Intermediate Programming

Day 14

Outline

- Binary file I/O
- Numerical representation
- Bitwise operations
- Review questions

Binary File I/O

When working with file handles we:

1. Create a file handle
2. Access the file's contents
3. Close the handle

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

When working with file handles we:

1. Create a file handle
   - `fopen` with the file-name and mode
     - "w" for (ASCII) write
     - "r" for (ASCII) read

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

When working with file handles we:

2. Access the file's contents
   - `fprintf` with file handle, format string, and values for (ASCII) write
   - `fscanf` with file handle, format string, and addresses for (ASCII read)

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

When working with file handles we:

3. Close the handle
   - `fclose` with file handle

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

If we write out a list of 100 ints to a file

Q: How big would the file be?

Q: How would we get the 7th value?

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

If we write out a list of 100 **int**s to a file

Q: How big would the file be?

A: 1056 bytes

We have 32 bits per **int**:

⇒ range of [0,4,000,000,000]

⇒ 9-10 decimal places (average)
+1 for the "**\n**"

⇒ $10 \times 100 - 11 \times 100$ bytes

⇒ Size is not fixed

But the values always require 400 bytes in memory!!!

```
>>./a.out
>> ls -l foo.txt
-rw-------. 1 misha users 1056 Mar 30 23:17 foo.txt
>>
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

If we write out a list of 100 **int**s to a file

Q: How would we get the 7th value?

A: 64 characters in

We have 32 bits per **int**

⇒ $10 \times 6 - 11 \times 6$ bytes

⇒ Offset is not fixed

✖ We cannot jump to the 7th value since we don't know the sizes of the values that come before

⇒**fscanf** one **int** at a time until we get the 7th value

```
>>./a.out
>> more foo.txt
1804289383
846930886
1681692777
1714636915
1957747793
424238335
719885386
...
>>
```

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0  ;i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.txt" , "w" );
        if( !fp ) return 1;
        for( int i=0 ; i<100 ; i++ ) fprintf( fp , "%u\n" , values[i] );
        fclose( fp );
}
```

Binary File I/O

- Until now, all files we've accessed in C have been plain *text files*
  - Write: Convert everything to a string of characters that is written to the file
  - Read: Convert everything from a string of characters that is read from the file

- Non-text files are known as *binary files* in C
  - Write: perform a bit-by-bit copy from memory to the file
  - Read: perform a bit-by-bit copy from the file to memory

Binary File I/O

As with text files, we:

1. Open the file
2. Access the file's contents
3. Close the file

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

```
FILE * fopen( const char *fileName , const char *mode );
```

1. Open the file
   - Use `fopen` to create a file handle:
     - `fileName`: name of the file
     - `mode`: mode of I/O
       - To open a file in binary mode, add the "b" flag in the string of `mode` characters*
     - Returns a file pointer (or NULL if the `fopen` failed)

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

*The file extension does not affect how the file is opened.

Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );
```

2. Access the file's contents
   - Use `fwrite` to write to a binary file:
     - `ptr`: starting address of data to write out
     - `sz`: size of a single data element
     - `count`: number of data elements
     - `fp`: file handle to write to
     - Returns the number of elements written

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

```
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

2. Access the file's contents
   - Use `fread` to read from a binary file:
     - `ptr`: starting address of data to read into
     - `sz`: size of a single data element
     - `count`: number of data elements
     - `fp`: file handle to read from
     - Returns the number of elements read

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

```
int fclose( FILE *fp );
```

3. Close the file
   - Use `fclose` to close the file handle
     - `fp`: file handle
     - Returns 0 if the stream was closed

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE *fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

If we write out a list of 100 ints to a file

Q: How big would the file be?

A: 400 bytes. Always!

```
>>./a.out
>> ls -l foo.dat
-rw-------. 1 misha users 400 Mar 30 23:17 foo.dat
>>
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
        unsigned int values[100];
        for( int i=0 ; i<100 ; i++ ) values[i] = rand();
        FILE* fp = fopen( "foo.dat" , "wb" );
        if( !fp ) return 1;
        fwrite( values , sizeof(int) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

As we read/write data, the `FILE` pointer tracks our position in the file:

- In some cases, we would like to change our position in the file:
  - Writing: To over-write something that was previously written
  - Reading: To jump to where the data we are interested in resides

Binary File I/O

```
int fseek( FILE *fp , long int offset , int whence );
```

- Use `fseek` to change the position of the file pointer
  - `fp`: file pointer
  - `offset`: number of bytes to move
    - Could be positive or negative, depending on whether we move forward or back
  - `whence`: where we move from:
    - `SEEK_SET`: beginning of the file
    - `SEEK_CUR`: current position
    - `SEEK_END`: end of the file
  - Returns zero if the change succeeded

Binary File I/O

main.c (part 1)

```
#include <stdio.h>
#include <stdlib.h>
unsigned int getValue( FILE *fp , size_t idx )
{
	if( fseek( fp , sizeof(unsigned int)*idx, SEEK_SET ) )
	{
		fprintf( stderr , "Failed to seek\n" );
		return -1;
	}
	unsigned int v;
	if( fread( &v , sizeof( int ) , 1 , fp )!=1 )
	{
		fprintf( stderr , "Failed to read\n" );
		return -1;
	}
	return v;
}
```

main.c (part 2)

```
int main( void )
{
	FILE *fp = fopen( "foo.dat" , "rb" );
	if( !fp )
	{
		fprintf( stderr , "Failed to open\n" );
		 return -1;
	}
	printf( "%u\n" , getValue( fp , 7 ) );
	fclose( fp );
}
```

Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

Note:

- The fread/fwrite functions only need to be able to read the bits/bytes from memory, they don't need to know the data-type stored.

⇒ We are not limited to reading/writing integers and numbers

main.c

```
#include <stdio.h>
typedef struct{ … } MyStruct;
int main( void )
{
        unsigned MyStruct values[100];
        FILE *fp = fopen( "foo.dat" , "rb" );
        if( !fp ) return 1;
        fread( values , sizeof(MyStruct) , 100 , fp );
        fclose( fp );
}
```

Binary File I/O

```
size_t fwrite( const void *ptr , size_t sz , size_t count , FILE *fp );
size_t fread( void *ptr , size_t sz , size_t count , FILE *fp );
```

Note:
If the **struct** contains pointers, the address is written out, not the contents at the address!!!

- The **fread/fwrite** functions only need to be able to read the bits/bytes from memory, they don't need to know the data-type stored.

⇒ We are not limited to reading/writing integers and numbers

main.c

```
#include <stdio.h>
typedef struct{ ... } MyStruct;
int main( void )
{
        unsigned MyStruct values[100];
        FILE *fp = fopen( "foo.dat" , "rb" );
        if( !fp ) return 1;
        fread( values , sizeof(MyStruct) , 100 , fp );
        fclose( fp );
}
```

Outline

- Binary file I/O
- Numerical representation
- Bitwise operations
- Review questions

Arithmetic

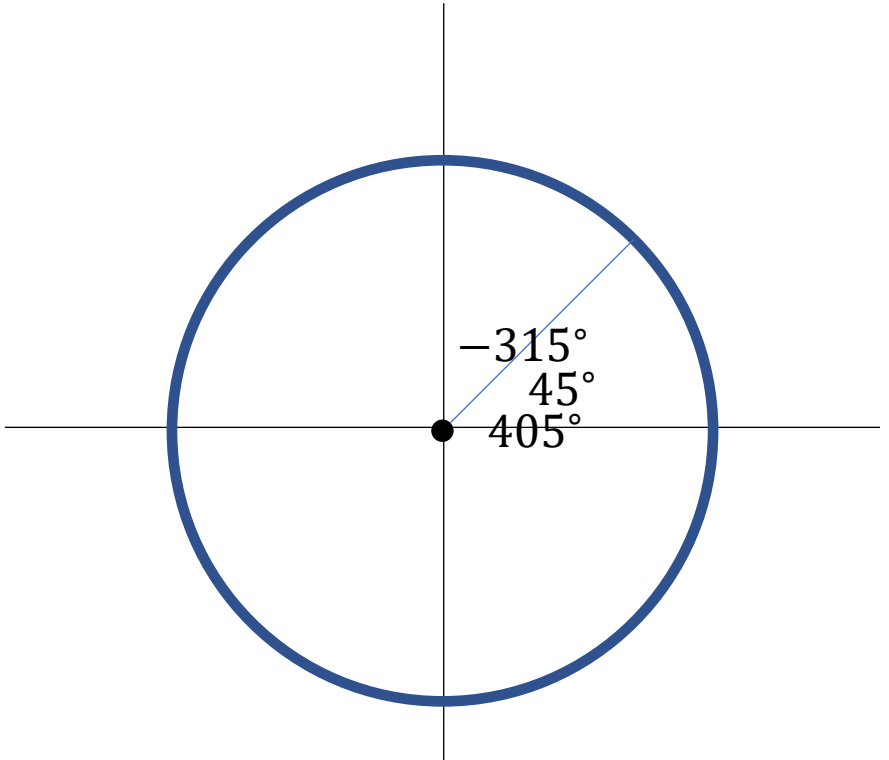
- The sets of integers is a set of numbers
- It has an addition operator, $+$, that takes a pair of integers and returns an integer
  - It contains a zero element, 0, with the property that adding zero to any integer gives back that integer:
$$a + 0 = a$$
  - Every integer a has an inverse $-a$ such that the sum of the two is zero:
$$a + (-a) = a - a = 0$$

Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

  - Degrees in a circle (mod $360°$)
  - Hours on a clock (mod 12)



Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can represent integers mod M using values in the range $[0, M)$
  - While an integer is bigger than or equal to M, repeatedly subtract M
  - While an integer is less than zero, repeatedly add M

Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can represent integers mod M using values in the range $[0, M)$
- Or, we can represent integers mod M using the range $[-10, M - 10)$
- Or, we can represent integers mod M using the range $\left[-\frac{M}{2}, \frac{M}{2}\right)$

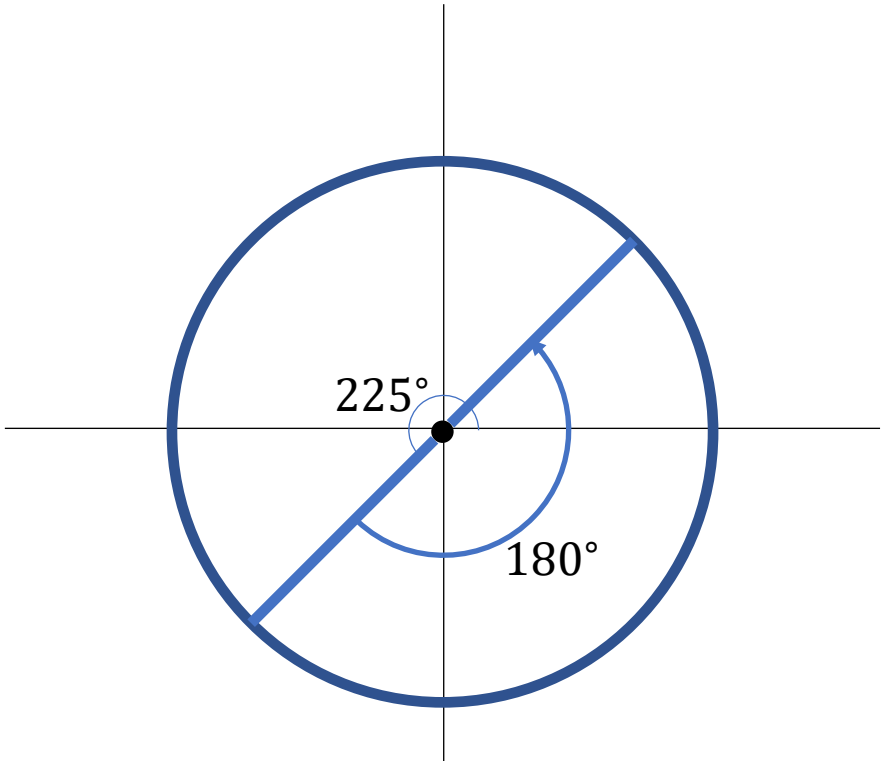
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ + 180^\circ = 405^\circ$$



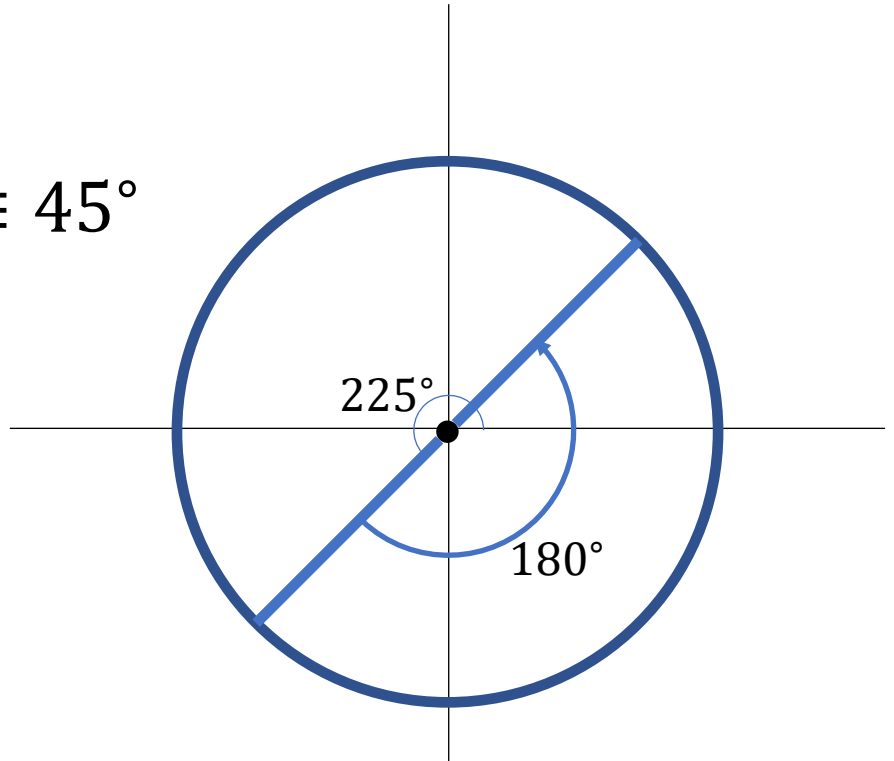
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ + 180^\circ = 405^\circ \equiv 45^\circ$$



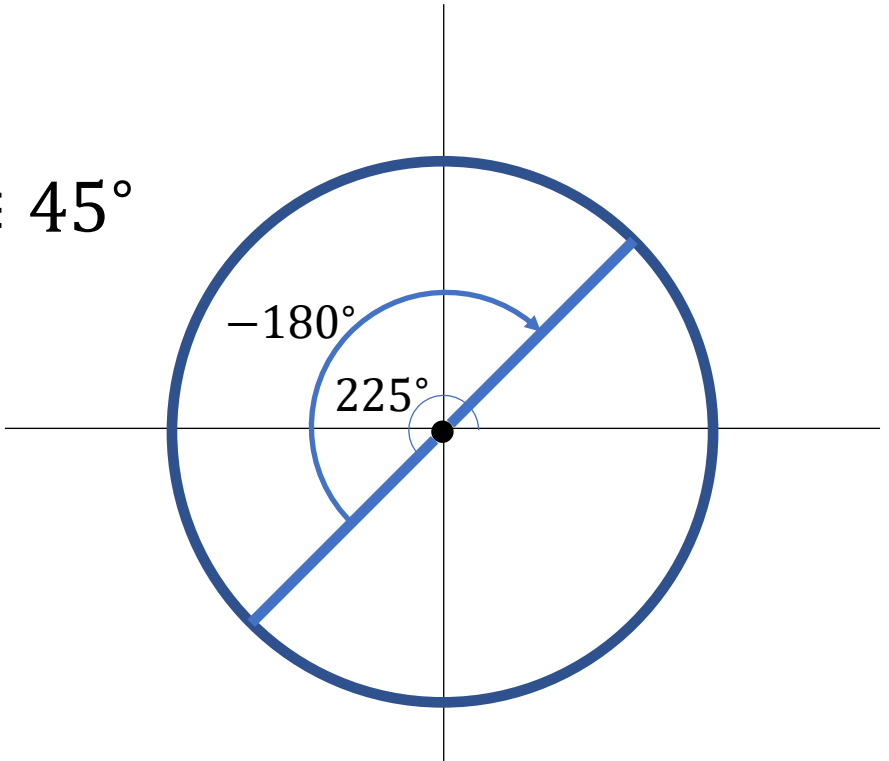
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M:

$$225^\circ - 180^\circ \equiv 225^\circ + 180^\circ = 405^\circ \equiv 45^\circ$$



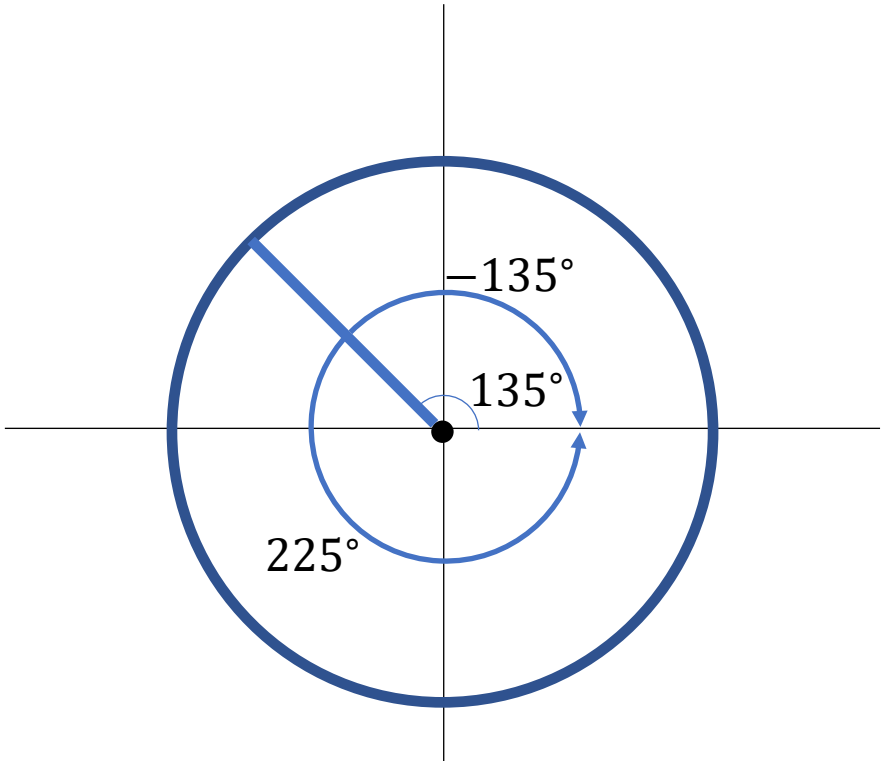
Modular arithmetic

- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$



Modular arithmetic

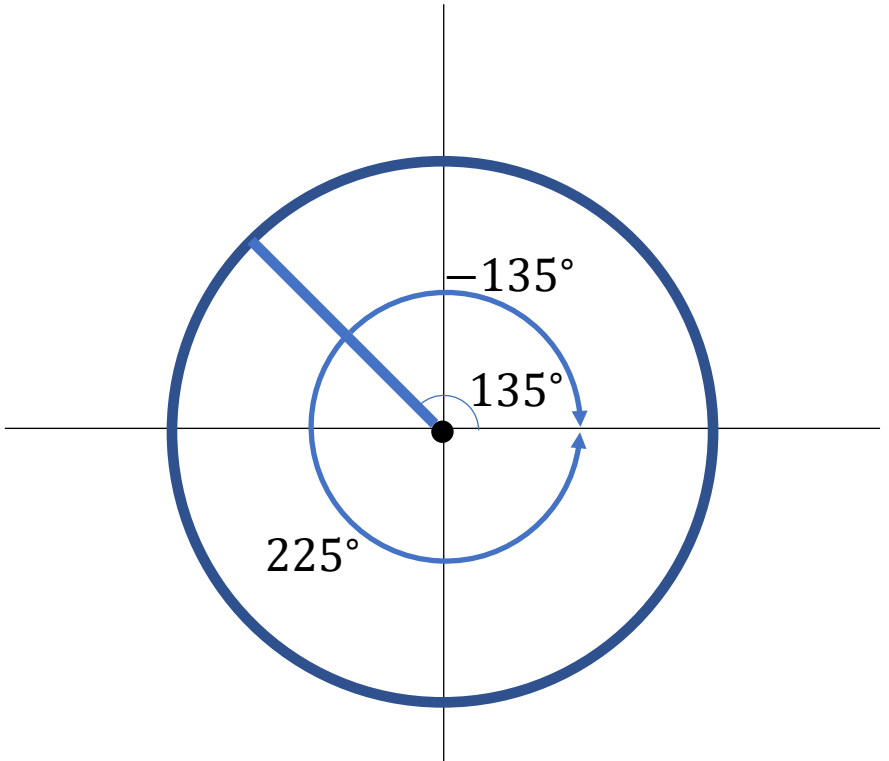
- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M$$



Modular arithmetic

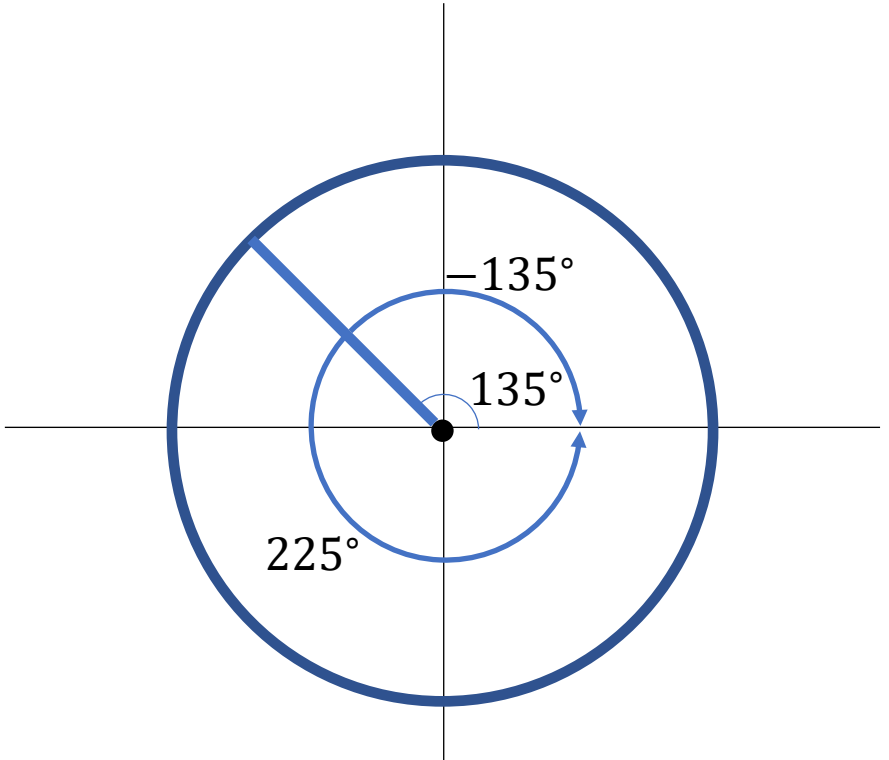
- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M = M$$



Modular arithmetic

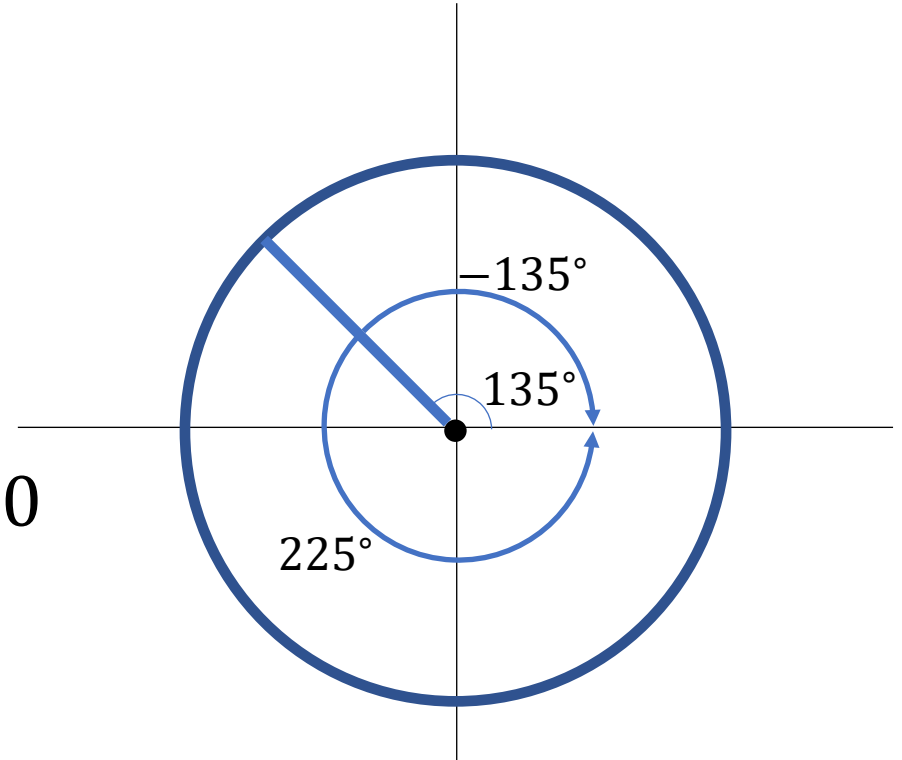
- Given a positive integer, M, we say that two integers a and b are equivalent modulo M, if there is exists some integer k such that:

$$a \equiv b + k \cdot M$$

- We can add numbers modulo M

- For any integer a, the negative of a modulo M can be represented by $M - a$:

$$a + (M - a) = (a - a) + M = M \equiv 0$$



Bases (decimal)

- When we write out an integer in decimal notation, we are representing it as a sum of “one”s, “ten”s, “hundred”s, etc.

$$365 = 3 \times 100 + 6 \times 10 + 5 \times 1$$
$$= 3 \times 10^2 + 6 \times 10^1 + 5 \times 10^0$$

- This is unique because each digit is in the range 0 to 9, written $[0,10)$
- Since 10^1, 10^2, ... are divisible by 10, we can check if a number is divisible by 10 by checking if the coefficient in the one’s place is zero
  - Since 10^1, 10^2, ... are divisible by 2, we can check if a number is divisible by 2 by checking if the coefficient in the one’s place is divisible by 2
  - Since 10^1, 10^2, ... are divisible by 5, we can ...

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we carry

$$\begin{array}{rrrr} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we carry

$$\begin{array}{rrrr} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & & & 8 \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we carry

$$\begin{array}{rccc} & {\scriptstyle 1} & & \\ & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & & 3 & 8 \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we carry

$$\begin{array}{cccc} 1 & 1 & & \\ & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & 0 & 3 & 8 \end{array}$$

Bases (decimal)

- We add two numbers by adding the digits from smallest to largest
  - If the sum of digits falls outside the range $[0,10)$ we carry

$$\begin{array}{cccc} 1 & 1 & & \\ & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline 1 & 0 & 3 & 8 \end{array}$$

Bases (decimal)

Q: If we use three digits, how many numbers can we represent?

A: $1000 = 10^3$ (including zero)

Note:

- The sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rcccc} & & 3 & 6 & 5 \\ + & & 6 & 7 & 3 \\ \hline & 1 & 0 & 3 & 8 \end{array}$$

Bases (decimal)

Q: If we use three digits, how many numbers can we represent?

A: $1000 = 10^3$ (including zero)

Note:

- The sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rccc} & 3 & 6 & 5 \\ + & 6 & 7 & 3 \\ \hline & 0 & 3 & 8 \end{array}$$

- If we only use three digits, we lose the leading digit to overflow
- This is the same as the number mod 10^3

Bases (general)

- We can use other bases to represent numbers, writing

$$(s_3 s_2 s_1 s_0)_b = s_3 \times b^3 + s_2 \times b^2 + s_1 \times b^1 + s_0 \times b^0$$

  where b is the base and s_0, s_1, s_2, s_3 are digits in the range $[0, b)$

- Since b^1, b^2, ... are divisible by b, we can check if a number is divisible by b by checking if the coefficient in the one's place is zero
  - If a divides b, we can check if a divides the number by checking if the coefficient in the one's place is divisible by a

"Base eight is just like base ten really, if you're missing two fingers."
-- Tom Lehrer

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{rcccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (& & &)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{rcccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (& & & 8)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{rcccc} & & 1 & & \\ & & (3 & 6 & 5)_9 \\ + & & (6 & 7 & 3)_9 \\ \hline & (& & 4 & 8)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{cccc} 1 & 1 & & \\ & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (& 1 & 4 & 8)_9 \end{array}$$

Bases (general)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum falls outside the range $[0, b)$:

$$\begin{array}{cccc} 1 & 1 & & \\ & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (1 & 1 & 4 & 8)_9 \end{array}$$

Bases (general)

Q: If we use three digits in base b, how many numbers can we represent?

A: b^3 (including zero)

Note:

- As before, the sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rcccc} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline (1 & 1 & 4 & 8)_9 \end{array}$$

Bases (general)

Q: If we use three digits in base b, how many numbers can we represent?

A: b^3 (including zero)

Note:

- As before, the sum of two numbers represented using three digits may require four digits to store:

$$\begin{array}{rrrr} & (3 & 6 & 5)_9 \\ + & (6 & 7 & 3)_9 \\ \hline & (1 & 4 & 8)_9 \end{array}$$

- If we only use three digits, we lose the leading digit to overflow
- This is the same as the number mod b^3

Bases (decimal)

- Given a number in base 10:

$$16{,}384 = 1 \times 10^4 + 6 \times 10^3 + 3 \times 10^2 + 8 \times 10^1 + 4 \times 10^0$$

we can get an expression of the number in base 100 by grouping digits:

$$16{,}384 = 1 \times 100^2 + 63 \times 100^1 + 84 \times 100^0$$

Similarly, we can get an expression of the number in base 1000, etc.

Bases (general)

- Similarly, given a number in base b:

$$(s_3 s_2 s_1 s_0)_b = s_3 \times b^3 + s_2 \times b^2 + s_1 \times b^1 + s_0 \times b^0$$

we can get an expression of the number in base b^2 by grouping digits:

$$(s_3 s_2 s_1 s_0)_b = (s_3 \times b + s_2) \times \left(b^2\right)^1 + (s_1 \times b + s_0) \times \left(b^2\right)^0$$

Similarly, we can get an expression of the number in base b^3, etc.

Bases (in the wild)

- Decimal (base 10)
  - We have ten fingers

- Sexagesimal (base 60):
  - Minutes / seconds
  - Easy to tell if a number is divisible by 2, 3, 4, 5, 6 , 10, 12, 15, or 30
  - Dates back to the Babylonians

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)
  - Numbers in a computer ($16 = 2^4$)
    - We can easily convert binary to hex by grouping sets of four digits
    - We get a more compact representation, replacing 4 digits with 1

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)

  Q: How should we separate the digits? $(115)_{16}$
  - $(115)_{16} = 1 \times 16^2 + 1 \times 16^1 + 5 \times 16^0$
  - $(115)_{16} = 1 \times 16^1 + 15 \times 16^0$
  - $(115)_{16} = 11 \times 16^1 + 5 \times 16^0$

Bases (in the wild)

- Binary (base 2)
  - Numbers in a computer

- Hexadecimal a.k.a. hex (base 16)

  Q: How should we separate the digits? $(115)_{16}$

  A: Use numbers and letters:

  - $\{0,1,2,3,4,5,6,7,8,9\}$ to represent numbers in the range $[0,10)$
  - $\{a, b, c, d, e, f\}$ to represent values in the range $[10,16)$:
    - $(115)_{16} = 1 \times 16^2 + 1 \times 16^1 + 5 \times 16^0$
    - $(1f)_{16} = 1 \times 16^1 + 15 \times 16^0$
    - $(b5)_{16} = 11 \times 16^1 + 5 \times 16^0$

Representing integers

- On most machines, [unsigned] ints are represented using 4 bytes*
  - Each byte is composed of 8 bits

  ⇒ An [unsigned] int is represented by 32 bits
  - Each bit can be either “on” or “off”

  ⇒ An [unsigned] int is represented in binary using 32 digits with values 0 or 1

  ⇒ An [unsigned] int can have one of 2^{32} values

*“[...]” notation indicates an optional argument

Representing integers

- On most machines, [unsigned] ints are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An [unsigned] int is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An [unsigned] int is represented in binary using 32 digits with values 0 or 1

  ⇒ An [unsigned] int can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 30;
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
30
>>
```

On the machine, a is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

Representing integers

- On most machines, [unsigned] ints are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An [unsigned] int is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An [unsigned] int is represented in binary using 32 digits with values 0 or 1

  ⇒ An [unsigned] int can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 0x1e;
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
30
>>
```

On the machine, a is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

- You can assign using base 16 by preceding the number with 0x to indicate hex

Representing integers

- On most machines, [unsigned] ints are represented using 4 bytes
  - Each byte is composed of 8 bits

  ⇒ An [unsigned] int is represented by 32 bits
  - Each bit can be either "on" or "off"

  ⇒ An [unsigned] int is represented in binary using 32 digits with values 0 or 1

  ⇒ An [unsigned] int can have one of 2^{32} values

```
#include <stdio.h>
int main( void )
{
        int a = 30;
        printf( "%x\n" , a );
        return 0;
}
```

```
>> ./a.out
1e
>>
```

On the machine, a is assigned the value:

$$a \leftarrow (\ 00000000\ 00000000\ 00000000\ 00011110\ )_2$$

$$a \leftarrow (\ 00\ 00\ 00\ 1e)_{16}$$

- You can assign using base 16 by preceding the number with 0x to indicate hex
- You can print the base 16 representation by using %x for formatting

Representing integers

- On most machines, [unsigned] chars are represented using 1 byte
  ⇒ A [unsigned] char can have one of 2^8 values
- On most machines, [unsigned] long ints are represented using 8 bytes
  - A [unsigned] long int can have one of 2^{64} values

Representing integers*

⇒ An [unsigned] char can have one of $2^8 = 256$ values

⇒ [unsigned] chars are integer values mod 2^8

- unsigned char: We will use the range $[0,256)$ to represent integers
- char: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?
Integers mod 2^8 are integers mod 2^8, regardless of the representation!!!

*For simplicity the following discussion will focus on chars, though it holds for other integer representations (e.g. ints and long ints)

Representing integers

- unsigned char: We will use the range $[0,256)$ to represent integers
- char: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?

A: Is $125 < 129 \bmod 256$?

Since $129 \equiv -127 \bmod 256$,
it depends on the range we use

```
#include <stdio.h>
int main( void )
{
        unsigned char c1 = 125 , c2 = 129;
        printf( "%d\n" , c1<c2 );
        return 0;
}
```

```
>> ./a.out
1
>>
```

Representing integers

- unsigned char: We will use the range $[0,256)$ to represent integers
- char: We will use the range $[-128,128)$ to represent integers

Q: What's the difference?

A: Is $125 < 129 \bmod 256$?

Since $129 \equiv -127 \bmod 256$,
it depends on the range we use

```
#include <stdio.h>
int main( void )
{
        char c1 = 125 , c2 = 129;
        printf( "%d\n" , c1<c2 );
        return 0;
}
```

```
>> ./a.out
0
>>
```

Representing integers

- Addition:
  We add two numbers, $a + b$, by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$\begin{array}{r} {\scriptstyle 1\,1\quad\;\; 11} \\ (11010011)_2 \\ +(01000110)_2 \\ \hline (1\,00011001)_2 \end{array}$$
$$= (00011001)_2$$

Representing integers

- Addition:
  We add two numbers, $a + b$. by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$\begin{array}{r} {\scriptstyle 1\,1\quad\;\; 11} \\ (11010011)_2 \\ +(01000110)_2 \\ \hline (00011001)_2 \end{array}$$

Q: What about subtraction, $a - b$?

Representing integers

- Addition:
  We add two numbers, $a + b$. by adding the digits from smallest to largest
  - We carry as necessary
  - And we cut off at 8 bits

$$\begin{array}{r} {\scriptstyle 1\,1\quad\;\;11} \\ (11010011)_2 \\ +(01000110)_2 \\ \hline (00011001)_2 \end{array}$$

Q: What about subtraction, $a - b = a + (-b)$?

Equivalently, how do we define the negative of a number?

Negation

- Recall:
  The negative of an integer is the number we would have to add to get back zero.
- Defining negative one:
  - Mod 256, we have $-1 \equiv 255 = (11111111)_2$

Negation

- Recall:
  The negative of an integer is the number we would have to add to get back zero.
- Defining negatives in general:
  1. Given a binary value in 8 bits:
     $$(10011101)_2$$
  2. We can flip the bits:
     $$(01100010)_2$$
  3. Adding the two values we get $255 \equiv -1$:
     $$(11111111)_2$$
  4. Adding one to that we get 0

Negation

- Recall:
  The negative of an integer is the number we would have to add to get back zero.
- 2's complement:
  To get the binary representation of the negative of a number
  1. Flip the bits
  2. Add 1

Floating point value representation

$$\pm b \times 2^{e}$$

- On most machines, `float`s are represented using 4 bytes (32 bits)
  - These are (roughly) used to encode:
    - The sign ($\pm$): 1 bit
    - The signed (integer) exponent (e): 8 bits*
    - The unsigned (integer) base (b): 23 bits

*This is what makes the point float

Floating point value representation

[WARNING]:

- Adding floating point values requires aligning their precisions first*

$$b_1 \times 2^{e_1} + b_2 \times 2^{e_2}$$
$$\Downarrow$$
$$(b_1 \times 2^{e_1 - e_2}) \times 2^{e_2} + b_2 \times 2^{e_2}$$
$$\Downarrow$$
$$(b_1 / 2^{e_2 - e_1} + b_2) \times 2^{e_2}$$

⇒ If b_1 is less than $2^{e_2 - e_1}$, then $b_1 / 2^{e_2 - e_1}$ will become zero

⇒ Addition of floating points may not be associative:

$$(a + b) + c \; ! = a + (b + c)$$

*Assume $e_2 > e_1$

Floating point value representation

[WARNING]:

- Adding floating point values requires aligning their precisions first*

```
#include <stdio.h>
int main( void )
{
        float a = 1e-4f , b = 1e+4f , c = -b;
        printf( "%.3e %.3e %.3e\n" , a , b , c );
        printf( "%.3e %.3e\n" , ( a + b ) + c , a + ( b + c ) );
        return 0;
}
```

```
>> ./a.out
1.000e-04 1.000e+04 -1.000e+04
0.000e+00 1.000e-04
>>
```

⇒ If $b_1 \times$ [illegible] zero

⇒ Addition of floating [illegible] $+ c)$

*Assume $e_2 > e_1$

Floating point value representation

$$\pm b \times 2^e$$

- On most machines, **double**s are represented using 8 bytes (64 bits)
  - These are (roughly) used to encode:
    - The sign ($\pm$): 1 bit
    - The signed (integer) exponent (e): 11 bits
    - The unsigned (integer) base (b): 52 bits

Outline

- Binary file I/O
- Numerical representation
- Bitwise operations
- Review questions

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
  - This is equivalent to multiplying by 2^k
  - Note that the new, right-most, bits are set to 0
  - Once shifted out, the left-most bits are lost

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = a << 2;    // (00010100)_2
        printf( "%d\n" , b );
        return 0;
}
```

```
>> ./a.out
20
>>
```

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
  - This is equivalent to dividing by 2^k
  - Note that the new, left-most, bits are set to 0
  - Once shifted out, the right-most bits are lost

```
#include <stdio.h>
int main( void )
{
        char a = 5;        // (00000101)_2
        char b = a >> 2;   // (00000001)_2
        printf( "%d\n" , b );
        return 0;
}
```

```
>> ./a.out
1
>>
```

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
  - The corresponding bit in the output is 1 if both bits are 1 in the input

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = 14;        // (00001110)_2
        char c = a & b;     // (00000100)_2
        printf( "%d\n" , a & b );
        return 0;
}
```

```
>> ./a.out
4
>>
```

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
  - The corresponding bit in the output is 1 if either (or both) bits are 1 in the input

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = 14;        // (00001110)_2
        char c = a | b;     // (00001111)_2
        printf( "%d\n" , a | b );
        return 0;
}
```

```
>> ./a.out
15
>>
```

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
- n ^ m: compute the bit-wise *exclusive or* of n and m
  - The corresponding bit in the output is 1 if an odd number of bit are 1 in the input

```
#include <stdio.h>
int main( void )
{
        char a = 5;       // (00000101)_2
        char b = 14;      // (00001110)_2
        char c = a ^ b;   // (00001011)_2
        printf( "%d\n" , c );
        return 0;
}
```

```
>> ./a.out
11
>>
```

Bit-wise operations: Integer types only

- n << k: shifts n to the left by k positions
- n >> k: shifts n to the right by k positions
- n & m: compute the bit-wise *and* of n and m
- n | m: compute the bit-wise *or* of n and m
- n ^ m: compute the bit-wise *exclusive or* of n and m
- ~n: flip the bits of n
  - The corresponding bit in the output is 1 if it is 0 in the input

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = ~a;        // (11111010)_2
        char c = b + 1;     // (11111011)_2
        printf( "%d\n" , c );
        return 0;
}
```

```
>> ./a.out
-5
>>
```

Bit-wise operations: Integer types only

- There are also variants of these that evaluate-and-set
  - n <<= k
  - n >>= k
  - n &= m
  - n |= m
  - n ^= m

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        a <<= 3;            // (00101000)_2
        printf( "%d\n" , a );
        return 0;
}
```

```
>> ./a.out
40
>>
```

Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char mask = 1<<2;// (00000100)_2
        char c = a & mask;
        printf( "%d %d\n" , c , c!=
        return 0;
}
```

```
>> ./a.out
4 1
>>
```

Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = a>>2;      // (00000001)_2
        char c = b & 1;
        printf( "%d %d\n" , c , c!
        return 0;
}
```

```
>> ./a.out
1 1
>>
```

Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &
  - Note:

    Integers in $[0,128)$ all have a binary representation of the form:

    $$(0\ *******)$$

    Integers in $[128,256) \equiv [-128,0)$ all have a binary representation of the form:

    $$(1\ *******)$$

Bit-wise operations: Integer types only

- Masking
  - We can determine if a bit is on or off using << and &
  - Or we can use >> and &
  - We can determine the sign by testing the highest (a.k.a. most significant) bit

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = -4;        // (11111100)_2
        char mask = 1<<7;// (10000000)_2
        printf( "%d %d\n" , ( a & mask )!=0 , ( b & mask )!=0 );
        return 0;
}
```

```
>> ./a.out
0 1
>>
```

Bit-wise operations: Integer types only

- Masking
  - We can
  - Or we ca
  - We can ificant) bit*

Note:
We set the mask to 1<<7 because chars are 8 bits long.
For ints we would use a mask of 1<<31.
Etc.

```
#include <stdio.h>
int main( void )
{
        char a = 5;         // (00000101)_2
        char b = -4;        // (11111100)_2
        char mask = 1<<7;// (10000000)_2
        printf( "%d %d\n" , ( a & mask )!=0 , ( b & mask )!=0 );
        return 0;
}
```

```
>> ./a.out
```

*This assumes that the most significant bit is on the left.
It's true for most (big-endian) machines but should not be assumed.

Outline

- Binary file I/O
- Numerical representation
- Bitwise operations
- Review questions

Review questions

1. How do we read/write binary files in C?

Review questions

2. What character represents the bitwise XOR operation? How does it differ from the OR operation?

Review questions

3. What happens if you apply the bitwise operation on an integer value? (extra: what if we apply to floats)

Review questions

4. What is the result of `(15>>2) || 7`?

Review questions

5. What is the result of `(15>>2) | 7`?

Exercise 5-2

- Website -> Course Materials -> ex5-2