Error handling:

1) Errno

2) Perror & Strerror

→ Print an error as a human-readable string

6/28

//GCC for hw assign

Character Encoding ASCII:

- 8 bit = byte
- ASCII is 7 bits value 0 to 127.
- 128 - 255 are not ASCII ← Don't try to print these!
- 0 to 31 Non-Printable control characters
- PRINTABLE CHARS 32 to 127
- All digits GROUPED TOGETHER
- Encoding digits < Uppercase < Lowercase

* Digit char - '0' = Arithmetic value of digit char

- Control Chars = char constants in C

C HEX				
0x00	- 00	'\0'	NUL	
0x07	- 07	'\a'	BELL	
0x08	- 08	'\b'	Backspace	
0x09	- 09	'\t'	Tab	
0x0A	- 0A	'\n'	NewLine	(00001010) BITS FOR NEWLINE
0x0B	- 0B	'\v'	Vertical Tab	
0x0C	- 0C	'\f'	form feed	
	- 0D	'\r'	Carriage Return	···· Start from most left again
	- 1B	'\e'	Escape	

- Some math

- Binary base 2
  - → Power of 2
  - → Available digits 0, 1

- Base 10
  - ↳ Available digits: 0 ~ 9

Important! ☆ Hexadecimal Base 16

ex) $256_{10} = 1 \times 16^2 + 0 \times 16^1 + 0 \times 16^0 = 100_{16}$

☆

DEC	Binary	HEX	DEC	Binary	HEX
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

↳ HEX $\quad$ 0 $\quad$ A

Binary $\underbrace{0000}\underbrace{1010}$ $\qquad$ 4bits = 1 HEX Digit

Char	HEX
'0'	0x30
'1'	0x31
'2'	0x32
⋮	⋮
'9'	0x39

HEX		DEC
☆ 0x39	!=	39

↳ How C communicates for HEX number.

- Octal Base 8
  - Available digits 0...7
  - Group of 3 bits

ex)

octal	HEX	DEC
33	1B	27

VAL = 033; ← octal 33

27

24

16+8+2+1

→ ~~0~~ $\underbrace{0 0 1 1}\underbrace{1 0 1 1}$

$$8 \overline{)27} \quad 3 \qquad \frac{-24}{3}$$

☆ • 32 bit number

31	32	33	34

octal	0	6	1	1	4	4	3	1	4	6	4

assume 0 $\quad$ 0011 $\quad$ 0001 $\quad$ 0011 $\quad$ 0010 $\quad$ 0011 $\quad$ 0011 $\quad$ 0011 $\quad$ 0100

HEX	3	1	3	2	3	3	3	4

- Fixed Point Integer Representation

Positive numbers	32 bit HEX
0	00000000
1	00000001
2	00000002

☆ Need to represent positive, negative, and zero values.

option 1. signed magnitude

- Integer has two parts
- Sign bit (0 = positive, 1 = negative)
- Magnitude 31 bits

sign bit

ex) $56_{10} = 38_{16} =$ 0011 1000 → 0000 0000 0000 0000 0000 0000 0011 1000

$-56_{10} = B8_{16} =$ 1011 1000

0 = 00_{16} = 0000 0000 Binary

-0 = 80_{16} = 1000 0000 Binary

Problem: · two zeros (+0 & -0)

· complicated hardware

Option #2. One's Complement (-127 ... +127)

- Positive numbers represented as before
- Negative numbers are ⊕ numbers with all bits flipped

ex) $56_{10} = 38_{16} =$ 0011 1000

↕ flipped bits

$-56_{10} = C7_{16} =$ 1100 0111

0 = 00_{16} = 0000 0000

-0 = FF_{16} = 1111 1111

Problem: Same problem as option #1

Compare

Option #3: Two's Complement (-128 ... +127)

- ⊕ Int. as before
- ⊖ Int. take ⊕ Int. → flip bits → Add 1
  → Discard carry

ex) $56_{10} = 38_{16} =$ 0011 1000

1100 0111 FLIP

$-56_{10} = C8_{16} =$ 1100 1000 Add 1

☆ ⊕ Int. → FLIP & Add 1 → ⊖ Int
→ FLIP & Add 1 → ⊕ Int.

0 = 00000000
11111111 FLIP BITS
1 0000 0000
← Discard carry

* 1 bit 0, 1
2 bits 00, 01, 10, 11
⇒ can represent 2^n distinct values in n bits

- -128_{10} = 10000000
01111111 FLIP * -128_{10} does not change!
-128_{10} = 10000000 Add 1

☆ LEFT & RIGHT Shifting

△ Only for fixed point Int. (ex) long, char)

Left Shift

- Operands are integer value, number of bits

ex) Before
| 00 | 00 | 00 | 01 |

After
| 00 | 00 | 00 | 02 | Left Shift ← 1 bit

Before
| 00 | 00 | 00 | 01 |

After
high order bits → | 00 | 00 | 00 | 04 | ← Low order bits
Left Shift ← 2 bits

ex) Before: 1000 0000 0000 0000 0000 0000 0000 0001
After: * 0000 0000 0000 0000 0000 0000 0000 0010 ← Left Shift 1 bit
↑ High order bits ↑ Low order bits

In C,

```
int A,B;
A=1;
B= A <<1;
B <<=1;
```

← Integer of bits, can be a variable or expression
← left shift operator

- can shift char, short, int, long
- Does NOT work for float, double

ex) '3' << 2
'3' = 0x33
$\underbrace{00}_{\text{Discard}}\overbrace{1100}^{c}\overbrace{1100}^{c}$

ex) 1 = 0000 0001
} FLIP
FFFF FFFE ← Look at HEX Table
↓ Add 1
−1 = FFFF FFFF

☆ Right Shift:

- Operands are int. value, number of bits
- can shift char, short, int, long
- Does not work for float, double

Two types
- Logical: Low Order bits are lost, 0 bits added at high order bits
- Arithmetic: Low order bits are lost, high order sign bit is added at high order bits.

{ Logical right Shift: 0 → [ ] → Discarded
{ Arithmetic " " : sign bit ↻ [ ] → Discarded

☆ very Important!

//shifts everything & replace sign bit with zero.

ex) Logical Right Shift
- Before: | 01 | 01 | B6 | A2 |
- After: | 00 | 10 | 1B | 6A | Logical Right shift 4bits

//preserves the sign bit + shift every thing right

ex) Arithmetic Right Shift
- Before: | B4 | A2 | C3 | E1 |
- After: | FB | 4A | 2C | 3E | ARS 4 bits

| 07 | B4 | C2 | E3 |
| 00 | 7B | 4C | 2E | ARS 4bits

10111000
① 110111000|0
② 1110111|000
③ 1111,0111|000
F
④ 11111011|1000
F B 4

C Programming Structure:

FFFFFFFF Smallest negative
⋮
80000000 Largest Negative
* 7FFFFFFF Largest Positive
1
0 zero
⋮
FFFF FFFF Smallest negative

⇒ Overflow

Source file

Header file

```
Main.C
#include "factorial.H" //for factorial()
#include <STDIO.H> // for printf()
```

preprocessor directives

```
Int
main()
{
    int x, y;
    X = 3;
    y = factorial(x);
    printf("The factorial of %d is %d\n", x, y);
    return 0;
}
```

C= Compiled language

```
Factorial.H
 #IFNDEF Factorial_H
 #DEFINE factorial_H
//Declare all types HERE!
 Int                 ex) Enum, struct, etc...
 Factorial(Int x);
#ENDIF
```

☆ Multiple Inclusion Protection

function declaration

```
Factorial.C
#INCLUDE factorial.H
Int
factorial(Int x)
{
    if (x <= 0) return 1;
    else return X* factorial(x-1);
}
```

functions are global! (unless static)

function definition

* Cannot define function more than once

* Can declare header files at multiple source files

on iLAB Linux Machine

```
gcc main.c factorial.c - - - - - - A.out
gcc -o factorial main.c factorial.c - - - factorial
gcc -WALL -o factorial main.c factorial.c
```

← executable program

Warn all
↑
show what the comp. is trying to do