

# Intermediate Programming

## Day 15

# Outline

- Exercise 14
- Numerical representation
- Casting
- Review questions

# Exercise 14

Convert `char *` message to  
`int` number

*encrypt.c*

```
...
int str_to_int( char msg[] , int len )
{
    int num = 0;
    if( len>32 )
    {
        fprintf( stderr , "[WARNING] Insufficient bits\n" );
        len = 32;
    }
    for( int i=0 ; i<len ; i++ ) if( msg[len-i-1]=='1' ) num += pow( 2 , i );
    return num;
}
...
```

# Exercise 14

Convert `char *` message to `int` number

- Note that  $2^i = 1 \ll i$

*encrypt.c*

```
...
int str_to_int( char msg[] , int len )
{
    int num = 0;
    if( len > 32 )
    {
        fprintf( stderr , "[WARNING] Insufficient bits\n" );
        len = 32;
    }
    for( int i=0 ; i<len ; i++ ) if( msg[len-i-1]=='1' ) num += 1<<i;
    return num;
}
...
```

# Exercise 14

Convert `char *` message to `int` number

- Note that  $2^i = 1 \ll i$
- Note that if  $i$  and  $j$  are variables with no common bits turned on ( $i \& j == 0$ ) then  $i + j = i | j$

*encrypt.c*

```
...
int str_to_int( char msg[] , int len )
{
    int num = 0;
    if( len > 32 )
    {
        fprintf( stderr , "[WARNING] Insufficient bits\n" );
        len = 32;
    }
    for( int i=0 ; i<len ; i++ ) if( msg[len-i-1]=='1' ) num |= 1<<i;
    return num;
}
...
```

# Exercise 14

Convert `int` number to  
`char *` message

*encrypt.c*

```
...
void int_to_str( int num_encrypted , char msg_encrypted[] , int len )
{
    for( int i=0 ; i<len ; i++ )
    {
        if( num_encrypted&1 ) msg_encrypted[len-i-1] = '1';
        else                  msg_encrypted[len-i-1] = '0';
        num_encrypted >>= 1;
    }
    if( num_encrypted )
        fprintf( stderr , "[WARNING] Insufficient bits\n" );
}
...
```

# Exercise 14

Compute the encrypted message by repeatedly left-shifting the message by 1 and XORing.

*encrypt.c*

```
...
int main( void )
{
    int num_msg = 0;
    char msg[33] = {'\0'};
    int n = -1;
    ...

    int num_encrypted = 0;

    for( int i=0 ; i<n ; i++ ) num_encrypted ^= num_msg<<i;
    ...

    return 0;
}
```

# Outline

- Exercise 14
- Numerical representation
- Casting
- Review questions



# Arithmetic

- The integers are a set (of numbers)
- There is an addition operator,  $+$ , that takes a pair of integers and returns an integer
  - There is 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:

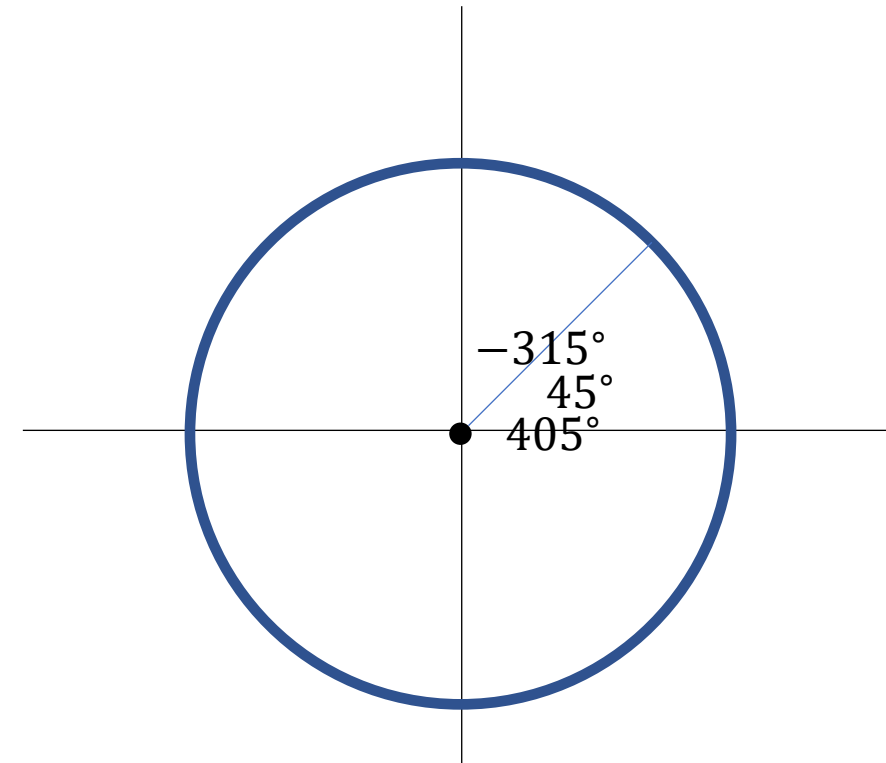
$$a + (-a) = a - a = 0$$

# Modular arithmetic

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

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

- Degrees in a circle (mod  $360^\circ$ )
- 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 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 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  $\left[-\frac{M}{2}, \frac{M}{2}\right)$

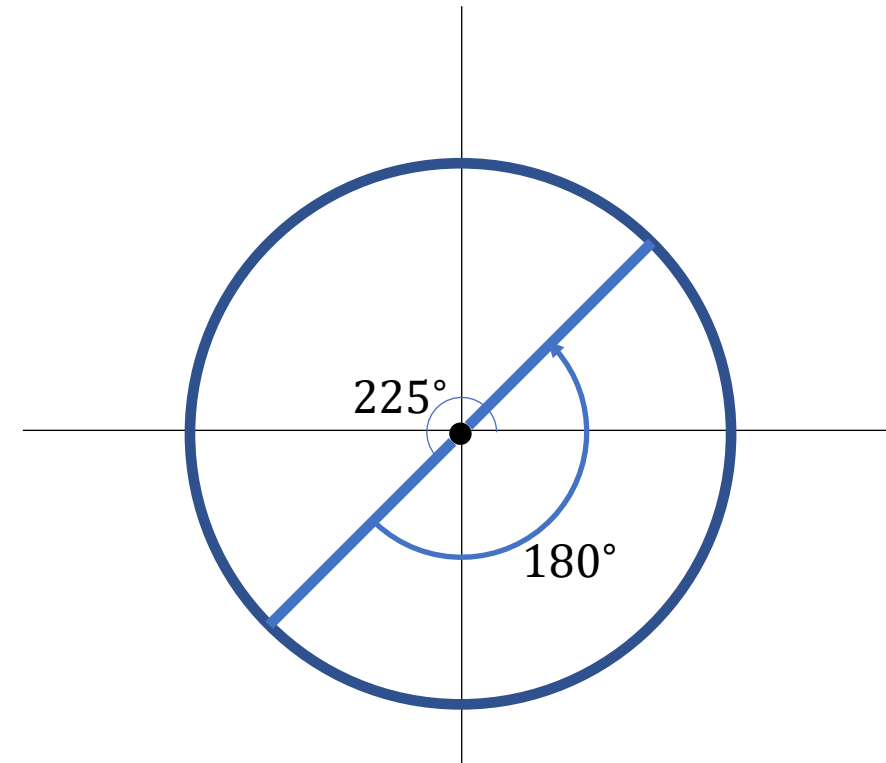
# Modular arithmetic

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

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

- We can add numbers modulo  $M$ :

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



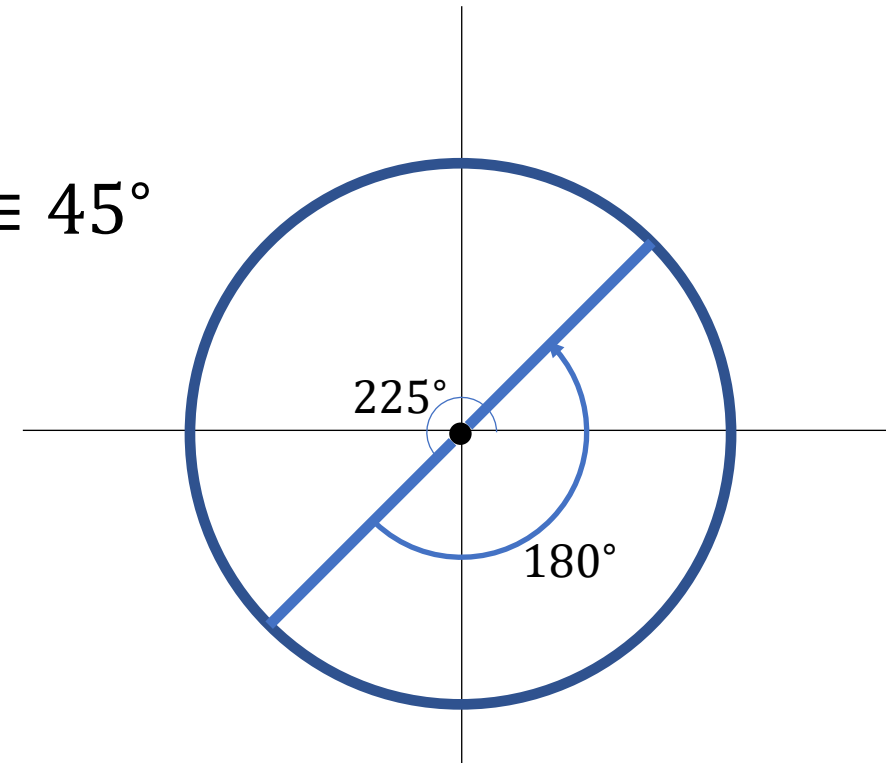
# Modular arithmetic

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

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

- We can add numbers modulo  $M$ :

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



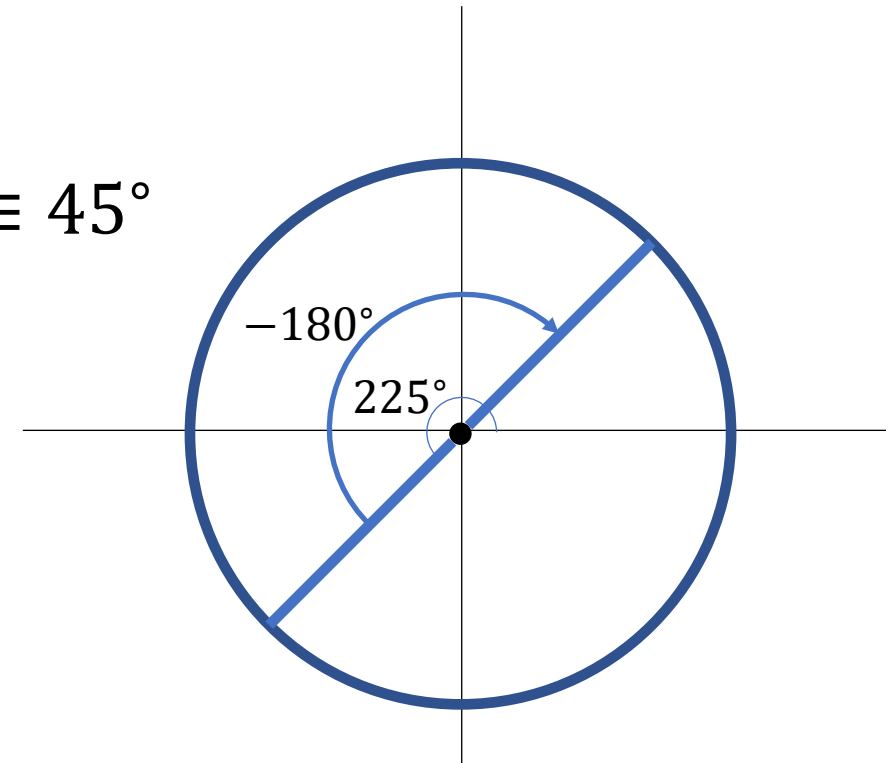
# Modular arithmetic

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

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

- We can add numbers modulo  $M$ :

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

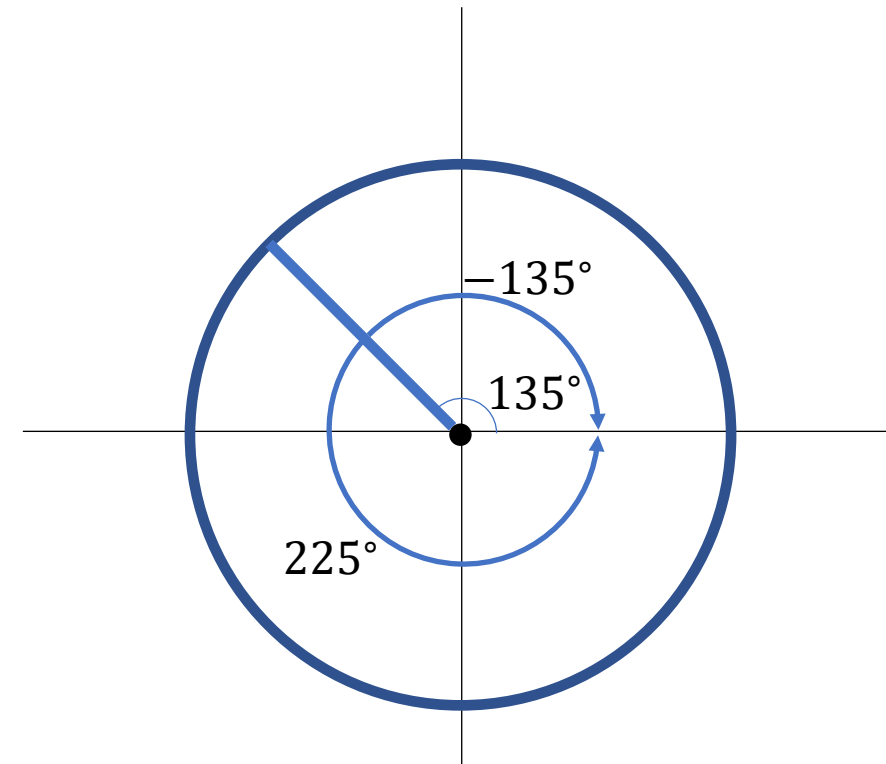


# Modular arithmetic

- Given a positive integer,  $M$ , we say that two integers  $a$  and  $b$  are equivalent modulo  $M$ , if there 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  $M - a$





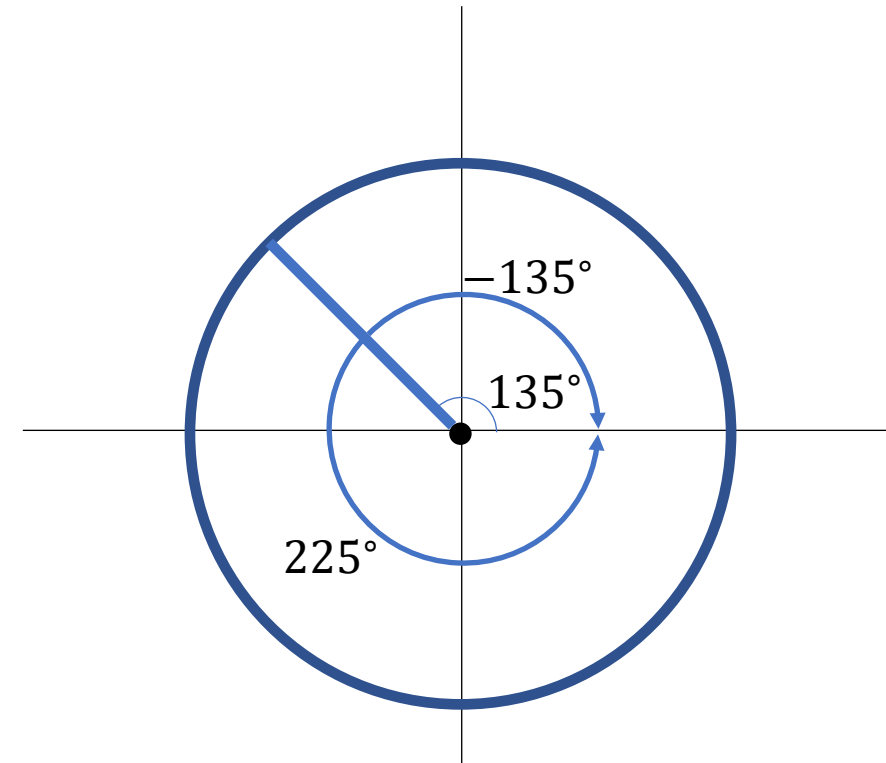
# Modular arithmetic

- Given a positive integer,  $M$ , we say that two integers  $a$  and  $b$  are equivalent modulo  $M$ , if there 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  $M - a$ :

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



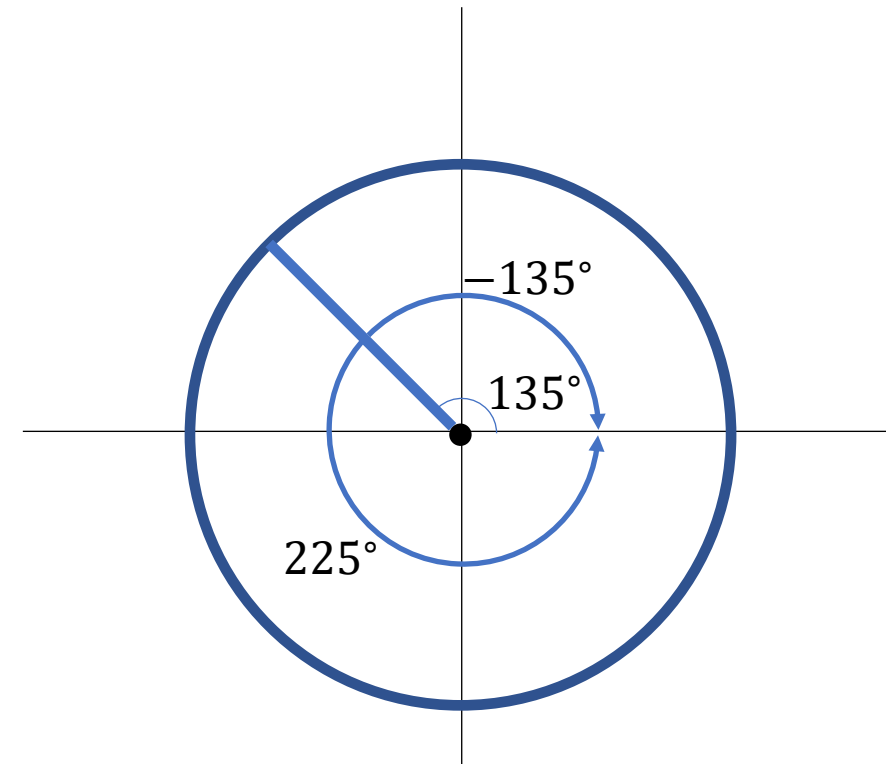
# Modular arithmetic

- Given a positive integer,  $M$ , we say that two integers  $a$  and  $b$  are equivalent modulo  $M$ , if there 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  $M - a$ :

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



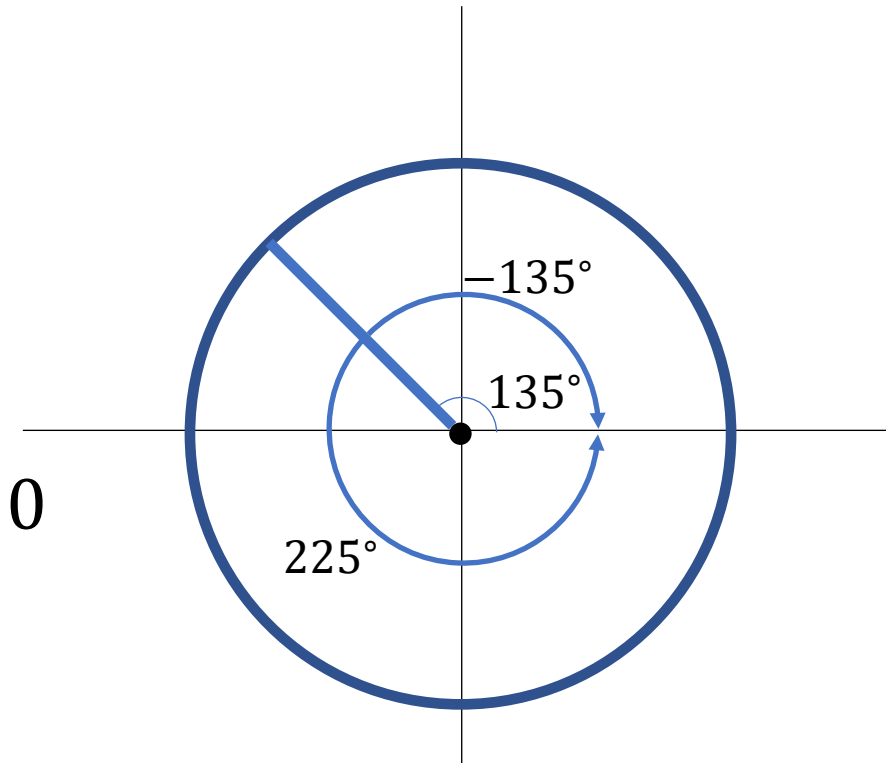
# Modular arithmetic

- Given a positive integer,  $M$ , we say that two integers  $a$  and  $b$  are equivalent modulo  $M$ , if there 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.

$$\begin{aligned} 365 &= 3 \times 100 + 6 \times 10 + 5 \times 1 \\ &= 3 \times 10^2 + 6 \times 10^1 + 5 \times 10^0 \end{aligned}$$

- This is unique because each digit is in the range 0 to 9, written  $[0,10)$

# 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

$$\begin{array}{r} 3 \quad 6 \quad 5 \\ + \quad 6 \quad 7 \quad 3 \\ \hline \end{array}$$

# 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

$$\begin{array}{r} 3 \quad 6 \quad 5 \\ + \quad 6 \quad 7 \quad 3 \\ \hline 8 \end{array}$$

# 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

$$\begin{array}{r} 1 \\ 3 \quad 6 \quad 5 \\ + \quad 6 \quad 7 \quad 3 \\ \hline \quad \quad 3 \quad 8 \end{array}$$

# 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

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



# 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

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

# 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:

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

# 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:

$$\begin{array}{r} 3 \quad 6 \quad 5 \\ + \quad 6 \quad 7 \quad 3 \\ \hline 0 \quad 3 \quad 8 \end{array}$$

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

# Bases (binary)

- We can also write out numbers in base two

$$\begin{aligned}(s_3 s_2 s_1 s_0)_2 &= s_3 \times 8 + s_2 \times 4 + s_1 \times 2 + s_0 \times 1 \\ &= s_3 \times 2^3 + s_2 \times 2^2 + s_1 \times 2^1 + s_0 \times 2^0\end{aligned}$$

where  $s_0, s_1, s_2, s_3$  are either 0 or 1.

# Bases (binary)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum is larger than 1:

$$\begin{array}{r} \phantom{+} (1 \phantom{0} 1 \phantom{0})_2 \\ + (0 \phantom{0} 1 \phantom{0})_2 \\ \hline ( \phantom{0} \phantom{0} \phantom{0} )_2 \end{array}$$

# Bases (binary)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum is larger than 1:

$$\begin{array}{r} \phantom{+} (1 \ 1 \ 0)_2 \\ + (0 \ 1 \ 1)_2 \\ \hline ( \phantom{0} \phantom{1} \ 1)_2 \end{array}$$

# Bases (binary)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum is larger than 1:

$$\begin{array}{r} 1 \\ (1 \ 1 \ 0)_2 \\ + (0 \ 1 \ 1)_2 \\ \hline ( \quad 0 \ 1)_2 \end{array}$$

# Bases (binary)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum is larger than 1:

$$\begin{array}{r} \phantom{0}1 \phantom{0}1 \\ \phantom{0}(1 \phantom{0}1 \phantom{0}0)_2 \\ + \phantom{0}(0 \phantom{0}1 \phantom{0}1)_2 \\ \hline ( \phantom{0}0 \phantom{0}0 \phantom{0}1)_2 \end{array}$$



# Bases (binary)

- As before, we add two numbers by adding the digits from smallest to largest, carrying if the sum is larger than 1:

$$\begin{array}{r} 1 \quad 1 \\ (1 \quad 1 \quad 0)_2 \\ + (0 \quad 1 \quad 1)_2 \\ \hline (1 \quad 0 \quad 0 \quad 1)_2 \end{array}$$

# Bases (binary)

Q: Using three digits in base two, how many numbers can we represent?

A: 8 (including zero)

Note:

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

$$\begin{array}{r} \phantom{+} \phantom{(} \phantom{0} \phantom{1} \phantom{)}_2 \\ \phantom{+} (1 \phantom{0} \phantom{1} \phantom{)}_2 \\ + (0 \phantom{0} \phantom{1} \phantom{)}_2 \\ \hline (1 \phantom{0} \phantom{0} \phantom{0} \phantom{)}_2 \end{array}$$

# Bases (general)

Q: Using three digits in base two, how many numbers can we represent?

A: 8 (including zero)

Note:

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

$$\begin{array}{r} (1 \ 1 \ 0)_2 \\ + (0 \ 1 \ 1)_2 \\ \hline (0 \ 0 \ 1)_2 \end{array}$$

- If we only use three digits, we lose the leading digit to overflow
- This is the same as the number mod 8.

# 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 (two)

- Similarly, given a number in base two:

$$(s_3s_2s_1s_0)_2 = s_3 \times 8 + s_2 \times 4 + s_1 \times 2 + s_0 \times 1$$

we can get an expression of the number in base 4 by grouping digits:

$$(s_3s_2s_1s_0)_2 = (s_3 \times 2 + s_2) \times 4 + (s_1 \times 2 + s_0) \times 1$$

Similarly, we can get an expression of the number in base 8, or base 16, or ...

# Bases (examples)

What is the value in base 10?

- $(1101)_2 =$

# Bases (examples)

What is the value in base 10?

- $(1101)_2 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$   
     $= 8 + 4 + 1$   
     $= 13$

# Bases (examples)

What is the value in base 2?

- $27 =$



# Bases (examples)

What is the value in base 2?

- $27 = 16 + 8 + 4 + 2 + 1$   
 $= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$   
 $= (11011)_2$

# Bases (examples)

What is the value in base 2?

- $27 = 16 + 8 + 4 + 2 + 1$   
 $= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$   
 $= (11011)_2$

What is the value in base 4?

- $27 =$

# Bases (examples)

What is the value in base 2?

- $$\begin{aligned} 27 &= 16 + 8 + 4 + 2 + 1 \\ &= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 \\ &= (11011)_2 \end{aligned}$$

What is the value in base 4?

- $$\begin{aligned} 27 &= 16 + 8 + 3 \\ &= 1 \times 16 + 2 \times 4 + 3 \times 1 \\ &= (123)_4 \end{aligned}$$

# 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?  $(115)_{16}$

- $(115)_{16} = 1 \times 16^2 + 1 \times 16^1 + 5 \times 16^0$
- $(115)_{16} = \quad \quad \quad 1 \times 16^1 + 15 \times 16^0$
- $(115)_{16} = \quad \quad \quad 11 \times 16^1 + 5 \times 16^0$

# 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] int`s 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

On the machine, **a** is assigned the value:

$$a \leftarrow (00000000\ 00000000\ 00000000\ 00011110)_2$$
$$a \leftarrow (00\ 00\ 00\ 1e)_{16}$$

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

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

# 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;
}
```

On the machine, **a** is assigned the value:

$$a \leftarrow ( 00000000\ 00000000\ 00000000\ 00011110 )_2$$
$$a \leftarrow ( 00\ 00\ 00\ 1e )_{16}$$

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

- 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

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



# 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

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

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

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

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, **floats** 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\*

$$\begin{aligned} & b_1 \times 2^{e_1} + b_2 \times 2^{e_2} \\ & \quad \Downarrow \\ & (b_1 \times 2^{e_1-e_2}) \times 2^{e_2} + b_2 \times 2^{e_2} \\ & \quad \Downarrow \\ & (b_1/2^{e_2-e_1} + b_2) \times 2^{e_2} \end{aligned}$$

⇒ 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:

$$(a + b) + c \neq a + (b + c)$$

\*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;
}
```

⇒ If  $b_1 \times$

⇒ Addition of floating point

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

zero

$+ c)$

\*Assume  $e_2 > e_1$



# Floating point value representation

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

- On most machines, **doubles** 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

# Bit-wise operations: Integer types only

## Recall:

- 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

## Recall:

- 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

## Recall:

- We can determine if a bit is on or off using  $\ll$  and  $\&$
- Or we can use  $\gg$  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

## Recall:

- 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

## Recall:

- We can
- Or we can
- We can

### Note:

We set the mask to  $1 \ll 7$  because `chars` are 8 bits long.  
For `ints` we would use a mask of  $1 \ll 31$ .  
Etc.

(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
>>
```

# Outline

- Exercise 14
- Numerical representation
- Casting
- Review questions

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

```
<type-1> lhs;
```

```
<type-2> rhs;
```

```
lhs = rhs;
```



# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are integers and `sizeof( LHS ) >= sizeof( RHS )`  
⇒ the conversion happens without loss of information

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

```
>> ./a.out
97 -> 97
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are integers and `sizeof( LHS ) < sizeof( RHS )`  
⇒ an implicit “modulo” operation is performed  
(modulo  $2^b$  where  $b$  is the number of bits in the LHS)

```
#include <stdio.h>
int main( void )
{
    int i = 511;
    char c = i;
    printf( "%d -> %d\n" , i , c );
    return 0;
}
```

```
>> ./a.out
511 -> -1
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are floats and `sizeof( LHS ) >= sizeof( RHS )`  
⇒ the conversion happens without loss of information

```
#include <stdio.h>
int main( void )
{
    float f = 1.5;
    double d = f;
    printf( "%.8f -> %.8f\n" , f , d );
    return 0;
}
```

```
>> ./a.out
1.50000000 -> 1.50000000
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If both are floats and `sizeof( LHS ) < sizeof( RHS )`  
⇒ rounding is performed

```
#include <stdio.h>
int main( void )
{
    double d = 1.7;
    float f = d;
    printf( "%.8f -> %.8f\n" , d , f );
    return 0;
}
```

```
>> ./a.out
1.70000000 -> 1.70000005
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If the LHS is an integer and the RHS is a floating point value  
⇒ the fractional part is discarded

```
#include <stdio.h>
int main( void )
{
    double d = -3.6;
    int i = d;
    printf( "%.8f -> %d\n" , d , i );
    return 0;
}
```

```
>> ./a.out
-3.60000000 -> -3
>>
```

Note that this is not the same thing as rounding down to the nearest integer

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

- If the LHS is a floating point value and the RHS is an integer  
⇒ the closest floating point representation is used

```
#include <stdio.h>
int main( void )
{
    int i = 123456789;
    float f = i;
    printf( "%d -> %.0f\n" , i , f );
    return 0;
}
```

```
>> ./a.out
123456789 -> 123456792
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

- double → unsigned char:  
511.5 → 511 → 255

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```



# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

- `double` → `unsigned char`:  
511.5 → 511 → 255
- `unsigned char` → `char`:  
255 → -1

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

# Casting between types (numbers)

When you assign a value to a variable, the right-hand-side (RHS) is implicitly converted (a.k.a. cast) to the type of the left-hand-side (LHS)

The same rules apply when passing values to/from a function

- double → unsigned char:  
511.5 → 511 → 255
- unsigned char → char:  
255 → -1
- char → float:  
-1 → -1.f

```
#include <stdio.h>
char foo( unsigned char c ){ return c; }
int main( void )
{
    double d = 511.5;
    float f = foo( d );
    printf( "%g -> %g\n" , d , f );
    return 0;
}
```

```
>> ./a.out
511.5 -> -1
>>
```

# Casting between types (numbers)

When casting, the types are ranked:

- Larger size integers/floats are “higher rank”  
char < int < long  
unsigned char < unsigned int < unsigned long  
float < double
- Unsigned integers are “higher rank” than signed integers\*  
char < unsigned char < int < unsigned int < long < unsigned long  
float < double
- Floating point values are “higher rank” than integers  
char < unsigned char < int < unsigned int < long < unsigned long < float < double

\*when they have the same size.

# Casting between types (numbers)

When casting, the types are ranked:

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

When we cast from lower rank to higher rank, we are **promoting**.

When we cast from higher rank to lower rank, we are **narrowing**.

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

char < unsigned char < int < unsigned int < long < unsigned long < float < double



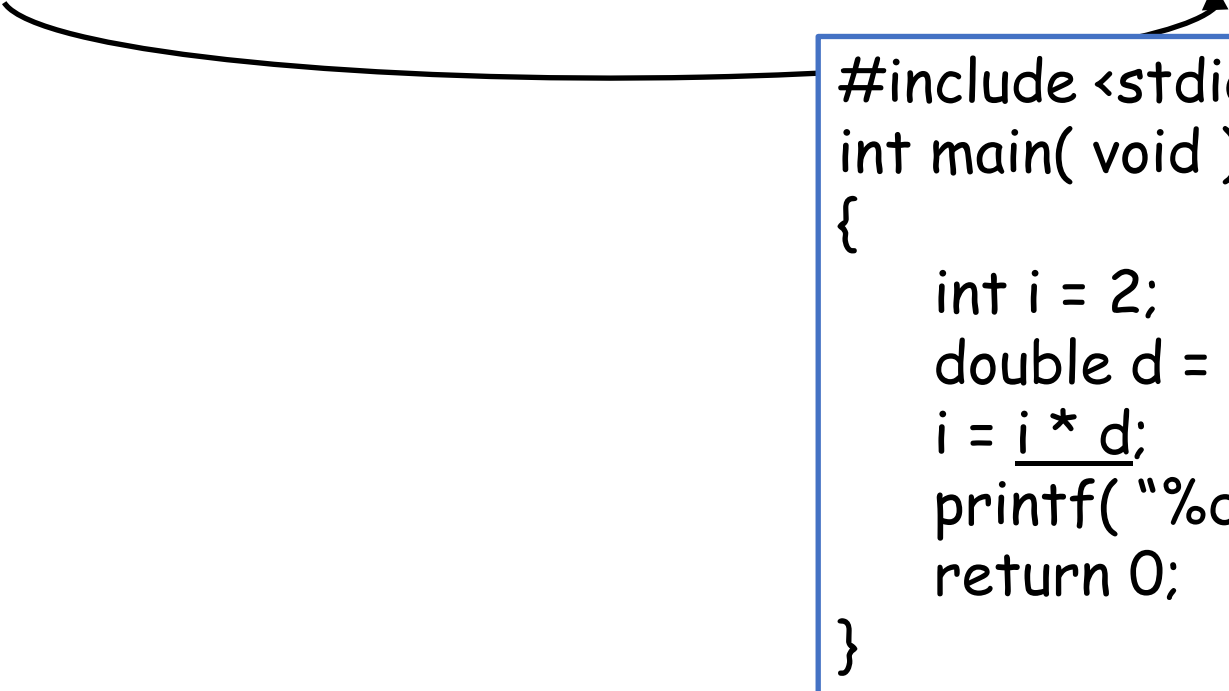
```
#include <stdio.h>
int main( void )
{
    int i = -1;
    unsigned int ui = 1;
    printf( "d\n" , i<ui );
    return 0;
}
```

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

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

char < unsigned char < int < unsigned int < long < unsigned long < float < double




```
#include <stdio.h>
int main( void )
{
    int i = 2;
    double d = 2.5;
    i = i * d;
    printf( "%d\n" , i );
    return 0;
}
```

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

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

char < unsigned char < int < unsigned int < long < unsigned long < float < double



```
#include <stdio.h>
int main( void )
{
    int i = 2;
    double d = 2.5;
    i *= d;
    printf( "%d\n" , i );
    return 0;
}
```

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

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

```
#include <stdio.h>
int main( void )
{
    int one = 1;
    int four = 4;
    int i = one / four * four;
    printf( "%d\n" , i );
    return 0;
}
```

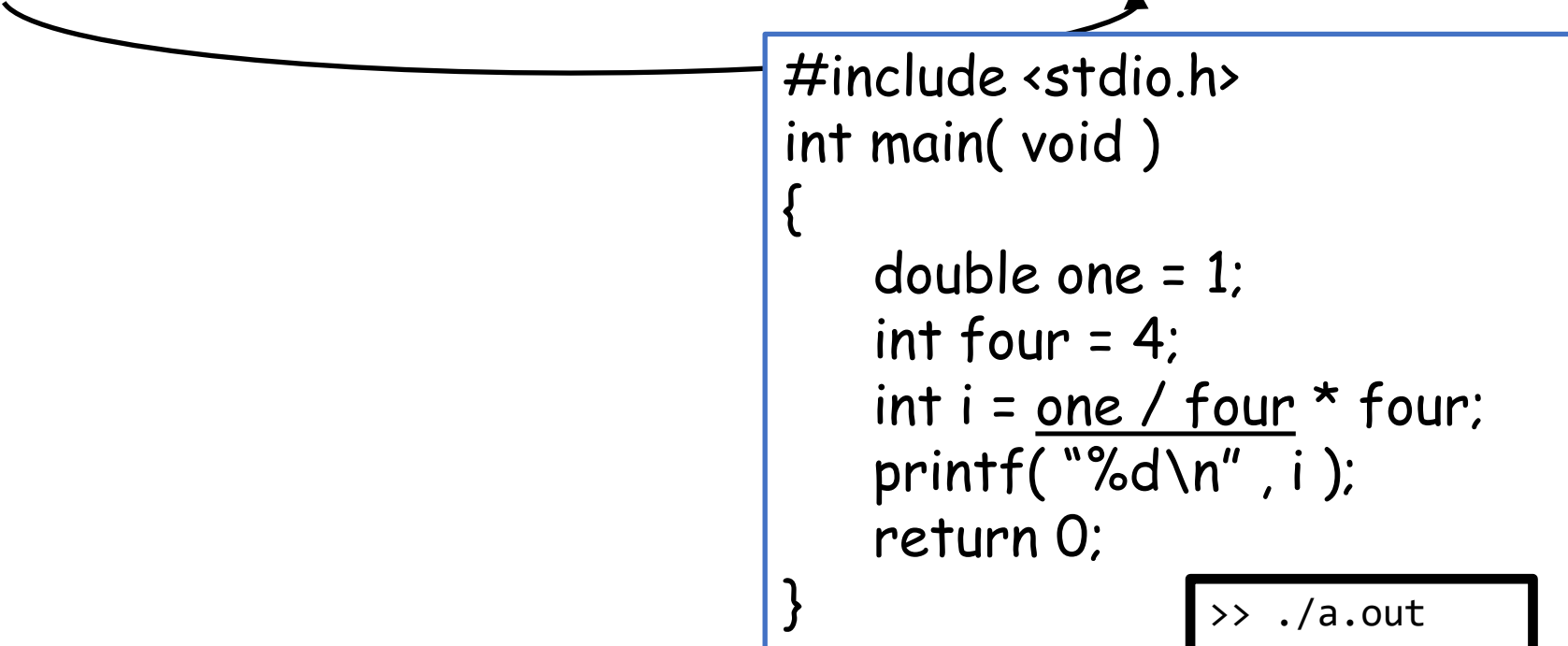
```
>> ./a.out
0
>>
```



# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

char < unsigned char < int < unsigned int < long < unsigned long < float < double



```
#include <stdio.h>
int main( void )
{
    double one = 1;
    int four = 4;
    int i = one / four * four;
    printf( "%d\n" , i );
    return 0;
}
```

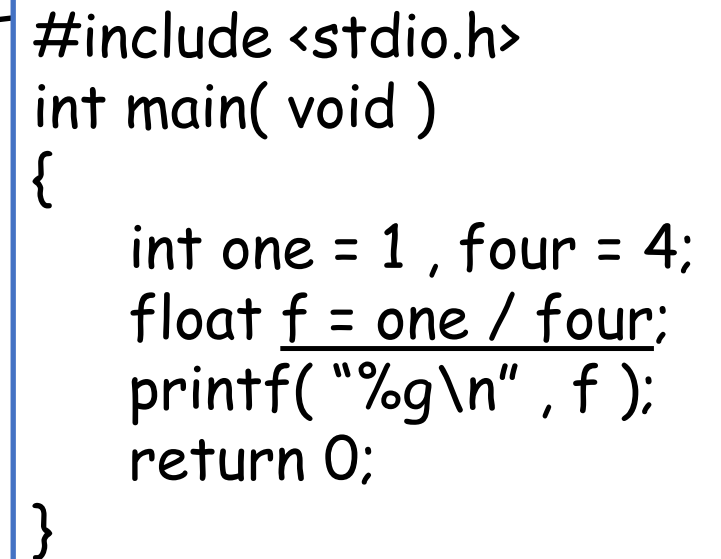
```
>> ./a.out
1
>>
```

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

Since evaluation precedes assignment, we get truncated results even though the LHS doesn't require it.



```
#include <stdio.h>
int main( void )
{
    int one = 1 , four = 4;
    float f = one / four;
    printf( "%g\n" , f );
    return 0;
}
```

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

# Casting between types (numbers)

When performing a binary operation (arithmetic or comparison) with different types the “lower rank” operand is implicitly promoted:

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

The desired behavior can be forced with casting:

- Preceding the variable name with `<type-name>` converts the variable to type `<type-name>`
- Since casting takes precedence over arithmetic operations:
  1. We convert `one` to a `float`
  2. And then divide a `float` by an `int`
    - a. This implicitly promotes `four` to a `float`
    - b. And then performs `float` by `float` division

```
#include <stdio.h>
int main( void )
{
    int one = 1 , four = 4;
    float f = (float)one / four;
    printf( "%g\n" , f );
    return 0;
}
```

```
>> ./a.out
0.25
>>
```

# Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly

```
#include <stdio.h>
int main( void )
{
    ...
    int i = 1;
    int* ip = &i;
    float *fp= (float*)ip;
    ...
}
```

# Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly
    - Unless one of them has type `void*`

```
...  
void * malloc( size_t );  
...
```

```
#include <stdio.h>  
int main( void )  
{  
    ...  
    float* a = malloc( 10 * sizeof( float ) );  
    ...  
}
```

# Casting between types (pointers)

- Since pointers represent locations in memory (independent of type)
  - We can cast between pointer types
    - This needs to be done explicitly
    - Unless one of them has type `void*`
  - We can also explicitly cast between pointers and integers
    - This needs to be done with care since a pointer can have different sizes on different machines:
      - 4 bytes on a 32-bit machine
      - 8 bytes on a 64-bit machine
    - The `size_t` type is guaranteed to always have the size of a pointer

```
#include <stdio.h>
int main( void )
{
    int i = 100;
    int* ip = &i;
    size_t addr = (size_t)ip;
    printf( "Address is: %zu\n" , addr );
    return 0;
}
```

# Casting between types

Three types of casting:

## 1. Nothing changes in the binary representation

- pointers  $\leftrightarrow$  pointers
  - A memory address is a memory address
  - The compiler needs to know the type to transform element offsets into byte offsets
- unsigned integers  $\leftrightarrow$  signed integers
  - Different representations of numbers modulo  $M$  still represent the same number
  - The compiler needs to know the type for comparisons

# Casting between types

Three types of casting:

1. Nothing changes in the b

- pointers ↔ pointers
  - A memory address is a mem
  - The compiler needs to know
- unsigned integers ↔ signed
  - Different representations of r
  - The compiler needs to know

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

```
void PrintBinary( const void* mem , size_t sz ){ ... }  
int main( void )  
{
```

```
    int iArray[] = { 1 , 2 , 3 , 4 };
```

```
    int* iPtr = iArray;
```

```
    char* cPtr = (char*)iPtr;
```

```
    PrintBinary( iPtr , sizeof(iPtr) );
```

```
    PrintBinary( cPtr , sizeof(cPtr) );
```

```
    return 0;
```

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

```
00100000 01001001 11010100 11001000 11111100 01111111 00000000 00000000
```

```
00100000 01001001 11010100 11001000 11111100 01111111 00000000 00000000
```

```
>>
```



# Casting between types

## Three types of casting:

### 1. Nothing changes in the b

- pointers ↔ pointers
  - A memory address is a memory address
  - The compiler needs to know
- unsigned integers ↔ signed integers
  - Different representations of memory
  - The compiler needs to know

```
#include <stdio.h>

void PrintBinary( const void* mem , size_t sz ){ ... }

int main( void )
{
    unsigned int ui =(1<<31)|1;
    int i = ui;
    printf( " %u = " , ui ); PrintBinary( &ui , sizeof(ui) );
    printf( "%d = " , i ); PrintBinary( &i , sizeof(i) );
    return 0;
}
```

```
>> ./a.out
2147483649 = 10000000 00000000 00000000 00000001
-2147483647 = 10000000 00000000 00000000 00000001
>>
```

# Casting between types

Three types of casting:

1. Nothing changes in the binary representation
2. Binary representations are truncated/expanded
  - integers  $\leftrightarrow$  integers (of different sizes)

# Casting between types

Three types of casting:

1. Nothing changes in the binary representation

2. Binary representations are preserved

- integers  $\leftrightarrow$  integers (of different sizes)

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

```
void PrintBinary( const void* mem , size_t sz ){ ... }
```

```
int main( void )
```

```
{
```

```
    int i = 254;
```

```
    unsigned char c = i;
```

```
    printf( "%d = " , i ); PrintBinary( &i , sizeof(i) );
```

```
    printf( "%d = " , c ); PrintBinary( &c , sizeof(c) );
```

```
    return 0;
```

```
}
```

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

```
254 = 00000000 00000000 00000000 11111110
```

```
254 = 11111110
```

```
>>
```

# Casting between types

Three types of casting:

1. Nothing changes in the binary representation
2. Binary representations are truncated/expanded
3. Binary representations are completely different
  - integers  $\leftrightarrow$  floating point values
  - floating point values  $\leftrightarrow$  floating point values (of different sizes)

# Casting between types

Three types of casting:

1. Nothing changes in the binary representation
2. Binary representations are preserved
3. Binary representations are converted
  - integers  $\leftrightarrow$  floating point values
  - floating point values  $\leftrightarrow$  integers

```
#include <stdio.h>

void PrintBinary( const void* mem , size_t sz ){ ... }

int main( void )
{
    int i = 1;
    float f = i;
    printf( "%d = " , i ); PrintBinary( &i , sizeof(i) );
    printf( "%.1f = " , f ); PrintBinary( &f , sizeof(f) );
    return 0;
}
```

```
>> ./a.out
1    = 00000000 00000000 00000000 00000001
1.0  = 00111111 10000000 00000000 00000000
>>
```

# Outline

- Exercise 14
- Numerical representation
- Casting
- Review questions

# Review questions

1. What is *two's complement* representation?

It is a signed integer representation. The negative of a number is obtained by flipping the bits and adding one.

# Review questions

2. How does representation of integers and floating-point values differ in C?

The bits of an integer correspond to its representation in base two.

The bits of a floating-point value are split in two parts – the mantissa and the exponent.



# Review questions

3. What is *type narrowing*?

Converting a “higher rank” data type into a “lower rank” one

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

# Review questions

4. What is *type promotion*?

Converting a “lower rank” data type into a “higher rank” one

`char < unsigned char < int < unsigned int < long < unsigned long < float < double`

# Review questions

5. What is *type casting*?

Explicitly or implicitly converting a value from one type to another

# Review questions

6. What is the output of:

```
int n = 32065;  
float x = 24.79;  
printf( "int n = %d but (char)n = %c\n" , n , (char)n );  
printf( "float x = %f but (long)x = %ld\n" , x , (long)x );
```

In binary, we have:

32065 = (00000000 00000000 01111101 01000001)\_2

Casting to a char we get:

(01000001)\_2 = 65 -> 'A'

```
int n = 32065 but (char)n = A  
float x = 24.790001 but (long)x = 24
```

# Exercise 14

- Website -> Course Materials -> Exercise 14