LECTURE 11: MORE BIT MANIPULATION

BIT MASKING

What if we want to turn the bits in specific positions in a binary number on or off?
This process is called bit masking.

FORCING GROUPS OF BITS OFF

Example: (using &) Given char n = '\xa5', Turn off all bits except the least significant 5		
Recall that & will compare each bit position and the bit in that position in the result will only be on if the bits in the operands were both on.		
This means we want a binary number with 1s in all the bit positions we want to keep as is, and 0s in the positions we want to turn off.		
In this case, we want the least significant 5 bits as is: 00011111, or '\x1f'		
n & '\x1f'	10100101 & 00011111 00000101	
Result: 00000101	The 3 most significant bits are off: 000	The 5 least significant bits match n: 00101
The key is that the values of the original number's bit positions are only preserved where there are 1s in the comparison number's bit positions.		

Example: (using & and ~) Given x (integer of unknown length) Turn off least significant 6 bits	
This time, we want the least significant 6 bits turned off, so we want 0s in those bit positions and 1s elsewhere.	
We can use ~077 (recall leading 0 indicates octal)	
Even if we don't know the size of x (the size of the int): ~077 == ~00 [...] 00111111 == 11 [...] 11000000	
x = x & ~077	Will turn off the least significant 6 bits and preserve all other bits.

FORCING GROUPS OF BITS ON

Example: (using |) Given char n = '\xa5' Turn on most significant 2 bits		
We want the most significant 2 bits turned n, which means leaving them on if they are already on.		
We can use OR for this.	1s in comparison number	These positions will always be ON: anything | 1 == 1
	0s in comparison number	These positions will remain as in original: anything | 0 == anything
In this case, we want the most significant 2 bits turned on, so we want 1s in those positions: 11000000, or '\xc0'		
n | '\xc0'	10100101 | 11000000 11100101	
Result: 11100101	The 2 most significant bits are on: 11	The 6 least significant bits match n: 100101

"ENCRYPTION" WITH EXCLUSIVE OR

Show that x ^ (x ^ y) == y	
char y = '\xa5'	10100101 (message)
char x = '\x69'	01101001 (encryption key)
x ^ y	11001100 (cipher text)
x ^ (x ^ y)	10100101 (decrypted message)

LECTURE 11: MORE STRING/ENUM, CONST DECLARATIONS (K&R, §§ 2.3, 2.4)

STRING CONSTANTS

"I am a string."	
Review:	An array (a pointer to a string) of char values, ending with NUL = '\0'.
	"0" is not the same as '0'. The value "0" can't be used in an expression, only as the argument to functions like printf().

There are various predefined functions for manipulating strings
#include <string.h>
See K&R Appendix B3 (page 249).
With these definitions, you can use:
len = strlen(msg);
(where msg is a string in a string array)

CONST QUALIFIER

Warns compiler that a variable's value should not change.
const char msg[] = "Warning: ...";
Commonly used in function arguments.
int copy(char to[], const char from[]);
If the copy function attempts to modify the "from" string, the compiler will give a warning.
(Remember that array names are pointers, so array arguments allow you to manipulate the original.)
The exact form of the warning and behavior of the code is implementation-defined.

ENUMERATION SYMBOLIC CONSTANTS

Shorthand for creating symbolic constants.	
Can be used instead of #define statements.	
Storage requirement is the same as int.	
Example:	
enum boolean {FALSE, TRUE};	Enumerated names are assigned values starting from 0: FALSE = 0 TRUE = 1
enum boolean x; x = FALSE;	Now you can declare a variable of type: enum boolean

Example:

```
int main() {
  enum month {ERR, JAN, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC};
  month this_month;
  this_month = FEB;

  [...]
}
```

LECTURE 11: ARITHMETIC/RELATIONAL/LOGICAL OPERATORS, PRECEDENCE (K&R §§ 2.5, 2.6, 2.12)

ARITHMETIC OPERATORS

Binary Arithmetic Operators (take 2 operands)		
+	addition	
-	subtraction	
*	multiplication	
/	division Note: Integer division truncates any fractional part.	
%	modulo (remainder after division) e.g., 5 % 7 == 5 7 % 7 == 0 10 % 7 == 3	% cannot be applied to float or double.

implementation-defined behavior with negative values	
direction of truncation with /	on our machines: -3 / 2 == -1
sign of result with %	on our machines: -3 % 2 == -1

LOGICAL OPERATORS

Apply logic functions to boolean arguments (arguments that evaluate to true or false).

Recall that in C:	0 is false nonzero is true

Evaluated left-to-right.
Evaluation stops as soon as truth or falsehood is known.

<table>
<tr><td>not</td><td>!x</td><td>converts a nonzero operand into 0
zero operand into 1</td></tr>
<tr><td>and</td><td>x && y && ... && z</td><td>1 if all operands are true,
0 otherwise</td></tr>
<tr><td>or</td><td>x || y || ... || z</td><td>1 if any operand is true,
0 otherwise</td></tr>
</table>

RELATIONAL OPERATORS

relation	a comparison between two arithmetic expressions

Relational operators are used to check the relationship between the values of their operands.

Defined by specification to always evaluate to 1 (true) or 0 (false).

	condition required to evaluate to 1 (true)
x == y	the values of x and y are equal
x != y	the values of x and y are not equal
x > y	x is greater than y
x < y	x is less than y
x >= y	x is greater than or equal to y
x <= y	x is less than or equal to y

OPERATOR PRECEDENCE

<table>
<tr><th></th><th>Operators</th><th>Associativity</th></tr>
<tr><td>first</td><td>() [] -> .</td><td>left to right</td></tr>
<tr><td></td><td>! ~ ++ -- + - * (type) sizeof</td><td>right to left</td></tr>
<tr><td></td><td>* / %</td><td>left to right</td></tr>
<tr><td></td><td>+ -</td><td>left to right</td></tr>
<tr><td></td><td><< >></td><td>left to right</td></tr>
<tr><td></td><td>< <= > >=</td><td>left to right</td></tr>
<tr><td></td><td>== !=</td><td>left to right</td></tr>
<tr><td></td><td>&</td><td>left to right</td></tr>
<tr><td></td><td>^</td><td>left to right</td></tr>
<tr><td></td><td>|</td><td>left to right</td></tr>
<tr><td></td><td>&&</td><td>left to right</td></tr>
<tr><td></td><td>||</td><td>left to right</td></tr>
<tr><td></td><td>?:</td><td>right to left</td></tr>
<tr><td></td><td>= += -= *= /= %= &= ^= |= <<= >>=</td><td>right to left</td></tr>
<tr><td>last</td><td>,</td><td>left to right</td></tr>
</table>

ASSOCIATIVITY	determines order if the operators have the same precedence, e.g.:
x = y += z -= 4 x = y += (z -= 4) x = (y += (z -= 4))	go right to left per above table

PRECEDENCE EXAMPLE

```
[...]
int year = 2016;
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
  printf( "%d is a leap year\n", year);
else
  printf( "%d is not a leap year\n", year);
```

((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)

Highest precedence? Parentheses.
((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)

(year % 4 == 0 && year % 100 != 0)

Highest precedence? Modulo.
(year % 4 == 0 && year % 100 != 0)

(0 == 0 && 16 != 0)

Highest precedence? == and != have the same precedence.
(0 == 0 && 16 != 0)

(1 && 1) evaluates to 1

We go back to original expression with the evaluated value of the expression within parentheses.

(1 || year % 400 == 0)

Highest precedence? Modulo.
(1 || year % 400 == 0)

But, remember that evaluation stop as soon as truth or falsehood is know. 1 || [any value] is true.

The expression evaluates to true (1) and the output is
2016 is a leap year