Version 2.15

UCSD CSE 30 Section B

Computer Organization and Systems Programming

C Part 1

Keith Muller

DEC PDP 11/45 - 1973

CSE30 Section B Spring 2024

Instructor: Keith Muller

- **I highly encourage feedback**
  - Please bring any issues to my attention, I will promptly address them
- How to contact me directly:
  - kmuller@ucsd.edu
  - Please do not use canvas messages

- In Person Office Hours: CSE 2109
  - Tue, Thu: 2:00 PM to 3:00 PM
  - These office hours are group meetings
  - Ask questions, review material, or just come to listen
  - Students who attend office hours tend to do better in the course

- Zoom Office Hours
  https://ucsd.zoom.us/j/97367536447
  - Friday: 4:00 PM to 4:45 PM
  - These office hours can be indivual of for a group if you like
  - Additional office times By Appointment
  - Send me email to schedule

x

CSE 30 Spring 2024 – Staff Covers Both Sections A & B

Section A (Cao) and B (Muller) share the same pool of TA's and Tutors

TA's

- Nitya Agarwal
- Mihir Kekkar
- Yuchen Jing
- Liam Fernandez

Tutors

Ali Alabiad
Bryan Cho
Charlotte Dong
Vivian Liu
Kate Romero
Kevin Shen
Charvi Sukla
Fong Vachirathanusorn
Joseph Edmonston
Thanh-Nhan Lam

Tutors

Christian Lee
Jessie Ouyang
Brandon Reponte
Adrian Rosing
Luffy Saito
Leica Shen
Shijie Wang
Alex Simonyan
Reese Whitlock

x

Overview of Grading - See Syllabus (Canvas) for More Details

70 pts – Attending Lecture in person

- 5 points per section B lecture

120 pts total – Canvas Quizzes

240 pts total – Programming Assignments

190 pts - Midterm – In Person

380 pts - Final – In Person

1000 pts total for graded assignments

- **Special grading circumstances** (e.g., extended absence, illness, other issues, etc.)
  - PROMPTLY Contact me directly (kmuller@ucsd.edu)

x

Lecture 1 QR Code

- Class attendance points: To encourage you to attend lecture
  - Over the years we have found that students that attend lectures in CSE30 get better grades
- Section B has 20 lectures, attend 14 to get the 70 points
  - Attending more than 14 gets you up to 30 more points
- Attendance is taken at the start of class using google forms that is accessed with a lecture QR code in the slides
  - For the first lecture only, the form will be open until 9 PM
  - bring a device that can use QR codes to access goggle forms and allows you to sign into UCSD SSO
  - You will be required to supply a code word announced in class
  - You will eventually get an email acknowledgement from google that your attendance was recorded
- **ONLY If you cannot access the google form,** send me email (kmuller@ucsd.edu) with the code word in the subject line
  - The email must be timestamped within the first 20 minutes of lecture

If you have issues with this method, please contact me

x

CSE30 Spring 2024 Section B Specific

- There are two sections: Section A (Cao) and **Section B (Muller)**
- **What is the same in the two sections**
  - **study topics** (roughly in-sync by the end of each week)
  - **quizzes**
  - **Programming Assignments**
- **What is different between the two sections**
  - **lecture materials**
  - **midterm questions (from Sect B lecture)**
  - **final questions (from Sect B lecture)**
- **In-person lecture attendance** is **strongly encouraged** (attendance points)
  - Lectures are podcast recorded
- **Discussion section attendance** is **optional but strongly encouraged**
  - You may attend either discussion section and still be enrolled in Sect B
  - Section B sections are podcast recorded
- **See the syllabus for grading details**

x

CSE30 Class Resources

- **Section B Lecture Slides:** https://github.com/cse30-sp24/Muller-Slides
  - Located on class github in both pptx and pdf format
  - Slides **are updated constantly** to correct errors and to improve content
    - Version is at the upper left on the title slide
    - Always check you have the current version the morning before lecture
- **Class github:** https://github.com/cse30-sp24
- **Piazza:** https://piazza.com/ucsd/spring2024/cse30_sp24_a0/home
  - **First Place to go to** for **Q/A** and **important announcements**
  - **Public piazza posts are for:** general questions on PA's and lectures
    - **Do not post publicly** any parts of an assignment, quiz or exam solution
  - **Private posts are for:** specific situation relating to just you or you are not sure
- **Tutor Lab hour schedule:** https://autograder.ucsd.edu
  - For getting help from the tutors
- **Canvas**: https://canvas.ucsd.edu/courses/54650
  - Links to quizzes, textbooks, programming assignments, exams
- **Gradescope:** https://www.gradescope.com
  - Quizzes and Submitting programming assignments

X

Surviving Section B Lectures (In-person)

- **Make sure you bring your copy of lecture slides to class, it helps**
- **How to get my attention in class**
  - **I never intentionally ignore questions**; **I just may not see you**
  - Raise your hand, or just **call out** if I appear to **ignore you by accident**
- **You must SLOW ME DOWN**: Otherwise, **I tend to speed up**
  - Please do not be shy, **speak up** and **remind me to slow down**
- If you have questions, or I went too fast, or the material is not clear, etc.
  - Please ask me to go over it again (do this right away, not 5 slides later)
    - Just don’t sit there and waste your time
  - **my responsibility:** help you learn the material
  - **your responsibility**: ask questions (I love questions, they also slow me down!)

x

How to do well in CSE30 - 1

- Go to lecture
  - Before lecture go over the class slides
  - Lecture slides are posted the day before class (last minute updates that morning)
  - Keep your lecture slides up to date (I update them to fix errors and address questions)
- Go to Discussion Sessions
  - ask the TA's and Tutors for help
- Studying for exams
  - All the exam question topics are found in my slides and the PA writeups
  - Try to write the exam yourself, with practice you will be able to guess the questions
- Post to piazza when you have questions
- Do the readings on time
- Review the material: watch the podcasts and occasional special topic videos

X

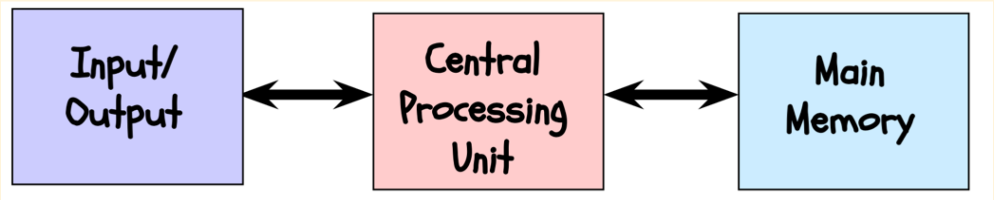
How to do well in CSE30 - 2

- **Most important:** Keep up, **do not procrastinate as it is hard to catch up**
  - The class material starts easy and gets much harder over the quarter
  - Do not expect you can do later programming assignments in less than 5 days
  - Do not expect to learn the material by binge watching podcasts, this never ends well
- **Please be careful when using web resources** for this class
  - a lot of the material you will find is either not correct or does not apply to our programming environment
  - this is especially true with assembly language programming topics
- **Are you struggling?**
  - **<u>Do not wait</u>**, **ask for help as soon as possible** – do not fall behind
  - **<u>Best advice: Come to my office hours</u>** (or schedule a zoom meeting)
    - Give me a chance to help you
    - I will spend as much time as necessary to help you understand the material

x

A General-Purpose Computer – Von Neuman Architecture

- Since the middle of the 20th century, many architectural approaches to the **general-purpose computer** have been tried
- The **architecture** which **nearly all modern computers** are based was proposed by John Von Neuman in the late 1940's
- The **major components** are:



- Central Processing Unit (CPU): a device which fetches, interprets (decodes), and executes a specified set of operations called instructions
- Memory: Storage of N words of W bits, where W is a fixed architectural parameter, and N can can be expanded to meet **workload** (the programs running on the CPU) and **cost requirements**
- I/O: Devices for communication with the outside world (including external persistent storage)
  - External connections (from CPU to memory and I/O) typically use industry **"standards"**
  - **Standards** enable technologies from **different companies to interoperate**

x

Memory is Organized in Units of Bytes

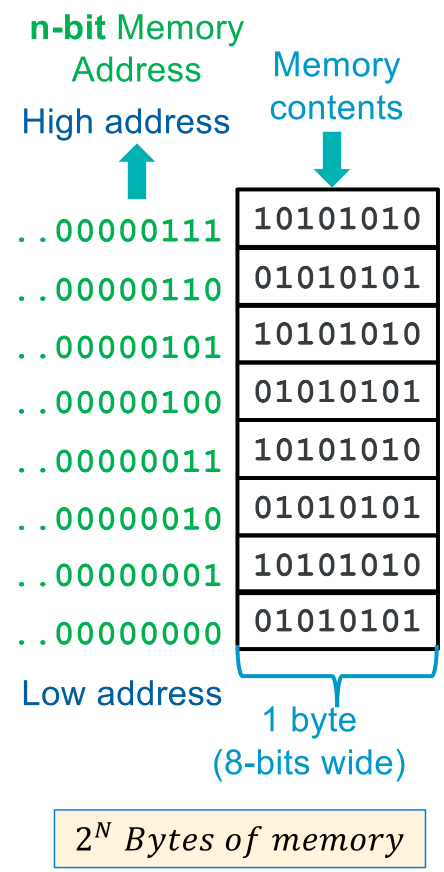
- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1
- Memory is organized into a **fixed unit** of 8 bits, called a **byte**

MSB → Most Significant Bit

Bit position in Byte							
7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
Content of each bit							

← LSB (Least Significant Bit)

- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*
- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*
- Each byte in memory can be **individually accessed** and operated on given its **unique address**
- **Word size: is the number of bits in an address**



x

What is Computer Architecture?

Instruction Set Architecture (ISA)

- Functional behavior of a computer system **as viewed by a programmer**
  - describes how the CPU is controlled by software programs
  - specifies both what the processor can do as well as how it gets it done
- Architectural Characteristics (partial list):
  - supported data types (how data is encoded)
  - CPU registers (number, size, use, etc.)
  - how the hardware manages main memory
  - instructions a microprocessor can execute
    - Operations they perform
    - Instruction "format" (bit patterns) in memory
  - input/output model

Machine Organization

- Physical (design) realization of what is specified by the instruction set architecture
- It deals with how the hardware components are linked together to meet the requirements specified by instruction set architecture
  - The ISA allows variability in the physical design implementation to address different workload needs (cost, scalability, etc.)
- Machine organizational characteristics (partial)
  - Hardware component choices to achieve:
    - Expandability
    - Configurability
    - etc.
  - Physical layout
  - Number and type of peripherals (I/O devices)

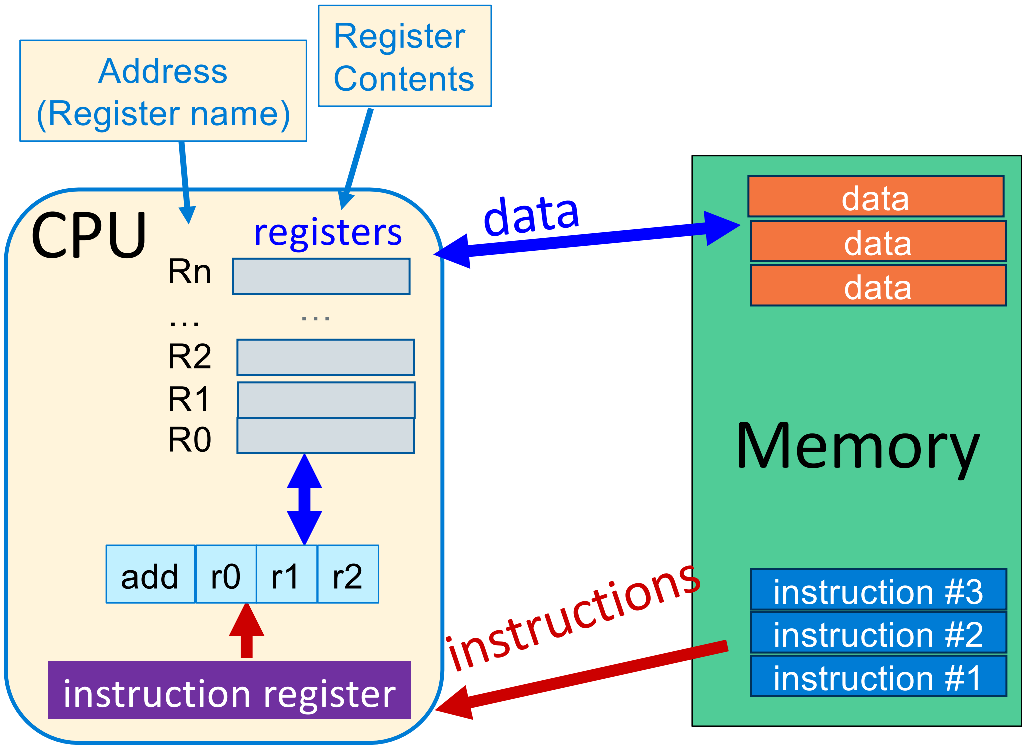
X

Von Neuman Architecture

- **Distinguishing feature**: Memory contains **both** program CPU instructions and data
- **CPU Instructions** are encoded in memory with patterns of ones and zeros (similar to binary numbers)
  - **Encoded CPU instructions** are called **machine code (or machine language)**
- **Example**: three 32-bit instructions (shown in hexadecimal format below)

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

- **Instructions** operate on **data** that is stored in a small capacity volatile (fast) memory in the CPU
  - This memory is called registers
  - ARM-32 has 32-bit registers
- CPU **reads/writes data** from **memory** from these **data registers** to **operate on the data**

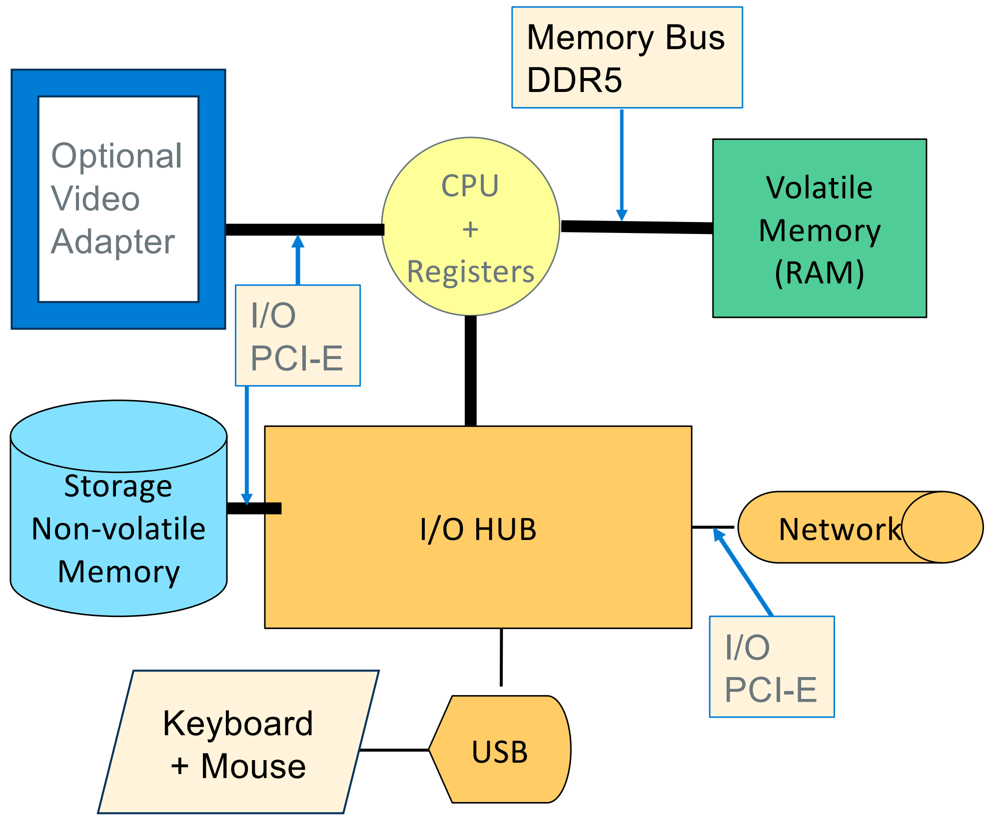


x

Machine Organization – Von Neuman

1. CPU executes a machine code program
   - Machine code is specific to a particular CPU Instruction set Architecture (ISA)
2. **Memory contains both data and programs**
3. I/O (input/Output): Connects the CPU and memory to the external world
   - An I/O operation is where data (including machine code) is copied between persistent storage (like an SSD) and ram memory

- Volatile (non-persistent) memory
  - contents lost when power is removed
  - Memory dimms (memory bus)
  - CPU registers (memory inside the CPU)
- Non-volatile (persistent) memory
  - contents preserved when power is removed
  - SSD (I/O bus attached)
  - NVDIMM (memory bus attached)

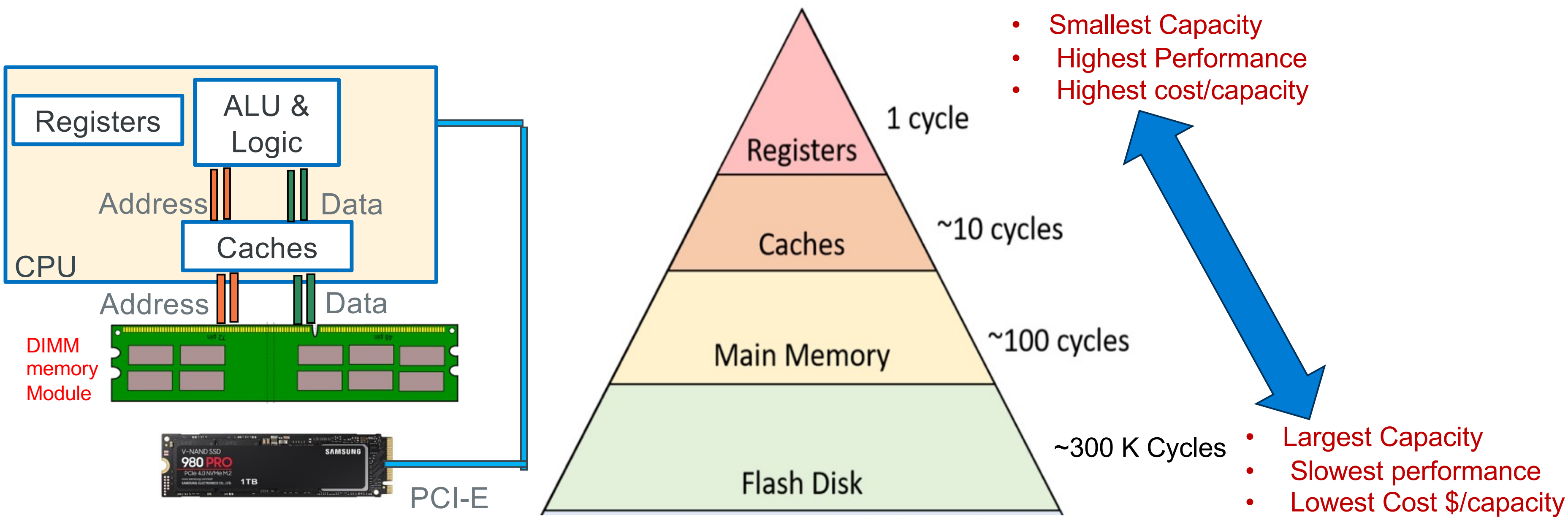


x

Memory Triangle: Hardware Cost/Performance/Capacity Tiers

Assume each instruction takes 1 clock cycle

Clock cycle =~ time to access; larger is slower

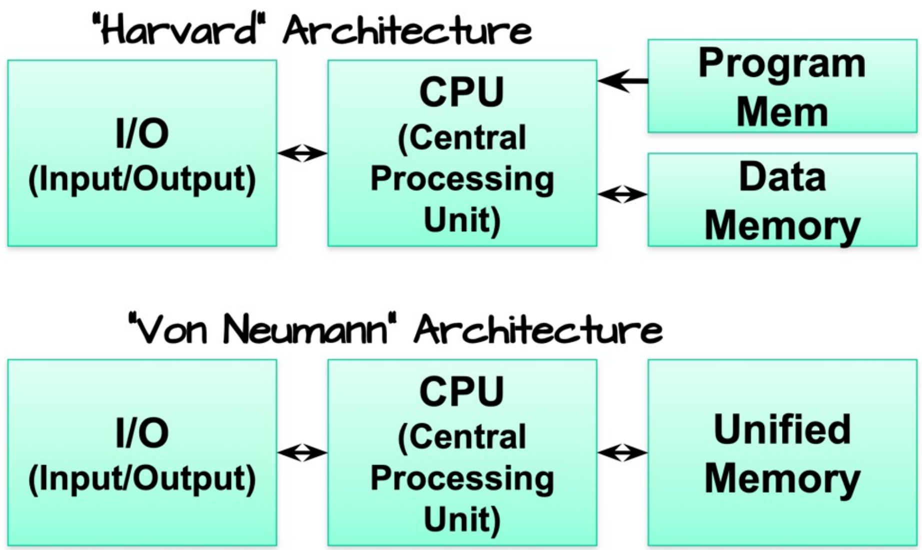


Design Goal: best performance at the lowest (or specific) cost
Other goals: performance/energy (operating cost), expandability, high margin (price/cost)

x

An Alternative that was not successful: Harvard Architecture

- **Harvard architecture premise:** Instructions and data should not interact (claim is higher higher performance), and they can have different word sizes
  - **Observation:** Two memory subsystems (using similar state of the art technologies) can be accessed concurrently for higher throughput
- **Distinguishing feature: Independent** instruction and data memories
- Do you agree and why?

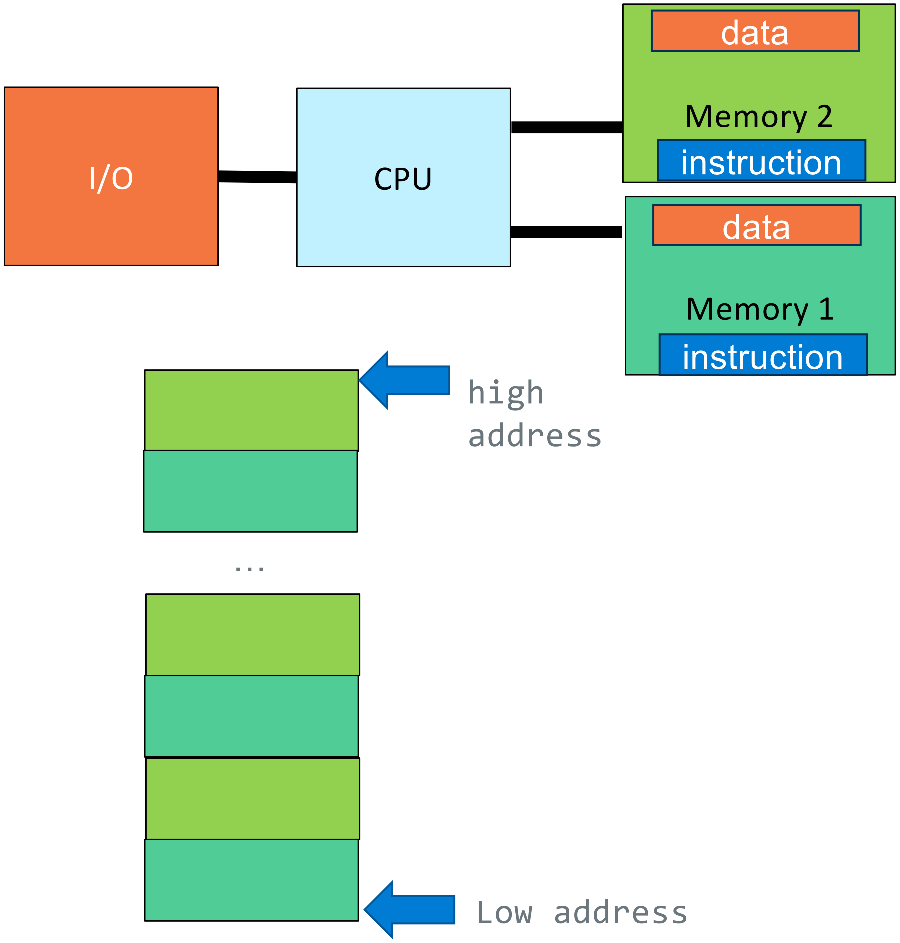


x

Machine Organization Example – Which Architecture is it?

- A good exam question
- Answer: Either you must be told where the Instructions and data are placed
- How can this be a Harvard architecture?

- Harvard Architecture: Use physical **memory interleaving** to achieve the performance increase with having to scale and size two different memory subsystems
- The size of the interleave is some multiple of bytes (like 1024)



x

C, Assembly and Executable Programs

- **Assembly language** is a symbolic version of the **machine code (language)**
  - **Instructions** describe operations the hardware can perform (e.g., =, +, -, *)
  - **Unique to a specific ISA**: e.g., ARM-32 versus IA-64
  - May be stored in a human readable text file
  - You can write in assembly language just like C or Java
    - Assembly is much easier to program than machine code
- A **high-level language** (like C) is **compiled** into an **assembly language equivalent**
  - A statement in C is represented by a sequence of one or more assembly language instructions (why a do you think it is a sequence?)
- **Assembly language program**
  - assembly language program is **translated (assembled)** into **machine code**
- An **executable program** contains
  - **series of instructions in machine code** (the program)
  - (maybe some) **data** to operate on

x

Assembly & Machine Code Example: ARM-32 (32-bits)

Consider an addition statement
R0 = R1 + R3;

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	0	Rn	Rd	00000000	Rm

Simple R-type instructions follow the following template:

```
OP    Rd, Rn, Rm
```

Observe there are only enough bits to specify 16 registers

```
0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN
```

List of Different operations for this type of instruction

Assembly Language (human readable)

```
ADD    R0, R1, R3
```

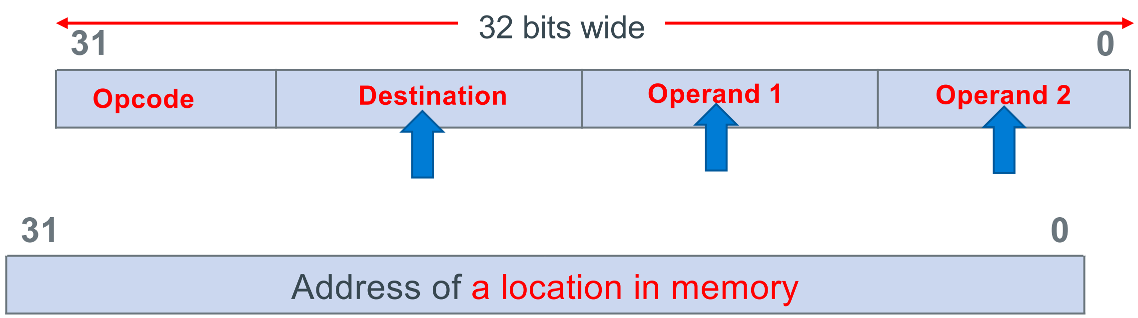
machine code pattern in memory

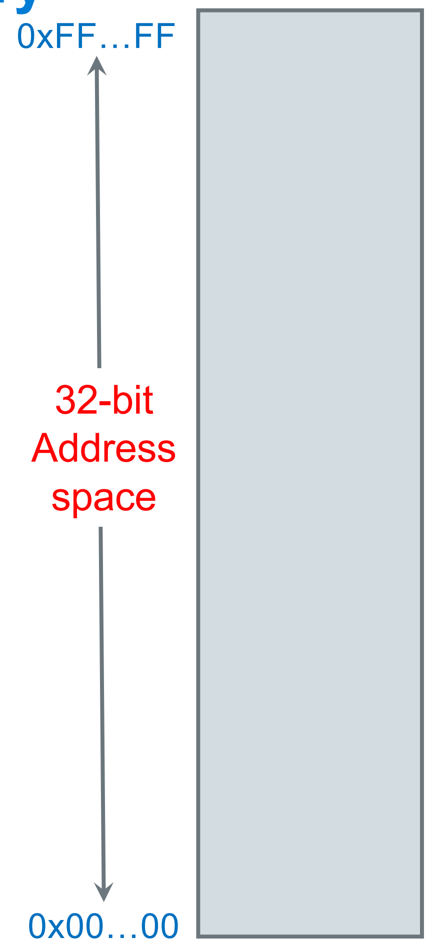
1110 0000 1000 0001 0000 0000 0000 0011

x

Why only 16 Registers & how to access all of memory

- Consider a = b + c are operands are in memory
  - Operation code: add — Destination: a
  - Operand 1: b — Operand 2: c
- Aarch32 Instructions were designed to always be: 32 bits wide
  - Some bits must be used to specify the operation code
  - Some bits must be used to specify the destination
  - Some bits must be used to specify the operands
- To address all of memory you must store an address in a register
  - ARM-32 registers (the contents) are 32-bits (can contain data or an address)

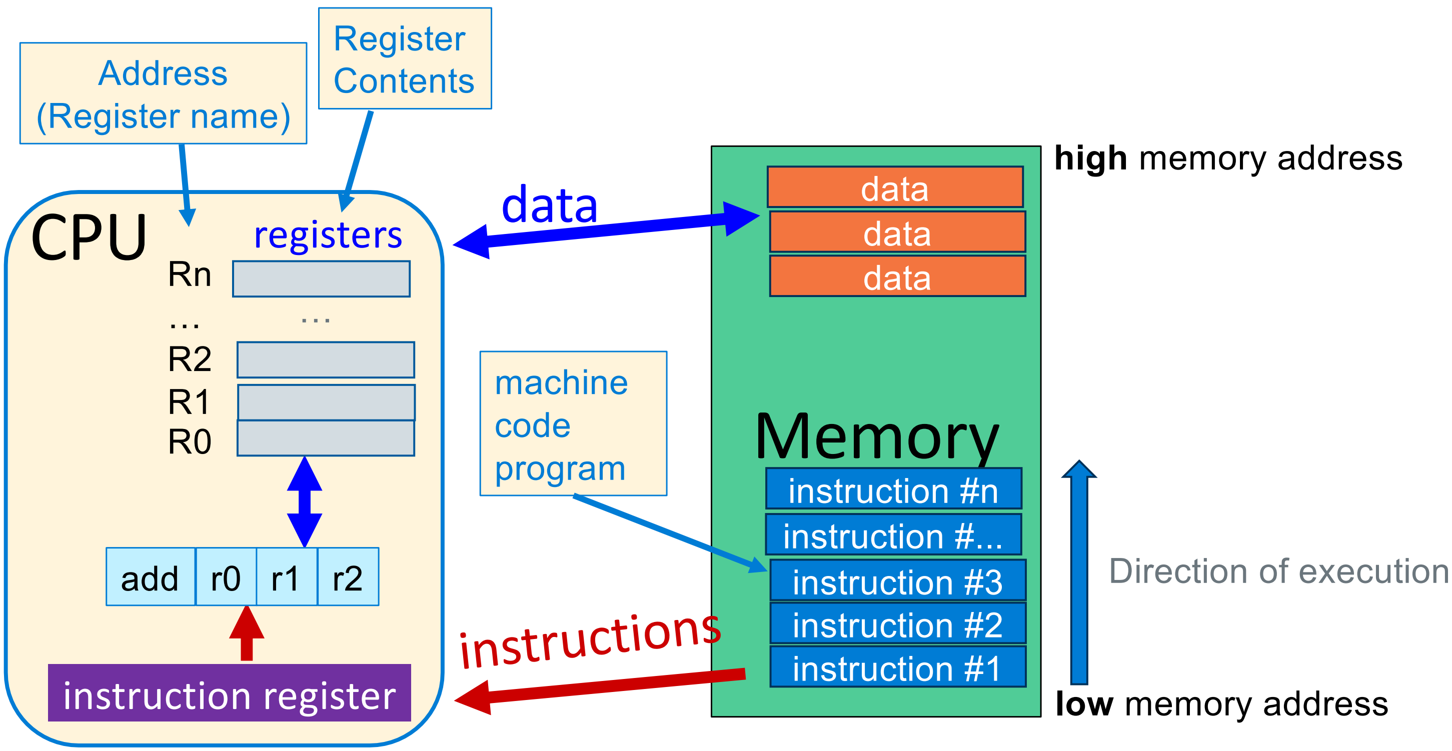




NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

x

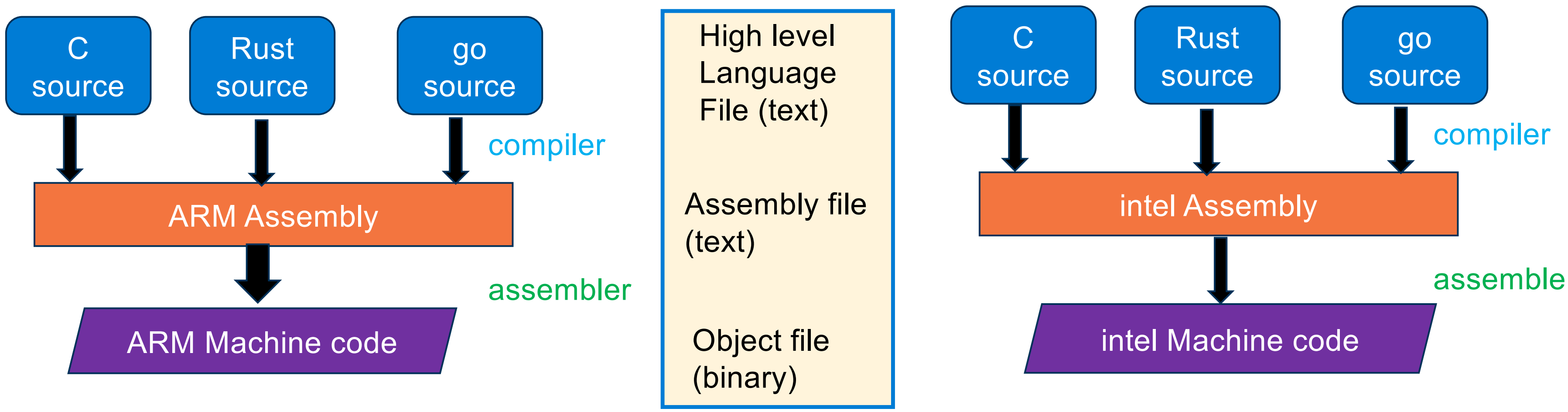
Machine code execution order



- **Execution order**: Programs execute from instructions located in **low address** memory to **high address** memory stepping **one machine instruction at a time (**called **execution order) unless there is a branch** (example: loop, if statement etc.)

x

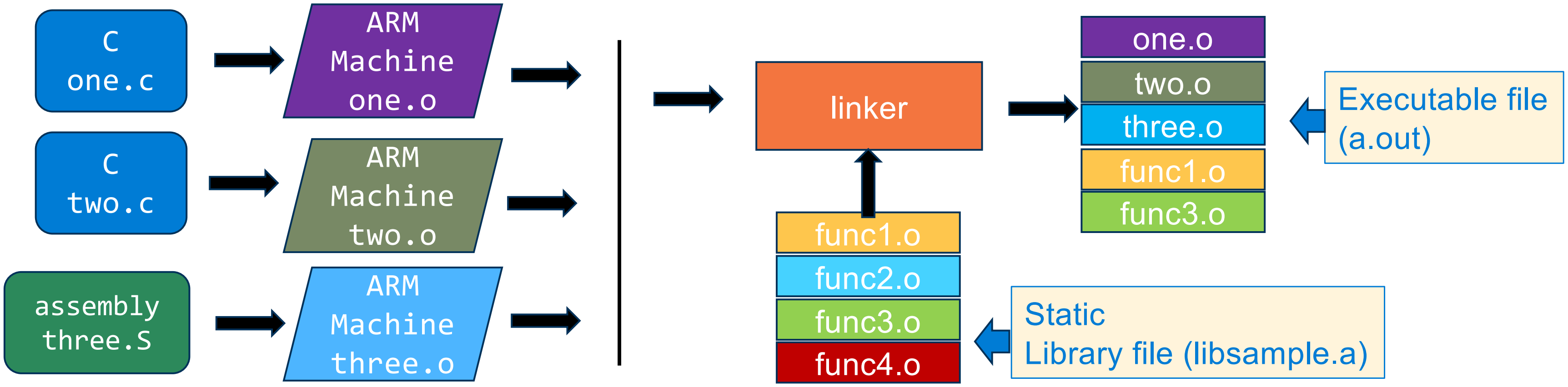
From Source to Machine code



- The **granularity** of **compilation and assembly** is a **single text file** (called a **translation unit**)
  - **.c** file is a C **source** file (file.c)
  - **.S** file (upper case S) is a **human** written **assembly source** file (file.S)
  - **.s** file (lower case s) is a **compiler** generated **assemble source** file (file.s)
  - **.o** file is a **machine code binary (object)** file (file.o)
- Multiple **.o** files are **combined** (linked) into an **executable file**

x

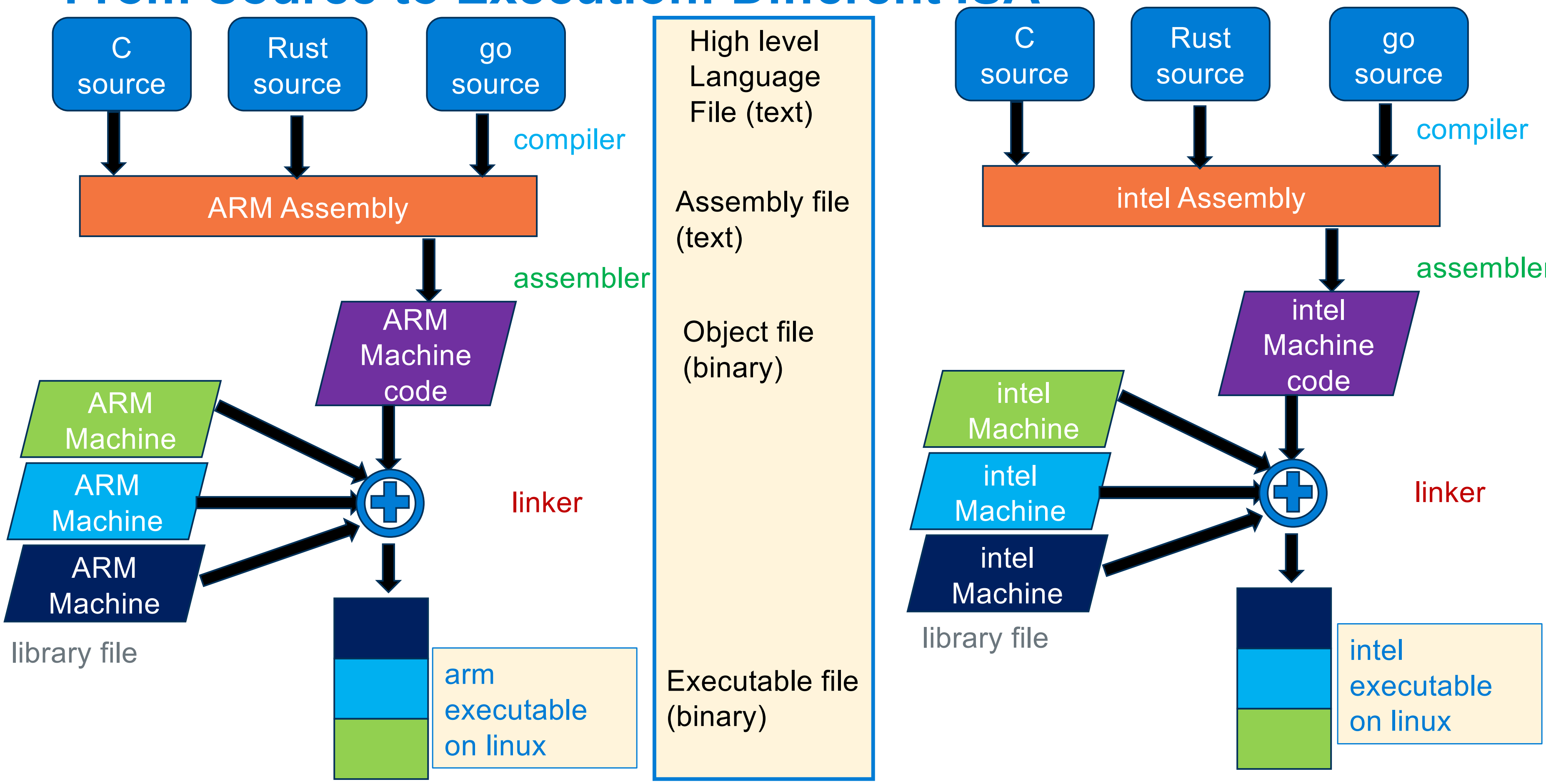
Linker: Combines object files to create an executable file



- Each source file (**Translation unit)** is compiled (or assembled) independently to an object file
  - 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 the other already compiled object files
- **Library file** (libXX.a – where XX is the library name) is an **aggregation** of **distinct object** (.o) files
- Linker combines all the listed object files together plus **just those object files in libraries** whose **contents are referenced**
  - **Example**: one.c and two.c call functions contained in func1.o and func3.o (no calls to func2.o or func4.o)
- Important: **Object files created from C and assembly source can be linked** (call each other) into a working executable when certain rules are followed (we will be doing a lot of this later this quarter)

x

From Source to Execution: Different ISA



Pi-cluster system (all CSE30 PA's)

ieng6

x

From Source code to Execution

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
int main (void)
{
    printf("Hello!\n");
    return EXIT_SUCCESS;
}
$ gcc -Wall -Wextra -Werror -c -S test.c
$ ls -ls
total 8
4 -rw-r--r-- 1 kmuller kmuller 109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller 725 Mar 14 15:58 test.s
$ gcc test.s
$ ls -ls
total 16
8 -rwxr-xr-x 1 kmuller kmuller 7708 Mar 14 15:58 a.out
4 -rw-r--r-- 1 kmuller kmuller  109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller  725 Mar 14 15:58 test.s
$ ./a.out
Hello!
```

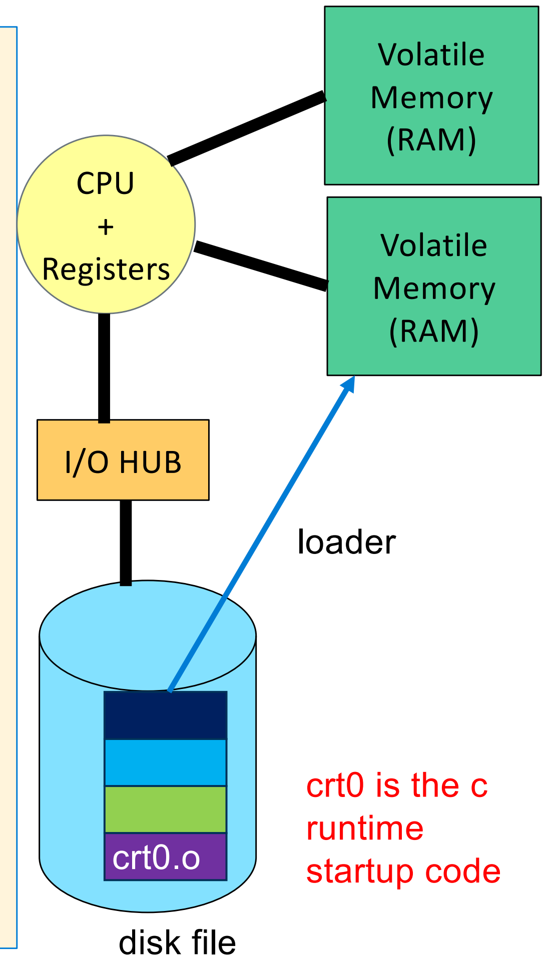
Source to Execution Steps

1. **Compile** (c source to assembler)
2. **Assemble** (assembler source to object)
3. **Link** (Combine object files to executable)
4. **Load** (Copy executable from into memory)
5. **Execute** (OS runs the code)

compile

assemble and link
gcc automatically calls the assembler with .S or .s files

load and then execute



crt0 is the c runtime startup code

x

Equivalent Code: C -> Assembly -> Machine

C source

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    printf("Hello!\n");
    return EXIT_SUCCESS;
}
```

ARM-32 assembly

```
        .section .rodata
mesg:   .string "Hello!\n"
        .text
        .global main
        .type   main, %function
        .equ    FP_OFF, 4
        .equ    EXIT_SUCCESS,   0
main:   push    {fp, lr}
        add     fp, sp, FP_OFF
        ldr     r0, L1
        bl      printf
        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
L1:     .word mesg
```

address of mesg

Code aka TEXT

memory address | high contents low bytes | corresponding assembly

```
00010408 <main>:
   10408: e92d4800        push {fp, lr}
   1040c: e28db004        add fp, sp, 4
   10410: e59f0010        ldr r0, [pc, 16] //10428 <L1>
   10414: ebffffb3        bl 102e8 <printf@plt>
   10418: e3a00000        mov r0, 0
   1041c: e24bd004        sub sp, fp, 4
   10420: e8bd4800        pop {fp, lr}
   10424: e12fff1e        bx lr

00010428 <L1>:
   10428: 0001049c

0001049c <mesg>:
   1049c: 6c6c6548       // 'l, 'l', 'e', 'h'
   104a0: 000a216f      // '\0', '\n', '!', 'o'
```

Machine code (instructions)

address of mesg

Data

x

Introduction: C Program Structure (Single file)

```c
#include <stdlib.h>
#include <stdio.h>
/*
 * This is a block comment
 */
// this is a line comment


int main(int argc, char *argv[])   // or int main() or int main(void)
{
    char x = '\n';

    printf("Hello World!%c", x);   // "Hello World!\n"
    return EXIT_SUCCESS;           // main always returns either
                                   // EXIT_SUCCESS or EXIT_FAILURE
}
```

directives to the preprocessor

main() is the first function to run
Every executable program must have one function called main()

char literal '\n'

library function for writing to stdout

string literal "Hello World!%c"

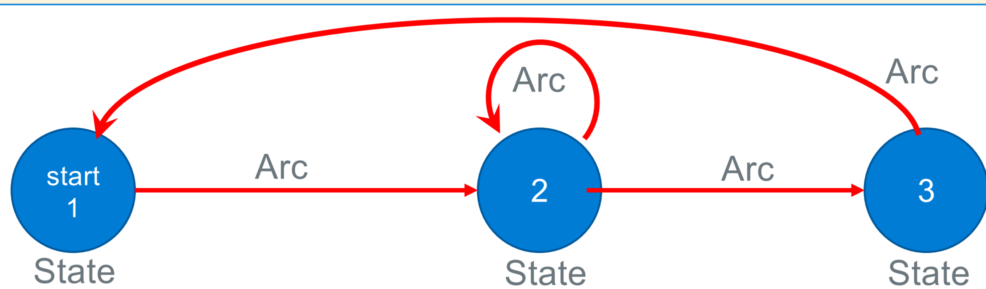
x

PA2/PA3 Design: Using a Finite State Machine

- Finite state machine (or Finite State Automaton) 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**

Circle (States) and **Arc** representation

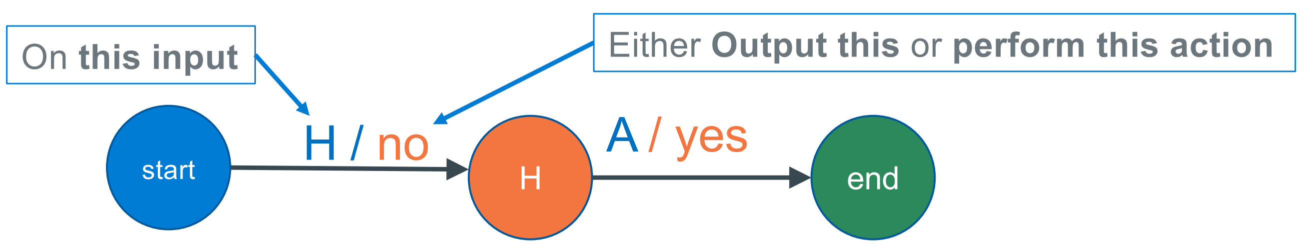
- A **circle** (state) **represents *(remembers – a "path")*** **what has already been seen** in the **input stream**
- An **arc** represents a **transition** from one state to the next state for a **specified input** and may **specify** an **optional output** (or **operation to be performed**)
  - The **next state** can be the **same state** or a **different state**
- At any point in time, **one of the states** is the **current state** of the machine
  - **Current state** "remembers" the **input sequence seen so far** by the machine
- **Whenever a state is entered**, it ***"reads"*** to get the next input (except the ***end*** state – next slide)



x

Machine States and Transitions

- Two Special states
  - start **start** state (machine starts "powers up" in this state) **required**
  - end **end** state (done or final state) **not required** – if not present DFA runs forever
- Each **arc** has a label(s) that uses the **notation**: **input1, …, input n / output or action taken**
  - When the **input to the machine matches one of the input labels**, it **selects** that **arc to be taken**
  - The **arc taken** also specifies the output produced or action taken
    - it is **ok to have no output,** or **no operation** associated with an arc
- **Example: FSA machine below** recognizes the sequence **HA** on an input stream, then stops
  - **Question:** what is missing here? – What do we do for inputs NOT specified?



x

Arc labeling

- **output**(c) indicates c is to be output (printed for example)
- An action of – means no action (or output)

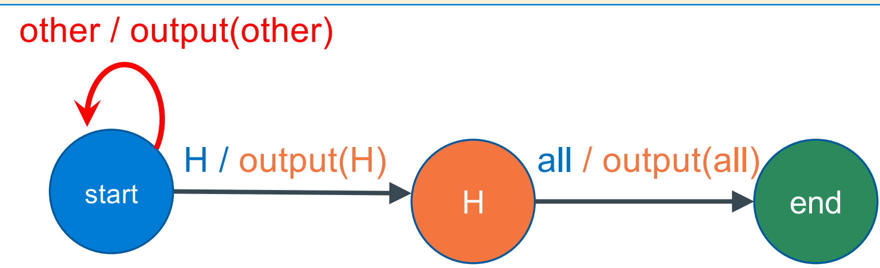
```
a / -
```

- The labels **all** and **other** have special meanings
- When an **arch is labeled** with an input of **other**, this represents all other character inputs that are not specified by other arcs
  - If you need ***to output the actual input character***, you will label the arch as:

```
other / output(other)
```

- When an **arch is labeled** with an input of **all**, then this arc is taken for all inputs
  - this **must be the only arc out of the state**
  - **Question**: Is the all label really needed?
  - If you need to output the actual input character, you will label the arch as:

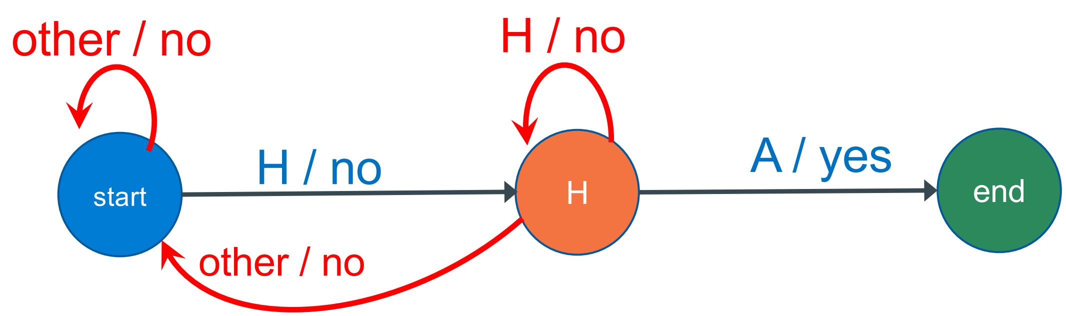
```
all / output(all)
```



x

Designing a Deterministic Finite State Automaton

- **Deterministic Finite State Automaton (or deterministic finite state machine)**
  - For any **given state**, then for **all possible inputs**, there is always **one next state**
- Step 1: Define the states (using the recognizer example from the previous slide)
  - ***Start (initial or power up) state:*** input has not seen an H (or no input so far)
  - ***H state***: input has seen at least one H (or more than one H
  - ***end state***: input has seen an H immediately followed by an A
- Step 2: Define the arcs
  - Specify arcs at each state for all possible inputs **(an arc can be taken on more than on input)**
  - Specify output or action (if any) on each arc
  - **Check**: each state transition (arc) is *unambiguous* (unique – a specific input selects just one arc)



x

DFA counting the instances of a pattern

- 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 and count all HA's
    - from the top (top of file)
    - to the bottom (end of file)

TOF

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

EOF

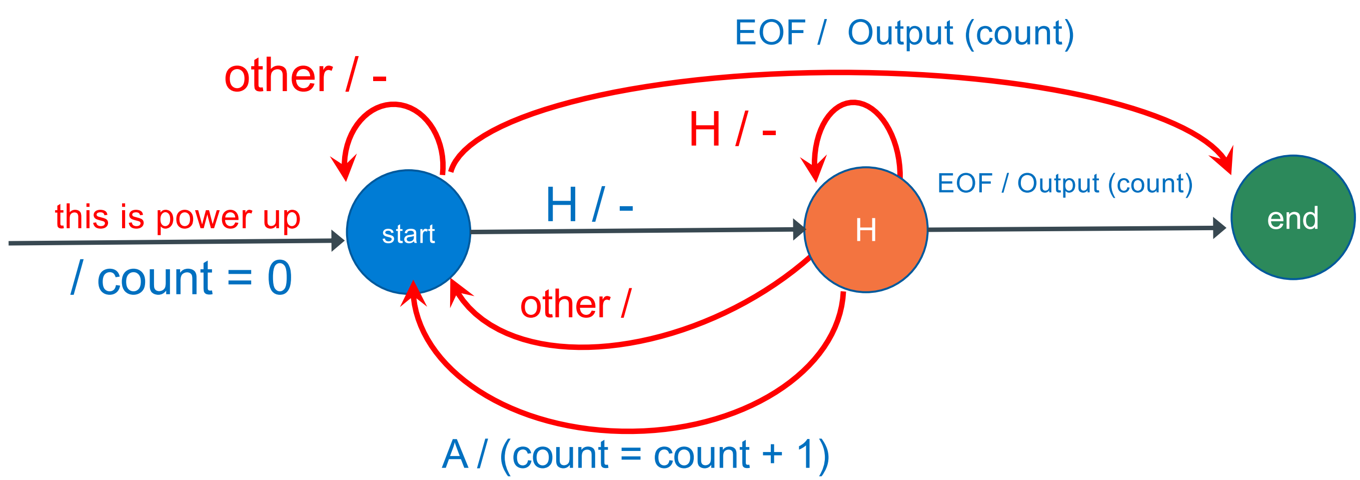
- **Action:** *Alter* the machine to process input from the file in sequential order until the end of the file (EOF) is reached

x

DFA counting the instances of a pattern - 2

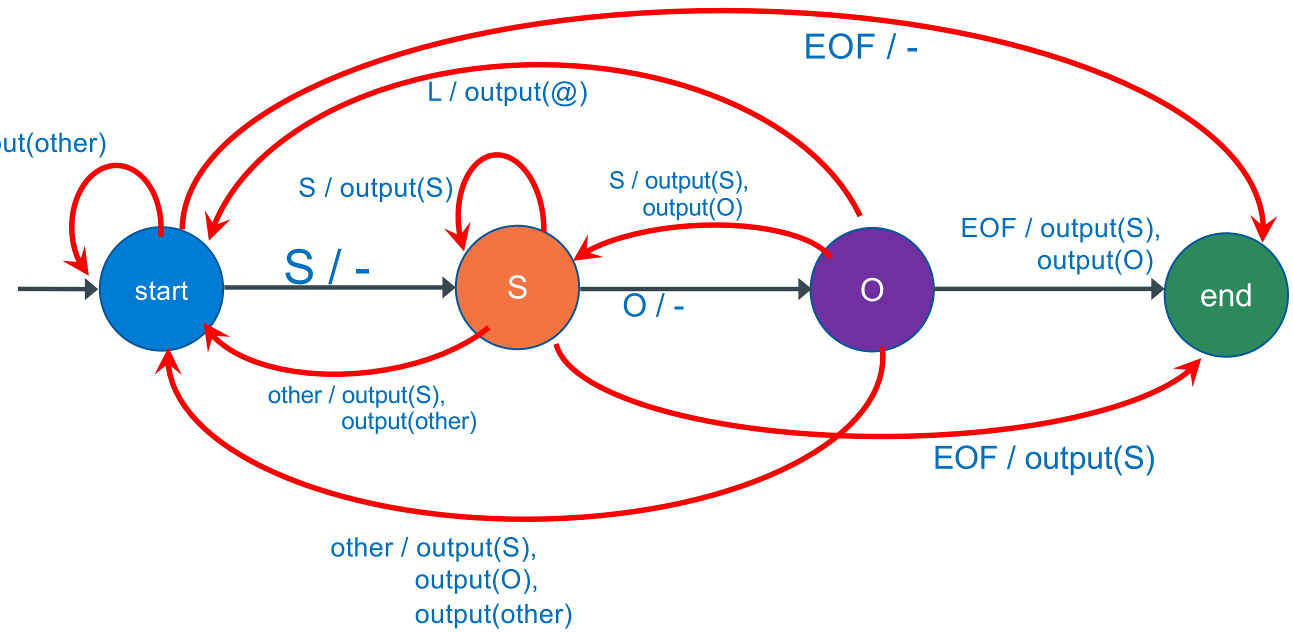
- We will adjust the DFA to act on continuous input (multiple instances of the pattern)
  - And to count the number of HA patterns

1. *"redirect"* the arc(s) that **pointed** at the **end state** to point to the **start state**
2. Convert output to counting actions
3. Add arcs from each state when EOF on input is detected to the eof state



x

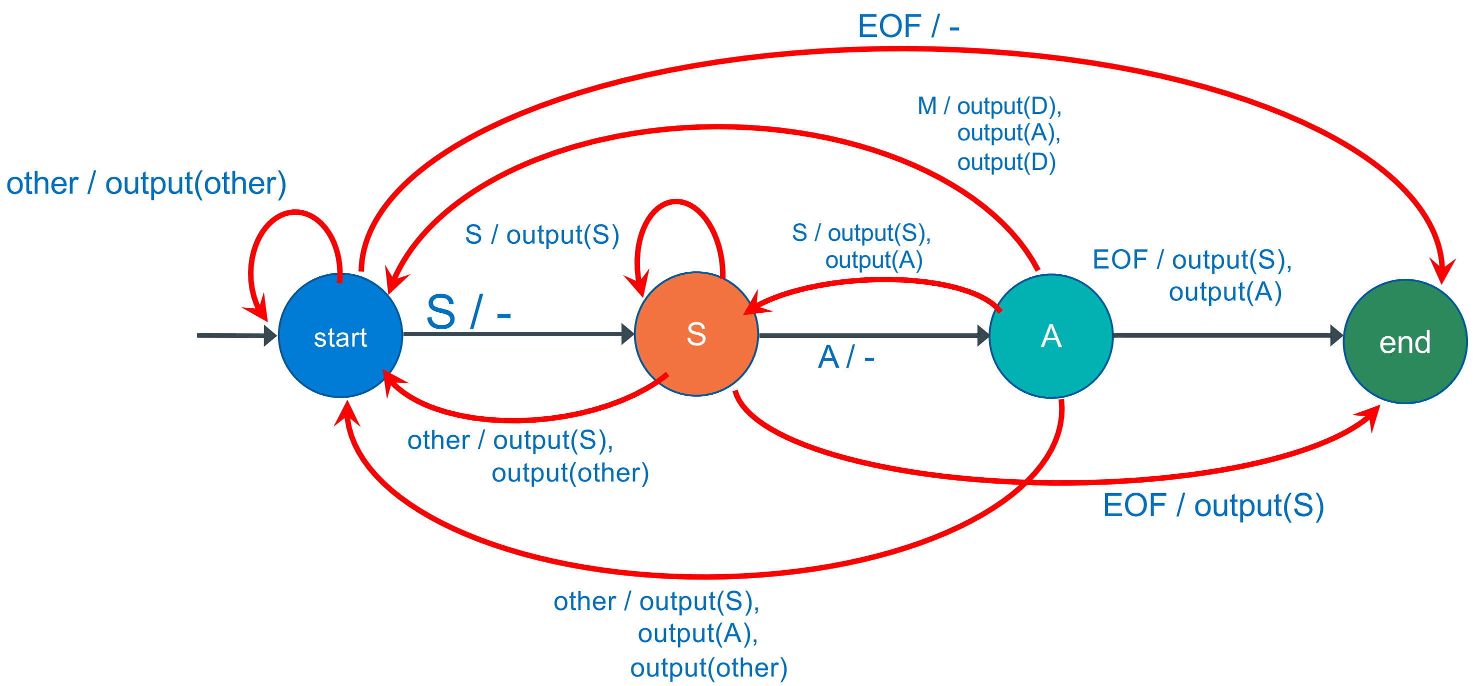
Merging DFA's: Step one design each sequence -1



This DFA replaces SOL with a @

x

Merging DFA's: Step one design each sequence - 2



This DFA replaces SAM with DAD

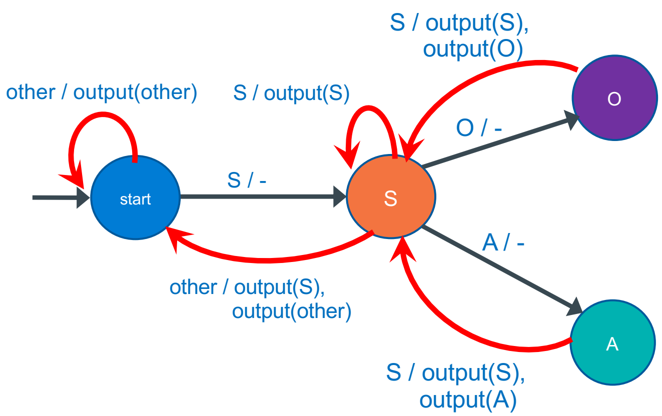
x

Merging DFA's: Step one design each sequence - 3

- To merge two DFA's
  - **combine common states**
  - make sure all the arcs out of the combined states represent the arcs in the two DFA's

Example:

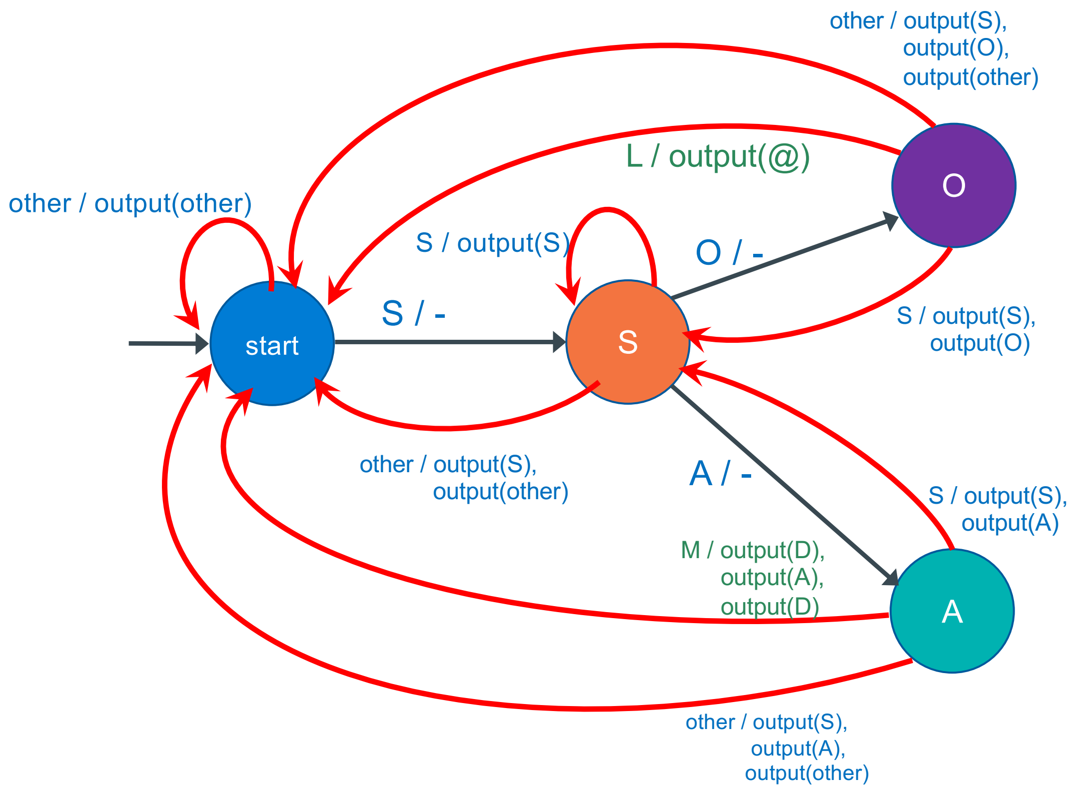
- The prior two DFAs both have the same initial state sequence (**start** and **S**) which will be in the merged DFA
- Next add all the unique arcs out of state S from both the separate DFAs and adjust the labels if necessary
- **Add the remaining states and arcs** that are unique to each DFA
- We are going to **simplify the combination** process by assuming the **input is infinite in length**
  - So, both EOF processing and the end state is not shown
  - **You will use this same assumption in PA2**



x

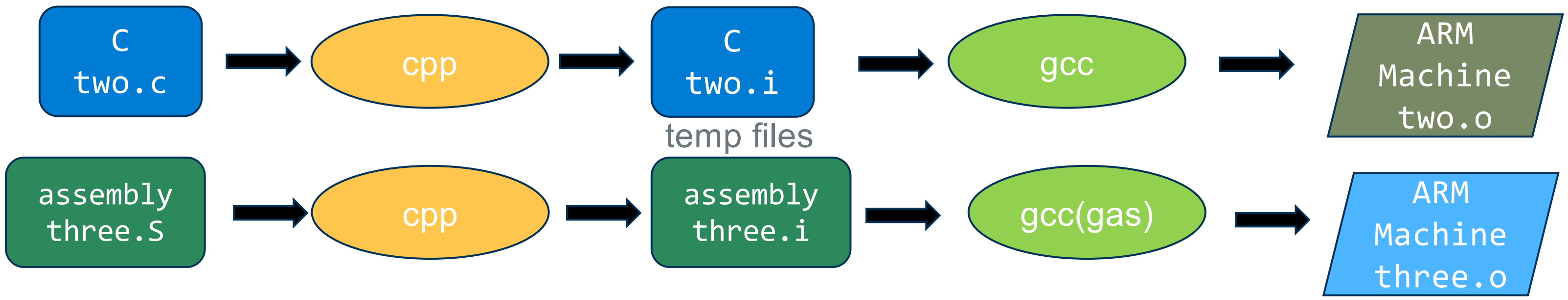
Merging DFA's – 3 (Finished)

This DFA replaces SOL with a @
and This DFA replaces SAM with DAD



x

What is the preprocessor (cpp)?



- **Preprocessing is the first phase** in the compilation (.c files) or assembly (.S files only) process
- The **preprocessor** (**`cpp`**) *transforms* your source code, then **passes it to the compiler** (on .c files) **or the assembler** (on .S files only, not .s files)
  - **cpp is automatically invoked by `gcc`**
- Usually, the input to **cpp** is a C source file (.c) or an assembly source file (.S only) and output from **cpp** is still a C file or assembly file
  - output from cpp is in a temporary .i file (deleted after use)
  - cpp **does not** modify the input source file
- **Common use**: When a **program is divided across multiple source files** (including library files), cpp helps you keep consistency among the files (**one version of the truth**)
  - Examples: Consistent values for a constants, correct function definitions, etc.

X

Common Preprocessor (cpp) Operations

- **Comments** are ***replaced with a single space*** `/*   */ , //`
  - You will do a design for this in PA2 and program it in PA3
- **Continued lines:** where the **last character in a line is a ** causes the line to be **joined with the next line**
- A ***preprocessor directive:*** commands to cpp to perform an operation (these start with a **#)**
  - `#include <stdio.h>` contents of the include file is to be *inserted* at that spot in the source file
  - `#define MAX 8`
    - **Does two things: Defines MAX** to be a ***macro*** *name* and assigns it the value 8
      - `#define MINE` just defines MINE to be a macro name with no value
    - **Convention**: **MACRO** names are in **CAPITAL** letters
  - Macros with values – `cpp replaces MAX with 8` everywhere in the source file

```
#define MAX 8
int main(void)
{
    int x[MAX]; // histogram array

    for (int i = 0; i < MAX; i++) {
        …
    }
     …
}
```

cpp input

```
int main(void)
{
    int x[8];

    for (int i = 0; i < 8; i++) {
        …
    }
     …
}
```

file ex.i

cpp out (only showing macro substitution)

x

First Look at Header Files (also called .h or "include" files)

- Header file: a file whose only purpose is to be **`#include`'d** by the **preprocessor**
  - Contains: **Exported (public) Interface declarations**
    - Examples: function prototypes, user defined types, global variable, macros, etc.
  - To import the public interface of another C source file
    - `#include` its header (interface) file
- **NEVER EVER use cpp to** `#include` a .c file, a .S or a .s file
- **Convention (strongly enforced):** header files use a .h filename extension (example: filename**`.h`**)
  - **Example**: Source file src.**c** exported (public) interface is located in the header file src.**h**
- How to specify the file to be **`#include`'d**
  - <system-defined> are system header files (typically located under /usr/include/…)

    ```
    #include <stdio.h>   // located in /usr/include/stdio.h
    ```

  - "programmer-defined" header files usually in a relative Linux path (see `–I` flag to gcc)

    ```
    #include "else.h"    // looks in the current directory first
    ```

- **Convention:** `#include` directives are usually placed at the top of a source file

x

Compilation Process Operations

```
#include <stdlib.h>

#include <stdio.h>


// A simple C Program

int

main(void)

{

    printf("Hello World!\n");

    return EXIT_SUCCESS;

}
```

preprocessor: inserts and processes the contents of files here.
Inserts: Function protype for printf (later in course)
macro value for EXIT_SUCCESS
File locations: /usr/include/stdio.h & /usr/include/stdlib.h

preprocessor: removes the Comment, replaces with one blank

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

cpp: replaces EXIT_SUCCESS with 0 on linux

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

1. cpp first processes the file (cpp is called by gcc)
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** (machine code)
   - -o specifies the name of the executable (default: **a.out**)

X

cpp conditional (and macro) only operations

- You can use **conditional preprocessor tests** (like if-else statements) around blocks of code

  `#ifdef MACRO, #ifndef MACRO, #else, #endif`

- In this use, the MACRO is called the guard MACRO ("guards" entry to the following block)

`#ifdef MACRO` if MACRO is defined the block is included otherwise `#else` block (if any) is included

`#ifndef MACRO` if MACRO is NOT defined the block is included otherwise `#else` block (if any) is included

`#endif` is the end of a block

```
#define MACRO           // defines MACRO  -- #define MACRO 8 defines macro and assigns a value of 8
#undef MACRO            // undefines MACRO
```

```
#define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    int x[8];
    ….
    return;
}
```

```
// #define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    short x[8];
    ….
    return;
}
```

cpp conditional tests: header guards

- Header guards ensure that only **one copy of a .h file** is **included in a source file**
- **A Convention**: header guard (macro) NAME (all capital letters) is created as follows:
  - use the **filename of header file but** in all caps
  - **replace** the **period in** header file **name** with an _
  - Example: file sum.h header guard macro name is SUM_H

- How do you use "header guards" in your code?

```
#ifndef NAME_H          // first line in the file
#define NAME_H
      . . .
#endif                  // last line in the file
```

```
#include "sum.h"
// file ex.c
int main(void)
{
    int x[MAX];
```

ex.i

```
int main(void)
{
    int x[8];
```

header guards (two lines)

header guards (one line)

```
#ifndef SUM_H
#define SUM_H

#define MAX 8

#endif
```

file sum.h

```
#include "sum.h"
// file func.c
int func(void)
{
    int z[MAX];
```

func.i

```
int func(void)
{
    int z[8];
```


X

Why header guards are needed

file sum.h

```c
#ifndef SUM_H
#define SUM_H

#define MAX 8
#endif
```

file bar.h

```c
#ifndef BAR_H
#define BAR_H
#include "sum.h"
#define MIN 1
#endif
```

```c
#include "sum.h"
#include "bar.h"

// file ex.c
int main(void)
{
    int x[MAX];
    int z = MIN;
```

```c
#ifndef SUM_H
#define SUM_H

#define MAX 8
#endif
#ifndef BAR_H
#define BAR_H
#ifndef SUM_H
#define SUM_H

#define MAX 8
#endif
#define MIN 1
#endif

// file ex.c
int main(void)
{
    int x[MAX];
    int z = MIN;
```

SUM_H **is defined here**

SUM_H **is already defined here**
So
#define MAX 8 **is not included**

```c
int main(void)
{
    int x[8];
    int z = 1;
```


x

Quick Look: Character and String Literals (more later)

- Usually used to store characters – thus things like file names
- char literals: a single (1) character **inside** a set of **single quotes** 'a'
- string literals: 0 or more characters inside a set of **double quotes** "string"

```
char x = 'a';            // 'a' is a character literal

printf("Hello World!"); // "Hello World!" is a string literal

char a1[] = "xyz";     // char array initialized with contents of a string literal
```

- Problem: How do you place a **non-printable character** like a **newline** in a literal?
  - The **following are not legal** in C as a **newline** in a **source file represents** a statement delimiter (white space) in C

```
char x = 'a
';
```

```
printf("Hello World!
");
```

  - Solution: C has a special **line continuation character **

x

There are three different uses for \ in C

1. Line continuation sequence a \ followed by zero or more whitespace ending in a newline at the end of a **source line**

Use only when no other choice

```
char a[] = "string: Hello \
World";
```

Poor style use a block comment

```
// line comment \
 rest of line comment
```

Not needed do not do this

```
x = x +\
5;
```

2. How do you put a single ' as a character literal or a single " inside a string literal?

- You use an **escape character \\:** which escapes the special meaning of the next character inside a character or a string literal

char sequence	Description
`'\\' or "\\"`	`\ char`
`'\'' or "\'"`	`single quote`
`'\"' or "\""`	`double quote`

```
char a = '\''; // char: '
```

```
char b = '\\'; // char: \
```

```
char c = '"'; // char: "
```

```
char d[] = "ab\""; // string: ab"
```

```
char e[] = "ab\\"; // string: ab\
```

```
char f[] = "ab'"; // string: ab'
```

```
char a[] = "a "string"";  // syntax error ; expected
char a[] = "a \"string\""; // ok
```


x

There are three different uses for \ in C - continued

3. Embed characters with a special meaning inside a (char or string) **literal** using a two-character sequence starting with a \ followed by a single character

- This is typically used for characters that are "non-printable". Here are some examples:

char sequence	Description
`'\n' or "\n"`	`newline char`
`'\r' or "\r"`	`carriage return`
`'\t' or "\t"`	`tab char`
`'\b' or "\b"`	`backspace`
`'\0' or "\0"`	`null char`

```
printf("\n\nHello World!\n\n");
```

```
printf("\n\nHello\tWorld!\n\n");
```


x

Characters In C

\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 column: decimal integers

ASCII Chars are 0-127
(stored in 8 bits)
Many of the values are not "printable"

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.

Understanding Comments in C (Prep for PA2 and PA3)

- In PA2 (design) and PA3 (program in C), you are going to **write equivalent preprocessor code** to **replace each comment in an input file** with a **single space character (a blank space)** while writing the rest of the input to output unaltered
- **IMPORTANT**: the preprocessor **does NOT perform** any **syntax checking**

```
/* this is /* one block comment */ text outside comment
```

```
// this is // one line comment
text outside comment
```

```
/* block comment
// part of block comment not a line comment
yet more block comment
*/ text outside comment
```

```
// line comment /* part of line comment not a block comment */
```

```
//  line comment /* part of line comment not the start of a block comment
oops! text outside of comment, this is not a comment anymore */
```

x

Complexity for programming a preprocessor: Literals may contain what appears to be comments, but are not

```
char x = 'a';                  // 'a' is a character literal

printf("Hello World!");        // "Hello World!" is a string literal
```

```
"/* text */" not a comment but a string literal whose contents looks like a block comment
```

```
"// text" not a comment but a string literal whose contents looks like a line comment
```

```
'/* text */' not a comment but a character literal whose contents looks like a block comment
```

```
'// text' not a comment but a character literal whose contents looks like a line comment
```


x

Data types: C Versus Java

Data Types	Java	C
Character	`char     // 16-bit unicode`	`char   // 8 bits (varies by hardware)`

integers	`byte     // 8 bits`
	`short    // 16 bits`
	`int      // 32 bits`
	`long     // 64 bits`
		`(unsigned, signed) char  // see row above`
		`(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

Review: Binary Numbering

- Binary is base 2
  - *adjective:* being in a state of one of two **mutually exclusive** conditions such as **on** or off, true or false, molten or frozen, presence or absence of a signal
  - From Late Latin *bīnārius* (“consisting of two”)
- Two symbols:

  0 1

- Numbers in C starting with 0b are binary
- <u>Example</u>: What is 0b110 in base 10?
  - 0b110 = $110_2 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$

powers of two

- A **bit** is a single binary digit
- A **byte** is an 8-bit value

$$Unsigned\ binary\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 2^i = b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \ldots + b_1 2^1 + b_0 2^0$$

x

Review: Hexadecimal Numbering

- hexadecimal is base 16
  - From “hexa” (Ancient Greek ἑξα-) ⇒ six
  - and from “decem” (Latin) ⇒ ten
- Sixteen symbols

  0 1 2 3 4 5 6 7 8 9 a b c d e f

- Numbers in C starting with 0x are hexadecimal
  - 16_{10} = $0x10_{16}$
- Example: What is 0xa5 in base 10?
  - 0xa5 = $a5_{16}$ = $(10 \times 16^1) + (5 \times 16^0) = 165_{10}$
- Hexadecimal numbers are very commonly used in programming to express binary values
  - Imagine the difficulty in correctly expressing a 64-bit binary value in your code

$$Unsigned\ Hex\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 16^i \ = \ b_{n-1}16^{N-1} + \ b_{n-2}16^{N-2} + \ \ldots + \ b_1 16^1 + \ b_0 16^0$$

x

Number Base Overview (as written in C)

- Decimal is base 10 and Hexadecimal is base 16,
- **Hex digits** have 16 values 0 - 9 a - f (written in C as 0x0 – 0xf)
- No standard prefix in C for binary (most use hex)
  - gcc (compiler) allows 0b prefix others might not

Hex digit	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

Hex digit	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

x

Binary <---> Hexadecimal Equivalences

- Hex → Binary: $16^1 = 2^4$ 1 digit hex = 4 digits binary
  1. Replace hex digits with binary digits
  2. drop leading zeros
  - Example: 0x2d to binary
    - 0x2 is 0b00**10**, 0xd is 0b1101
    - Drop two leading zeros, answer is 0b**10**1101
- Binary → Hex: $2^4 = 16^1$
  1. Pad with enough leading zeros until number of digits is a multiple of 4
  2. replace each **group of 4** with the **HEX equivalent**
  - Example: 0b101101
    - **Pad on the left** to: 0b 0010 1101
    - Replace to get: 0x2d

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

x

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$

$$0x\ \overbrace{1111}^{f}\ \overbrace{1010}^{a}\ \overbrace{0101}^{5}\ \overbrace{0011}^{3}$$

$$\boxed{0b1111101001010011}$$

binary start with a 0b in C

x

Binary to Hex (group 4 bits per digit from the right)

- 4 binary bits is one Hex digit $2^4 = 16^1$

$$0b\ \underbrace{0110}_{6}\ \underbrace{1010}_{a}\ \underbrace{0011}_{3}\ \underbrace{1111}_{f}$$

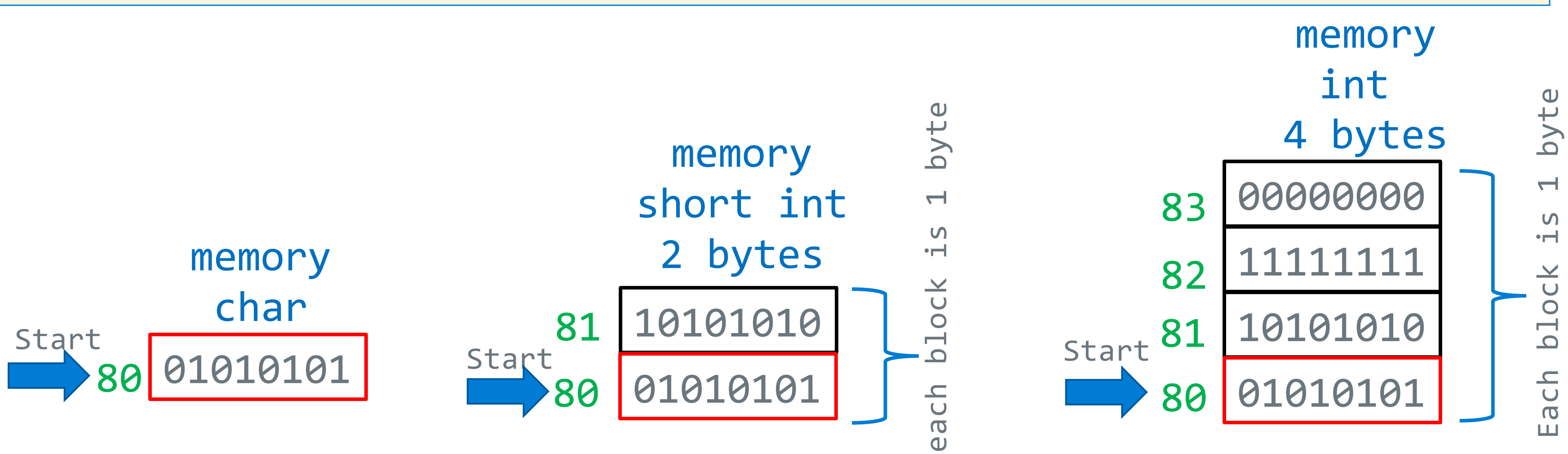
$$\boxed{0x6a3f}$$

hex start with 0x in C

x

Variables in Memory: Size and Address

- The number of **contiguous bytes** a variable uses is based on the *type* of the variable
  - Different variable types require different numbers of contiguous bytes
- ***Variable names*** map to a *starting address in memory*
- Example Below: Variables all starting at address 0x80, each box is a byte

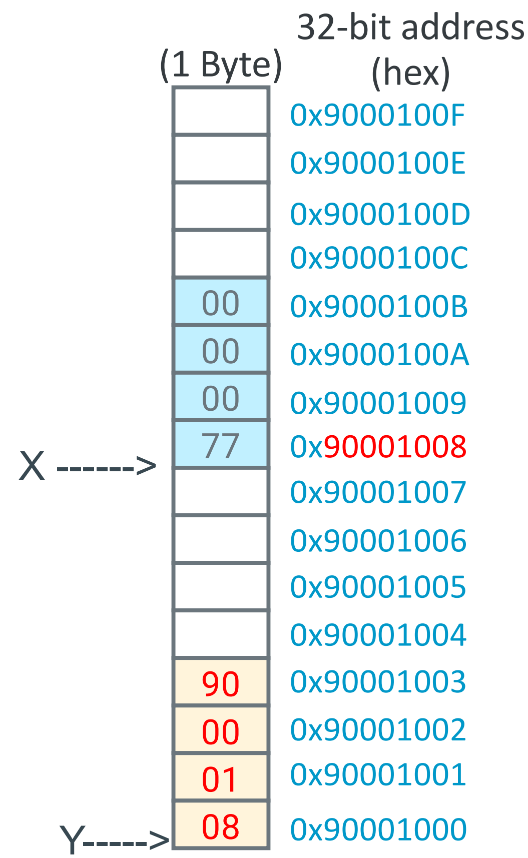


x

Memory and Variables

- An address refers to a location in memory, the lowest or first byte in a contiguous sequence of bytes
- Consider the following situation
  - The variable x is at memory address 0x90001008
  - The variable y is at memory location 0x90001000
  - and the statement

    x = x + y
- The **name** of a variable is on the **right side** of the = evaluates to a **memory address**
- The **name** of a variable is on the **left side** of the = evaluates to the **contents of memory at that address**



x

Variables in C

- **Integer types**
  - `char` [default: unspecified!]
  - `int` [default: signed]
- **Floating Point**
  - `float`, `double` [always signed]
- **Optional Modifiers** for each base type
  - `short` [int]
  - `long` [int, double]
  - `signed` [char, int]
  - `unsigned` [char, int]
  - `const`: read only
- **char type**
  - **One byte** in a **byte addressable** memory
  - **Signed** vs **Unsigned** implementation dependent
  - **Be careful** char is unsigned on arm and signed on other HW like intel

C Data Type	AArch-32 contiguous Bytes	AArch-64 contiguous Bytes
char (arm unsigned)	1	1
short int	2	2
unsigned short int	2	2
int	4	4
unsigned int	4	4
long int	4	8
long long int	8	8
float	4	4
double	8	8
long double	8	16
pointer *	4	8

word size is the size of the address (pointer)

x

Caution: Char type can be either signed or unsigned

- **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)
  - char default is unsigned

    255
- On Intel 64-bit (ieng6)
  - char default is signed

    -1

x

Fixed size types in C (later addition to C)

- Sometimes programs need to be written for a particular range of integers or for a particular size of storage, regardless of what machine the program runs on
- In the file **`<stdint.h>`** the following fixed size types are defined for use in these situations:

Signed Data types	Unsigned Data types	Exact Size
`int8_t`	`uint8_t`	8 bits (1 byte)
`int16_t`	`uint16_t`	16 bits (2 bytes)
`int32_t`	`uint32_t`	32 bits (4 bytes)
`int64_t`	`uint64_t`	64 bits (8 bytes)

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);
double pi = 3.14159;
char c = 'Q';
```

```
for (int i = 0; i < end; i++) { // variable i is a loop guard
    if (i % 2 == 0) {
        x += i;
    }
}
```

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

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

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?

- 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

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal** evaluation

```
if ((x == 5) || (y > 3))  // if x == 5 then y > 3 is not evaluated
```

- **Each** expression argument is evaluated **in sequence** from left to right including any side effects (modified using parenthesis), **before** (optionally) evaluating the next expression argument
- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated *(for performance)*

x

Program Flow – Short Circuit or Minimal Evaluation

```
if ((a != 0) && func(b))     // if a is 0, func(b) is not called
    do_something();
```

```
// if (((x > 0) && (c == 'Q')) evaluates to non zero (true)
// then (b == 3) is not tested

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


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

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 initial (start) function: int main()
- main() is the **first function in your code** to run/execute
  - main() is **not the first function** to run in a Linux process, it is the ***C runtime startup code***
    - later in course
  - **You should never make a call to main() from your code**

```
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 definition

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

  return sum;
}
```


x

C Function Definitions - 2

remember this is a 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

```c
// 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 in other languages)*

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

- **Assignment inside conditional test** with **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

Textbook Over-ride: Linux Return Value Convention

- **In your code, 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**, **<u>always</u> 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;
```


x

Setting program termination return (status) values

Indicating your program operated correctly

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

Indicating your program operated incorrectly/errors

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

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 */`

```
#include <stdio.h>  // import the public interface

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

- Write chars to the file identified by **file** (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, ...);`

- Type **%** `man 3 printf` for more information on *format*

x

Some Formatted Output Conversion Examples

- Conversion specifications example
  - **%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;
char z = 'i';
char a[] = " Hello\n";

printf("%c = %d,%s", z, i, a); // write to stdout
fprintf(stderr, "This is an error message to stderr\n");
```

  - Output

```
i = 10, Hello
This is an error message to stderr
```


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 public interface
```

```
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

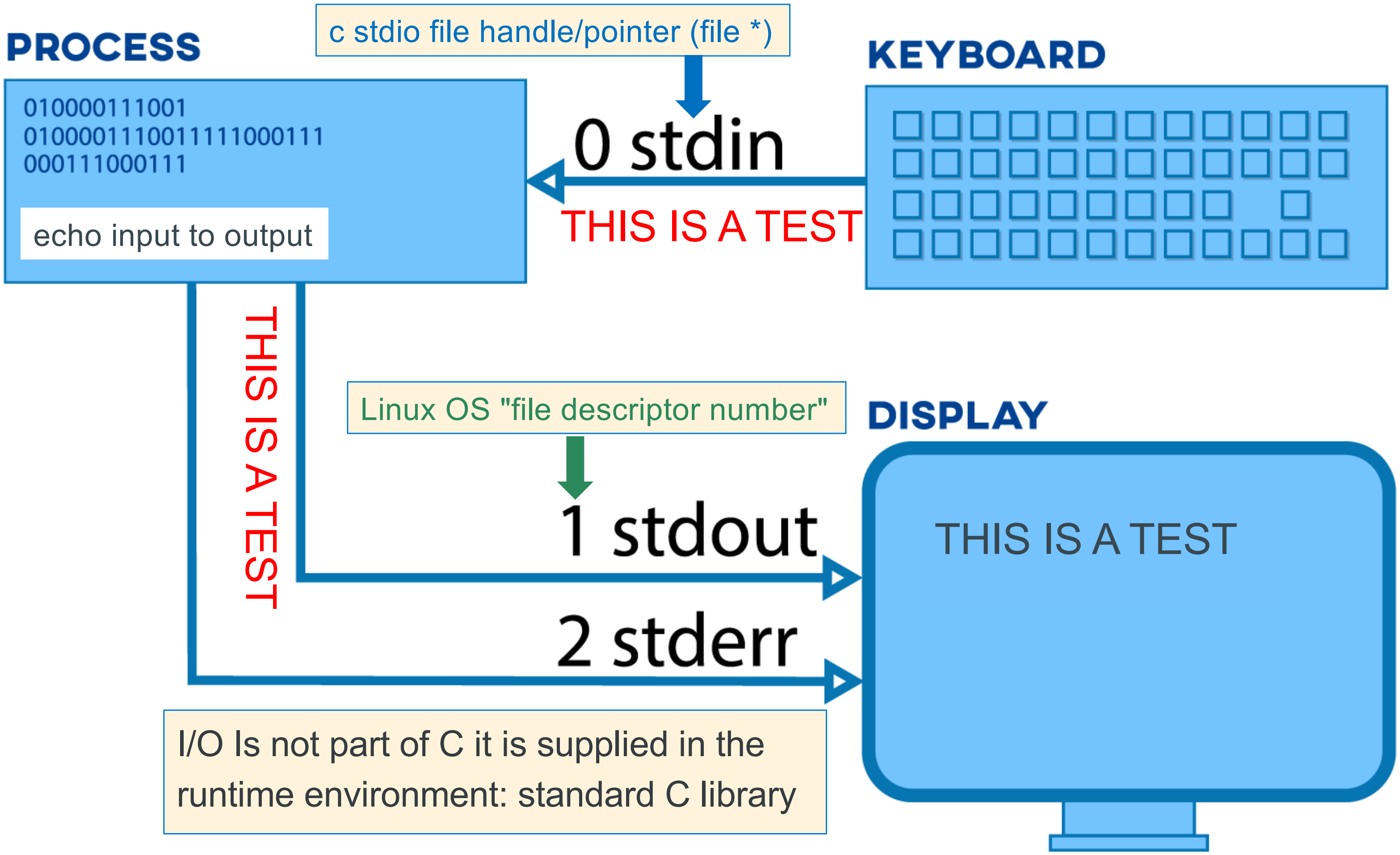
- Make sure you use int variables with putchar() and putchar()
- Both functions return an int because they must be able to return both valid chars **and** indicate the **EOF condition (**-1) which is outside the range of valid characters

Why is character I/O using an int?

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

x

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

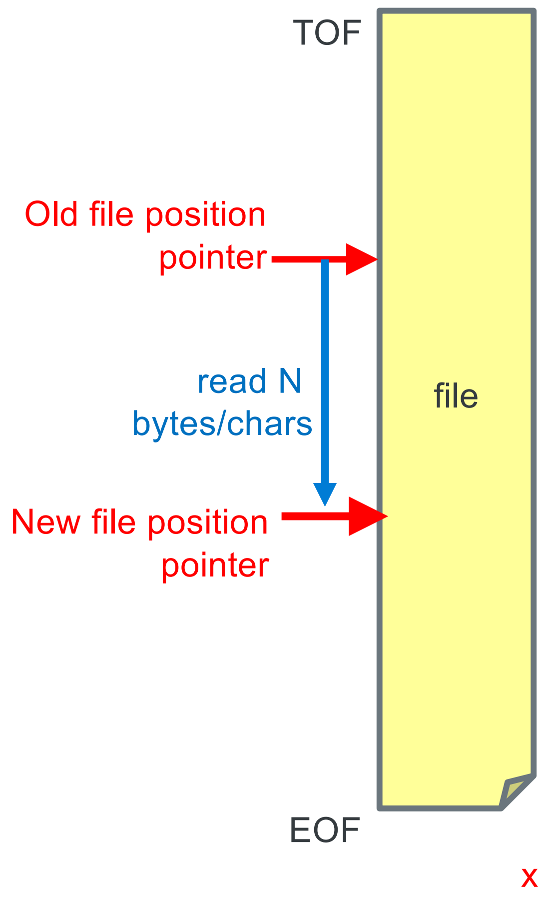


I/O Is not part of C it is supplied in the runtime environment: standard C library

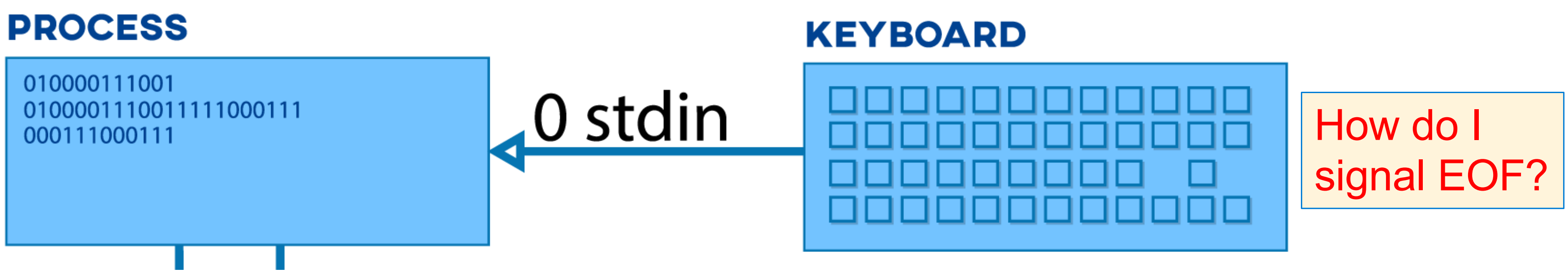
x

C standard I/O Library (stdio) File I/O
File Position Pointer and EOF

- Read/write functions in the standard I/O library *advances* the ***file position pointer*** from the ***top of a file*** (before the 1st byte if any) towards the ***end of the file*** **after each call** to a read/write function
  - **Side effect of call:** file position pointer moves towards the **end of file** by number of bytes read/written
- **standard I/O File position pointer** indicates where in the file (byte distance from the top of the file) the next read/write I/O will occur
- Performing a sequence of read/write operations (without using any other stdio functions to move the file pointer between the read/write calls) performs what is called **Sequential I/O** (sequential read & sequential write)
- EOF condition state may be set after a **read operation**
  - After the last byte is read in a file, additional reads results in a **function return value** of EOF
  - **EOF signals** no more data is available to be read
  - EOF is **NOT a character in the file**, but a condition state on the stream
  - EOF is usually a #define EOF -1 macro located in the file stdio.h (later in course)



stdio File I/O – Working with a Keyboard



- How can you have an EOF **when reading from a keyboard**?
- stdio I/O library functions **designed** to work primarily on **files**
  - With keyboard devices the semantics of *file operations* needs to be *"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
  - **This is NOT an EOF condition**
- To set an *EOF condition from the keyboard,* type on an input line all by itself:

  *two key combination (**ctrl key** and the **d key** at same time),* **followed by a return/enter**

  ctrl-d   *often shown in slides etc. as **^d***

x

Character I/O (Also the Primary loop in PA3)

```
// copy stdin to stdout one char at a time
#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

```
% ./a.out
thIS is a TeSt
thIS is a TeSt
^d
%
%./a.out < a > b
```

Typed on keyboard

Printed by program

Typed on keyboard

Copies file a to file b

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

Make sure you use int variable with getchar() and putchar()!

x

Background: What is a Definition?

- **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

Background: What is a Declaration?

Declaration: describes a ***thing*** – specifies types, **does not create** an instance

- **Each declaration** has an associated ***identifier*** (the name)

1. **Function prototype** describes how to write the code to call a function defined elsewhere
   - **Identifier** is the **function name**
   1. Describes the **type of the function return value**
   2. Describes the **types of each of the parameters**
2. **Variable declaration** describes how to write the code to use a variable in a statement
   - **Identifier** is the **variable name**
   - Describes the **type of a variable** that is **defined elsewhere**
3. **Derived and defined type description**
   - **Identifier** describes the derived/defined type
   - struct, arrays, plus others (covered later)

- An **identifier** may be **declared multiple times**, but **only defined once**
- A **definition is also a declaration in C**

X

Definitions and Declarations Use in C

You must **declare a function or variable <u>before</u> you use it**

- **Warning:** Use before declaration will implicitly default to **int**

sumit() is defined and declared here

Independent Translation Unit: the granularity (unit) of source which is compiled or assembled

Default Definition and declaration validity:

1. Restricted to the file (translation unit) where they are located **<u>and</u>**
2. **Start at the point** of definition or declaration in the file to the end of the source file (**translation unit**)

Restrictions that we need to relax
(1) sum() must be defined in the same source files
(2) sum() appear before it is used by main()

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 8
int sumit(int max)
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}

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

i, sum, are defined and declared here

sumit() is used here

x

Function Prototypes: Creating a Function Declaration

Function prototype is a **function declaration** in C

```
returnType fname(type_1, …, type_n);   // function prototype
```

- **Function prototype** is function definition header followed by a single semicolon (;) **NO code block**
- **Describes the function** from that **point** in the source file

- C requires the **function declaration <u>to be seen</u> in the source file <u>before use</u>**

- A **function prototype** for sum() enables:

1. body of sum() to be either after main() in the same source file **or**
2. body of sum() to be in a different source file

```c
#include <stdlib.h>
#define NUM 100
int sum(int);    // function declaration starts here
int main(void)
{
   sum(NUM);     // rest of code not shown
   return EXIT_SUCCESS;
}
int sum(int max) // function definition is here
{
  int i, sum = 0;
  for (i = 1; i <= max; i++) {
    sum += i;
  }
  return sum;
}
```

this is the code block

Common practice: Function prototypes in a .C file are usually **placed at the top the file**

X

C Variable Storage Lifetime

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 or program termination
   - 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

C and Scope

- **Scope**: **Range** (or the extent) of instructions over which a name/identifier **is allowed be referenced** by C instructions/statements
  1. **File Scope: Range is within** a **single source file** (also called a **translation unit**)
  2. **Block Scope: Range is within** an **enclosing block** (for variables only)

```
int global;                                 // global variable with file scope

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


x

Nested Scope

- **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

Variables in C

- **Global variables**
  - **Defined at file scope** (outside of a block)
  - have static storage duration
  - global variables defined without an initial value default to 0 (set prior to program execution start)
  - global variables defined with an initial value are set at program start
- **Local (block scope) variables** (including function parameter variables)
  - **Defined at block scope** (inside of a block)
  - have automatic storage duration
  - block scope variables defined without an initial value have an **undefined** initial value
  - block scope variables defined with an initial are set each time the block is entered
  - All block scope variables become **undefined** at block exit
- **Variable definitions preceded by the keyword** *`static`* have static storage duration including variables defined with block scope

```
int global;           // global with static storage duration
int foo(void)
{
    static int s = 0; // "local" with static storage duration
    int x;            // "local" with automatic storage duration
}
```


x

Example:
Block scope (local) static storage duration variables

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int foo(void)
{
    static int s = 0; //static storage duration, set to 0 at program start
    return s += 1;
}

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

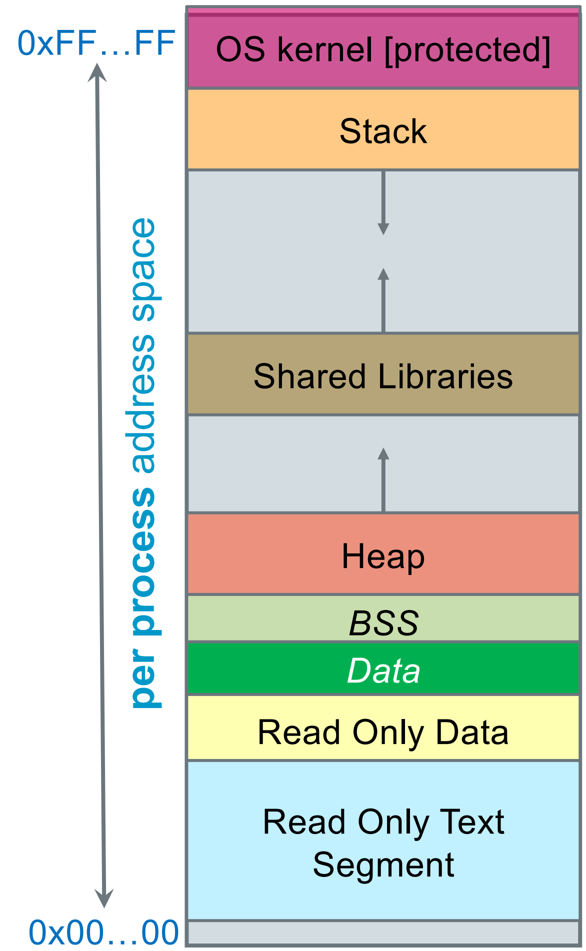
```
% ./a.out
1 2 3 4 5
%
```


x

Where things are in Memory

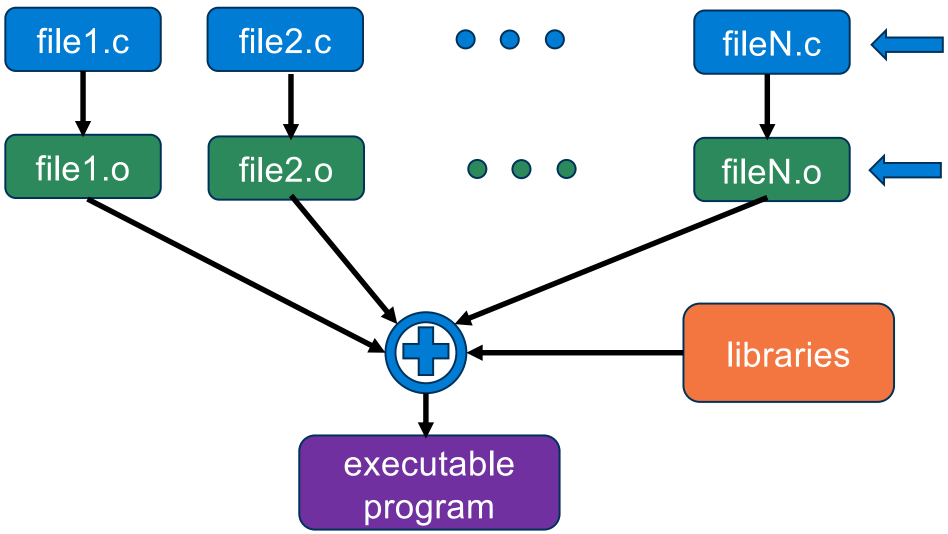
- When your program is running it has been loaded into memory and is called a process (under the control of the OS)
- *Stack segment: Local variables: defined in functions*
    - Allocated/freed at function call entry & exit
- *Data segment + BSS: Global* and *static* variables
    - Allocated/freed when the entire process starts/exits
        - BSS - Static variables with an implicit initial value
        - Static Data - Initialized with an explicit initial value
- Heap segment: *dynamically-allocated (during runtime)* variables
    - Allocated with a function call to a library routine
    - Managed by the library routines linked to your code
- *Read Only Data: immutable* Literals
- *Text*: Your code in machine language + non-shared libraries

Process memory during execution



x

Real programs are distributed across multiple files



Example: fixing a bug in a existing program

1. You fix bug in just fileN.c
2. Only need to recompile fileN.c to FileN.o (all the other .o files are fine)
3. Relink all .o files and libraries
4. Test the executable

- **Large programs** in one source file can be very difficult to manage
  - Consider a program with many millions of lines of code
  - And there are 100's developers working on it, changing source parts of the code
  - The program is being rebuilt (compiled/linked) and tested several times a day
- **Approach:** Break a program into **individual translation units** (source files)
  - **Compile them individually** and **then link them together**
  - Only need to recompile those source files that have changed

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 **source file it is defined in**
- External Linkage: function and variables with external linkage **can be referenced anywhere in the entire program**
  - **Global variables** and **all functions** have external linkage by **default**
  - **Unless explicitly declared,** the **default type** is **int** for **both functions** and **global variables**
  - **However**, the compiler must know the correct types before the use of a function or a variable, so it is able to generate the correct code
    - **NEVER DEPEND implicit default typing**
  - Use **function prototypes** to **declare functions** before use
  - Use the keyword **extern** to "extend the visibility", e.g., declare a global variable before use

```
// example here is at file scope
extern int x;     // declaration
int x = 10;       // definition
```


x

Controlling Linkage Across Files in Multi-File C Programs

- Internal Linkage (private): function and global variables with internal linkage can **only be referenced** in the **same source file**
  - Global variables and functions can be changed to internal linkage by using the keyword **static** in front of the definition (confusingly another use of the word static)
  - Use of the keyword static in front of a **global variable definition** or **function definition** changes it to internal linkage and effectively makes it private to the file they are defined in (It cannot be referenced by another file
  - 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)**
- No Linkage: function parameters, variables defined inside a block (including a functions body)
  - **Remember**: the keyword static in front of a **block scope variable** changes the variable to **static storage duration** (it does not change the linkage)

X

Linkage Examples

```
int global0 = 1;          // external linkage
static int global2;       // internal linkage restricted to this file
int funcA(int x)          // funcA has external linkage; x has no linkage
{
    int y;               // no linkage
}
static int funcB(void) // internal linkage restricted to this file
{ }
```


x

Creating Public Interface files (header files)

- To enable a **source file** to **use any of the functions, global variables**, and **MACROS** defined in another file (separate translation unit)
  - You must create a file that exports all permitted accesses so the compiler can generate the correct code
- **Convention:** For each source file, `file.c`, the **public interface file** is `file.h`
- If a file has no external interfaces, then it does not need a .h file

file.h

declarations | exported information how to use functions etc. in file.h

file.c

definitions | the definitions of functions etc.

- `file.h` contains any
  - public preprocessor macros
  - **function prototypes** for the functions defined in the source file, `file.c` **that you want visible (exported)** for use (called) by functions defined in **other source files**
  - *global variable declarations (external linkage)*
  - **Do not put any** ***definition statements*** in a header file

- `file.c` contains
  - All function and global variable definitions (internal and external linkage)
  - Any private preprocessor macros
  - Any private (internal linkage) function prototypes

X

Creating Public Interface files (header files)

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

```
// file.c
#include <stdlib.h>
#include "file.h"

static int P(char );
        // above: private function prototype

int global;            // initial value is 0
static int private = 1; // private global

int A(int c)
{
// body not shown
}

char B(int x, int y)
{
// body not shown
}

static int P(char z)
{
// body not shown
}
```

public interface for file.c

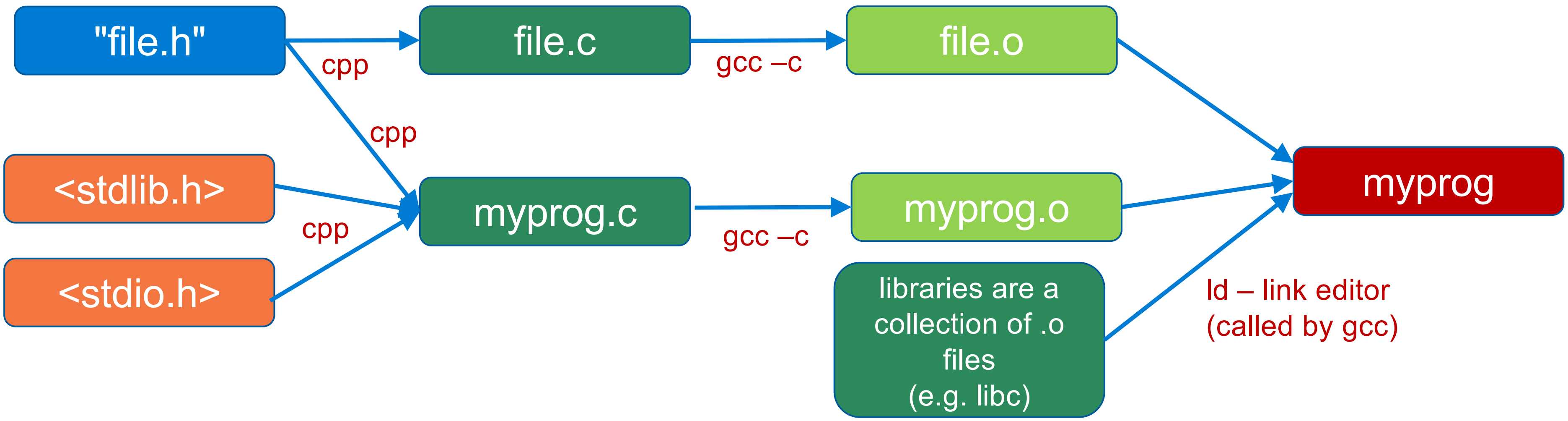
```
// file.h
#ifndef FILE_H
#define FILE_H

#define MAX 5

extern int global;

int A(int);
char B(int, int);

#endif
```

using the public interface

```
// myprog.c
#include <stdlib.h>
#include <stdio.h>
#include "file.h"

// code not shown
int main(void)
{
// body not shown
}
```


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 file.c   # creates file.o
gcc -Wall -Wextra -Werror -c myprog.c   # creates myprog.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 myprog.o file.o -o myprog
```


x

Compiler Warning and unused variable and parameters

- C programming language has many features that when used improperly can lead to runtime issues due to focus on creating code that optimizes performance
  - Example: runtime checks on array boundaries
- gcc besides checking proper language syntax, has the option to include **include heuristic warnings** about potential issues that some consider potential issues in your code
- In CSE30 we require compiling with heuristic checking enabled so you learn to be careful when writing your code, these flags do the checking and requires you to fix them

```
gcc -Wall -Wextra -Werror
```

- As an example, lets look at a couple of warning messages and how to deal with them

x

Compiler warnings 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 (this is legal to do in C)
  - 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) decide to flag this as a potential error
  - The Fix: use the comment `/* FALL THROUGH */` (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

Compiler warnings on unused variables and parameter

- While you are developing a program, you may have functions that you are writing but have not completed the body of the code, but you are compiling it while working on other code
- **TEMPORAILY** suppress warning statement use the following for a used variable or parameter: var

```
(void) var;    // do not submit code to gradescope with this, it will cost you points….
```

```
...
int c = 0;
...
state = nextstate(c);
...
```

```
int nextstate(int c)
{
    int j;

    return 0;
}
```

```
int nextstate(int c)
{
    int j;

    (void) c;
    (void) j;
    return 0;
}
```

```
% gcc -c sample.c
% ls –l
total 4
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24  45 Mar 28 13:14 sample.c
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 840 Mar 28 13:17 sample.o
%
% gcc -c -Wall -Wextra -Werror sample.c
sample.c: In function ‘nextstate’:
sample.c:3:6: error: unused variable ‘j’ [-Werror=unused-variable]
  int j;
      ^
sample.c:1:19: error: unused parameter ‘c’ [-Werror=unused-parameter]
 int nextstate(int c)
               ~~~~^
cc1: all warnings being treated as errors
```

```
% gcc -c -Wall -Wextra -Werror sample.c
% ls –l
total 4
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24  45 Mar 28 13:18 sample.c
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 840 Mar 28 13:19 sample.o
```


x

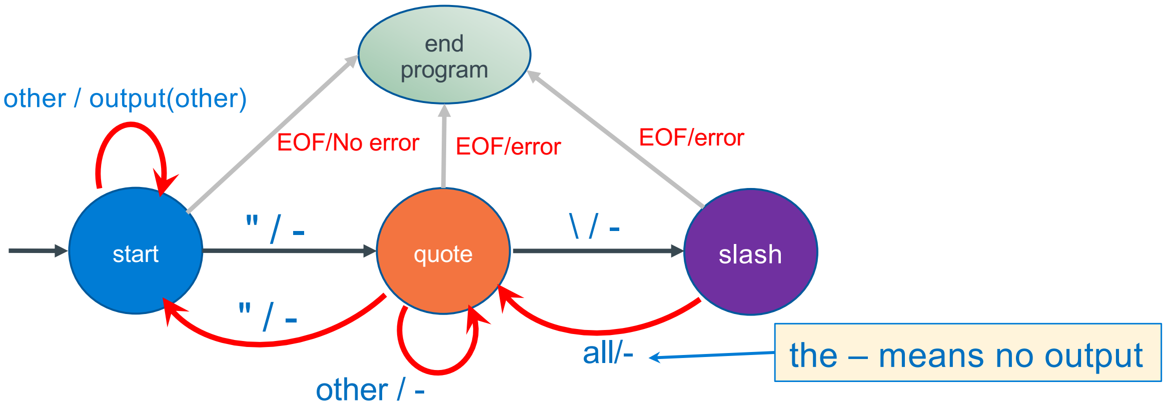
Reference Slides

- Slides in this section are not used in class but contain material that you will find useful

x

PA3: 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 – The Files

- Break the program into three files
- `noq.c` is where main loop is, imports declarations in `states.h`
- `states.h` is the public interface to the state handlers in `states.c`
- `states.c` definition of the state handler functions, imports declarations in `states.h`
- Observe there is no .h file for noq.c, as it does not have any exports

```
noq.c
#include "states.h"
main() function
   current state variable
```

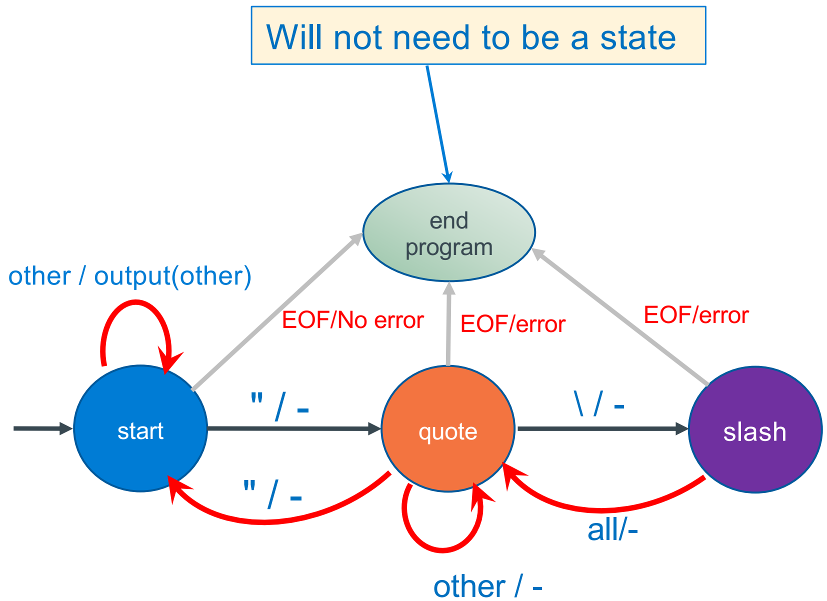
```
states.h
#defines for each state "value"
function prototypes for each state
```

```
states.c
#include "states.h"
function definitions for each state
```


x

Programming a Deterministic Finite Automaton - states.h

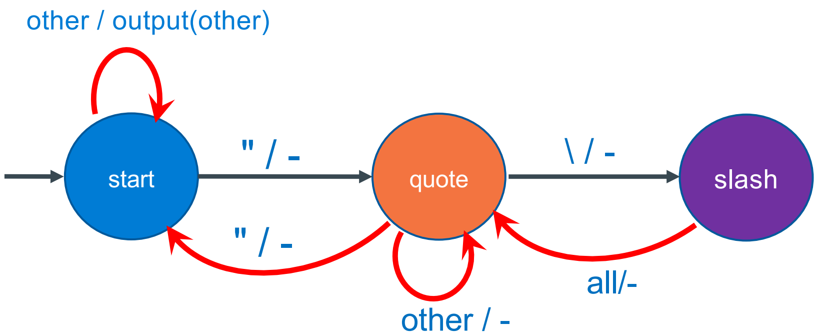
```c
// public interface file states.h
#ifndef STATES_H
#define STATES_H

// Assign a value for each state
#define START   0
#define QUOTE   1
#define SLASH   2

// Function prototypes
// for each state handler
int STARTstate(int);
int QUOTEstate(int);
int SLASHstate();

#endif
```



- Each function implements the **arcs** out of that state
  1. **returns the next state** based on the input
  2. **performs** any **actions associated** with **arc taken**

x

Programming a Deterministic Finite Automaton – states.c



states.c

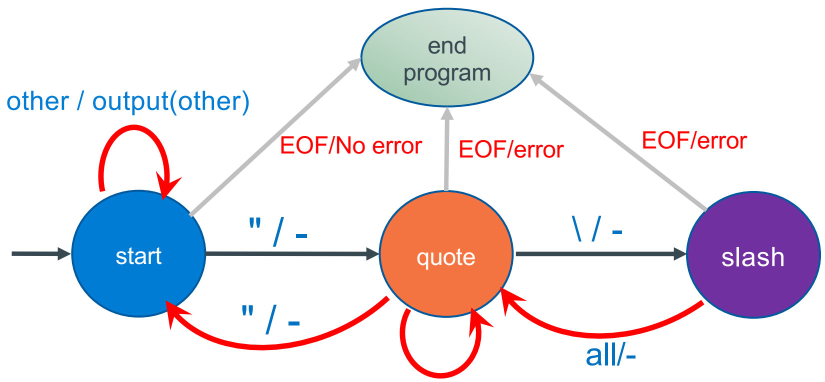
```
#include <stdio.h>
#include "states.h"
int STARTstate(int c)
{
    if (c == '\"')
        return QUOTE;       // saw a double quote
    putchar(c);             // echo input
    return START;           // stay in START
}

int QUOTEstate(int c)
{
    if (c == '\\')
        return SLASH;       // backslash ignore next char
    else if (c == '\"')
        return START;       // closing " go to START
    return QUOTE;
}

int SLASHstate()
{
    return QUOTE;
}
```


x

Programming a Deterministic Finite Automaton – noq.c

primary loop read a char at a time until EOF

process input based on current state




```
int main(void)
{
    int c;                   // input char
    int state = START;       // initial state of DFA

    while ((c = getchar()) != EOF) {
        switch (state) {
        case START:
            state = STARTstate(c);
            break;
        case QUOTE:
            state = QUOTEstate(c);
            break;
        case SLASH:
            state = SLASHstate();
            break;
        default:
            fprintf(stderr, "Error: Invalid state (%d)\n");
            return EXIT_FAILURE;
        }  // end switch
    }  // end while
    /*
     * All done. No explicit end state used here.
     * if not in start state, we have an error
     */
    if (state == START)
        return EXIT_SUCCESS;
    // ok we had an error
    fprintf(stderr, "noq error: Missing end quote \"\n");
    return EXIT_FAILURE;
}
```

call state handlers based on current state
state handlers return next state

check ending "state"

x

Aside: Remember make from CSE15L?

```
# CSE30SP24 DFA Example

# if you type 'make' without arguments, this is the default
PROG        = noq
all:        $(PROG)

# header files and the associated object files
HEAD        =  states.h
SRC         =  noq.c states.c
OBJ         =  ${SRC:%.c=%.o}

# special libraries
LIB         =
LIBFLAGS    = -L ./ $(LIB)

# select the compiler and flags you can over-ride on command line
# e.g., make DEBUG=
CC          = gcc
DEBUG       = -ggdb
CSTD        =
WARN        = -Wall -Wextra -Werror
CDEFS       =
CFLAGS      = -I. $(DEBUG) $(WARN) $(CSTD) $(CDEFS)

$(OBJ):     $(HEAD)

# specify how to compile/assemble the target
$(PROG):    $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) $(LIB) -o $@

# remove binaries
.PHONY: clean clobber
clean:
    rm -f $(OBJ) $(PROG)
```


X

Programming a Deterministic Finite Automaton - testing

```
$ make
gcc -I. -ggdb -Wall -Wextra -Werror     -c -o noq.o noq.c
gcc -I. -ggdb -Wall -Wextra -Werror     -c -o states.o states.c
gcc -I. -ggdb -Wall -Wextra -Werror   noq.o states.o  -o noq
$ ./noq
123"456"789
123789
"123"45"67"
45
"123
456
78"9
9
"test
^d
noq error: Missing end quote "
$ cat in
line1"34"
"line2"line2
line3"
line4
$ ./noq < in > out
noq error: missing end quote "
$ cat out
line1
line2
line3$
```

typed input in red
output in blue

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 -l*libname
Building	% javac Hello.java	% gcc -Wall -Wextra -Werror hello.c -o hello
Running (execution)	% java Hello hello world!	% ./hello hello world!

x

C Versus Java

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

x

C Versus Java

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

x

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…`

x

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** public interface *"header or include files" often used as <filename.h> or "filename.h"*
  - common contents: public (exported) 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 (programmer created)
  - processed by gcc which calls gas (assembler)
- filename.**s**
  - 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, .s,** 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**