Week 11 Lecture 2

Little Things Count in Programming…


ORDERED PAIRS WITHOUT PARENTHESES?
Spaces, Parentheses, Precedence
NO. NO. (N,O).

Evaluating Expressions with Multiple Operators

- If there is more than one operators in an expression, C has a predefined rule of priority for the operators.
- This is called **operator precedence**.
- It is easy to remember the 3 levels of precedence:
  - **arithmetic** operators (*, %, /, +, -) are higher than
  - **relational** operators (==, !=, >, <, >=, <=) are higher than
  - **logical** operators (&&, || and !)

Category	Operator	Associativity
Postfix	( ) [ ] -> . ++ - -	Left to right
Unary	- + ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Category	Operator	Associativity
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

Examples

- What does the following do?

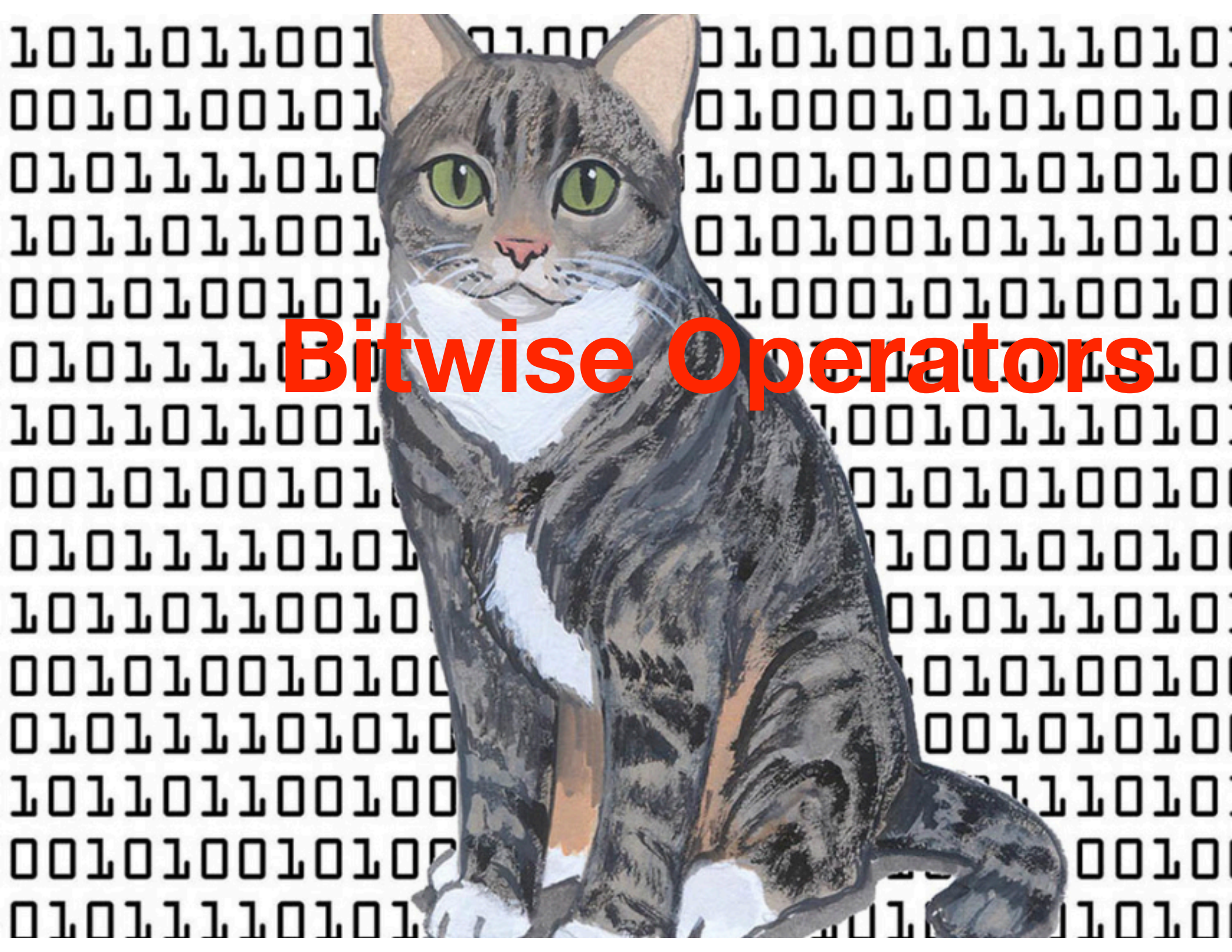
```
if (3 > 2 + 3 && 4 * 8 > 30) {

    printf ( "Expression - True\n" );

} else {

    printf ( "Expression - False\n" );

}
```

```
(4 > 2 + 3) && (4 * 8 > 30)
```

Arithmetic
Relational
Logical

```
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
printf("a = %d b = %d c = %d d = %d\n", a,b,c,d);
e = a + b * c / d;
printf("Value of a + b * c / d is : %d\n", e);

e = a + b * c / d + a;
printf("Value of a + b * c / d + a is : %d\n", e);

e = a + b * c + a / d;
printf("Value of a + b * c + a / d is : %d\n", e);
```

```
a = 20 b = 10 c = 15 d = 5
Value of a + b * c / d is : 50
Value of a + b * c / d + a is : 70
Value of a + b * c + a / d is : 174
```


Bitwise Operators

Bitwise Operators

- Bitwise operators work on bits and perform bit-by-bit operations.
- Truth tables for the bitwise operators.

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

https://www.tutorialspoint.com/cprogramming/c_operators.htm

Bitwise Examples

- Let us declare two **`short int`**'s and call them **`A`** and **`B`**.
- There binary and decimal formats would be the following:

A: 0000 0000 0011 1100 [60]

B: 0000 0000 0000 1101 [13]

0000 0000 0011 1100 = A [60]

0000 0000 0000 1101 = B [13]

0000 0000 0000 1100 = A & B [12]

Bitwise Examples

0000 0000 0011 1100 = A [60]

0000 0000 0000 1101 = B [13]

0000 0000 0011 1101 = A | B [61]

0000 0000 0011 1100 = A [60]

0000 0000 0000 1101 = B [13]

0000 0000 0011 0001 = A ^ B [49]

Bitwise/Binary Operators

- Binary **AND** (&) copies a bit to the result if it exists in both operands.
- Binary **OR** (|) copies a bit if it exists in either operand.
- Binary **XOR** (^) copies the bit if it is set in one operand but not both.
- Binary **One's Complement** is unary and works by '**flipping**' bits.
  - e.g. A = 0000 0000 0011 1100 so

    (~A) = 1111 1111 1100 0011

Binary Shift Operators

- Binary Left Shift (<<)
  - The left operands value is moved **left** by the number of bits specified by the right operand.
  - 0000 0000 0011 1100 << 2
  - 0000 0000 1111 0000
  - A << 2 = 240

Binary Shift Operators

- Binary Right Shift (>>)
  - The left operand's value is moved **right** by the number of bits specified by the right operand.
  - 0000 0000 0011 1100 >> 2
  - 0000 0000 0000 1111
  - A >> 2 = 15

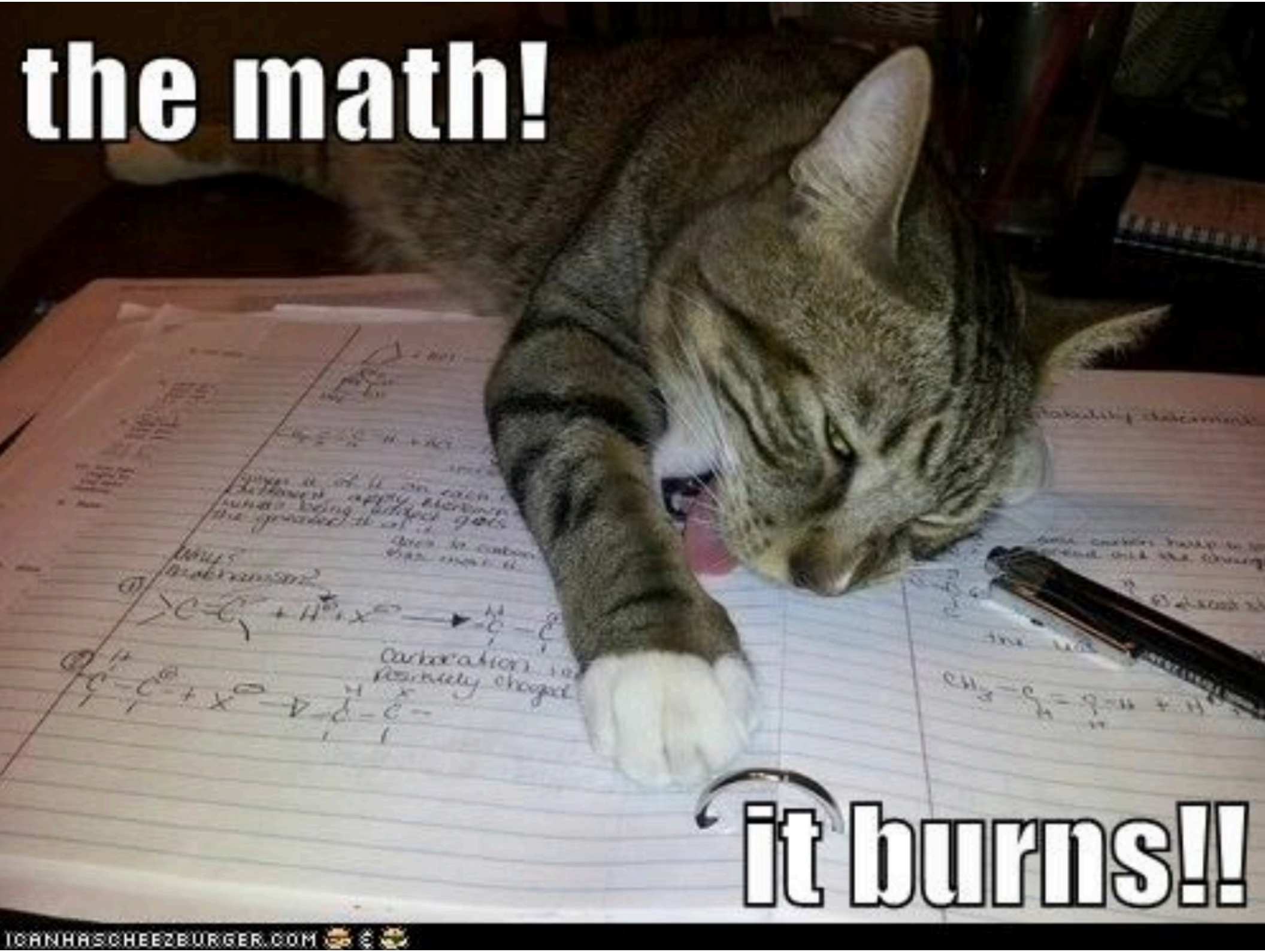
```
unsigned short int bits1 = 60;
/* 0000 0000 0011 1100 */

printf ("bits1 = %d and bits1 >> 2 = %d\n",
          bits1, bits1 >> 2);
bits1 = bits1 >> 2;

printf ("bits1 = %d and bits1 << 2 = %d\n",
          bits1, bits1 << 2);

bits1 = bits1 << 2;
printf ( "bits1 = %d and bits1 << 2 = %d\n",
           bits1, bits1 << 2);
```

```
$ ./binaryShift
bits1 = 60 and bits1 >> 2 = 15
bits1 = 15 and bits1 << 2 = 60
bits1 = 60 and bits1 << 2 = 240
```

```
int binary (unsigned short int binNum, char *binString) {
   unsigned short int mask = 32768;
                                        /* 1000 0000 0000 0000 */
   int i = 1;
   int j = 0;
   while ( mask > 0 ) {
      if ( (binNum&mask) > 0 ) {
         *(binString+j++) = '1';
      } else {
         *(binString+j++) = '0';
      }
      if ( i % 4 == 0 ) {
         *(binString+j++) = ' ';
      }
      i++;
      mask = mask >> 1;
   }
   *(binString+j) = '\0';
   return ( 0 );
}
```

```
unsigned short int bits1 = 60;    /* 0000 0000 0011 1100 */
unsigned short int bits2 = 128;   /* 0000 0000 1000 0000 */
unsigned short int bits3 = 32768; /* 1000 0000 0000 0000 */
char bString[20];

printf ( "bits1 = %d\n", bits1 );
binary ( bits1, bString );
printf ( "%s\n", bString );

printf ( "bits2 = %d\n", bits2 );
binary ( bits2, bString );
printf ( "%s\n", bString );

printf ( "bits3 = %d\n", bits3 );
binary ( bits3, bString );
printf ( "%s\n", bString );

printf ( "~bits1 = %u\n", (~bits1) );
bits1 = ~(bits1);
binary ( bits1, bString );
printf ( "%s\n", bString );
```

bits1 = 60
0000 0000 0011 1100
bits2 = 128
0000 0000 1000 0000
bits3 = 32768
1000 0000 0000 0000
~bits1 = 4294967235
1111 1111 1100 0011


the math!
it burns!!
ICANHASCHEEZBURGER.COM

```
unsigned short int bits1 = 60; /* 0000 0000 0011 1100 */
unsigned short int bits2 = 0;
char bString1[20], bString2[20];

binary ( bits1, bString1 );
bits2 = bits1;
bits2 = bits2 >> 1;
binary ( bits2, bString2 );
printf ("%d (%s) >> 1 = %3d (%s)\n", bits1, bString1,
                                     bits2, bString2);

bits2 = bits1;
bits2 = bits2 << 1;
binary ( bits2, bString2 );
printf ("%d (%s) << 1 = %3d (%s)\n", bits1, bString1,
                                     bits2, bString2);
```

60 (0000 0000 0011 1100) >> 1 = 30 (0000 0000 0001 1110)

60 (0000 0000 0011 1100) << 1 = 120 (0000 0000 0111 1000)

>>1 is equivalent to / 2

<<1 is equivalent to * 2

```
bits2 = bits1;
bits2 = bits2 >> 2;
binary ( bits2, bString2 );
printf ("%d (%s) >> 2 = %3d (%s)\n", bits1, bString1,
                                     bits2, bString2);

bits2 = bits1;
bits2 = bits2 << 2;
binary ( bits2, bString2 );
printf ( "%d (%s) << 2 = %3d (%s)\n", bits1, bString1,
                                      bits2, bString2);
```

60 (0000 0000 0011 1100) >> 2 = 15 (0000 0000 0000 1111)
60 (0000 0000 0011 1100) << 2 = 240 (0000 0000 1111 0000)

>>2 is equivalent to / 4
<<2 is equivalent to * 4

Conditional

A Ternary Operator

Conditional Operator

- The conditional operator returns one value if the condition is **true** and returns another value is the condition is **false**.

- It is also called as ternary operator since it has 3 parts.

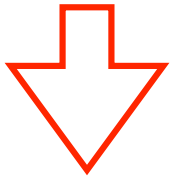
(Condition**?** true_value**:** false_value);

(A > 100 ? 0 : 1);

  - If A is greater than 100 (**true**) return 0, otherwise (**false**) return 1.

Modifying the `binary()` function

```
if ( (binNum&mask) > 0 ) {
        *(binString+j++) = '1';
} else {
   *(binString+j++) = '0';
}
```



```
*(binString+j++) =
        ( (binNum&mask) > 0 ? '1': ‘0' );
```

Static Variables

A Problem….

- You are writing a function and you want to know how many times the function has been called - sounds easy but you want to know this information from **inside** the function.

```
int countMe ( ) {
  int count = 0;
  count++;
  return ( count );
}
```

- Will this work?

```
#include <stdio.h>
#include <stdlib.h>
int countMe ( );

int main ( int argc, char *argv[] ) {
   int counter = 0;
   int ret = 0;
   int max = atoi ( argv[1] );

   for ( counter=0; counter<max; counter++ ) {
      ret = countMe( );
   }
   printf ("countMe() was called %d times.\n", ret);

   return ( 0 );
}
```

```
$ ./saveMe 100
countMe() was called 1 times.
```

Static Variables

- Our function did not work because variables inside a function only have scope in that function and are destroyed when you leave the function.
- They are stored on something called the stack and not in the main's memory (data segment).
- But there is a type of variable in a function that is not stored on the stack - the `static` variable.

```
static int count = 0;
```

```
int countMe ( ) {
  static int count = 0;
  count++;
  return ( count );
}
```

$ ./saveMe 100
countMe() was called 100 times.

$ ./saveMe 10
countMe() was called 10 times.

$ ./saveMe 4000
countMe() was called 4000 times.


Randomness
It makes everything better…

Adding Spice to Life…

- In Conway's Game of Life (Assignment 4), some of the seeds lead to games that are static or oscillating.
- Boring! Nothing will every change…but…
- Could you give it a kick?
- The solution is … randomness!

`random()`

- The **`random()`** function uses a nonlinear additive feedback random number generator employing a default table of size 31 long integers to return successive **pseudo-random numbers** in the range from 0 to **`RAND_MAX`**.

- The period of this random number generator is very large, approximately 16 $((2^{31}) - 1)$.

- The **`srandom()`** function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by **`random()`**.

- These sequences are repeatable by calling **`srandom()`** with the same seed value. If no seed value is provided, the **`random()`** function is automatically seeded with a value of 1.

```
#include <stdio.h>
#include <stdlib.h>
int main ( int argc, char *argv[] ) {
   unsigned int seed;
   int count = 0;
   seed = (unsigned int) atoi ( argv[1] );

   srandom(seed);
   printf ("random() generates numbers from 0 to %d\n”,
                                                RAND_MAX );
   for ( count=0; count<10; count++ ) {
      printf ( "%ld\n", random() );
   }
   printf ("Let's generate numbers from 0 to 100: \n");
   for ( count=0; count<10; count++ ) {
      printf (“%d\n",(int)((float)random()RAND_MAX*100));
   }
   return ( 0 );
}
```

```
$ ./randomWalk 123
random() generates numbers from 0 to 2147483647
128959393
1692901013
436085873
748533630
776550279
289134331
807385195
556889022
95168426
1888844001
Let's generate numbers from 0 to 100:
63
37
31
82
42
48
79
95
71
35
```

```
$ ./randomWalk 12390
random() generates numbers from 0 to 2147483647
1719826980
2111644270
604018000
1313229968
1261614062
2117208981
1106270491
680629438
133059595
162042964
Let's generate numbers from 0 to 100:
28
50
76
2
65
1
30
3
21
14
```