CSE 351 Section 2

Pointers, Bitwise, Ints
Autumn 2022

Administrivia

- Lab 1a due (Monday 10/10 11:59 PM)
  - Give yourself time to go into Thursday, Friday, and Monday OH if needed
  - Homework 4 also due Monday
- Homework due every day we have lecture
  - Unlimited attempts
  - Homework 3 due (Friday 10/07 11:59 PM)
  - Homework 4 due (Monday 10/10 11:59 PM)
  - More throughout next week

Pointers

Pointer Operations

&x

Gives the memory address of the variable x, 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 (note: not always! e.g. `sizeof`)

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

Example #1

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

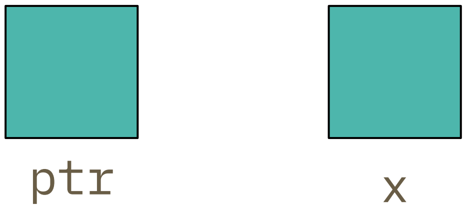
Consider the code to the left. How can we represent the result after each line diagrammatically?

Example #1

```
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 mystery data.

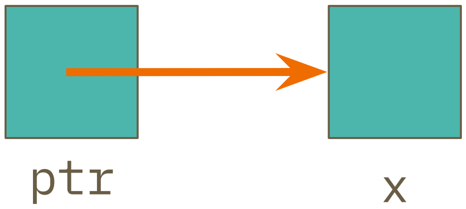


Example #1

```
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.

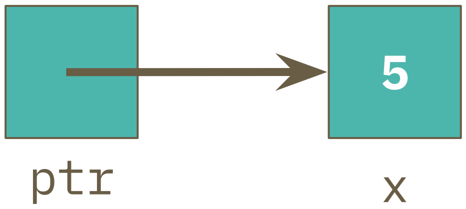
Remember, a pointer is just a variable which holds an address!



Example #1

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

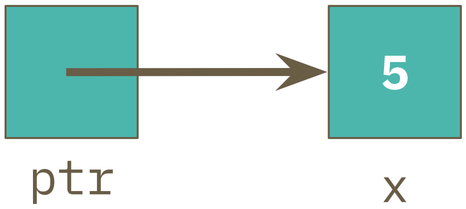
Now we assign x a value.



Example #1

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

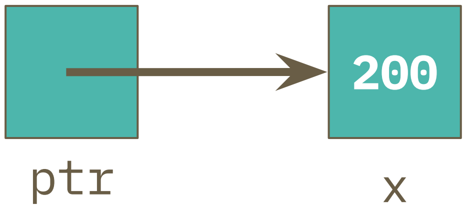
Now we assign x a value.



Example #1

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

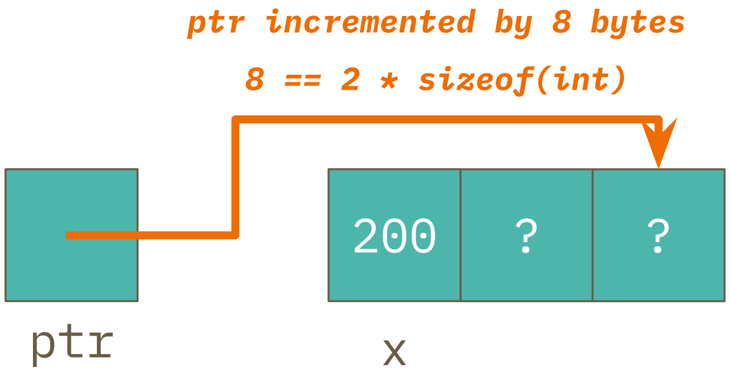
Dereference ptr and assign a value at the location pointed to. This is the location where x is, so we've changed the value of x!



Example #1

```
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 #1

```
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”- first page of the section handout

Bitwise Operators

Bitwise Operators in C

These perform operations on *each bit independently* in a value.

`NOT: ~x`

0	**1**
1	**0**

"Flips" all bits

`AND: x & y`

	0	1
0	**0**	**0**
1	**0**	**1**

1 iff both bits 1

`OR: x | y`

	0	1
0	**0**	**1**
1	**1**	**1**

1 iff either or both 1

`XOR: x ^ y`

	0	1
0	**0**	**1**
1	**1**	**0**

1 iff exactly one is 1

Bitwise Operators in C

These perform operations on *each bit independently* in a value.

NOT: ~x

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

"Flips" all bits

AND: x & y

x	y	x & y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

1 iff both bits 1

OR: x | y

x	y	x | y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

1 iff either or both 1

XOR: x ^ y

x	y	x ^ y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

1 iff exactly one is 1

Bitwise vs Logical

Bitwise operators are **not the same as logical operators (!, &&, ||)**.

While they perform similar "logical" operations (AND, OR, NOT), bitwise operators transform the *individual bits* of a value, whereas logical operators are used in boolean expressions and treat *entire values* as either true or false.

For example, `0xA & 0x5 = 0x0`, but `0xA && 0x5 = 0x1`.

```
   0xA = 0b1010
 & 0x5 = 0b0101
 --------------
   0x0 = 0b0000
```

```
   0xA = 0b1010
&& 0x5 = 0b0101
 --------------
   0x1 = 0b0001
```

Masking Example

Masking is using a specific bit vector and operator to change data or extract information.

How would you replace the least significant byte of x with 0xAA? For example: 0x2134 should become 0x21AA.

1. Zero out the LS byte with an AND mask.
   - **`x = x & ~0xFF`** (or `x &= ~0xFF`)

1. Use an OR to set the LS byte.
   - **`x = x | 0xAA`** (or `x |= 0xAA`)

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

Masking Example

Masking is using a specific bit vector and operator to change data or extract information.

How would you replace the least significant byte of x with 0xAA? For example: 0x2134 should become 0x21AA.

1. Zero out the LS byte with an AND mask.
   - `x = x & ~0xFF` (or `x &= ~0xFF`)

1. Use an OR to set the LS byte.
   - `x = x | 0xAA` (or `x |= 0xAA`)

0x2134	0010 0001 0011 0100
& ~0xFF	1111 1111 0000 0000
	0010 0001 0000 0000
| 0xAA	0000 0000 1010 1010
0x21AA	0010 0001 1010 1010

Exercise 1

If **`signed char a = 0x88`**, complete the *bitwise* C statement so that **`b = 0xF1`**. The first blank should be an operator and the second should be a numeral.

```
  a = 0b10001000

______________________
0xF1 = 0b11110001
```

```
b = a ____ 0x ____
```

Exercise 2

```
// 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.
// eg: 00 11 01 00 11 10 should return 2 (note that int type is much longer)

int num_pairs_opposite(int x) {
  int count = 0;
  for (int i = 0; i < 8 * sizeof(int) / 2; i++) {
    // fill in the for loop!



  }
  return count;
}
```

Integers

What's Two's Complement?

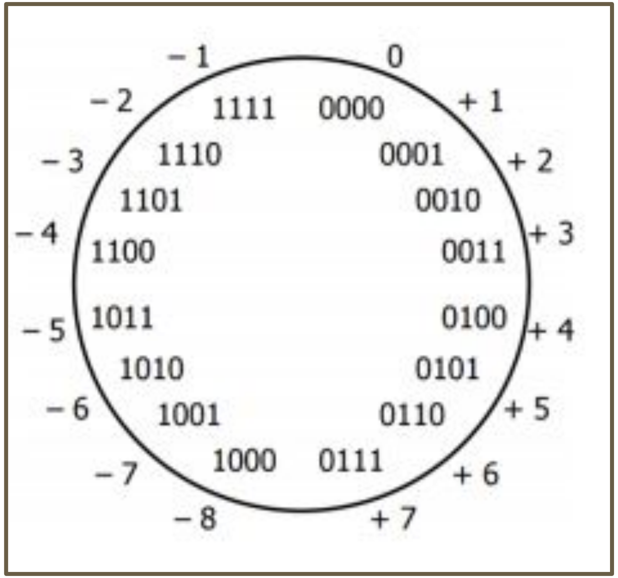
A way of representing *signed* integers (positive or negative)

Similar to unsigned 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**

Exercise 1a

What is the largest 8-bit integer? What happens when we add 1? What is the **most negative** integer we can represent?

Unsigned	Two's Complement
Largest: Largest + 1: Most Negative:	Largest: Largest + 1: Most Negative:

Exercise 1b

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

	Unsigned	Two’s Complement
39:		
-39:		
127:		

Remember! $-x = \sim x + 1$

Exercise 2

Take the 32-bit numeral 0xC0800000. Circle the number representation below that has the most negative value for this numeral.

Sign & Magnitude	Two's Complement	Unsigned

Exercise 3

Given the 4-bit bit vector 0b1101, what is its value in decimal (base 10)? Circle your answer.

13	-3	-5	Undefined