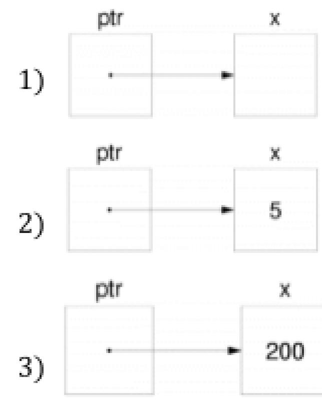
CSE 351 Section 2 – Pointers, Bit Operators, Integers

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) 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:

C Bitwise Operators

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

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

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

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

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

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

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.

<u>Exercises</u>:

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) Bit Manipulation/Number Representation exercises:

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 ____________________________________________;
}
```

3) Implement the following C function using control structures and bitwise operators.

```
// returns the number of pairs of bits that are the
// opposite of each other (i.e. 0 and 1 or 1 and 0)
//
// bits are "paired" by taking adjacent bits
// starting at the lsb (0) and pairs do not overlap.
// For example, there are 16 distinct pairs in a 32-bit integer
int num_pairs_opposite(int x) {





}
```

Signed Integers with Two's Complement

Two's complement is the standard for representing signed integers:

- The most significant bit (MSB) has a negative value; all others have positive values (same as unsigned)
- Binary addition is performed the same way for signed and unsigned
- The bit representation for the negative value (additive inverse) of a Two's Complement number can be found by:

  flipping all the bits and adding 1 (i.e. $-x = \tilde{}x + 1$).

The "number wheel" showing the relationship between 4-bit numerals and their Two's Complement interpretations is shown on the right:

- The largest number is 7 whereas the smallest number is -8
- There is a nice symmetry between numbers and their negative counterparts except for -8

Exercises: (assume 8-bit integers)

1) What is the **largest integer**? The **largest integer + 1**?

Unsigned:	Two's Complement:

2) How do you represent (if possible) the following numbers: **39**, **-39**, **127**?

<u>Unsigned</u>: 39: -39: 127:	<u>Two's Complement</u>: 39: -39: 127:

3) Compute the following sums in binary using your **Two's Complement** answers from above. *Answer in hex.*

a. 39 -> 0b _ _ _ _ _ _ _ _ +(-39) -> 0b _ _ _ _ _ _ _ _ 0x _ _ <- 0b _ _ _ _ _ _ _ _	**b.** 127 -> 0b _ _ _ _ _ _ _ _ + (-39)-> 0b _ _ _ _ _ _ _ _ 0x _ _ <- 0b _ _ _ _ _ _ _ _
c. 39 -> 0b _ _ _ _ _ _ _ _ +(-127)-> 0b _ _ _ _ _ _ _ _ 0x _ _ <- 0b _ _ _ _ _ _ _ _	**d.** 127 -> 0b _ _ _ _ _ _ _ _ + 39 -> 0b _ _ _ _ _ _ _ _ 0x _ _ <- 0b _ _ _ _ _ _ _ _

4) Interpret your answers from 2 & 3 and indicate if overflow has occurred for each of the representations. (For values that cannot be represented, interpret as Two's Complement, then convert to unsigned.)

a. 39+(-39) Unsigned: Two's Complement:	**b.** 127+(-39) Unsigned: Two's Complement:
c. 39-127 Unsigned: Two's Complement:	**d.** 127+39 Unsigned: Two's Complement: