Tutorials 2-3:
More C, Computer Arithmetic

- More C operators
- More C commands
- Command line arguments (argc, argv[])
- Assignment #1 Information
- Converting between bases
- Long division
- IEEE 754
- Max/Min numbers in representations

Bitwise C Operators

&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise NOT (ones complement)
<<	shift left
>>	shift right

If Statement

```
if (<expression> )
{
  <statements_1>
}
else
{
  <statements_2>
}
```

- if <expression> is true, will execute <statements_1>, otherwise will execute <statements_2>.

- else portion is optional

While Statement

```
while ( <expression> )
{
  <statements>
}
```

- will execute <statements> while <expression> is true (or not zero).

Example:

```
  x = 0;
  while ( x < 10)
  {
     <statements>
     ++x;
  }
```

For Statement

```
for ( <start>; <expression>; <action> )
{
  <statements>
}
```

- will do <start>, then as long as <expression> is true, will repeatedly execute <statements> and <action>

Example:

```
  for (x = 0; x < 10; ++x)
  {
     <statements>
  }
```

Command Line Arguments

e.g. "echo hello, world"
output: "hello, world"

Arguments are passed using:

- **`argc`** number of command-line arguments (including the program name)
- **`argv[]`** array of strings, one for each argument (in other words, an array of char arrays)

echo.c

```
main(int argc, char *argv[])
```

argc: 3

argv[0]: ⟶ "echo"

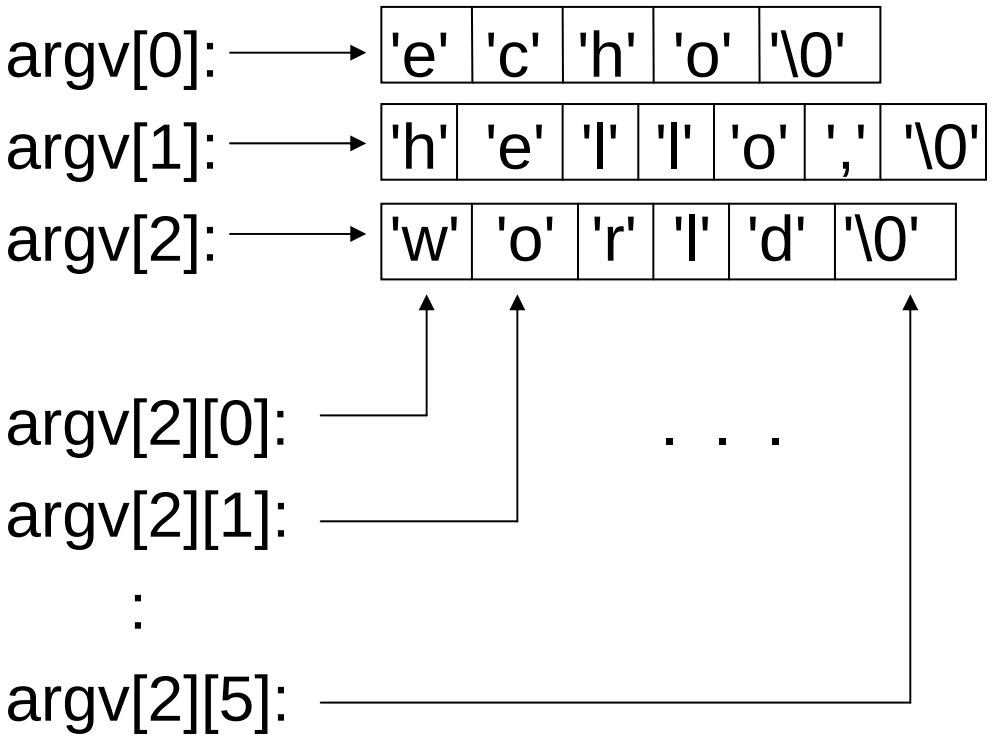
argv[1]: ⟶ "hello,"

argv[2]: ⟶ "world"

```
for (i = 1; i < argc; ++i)
  printf("%s ",argv[i]);
```

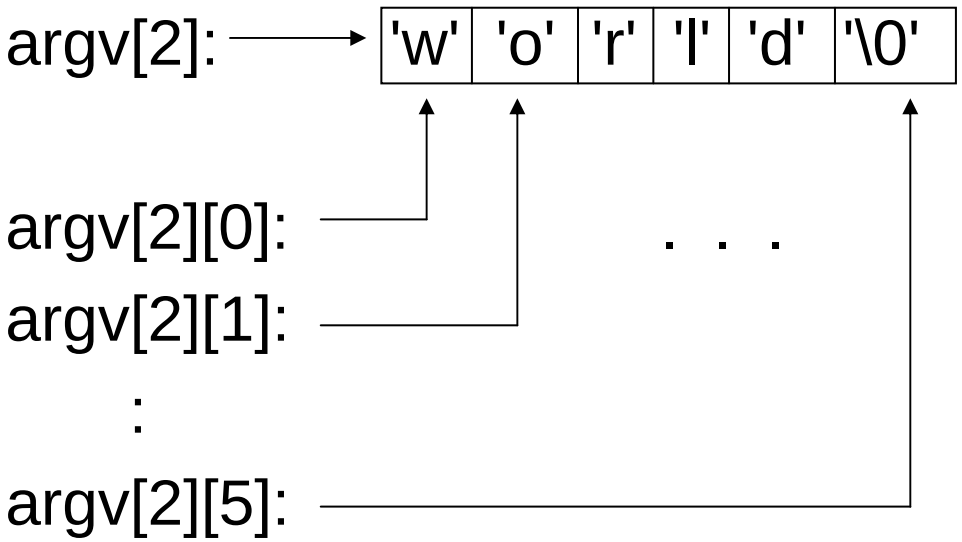
echo.c

```
main(int argc, char *argv[])
```



echo.c

```
i = 0;
while (argv[2][i] != '\0')
  printf("%c",argv[2][i++]);
```



Non-character arguments

```
mult  13  15
```

argc: 3

argv[0]: ⟶ "mult"

argv[1]: ⟶ "13"

argv[2]: ⟶ "15"

```
int m1, m2;
sscanf(argv[1], "%d", &m1);
sscanf(argv[2], "%d", &m2);
```

Fun with pointers

"C" doesn't check type of pointers

So, we can trick it:

```
float num = -12.34e17;
int *ptr;

ptr = &num;
printf("%lx ", *ptr);
```

Result: dd890064

Decimal vs. Hexadecimal

Decimal:	3	0	1	.	9	6
	Hundreds	Tens	Ones		Tenths	Hundredths
Hexadecimal:	1	2	D	.	F	7
	256's	Sixteens	Ones		Sixteenths	1/256's

Hexadecimal numbers

base10	base2	base16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

base10	base2	base16
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting between bases

$12D.F7_{16}$: convert to base2

1	**2**	**D**	**.**	**F**	**7**
0001	**0010**	**1101**	**.**	**1111**	**0111**

000100101101.11110111_2

Converting between bases

$12D.F7_{16}$: convert to base10

First, do integer part:

$1 * 16^2 + 2 * 16^1 + D * 16^0$

$1 * 256 + 2 * 16 + 13 * 1$

$256 + 32 + 13$

301

Converting between bases

$12D.F7_{16}$: convert to base10

How many digits do we need after decimal?

$$\#digits_{10} = \#digits_{16} * \frac{\log(16)}{\log(10)}$$

$$= 2 * \frac{\log(16)}{\log(10)}$$

$$= 2.4 \qquad \text{(so need 3 digits)}$$

Converting between bases

$12D.F7_{16}$: convert to base10

Now do fractional part:

$F * 16^{-1} + 7 * 16^{-2}$

$15 * 1/16 + 7 * 1/256$

$15 * 0.0625 + 7 * 0.00390625$

0.96484375

Only need 3 digits: 0.965

Converting between bases

$12D.F7_{16}$: convert to base10

Putting it all together:

301.965

Converting between bases

$12D.F7_{16}$: convert to base9

How many digits do we need after decimal?

$$\#digits_9 = \#digits_{16} * \frac{\log(16)}{\log(9)}$$

$$= 2 * \frac{\log(16)}{\log(9)}$$

$$= 2.5 \qquad \text{(so need 3 digits)}$$

Converting between bases

$12D.F7_{16}$: convert to base9

Would have to do math in two bases!

Easier just to use the base10 number: 301.965

Converting between bases

301.965_{10}: convert to base9

9^3	729	301.965	0	0	301.965
9^2	81	301.965	3	243	58.965
9^1	9	58.965	6	54	4.965
9^0	1	4.965	4	4	0.965
9^{-1}	0.11111	0.965	8	0.88888	0.07612
9^{-2}	0.01234	0.07612	6	0.07404	0.00208
9^{-3}	0.00137	0.00208	1	0.00137	0.00071
9^{-4}	0.00015	0.00071	4	0.00060	0.00011

Converting between bases

301.965_{10}: convert to base9

Result: 364.8614_9

Rounding to 3 places: 364.861_9

Problem with technique – can have errors accumulate. So, better to do repeated division / multiplication of goal base

Converting between bases

$\mathbf{301.965_{10}}$: convert to base9

Start with integer part:

301 / 9 = 33 remainder 4

33 / 9 = 3 remainder 6

3 / 9 = 0 remainder 3

Taking remainders and reversing: 364

Converting between bases

$\mathbf{301.965_{10}}$: convert to base9

Now do fractional part:

.965 * 9 = 8 . 685

.685 * 9 = 6 . 165

.165 * 9 = 1 . 485

.485 * 9 = 4 . 365

So, fractional part is: 0.8614

Rounded to 3 places: 0.861

Converting between bases

301.965_{10}: convert to base9

Putting it all together:

364.861_{9}

Base 10 Division

Compute 367 / 24 to 1 decimal place

```
          1 5. 2 9
       ------------
2 4  ) 3 6 7. 0 0
       2 4
       -----
       1 2 7
       1 2 0
       -----
           7  0
           4  8
           ----
           2  2  0
```

Base 2 Division

- Compute 1110 / 101 to 2 binary places

```
          1 0 .1 1 0
      ---------------
1 0 1 ) 1 1 1 0 .0 0 0
        1 0 1
        -----
          1 0 0
              0
          -----
          1 0 0 0
            1 0 1
          -------
            0 1 1 0
              1 0 1
            -------
                1 0
```

IEEE 754

Will demonstrate the following:

- Converting from IEEE 754 to decimal
- Converting from decimal to IEEE 754

Fun with pointers

"C" doesn't check type of pointers

So, we can trick it:

```
float num = -12.34e17;
int *ptr;

ptr = &num;
printf("%lx ", *ptr);
```

Result: dd890064

IEEE 754

Converting **dd890064** to decimal:

- First, convert to binary.

D D 8 9 0 0 6 4

1101 1101 1000 1001 0000 0000 1010 0100

- Now, regroup the digits: 1 sign, 8 exponent digits, 23 mantissa digits:

1 10111011 00010010000000010100100

sign exponent mantissa

1 10111011 00010010000000010100100

sign exponent mantissa

Sign = 1, so it is a negative number

Exponent = 10111011 =

= 128+32+16+8+2+1 = 187

- but exponents are all offset by 127

- so actual exponent is 187 – 127 = 60

1 10111011 00010010000000010100100

sign exponent mantissa

Mantissa = 00010010000000010100100

- these are the bits after the binary point
- but remember that it does not include the leading 1
- so the actual mantissa is:

  1. 00010010000000010100100

0 10000100 01010010001001011100

sign exponent mantissa

So, we know that this is the number:

$-1.00010010000000010100100_2 * 2^{60}$

Let's convert the mantissa to base-10:

$1.00010010000000010100100_2$

$= 1 + (1/2^4) + (1/2^7) + (1/2^{16})$

$+ (1/2^{18}) + (1/2^{21})$

$= 1.07033205..._{10}$

Mantissa is: $1.07033205..._{10}$

But how many digits do we need?

Binary mantissa is 24 bits, so:

$$\#digits_{10} = 24 * \frac{\log(2)}{\log(10)}$$

$$= 7.225 \text{ (so need 8 digits)}$$

So, mantissa in base-10 is: 1.0703321

0 10000100 01010010001001001100

sign exponent mantissa

So, in base-10 the number is:

- $1.0703321 * 2^{60}$

Now, calculate it out:

$= -1.2340088 * 10^{18} = -12.34009 * 10^{17}$

That's our original number (with a little bit of error, since decimal fractions can't be represented exactly in binary)

IEEE 754

Converting $-12.34 * 10^{17}$ to IEEE 754:

- First, normalize so in the form $-1.XXX * 2^{X}$

$$\frac{\log(12.34 * 10^{17})}{\log(2)}$$

$$= 60.098 = 61$$

- So, exponent is 2^{61}

IEEE 754

Converting $-12.34 * 10^{17}$ to IEEE 754:

- Exponent is 2^{61}, so divide that out:

$$\frac{12.34 * 10^{17}}{2^{61}}$$

$$= 0.53516219$$

So, $-12.34 * 10^{17} = 0.53516219 * 2^{61}$

$$= 1.07032438 * 2^{60}$$

(since we need it in the form $1.XXX * 2^{X}$)

IEEE 754

Converting $-12.34 * 10^{17}$ to IEEE 754:

- How many digits do we need in base2?

$$\#digits_2 = 4 * \frac{\log(10)}{\log(2)}$$

$$= 13.28 \text{ (so need 14 digits)}$$

Converting $\mathbf{-12.34 * 10^{17}}$ to IEEE 754:

- convert mantissa (1.07032438) to binary
- Integer part is 1.
- Fractional part (13 more digits):

0.07032438 * 2 = 0.14064876
0.14064876 * 2 = 0.28129752
0.28129752 * 2 = 0.56259504
0.56259504 * 2 = 1.12519008
0.12519008 * 2 = 0.25038016
0.25038016 * 2 = 0.50076032
0.50076032 * 2 = 1.00152064
0.00152064 * 2 = 0.00304128
0.00304128 * 2 = 0.00608256
0.00608256 * 2 = 0.01216512
0.01216512 * 2 = 0.02433024
0.02433024 * 2 = 0.04866048
0.04866048 * 2 = 0.09732096
0.09732096 * 2 = 0.19464192 (for rounding, round down)

So, is 1.0001001000000 in binary

Now, have $-1.0001001000000 * 2^{60}$

Sign: 1 (since is negative)

Exponent: 60

- but must add offset of 127 to get 187
- 187 in binary is: 1011 1011

Mantissa: 1.0001001000000

- but don't need leading one
- do need 23 bits – so just pad on the end
- gives 00010010000000000000000

Putting it all together:

Sign: 1 (since is negative)
Exponent: 1011 1011
Mantissa: 00010010000000000000000

RESULT:

1 **10111011 00010010000000000000000**

Group in 4's, convert to hex:

1101 1101 1000 1001 0000 0000 0000 0000

D D 8 9 0 0 0 0

Converting **-12.34 * 10^{17}** to IEEE 754:

Final result:

`dd890000`

Why is it different from:

`dd890064` ?

Because second one is conversion from
12.340000 * 10^{17}

- We only carried enough digits to get **4** digits of accuracy

Max/Min – 4 digits

What's max, min for unsigned binary?

0000_2 to 1111_2 (0 to 15 decimal)

16 numbers

For Sign + Mantissa binary?

1111_2 to 0111_2 (or -7 to 7 decimal)

15 numbers (why?)

For Two's Complement binary?

1000_2 to 0111_2 (or -8 to 7 decimal)

16 numbers

Max/Min – 4 digits

What's max, min for unsigned base 5?

0000_5 to 4444_5 (0 to 624 decimal)

625 numbers (5^4)

For 5's complement base 5?

- 0 and 4 are complements, 1 and 3 are, 2 complements 2

  e.g. -1: -0001 -> 4443 + 1 = 4444

- where is crossover?
- largest positive number is 2222 (+312)
- largest negative number is 2223 (-312)
- so, 312 positive numbers, 312 negatives, one zero: 625

Assignment Information

Be careful when using logical operators; they have lower precedence than comparison operators.

```
if (num & 0x1 == 0) { ... }
```

This is actually evaluated in the following order (!):

```
if (num & (0x1 == 0)) { ... }
```

So, use brackets to fix it:

```
if ((num & 0x1) == 0) { ... }
```

Assignment Information

Some hints:

- make sure to show all your work for Q1-Q5.

- For Q6, make sure to test your program with appropriate test cases, and include an output file showing all these tests.