Version 1.6

UCSD CSE 30

Computer Organization and Systems Programming

Class Overview, PA 2 and Introduction to C

Lectures 1- 4 (first part of 4 only)

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**
  - **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

All free via library link

Weekly Canvas Reading quiz

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

Midterm – In Person

- **150** pts
- Oct. 27th evening in-person

Final – In Person

- **350** pts
- Dec. 9th 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

x

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 you 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

x

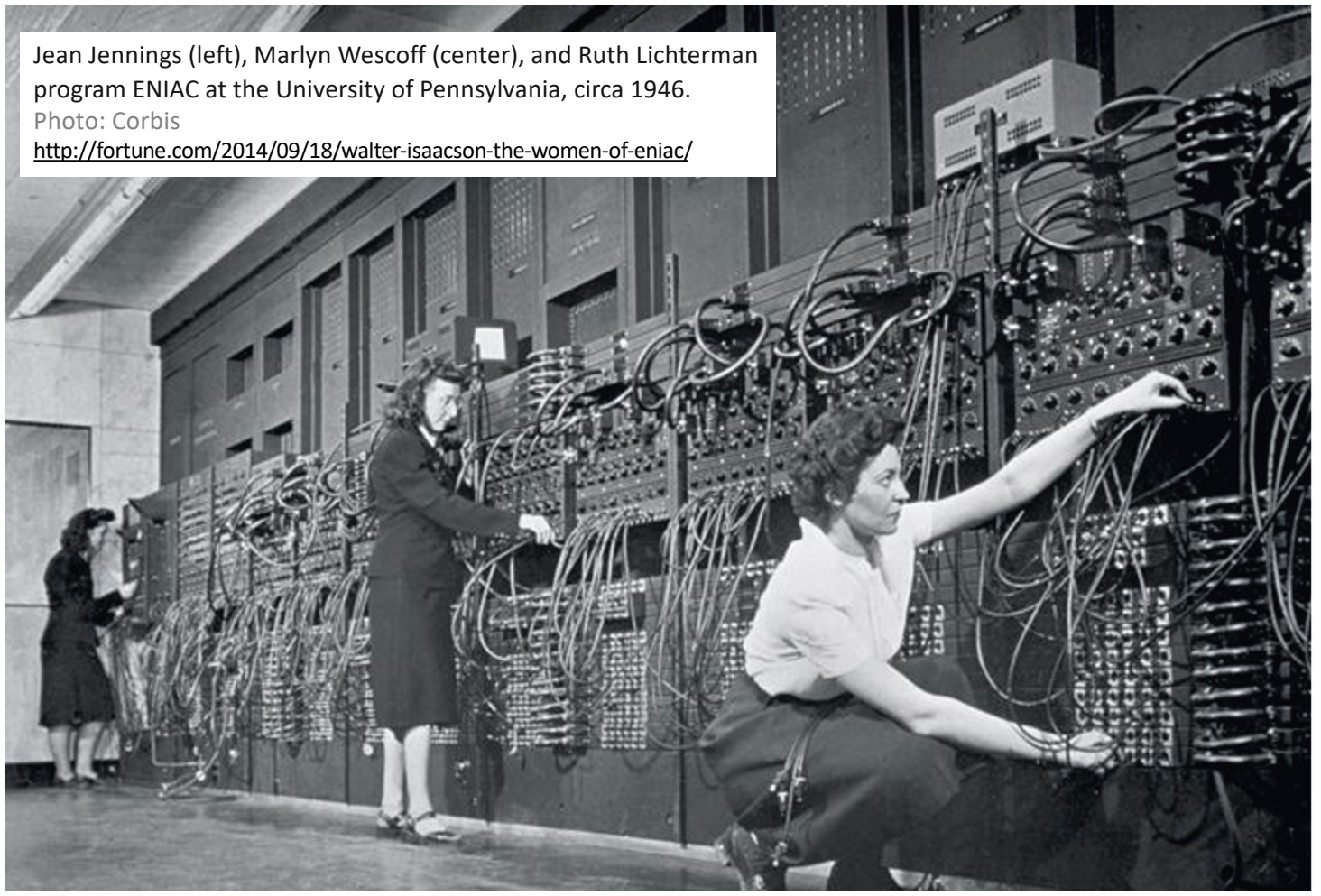
Additional Information

- **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 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
  - 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

x

Back in those Early Years…. Physically Wiring your ”Program”

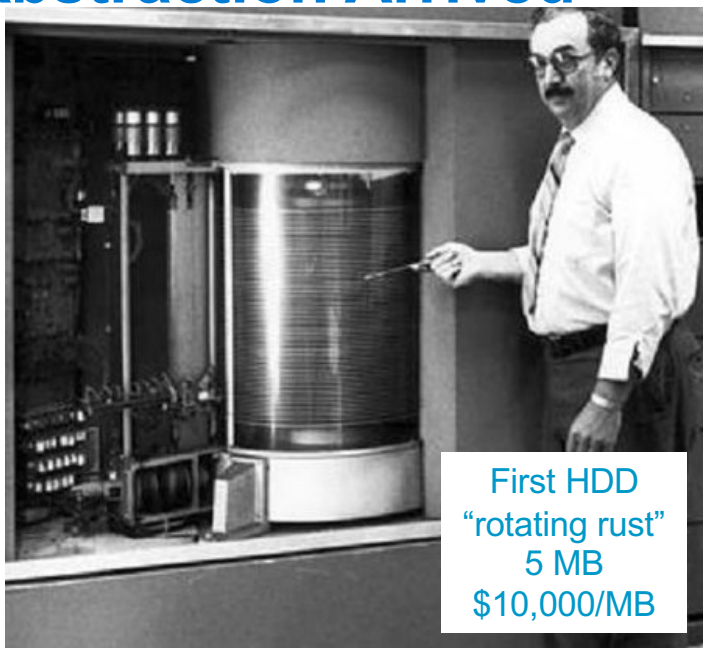
- Early Hardware
  - Used very simple CPU instructions (*primitives*)
  - e.g., a single instruction for adding two integers
- Software simple
  - Closely reflected the actual hardware it was running on
  - Specify each step manually
- How to program:
  - **Physical re-wiring** was a major part of programming

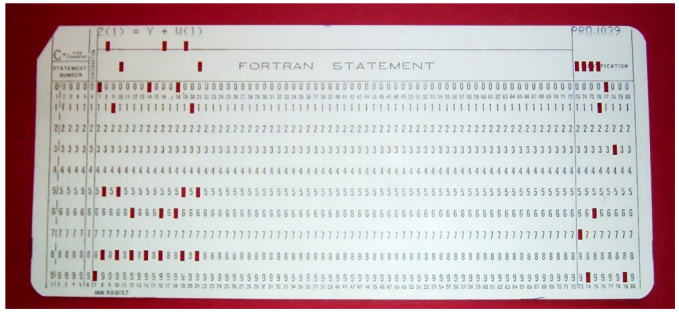


Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC at the University of Pennsylvania, circa 1946.
Photo: Corbis
http://fortune.com/2014/09/18/walter-isaacson-the-women-of-eniac/

X

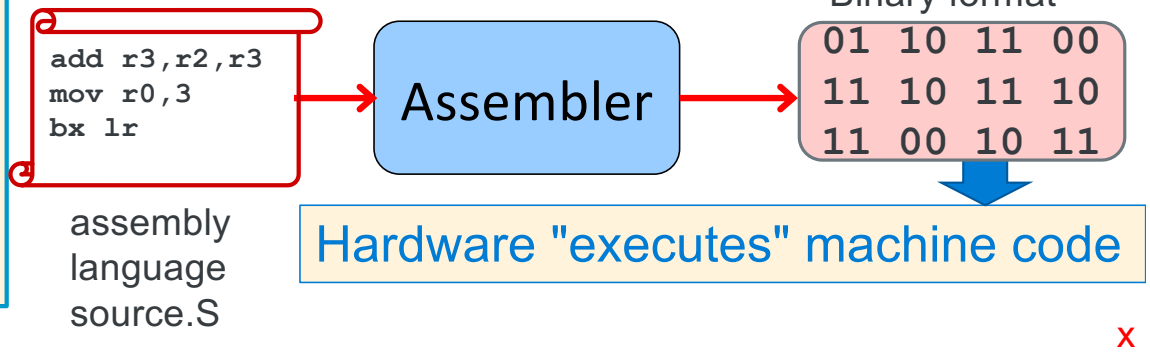
1950-1970 Computers – Abstraction Arrived





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

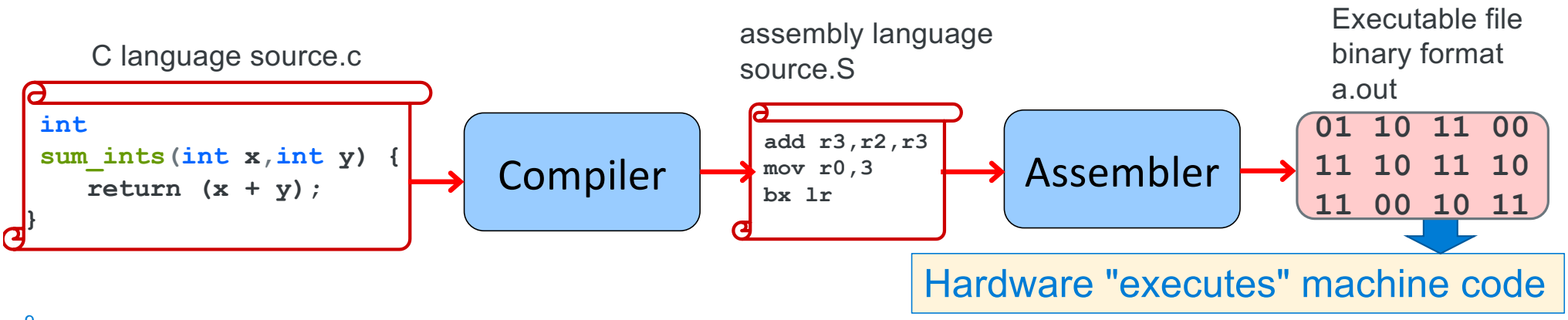
- Assembly Language Created
  - Programming Got Easier with assemblers
  - **1 assembly instruction = 1 machine hardware instruction**
  - More human-readable syntax



x

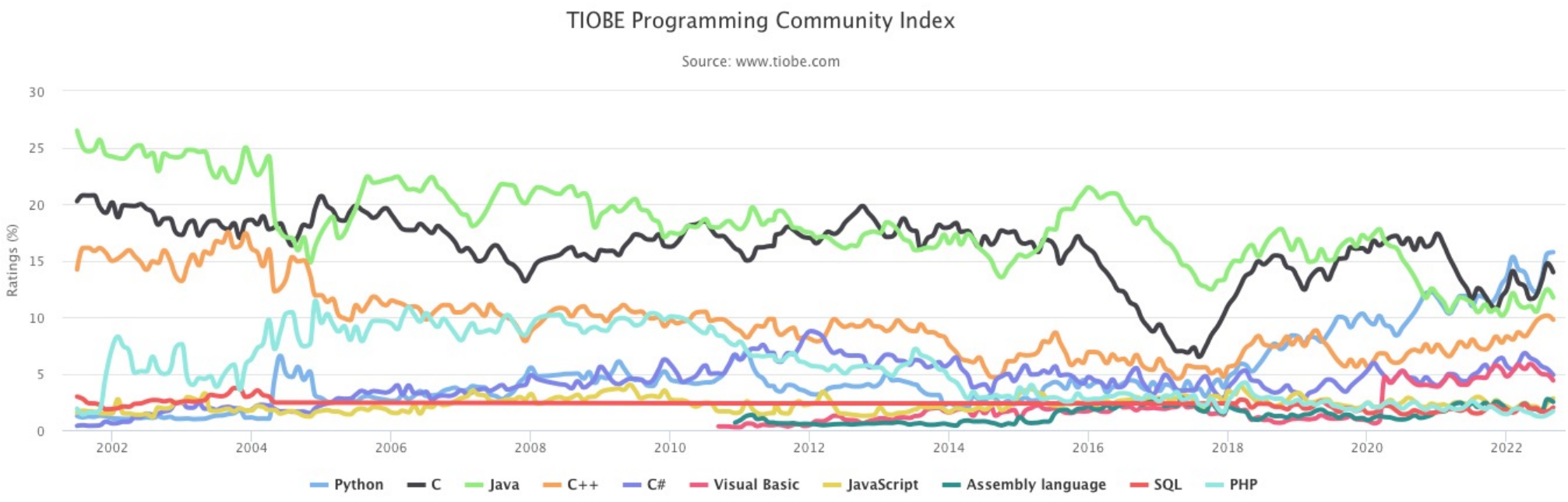
C Programming Language – High Level Language

- 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”



x

Programming Language Popularity



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

x

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

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

Performance Summary *(Ratios Best to Worse)*

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

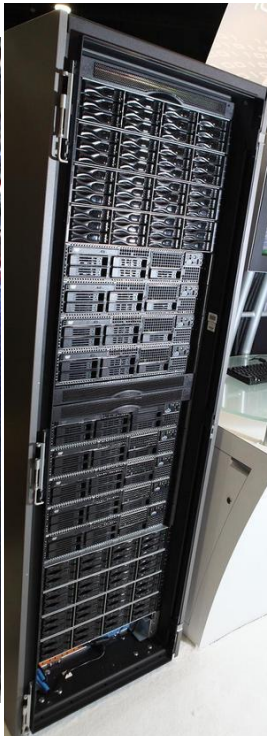
	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

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

x

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

```
int
sum(int x, int y) {
        return(x + y);
}
```

C language
Text File

Compile

```
sum_ints:
    .....
    add r3, r2, r3
    mov r0, r3
    bx lr
```

Assembly language
Human readable version of machine code **Text file**

Assemble

```
400575: 81 fe 89 32
        81 54 22 af
        81 22 10 9A
```

Machine code
Translation of Assembly
object File (.o)

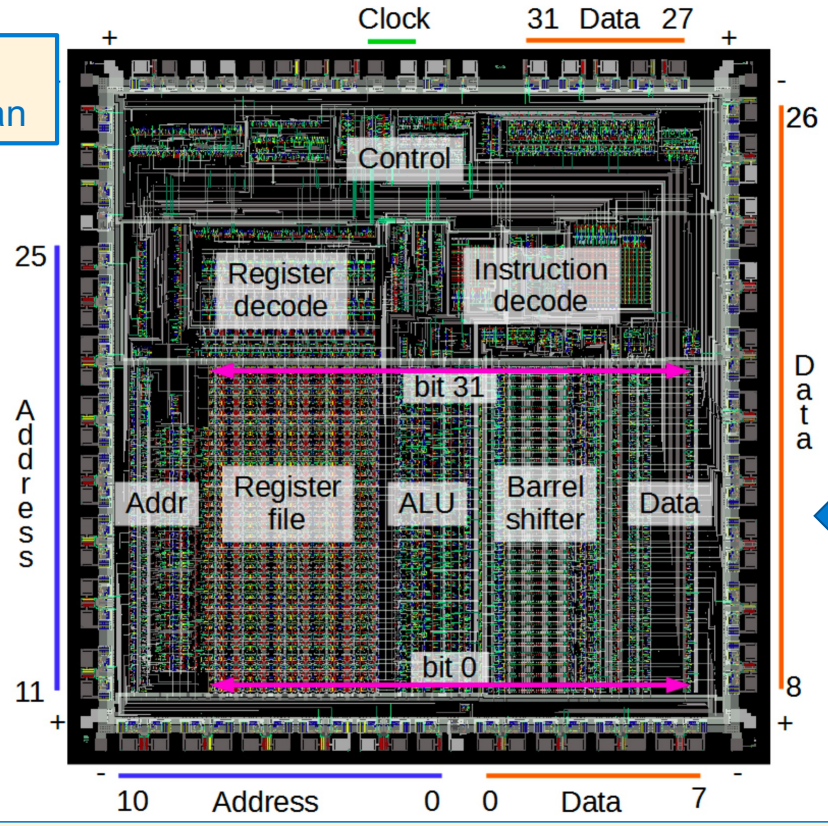
raspberry pi4

Raspberry Pi4B
programs execute under the Linux Operating system on ARM CPU's

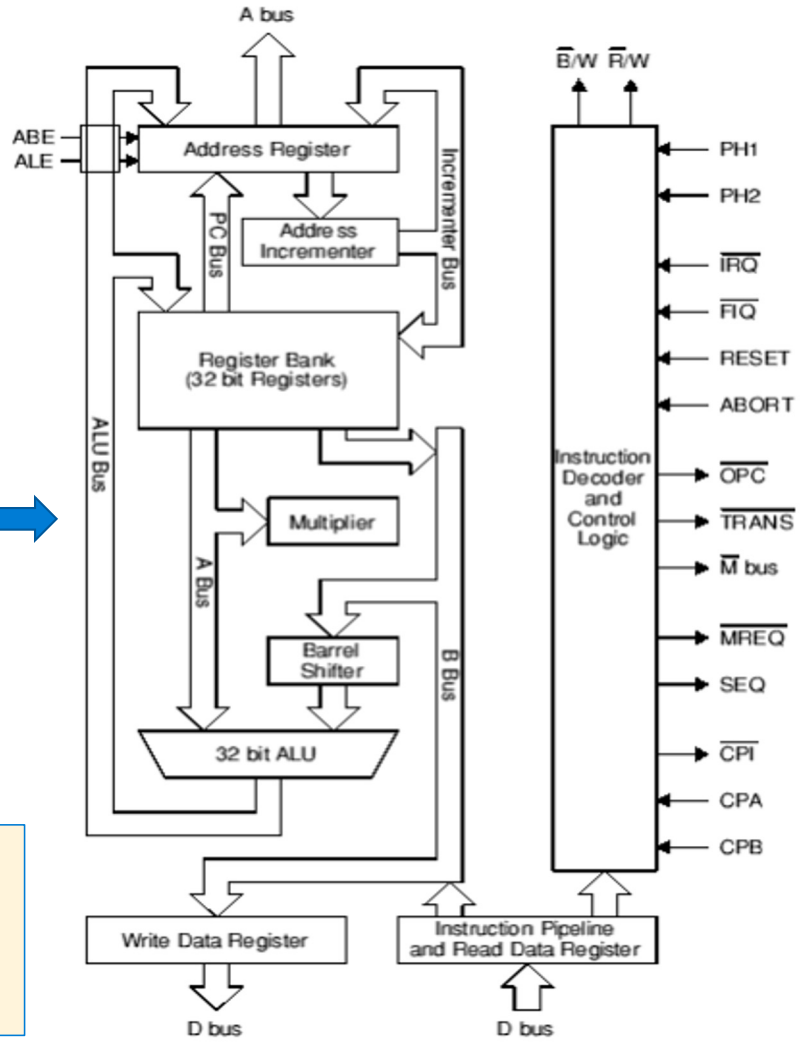
x

Course Goals – 3 (as time permits)

Single ARM Core Floorplan



CPU Datapath



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

x

Introduction: C Program Structure (Single file)

cpp directives

first function to run

```
#include <sys/system_files.h>  // /usr/include/sys/system_files.h
#include <system_files.h>      // /usr/include/system_files.h
#include "local_file.h"        //  local_file.h

/* 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 */
```

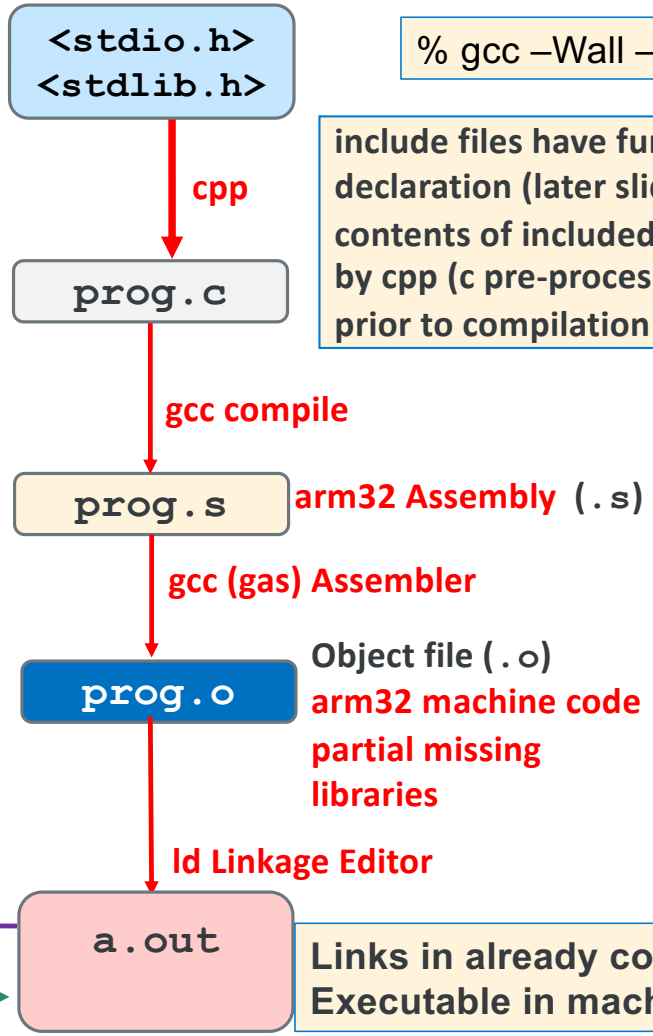
char literal '\n'

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

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



% gcc –Wall –Wextra –Werror prog.c

include files have function prototype and other declaration (later slides)
contents of included .h files are inserted/expanded by cpp (c pre-processor phase of gcc) into `files` prior to compilation start at their position in the file

Source to Execution Five Step Process

1. Compile
2. Assemble
3. Link
4. Load
5. Execute

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

Links in already compiled standard c library
Executable in machine code file

x

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

x

Compilation Process Actions

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


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

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

cpp (c pre-processor): removes the Comment

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

cpp: replaces EXIT_SUCCESS with 0

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)

x

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);
```


x

Main() Return Value Under Linux - 2

Example 1

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

Example 2

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

Example 3

```c
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;
}
```

x

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;`

x

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 */
```


x

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;
    }
}
```


x

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?

Which statement is executed after the if statement test? Depends on what value of i, is i zero or non-zero

```
int i;

// other code
if (i)
   statement1;
else
   statement2;
```

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

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

Assignment operators evaluate to the value that is assigned, so…. Which statement is executed after the if statement test?

x

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

x

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 (assigns first)
    i = (64, 323);      // i = 323 (value of expression)
```


x

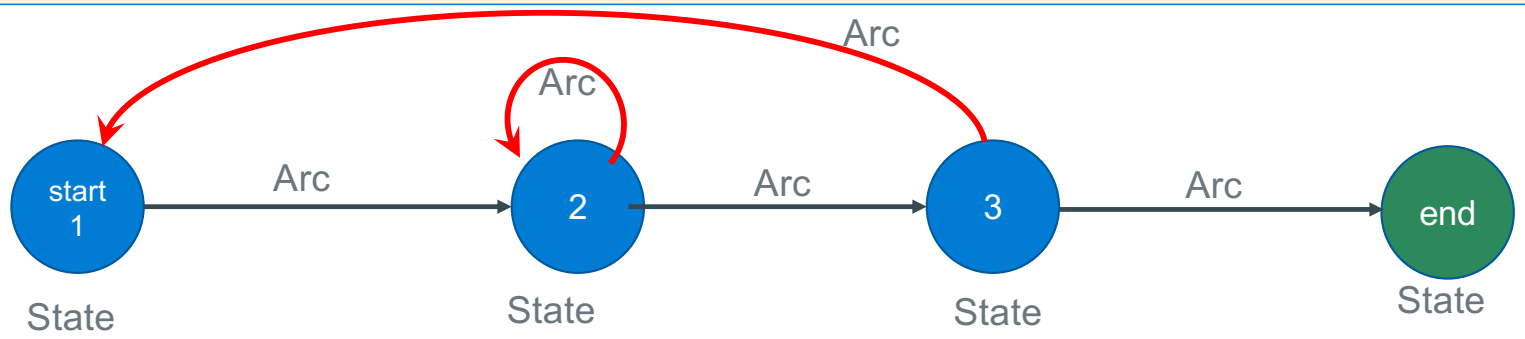
PA # 2: Designing a Deterministic Finite State Automaton

- 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

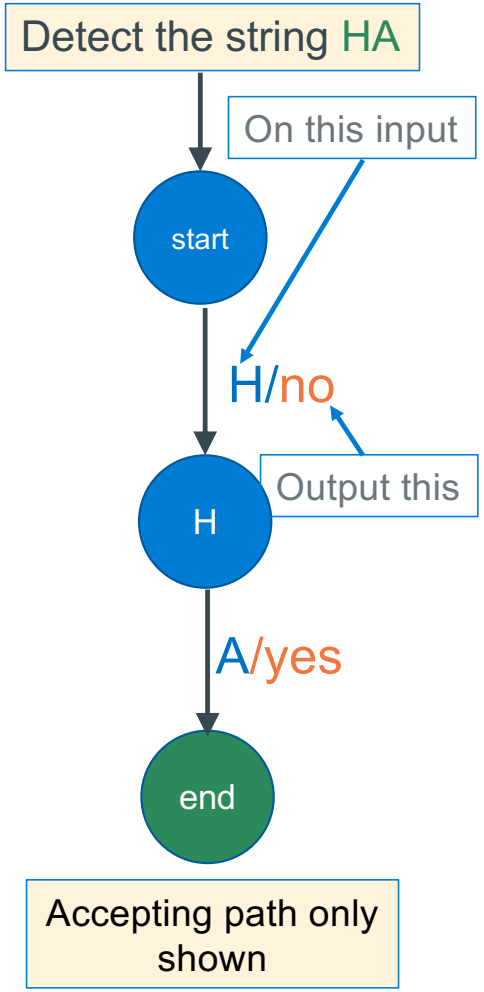
- A circle (state) represents memory of **what has already been seen before** based on the input
- An arc represents a **transition** from one state to the next state for a specified input
  - The next state can be the same state or a different state
- For all possible inputs, there must be just **one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



x

States and Transitions

- **States**: Two Special states
  - start **`start`** state (initial or first)
  - end 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)



x


Version 1.5
UCSD CSE 30
Computer Organization and Systems Programming
Class Overview, PA 2 and Introduction to C
Lecture 2 - Sept 27, 2022
Keith Muller
DEC PDP 11/45 - 1973

PA2/PA3: Removing comments from a C program

```c
#include <sys/system_files.h>  // /usr/include/sys/system_files.h
#include <system_files.h>      // /usr/include/system_files.h
#include "local_file.h"        //  local_file.h

/* 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! /* not a comment*/\n");
    // C++ comment \
      printf("Hello Tritons\n");

return EXIT_SUCCESS;      // for linux scripts; or EXIT_FAILURE
}
```

char literal '\n'

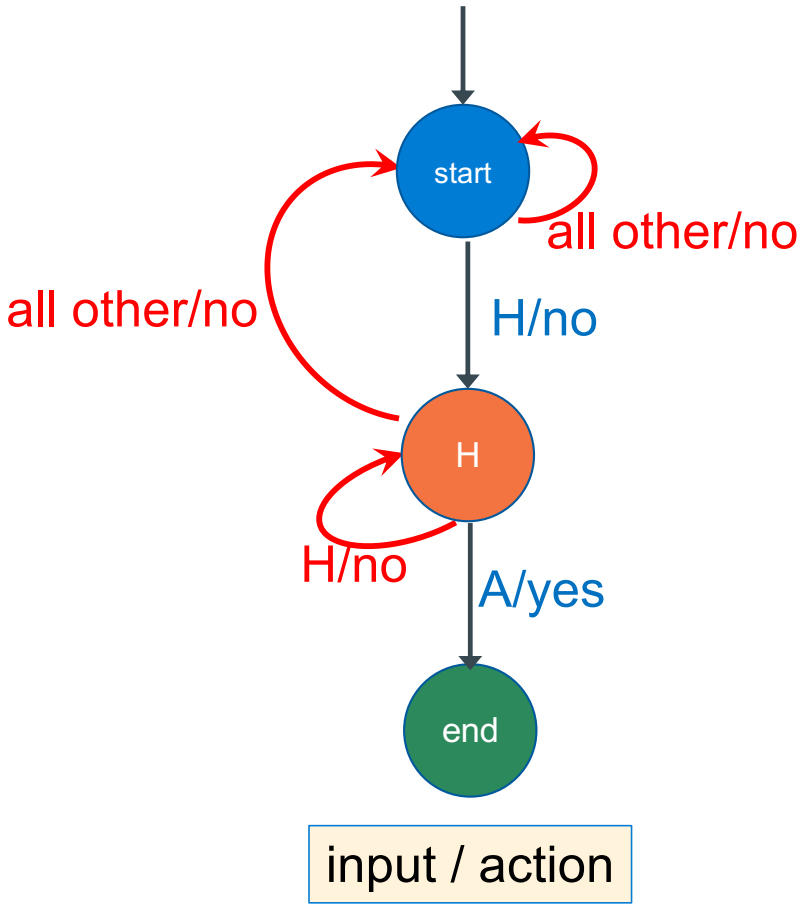
string literal "Hello World! /* not a comment */\n"

\ immediately followed by a newline is a line continuation; turns the line printf("Hello Tritons\n"); into a comment

x

Designing a Deterministic Finite State Automaton -2

- Problem: detect the string *HA*
- Complete the sequence from 3 slides back to make it deterministic
  1. Specify responses on all possible inputs at each state
  2. Make sure every next state transition (arc) is *unambiguous* (unique – each input selects only one arc)
- Recap: Each state represent the memory (history) of inputs in sequence necessary to reach that state
  - ***H state***: have seen an H
  - ***end state***: have seen an H immediately followed by an A



x

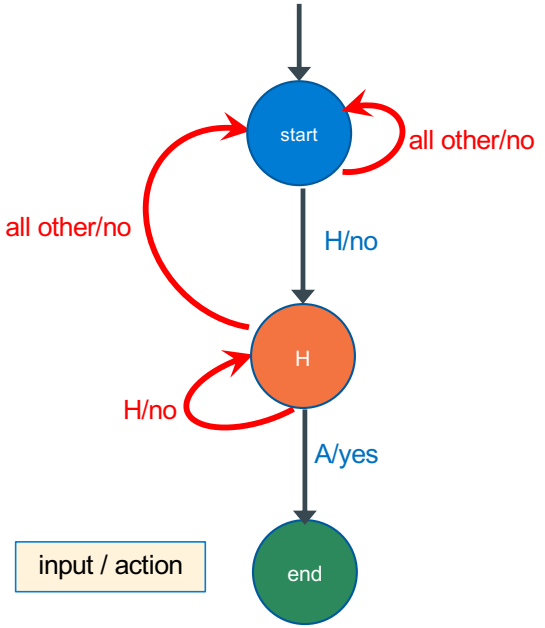
Designing a Deterministic Finite State Automaton - 3

- The state machine on the previous slide would stop after seeing the first HA, and does not take any more input, missing later occurrences of HA in the input
- Say you want to process the entire contents of a text file to find all HA's
  - from the top (top of file)
  - to the bottom (end of file)
- **Action:** *Alter* the machine to process input from a text file until the end of the file (EOF) is reached (you process the entire 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.

TOF

EOF



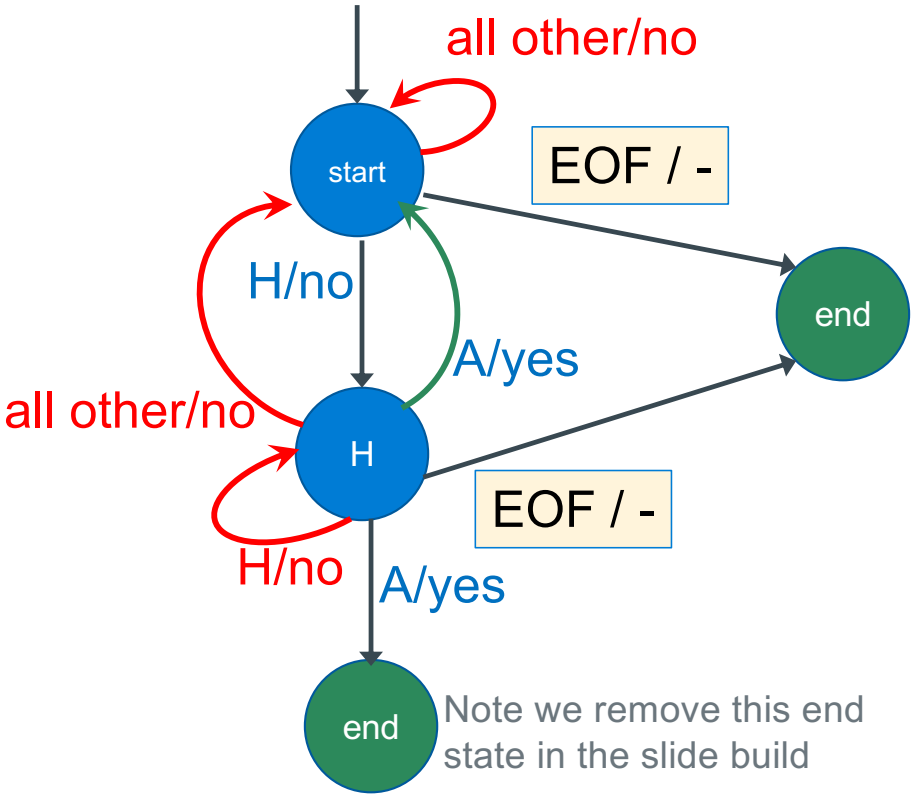
x

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

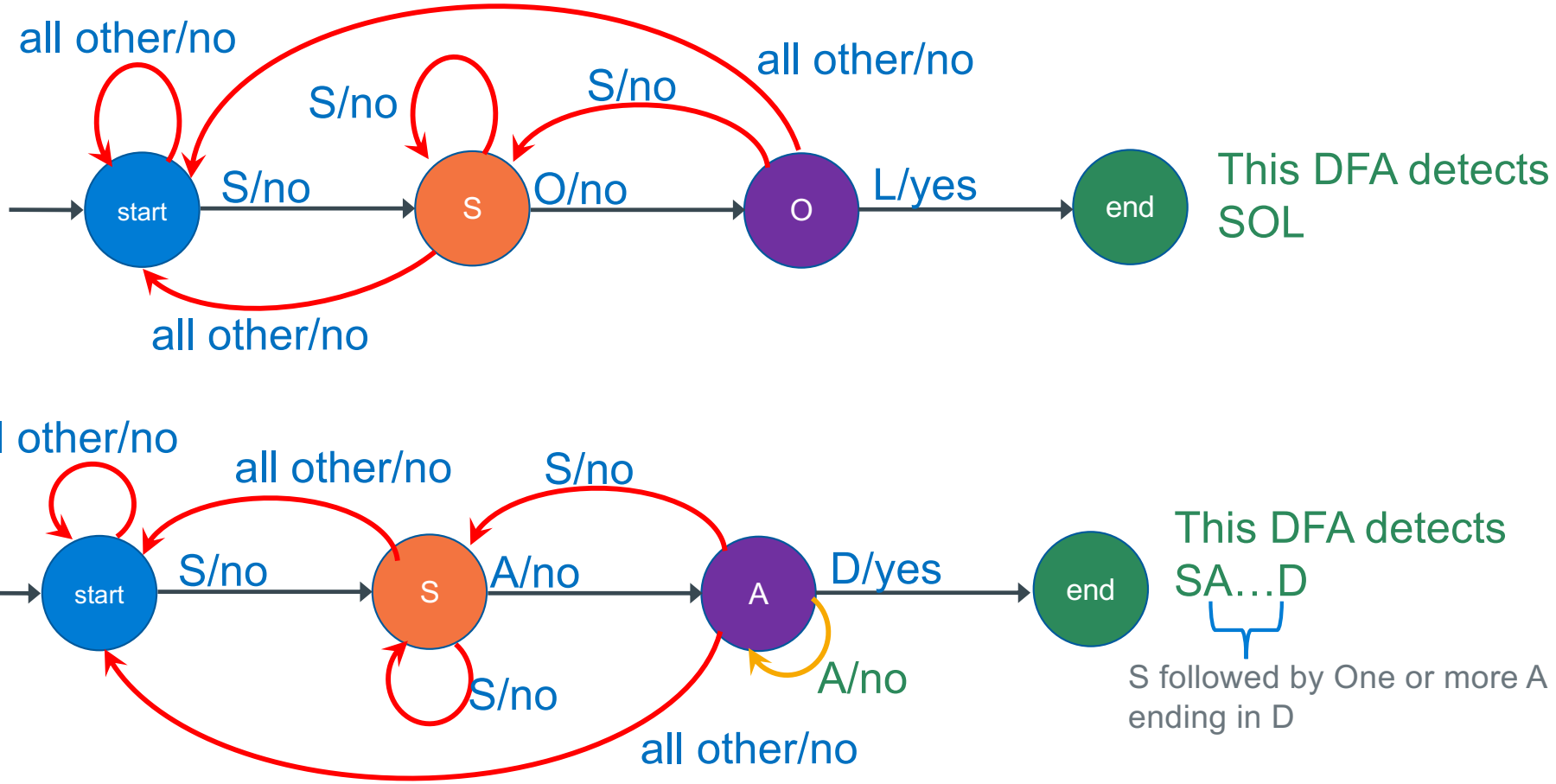
1. Delete the existing end state at the bottom (delete shown in the slide build)
2. Add arcs from each state when EOF on input is detected to a new end state

- Remember: Transition (arc) actions (zero or more) associated with a single arc can be anything you need to do
  - Example: output a string, perform a computation or do noting, etc.



x

Merging DFA's: Step one design each sequence



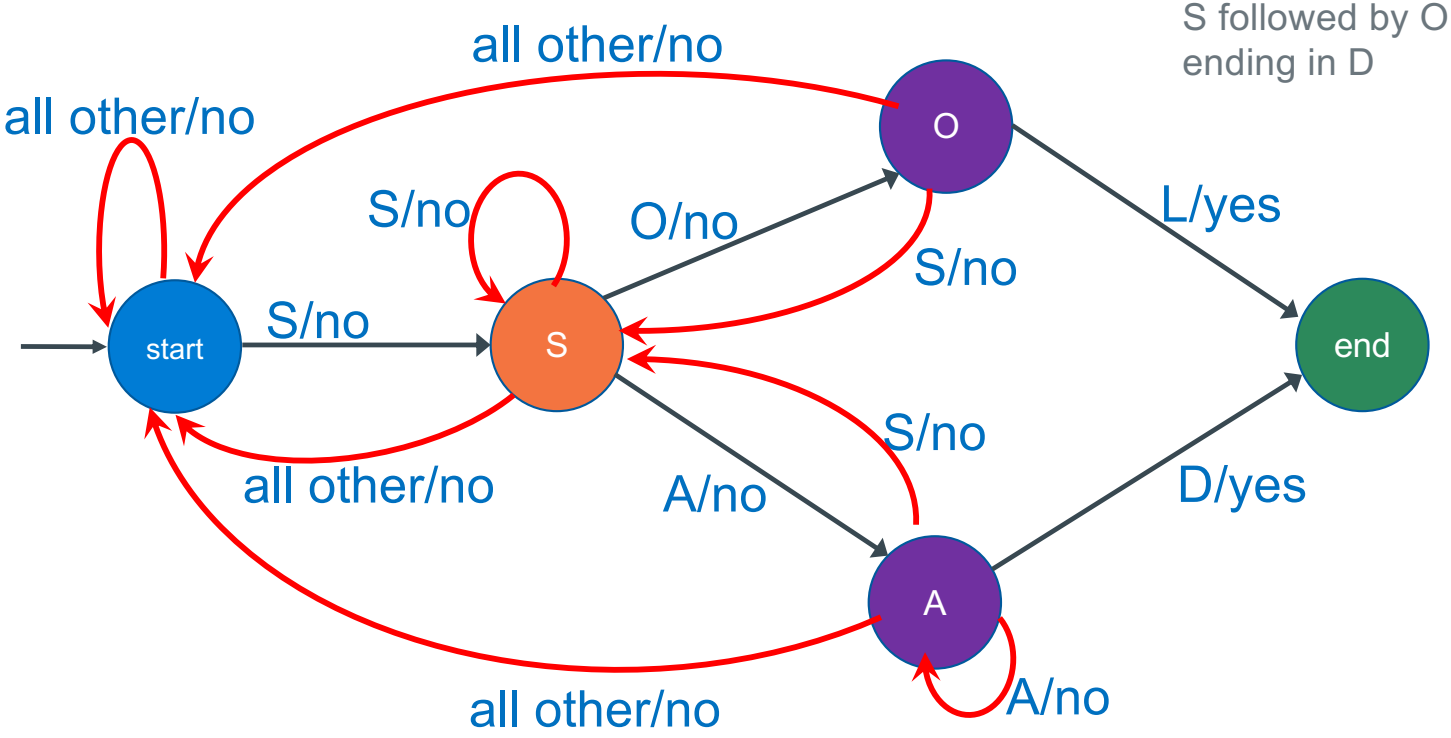
Merge the start states, share initial common sequences (First two states are the same in both)

x

Merging DFA's – 2 (Finished)

This DFA detects both SOL and SA…D

S followed by One or more A ending in D



x

Background: What is a Definition in a program language?

- **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 definition (compiler actions)**
  1. **creates code** you wrote in the functions body
  2. **allocates** memory to store the code
  3. **binds** the function name to the allocated memory
- **Variable definitions (compiler actions)**
  1. **allocates memory:** generate code to allocate space for local variables
  2. **initialize memory:** generate code to initialize the memory for local variables
  3. **binds (or associates)** the variable name to the allocated memory

x

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;
}
```


x

C Function Definitions - 2

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

- A 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 an ignored parameter list
- At runtime, 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 in their body without side effect to the calling function
  - C Parameter passing is called: call by value

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

int
sum(void)      // or sum()
{
  int i, total = 0;

  for (i = 1; i <= MAX; i++) {
    total += i;
  }

  return total;
}
```


x

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
    {
    }
}
```


x

Real C programs are distributed across multiple files

- **Large programs** in one source file can be very difficult to manage
  - Consider a program with 10 million lines of code
  - And there are many developers working on it
- Approach: C supports programs divided across multiple files (called *independent translation units*)
- While a file is being processed by the compiler, the following is required to generate the assembly:
  - variables
    - type, location of variable (adjusted later)
  - functions
    - return value type, Location of code (adjusted later)
    - Number and type of arguments to a function

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;
}
```

x

Background What is a Declaration in programming language?

- **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**

x

C Translation unit -single source file

Range of definition and declaration validity

sum() is defined and declared here

Default Definition and declaration validity:

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

sum() is used here

```c
// 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: %d\n", sum(MAX));
  return EXIT_SUCCESS;
}
```

x

Function Prototypes –Definitions and Declaration Sections

```
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
```

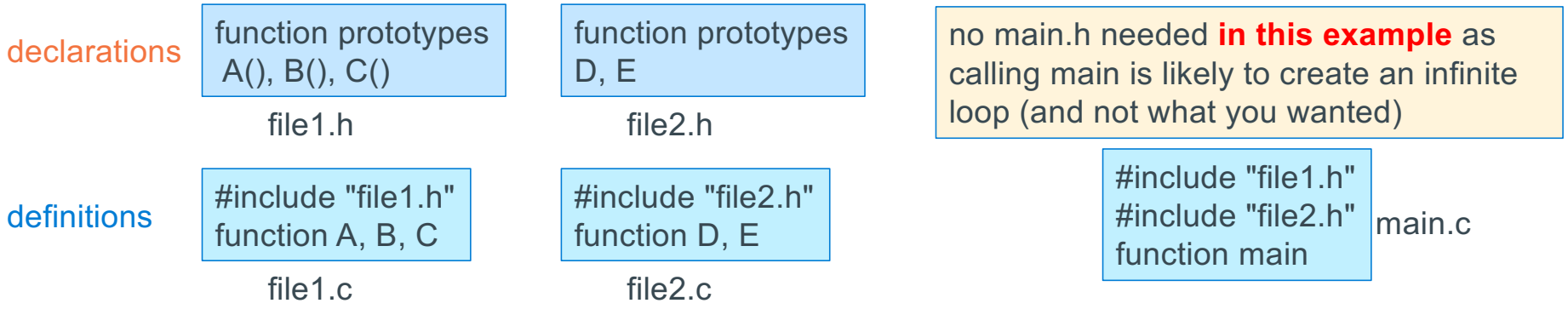
this is the code block

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


x

Breaking a Program into Multiple files: Declaration (Header) Files and Definition (Source) files

- Desired Effect: Say we want to distribute the functions in a program across several source files (translation units), yet allow the functions to call each other
- Approach: For each source file, `file.c` create a header file named `file.h`
  - `file.h` contains just the declarations of the functions (and other things, later…) defined in the source file, `file.c` that you want visible (exported) for use (called) by functions in other .c files



- Implementation: To call a function defined in another file compiler must know the types and parameter to generate code, so use a #include to import the other source files declaration file
  - #include the declaration file in your source file at **the top of the file**; this imports the declaration

x

Header Files (often called .h files) – Where to put declarations

- Header file: a file whose only purpose is to be **`#include`'d**
  - Declares interfaces (function prototypes, user defined types, global variable, macros, etc.)
  - Standard convention (strongly enforced), 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"
    ```
  - Import the interfaces in a C source file by `#include`-ing it's 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
- All #includes should go at the top of the file before anything else in the file

x

Placing Declarations and Definitions across Multiple files

main.c

```c
// sum of integers from 1 to MAX
#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;
}
```

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

sum.c

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

inside include files

header (or include) guard labels are as follows:

1. filename in all caps
2. replace the period in the files name with an _

Why needed:

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

header guards (two lines)

header guards (one line)

sum.h

```c
#ifndef SUM_H
#define SUM_H

int sum(int); // function declaration!

#endif
```

x

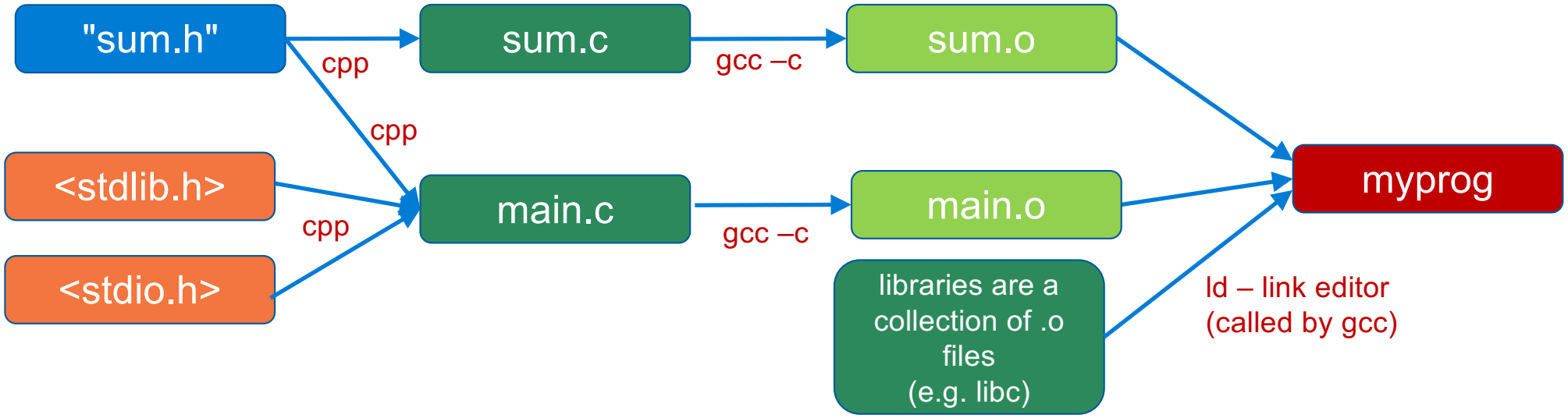
Breaking a Program into Multiple files: Separate Compilation Support

- 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: Compiler support for independent translation units is called: partial compilation
  1. Compile option to just create a machine code version for each file by itself
     - file may not be 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
     - Exception: code for dynamically linked libraries (you will learn about these in CS120 Operating systems), is linked at run time (when the machine code file is copied into memory and executed)

x

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
```


x

ASCII Use on Linux

\0 in c encodes a null

\b in c encodes a backspace

\t in c encodes a horizontal tab

\n in c encodes a linefeed

Ascii	Char	Ascii	Char	Ascii	Char	Ascii	Char
0	Null	32	Space	64	@	96	`
1	Start of heading	33	!	65	A	97	a
2	Start of text	34	"	66	B	98	b
3	End of text	35	#	67	C	99	c
4	End of transmit	36	$	68	D	100	d
5	Enquiry	37	%	69	E	101	e
6	Acknowledge	38	&	70	F	102	f
7	Audible bell	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Horizontal tab	41	)	73	I	105	i
10	Line feed	42	*	74	J	106	j
11	Vertical tab	43	+	75	K	107	k
12	Form feed	44	,	76	L	108	l
13	Carriage return	45	-	77	M	109	m
14	Shift in	46	.	78	N	110	n
15	Shift out	47	/	79	O	111	o
16	Data link escape	48	0	80	P	112	p
17	Device control 1	49	1	81	Q	113	q
18	Device control 2	50	2	82	R	114	r
19	Device control 3	51	3	83	S	115	s
20	Device control 4	52	4	84	T	116	t
21	Neg. acknowledge	53	5	85	U	117	u
22	Synchronous idle	54	6	86	V	118	v
23	End trans. block	55	7	87	W	119	w
24	Cancel	56	8	88	X	120	x
25	End of medium	57	9	89	Y	121	y
26	Substitution	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	File separator	60	<	92	\	124	|
29	Group separator	61	=	93	]	125	}
30	Record separator	62	>	94	^	126	~
31	Unit separator	63	?	95	_	127	Forward del.

ASCII Chars are 0-127 (stored in 8 bits)

X


Version 1.6
UCSD CSE 30
Computer Organization and Systems Programming
Class Overview, PA 2 and Introduction to C
Lecture 3 - Sept 29, 2022
Keith Muller
DEC PDP 11/45 - 1973

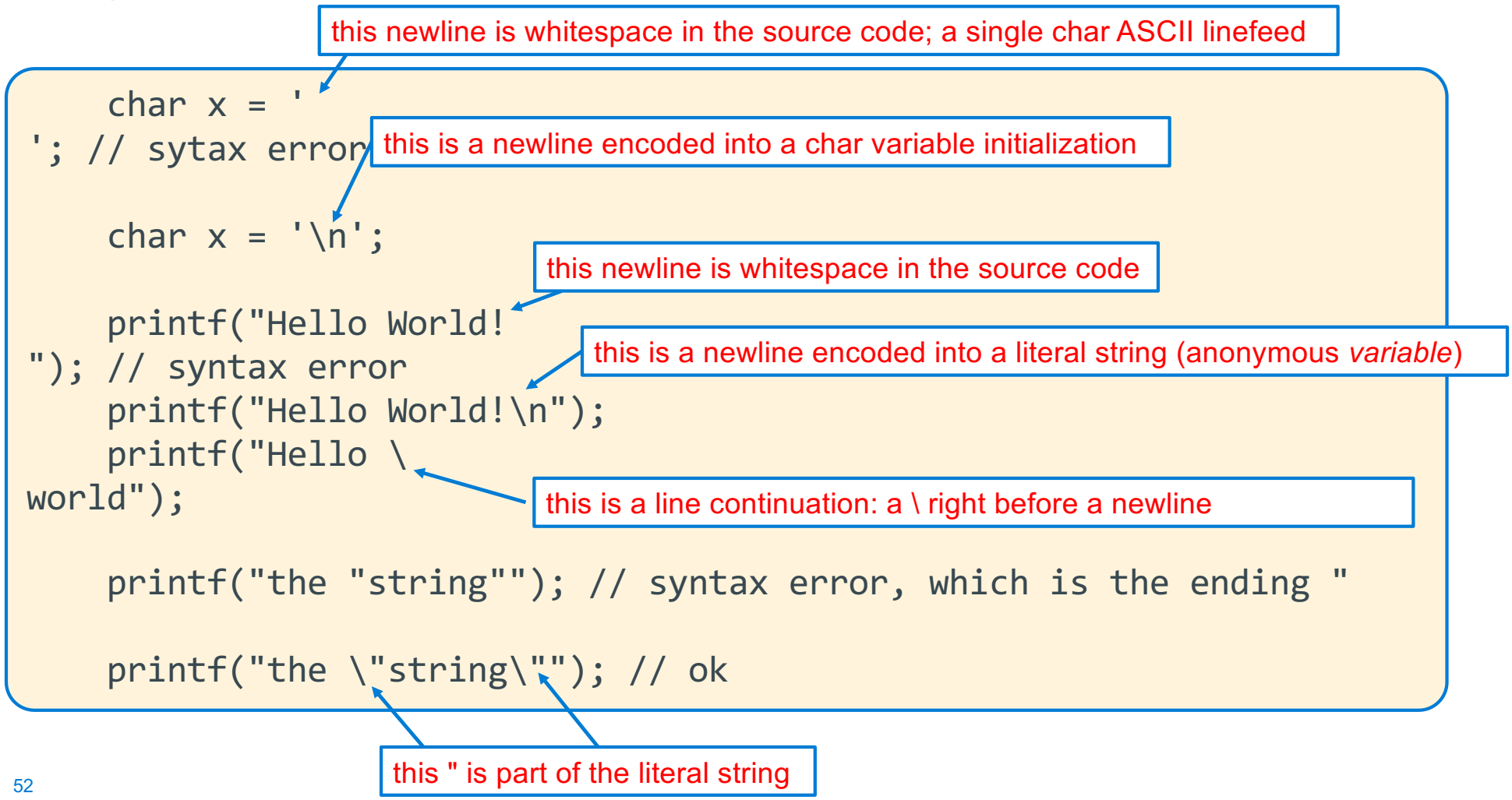
char & string literals In C

- Usually used to store characters – thus things like file names
- char literals: single quotes 'a'
- string literals: Double quotes "string"
- Three uses for \ in a C source file
  - Characters inside literals that the compiler acts on (all forms of white space, a null) are encoded using a two-character sequence starting with a \ to be used within a literal
  - to escape the special meaning of the next char such as: \' , \", \\ inside literals
  - Line continuation char \ before a newline at the end of a source line

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

x

Why literal sequences?



x

Recap: Benefit of Separate Declarations and Definitions

- Using function protypes, putting them into a header file, and including the header file has the following advantages:
  1. Allows any order of function use as the functions are all declared from the point in the file where the #include is located until the end of the file (why we put #include at the top of the file)
  2. Allows the functions to be called from functions defined in a different file
  3. Facilitates separate compilation, as refences to function definitions do not need to be resolved until the object files are recombined by the linker
- In the examples to the right
  - The helper function funcA() in main.c, has a function prototype in main.h, so funcA() does not have to be defined before main() in main.c
  - The function funcB() which is defined elsewhere, can be used by #including subs.h

subs.h

```
#ifndef SUBS_H
#define SUBS_H
int funcB(int);
#endif
```

main.h

```
#ifndef MAIN_H
#define MAIN_H
int funcA(int);
#endif
```

main.c

```
#include "main.h"
#include "sub.h"
int main(void) {
    /* lots of code */
    x = funcA(y);
    return EXIT_SUCCESS;
}
int funcA(int z)
{
        return funcB(z);
}
```


x

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: function parameters, variables defined inside a block (including a functions body)

x

Linkage Example - Simplified

file foo.c

```c
#include "foo.h"
int global0 = 1;          // external linkage
static int global2;       // internal linkage
int funcA(int x)          // funcA has external linkage
{
    int y;               // no linkage
}
static int funcB(void) // internal linkage
{ }
```

file foo.h

```c
extern int global0;
int funcA(int);
```

- Keyword extern to "extend the visibility", e.g. declare a global variable defined elsewhere

```c
extern int x; // declaration
int x = 10;  // definition
```

- Use of the keyword static in front of a global variable or function definition changes it to internal linkage and effectively makes it private to the file they are defined in
- Function definitions in different files (translation units) can re-use the same name if at most one has **external linkage (all others must be internal linkage)**

x

C Storage Durations

- C variables have one of the following lifetimes (durations)
  1. **Static Storage Lifetime:** valid while program is executing
     - Storage allocated and initialized **prior to runtime** (implicit default = 0)
  2. **Automatic Storage Lifetime:** valid while enclosing block is activated
     - Storage allocated and is not implicitly initialized (value = 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 (value = 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)

x

Static Storage Duration

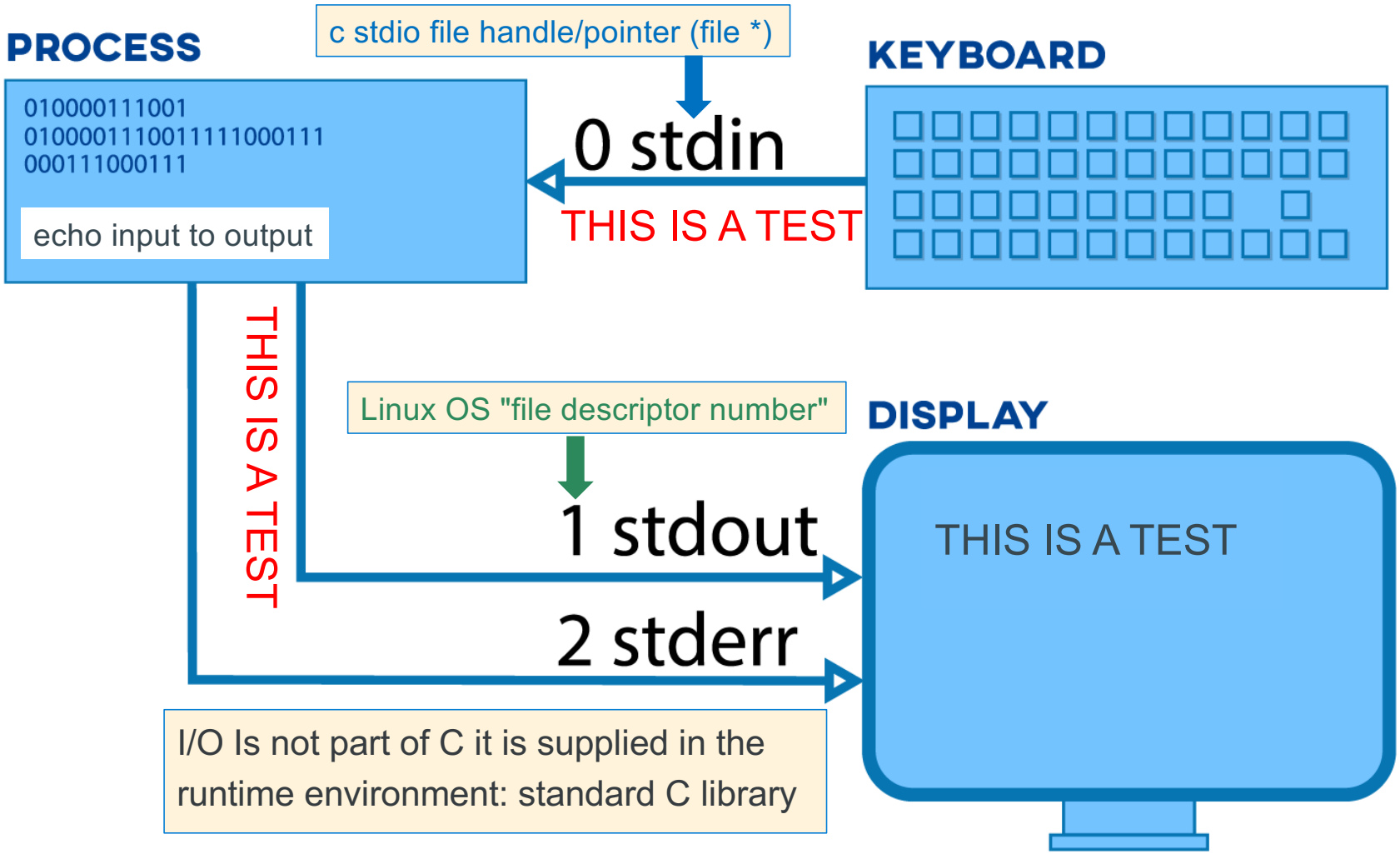
- Sadly, the keyword static has more than one meaning (in addition to controlling linkage) when used with variables
- Variables defined **with the storage class specifier** *static* always have static storage duration including variables with block scope
- Global variables **without the storage class specifier** *static* also have static storage duration
- All variables with static storage duration are allocated space and initialized before execution starts (default is 0)

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

int unused; //global static storage duration

int foo(void)
{
    static int s=0; //static storage duration
    return s += 1;
}

int main(void)
{
    for (int i = 0; i < 5; i++)
        printf("%d ", foo());
    return EXIT_SUCCESS;
}
```

```
% ./a.out
1 2 3 4 5
%
```

x

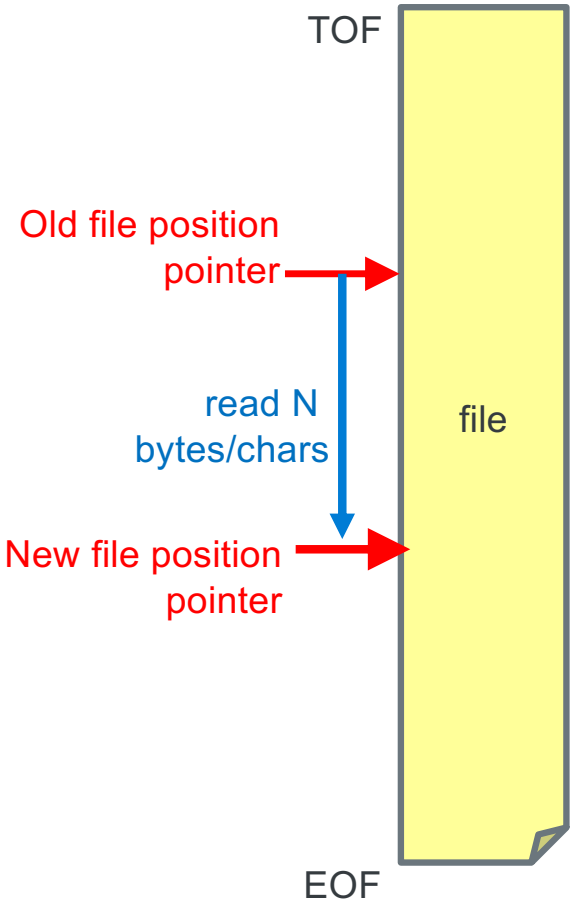
Linux/Unix Process and Standard I/O (CSE 15L) - Defaults



X

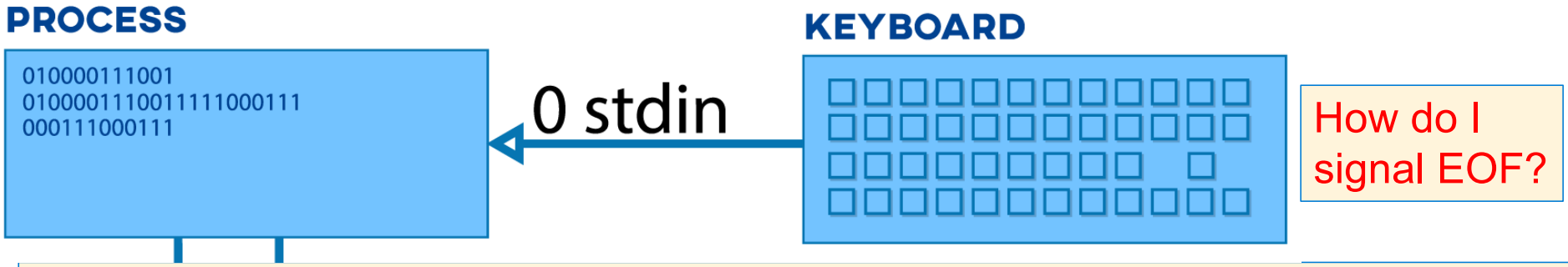
Under the Hood: stdio File I/O – File Position Pointer On Files

- 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
- 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: specified count of bytes at the file pointer
    - When file pointer is positioned after the last byte *extends the length* of the file (increases file length)



x

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



- How can you have an 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 (it is waiting for you to type a line):
- To set 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:**

  cntrl-d *(often shown in slides etc. as ^d)*

x

C Library Function: Simple Formatted Printing

Task	Example Function Calls

Write formatted data	`int status;`
	`status = fprintf(stderr, "%d\n", i);`
	`status = printf("%d\n", i);  /* Writes to stdout */`

```
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*

x

Some 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
```


x

C Library Function API : Simple Character I/O – Used in PA3

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

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

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

- writes c (demoted to a char) **to** `stdout`
- returns either: 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) **promoted to an int** read **from** `stdin`
- see % man 3 getchar

- Make sure you use int variables with putchar() and putchar()
- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition (**-1) is outside the range of valid characters

Why is character I/O using an int?

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

x

Simple Character I/O – Using STDIO

Make sure you use int variables here!

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

int main(void)
{
    int c;

    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

Why is c an int?
Returns char in an int or EOF (-1)

```
% ./a.out
thIS is a TeSt
thIS is a TeSt
^d
%
```

Typed on keyboard

Printed by program

Typed on keyboard

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

x

C Enumerated Data Type – Use to Specify DFA States

```
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

x

C Enumerated Data Type

- **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: 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

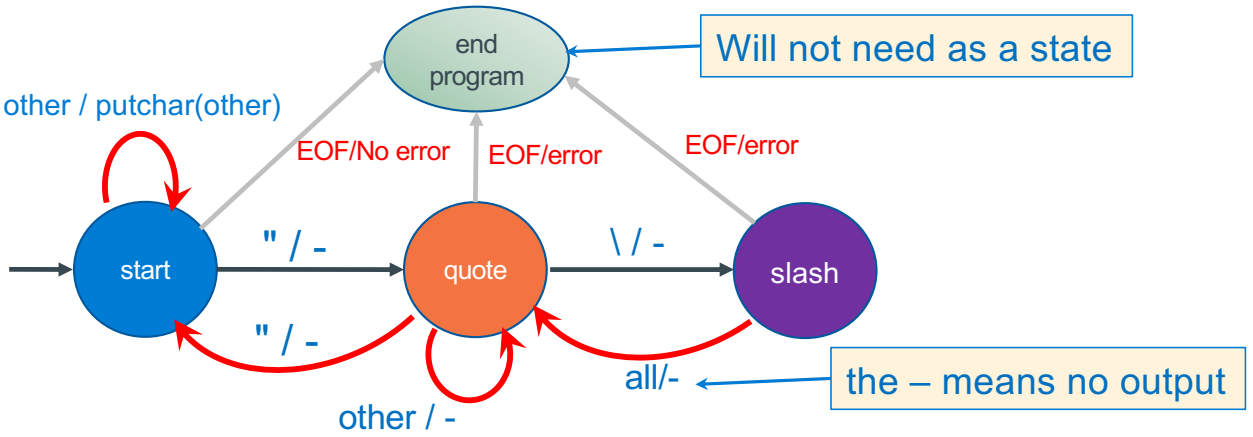
```
enum compass direction = WEST; // direction defined & initialized to WEST
```

.c file

x

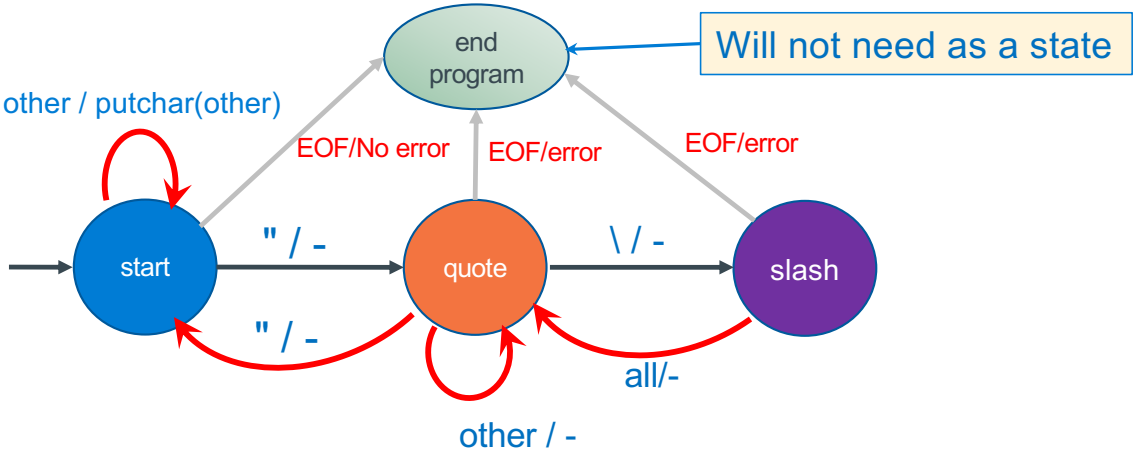
Example: Programming a Deterministic Finite Automaton

```
Rules for this DFA example
Copy input to output while removing everything in "strings" from output

      input:  ab"foo"cd
      output: abcd

Special Case: If Inside a string, a \ is an escape sequence, ignore the next char
            Allows you to put an " in a string

      input:  ab"foo\"bar"cd
      output: abcd
```



x

Programming a Deterministic Finite Automaton



- 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);
```

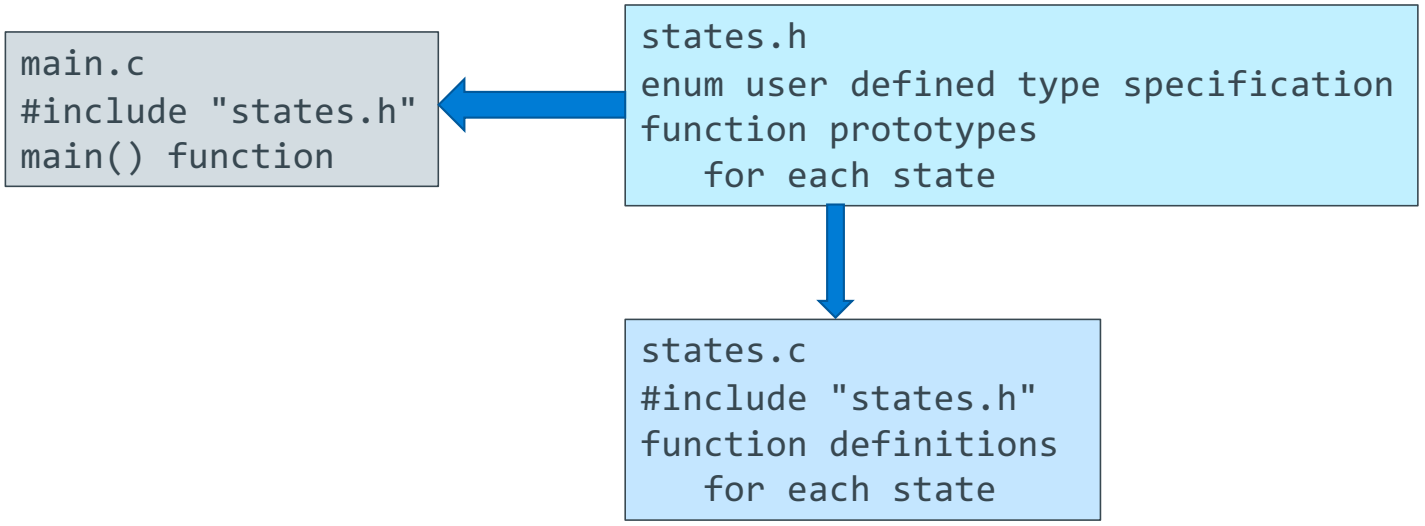
- Each function implements the outbound arcs
  1. returns the next state based on the next input
  2. performs actions associated with arc taken

```
// process input until EOF in main()

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


x

Programming a Deterministic Finite Automaton – The Files

- Break the program into three files
- `main.c:` is where main loop is, imports the declarations in `state.h`
- `states.h`: is the interface to the state handlers in `states.c`
- `states.c:` definition of the state handler functions



x

Version 1.6

UCSD CSE 30

Computer Organization and Systems Programming

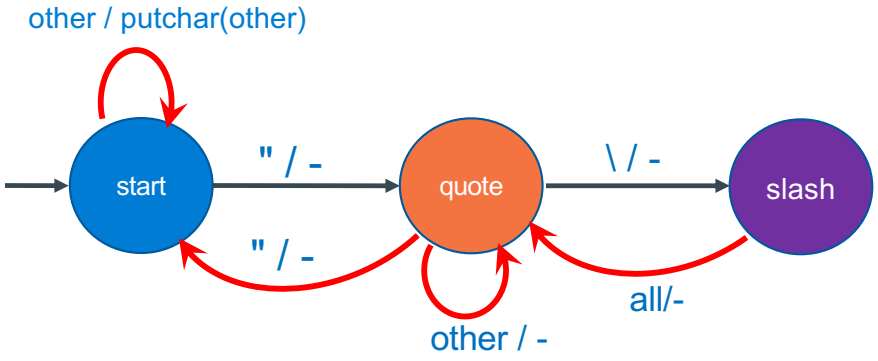
Class Overview, PA 2 and Introduction to C

Lecture 4 - Oct 4, 2022

Keith Muller

DEC PDP 11/45 - 1973

Programming a Deterministic Finite Automaton – states{.h,.c}



states.h

```
#ifndef STATES_H
#define STATES_H
/*
 * type specification for DFA states
 */
enum typestate {START, QUOTE, SLASH};
/*
 *  function prototypes state handlers
 */
enum typestate startSTATE(int);
enum typestate quoteSTATE(int);
enum typestate slashSTATE(int);
#endif
```

states.c

```
#include <stdio.h>
#include "states.h"

enum typestate startSTATE(int c)
{
    if (c == '\"')
        return QUOTE;       /* saw a double quote */
    putchar(c);             /* ok, echo input for everything else */
    return START;           /* stay in START */
}

enum typestate quoteSTATE(int c)
{
    if (c == '\\')
        return SLASH;       /* saw a backslash ignore next char */
    else if (c == '\"')
        return START;       /* saw the closing " go to START */
    else
        return QUOTE;
}
enum typestate slashSTATE(int c)  /* do we need a function here? */
{
    (void) c;      /* parameter ignore; a questionable practice! */
    return QUOTE;
}
```


x

Programming a Deterministic Finite Automaton – main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "states.h"
int main(void)
{
    int c;                  /* input char  */
    int linecnt = 1;        /* counts line in input */
    int startline = 1;      /* starting line number for quote */
    enum typestate state = START;  /* initial state of DFA */

    while ((c = getchar()) != EOF) {
        switch (state) {
        case START:
            state = startSTATE(c);
            startline = linecnt;
            break;
        case QUOTE:
            state = quoteSTATE(c);
            break;
        case SLASH:
            state = slashSTATE(c);
            break;
        default:
            fprintf(stderr, "Error: DFA state not handled\n");
            return EXIT_FAILURE;
        }  // end switch

        if (c == '\n')
            linecnt++; // count the lines

    }  // end while
```

```
    /*
     * All done. No explicit end state used here.
     * We may have had an error condition, what state did we end with?
     */
    if (state == START)
        return EXIT_SUCCESS;
    fprintf(stderr,
        "Error: line %d: unterminated quote (\")\n", startline);
    return EXIT_FAILURE;;
}
```

console – I type in blue

```
% gcc -I. -ggdb -Wall -Wextra -Werror -c main.c
% gcc -I. -ggdb -Wall -Wextra -Werror –c states.c
% gcc -I. -ggdb -Wall -Wextra -Werror main.o states.o
% ./a.out
123"456"789
123789
"123"45"67"
45
"123
456
78"9
9
"
<cntrl-d>
Error: line 6: unterminated quote (")
%
```


x

Aside: Suppress compiler errors on fall throughs

- When writing switch statements in C it is not uncommon to see a case use a fall through to the next case below it
  - Why do this: First state does extra steps and then the same steps as the "fall through " state
  - But compilers often (with extra checking flags, using heuristics) will flag this a potential error
  - Approach: use the comment `/* FALL THROUGH */` to fix this (a bit of a "hack" ☺)

```
int a = 2;
switch (a) {
case 1:
    printf("1\n");
    break;
case 2:
    printf("2\n");
default:
    printf("default\n");
    break;
}
```

```
% gcc -ggdb -Wall -Wextra -Werror switch.c
switch.c: In function ‘main’:
switch.c:11:9: error: this statement may fall through [-Werror=implicit-fallthrough=]
   11 |         printf("2\n");
      |         ^~~~~~~~~~~~~
switch.c:12:5: note: here
   12 |     default:
      |     ^~~~~~~
cc1: all warnings being treated as errors
```

```
int a = 2;
switch (a) {
case 1:
    printf("1\n");
    break;
case 2:
    printf("2\n");
    /* FALL THROUGH */
default:
    printf("default\n");
    break;
}
```

```
% gcc -ggdb -Wall -Wextra -Werror switch.c
% ./a.out
2
default
%
```


x

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***

x

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: goto labels -** Not used in CSE30

```
int global0;              /* global variable file scope */

void                      /* function foo with file scope */
foo(int parm)             /* parameter parm block scope begins */
{                         // the body (block) of the function begins
    int i, j = 5;        /* variables with block scope */
    for (int k = 0; k < 10; i++) { // inner k block scope
        // some code
    }
}                         // the body of the function ends
```


x

File Scope Example

- The scope of a function or any variable declared outside of any block or parameter list begins at the point of declaration (within the translation unit) and ends at the end of the translation unit (file)

```
int i;                  // file scope of i begins
static int g(int a)     // file scope of g() begins (note, "a" has block scope)
{
     return a;
}

int main(void)          // file scope of main() begins
{
    int x = 2;          // block scope of x begins
    i = g(x);           // i and g are in scope
    return EXIT_SUCCESS;
}
```


x

File Scope and Block Scope Example

```c
int a;                  // file scope of name a begins here
void f(void)
{
    int a = 7;          // block scope of the variable a begins here; hides file-scope a
    {
        int a = 4;              // the scope of the inner a begins here, outer a is hidden
        printf("%d\n", a);      // inner a is in scope, prints 4
    }                           // the block scope of the inner a ends here

    printf("%d\n", a);          // the outer a is in scope here, prints 7
}                       // the scope of the outer a ends here
```


x

Nested Scope Example

- Nested Scope: When two different variables have the same name are in scope at the same time, the declaration (remember definitions are also declarations) that appears in the inner scope hides the declaration that appears in the outer scope

```
void funcA(int n)      // scope of the function parameter 'n' begins
{                      // the body of the function begins
    ++n;               // 'n' is in scope and refers to the function parameter
    // int n = 2;      // error: cannot redeclare identifier in the same scope

    for(int n = 0; n<10; ++n) {      // scope of loop-local 'n' begins
         printf("%d\n", n);          // prints 0 1 2 3 4 5 6 7 8 9
    }                                // scope of the loop-local 'n' ends

    // the function parameter 'n' is back in scope
    printf("%d\n", n);               // prints the value of the parameter

}  // scope of function parameter 'n' ends
```


x

Extra Slides

- Slides in this section are not used in class but contain material that you will find useful
- Some are slides that were removed due to time constraints
- Some will be used later in the course

Re-declaration of Global variables

- You can have multiple definitions (if they are consistent), this is called a re-declaration, but only one can give it an explicit value

```
extern int x;
int x = 10;          // OK
extern int x;        // OK
int x;               // OK

static int n;
static int n = 10;   // OK
static int n;        // OK
```

C Variable Definitions

- **Local Variables & Function Parameters**: are 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
- **Global variables:** are defined **outside a function body**
  - Scope: valid from **point of definition** to end of file
  - **Static storage lifetime -** valid while program is executing

```
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
}
```


x

Watch out for Hardware differences: Why getchar()/putchar use an int and not a char

- unsigned char: 8 bits positive values only 0 to 255
- signed char: 8 bits negative & positive values (-128 to +127)
- char (with no modifier): 8-bit (signed or unsigned: implementation dependent)

```
#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

x

C Versus Java

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

	Java	C

Overall Program Structure	source file: Hello.java
	
	public class Hello
	{ public static void main
	(String[] args)
	{ System.out.println(
	"hello world!");
	}
	}
		source file: hello.c
		#include <stdio.h>
		#include <stdlib.h>
		
		int
		main(void)
		{
		printf("hello world!\n");
		return EXIT_SUCCESS;
		}

Access a library	import java.io.File;	#include <stdio.h> // may need to specify library at compile time with -llibname
Building	% javac Hello.java	% gcc –Wall –Wextra -Werror hello.c –o hello
Running (execution)	% java Hello hello world!	% ./hello hello world!

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	`int [] a = new int [10];`
	`float [][] b =`
	`  new float [5][20];`
		`int a[10];`
		`float b[5][20];`

Array bounds checking	`// run time checking`	`// no run time checks – speed optimized`
Pointer type	`// Object reference is an` `// implicit pointer`	`int *p;` `char *p;`

Record type	`class Mine`
	`{  int x;`
	`  float y;`
	`}`
		`struct Mine`
		`{  int x;`
		`  float y;`
		`};`

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

x

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)

x

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**)

x

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**

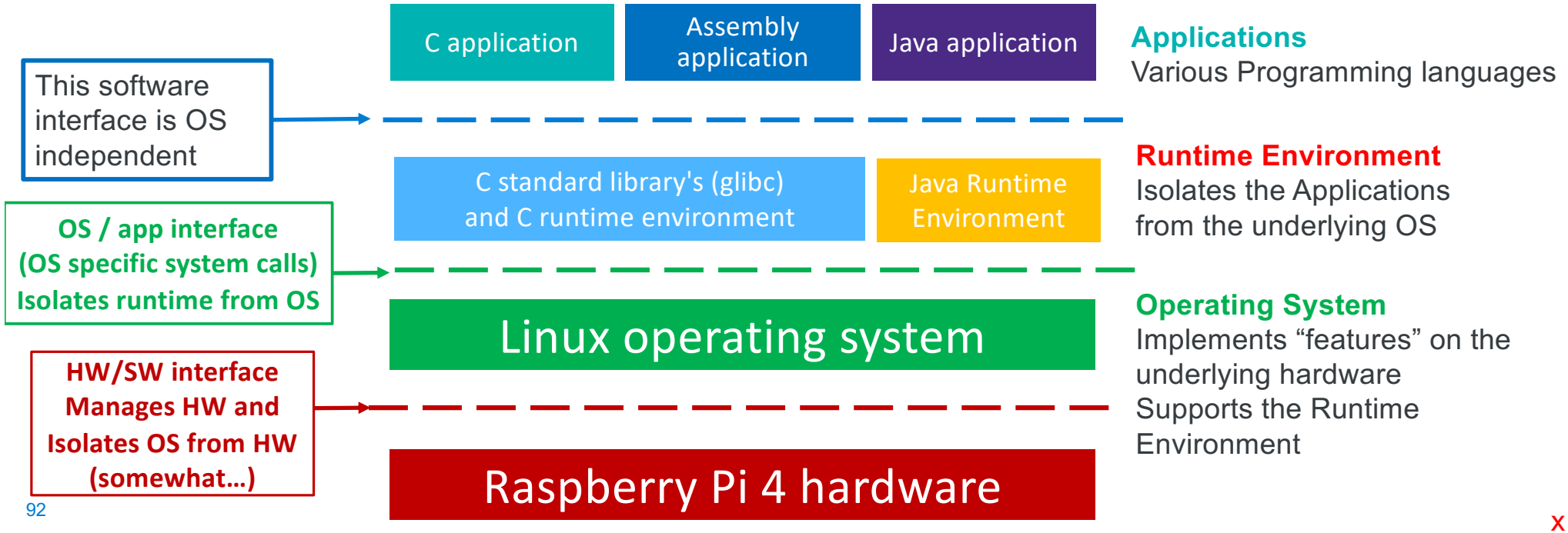
Reference: printf(), fprintf() Formatting basics

- 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

x

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



x