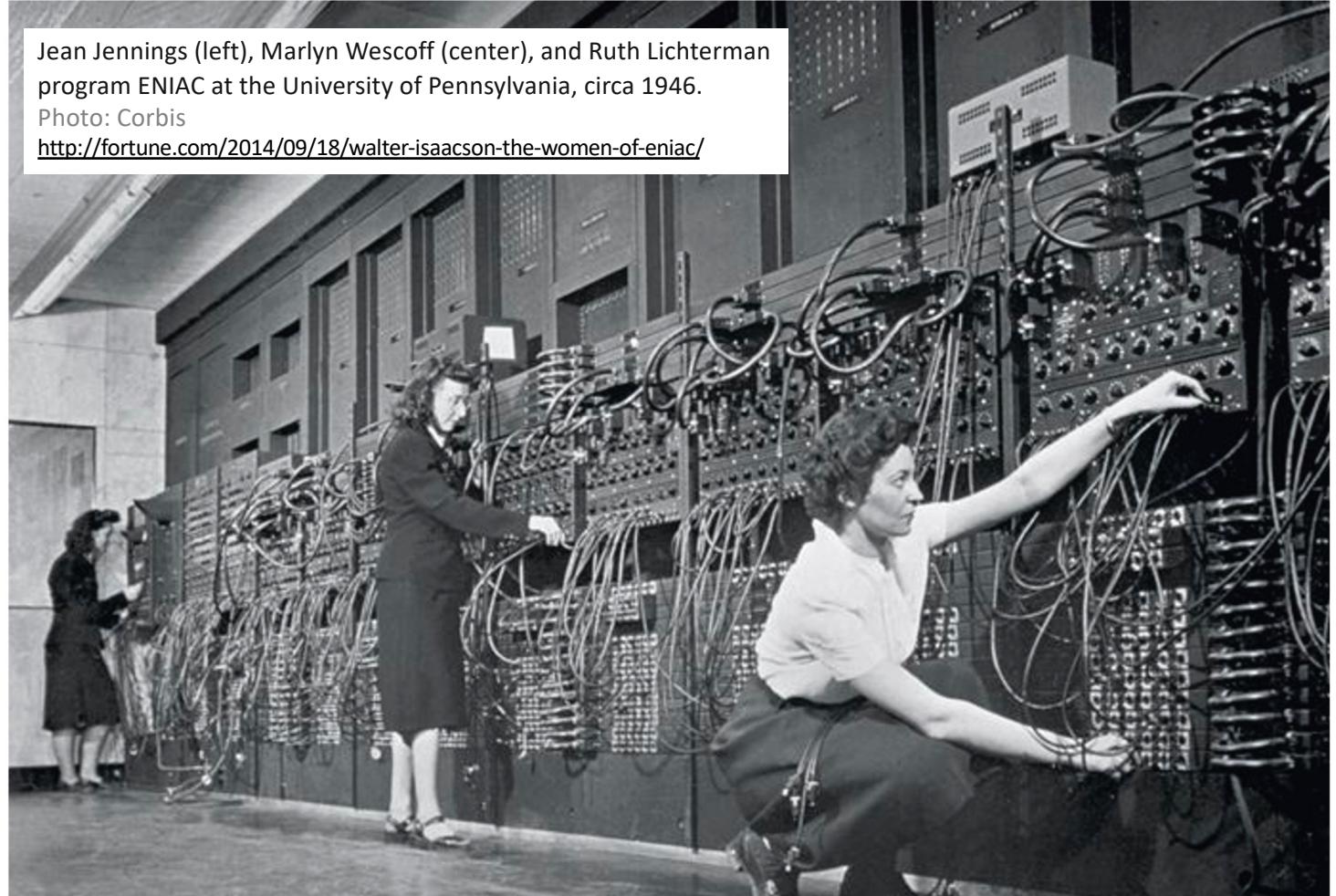
# Additional Information

- **Canvas and Piazza are the primary interfaces for this class**
  - All material, quizzes, assignments, etc., can be found **on canvas**
  - **Piazza for Q/A and announcements**
    - Send me **email** for personal matters (illness, family emergencies etc)
    - Public posts for general questions on PA's and lectures
    - Private posts if you are not sure
    - DO NOT POST YOUR ASSIGNMENTS ON Piazza
- **Lecture Support Materials**
  - Slides are available in **pptx** and **pdf** format on canvas and I am trying github  
[https://github.com/cse30-fa22/Muller\\_CSE30\\_Lecture\\_Slides.git](https://github.com/cse30-fa22/Muller_CSE30_Lecture_Slides.git)
  - Each slide set covers about 1 week (or so) of lectures
  - Slides **are updated constantly** to correct errors and to improve content
    - Version is at the upper left on the title slide
    - Please **check you have the current version** the morning before lecture

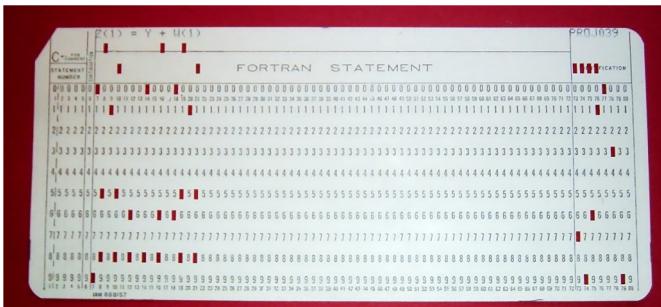
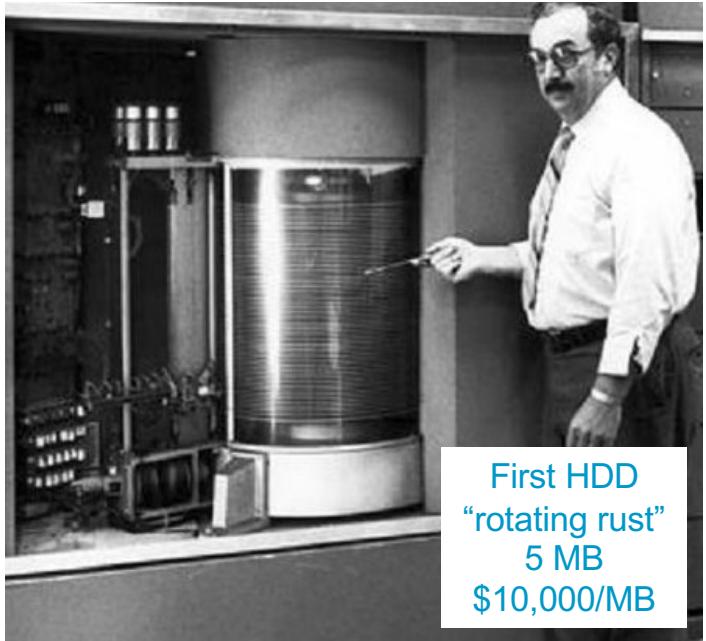
# Back in those Early Years.... Physically Wiring your "Program"

- Early Hardware
  - Used very simple CPU instructions (*primitives*)
  - e.g., a single instruction for adding two integers
- Software simple
  - Closely reflected the actual hardware it was running on
  - Specify each step manually
- How to program:
  - **Physical re-wiring** was a major part of programming

Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946.  
Photo: Corbis  
<http://fortune.com/2014/09/18/walter-isaacson-the-women-of-eniac/>

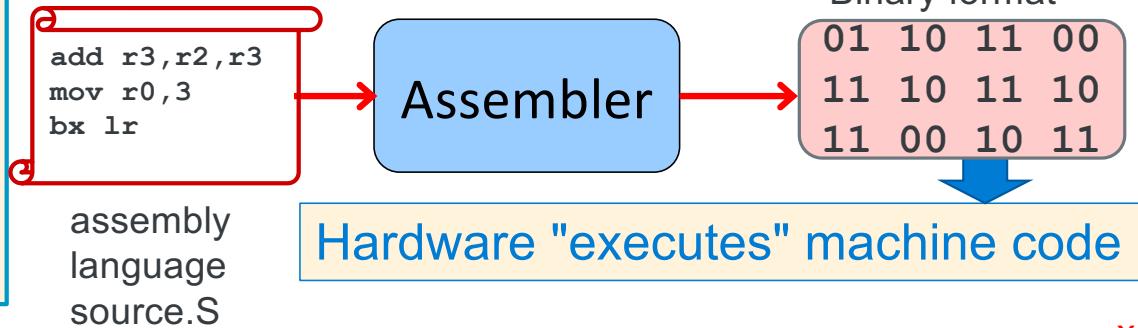


# 1950-1970 Computers – Abstraction Arrived



Code onto Punched cards,  
no back spacing to correct  
typos! I was doomed to  
fail....

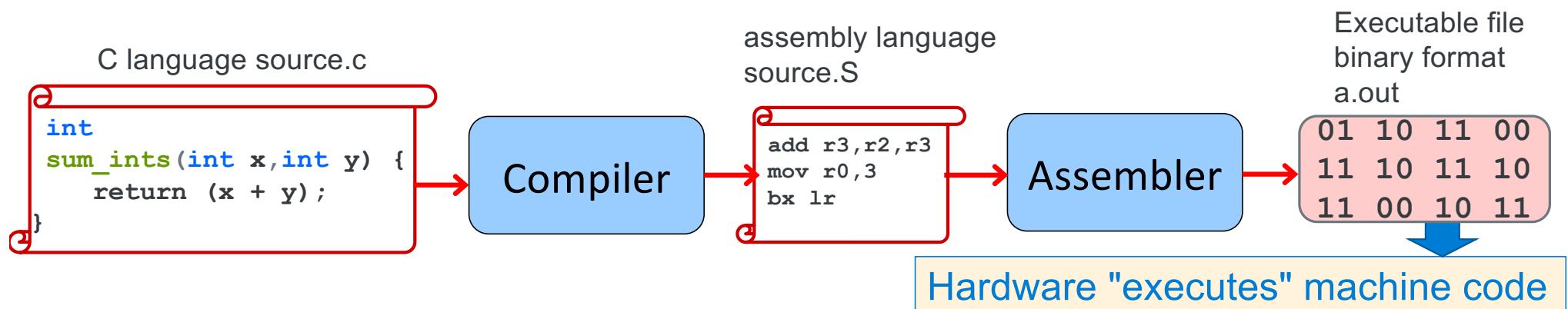
- Assembly Language Created
  - Programming Got Easier with assemblers
  - **1 assembly instruction = 1 machine hardware instruction**
  - More human-readable syntax



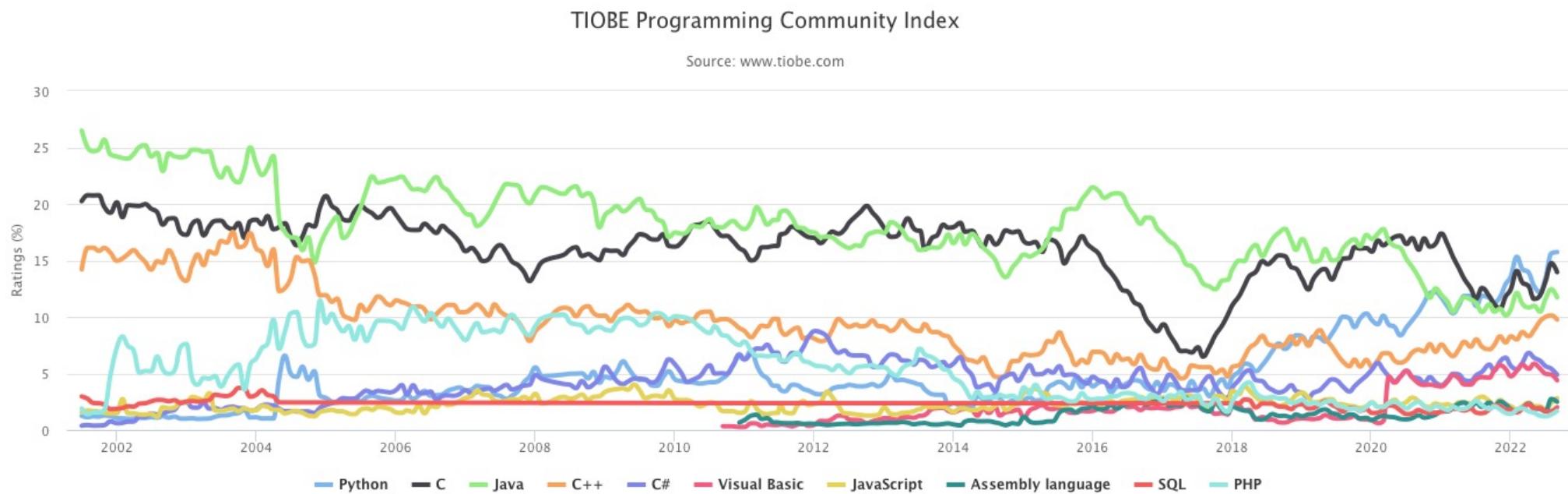
# C Programming Language – High Level Language



- C was developed by Dennis Ritchie in 1969–73 to make the Unix operating system more portable
- **Procedural language** based on the use of functions()
- C was designed to give programmers “low-level” access to memory and be easily translatable into “machine code”



# Programming Language Popularity



<https://www.tiobe.com/tiobe-index/>

# Program Performance by Language

Benchmark	Description	Input
n-body	Double precision N-body simulation	50M
fannkuch-redux	Indexed access to tiny integer sequence	12
spectral-norm	Eigenvalue using the power method	5,500
mandelbrot	Generate Mandelbrot set portable bitmap file	16,000
pidigits	Streaming arbitrary precision arithmetic	10,000
regex-redux	Match DNA 8mers and substitute magic patterns	fasta output
fasta	Generate and write random DNA sequences	25M
k-nucleotide	Hashtable update and k-nucleotide strings	fasta output
reverse-complement	Read DNA sequences, write their reverse-complement	fasta output
binary-trees	Allocate, traverse and deallocate many binary trees	21
chameneos-redux	Symmetrical thread rendezvous requests	6M
meteor-contest	Search for solutions to shape packing puzzle	2,098
thread-ring	Switch from thread to thread passing one token	50M

Performance Summary (Ratios Best to Worse)

	Energy	Time
(c) C	1.00	1.00
(c) Rust	1.03	1.04
(c) C++	1.34	1.56
(c) Ada	1.70	1.85
(v) Java	1.98	1.89
(c) Pascal	2.14	2.14
(c) Chapel	2.18	2.83
(v) Lisp	2.27	3.02
(c) Ocaml	2.40	3.09
(c) Fortran	2.52	3.14
(c) Swift	2.79	3.40
(c) Haskell	3.10	3.55
(v) C#	3.14	4.20
(c) Go	3.23	4.20
(i) Dart	3.83	6.30
(v) F#	4.13	6.52
(i) JavaScript	4.45	6.67
(v) Racket	7.91	11.27
(i) TypeScript	21.50	26.99
(i) Hack	24.02	27.64
(i) PHP	29.30	36.71
(v) Erlang	42.23	43.44
(i) Lua	45.98	46.20
(i) Jruby	46.54	59.34
(i) Ruby	69.91	65.79
(i) Python	75.88	71.90
(i) Perl	79.58	82.91

Energy Efficiency across Programming Languages, How Do Energy, Time, and Memory Relate? , Pereira, Couto, Ribeiro, Rua, Cunha, Fernandes, Saraiva, SLE 2017

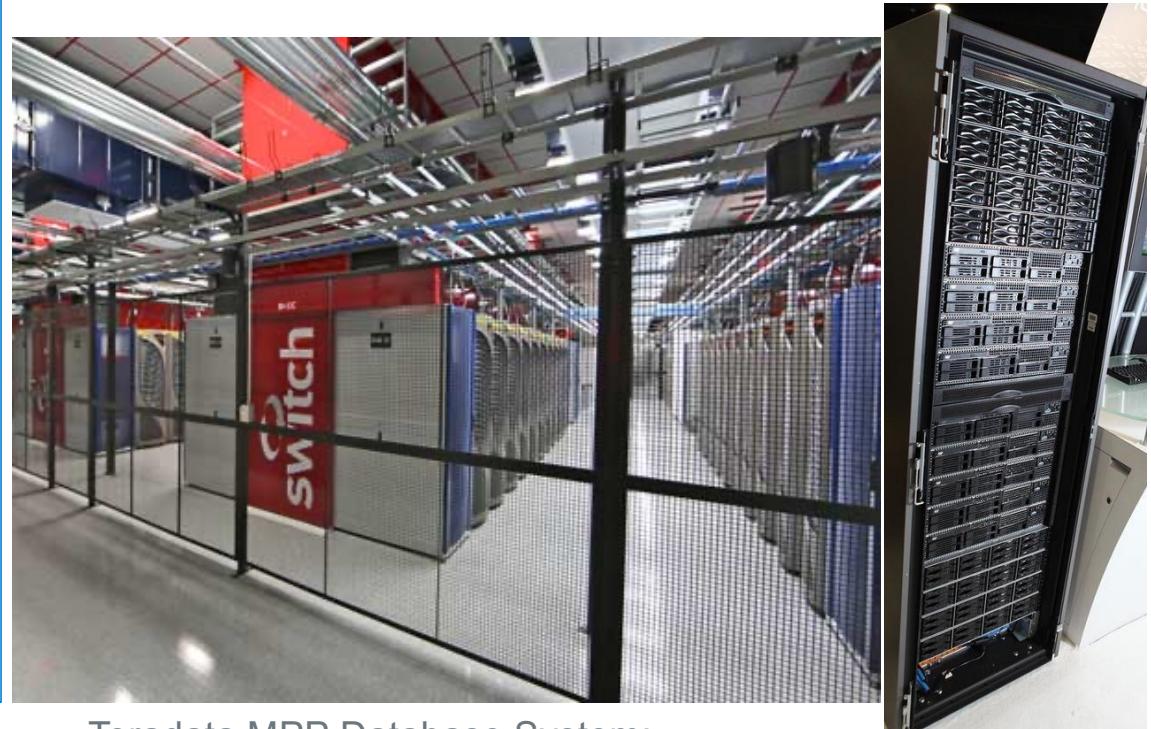
<https://greenlab.di.uminho.pt/wp-content/uploads/2017/10/sleFinal.pdf>

# Course Goals - 1

## Introduction to System Programming

### "Back-end/enterprise" server software

- Learn a new programming language:
  - C is widely used in industry
- Basic tools for large software projects
  - Development: make, gcc, gas
  - Debugging: gdb, valgrind
- Linux application programming
  - C System libraries
  - Some Linux Operating system calls
  - I/O: files, keyboard/screen, basic Inter-process communication

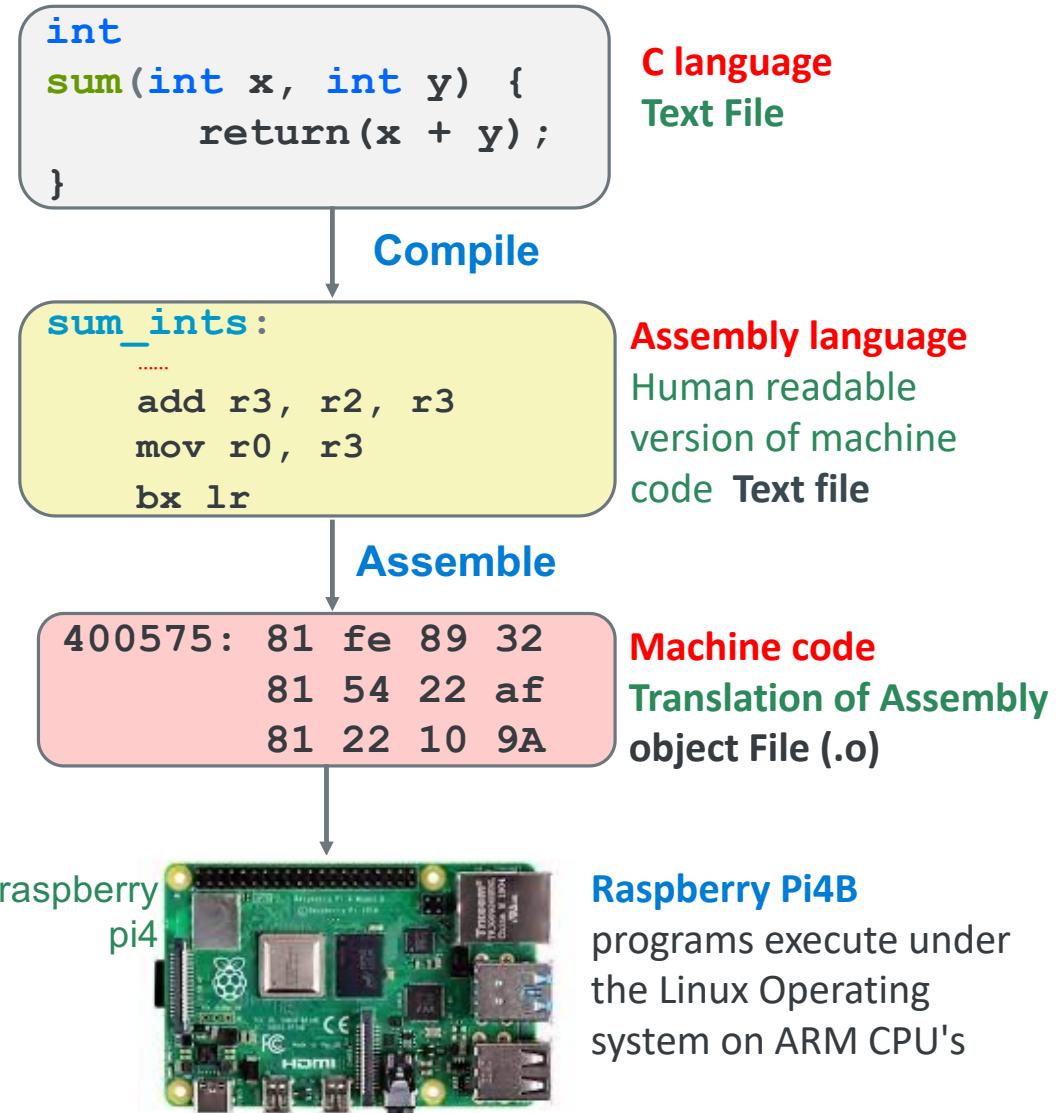


Teradata MPP Database System:  
Ebay at Switch Datacenter Las Vegas NV

## Course Goals – 2

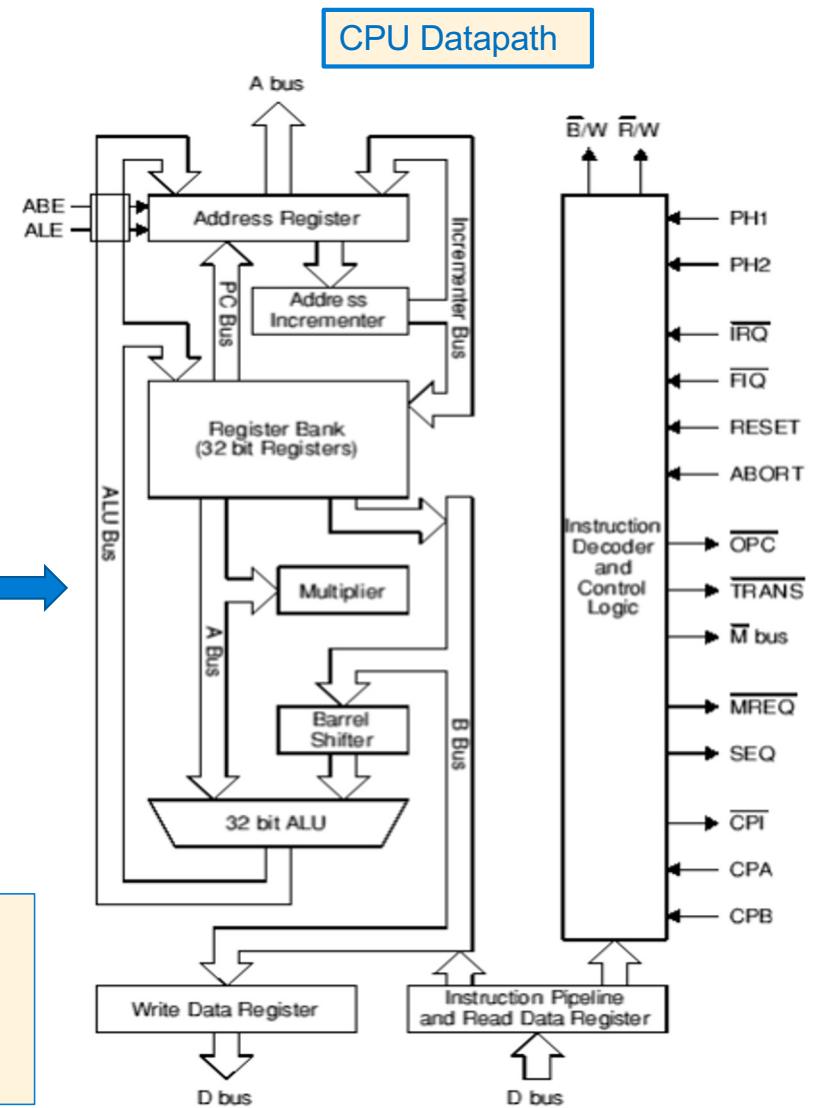
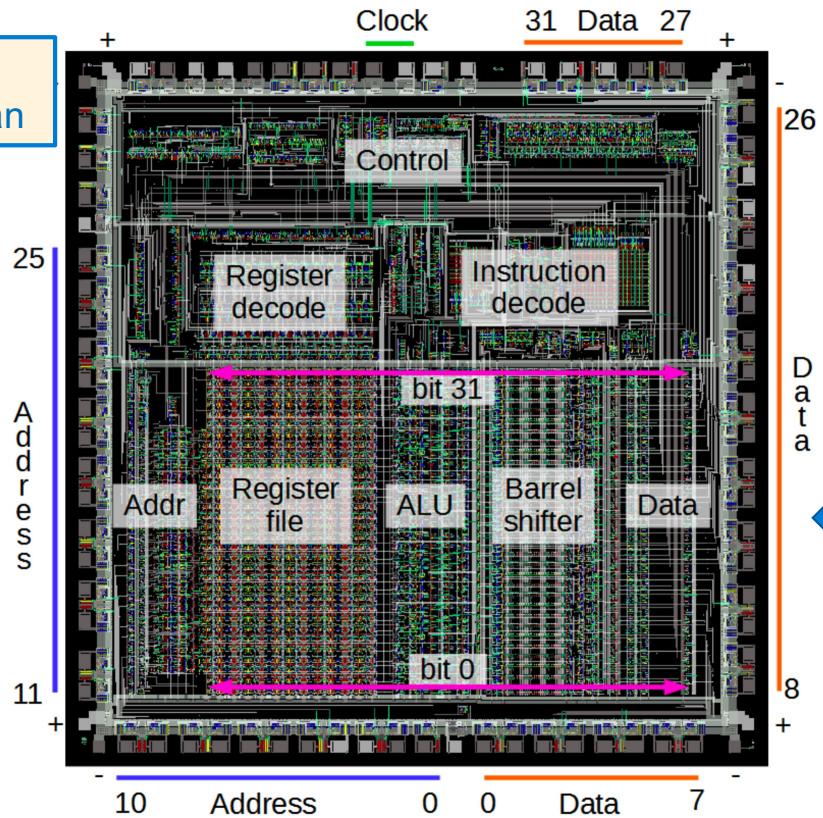
### Runtime Implementation In Linux

- Looking "*under the hood*"
- Design of a Linux executable (process runtime environment)
- **Instruction Set Architecture:** Learn a subset of the ARM 32-bit assembly language
- **Linux runtime (execution) support:** How to implement high-level language **semantics (local variables, function calls etc.)** in assembly



## Course Goals – 3 (as time permits)

Single ARM  
Core Floorplan



3. Brief Introduction to Computer Hardware and Architecture (CPU Datapath)
- gates, muxes, ALU, memory, shifters

# Introduction: C Program Structure (Single file)

```
cpp  
directives  
  
/* This is a C style comment */  
// this is a C++ style comment  
  
int main(int argc, char *argv[]) // or int main() or int main(void)  
{  
    char x = '\n';  
    printf("Hello World!%c", x);  
    printf("Hello\\  
Tritons\n");  
    return EXIT_SUCCESS; // for Linux scripts; or EXIT_FAILURE  
}  
/* define other functions */
```

first function to run

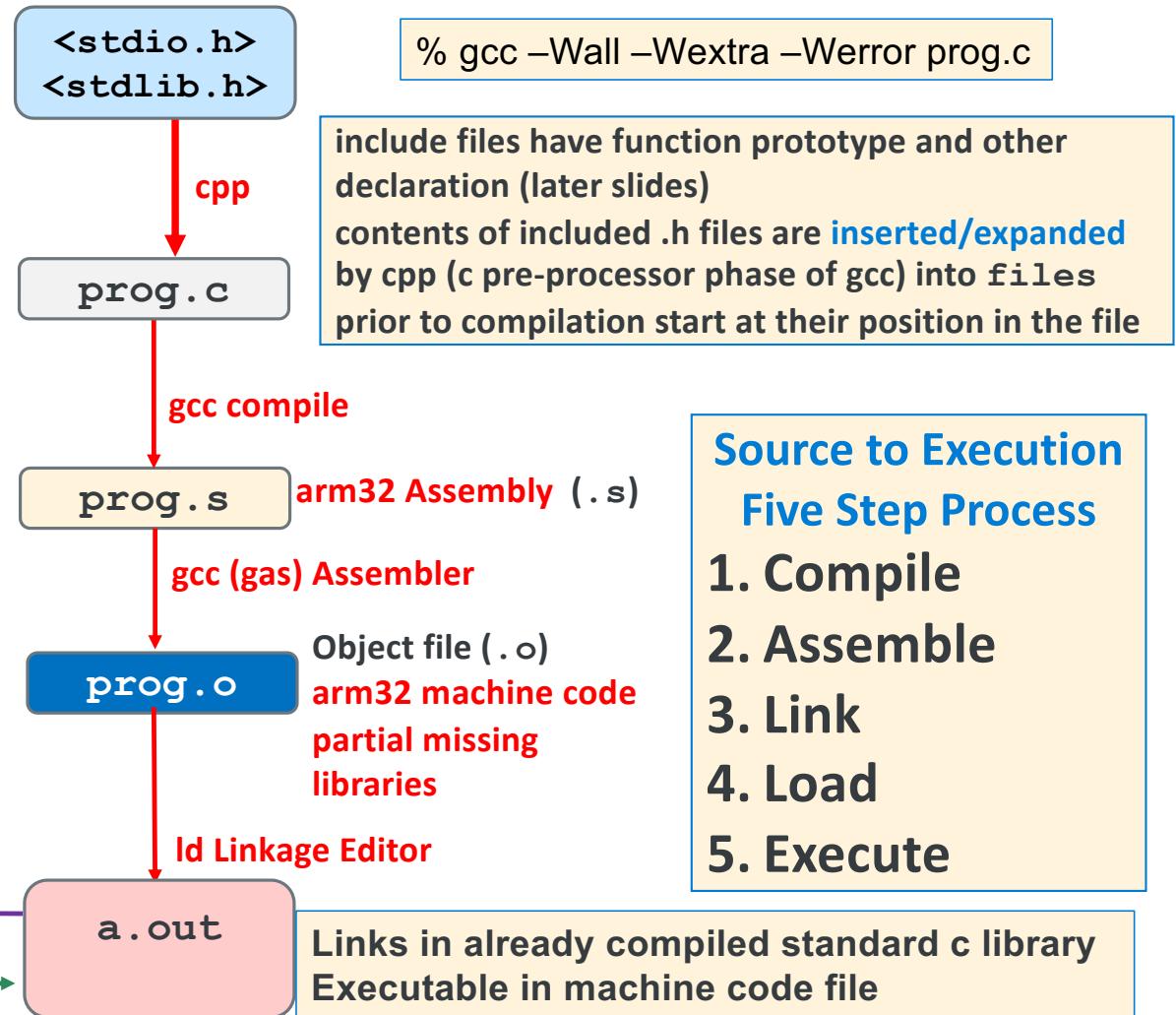
char literal '\n'

string literal "Hello World!%c"

\ immediately followed by a newline is a line continuation  
same as "Hello Tritons\n"

## Five Step Workflow in the Linux Environment, Single Source File

```
#include <stdio.h>
#include <stdlib.h>
/* A simple C Program */
int
main(void)
{
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```



1. Create a linux process
2. Load into memory & Execute a.out

## C Preprocessor (cpp): #include, #define and more

- The C preprocessor (**cpp**) *transforms* your source code before the compiler runs
  - **First step** in the C compilation process
  - **cpp is automatically invoked by gcc during compilation**
  - Input to **cpp** is a C file (text) and output from **cpp** is still a C file (text - temporary)
- A *preprocessor directive* (**#directive**) temporarily “modifies” a source file *just before* it is compiled (but it *does not save* the modifications to the source file)
- **Preprocessing:** source code is *expanded* and *preprocessed* for the compiler
  1. Include files are *inserted* into the source file (`#include <stdio.h>`)
  2. Macros are *expanded* (`#define MAX 8`) – replaces MAX with 8
  3. Comments are *removed* `/* */`, `//`
  4. Continued lines (line split) are *joined* `\`
- You are going to design #3 for PA2 and implement in a C program for PA3

# Compilation Process Actions

```
#include <stdio.h>
#include <stdlib.h>
```

cpp: inserts and processes the contents of files here  
(definitions for getchar() and putchar()):  
*/usr/include/stdio.h & /usr/include/stdlib.h*

```
/* A simple C Program */
int
main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

cpp: replaces EXIT\_SUCCESS with 0

cpp (c pre-processor): removes the Comment

compiler generates assembly code to call the library  
function printf() and pass the string "Hello World!"

compile: **gcc -Wall -Wextra -Werror prog.c -o prog**

1. cpp processes the file
2. Compiler (gcc) **compiles** main to assembly
3. Assembler (gas – called by gcc) translates the assembly to machine code
4. Linker (ld) merges the machine code for printf()  
(from a library) with your programs machine code to  
create the **executable file** prog (also machine code)

## Textbook Over-ride: Linux Return Value Convention - 1

- **main()** is the first function to start to execute and *usually* the last
- **Linux** uses a **convention** on **signaling errors** at process termination to the "shell"
  - Remember checking return values in CSE15L scripts?
  - It is the value often associated with the **return** statement from **main()**
- **In this class, always use the Linux standard return codes** as defined in **<stdlib.h>** when returning from **main()** or exiting your program

```
EXIT_SUCCESS      // program completed ok; usually 0
EXIT_FAILURE      // program completed with error; non-zero value
return EXIT_SUCCESS;
```

- To force a clean program termination *from any function* in your program, you can call **exit()**  
`exit(EXIT_FAILURE); // or exit(EXIT_SUCCESS);`

## Main() Return Value Under Linux - 2

Example 1

```
#include <stdio.h>
#include <stdlib.h>
int
main(void) {
    /* Your code here */
    /* code was successful */
    return EXIT_SUCCESS;
}
```

Example 2

```
#include <stdio.h>
#include <stdlib.h>
int
main(void) {
    /* Your code here */
    /* a failure occurred */
    return EXIT_FAILURE;
}
```

Example 3

```
void funcB(void)
{
    /* lots of code */
    .... exit(EXIT_FAILURE);
    /* lots of code */
}

int funcA(void)
{
    /* lots of code */
    funcB();
    return z;
}

int main(void) {
    /* Your code here */
    x = funcA();
    /* Lots of code */
    return EXIT_SUCCESS;
}
```

## Data types: C Versus Java

Data Types	Java	C
Character	char // 16-bit unicode	char // 8 bits (signed or unsigned)
integers	byte // 8 bits short // 16 bits int // 32 bits long // 64 bits	(unsigned, signed) char // 8-bits (unsigned, signed) short // unspecified (unsigned, signed) int // unspecified (unsigned, signed) long // unspecified
Floating Point	float // 32 bits double // 64 bits	float // unspecified double // unspecified
Logical type	boolean	#include <stdbool.h> bool conditional tests that evaluate to 0 are false, true for all other values
Constants	final int MAX = 1000;	// two alternatives to do this #define MAX 1000 // C preprocessor const int MAX = 1000;

## C vs Java: Expression Type Promotions, Demotions, Casts

- Java: demotions are not automatic  
C: demotions are automatic
- Cast: a unary operator (`variable_type`) explicitly converts the type the value of an expression to `variable_type`
- To explicitly get the floating-point equivalent of the *integer variable a* you would use a cast and write `(float)a`

```
int i;
char c;
i = c;          /* Implicit promotion */
                 /* OK in Java and C */
c = i;          /* Implicit demotion */
                 /* Java: Compile time error */
                 /* C: OK; truncation */
c = (char)i;   /* Explicit demotion using a cast */
                 /* Java: OK; truncation */
                 /* C: OK; truncation */
```

## Java versus C: Mostly Similar Syntax

```
int x = 42 + (7 * -5);          // variables, types
double pi = 3.14159;            // C++ style comment
char c = 'Q';                  /* C style comment */
```

```
for (int i = 0; i < end; i++) {      // for Loops
    if (i % 2 == 0) {                // if statements
        x += i;
    }
}
```

```
while (((x > 0) && (c == 'Q')) || (b == 3)) { // while Loops, Logic
    x = x / 2;
    if (x == 0) {
        return 0;
    }
}
```

## Conditional Statements (if, while, do...while, for)

- C conditional test expressions: **0 (NULL) is FALSE, any non-0 value is TRUE**
- C comparison operators ( ==, !=, >, etc.) evaluate to either 0 (false) or 1 (true)
- Legal in Java and in C:

```
i = 0;  
if (i == 5)  
    statement1;  
else  
    statement2;
```

Which statement is executed after the if statement test?

```
i = 0;  
if (i)  
    statement1;  
else  
    statement2;
```

- Illegal in Java, but legal in C (often a typo!):

```
i = 0;  
if (i = 5)  
    statement1;  
else  
    statement2;
```

Which statement is executed after the if statement test?

# Logical Data Shortcuts

```
...
int i;
...
if (i) /* is the same as (i != 0) */
    statement1;
else
    statement2;
...
```

- Assignment inside conditional test – often includes a function call (**this is very common!**)

```
if ((i = SomeFunction()) != 0)
    statement1;
else
    statement2;
```

assignment returns the value that is placed into the variable to the left of the **= sign**, then the test is made

## Be Careful with the comma , sequence operator

- Sequence Operator ,  
 $expr1,expr2$
- Evaluates  $expr1$  first and then  $expr2$  evaluates to or returns  $expr2$

```
for (i = 0, j = 0; i < 10; i++, j++)  
...
```

- Unexpected results with , operator (some compilers will warn)

```
i = 64,323;           // i = 64  
i = (64, 323);       // i = 323
```

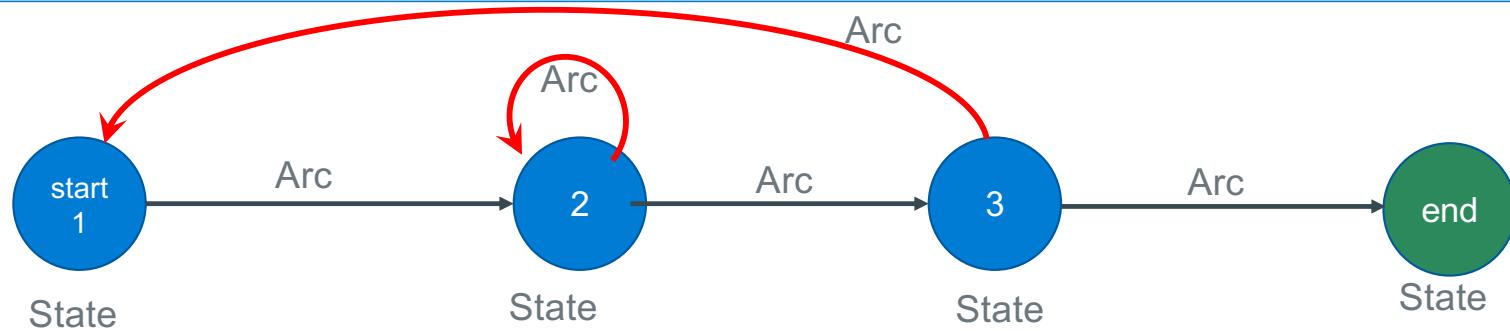
## PA # 2: Designing a Deterministic Finite State Automaton

- Deterministic Finite state machine (or a DFA) is a way of representing (or *detecting*) a *language*
  - Example: set of string patterns (e.g., **HA**) *accepted* or *rejected* based on an **input sequence**
- Use Examples
  - Text Editor: Finding a pattern of text in a file
  - Hardware: Traffic Light sequencing

Also: CSE 105, CSE 140, CSE 131, CSE 123

### Circle (States) and Arc representation

- A **circle (state)** represents **memory of what has already been seen before** based on the input
- An **arc** represents a **transition** from one state to the next state for a specified input
  - The next state can be the same state or a different state
- **For all possible inputs, there must be just one next state** (makes it **deterministic**)

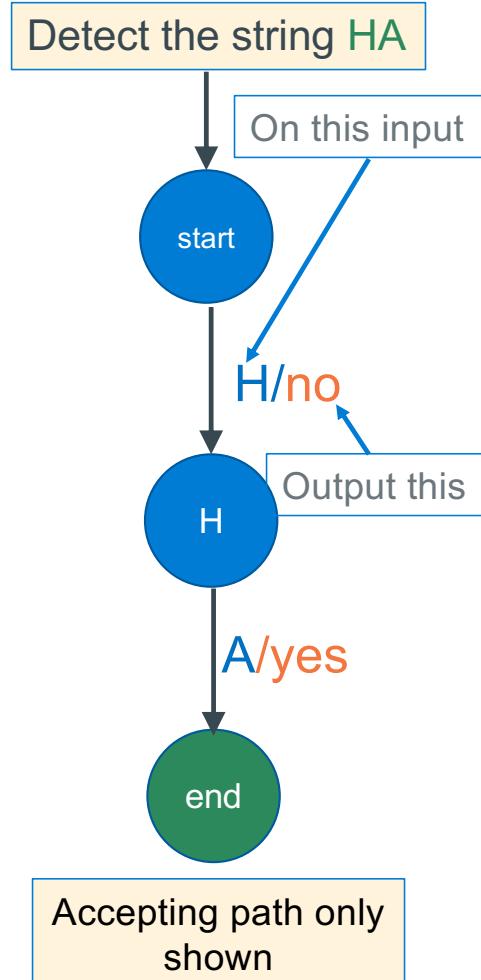


# States and Transitions

- **States:** Two Special states

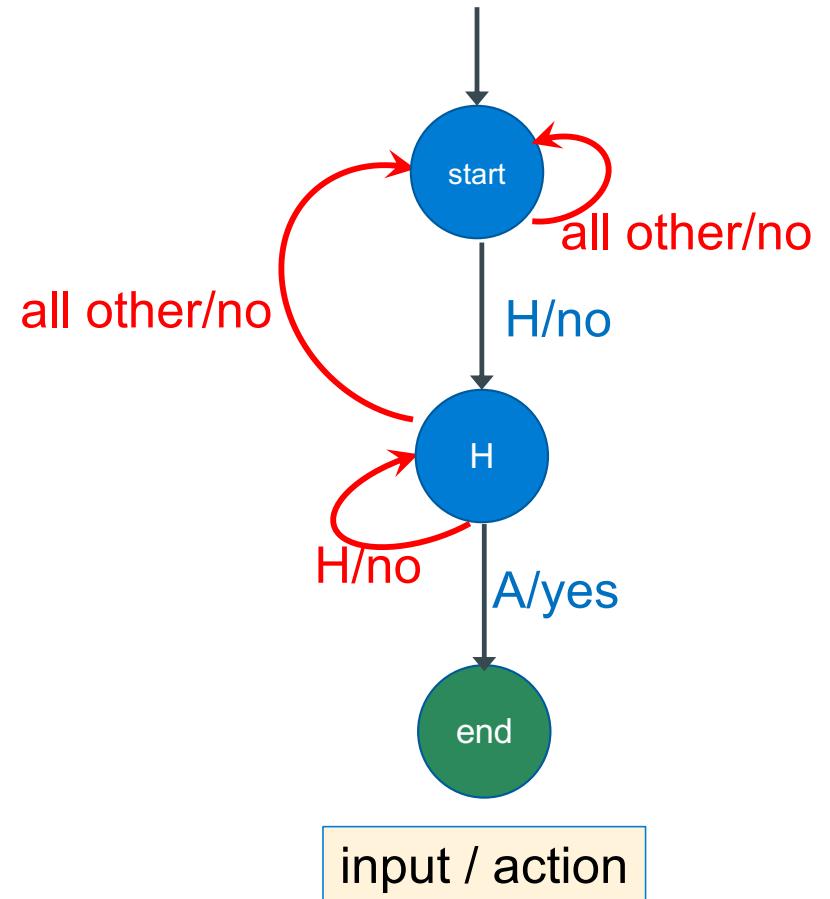
-  **start** state (initial or first)
-  **end** state (we are done or final)

- Each arc is labeled with the next input that causes it to be taken
- Each arc has a **label** with the **notation** (**next input / next output**):
  1. When an **input matches next input** that **arc is taken**
  2. **output or action** (if any) associated with that input occurs when making that transition (it is ok to have no output specified)



## Designing a Deterministic Finite State Automaton -2

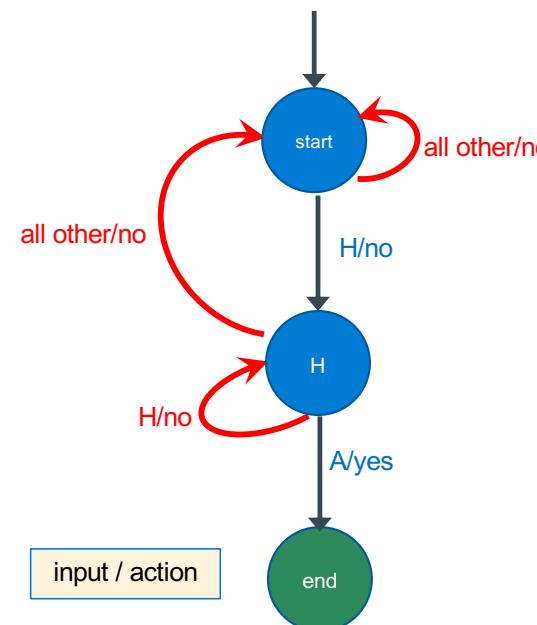
- Problem: detect the string *HA*
- Refine the DFA from the previous slide to specify responses on all possible inputs at each state
- Make sure the next state transition is (arc) is *unambiguous* (unique – each input selects only one arc)



## Designing a Deterministic Finite State Automaton - 3

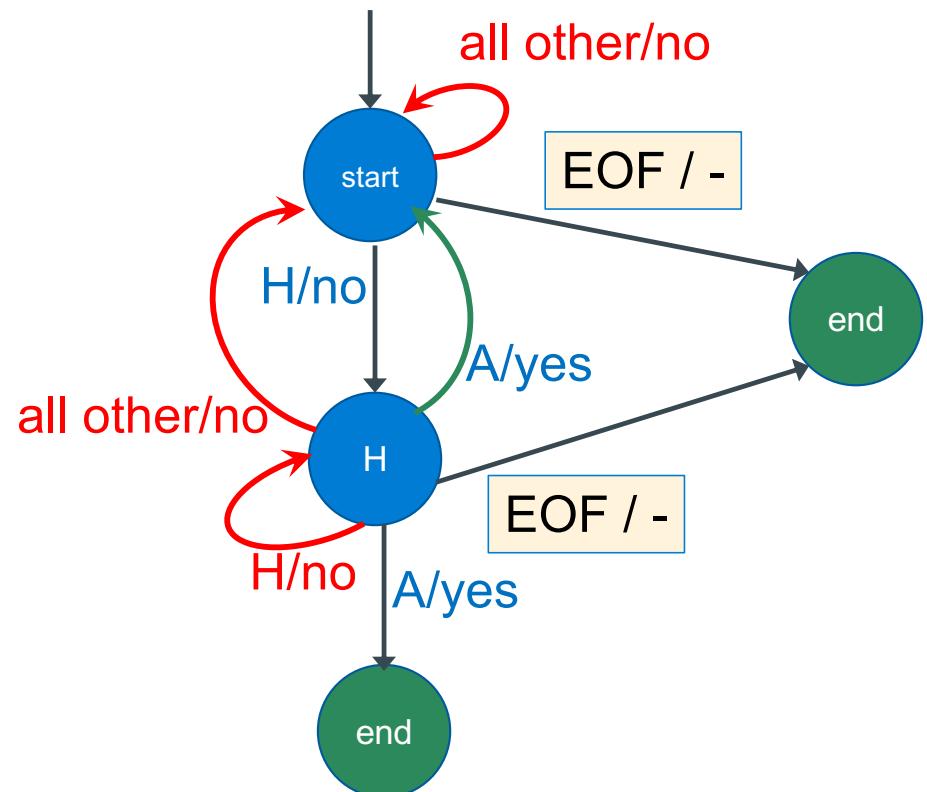
- Say you want to process the entire contents of a text file
  - from the top (top of file)
  - to the bottom (end of file)
- The state machine would stop after seeing the first HA, missing the other occurrences of HA
- **Action:** Alter the machine to process input from a text file until the end of the file (EOF) is reached (you processed the whole file)

This is a text file with a lot of HA in it.  
There is a HA here and a HA there and a HA everywhere.  
There is also HA HA HA.

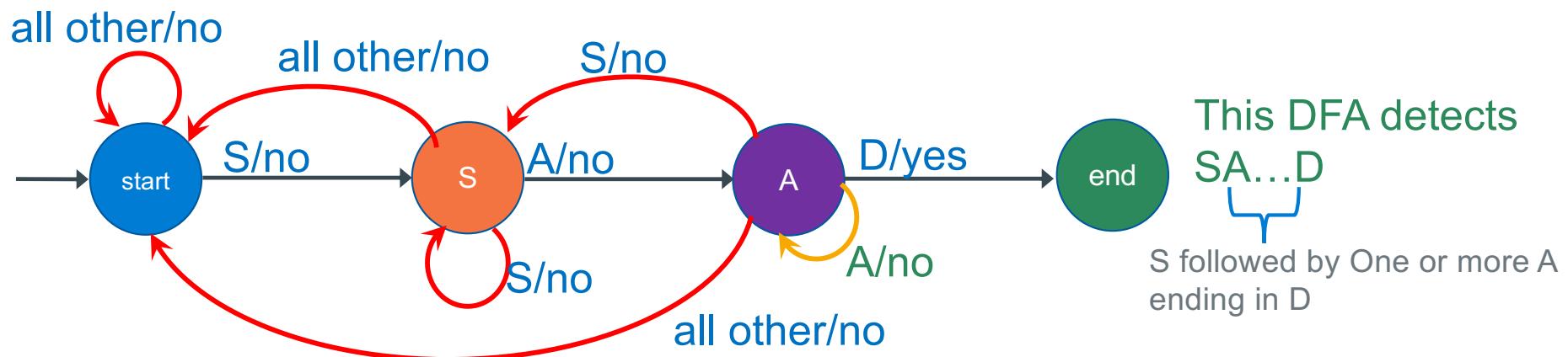
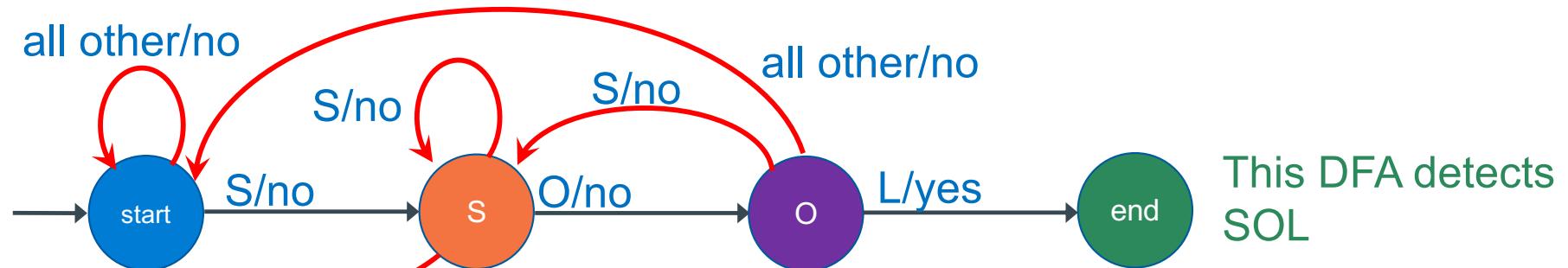


## Designing a Deterministic Finite State Automaton - 4

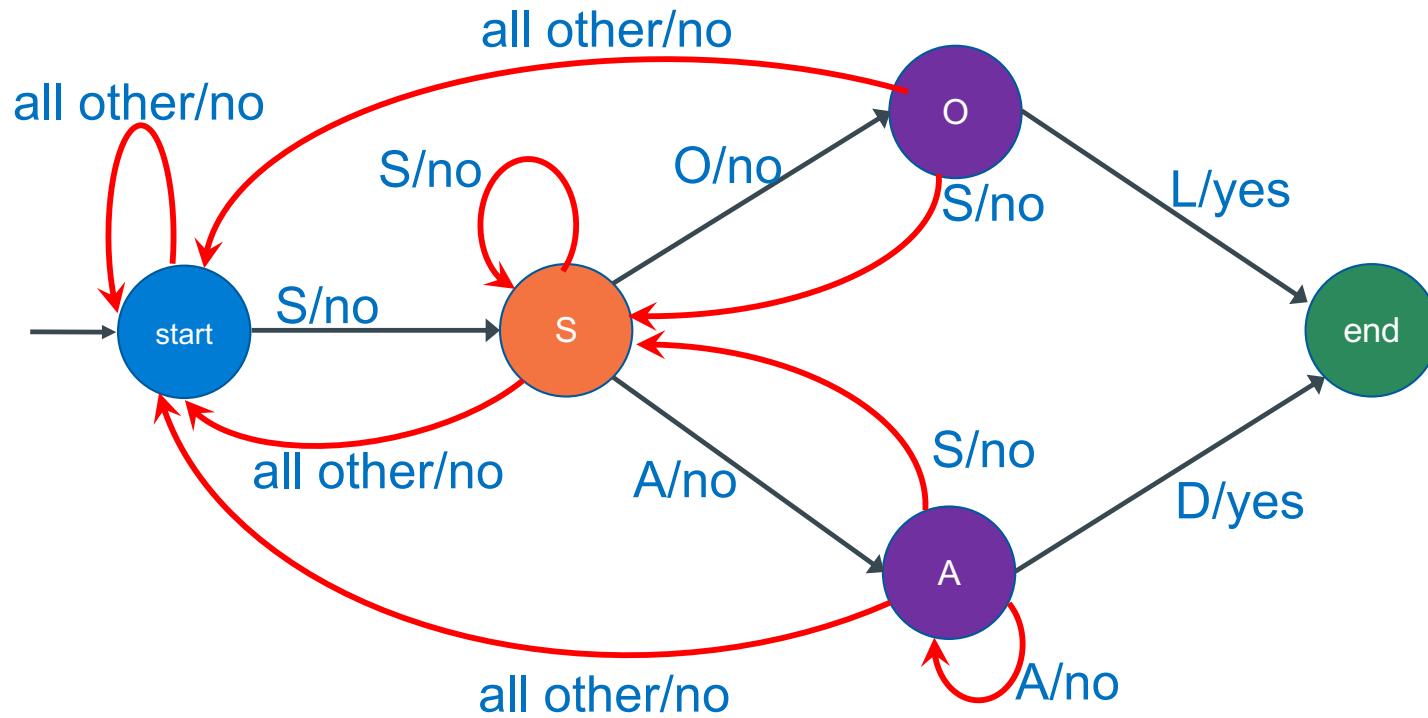
- To adjust the DFA to act on continuous input (multiple instances of the pattern), "redirect" the arc(s) that point to the end state to point back at the start state
- Add arcs from each state when EOF on input is detected to the end state
- Remember: Transition actions can be anything you need to do
  - Example: output a string, perform a computation or do noting, etc.
- Let's look at an example



## Merging DFA's - 1



## Merging DFA's – 2 (Finished)



# Background: What is a Definition in C?

- **Definition:** creates an instance of a *thing*
  - There **must be exactly one** definition of each *function or variable* (no duplicates)
  - In C you must define a **variable or a function before first use** in your code
- 
- **Function definitions**
    1. **creates code** as specified by the body
    2. **allocates** memory for the code
    3. **binds** the function name to the allocated memory
  - **Variable definitions**
    1. **allocates memory:** at compile time (global vars) or at runtime (local vars)
    2. **initialize memory:** (global vars) or **create code** to initialize the memory (local vars)
    3. **binds (or associates)** the variable name to the allocated memory

# C Function Definitions - 1

- C Functions are not methods
  - no classes, no objects
- C function definition
  - returns a value of returnType
  - zero or more *typed* parameters
- Every program must have a main() function
- main() is the first function in your code to run/execute
  - main() is not the first code to run in a Linux process, it is called by the C runtime startup code
    - later in course

```
returnType fname(type param1, ..., type paramN)
{
    // statements
    return value;
}
```

function definition

```
// returns: sum of integers from 1 to max
int
sum(int max)          // function definition
{
    int i, sum = 0;   // variable def

    for (i = 1; i <= max; i++) {
        sum += i;
    }

    return sum;
}
```

## C Function Definitions - 2

this a C pre-processor (cpp) macro  
it is not a variable it is a "substitution"

- Function of type **void** does not return a value
- **A void parameter or an empty parameter list** specifies this is a function with no parameters
  - A common practice is to use the keyword **void** to specify an empty or ignored parameter list)
- Function arguments are evaluated, then the resulting value is COPIED to a memory location allocated for the argument (like a local variable)
  - So, functions are free to change parameter values without side effect
  - This method of parameter passing is called **call by value**

```
// prints sum of integers 1 to MAX
#define MAX 8
```

```
void
sum(void)      // or sum()
{
    int i, sum = 0;
    for (i = 1; i <= MAX; i++) {
        sum += i;
    }
}
```

the %d specifies the next arg to printf is an int

```
printf("%d\n", sum);
return; // optional
}
```

## C Function Definitions - 3

- In standard C, functions **cannot be nested** (defined) inside of another function (called *local functions*)

```
int outer(int i)
{
    int inner(int j) // not in standard c
    {
    }
}
```

# What is a Declaration in C?

- **Declaration:** describes a *thing* – specifies types, does not create an instance
- **Function prototype** describes (more in a few slides ...)
  - The type of the function return value
  - The types of each of the parameters
- **Variable declaration** describes
  - The type of a variable that is defined elsewhere
- **Derived and defined type description**
  - *Later slides:*(enums, struct, arrays, unions)
- In C, you must **declare a function or variable before you use it**
  - Use before declaration will implicitly default to int (and a compiler warning/error – not good)
- An **identifier** can be **declared multiple times**, but **only defined once**
- **A definition is also a declaration in C**

## Translation units (single file): Control & Limit definition and declaration validity range

sum() is defined and declared here

Definitions and declarations validity:

1. Only in the file (also called a translation unit) where they are located and
2. From the point of definition or declaration to the end of the file (translation unit)

sum() is used here

```
// sum of integers from 1 to max
#include <stdio.h>
#include <stdlib.h>
#define MAX 8

int sum(int max)
{
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}

int main(void)
{
    printf("sum(8): %d\n", sum(MAX));
    return EXIT_SUCCESS;
}
```

## Real C programs are distributed across multiple files

- Large programs in one source file are very difficult to manage with development teams
  - Consider a program with 10 million lines of code
  - Many developers working on it
- C supports programs divided across multiple files (what are called *independent translation units*)
- C when a **function call** or a **variable reference** is used in a statement the, compiler must know:
  - variable
    - type
  - functions
    - return value type
    - Number of arguments to a function
    - type of each argument

in this example funcA() must appear before it is used by main()  
– this is ordering is a pain...

```
int funcA(int z)
{
    int x;
    /* lots of code */
    return z + x;
}
int main(void) {
    int y;
    /* Lots of code */
    x = funcA(y);
    /* Lots of code */
    return EXIT_SUCCESS;
}
```

# Breaking a Program into Multiple files: Header Files and Source files

- **Desired Effect:** Say we want to distribute the function in a program across several source files yet allow the functions to call each other
- **Approach:** For each source file, `file.c` create a header file named `file.h` that contains just the declarations of the functions (and other things, later...) you want to be made visible (exported) to functions in other files

declarations

function prototypes  
A(), B(), C()

file1.h

function prototypes  
D, E

file2.h

definitions

#include "file1.h"  
function A, B, C

file1.c

#include "file2.h"  
function D, E

file2.c

no main.h needed **in this example** as  
calling main is likely to create an infinite  
loop (and not what you wanted)

#include "file1.h"  
#include "file2.h" main.c  
function main

- **How to implement:** To call a function defined in another file compiler must know the types and parameter to generate code
  - **#include the declaration file** in your source file at the top of the file; this imports the declaration

# Header Files (often called .h files)

- Header file: a file whose only purpose is to be **#include**'d
  - Declares interfaces (function prototypes, user defined types, global variable, cpp macros, etc.)
  - Generally, uses the filename .h extension
  - There are <system-defined> header files (usually located in /usr/include/...)  
`#include <stdio.h>`
  - "programmer-defined" header files (usually in a relative Linux path - see -I flag to gcc)  
`#include "else.h"`
  - You use/import the interfaces to a C source file by **#include**-ing the header (interface) file
- What goes where....
  - .h files **only contain declarations**, **never even think of including a definition statement**
  - .c files **never contain prototype declarations** for functions that are intended to be exported through **the module interface**
- NEVER EVER **#include** a .c file

## Function Prototypes – Separate Definition and Declaration

returnType fname(type\_1, ..., type\_n); function prototype

Function prototype (a function declaration)

- function header , followed by a semicolon (;) instead of **a code block**
- It **specifies** the **function type** (the **return type**) and **parameter types**

int sum(int); *// function declaration*

int sum(int max) *// function definition*  
{  
 int i, sum = 0;  
 for (i = 1; i <= max; i++) {  
 sum += i;  
 }  
 return sum;  
}

this is the  
code block

## Function Prototype Declarations – Multiple files

```
// sum of integers from 1 to MAX    main.c
#include <stdio.h>
#include <stdlib.h>
#include "sum.h"
#define MAX 8

int main(void)
{
    printf("sum(%d): %d\n", MAX, sum(MAX));
    return EXIT_SUCCESS;
}
```

```
#include "sum.h"                                sum.c
int sum(int max) // function definition
{
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

- Always #include your own declaration files
  - compiler will then check that the **definition** and **declarations are consistent**

header guards  
typical: header name with an \_

Why needed:  
protects the include file from  
being included multiple times  
(we will discuss later....)

header guards

header guards

```
#ifndef sum_h
```

```
#define sum_h
```

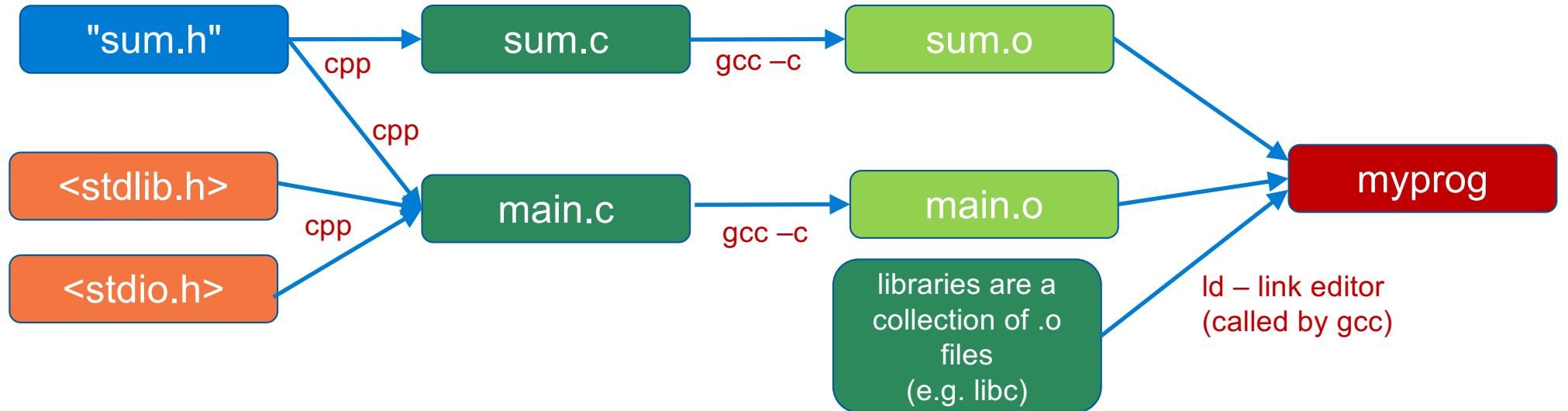
```
int sum(int); // function declaration
```

```
#endif
```

## Breaking a Program into Multiple files: Separate Compilation

- **Desired Effect:** When we modify a single file in a multi-source file program, we want to only recompile the file that changed and combine it with an already compiled version of the other files (think of the time to compile a program with 10 million lines of source code)
- **Approach:** Compilers supports partial compilation
  1. Compile option to just create a machine code version for each file by itself
    - **file is not complete** (it is missing parts, like the machine code for functions in other files)
  2. Combine the machine code versions together in a separate process (**linking**) to create an executable machine code file that has machine code for all functions and variable

## Compiling Multi-File Programs (assembly steps not shown)



1. compile each .c file independently to a .o object file (incomplete machine code)

```
gcc -Wall -Wextra -Werror -c sum.c # creates sum.o
```

```
gcc -Wall -Wextra -Werror -c main.c # creates main.o
```

2. link all the .o objects files and library's (aggregation of multiple .o files) to produce an executable file (complete machine code) (gcc calls ld, the linker)

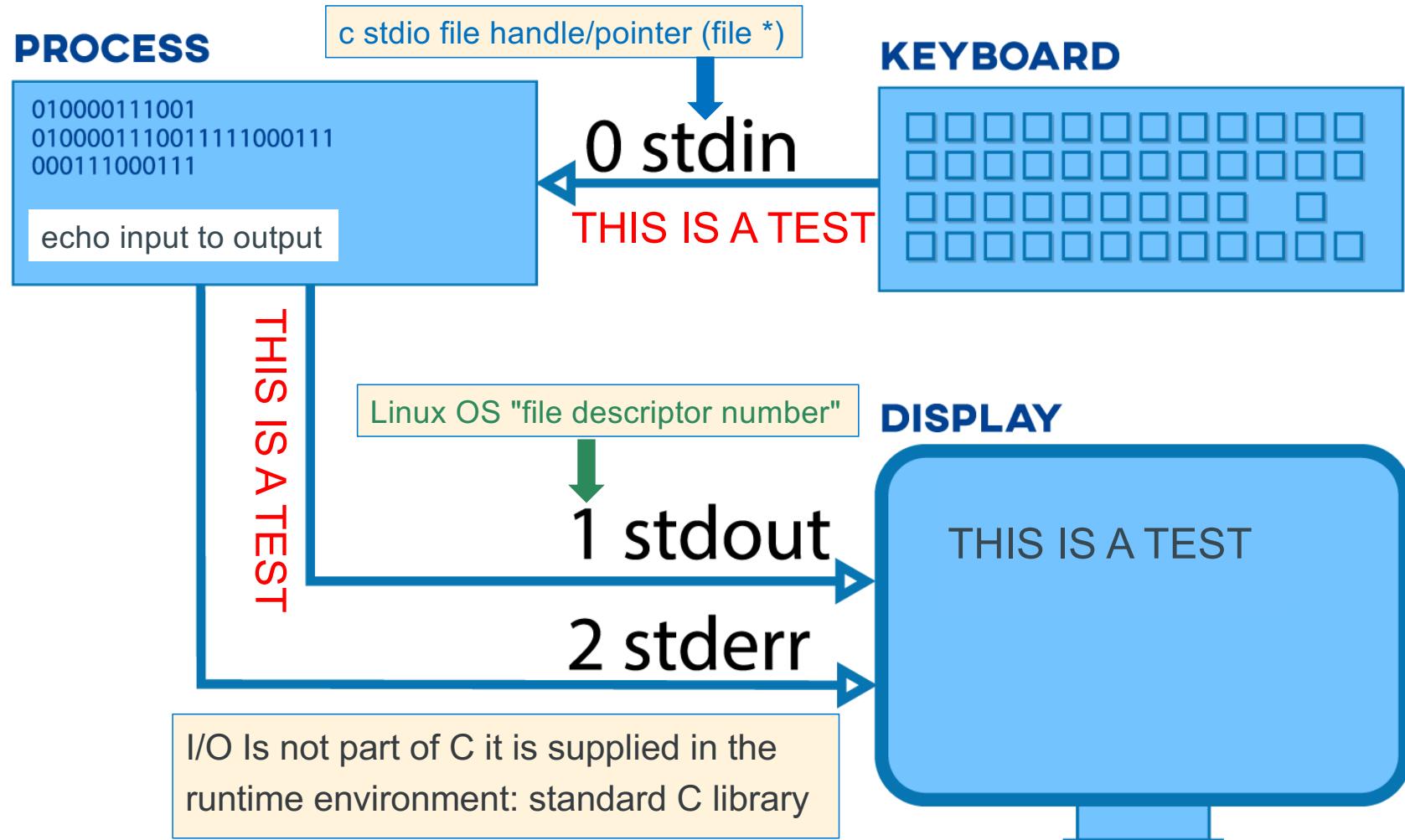
```
gcc -Wall -Wextra -Werror main.o sum.o -o myprog
```

## char literals In C – Similar to Java

- Usually used to store characters – thus things like file names
- Use single quotes 'a' for chars
- Double quotes "string" are for string literals (later)

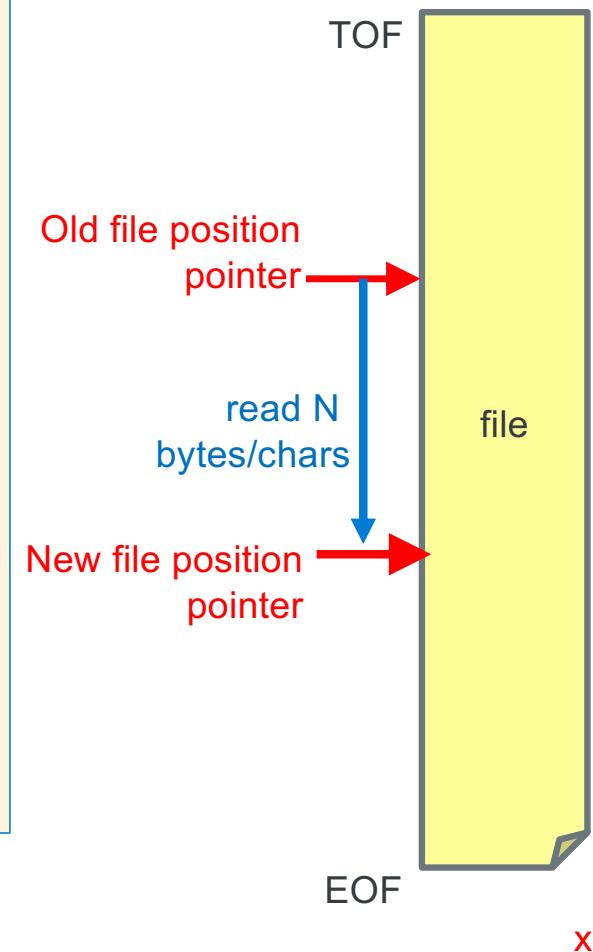
escape sequence	Notes
'a'	letter char
'0'	digit char
'\n'	newline char
'\r'	carriage return
'\t'	tab char
'\b'	back space
'\0'	null char
'\\'	\ char
'\''	single quote
'\"'	double quote

# Linux/Unix Process and Standard I/O (CSE 15L) - Defaults



## Under the Hood: `stdio` File I/O – File Position Pointer On Files

- Read/write functions *advance the file position pointer* from TOF towards EOF on each I/O
  - Moves towards EOF by number of bytes read/written
- Called **sequential I/O** (sequential read & sequential write)
  - READ: After the last byte is read in a file, additional reads results in a function return value of EOF
    - EOF is **NOT a character in the file**, but a condition on the stream
    - EOF **signals** no more data is available to be read
    - EOF is usually a `#define EOF -1` macro located in the file `stdio.h`
  - WRITE: write positioned *after the last byte extends the length* of the file (increases file length)

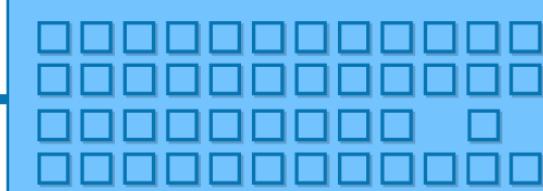


# Under the Hood: stdio File I/O – Working with a Keyboard

## PROCESS

```
010000111001  
0100001110011111000111  
000111000111
```

## KEYBOARD



How do I  
signal EOF?

- How can you have a **TOF** and **EOF** with a keyboard?
- Streams were **designed** to work primarily on **files**
  - With **keyboard devices** the **semantics** of *file operations* are “*simulated*”
- Example: when a program (or a shell) is **reading** the keyboard and is blocked waiting for input (waiting for you to type a line),
- To signal an **EOF condition from the keyboard**, type on an input line all by itself: **two key combination (at same time), followed by a return/enter:**  
**ctrl-d**      (*often shown in slides etc. as ^d*)

## C Library Function API : Simple Character I/O

Operation	Usage Examples
Write a char	<pre>int status; int c; status = putchar(c);           /* Writes to screen stdout */</pre>
Read a char	<pre>int c; c = getchar();                /* Reads from keyboard stdin */</pre>

```
#include <stdio.h> // import the API declarations
```

```
int putchar(int c);
```

- writes c (converted to a char) to **stdout**
- returns **c** on success **OR** EOF (a macro often defined as -1) on failure
- see man 3 putchar

```
int getchar(void);
```

- returns the next input character (if present) **converted to an int** read **from stdin**
- see man 3 getchar
- Both functions return an int because they must be able to return both valid chars and indicate the **EOF condition is really a -1**, not a valid char

Why is  
character I/O  
using an int?

Answer: Needs  
to indicate an  
EOF (-1)  
condition that is  
not a valid char

# Simple Character I/O – Using STDO

```
// echo stdin to stdout
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int c; // Why is c an int?
            // Returns char in an
            // int or EOF (-1)

    while ((c = getchar()) != EOF) {
        (void)putchar(c); // ignore return value
    }
    return EXIT_SUCCESS;
}
```

Always check return code to handle EOF  
EOF is a macro integer in stdio.h

Always check return codes unless you do not need it  
Sometimes you may see a (void) cast which indicates ***ignoring the return value is deliberate*** this is often required by many coding standards

% ./a.out

thIS is a TeSt

thIS is a TeSt

^d %

Typed on keyboard

Printed by program

Typed on keyboard

## Watch out for Hardware differences: Example: Is a Char signed or unsigned?

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    char c = 255;

    printf("%d\n", (int)c);

    return EXIT_SUCCESS;
}
```

- variable c is being cast promoted to an int
- So, what is printed?
- Depends on the hardware
- On arm (pi-cluster) it is unsigned  
255
- On Intel 64-bit (ieng6) it is signed  
-1

# C Library Function: Simple Formatted Printing

Task	Example Function Calls
Write formatted data	<pre>int status; status = fprintf(stderr, "%d\n", i); status = printf("%d\n", i);      /* Writes to stdout */</pre>

```
int fprintf(FILE *file, const char *format, ...);
```

- Write chars to the file identified by **file** (stdin, stdout, stderr are already open)
- Convert values to chars, as directed by **format**
- Return count of chars successfully written
- **Format** is the output specifications enclosed in a “string”
- Returns a negative value if an error occurs

```
int printf(const char *format, ...); // *format - Later in course
```

- Equivalent to **fprintf(stdout, format, ...);**
- See **man 3 printf** for more information on the **format**

# Formatted Output Conversion Examples

- Conversion specifications example
  - Begin with the `%` character
  - Describe a value to be filled in (from a variable or expression argument) during printing
  - `%f` conversion specifier for **float** variables; the default precision is 6
  - `%d` conversion specifier for **int** variables
  - `%c` conversion specifier for **char** variables
  - many more conversion specifiers (online manual: `% man printf` and the textbooks)

```
int i = 10;
float x = 43.2892;
char z = 'i';

printf("%c = %d, x = %f\n", z, i, x);
```

- Output from the `printf()` would look like

```
i = 10, x = 43.289200
```

## C Enumerated Data Type

```
enum tag {enum_0, enum_1, ..., enum_n};      // defines just the type  
enum tag var;                                // defines a variable instance
```

- **Enumerated type:** is a *user defined type* whose **all possible values** (typically a small list) are listed by the programmer in the enumeration list
- **Enumeration tag:** identifies the specific enumeration type
  - The new type is **enum tag**
- **Enumeration list:** lists one or more **enumeration identifier names** for each of the possible values

```
enum suit {CLUBS, DIAMOND, HEARTS, SPADES}; // user defined type
```

Put this in a .h file

```
enum suit card = HEARTS; // card defined & initialized to HEARTS
```

Put this in a .c file

## C Enumerated Data Type

```
enum tag {enum_0, enum_1, ..., enum_n};      // defines just the type  
enum tag var;                                // defines a variable instance
```

```
enum suit {CLUBS, DIAMOND, HEARTS, SPADES}; // CLUBS=0, DIAMOND=1, etc .h file
```

```
enum suit card = HEARTS; // card defined & initialized to HEARTS .c file
```

- **Enumeration identifiers** are really **integer constants** and can be used as such
- **Default:** compiler **assigns a sequential integer value** to each name, starting with 0 and incrementing from there; so, the state values are 0, 1, 2, 3, ...
- **Over-ride the default value** by specifying a **unique integer value** for each enumeration name

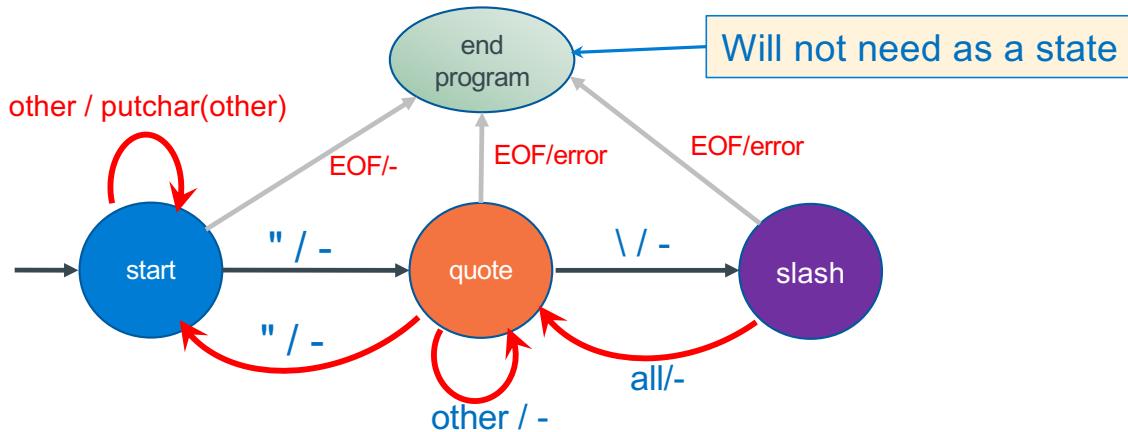
```
enum compass {NORTH=0, EAST=90, SOUTH=180, WEST=270}; .h file
```

```
enum compass direction = WEST; // direction defined & initialized to WEST .c file
```

# Programming a Deterministic Finite Automaton -1

DFA program operation

Copy input to output  
while removing  
"strings" from output



- Use enum to define the states

```
enum typestate {START, QUOTE, SLASH};
```

- Declare a set of functions for each state

```
enum typestate startSTATE(int);  
enum typestate quoteSTATE(int);  
enum typestate slashSTATE(int);
```

- Each function implements the outbound arcs

1. returns the next state based on the next input
2. performs actions associated with arc taken

// process input until EOF

```
while ((c = getchar()) != EOF) {  
    // (1) call the current state  
    //      and pass it the input  
    // (2) it will determine the arc  
    // (3) it performs any output  
    // (4) return the next state  
}
```

**To be continued....**

## Extra Slides Section

- Slides in this section may be used in class but contain material that you will find useful

# Programming Language Concepts: Scope

- **Scope:** Range (or the extent) of instructions over which a name/identifier can be referenced with C instructions
  - 1. File Scope:** Within a single source file (also called a translation unit)
  - 2. Block Scope:** Within an enclosing block (variables only)
  - 3. Function Scope:** Within the enclosing function (goto **label**)

```
int global0;          /* global variable file scope */  
void                /* function with file scope */  
foo(int parm)        /* parameter parm block scope to function foo */  
{  
    int i, j = 5;      /* block scope - entire function */  
    for (int i = 0; i < 10; i++) { // I has block scope for; hides outer i  
        label:          /* function scope not block scope */  
    }  
}
```

# Programming Language Concepts: Lifetime

- **Lifetime (Or Storage Duration)**
  - Duration **in** terms of program execution is where the contents of a variable is valid to reference in a C statement (by C language specs – not the OS!)
- **Important:** Linux may allow access to a memory location *even though* the language says you cannot reference the variable (local variables in particular) – later in course when we talk about **aliases**
  - ***This is core concept behind many security exploits***

## C Storage Durations

- C variables have one of the following lifetimes (durations)
  1. **Static Storage Lifetime:** valid while program is executing
    - Storage allocated and is initialized prior to runtime (implicit default is 0)
  2. **Automatic Storage Lifetime:** valid while enclosing block is activated
    - Storage allocated and is not implicitly initialized (garbage) by executing code when entering scope
  3. **Allocated Storage Lifetime:** valid from point of allocation until freed
    - Storage allocated by call to an allocator function (malloc() etc.) at runtime and is not implicitly initialized (garbage) - one allocator does initialize to zero at runtime calloc() – later in course
  4. **Thread Storage Lifetime:** valid while thread is executing (not CSE 30)

# C Variable Definitions

- **Global variables:** Defined **outside a function body**
  - Scope: valid from **point of definition** to end of file
  - **Static storage lifetime** - valid **while program is executing**
- **Local Variables & Function Parameters:** Defined **within a block**
  - Scope: valid from **point of definition** to the end of the code block where defined
  - **Lifetime: Automatic storage lifetime** - valid while enclosing block is activated

```
int global0;          /* global variable default initial value is 0 */
int global1 = 1;      /* global variable explicitly set to 1 */
void foo(int parm)   /* automatic parameter parm is "Local" to foo */
{
    int i, j = 5;    /* automatics i initial value is unknown, j is 5 */
    const int n = 5; /* automatic value of n cannot change */
    static int s;    /* block scope, static lifetime initial value: 0 */
    // body of code
}
```

# Static Storage Duration

- Variables defined **with the storage class specifier static** have **static storage duration** including variables with block scope
- Global variables **without the storage class specifier static** also **have static storage duration**
- All variables with **static storage duration are allocated space and initialized before execution starts** (default is 0)

```
% ./a.out  
2 3 4 5 6  
%
```

```
#include <stdio.h>  
#include <stdlib.h>  
int unused; //global static storage duration  
int  
foo(void)  
{  
    static int s=1; //static storage duration  
    return s += 1;  
}  
  
int main(void)  
{  
    for (int i = 0; i < 5; i++)  
        printf("%d ", foo());  
    return EXIT_SUCCESS;  
}
```

## Example of Variable Scope - 1

```
int j;                                // static storage; file scope of j begins
void x(int i) {                         // block scope of i begins
    int j = 1;                           // block scope of j begins; hides file scope of j
    i++;                                // refers to the function parameter
    for (int i = 0; i < 2; i++) { // block scope of block-local i begins
        int j = 2;                     // block scope of the inner j begins; hides outer j
        printf("%d\n", j); // inner j is in scope; prints 2
    }                                    // block scope of i and j ends
    printf("%d\n", j); // the outer j is in scope, prints 1
}
```

// block scope of i and j ends

## Example of Variable Scope - 2

```
#include <stdio.h>
#include <stdlib.h>
int itsGlobal = 0;

int
main(void)
{
    int itsLocal = 1;          /* Local to this function */

    printf("A: Global %d Local %d\n", itsGlobal, itsLocal);

    while (itsLocal != 2) {    /* which itsLocal here? */
        int itsLocal = 2;      /* Local to just this block */

        itsGlobal = 4;         /* changes global variable */
        printf("B: Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("C: Global %d Local %d\n", itsGlobal, itsLocal);
    return EXIT_SUCCESS;
}
```

```
% ./a.out
A: Global 0 Local 1
B: Global 4 Local 2
... infinite loop
```

# Header Guards Prevent Multiple Includes

contents of file else.h

```
#ifndef _ELSE_H_ // any unique name
#define _ELSE_H_
enum typestate {START, E, EL, ELS};
#endif
```

contents of file subs.h

```
#ifndef _SUBS_H_
#define _SUBS_H_
#include "else.h"
enum typestate {START, E, EL, ELS};
enum typestate STARTst(int);
enum typestate Est(int);
enum typestate ELst(int);
#endif
```

- Solution tell cpp to only include a file once
- Uses a macro definition (`#define`) with a conditional compile (`#ifndef` and `#endif`)

`#ifndef _ELSE_H_` is true only  
when `_ELSE_H_` is not defined

output of cpp on else.c

`#include "else.h"`

`#include "subs.h"`

```
int main(void)
{
    // rest of main
}
```

# Controlling Linkage Across Files in Multi-File C Programs

- Linkage determines whether an object (like a variable or a function) can be referenced outside the file it defined in
- External Linkage: function and variables with external linkage can be referenced anywhere in the entire program
  - Global variables and functions have external linkage by default
- Internal Linkage: function and global variables with internal linkage can only be referenced in the same file
  - Global variables and functions can be changed to internal linkage by using the keyword **static**
- No Linkage across files: function parameters, variables defined inside a function (without a keyword **extern** - declaration)

## Linkage Example - Simplified

file foo.c – definitions are here

```
int global0 = 1;          // external linkage
static int global2;       // internal linkage
int funcA(int x)         // external linkage
{
}
static int funcB(void) // internal linkage
{ }
```

- Use the keyword `extern` to declare a global variable
- Use keyword `static` in front of a global variable or function to make it private to the file it is defined in
- The two `funcB()` are different functions

file foo.h – declarations are here

```
extern int global0;
// cannot link to global2
int funcA(int);
// cannot link funcB in foo.c
```

file sub.c

```
#include "foo.h"
void funB(void) // ext link
{
    int x;    // no linkage
    x = funcA();
    x += global0;
    // and so on
```

# Reference Slide: Scope and Lifetime Summary

where defined	definition type	definition preceded with <b>static</b> keyword	name exported for linking to other files	definition category	scope	lifetime (of variables)	Default Initial Value	Where located in system memory
inside function or block	variable	no	no	local automatic variable (no linkage)	enclosing function or block	enclosing function or block activation	garbage (if initial value given, done by code at runtime)	stack segment
inside function or block	variable	yes	no	function/block private static variable (no linkage)	enclosing function or block	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
outside of function	variable	no	yes	global static variable (external linkage)	file; extendable to other files up to program	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
outside function	variable	yes	no	file private global static variable (internal linkage)	file	process lifetime	zero; Once before program start	data segment if initial value specified, bss otherwise
	function	no	yes	function (external linkage)	file; extendable to other files (up to program)			text
71	function	yes	no	file private (internal linkage)	file			text

x

# C Versus Java

Note: Sorry for the "poor" code indentation; adjusted to fit into the table

	Java	C
Overall Program Structure	<pre>source file: Hello.java</pre> <pre>public class Hello {     public static void main         (String[] args)     {         System.out.println(             "hello world!");     } }</pre>	<pre>source file: hello.c</pre> <pre>#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt;</pre> <pre>int main(void) {     printf("hello world!\n");     return EXIT_SUCCESS; }</pre>
Access a library	<pre>import java.io.File;</pre>	<pre>#include &lt;stdio.h&gt;</pre> <i>// may need to specify Library at compile time with -llibraryname</i>
Building	<pre>% javac Hello.java</pre>	<pre>% gcc -Wall -Wextra -Werror hello.c -o hello</pre>
Running (execution)	<pre>% java Hello</pre> <pre>hello world!</pre>	<pre>% ./hello</pre> <pre>hello world!</pre>

# C Versus Java

	Java	C
Strings	String s1 = "Hello";	char *s1 = "Hello"; // pointer version char s1[] = "Hello"; // array version
String Concatenation	s1 + s2 s1 += s2;	#include <string.h> strcat(s1, s2);
Logical ops	&&,   , !	&&,   , !
Relational ops	==, !=, <, >, <=, >=	==, !=, <, >, <=, >=
Arithmetic ops	+, -, *, /, %, unary -	+, -, *, /, %, unary -
Bitwise ops	<<, >>, >>>, &, ^,  , ~	<<, >>, &, ^,  , ~
Assignment ops	=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, ^=,  =	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=,  =

## C Versus Java

	Java	C
Arrays	<pre>int [] a = new int [10]; float [][] b =     new float [5][20];</pre>	<pre>int a[10]; float b[5][20];</pre>
Array bounds checking	<pre>// run time checking</pre>	<pre>// no run time checks - speed optimized</pre>
Pointer type	<pre>// Object reference is an // implicit pointer</pre>	<pre>int *p; char *p;</pre>
Record type	<pre>class Mine {     int x;     float y; }</pre>	<pre>struct Mine {     int x;     float y; };</pre>

## C Versus Java

	Java	C
if, switch, for, do-while, while, continue, break, return	// equivalent	// equivalent
exceptions	throw, try-catch-finally	// no equivalent
labeled break	break somelabel;	// no equivalent
labeled continue	continue somelabel;	// no equivalent
calls: Java method C function	f(x, y, z); someObject.f(x, y, z); SomeClass.f(x, y, z);	f(x, y, z); // other differences, later...

# C Programming Toolchain - Basic Tools

- **gcc**
  - Is a front end for all the tools and by default will turn C source or assembly source into executable programs
- **preprocessor**
  - Insertion into source files during compilation or assembly of files containing macros (expanded), declarations etc.
- **compiler**
  - Translates C programs into hardware dependent assembly language text files
- **assembler**
  - Converts hardware dependent assembly language source files into machine code object files
- **Linker (or link editor)**
  - Combines (links) one or more object files and libraries into executable program files
  - this may include modification of the code to resolve uses with definitions and relocate addresses

# C Programming Toolchain: The Source files

- The C development toolchain uses several different file types (indicated by .suffix in the filename)
- **filename.h** "*header or include files*" often used as `<filename.h>` or "filename.h"
  - a source text file whose contents need to be the same (constant) in other source files
  - **common contents**: function and variable declarations, and constants and language macros
  - Processed by **cpp** (the **C pre-processor**) to do inline expansion of the include file contents and insert it into a source file before the compilation starts, enables consistency
- **filename.c**
  - a source text file in **C language source**
  - Processed by **gcc**
- **filename.s**
  - a source text file in **hardware specific assembly language** (this is either programmer created or is machine generated by the compiler from a **.c** file)
  - processed by gcc which calls gas (assembler)

# C Programming Toolchain: The Generated files

- **filename.o "relocatable object file"**
  - Compiled from a single source file in a .c file or assembled from a single .s file into machine code
  - A .o file is an incomplete program (not all references to functions or variables are defined) this code will not execute
  - The .o and .c or .s files share the same root name by convention
  - created by gcc calling ld (linkage editor)
- **library.a "static library file"**
  - aggregation of individual .o files where each can be extracted independently
  - during the process of combining .o files into an executable by the [linkage editor](#), the files are extracted as needed to [resolve missing definitions](#)
  - created by ar, processed by ld (usually invoked via gcc)
- **a.out "executable program"**
  - Executable program (may be a combination of one or more .o files and .a files) that was compiled or assembled into machine code and [all variables and functions are defined](#)
  - processed by ld (usually invoked via gcc)

# Basic gcc toolchain usage

- Run gcc with flags
  - **-Wall -Wextra**
    - required flag for c programs in cse30
    - output all warning messages
  - **-c**
    - *Optional* flag (lower case)
    - Compile or assemble to object file only do not call **ld** to link
    - creates a **.o** file
  - **-ggdb**
    - *Optional* flag
    - **Compile with debug support** (gdb)
    - generates code that is easier to debug
    - removes many optimizations
  - **-o <filename>**
    - specifies *filename* of executable file
    - **a.out** is the default
  - **-S**
    - *Optional* flag (upper case **S**)
    - Compiles to assembly text file and stops
    - creates a **.s** file
- Producing an executable file
  - **gcc -Wall -Wextra -Werror mysrc.c**
    - creates an executable file **a.out**
- To use a specific version of C use of one the std= option
  - **gcc -Wall -Wextra -Werror -std=c11 mysrc.c**
- Producing an object file with gdb debug support add **-ggdb**
  - **gcc -Wall -Wextra -Werror -c -ggdb mysrc.c**
    - creates an object file **mysrc.o**
  - **gcc -Wall -Wextra -Werror -c -ggdb mymain.c**
    - creates an object file **mymain.o**
- Linkage step
  - combining a program spread across multiple files
  - **gcc -Wall -Wextra -Werror -o myprog mymain.o mysrc.o**
    - creates executable file **myprog**
- Compile and linkage of file(s) in one step
  - **gcc -Wall -Wextra -Werror -o myprog mysrc.c mymain.c**
- run the program (refer to cse15l notes)
  - **% ./myprog**

## Reference: printf(), fprintf() Formatting basics

- General form: `%m.pX`
  - `X` is a letter, indicating which conversion should be applied to the value before it is printed
  - `m` and `p` are optional integers
  - `p` is the number of digits after the decimal point for `e` & `f` formats or maximum number of chars to be printed for `s`
  - if `p` is omitted, the period that separates `m` and `p` is also dropped
  - `m`: minimum field width, specifies the minimum number of characters to print
- Often used conversion specifier `X` (one of the below)
  - `d`: an integer in decimal (base 10) form
  - `x`: an integer in hexadecimal (base 16) form
  - `e`: a floating-point number in exponential format (scientific notation)
  - `f`: a floating-point number in “fixed decimal” format, without an exponent.
  - `c`: a single character
  - `p`: a pointer address stored in pointer
  - `s`: the string (must be '\0' terminated) pointed at by a char pointer (or a string name) - later
  - "`\n`" prints a newline; "`\t`" prints a tab

# Software Layering Foundation Of System Implementation

- Linux runtime environment consists of code and data that **supports the semantics** defined by the **programming languages**
- Runtime environment spans **many parts of the system implemented in layers: Hardware, compilers, libraries and the OS**

