Week 3 - Monday

COMP 2400

Last time

- What did we talk about last time?
- System limits
- ASCII table
- **`printf()`** format strings
- Started bitwise operations

Questions?

Project 2

Quotes

> *One thing I've noticed with C/C++ programmers, particularly (which is, again, the pool from which most C# programmers will be drawn), is that many of them are convinced that they can handle dangerous techniques which experience shows they can't handle. They say things such as, "I like doing my own memory management, because it gives me more control," but their code continually suffers from memory leaks and other pointer-related problems that show quite clearly that they are not to be trusted with these things that give them "more control." This, in my view, is just one more reason why "unsafe" features should not be built into mass-market languages like C#.*
>
> Craig Dickson

Bitwise Operators

Bitwise operators

- Now that we have a deep understanding of how the data is stored in the computer, there are operators we can use to manipulate those representations
- These are:

&	Bitwise AND
|	Bitwise OR
~	Bitwise NOT
^	Bitwise XOR
<<	Left shift
>>	Right shift

Bitwise XOR

- The bitwise XOR operator (**^**) takes:
  - Integer representations **a** and **b**
- It produces an integer representation **c**
  - Its bits are the logical XOR of the corresponding bits in a and b
- Example using 8-bit **char** values:

	0	0	1	0	1	1	1	0	a
^	0	1	0	0	1	1	0	1	b
	0	1	1	0	0	0	1	1	c

```
char a = 46;
char b = 77;
char c = a ^ b; //99
```

Swap without a temp!

- It is possible to use bitwise XOR to swap two integer values without using a temporary variable
- Behold!

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

- Why does it work?
- Be careful: If **x** and **y** have the same location in memory, it doesn't work
- It is faster in some cases, in some implementations, but should not generally be used

Bitwise shifting

- The << operator shifts the representation of a number to the left by the specified number of bits

```
char a = 46;
char b = a << 2;   // -72
```

- The >> operator shifts the representation of the number to the right by the specified number of bits

```
char a = 46;
char b = a >> 3;   // 5
```

- Ignoring underflow and overflow, left shifting is like multiplying by powers of two and right shifting is like dividing by powers of two

Shift and mask examples

- Things smaller than **`int`** will be promoted to **`int`**
- What are the following?
  - **`4 & 113`**
  - **`15 | 39`**
  - **`31 << 4`**
  - **`108 >> 5`**
  - **`~80`**

Why do we care about bitwise operations?

- The computer uses bitwise operations for many things
- These operations are available for our use and are very fast
- Shifting is faster than multiplying or dividing by powers of 2
- You can keep a bitmask to keep track of 32 different conditions
    - That's quite a lot of functionality for four bytes!

Precedence

- Operators in every programming language have precedence
- Some of them are evaluated before others
  - Just like order of operations in math
- ***** and **/** have higher precedence than **+** and **–**
  - **=** has a very low precedence
- I don't expect you to memorize them all, **but**
  - Know where to look them up
  - Don't write confusing code

Precedence table

Type	Operators	Associativity
Primary Expression	() [] . -> expr++ expr--	Left to right
Unary	* & + - ! ~ ++expr --expr (typecast) sizeof	Right to left
Binary	* / %	Left to right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	|	
	&&	
	||	
Ternary	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Insane precedence example

- What happens here?
  - `x++ >> 5 == 4 % 12 & 3`
- It's also worth noting that precedence doesn't tell the whole story
- What about multiple assignments in a single line of code?
- C doesn't give you guarantees about what happens when
- The following could have different results on different compilers:

```
printf("%d %d", x++, (x + 5));
a[x] = x++;
x = x++;
```

Control Flow

Control flow

- Sequences of statements surrounded by braces are treated like a single statement with no value
  - Braces can be thrown in whenever you want
  - We used to say that "braces were optional" for one-line blocks, but this is the more accurate way to look at it
- An expression can always become a statement

```
int a = 150;
a; // Legal (but silly) in C, illegal in Java
```

Selection

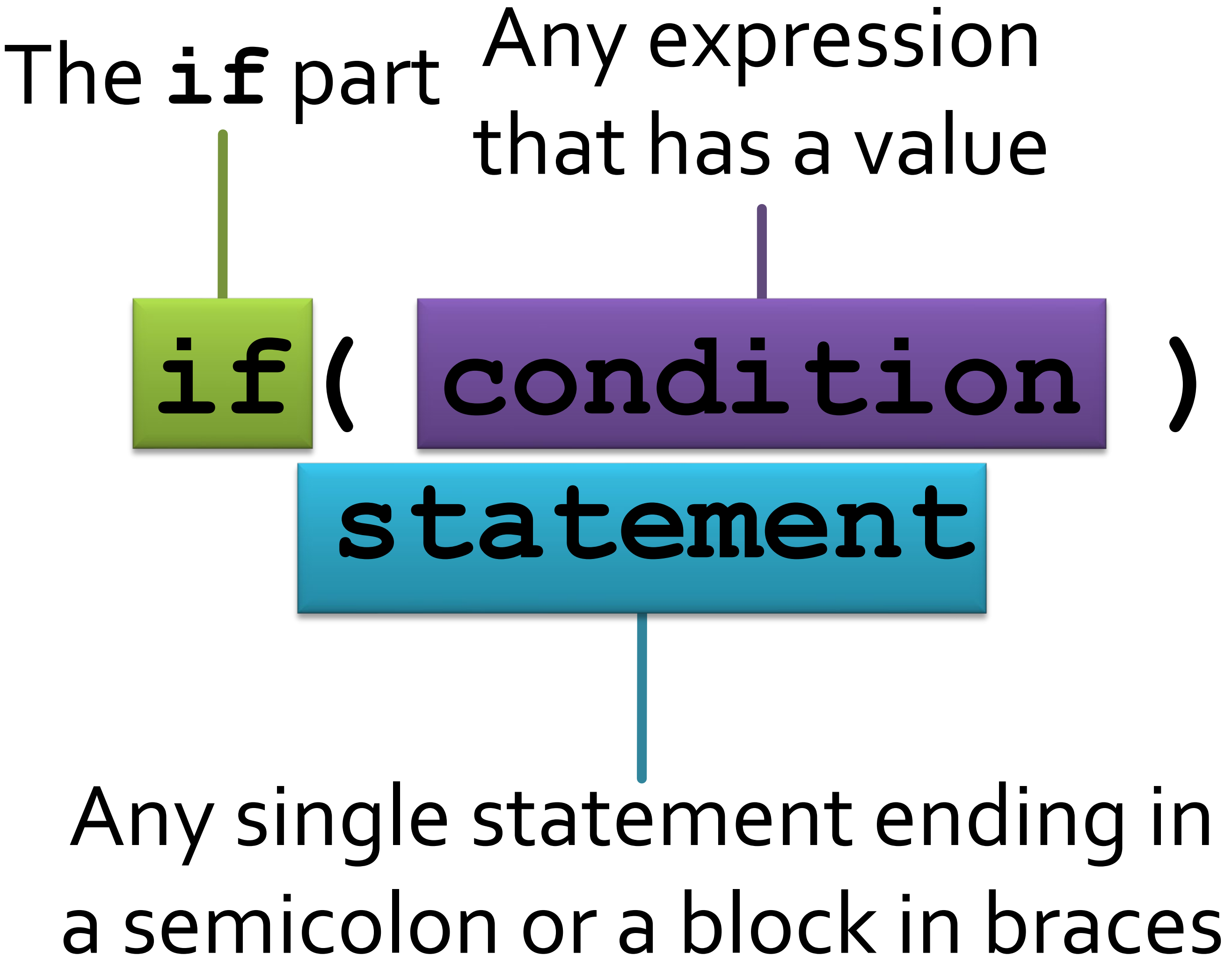
`if` statements

- Like Java, the body of an **`if`** statement will only execute if the condition is true
  - The condition is evaluated to an **int**
  - True means not zero

Sometimes this is natural and clear; at other times it can be cryptic.

- An **`else`** is used to mark code executed if the condition is false

Anatomy of an `if`



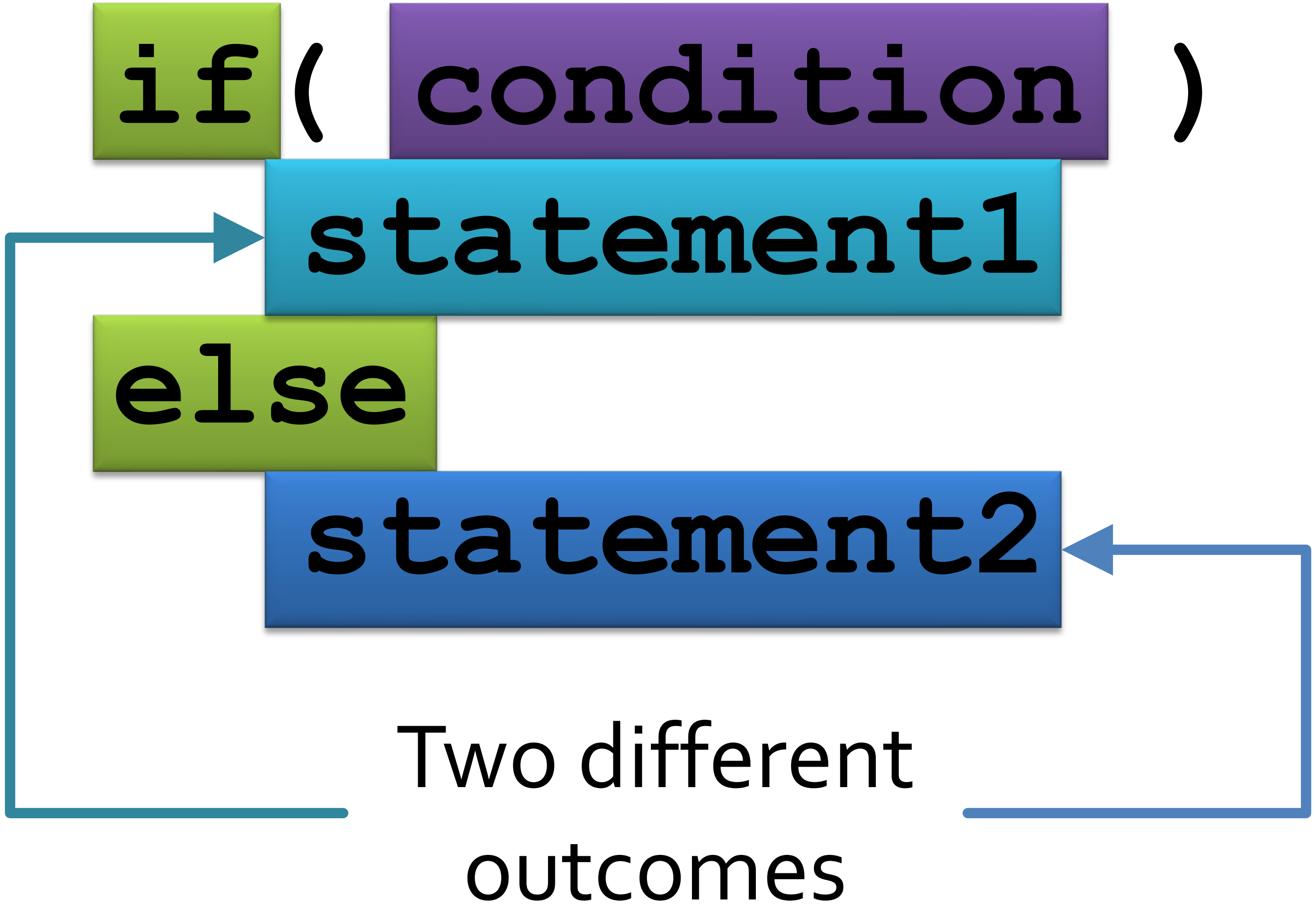
The **`if`** part

Any expression that has a value

```
if( condition )
    statement
```

Any single statement ending in a semicolon or a block in braces

Anatomy of an `if-else`



```
if( condition )
   statement1
else
   statement2
```

Two different outcomes

Nesting

- We can nest **`if`** statements inside of other **`if`** statements, arbitrarily deep
- Just like Java, there's no such thing as an **`else if`** statement
- But we can **pretend** there is because the entire **`if`** statement and the statement beneath it (and optionally a trailing **`else`**) are treated like a single statement

`switch` statements

- **`switch`** statements allow us to choose between many listed possibilities
- Execution will jump to the matching label or to **`default`** (if present) if none match
  - Labels must be constant (either literal values or **`#define`** constants)
- Execution will continue to fall through the labels until it reaches the end of the switch or hits a **`break`**
  - Don't leave out **`break`** statements unless you really mean to!

Anatomy of a switch statement

```
switch( data  )
{
        case constant1:
                statements1
        case constant2:
                statements2

        …
        case constantn:
                statementsn
        default:
                default statements
}
```

Example

- Use bitwise operations and selection statements to test if the 7th bit (starting from bit 0) in an integer value is a 1 or a 0

Upcoming

Next time…

- More control flow
  - Finish selection
  - Loops

Reminders

- Keep reading K&R chapter 3
- Start on Project 2
  - Form teams if you haven't already!
  - Due **next** Friday by midnight