Week 2 - Friday

COMP 2400

Last time

- What did we talk about last time?
- Math library
- Character I/O

Questions?

Project 1

Project 2

Quotes

It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so.

Mark Twain

Preprocessor Directives

Preprocessor directives

- There are preprocessor directives which are technically not part of the C language
- These are processed before the real C compiler becomes involved
- The most important of these are
  - **`#include`**
  - **`#define`**
  - Conditional compilation directives

#include

- You have already used **#include** before
  - **#include <stdio.h>**
- It can be used to include any other file
  - Use angle brackets (**< >**) for standard libraries
  - Use quotes (**" "**) for anything else
- It literally pastes the file into the document where the **#include** directive is
- Never **#include .c** files (executable code), only **.h** files (definitions and prototypes)
- It is possible to have a circular include problem

#define

- The primary way to specify constants in C is with a **#define**
- When you **#define** something, the preprocessor does a find-and-replace
  - Don't use a semicolon!
- **#define** directives are usually put close to the top of a file, for easy visibility

```
#define SIZE 100

int main()
{
      int array[SIZE];
      int i = 0;
      for (i = 0; i < SIZE; ++i)
            array[i] = i*i;

      return 0;
}
```

#define macros

- You can also make macros with **#define** that take arguments

```
#include <math.h>
#define TO_DEGREES(x) ((x) * 57.29578)
#define ADD(a,b) ((a) + (b))

int main()
{
	double theta = TO_DEGREES(2*M_PI);
	int value = ADD(5 * 2, 7);

	return 0;
}
```

- You need to be careful with parentheses
- Constants and macros are usually written in **ALL CAPS** to avoid confusion

Conditional compilation

- You can use directives **`#if`**, **`#ifdef`**, **`#ifndef`**, **`#else`**, **`#elif`** and **`#endif`**
- These are mostly used to avoid infinite include problems
- Sometimes they will change what gets compiled based on compiler version, system libraries, or other stuff

```
#ifndef SOMETHING_H
#define SOMETHING_H

int something(int a, int b);
#endif
```

Other C Features

sizeof

- We said that the size of **`int`** is compiler dependent, right?
  - How do you know what it is?
- **`sizeof`** is a built-in operator that will tell you the size of a data type or variable in bytes

```
#include <stdio.h>

int main()
{
	printf("%d", sizeof(char));
	int a = 10;
	printf("%d", sizeof(a));
	double array[100];
	printf("%d", sizeof(array));

	return 0;
}
```

const

- In Java, constants are specified with the **`final`** modifier
- In C, you can use the keyword **`const`**
- Note that **`const`** is only a suggestion
  - The compiler will give you an error if you try to assign things to **`const`** values, but there are ways you can even get around that

```
const double PI = 3.141592;
```

- Arrays have to have constant size in C
- Since you can dodge **`const`**, it isn't strong enough to be used for array size in C89
- That's why **`#define`** is more prevalent

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

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


int main()
{
      long value = sysconf(_SC_LOGIN_NAME_MAX);
      printf("Maximum login name size: %ld\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()
{
    for (char 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

Upcoming

Next time…

- Selection
- Loops

Reminders

- Read K&R chapter 3
- Finish Project 1
  - Due tonight by midnight!
- Start Project 2