Week 3 - Wednesday

CS222

Last time

- What did we talk about last time?
- Preprocessor directives
  - `#define`
  - `#include`
  - Conditional compilation
- `const` and `sizeof`
- Character I/O
- Lab 2

Questions?

Project 1

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

Other C Features

System limits

- The header **`limits.h`** includes a number of constants useful in C programming
- There are some for basic data types
- **`float.h`** has similar data for floating-point types, but it isn't as useful for us

Constant	Typical Value	Constant	Typical Value
SCHAR_MIN	-128	INT_MIN	-2147483648
SCHAR_MAX	127	INT_MAX	2147483647
UCHAR_MAX	255	UINT_MAX	4294967295
CHAR_MIN	-128	LONG_MIN	-2147483648
CHAR_MAX	127	LONG_MAX	2147483647
SHRT_MIN	-32768	ULONG_MAX	4294967295
SHRT_MAX	32767	CHAR_BIT	8
USHRT_MAX	65535		

Other limits

- `limits.h` has other system limits
- C and Linux have their roots in old school systems programming
- Everything is limited, but the limits are well-defined and accessible
- You may need to know:
  - How many files a program can have open at the same time
  - How big of an argument list you can send to a program
  - The maximum length of a pathname
  - Many other things…

Getting these limits

- For system limits, a minimum requirement for the maximum value is defined in **`limits.h`**
- If you want the true maximum value, you can retrieve it at runtime by calling **`sysconf()`** or **`pathconf()`** (defined in **`unistd.h`**) with the appropriate constant name

```
#include <stdio.h>
#include <unistd.h>


int main()
{
      long value = sysconf(_SC_LOGIN_NAME_MAX);
      printf("Maximum login name size: %d\n", value);

      return 0;
}
```

Examples of system limits

limits.h Constant	Minimum Value	sysconf() Name	Description
ARG_MAX	4096	**_SC_ARG_MAX**	Maximum bytes for arguments (**argv**) plus environment (**environ**) that can be supplied to an **exec()**
none	none	**_SC_CLK_TCK**	Unit of measurement for **times()**
LOGIN_NAME_MAX	9	**_SC_LOGIN_NAME_MAX**	Maximum size of a login name, including terminating null byte
OPEN_MAX	20	**_SC_OPEN_MAX**	Maximum number of file descriptors that a process can have open at one time, and one greater than maximum usable
none	1	**_SC_PAGESIZE**	Size of a virtual memory page
STREAM_MAX	8	**_SC_STREAM_MAX**	Maximum number of **stdio** streams that can be open at one time
NAME_MAX	14	**_PC_NAME_MAX**	Maximum number of bytes in a filename, excluding terminating null byte
PATH_MAX	256	**_PC_PATH_MAX**	Maximum number of bytes in a pathname, including terminating null byte

`char` values

- C uses one byte for a **`char`** value
- This means that we can represent the 128 ASCII characters without a problem
  - In many situations, you can use the full 256 extended ASCII sequence
  - In other cases, the (negative) characters will cause problems
- Let's see them!
- Beware the ASCII table!
  - Use it and die!

ASCII table

If you ever put one of these codes in your program, you deserve a zero.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Character values

```
#include <stdio.h>

int main()
{
    char c;
    for (c = 1; c != 0; c++)
        printf("%c\n", c);
    return 0;
}
```

Trouble with `printf()`

- There is nothing type safe in C
- What happens when you call **`printf()`** with the wrong specifiers?
  - Either the wrong types or the wrong number of arguments

```
printf("%d\n", 13.7);
printf("%x\n", 13.7);
printf("%c\n", 13.7);
printf("%d\n");
```

Format string practice

- What's the difference between `%x` and `%X`?
- How do you specify the minimum width of an output number?
  - Why would you want to do that?
- How do you specify a set number of places after the decimal point for floating-point values?
- What does the following format string say?
  - `"%6d 0x%04X\n"`

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 AND

- The bitwise AND operator (**&**) takes:
  - Integer representations **a** and **b**
- It produces an integer representation **c**
  - Its bits are the logical AND 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	0	0	0	1	1	0	0	c

```
char a = 46;
char b = 77;
char c = a & b; //12
```

Bitwise OR

- The bitwise OR operator (**|**) takes:
  - Integer representations **a** and **b**
- It produces an integer representation **c**
  - Its bits are the logical OR 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	1	1	1	1	c

```
char a = 46;
char b = 77;
char c = a | b; //111
```

Bitwise NOT

- The bitwise NOT operator (**~**) takes:
  - An integer representation **a**
- It produces an integer representation **b**
  - Its bits are the logical NOT of the corresponding bits in **a**
- Example using 8-bit **char** values:

~	0	0	1	0	1	1	1	0	a
	1	1	0	1	0	0	0	1	b

```
char a = 46;
char b = ~a; // -47
```

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 4 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 lowest 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

- 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 in C, illegal in Java
```

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 in C, illegal in Java
```

Selection

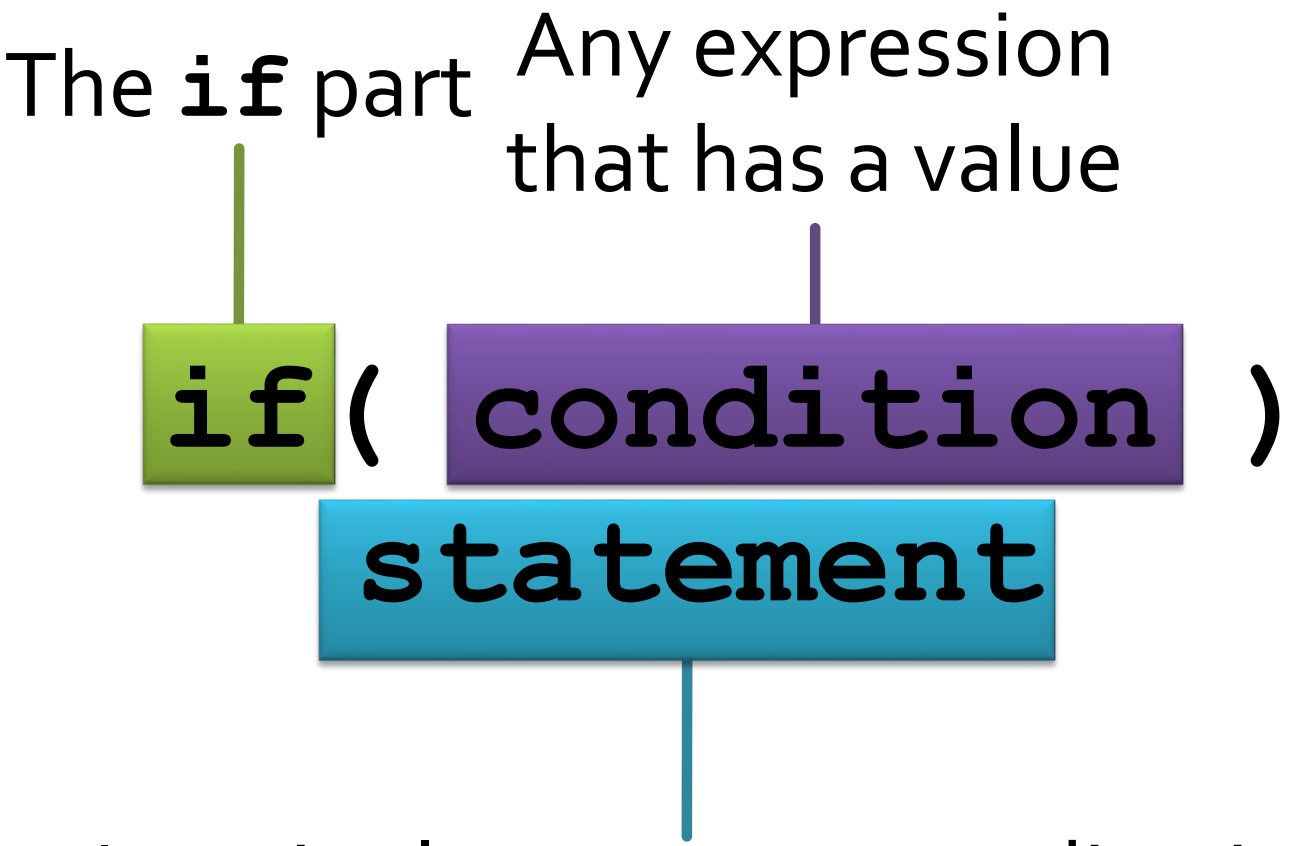
`if` statements

- 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

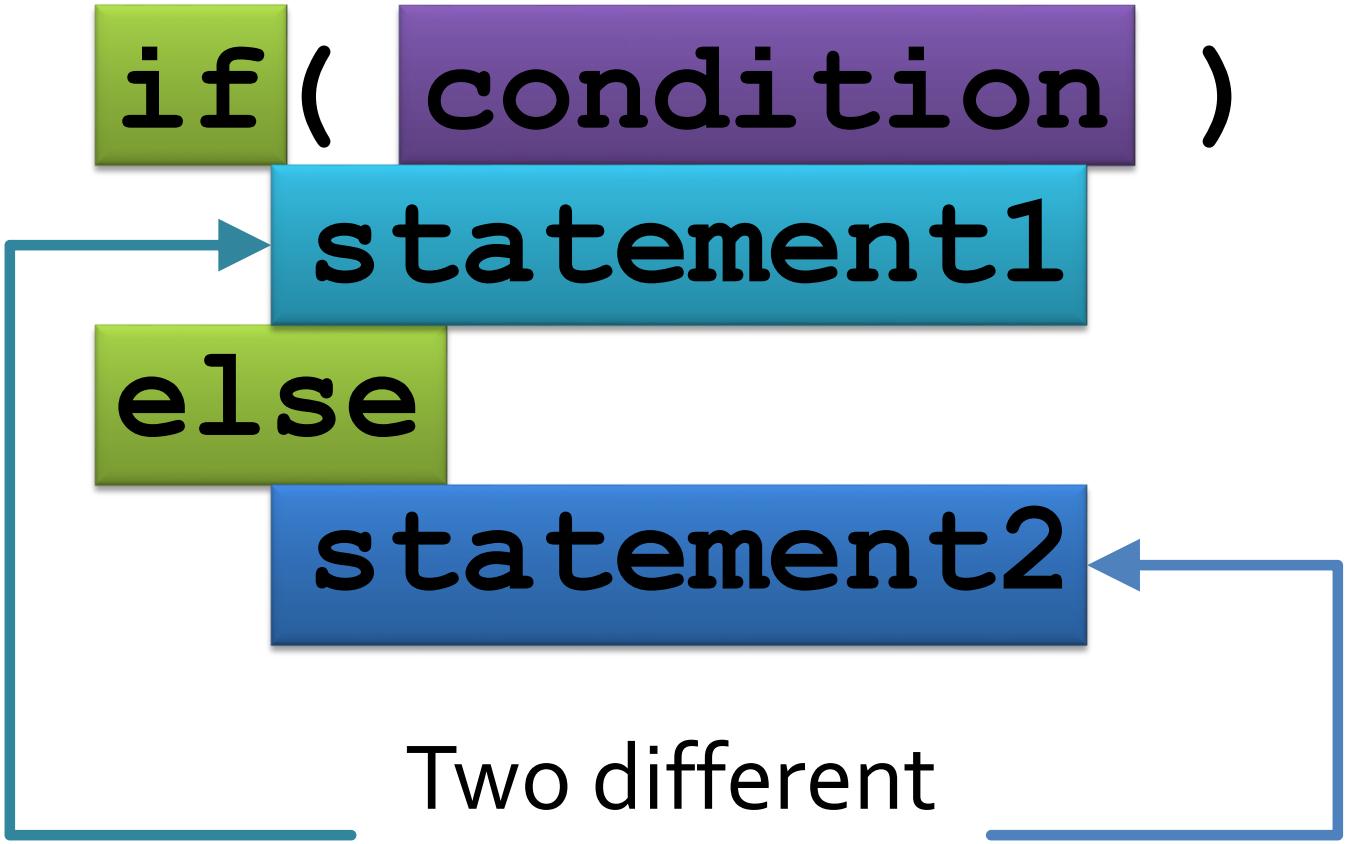
Sometimes this is natural and clear; at other times it can be cryptic.

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

Anatomy of an `if`



Anatomy of an `if-else`



Nesting

- We can nest **`if`** statements inside of other if statements, arbitrarily deep
- Just like Java, there is 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`**) is 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
 }
```

Quiz

Upcoming

Next time…

- Finish selection
- Loops
- Lab 3

Reminders

- Read K&R chapter 3
- Keep working on Project 1
  - Due Friday by midnight