CSE 351
Section 2

Download the Section Handout!

https://courses.cs.washington.edu/courses/cse351/20su/sections/02/cse351_sec2.pdf

Pointers

Pointer Operations

&p

Gives the memory address of the variable p, rather than its value.

*p

Give the value at address p, rather than the value p itself. We often call this “dereferencing.”

Say we had a variable **x** with the value **0x15F**, stored at **0x400**. Then:

- The expression **&x** would evaluate to 0x400
- The expression **x** would evaluate to 0x15F
- The expression ***x** would evaluate to (the value stored at address 0x15F)

Pointer Arithmetic

In C, arithmetic on pointers (++, +, --, -) is *scaled by the size of the data type the pointer points to*. Consider **p** declared with pointer `type* p`;

- The expression `p = p + i` will change the value of **p** (an address) by `i*sizeof(type)` (in bytes).
- By contrast, the line `*p = *p + 1` will perform regular arithmetic unless `*p` is also of a pointer data type.

What About Arrays?

```
int y[10];
int *z;
z = y;

y[2] = 5;
z[2] = 5;
*(z + 2) = 5;
```

These are equivalent!

Arrays in C are contiguous chunks of memory, but they have a special relationship with pointers.

If we have an array variable, it functions like a constant pointer to the first element in the array.

We will discuss arrays in more detail in a future section!

Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

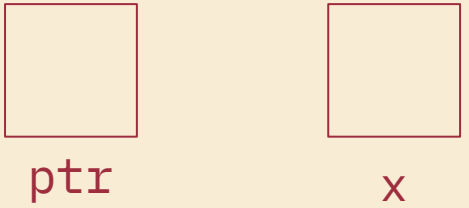
Consider the code to the left. How can we represent the result after each line diagrammatically?

Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

Declare two variables, an int and a pointer to an int.

Note that neither is initialized! We've set aside space for the variables but they're full of garbage.

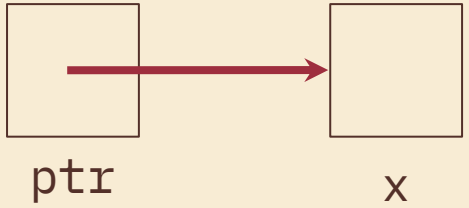


Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

We use the address-of operator to assign the address where the variable x is stored to ptr.

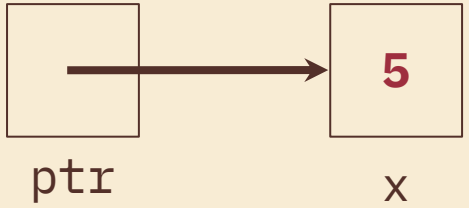
Remember, a pointer is just a variable which holds an address!



Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

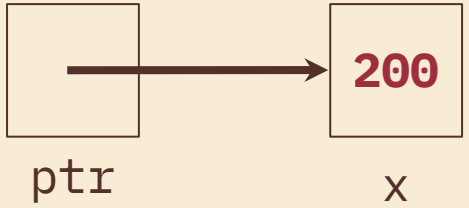
Now we assign x a value.



Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

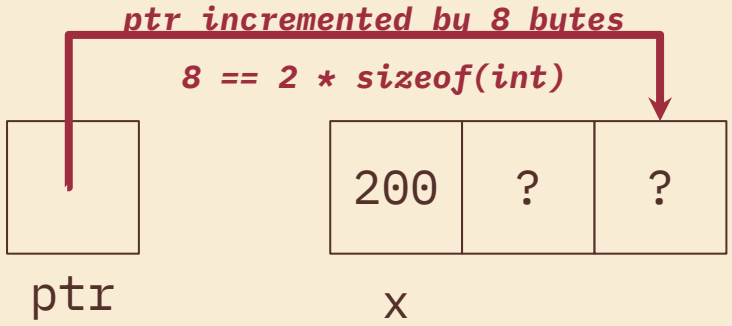
Dereference ptr and assign a value at the location pointed to. This is the location where x is, so we've changed the value of x!



Example

```
int x;
int *ptr;
ptr = &x;
x = 5;
*ptr = 200;
ptr += 2;
```

Increment ptr by 2. Now that we're manipulating a pointer variable, we perform pointer arithmetic. The value of x does not change.



Exercise

```
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;
}
```

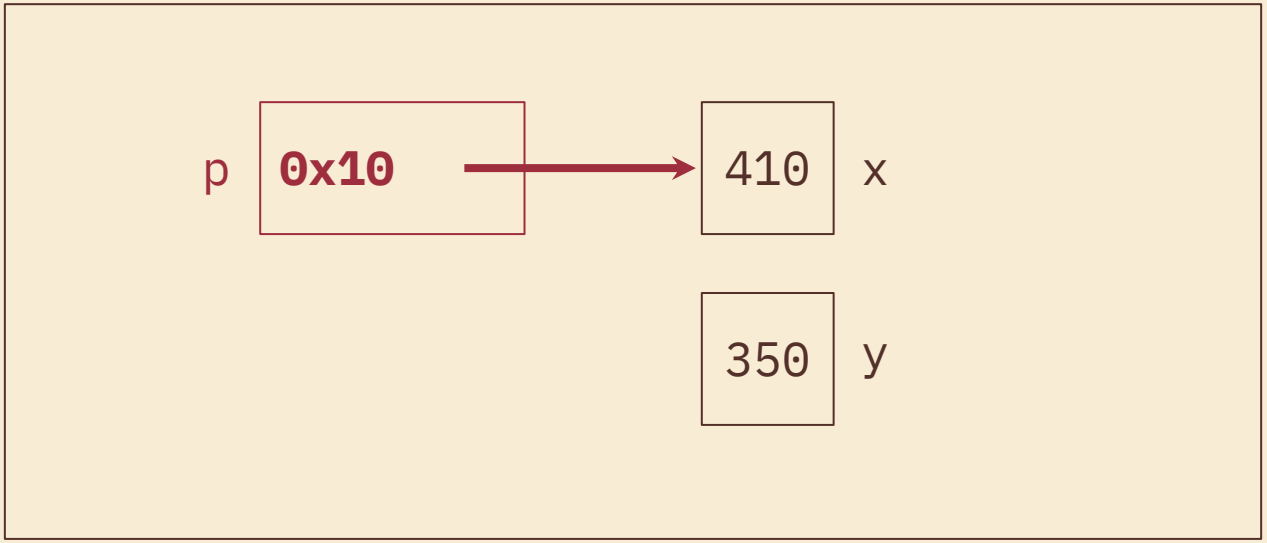
You try!

Exercise

```
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;
}
```

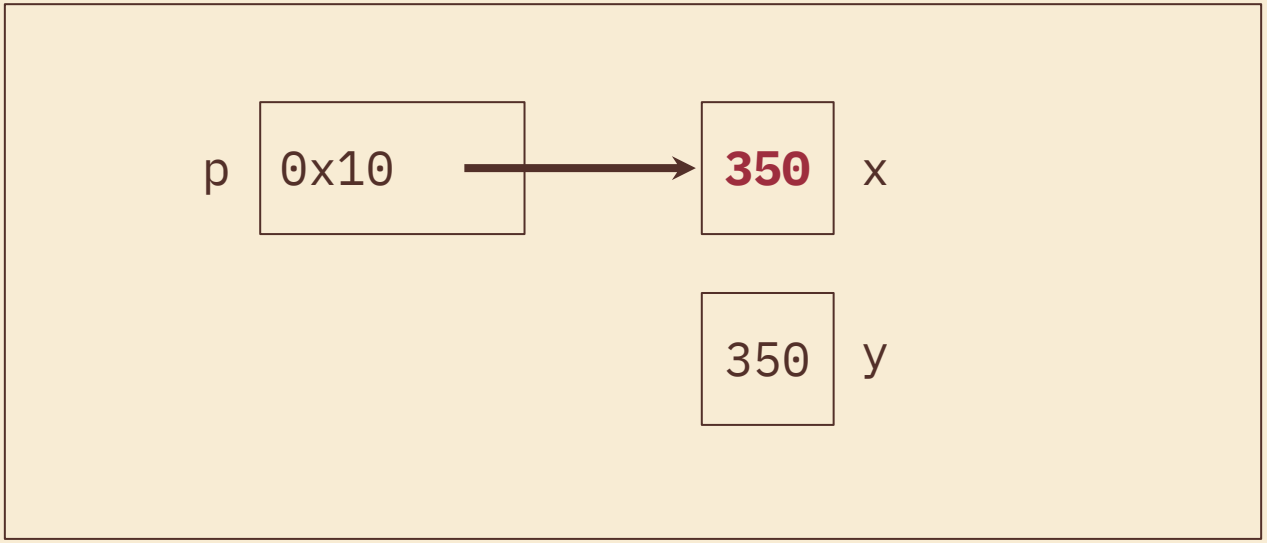
Exercise

```
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;
}
```



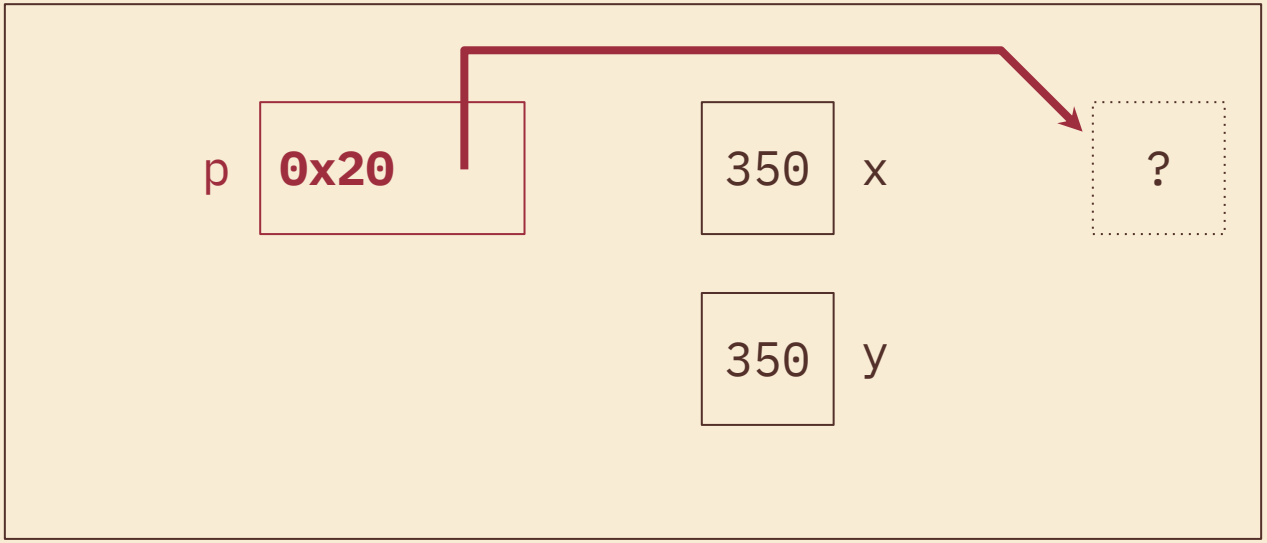
Exercise

```
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;
}
```



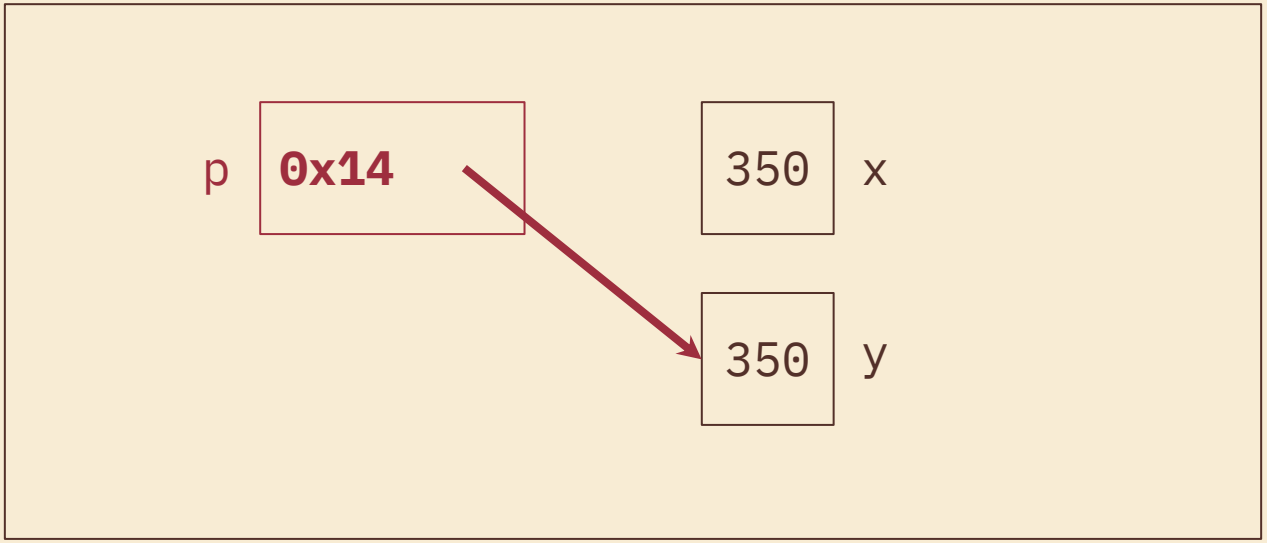
Exercise

```
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;
}
```



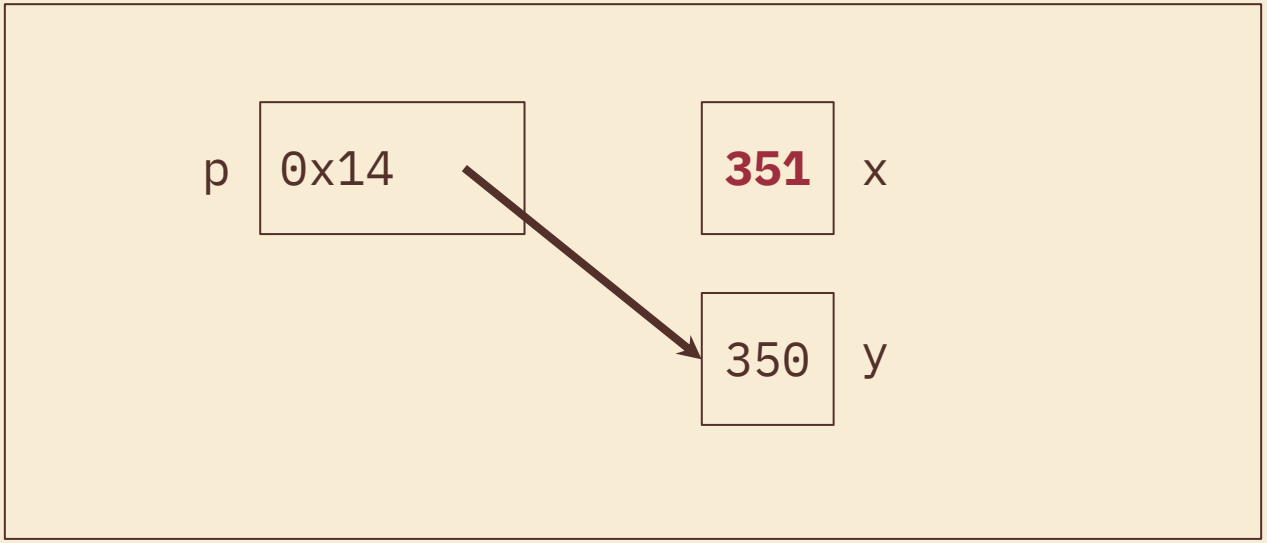
Exercise

```
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;
}
```



Exercise

```
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;
}
```



Bit Operators

Bitwise Operators in C

NOT: `~`

This will flip all bits in the operand (tip: `-x = ~x + 1`)

AND: `&`

This will perform a bitwise AND on every pair of bits

OR: `|`

This will perform a bitwise OR on every pair of bits

XOR: `^`

This will perform a bitwise XOR on every pair of bits

SHIFT: `<<, >>`

This will shift the bits right or left

logical vs. arithmetic

(tip: can be used to multiply and divide by powers of 2)

Bitwise Exercise 1:

```
x & 0 = 0        x & 1 = x
x | 0 = x        x | 1 = 1
x ^ 0 = x        x ^ 1 = ~x
```

Masking

Using a specific bit vector and operator to change data or extract information

Examples:

- Zero out the least significant byte
- Get the most significant bit of x

Lab 1 Helper Exercises

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

```
int extract19(int x) {
        (x >> 18) & 0x1;
}
```

Lab 1 Helper Exercises

```
// 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) {
  int count = 0;
  for (int i = 0; i < 16; i++) {
    int bit0 = x & 1;
    int bit1 = (x >> 1) & 1;
    count += bit0 ^ bit1;
    x >>= 2;
  }
  return count;
}
```

Two's Complement

What's Two's Complement?

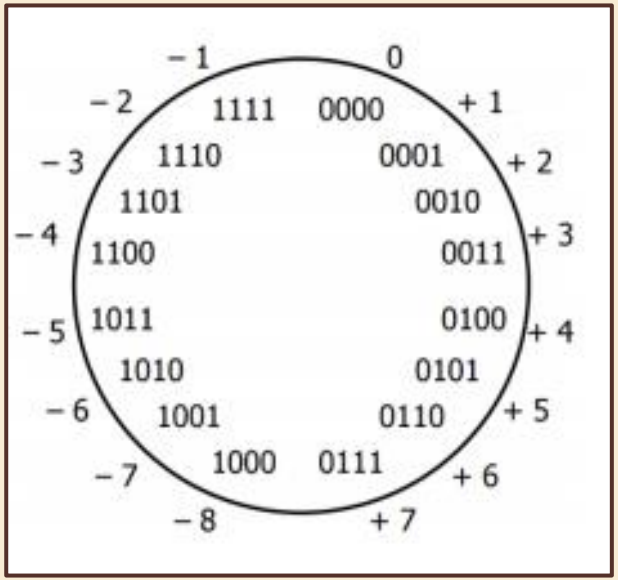
A way of representing *signed* integers (positive or negative)

Similar to signed integers, except the most significant bit has negative "weight" (but equivalent magnitude)

Why Two's Complement?

We use two's complement because it has many handy properties:

- Addition and subtraction are performed the same way as unsigned
- Positive numbers are represented the same way as unsigned
- Single zero (compare sign-magnitude)
- The representation of 0 is all zeroes (0b0...0)
- Roughly the same number of negative and positive integers



Negation

If we want to negate a two's complement integer, we **flip every bit and add 1:**

```
-x = ~x + 1
```

1	1	0	0	0	1	0	0	
-128	64	0	0	0	4	0	0	-60

0	0	1	1	1	0	1	1	
0	0	32	16	8	0	2	1	59

0	0	1	1	1	1	0	0	
0	0	32	16	8	4	0	0	60

Overflow

We use the term **overflow** to describe when an integer operation would give a result outside the range of representable values (computers are finite!). This happens when we have to “carry” to one bit beyond the MSB.

For signed integers, we can identify it when the resulting value is not what we would expect mathematically:

```
    0b 0110 0100      100
+   0b 0011 1100      60
-------------------------
    0b 1010 0000      -96??
```

Would this be unsigned overflow?

Exercise 1

What is the largest 8-bit integer and what happens when we add 1?

Unsigned	Two's Complement
`0b11111111 (255)` `0b00000000 (0)`	`0b01111111 (127)` `0b10000000 (-128)`

Exercise 2

What are the 8-bit representations of the following numbers?

	Unsigned	Two's Complement
39:	**0b00010111**	**0b00010111**
-39:	*Can't do it!*	**0b11101001**
127:	**0b01111111**	**0b01111111**

Remember! **`-x = ~x + 1`**

Exercise 3

Compute using two's complement:

a. 39 -> 0b 0 0 1 0 0 1 1 1 +(-39) -> 0b 1 1 0 1 1 0 0 1 0x 0 0 <- 0b 0 0 0 0 0 0 0 0	**b.** 127 -> 0b 0 1 1 1 1 1 1 1 + (-39)-> 0b 1 1 0 1 1 0 0 1 0x 5 8 <- 0b 0 1 0 1 1 0 0 0
c. 39 -> 0b 0 0 1 0 0 1 1 1 +(-127)-> 0b 1 0 0 0 0 0 0 1 0x A 8 <- 0b 1 0 1 0 1 0 0 0	**d.** 127 -> 0b 0 1 1 1 1 1 1 1 + 39 -> 0b 0 0 1 0 0 1 1 1 0x A 6 <- 0b 1 0 1 0 0 1 1 0

Exercise 4

Interpret results. Did overflow occur?

	`39 + (-39) = 0x00`	`127 + (-39) = 0x58`
Unsigned	**0** overflow	**88** overflow
Two's Complement	**0** no overflow	**88** no overflow
	`39 + (-127) = 0xA8`	`127 + 39 = 0xA6`
Unsigned	**168** no overflow	**166** no overflow
Two's Complement	**-88** no overflow	**-90** overflow