CS 449 - Data Representation

Numerals

A *numeral* is a symbolic representation of a number. For the purposes of this class, we will define a numeral as a sequence of digits (symbols).

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

Number Bases

If we have an n-digit numeral $d_{n-1}d_{n-2} \ldots d_0$ in base b, then the value of that numeral is $\sum_{i=0}^{n-1} \boldsymbol{d_i\, b^i}$, which is just fancy notation to say that instead of a 10's or 100's place we have a b's or b^2's place.

The most common bases we will use in this class are 2, 10, and 16, which are called binary, decimal, and hexadecimal (or hex), respectively. In base b, each digit d_i can only be one of b fixed symbols (0-1 for binary, 0-9 for decimal, etc.).

The table on the right shows the equivalent numerals for the numbers 0 through 15 in these three major number bases. We differentiate between these bases by using the prefix '`0b`' for binary and '`0x`' for hexadecimal.

<u>Exercises</u>:

1. Complete the table below by converting the numbers into the other two common bases. You may leave the "Decimal" column unsimplified.

Binary	Decimal	Hexadecimal
0b10010011		
		0x16
	63	
0b100100		
		0xC30
	0	
		0xBAD
	437	

C Bitwise Operators

AND (&) outputs a 1 only when both input bits are 1.

&	0	1
0	**0**	**0**
1	**0**	**1**

OR (|) outputs a 1 when either input bit is 1.

|	0	1
0	**0**	**1**
1	**1**	**1**

XOR (^) outputs a 1 when either input is *exclusively* 1.

^	0	1
0	**0**	**1**
1	**1**	**0**

NOT (~) outputs the opposite of its input.

~	
0	**1**
1	**0**

Masking is very commonly used with bitwise operations. A mask is a binary constant used to manipulate another bit string in a specific manner, such as setting specific bits to 1 or 0.

Exercises:

1) What happens when we fix/set one of the inputs to the 2-input gates? Let x be the other input.
Fill in the following blanks with either 0, 1, x, or $\bar{x}$ (NOT x):

```
x & 0 = ______      x | 0 = ______      x ^ 0 = ______

x & 1 = ______      x | 1 = ______      x ^ 1 = ______
```

2) **Lab 1 Helper Exercises:** Lab 1 is intended to familiarize you with bitwise operations in C through a series of puzzles. These exercises are either sub-problems directly from the lab or expose concepts needed to complete the lab. Start early!

Bit Extraction: Returns the value (0 or 1) of the 19th bit (counting from LSB). Allowed operators: >>, &, |, ~.

```
int extract19(int x) {
    return ________________________________;
}
```

Subtraction: Returns the value of x−y. Allowed operators: >>, &, |, ~, +.

```
int subtract(int x, int y) {
    return ________________________________;
}
```

Equality: Returns the value of x==y. Allowed operators: >>, &, |, ~, +, ^, !.

```
int equals(int x, int y) {
    return ________________________________;
}
```

Divisible by Eight? Returns the value of (x%8)==0. Allowed operators: >>, <<, &, |, ~, +, ^, !.

```
int divisible_by_8(int x) {
    return ________________________________;
}
```

Greater than Zero? Returns the value of x>0. Allowed operators: >>, &, |, ~, +, ^, !.

```
int greater_than_0(int x) {
    return ________________________________;
}
```

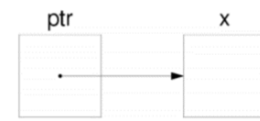
Pointers & Bit Operators

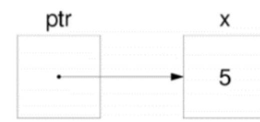
Pointers

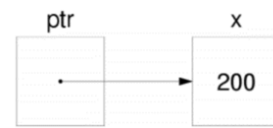
A pointer is a variable that holds an address. C uses pointers explicitly. If we have a variable `x`, then `&x` gives the address of `x` rather than the value of `x`. If we have a pointer `p`, then `*p` gives us the value that `p` points to, rather than the value of `p`.

Consider the following declarations and assignments:

```
int x;
int *ptr;
ptr = &x;
```

1) 

2) 

3) 

1) We can represent the result of these three lines of code visually as shown. The variable `ptr` stores the address of x, and we say "`ptr` points to x." `x` currently doesn't contain a value since we did not assign `x` a value!
2) After executing `x = 5;`, the memory diagram changes as shown.
3) After executing `*ptr = 200;`, the memory diagram changes as shown. We modified the value of `x` by dereferencing `ptr`.

Pointer Arithmetic

In C, arithmetic on pointers (++, +, --, -) is scaled by the size of the data type the pointer points to. That is, if `p` is declared with pointer **`type*`** `p`, then `p + i` will change the value of `p` (an address) by `i*sizeof(`**`type`**`)` (in bytes). If there is a line `*p = *p + 1`, regular arithmetic will apply unless `*p` is also a pointer datatype.

<u>Exercise</u>:

Draw out the memory diagram after sequential execution of each of the lines below:

```
int main(int argc, char **argv) {
  int x = 410, y = 350;   // assume &x = 0x10, &y = 0x14
  int *p = &x;            // p is a pointer to an integer
  *p = y;
  p = p + 4;
  p = &y;
  x = *p + 1;
}
```

Line 1:	Line 2:	Line 3:
Line 4:	Line 5:	Line 6: