BIT TWIDDLING; THE C++ STRING CLASS

Logical Operators

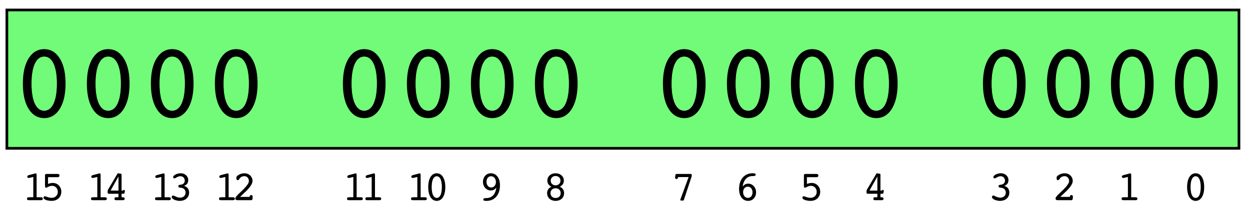
- These familiar operators (&&, ||, !) are called **logical operators**
- They operate on *entire* expressions
- So **a** || **b** is going to be true only if a is true, or b is true
- ... aka, when we evaluate **a** and **b**, at least one of them comes out as non-zero

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:



- 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

a val:	0110	1001	0100	0011
	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
b val:	1100	0101	0110	1000
	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
result:	0100	0001	0100	0000
	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ios::nocreate	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ios::trunc	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

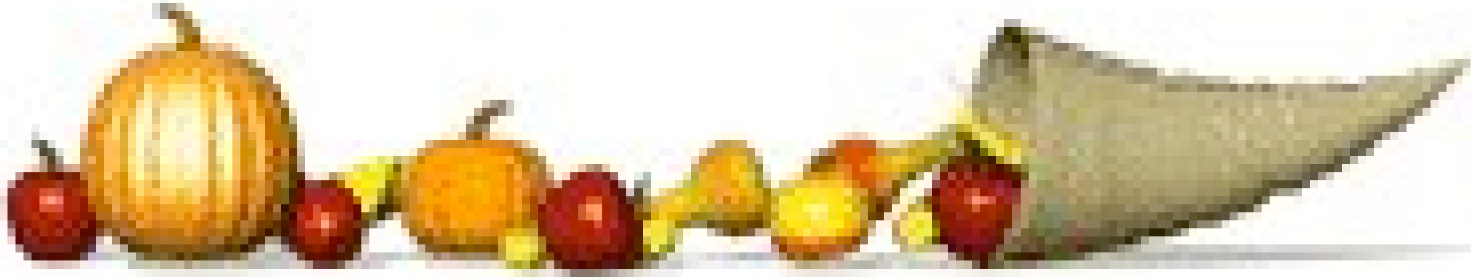
ios::out | ios::nocreate | ios::trunc

0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0
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!
```

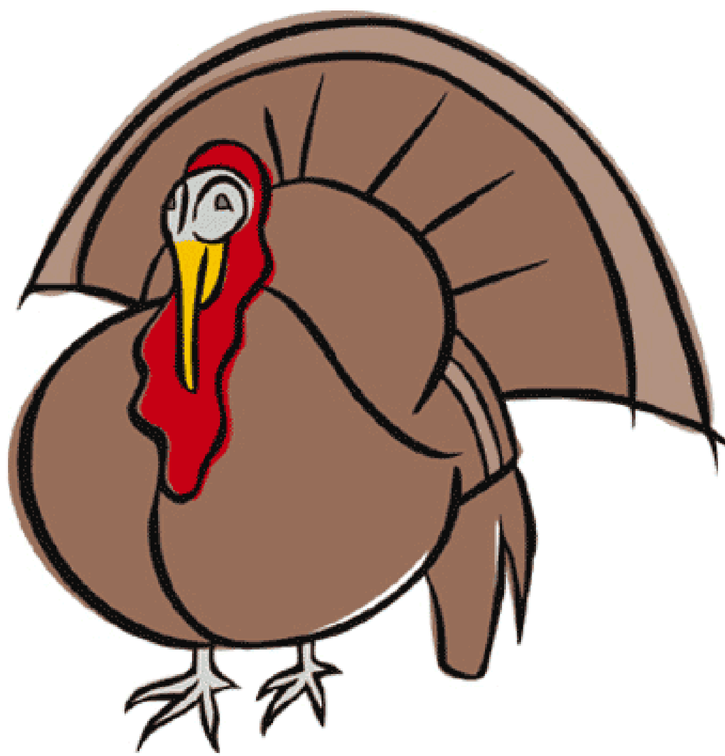
be careful here... what happens if you accidentally use **&&**?

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:

0	0	1	1	0	0	1	0	1	0	1	0	0	1	1	1
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:

0	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0
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
7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0
0000 1010 = 10	<< by **1** bit	0001 0100 = 20
7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0
0000 1010 = 10	<< by **3** bits	0101 0000 = 80
7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0

... see what's going on here?

Left Shifting

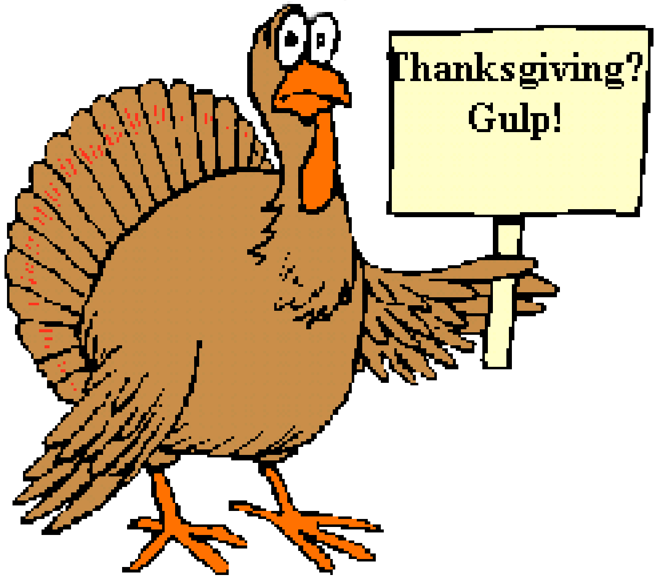
- When you left shift by **n** bits, you are actually *multiplying* by 2_n
  - **q << 1 = q * 2**
  - **q << 3 = q * 8**

- 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

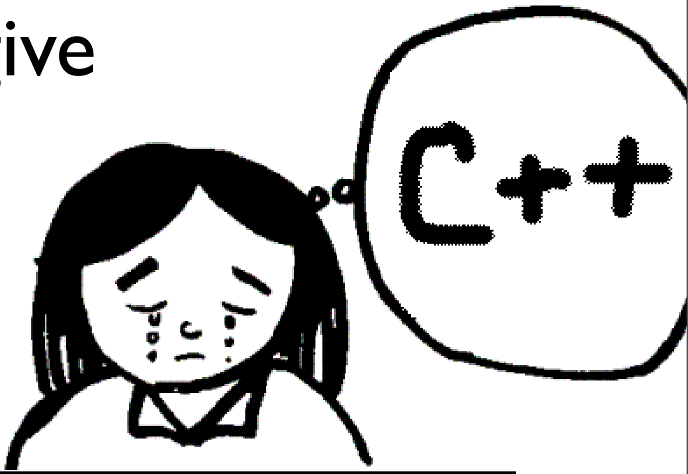
- Just like there's **i += 2** to simplify **i = i + 2**, there are corresponding operators for all the shift/bitwise operators
- **<<=, >>=, ^=, &=, |=, ~=**

C++ Strings

- The C++ **string** class encapsulates (duh) a string
- These tend to be much easier to work with than C-style strings
- Also: no concerns about about length; C++ strings resize themselves as required
- Also very fast to pass by value: they use a technique called **copy on write** in which the data buffer isn't copied until it absolutely has to be

Declaring a string

- First, **#include <string>**
- **string** is part of the **std** namespace, so **using namespace std** will give you access
- Or you can use the std:: prefix:




```
std::string myString = “greg was here”;

std::string anotherString(“yep, still here”);
```

String Input

- **operator>>** has been overloaded for the string class, so you can use them with an istream:

```
string myString;

// reads in a single word
cin >> myString;

// reads in an entire line
getline(cin, my_string, '\n');
```

operator+

- The + and += operators are defined for strings, for easy concatenation:

```
string one = “pointers ”;
string two = “are fun!”;
string three = one + two;
three += “ hooray!”

cout << “hey! ” + three << endl;
```

- Notice that this also works for regular C-style strings as well.

Comparisons

- The string class overloads all the comparison operators - no more strcmp()!
- Again, this is a lexicographic (dictionary-order) comparison

```
string apple("apple");
string banana("banana");

if( apple == banana )
    cout << "apple == banana";
else if( apple > banana )
    cout << "apple > banana";
else
    cout << "apple < banana";
```

Individual elements

- You can get a string's length using the **length** or **size** methods
- ... and access individual characters using square brackets (the string has overloaded **operator[]**)
- To print out each character in a string:

```
for( int i = 0; i < myStr.length(); i++ )
{
    cout << myStr[i] << endl;
}
```

Iterating Through a String

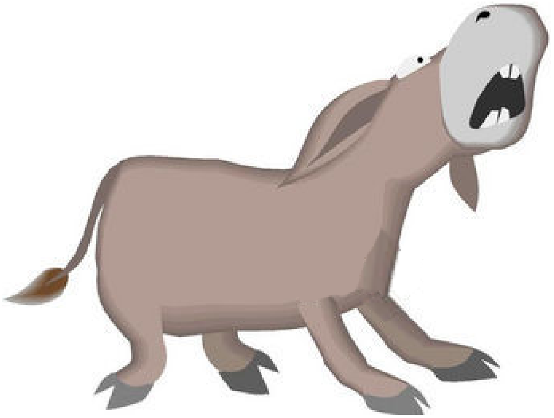
```
for( int i = 0; i < myStr.length(); i++ )
{
    cout << myStr[i];
}
```

- This is the safe way to do things...
- We can’t use the usual C-string tricks because string objects are **not** guaranteed to be NULL-terminated

Iterating Through a String

- ... or, since a string is actually an STL container that contains a bunch of **char**s, we can use STL iterators:

```
string::iterator iter;
for( iter = myStr.begin(); iter != myStr.end(); iter++ )
{
    cout << *iter;
}
```



- One thing to note: iterators are often invalidated when you change the string, so be cautious using them after you do

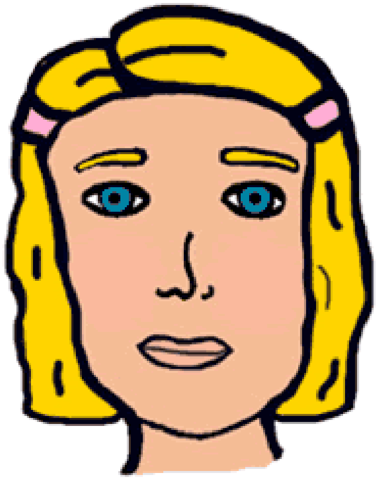
Searching

- **find** and **rfind** search for a substring; **find** starts from the left, **rfind** starts from the right
- These return a **size_type**, which is basically an unsigned int - tells you where the substring is located with the string
- Returns **string::npos** upon failure
- Second (optional) parameter is where to start looking

```
string example("C and C++ are fun!");
cout << example.find("C") << endl;
cout << example.find("C", 1) << endl;
```

Erasing From A String

- the **erase()** method removes chunks from a string
- First parameter is where to start erasing...
- Second (optional) parameter is how many characters to erase; if omitted the rest of the string will be erased



```
string example("I will not miss C++ class!");
example.erase( 7, 4 );
cout << example << endl;
```

Inserting Into a String

- The **insert** method inserts new characters into an existing string
- First parameter: where you want to put the new characters
- Second parameter: the characters to insert

```
string example( "I don't like pointers!" );
example.erase( 2, 10 );
example.insert( 2, "couldn't live without" );
cout << example << endl;
```

Retrieving a **char***

- Sometimes you need to convert a string into a C-style character array...
- To call a function that can't deal with C++ strings maybe
- You do this with the **c_str** method, which makes sure the string is NULL-terminated and returns a pointer to the data

```
string example("I'm already studying for the final!");
cout << strlen( example.c_str() ) << endl;
```