BIT TWIDDLING

Stream Revue

- How do you read in an entire line from cin?
- What object do we use for opening files for output? For input?
- How would we check to see if our output object has any errors?
- What are these operators called? **<< >>**

More Revue

```
class time
{
   private:
      int hour, min;
};
```

```
time u, t;
t = u;
```

- How would we go about fixing this class so we can use cin/cout?
- How would we go about fixing this class so we can use the addition operator on it?
- How would we make this code compile and run properly?

Even More Revue!

```
class time
{
    public:
        void chime();
    protected:
        int hour, min;
};

class secTime : public time
{
    protected:
        int sec;
};
```

```
void doStuff( time t )
{
    // ...
}
```

```
int main()
{
    secTime st;
    doStuff( st );
}
```

Does this code work?
What be wrong with it?

Logical Operators

- These familiar operators (&&, ||, !) are called **logical operators**
- They operate on *entire* expressions
- So **a** || **b** is going to be true only if a is true, or b is true
- ... aka, when we evaluate **a** and **b**, at least one of them comes out as non-zero

Son of Logical Operators

- The logical and/or/not operators operate on the *entire value* at once - all of **a**, or all of **b**
- A given value is made up of 32 bit (mostly)
- Sometimes we want to do things with individual bits!
- We can do this with a different set of operators called **bitwise operators**

Writing in Binary

- For this lecture we're going to mostly stick with integers - unsigned integers in particular
- In memory, each integer is made up like this:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- There are 32-bits (16 in this picture) and we usually write them right-to-left
- This is how we write base-10 numbers too, if you think about it - least significant # goes last

Bitwise Ops

- There are unary bitwise operators (one argument) and binary bitwise arguments (two arguments)
- The bitwise versions operate on the corresponding bits of each of their arguments)
- So for **a & b**, the 0th bit of **a** is *and*-ed with the 0th bit of **b**, and so on

Son of Bitwise

- Bitwise AND (**&**): resulting bit is true only if **both** input bits are true
- Bitwise OR (**|**): resulting bit is true if **either** of the input bits are true
- Bitwise XOR (**^**): resulting bit is true if **exactly one** of input bit is true
- Unary Bitwise NOT (~): flips each bit

Bitwise And-ing

Example:

a & b

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a val:	0	1	1	0	1	0	0	1	0	1	0	0	0	0	1	1
b val:	1	1	0	0	0	1	0	1	0	1	1	0	1	0	0	0
result:	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0

Why Bother?

- This stuff is used quite often in low-level programming
- One use often seen in *high-level* programming, however:
- A boolean value is either true or false, which can be represented by a single bit
- So we can cram 32 boolean values into a single 32-bit integer!

Specifying Flags

constant name	value
ios::in	1
ios::out	2
ios::app	4
ios::ate	8
ios::nocreate	16
ios::noreplace	32
ios::trunc	64
ios::binary	128

- This is very common: each potential option is often called a **flag**; we can combine multiple flags together into a single integer
- These a few of the open mode flags; how could we combine a few of them together?
- Why do the values need to be powers of 2?

ios::out

0000	0000	0000	0010
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

ios::nocreate

0000	0000	0001	0000
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

ios::trunc

0000	0000	0100	0000
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

ios::out | ios::nocreate | ios::trunc

0000	0000	0101	0010
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

- In a power-of-2 constant, only a single bit will be “on”
- So we can combine many of them together without “interfering” with other constants

Getting 'em Out

- Now we know how to put a bunch of constants **in** to a bitmask - how do we get them **out**?
- Given an integer **options**, how do we tell if the ios::trunc flag is set?
- How about ios::binary?

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
options	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	
ios::trunc	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	
ios::binary	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	

results:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
options & ios:: trunc	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	= 64
options & ios::binary	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	= 0

- Once we’ve **&**-ed the **options** with the constant we’re checking, the rest is easy
- If none of the bits matched (aka, the **ios::trunc** bit was not set in **options**) the result will be zero
- ... otherwise (if the bit *was* set) it’ll be non-zero
- So we can check the whole thing with a simple **if** statement:

```
if( options & ios::trunc )
    // ... ios::trunc was set!
```

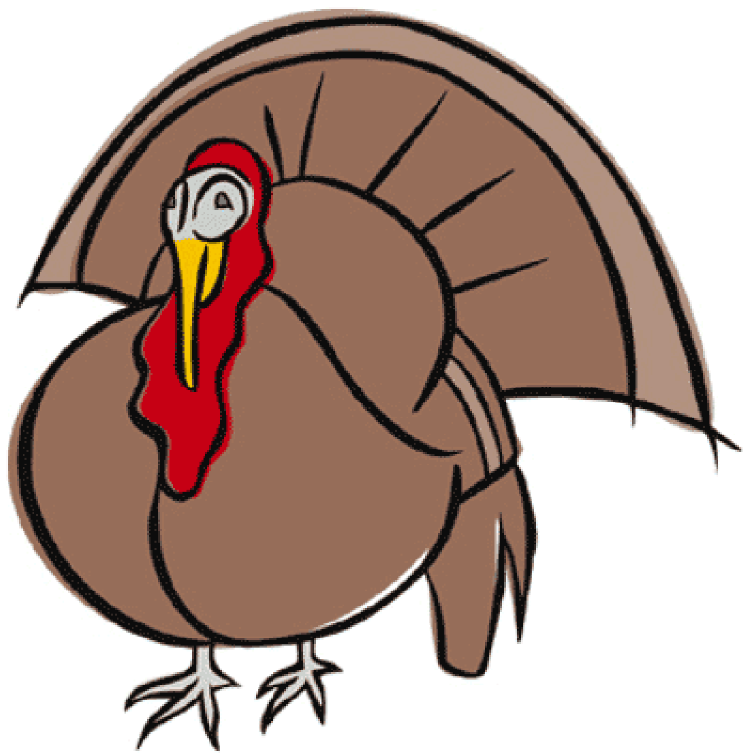
be careful here... what happens if you accidentally use **&&**?

Shifting

- We use the **>>** and **<<** operators all the time, for iostreams (cin, cout, etc)
- ... but that's not what these operators were *meant* for!
- The iostream library turns them into stream operators by overloading them...
- ... but in C (and therefore in C++) these are the **bitshift operators**.

More Shifting

- There are two bitshift operators:
  - Shift left: **<<** (shifts bits to the left)
  - Shift right: **>>** (shifts bits to the right)
- These look like:
  - **x << n**
  - **x >> n**
  - ... where **x** and **n** must be integer variables



Left-Shifting

Take a look at the following input:

0011	0010	1010	0111
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

We can shift by any number of bits, but let's shift left by 4 bits. We get the following results:

0010	1010	0111	0000
15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0

The bits "fall off" the high end of the integer, and the empty spaces on the low end get filled with zeros

An Interesting Effect...

0000 0100 = 4	<< by **1** bit	0000 1000 = 8
0000 1010 = 10	<< by **1** bit	0001 0100 = 20
0000 1010 = 10	<< by **3** bits	0101 0000 = 80

(bit positions: 7 6 5 4 3 2 1 0)

... see what's going on here?

Left Shifting

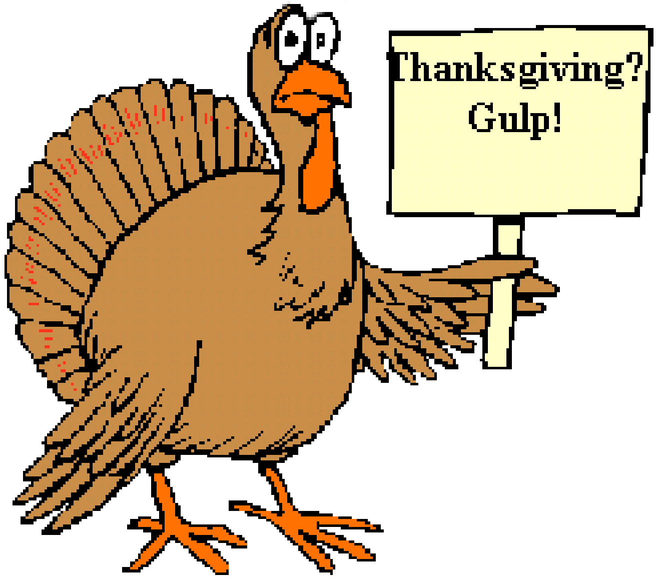
- When you left shift by **n** bits, you are actually *multiplying* by 2_n
  - **q << 1 = q * 2**
  - **q << 3 = q * 8**

- We've been talking mostly about *unsigned* integers, but this works for signed integers too
- Shift too far, though, and you get *overflow* - a number bigger than an int can hold - and therefore the wrong answer
- Shifts and bitwise ops are very efficient

Right Shifting

- When you **right** shift by **n** bits, you are actually *dividing* by 2^n
  - **q >> 1 = q / 2**
  - **q >> 4 = q / 16**
- This is all *integer* division, so the result will just be the quotient - no remainder!
- This works for *unsigned* integers - signed integers are much more unpredictable (depends on how the compiler handles it)

- One thing to remember: just like **a+b**, bitwise/shift operations don't change anything unless you save the result!
- The result of **a+b** needs to be assigned to something to have an effect
- Similarly, **a&b** or **a^b**, etc. does nothing unless you save the result
- A tricksy example:




```
unsigned u = 15;
u << 3;
```

More Operators

- Just like there's **i += 2** to simplify **i = i + 2**, there are corresponding operators for all the shift/bitwise operators
- **<<=, >>=, ^=, &=, |=, ~=**

code

- print numbers in binary
- maybe some more...