Version 2.20

UCSD CSE 30 Section B

Computer Organization and Systems Programming

C Part 1

Keith Muller

DEC PDP 11/45 - 1973

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 again**

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


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)
- **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 initial (start) function: `int main(int argc, char **argv)`
- 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 values are COPIED to a memory location allocated for the argument (like a local variable) – this is very important to know
  - 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;
}
```

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 cause types to default to be of type **int**

sumit() is BOTH 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**)

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

i, sum, are both defined and declared here

sumit() is used here

Forcing function order in a file is a pain….
(1) sum() must be defined in the same source files
(2) sum() appear before it is used by main()
Question: How do we remove this limitation?

x

Function Prototypes: Creating a Function Declaration

Function prototype is how we **declare a function** in C

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

- **Function prototype** is *function definition header* followed by a single semicolon (;) **NO code block**
- **Declares the function** from that **point** in the source file **to the end of JUST THAT 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 (but this will use of interface files – in a few slides)

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

```
#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;
}
```

code block

X

C and Scope Review

- **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 an 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

C Variable Storage Lifetime

1. **Static Storage Lifetime:** valid while program is executing
   - Storage allocated and initialized **prior to program start** (implicit default = 0)

2. **Automatic Storage Lifetime:** valid while enclosing block is activated
   - **Storage allocated and is not implicitly initialized (value = unspecified)** by **executing code** when entering scope and **made available for reuse** by **executing code** when exiting scope
   - It is **not correct to say that automatic storage has been deallocated on exit** (it ***might be)*** but more often is *still part of your program* **and may be referenced from the viewpoint of the OS without causing a runtime fault** if you have an address (pointers later in course)
   - Contents of storage after exiting scope is not changed (why would C act this way?)

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

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, or automatic) variables** (including function parameter variables)
  - **Defined at block scope** (inside of a block)
  - have automatic storage duration, **with one exception (see below)**
  - block scope variables defined without an initial value have an **unspecified** initial value
  - block scope variables defined with an initial are set each time by code when the block is entered
  - All block scope variables become **unspecified** at block exit
- **Variable definitions preceded by the keyword** *`static`* always have static storage duration including variables defined with block scope (when used global variables it restricts scope – later slides)

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


x

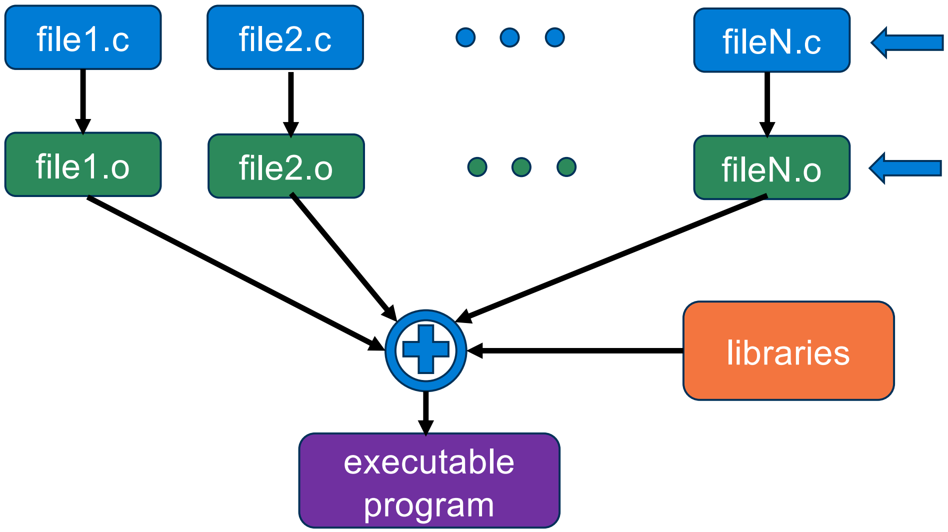
Example:
Block scope (local) static storage duration variables

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

int foo(void)
{
    static int s;  //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());
    printf("\n");
    return EXIT_SUCCESS;
}
```

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


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): functions and global with internal linkage can **only be referenced** in the **same source file**
- Global variables and functions can be defined to have internal linkage by using the keyword **static** in front of the definition (confusingly another use of the word static)

```c
static int global2;
static int funcB(void) { }
```

- 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 to this file, default initial value = 0
int funcA(int x)         // funcA has external linkage; x has no linkage
{
    int y;              // no linkage, no initial value set (value unspecified)
}
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

using the public interface

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

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

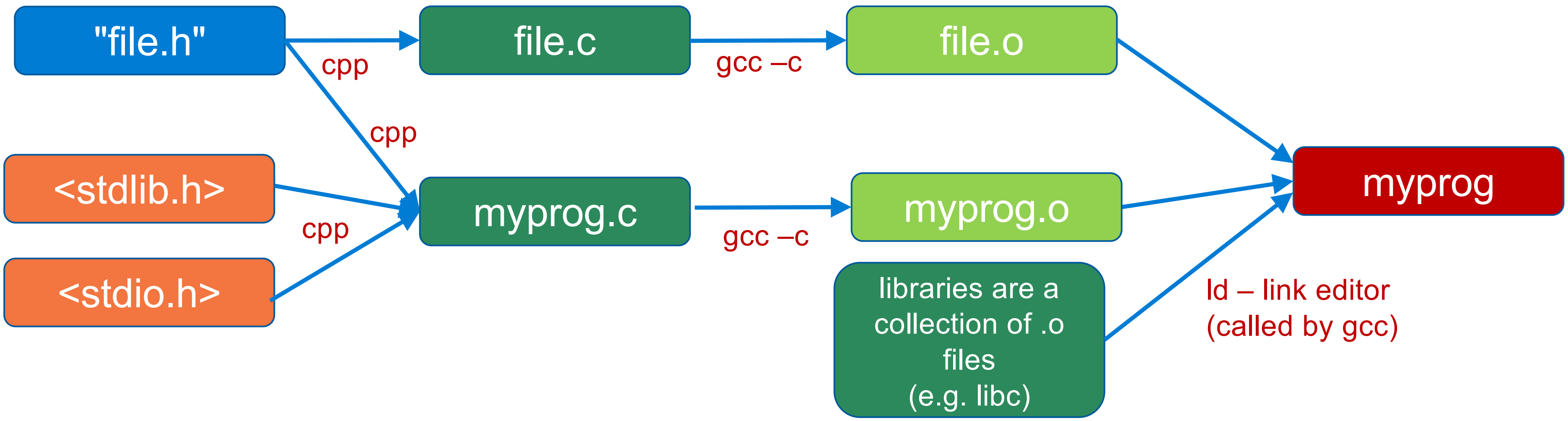
public interface for file.c

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

#define MAX 5

extern int global;

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

#endif
```

```c
// 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
}
```


X

Compiling Multi-File Programs (assembly steps not shown)



1. compile each .c file independently to a .o object file this requires you use the –c flag to gcc to only compile and assemble and NOT to call the liner yet

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

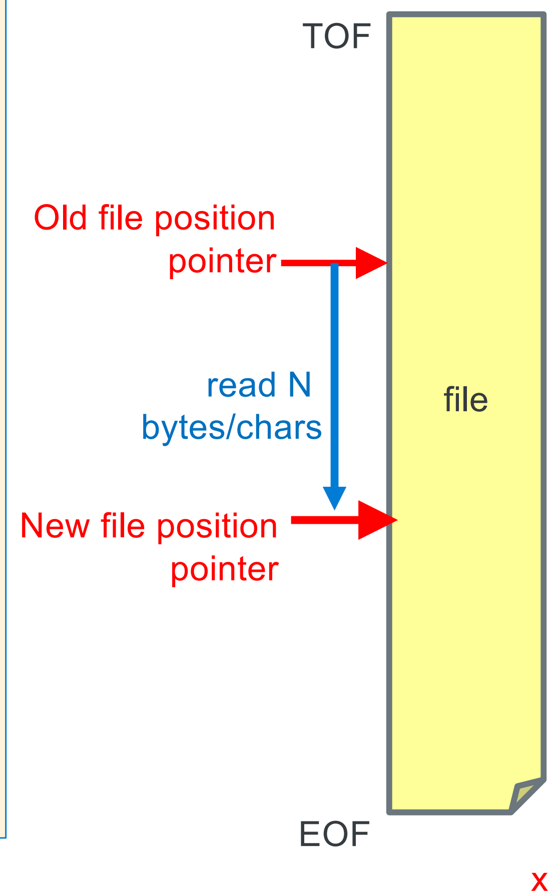
2. link all the .o objects files and libraries (aggregation of multiple .o files) to produce an executable file (gcc calls ld, the linker
   - The .o's in the libraries are automatically linked in as needed to produce an executable file

```
gcc -Wall –Wextra myprog.o file.o –o myprog
```


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)



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 */`

Why is character I/O using an int?

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

```
#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

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

Always check return codes unless you do not need it

Sometimes you may see a (void) cast which indicates ***ignoring the return value is deliberate*** this is often required by many coding standards

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

Typed on keyboard

Printed by program

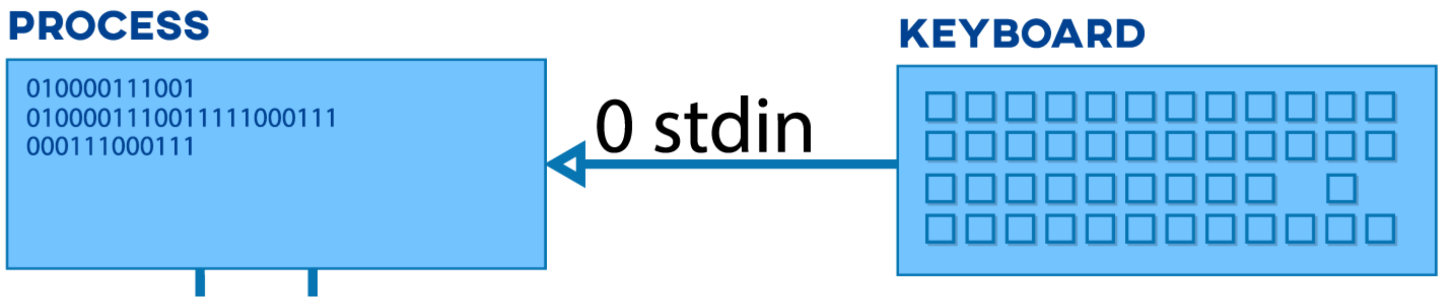
Typed on keyboard

Copies file a to file b

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

X

stdio File I/O – Working with a Keyboard



How do I signal EOF?

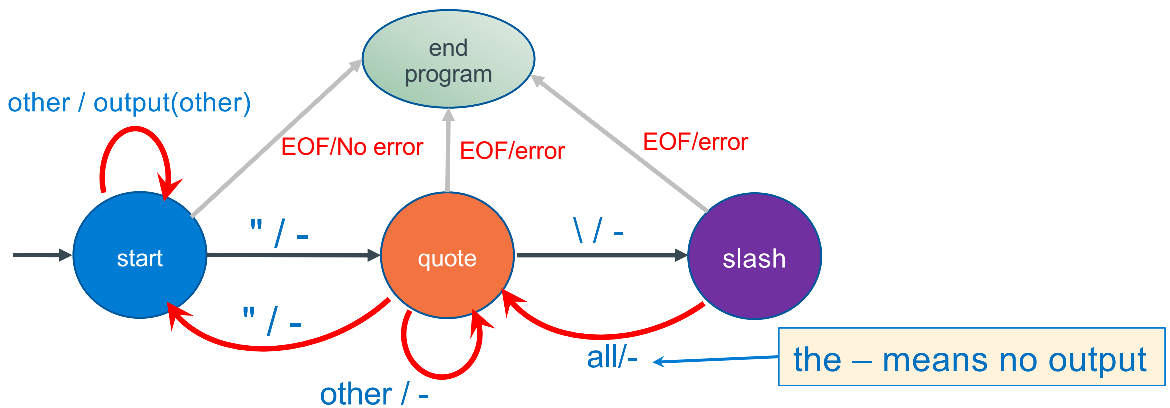
- How can you have an EOF **when reading from a keyboard**?
- stdio I/O library functions **designed** to work primarily on **files**
  - With keyboard devices the semantics of *file operations* needs to be *"simulated"*
- Example: when a program (or a shell) is reading the keyboard and is blocked waiting 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`***

PA3: Programming a Deterministic Finite Automaton

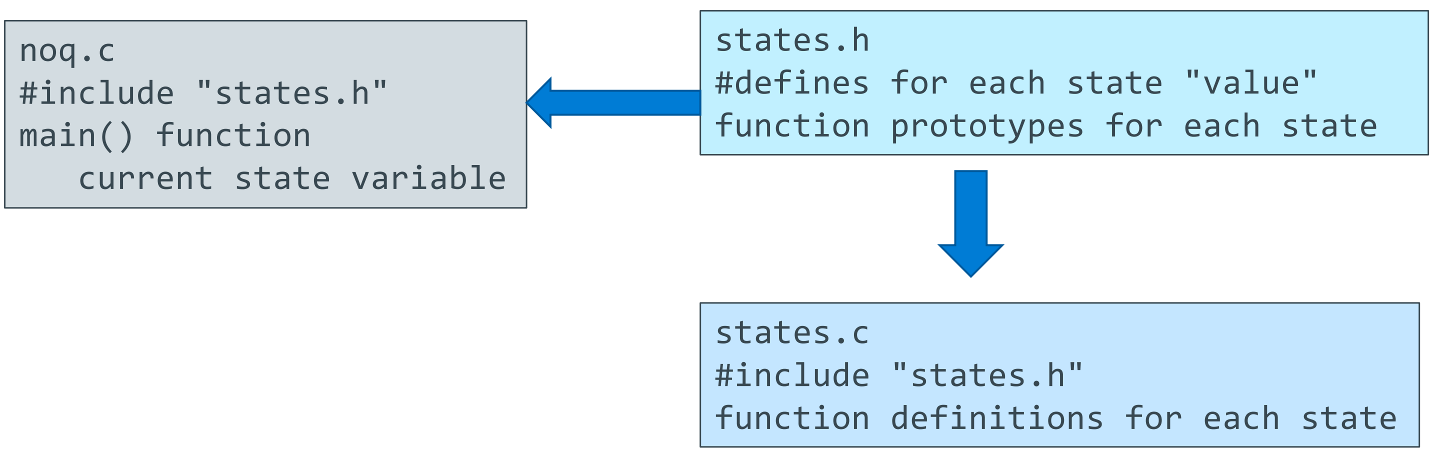
```
Rules for this DFA example
Copy input to output while removing everything in "strings" from output

      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

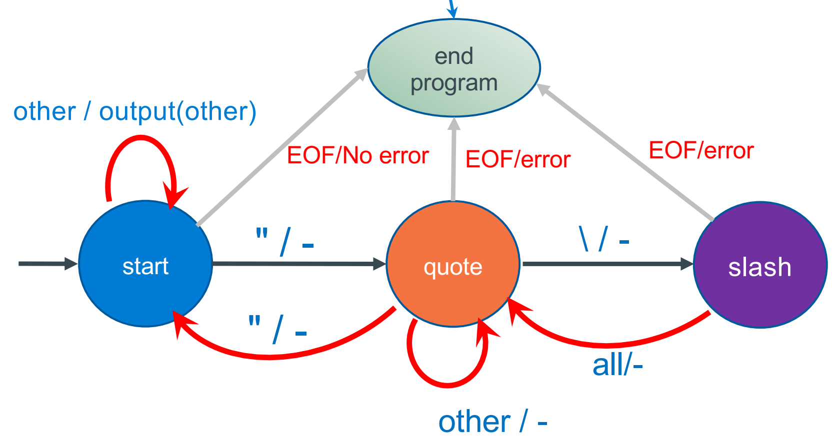


x

Programming a Deterministic Finite Automaton - states.h

```
// 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
```

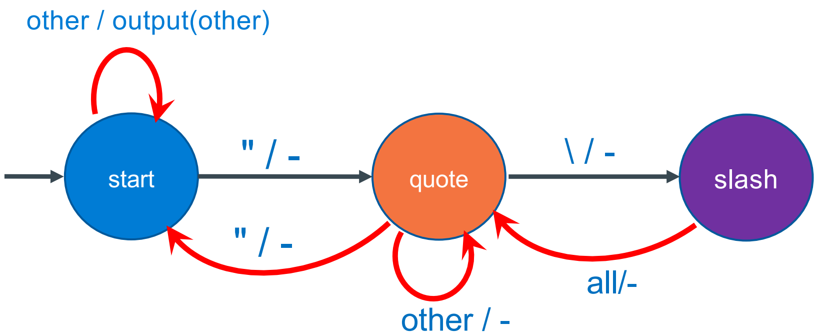
Will not need to be a state



- 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

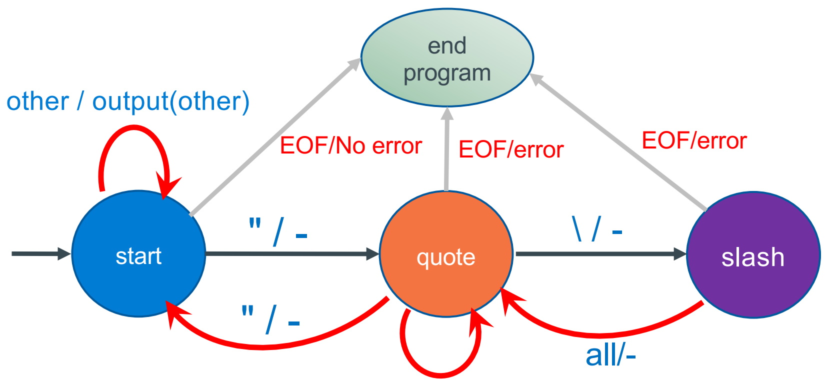
```
#include <stdio.h>
#include "states.h"
int STARTstate(int c)
{
    if (c == '\"')
        return QUOTE;          // saw a double quote
    putchar(c);                // echo input
    return START;              // stay in START
}

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



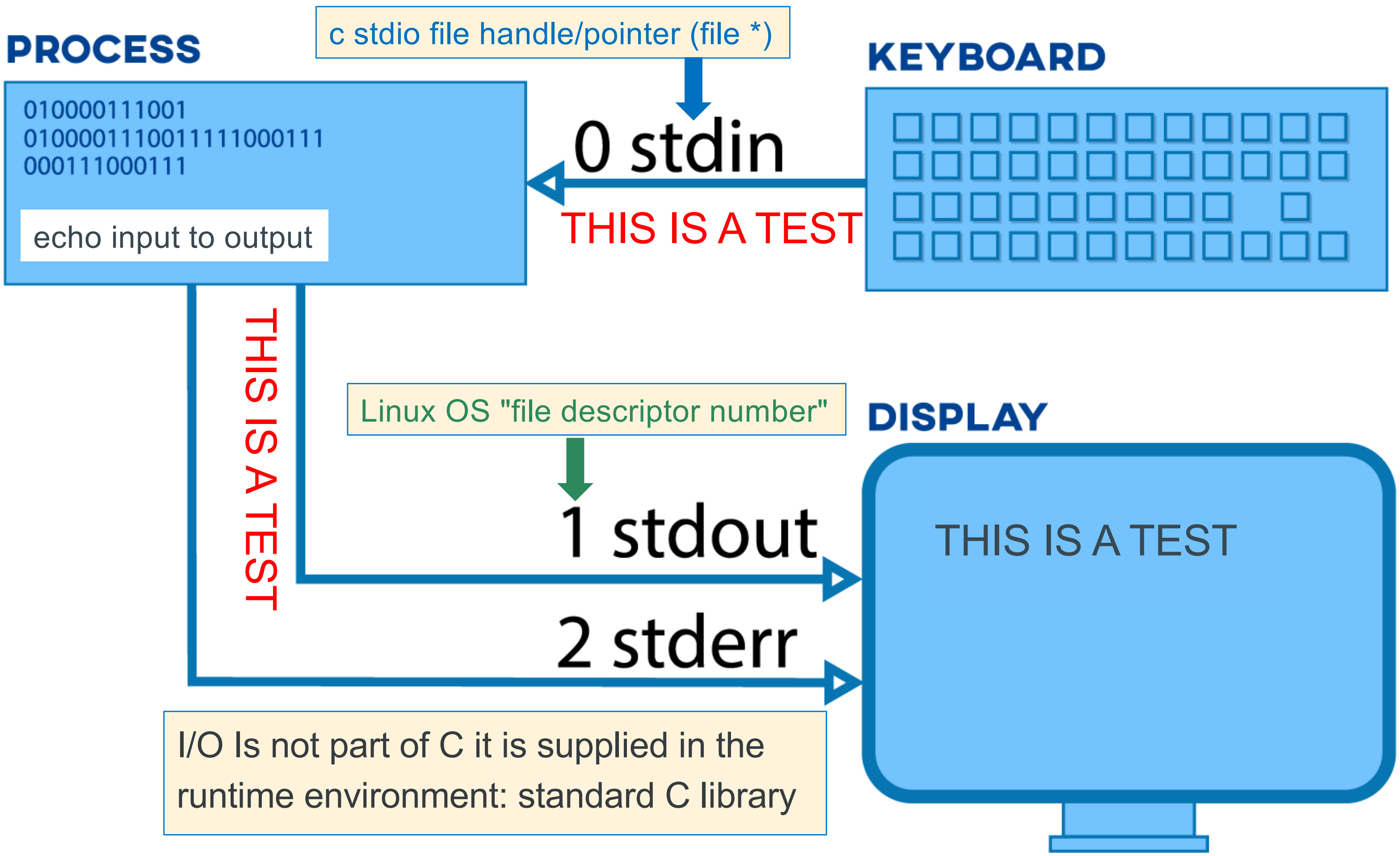

```
int main(void)
{
    int c;                    // input char
    int state = START;        // initial state of DFA

    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

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



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 */`

```
#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

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

- **C conditional test expressions**: **0 (NULL) is FALSE**, **<u>any</u> 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

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
- Example: 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

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

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 $2^4 = 16^1$

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

$$\boxed{0x6a3f}$$

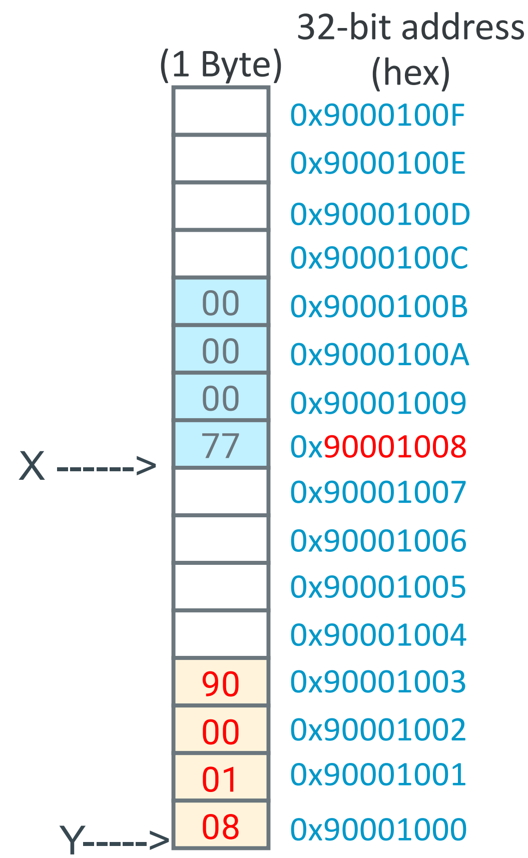
hex start with 0x in C

x

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

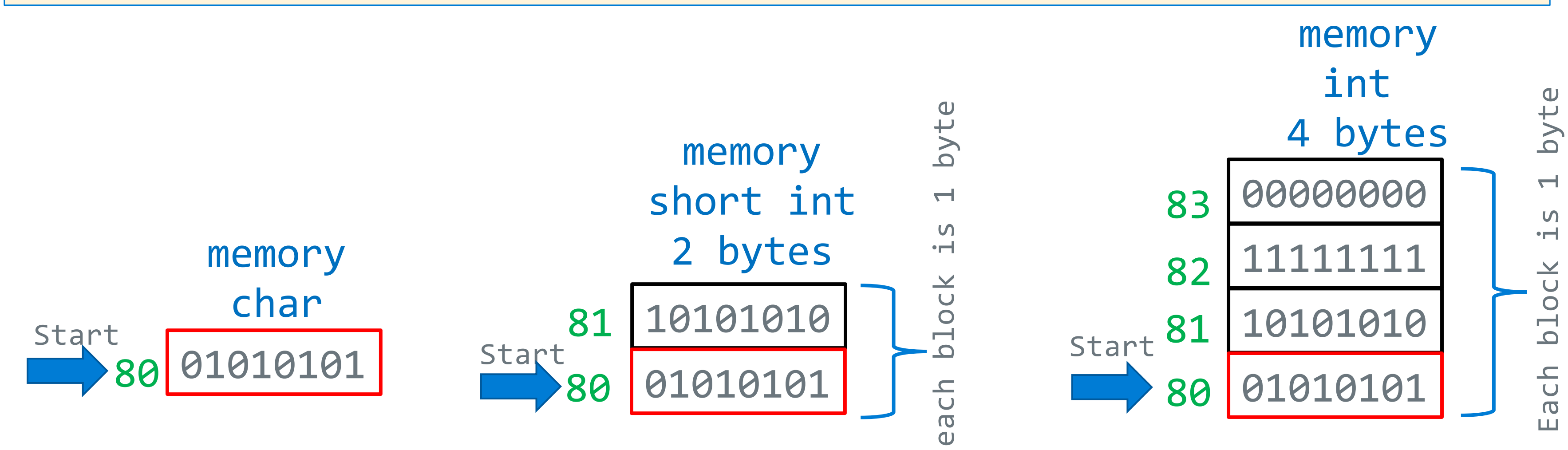
    x = x + y
- The **name** of a variable is on the **right side** of the = evaluates to a **memory address**
- The **name** of a variable is on the **left side** of the = evaluates to the **contents of memory at that address**



x

Variables in 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



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:

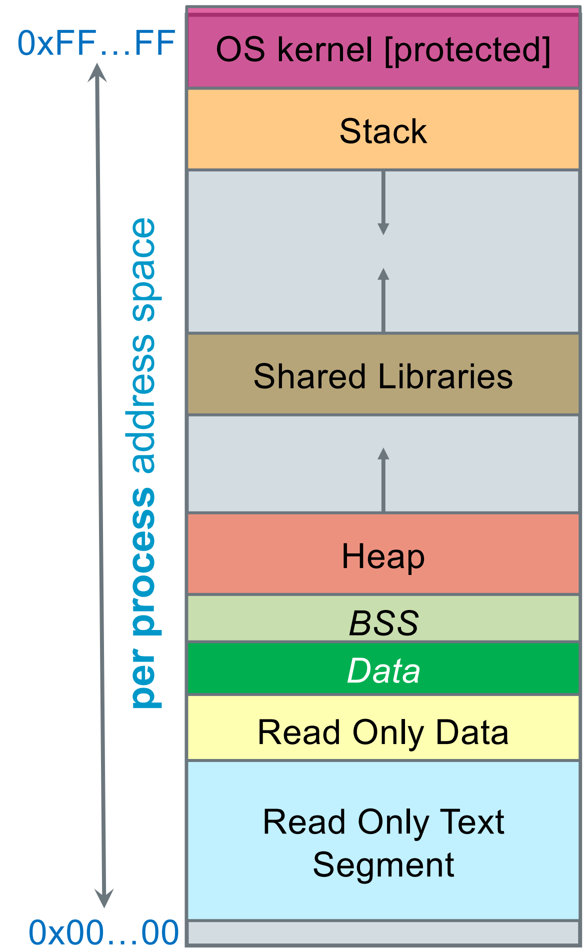
Signed Data types	Unsigned Data types	Exact Size
int8_t	uint8_t	8 bits (1 byte)
int16_t	uint16_t	16 bits (2 bytes)
int32_t	uint32_t	32 bits (4 bytes)
int64_t	uint64_t	64 bits (8 bytes)

x

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

Reference Slides

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

x

C vs Java: Expression Type Promotions, Demotions, Casts

- Java: demotions are <u>not</u> 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

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
```

- 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 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:
      |     ^~~~~~~
```

```
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 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 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)
               ~~~~^
```

```
% gcc -c -Wall -Wextra 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

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

C Versus Java

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

x

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;`
		`};`

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 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 hello.c -o hello
Running (execution)	% java Hello hello world!	% ./hello hello world!

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**
    - upper case S, not normally used
    - Compiles to assembly text file and stops
    - creates a **.s** file

- Producing an executable file
  - **gcc –Wall –Wextra mysrc.c**
    - creates an executable file ***a.out***
- To use a specific version of C use of one the std= option
  - **gcc –Wall –Wextra –std=c11 mysrc.c**
- Producing an object file with gdb debug support add -ggdb
  - **gcc –Wall -Wextra –c –ggdb mysrc.c**
    - creates an object file **mysrc.o**
  - **gcc –Wall –Wextra –c –ggdb mymain.c**
    - creates an object file **mymain.o**
- Linkage step
  - combining a program spread across multiple files
  - **gcc –Wall -Wextra –o myprog mymain.o mysrc.o**
    - creates executable file **myprog**
- Compile and linkage of file(s) in one step
  - **gcc –Wall –Wextra -o myprog mysrc.c mymain.c**
- run the program (refer to cse15l notes)
  - **% ./myprog**

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
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   -c -o noq.o noq.c
gcc -I. -ggdb -Wall -Wextra   -c -o states.o states.c
gcc -I. -ggdb -Wall -Wextra   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