CSE 351 Section 2 – Pointers and 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:

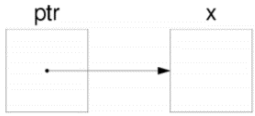
```
int x;
int *ptr;
ptr = &x;
```

1) 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!

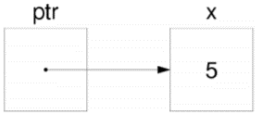
2) After executing `x = 5;`, the memory diagram changes as shown.

3) After executing `*ptr = 200;`, the memory diagram changes as shown. We modified the value of `x` by dereferencing `ptr`.

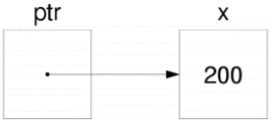
1)



2)



3)



Pointer Arithmetic

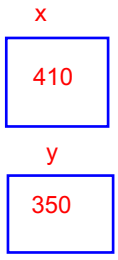
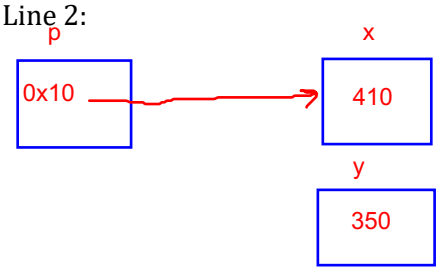
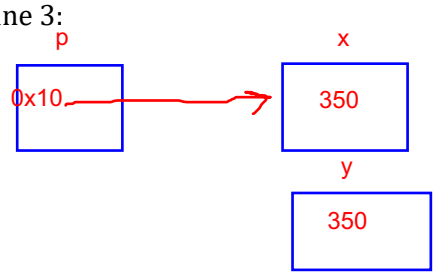
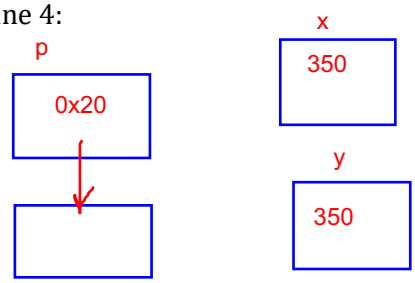
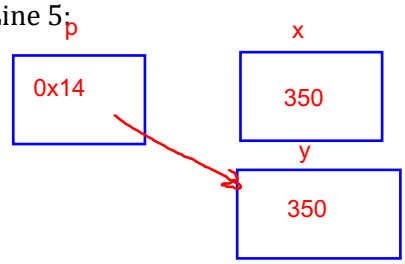
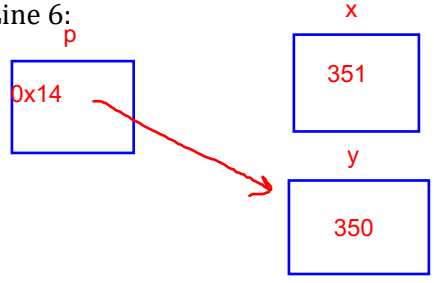
In C, arithmetic on pointers (++, +, --, -) is scaled by the size of the data type the pointer points to. That is, if `p` is declared with pointer **`type*`** `p`, then `p + i` will change the value of `p` (an address) by `i*sizeof(`**`type`**`)` (in bytes). However, `*p` returns the data *pointed at* by `p`, so pointer arithmetic only applies if `p` was a pointer to a pointer.

Exercise:

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;
}
```

pointer arithmetic applies; namely the RHS is 0x10 + 4 * sizeof(int) =/= 0x14 = y

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.

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

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

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

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

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

NOT (`~`) outputs the opposite of its input. →

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 =` 0 `x | 0 =` x `x ^ 0 =` x

`x & 1 =` x `x | 1 =` 1 `x ^ 1 =` NOT x

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 (x>>18) & 0x1;
}
```

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

```
int subtract(int x, int y) {
    return x + ((~y) + 1);    Remember from the 2's complement encoding that -y = ~y + 1
}
```

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

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

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

divisible by 8 IFF the last 3 bits of x are zero (correspond to 1, 2, 2^2 terms)

```
int divisible_by_8(int x) {
    return !(x & 0x7);    0x7 mask zeros out everything besides last three bits.
}
```

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

in the 2's complement encoding, the most significant bit encodes the sign of the number (effectively). The number is negative if this bit is 1

```
int greater_than_0(int x) {
    return ((~x) + 1) >> 31 ) & 0x1;
}
```

Here since we want to check x > 0, not x >= 0, it helps to consider -x = ~x + 1 instead as x > 0 IFF -x < 0 IFF -x has a 1 if its most significant bit. Since ints are stored in 4 bytes = 32 bits, we throw away the other 31 bits with the arithmetic right shift >>