Primitive Data Types

“Elementary Elements”

Prerequisite: None

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

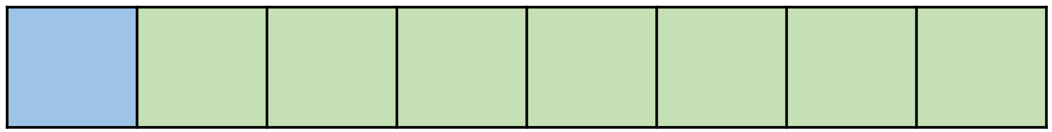
Primitive Data Types

- Char
- Integer
- Float
- Double

- Array
- Multidimensional Array

char Data Type

8 bit



1 sign bit 7 bit data -128 to 127

Code	Char	Code	Char	Code	Char	Code	Char
0128	€	0160		0192	À	0224	à
0129	_	0161	¡	0193	Á	0225	á
0130	‚	0162	¢	0194	Â	0226	â
0131	ƒ	0163	£	0195	Ã	0227	ã
0132	„	0164	¤	0196	Ä	0228	ä
0133	…	0165	¥	0197	Å	0229	å
0134	†	0166	¦	0198	Æ	0230	æ
0135	‡	0167	§	0199	Ç	0231	ç
0136	ˆ	0168	¨	0200	È	0232	è
0137	‰	0169	©	0201	É	0233	é
0138	_	0170	ª	0202	Ê	0234	ê
0139	‹	0171	«	0203	Ë	0235	ë
0140	Œ	0172	¬	0204	Ì	0236	ì
0141	_	0173	-	0205	Í	0237	í
0142	_	0174	®	0206	Î	0238	î
0143	_	0175	¯	0207	Ï	0239	ï
0144	_	0176	°	0208	Ð	0240	ð

`char` Range

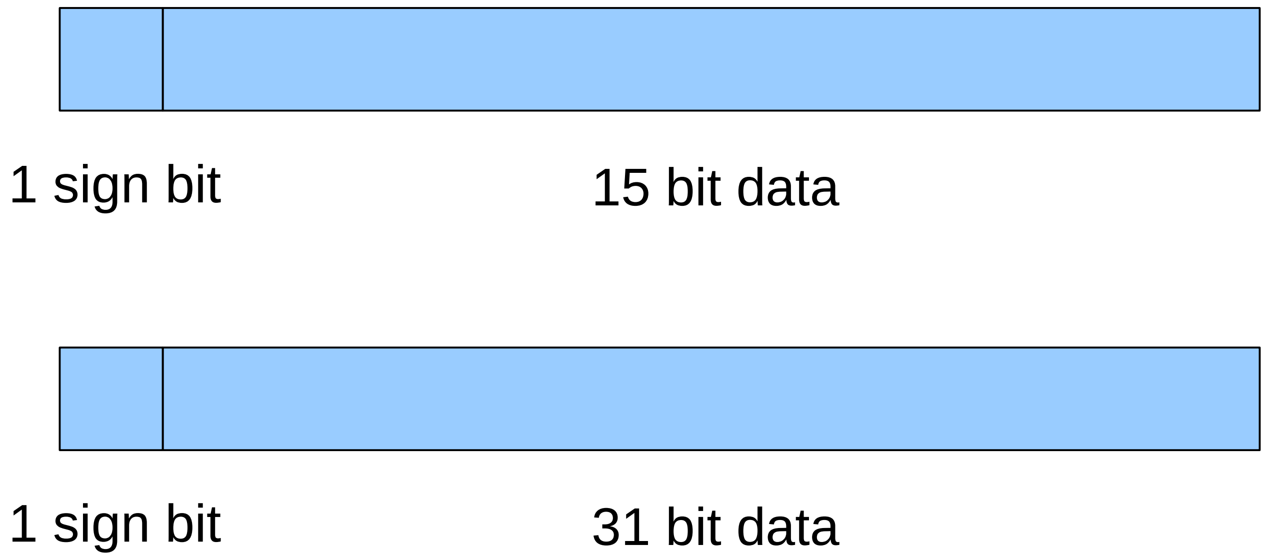
8 bit

8 bit data = 2^8 = 256 characters

Code	Char	Code	Char	Code	Char	Code	Char
0128	€	0160		0192	À	0224	à
0129	_	0161	¡	0193	Á	0225	á
0130	‚	0162	¢	0194	Â	0226	â
0131	ƒ	0163	£	0195	Ã	0227	ã
0132	„	0164	¤	0196	Ä	0228	ä
0133	…	0165	¥	0197	Å	0229	å
0134	†	0166	¦	0198	Æ	0230	æ
0135	‡	0167	§	0199	Ç	0231	ç
0136	ˆ	0168	¨	0200	È	0232	è
0137	‰	0169	©	0201	É	0233	é
0138	_	0170	ª	0202	Ê	0234	ê
0139	‹	0171	«	0203	Ë	0235	ë
0140	Œ	0172	¬	0204	Ì	0236	ì
0141	_	0173	-	0205	Í	0237	í
0142	_	0174	®	0206	Î	0238	î
0143	_	0175	¯	0207	Ï	0239	ï
0144	_	0176	°	0208	Ð	0240	ð

Memory Allocation for int

16 or 32 bit



Memory Allocation for int

16 bit or 32 bit

16 bit range = -32,767 to 32,767

15 14… …0

1 sign bit 15 bit data

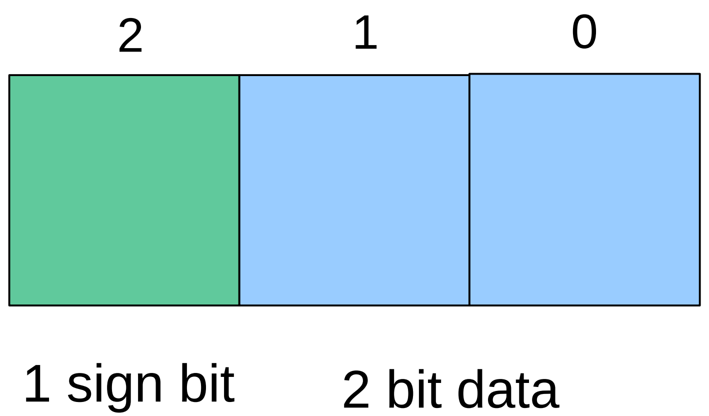
32 bit range = -2,14,74,83,647 to 2,14,74,83,647

31 30… …0

1 sign bit 31 bit data

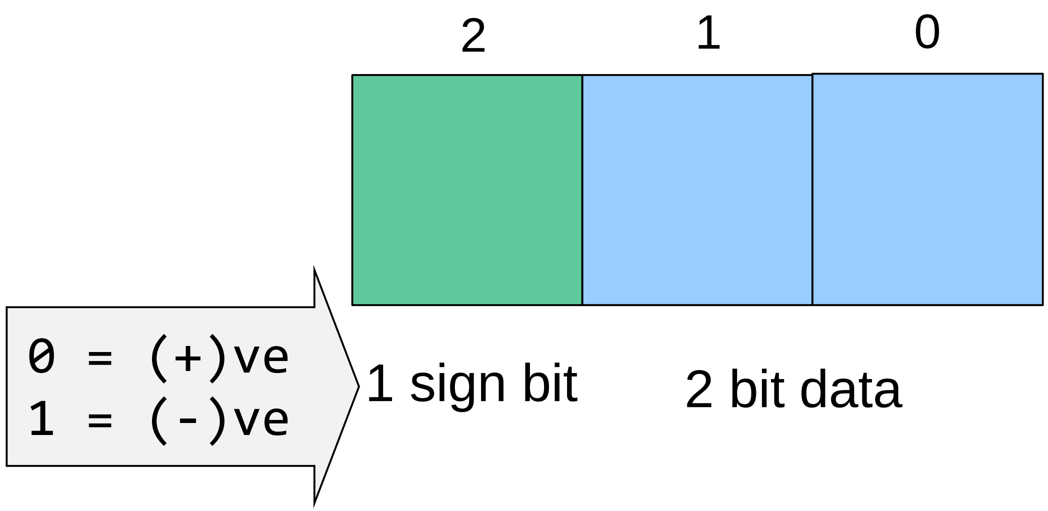
Where does this range come from?

Suppose `int` occupies only 3 bits



Where does this range come from?

Suppose `int` occupies only 3 bits



Understanding 2s complement

How to perform 2s complement

1. Take binary representation 101
2. Take 1s complement (Invert all bits) 010
3. Add 1 011

Representing signed numbers

Suppose `int` occupies only 3 bits

`0 00`	`+0`
`0 01`	`+1`
`0 10`	`+2`
`0 11`	`+3`

Representing signed numbers

Suppose `int` occupies only 3 bits

```
1 00

1 01

1 10

1 11
```

Representing signed numbers

Suppose `int` occupies only 3 bits

`1 00`	-4
`1 01`	-3
`1 10`	-2
`1 11`	-1

Representing signed numbers

Suppose `int` occupies only 3 bits

Binary	Decimal
0 00	+0
0 01	+1
0 10	+2
0 11	+3
1 00	-4
1 01	-3
1 10	-2
1 11	-1

3 bit = 2^3 = 8 numbers

When 2s complement is used:

4 positive : 0, 1, 2, 3

4 negative: -1, -2, -3, -4

Reviewing the ranges

Data-type	Size in bits	Range
int	16 or 32	-32,767 to 32,767
char	8	-127 to 127

Source: C++: The Complete Reference, 4th Edition (Herb Schldt, page 15)

Recap

Why do we use 2s complement instead of just flipping the sign bit?

Recap

Why do we use 2s complement instead of just flipping the sign bit?

Because otherwise it would produce minus zero (-0)

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

16 bit can contain = 2^{16} = 65536 numbers

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

16 bit can contain = 2^{16} = 65536 numbers

Half contains negative number,
The rest of the half contains positive numbers (incl 0)

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

16 bit can contain = 2^{16} = 65536 numbers

Half contains negative number,
The rest of the half contains positive numbers (incl 0)

$$\frac{65536}{2} = 32768$$

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

16 bit can contain = 2^{16} = 65536 numbers

Half contains negative number,
The rest of the half contains positive numbers (incl 0)

$$\frac{65536}{2} = 32768$$

So largest negative number = -32768
Largest positive number = 32768 – 1= 32767

Recap

How do we calculate the range of `int`? Suppose the size is 16 bit

16 bit can contain = 2^{16} = 65536 numbers

Half contains negative number,
The rest of the half contains positive numbers (incl 0)

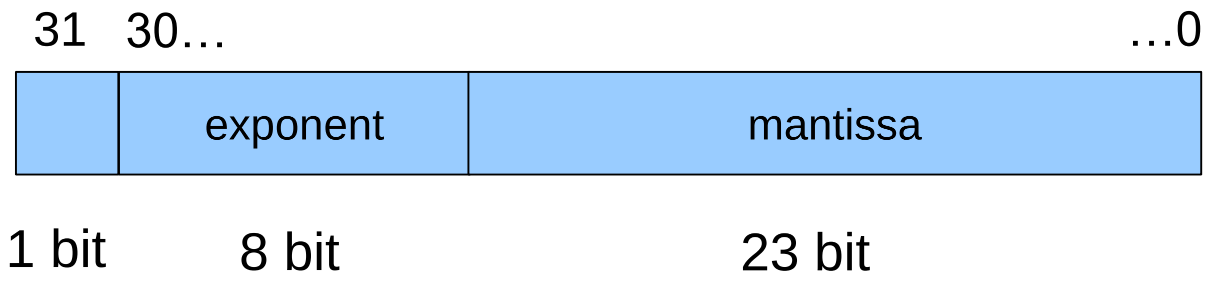
$$\frac{65536}{2} = 32768$$

So largest negative number = -32768
Largest positive number = 32768 – 1= 32767

So the range of 16 bit int is from -32768 to 32767

Memory Allocation of float

32 bit memory



Memory Allocation of float

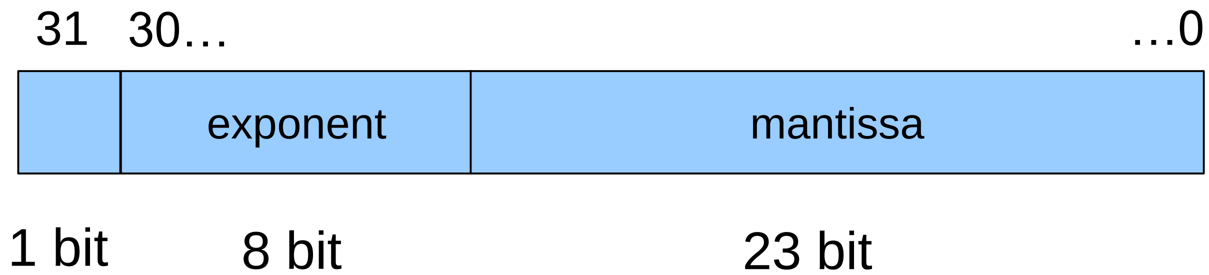
IEEE-754 floating-point standard

31	30…	…0
	exponent	mantissa
1 bit	8 bit	23 bit

Suppose we have to store 10.375

Memory Allocation of float

IEEE-754 floating-point standard

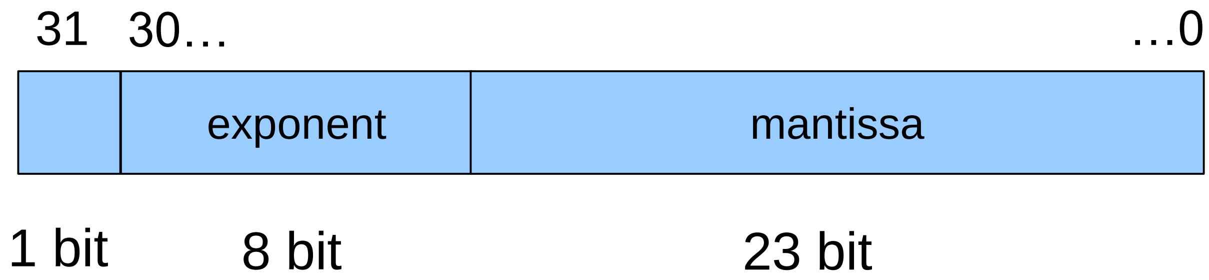


Suppose we have to store 10.375

1. Convert to Binary (=1010.011)

Memory Allocation of float

IEEE-754 floating-point standard



Suppose we have to store 10.375

1. Convert to Binary (=1010.011)
2. Move the floating point in front of the first 1, ie.

1.010011e+3

Memory Allocation of float

IEEE-754 floating-point standard

31	30…	…0
	exponent	mantissa
1 bit	8 bit	23 bit

Suppose we have to store 10.375

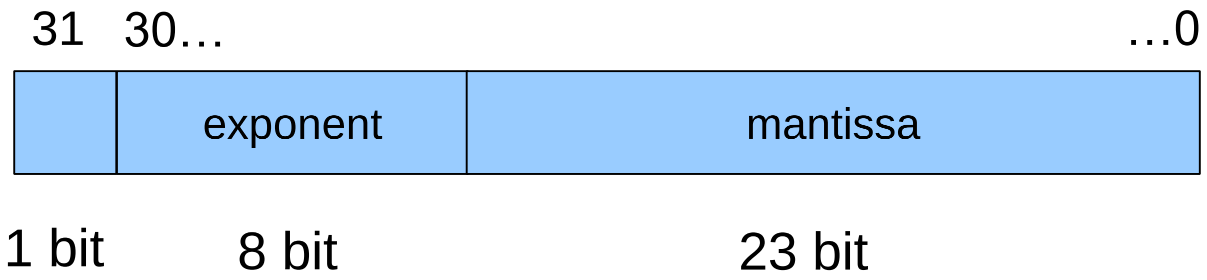
1. Convert to Binary (=1010.011)
2. Move the floating point in front of the first 1, ie.

1.010011e+3

[we had written e+3 because we have moved the point 3 times on the right]

Memory Allocation of float

IEEE-754 floating-point standard

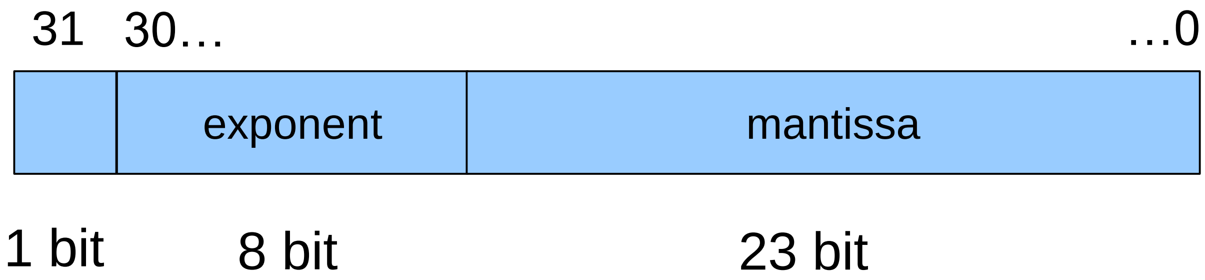


Suppose we have to store 10.375

1. Convert to Binary (=1010.011)
2. Move the floating point in front of the first 1, ie. 1.010011e+3
3. Now the value after the floating point is the mantissa (010011)

Memory Allocation of float

IEEE-754 floating-point standard

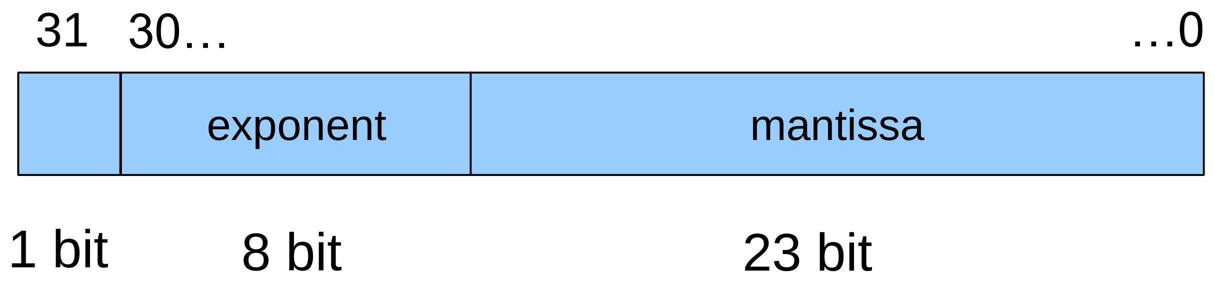


Suppose we have to store 10.375

1. Convert to Binary (=1010.011)
2. Move the floating point in front of the first 1, ie. 1.010011e+3
3. Now the value after the floating point is the mantissa (010011)
4. To find the exponent, add 127 with 3 (=130) and convert it to binary (=10000010)

Memory Allocation of float

IEEE-754 floating-point standard



Suppose we have to store 10.375

1. Convert to Binary (=1010.011)
2. Move the floating point in front of the first 1, ie. 1.010011e+3
3. Now the value after the floating point is the mantissa (010011)
4. To find the exponent, add 127 with 3 (=130) and convert it to binary (=10000010)
5. The sign bit is 0 for +

Memory Allocation of float

IEEE-754 floating-point standard

31	30…	…0
	exponent	mantissa
1 bit	8 bit	23 bit

Suppose we have to store 10.375

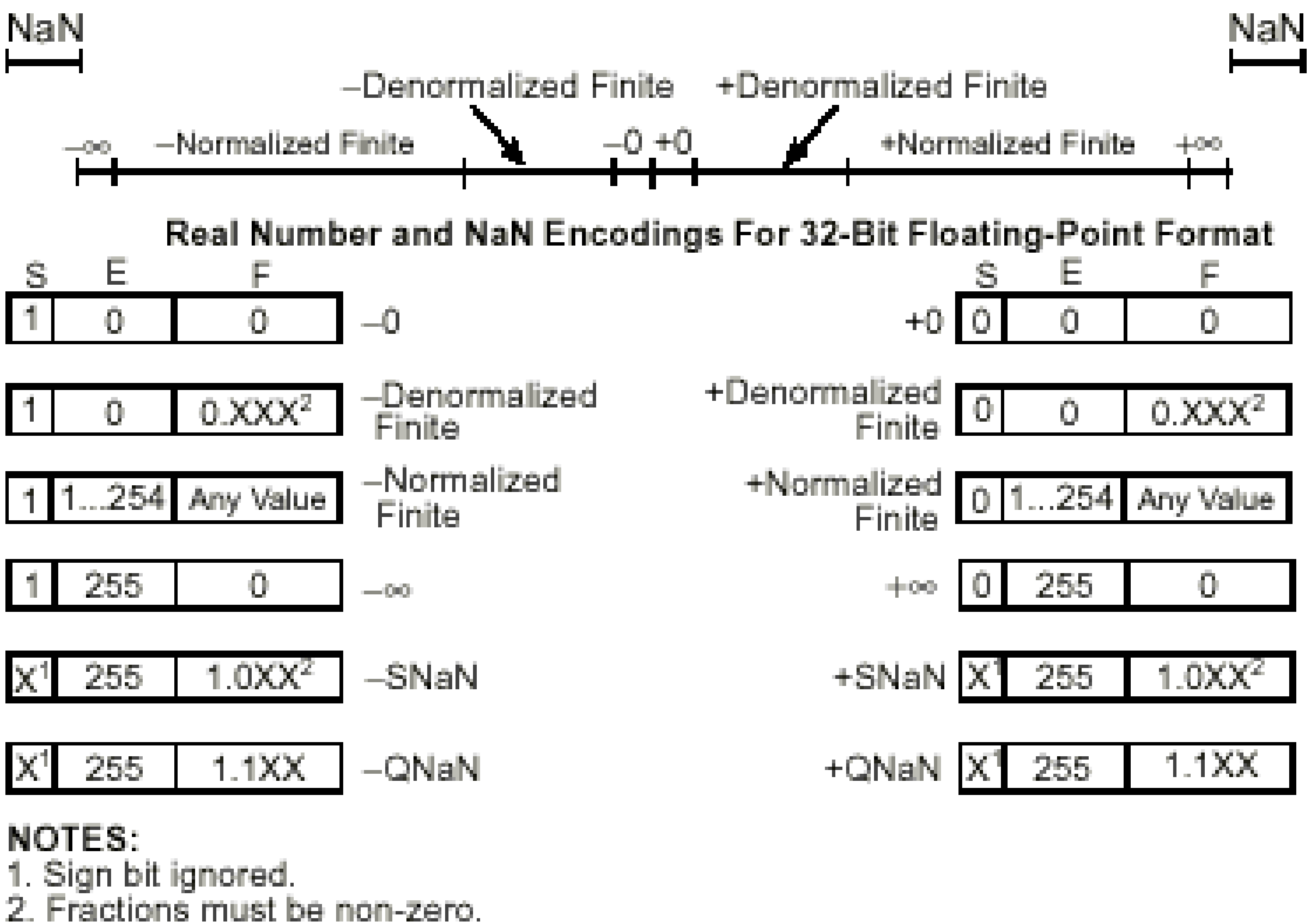
1. Convert to Binary (=1010.011)
2. Move the floating point after the first 1, ie. 1.010011e+3
3. Now the value after the floating point is the mantissa (010011)
4. To find the exponent, add 127 with 3 (=130) and convert it to binary (=10000010)
5. The sign bit is 0 for +

`0`	`10000010`	`010011` ...and rest 17 zero...
1 bit	8 bit	23 bit

Exercise

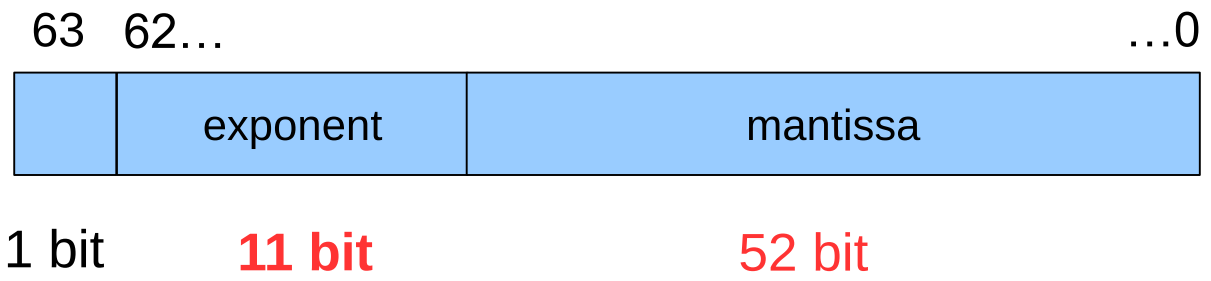
Convert 15.875 into IEEE-754 floating-point standard

Special Float Values



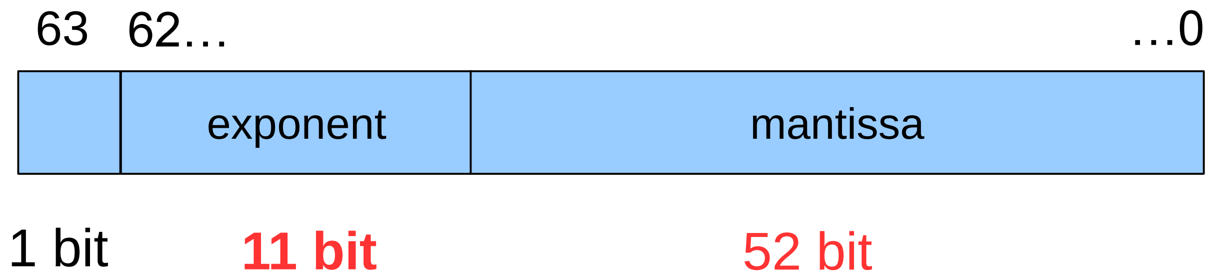
Memory Allocation of `double`

64 bit memory



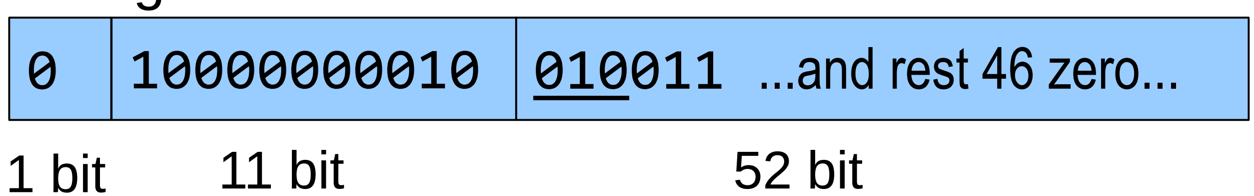
Memory Allocation of `double`

64 bit memory



Suppose we have to store 10.375

1. Convert to Binary (=1010.011)
2. Move the floating point after the first 1, ie. 1.010011e+3
3. Now the value after the floating point is the mantissa (010011)
4. To find the exponent, add 1023 with 3 (=1026) and convert it to binary (=10000000010)
5. The sign bit is 0 for +



Modifier

Four modifers.

```
1. signed
2. unsigned
3. long
4. short
```

Modifier

- For `char`

Modifier

For `char`

1. `signed char (7 bit)`
2. `unsigned char (8 bit)`

See Chapter 4.1 of Teach Yourself C, 3rd edition by Herb Schildt for size and range

Modifier

For `int`

1. `signed int`
2. `unsigned int`
3. `short int`
4. `long int`

5. `unsigned short int`
6. `unsigned long int`

See Chapter 4.1 of Teach Yourself C, 3rd edition by Herb Schildt for size and range

Modifier

For `float`

`No modifier`

`(Previously, long float was considered double. But it is obsolete in ANSI C)`

See Chapter 4.1 of Teach Yourself C, 3rd edition by Herb Schildt for size and range

Modifier

For `double`

```
long double (10 byte: 1_sign+15_exp+64_man)*
short double (Same as double)

*In this case, exponent bias is 16383
```

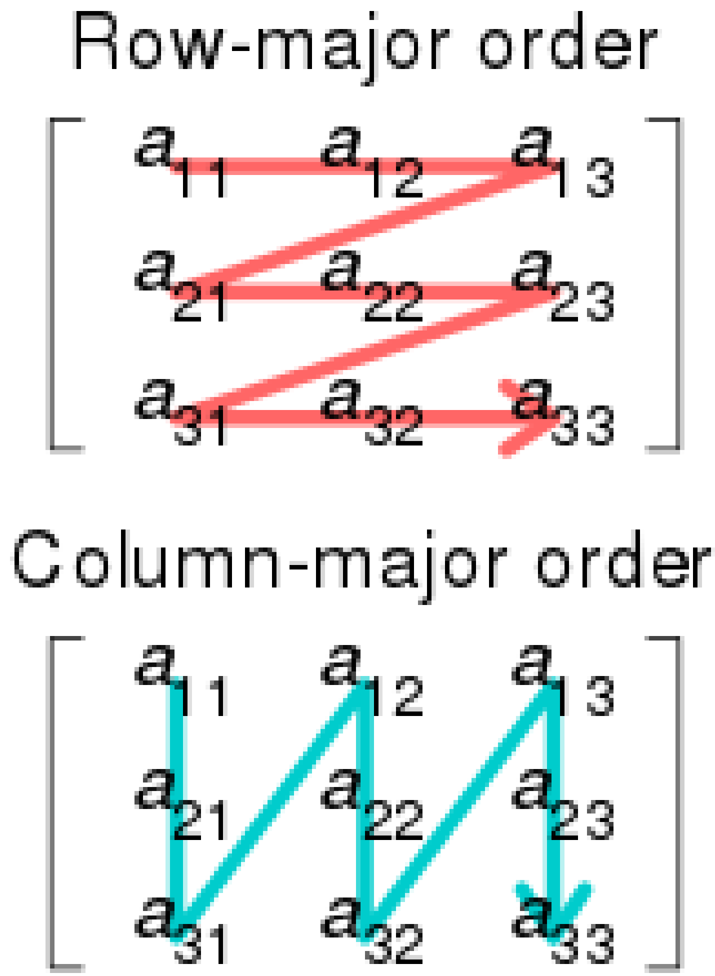
See Chapter 4.1 of Teach Yourself C, 3rd edition by Herb Schildt for size and range

Array

Contagious memory location

Base Address

2D Array



Source: http://cis.poly.edu/~mleung/CS1114/f04/ch10/MDmemory.htm

2D Array

1. A 2D array is stored in the computer's memory one row following another.

2. The address of the first byte of memory is considered as the memory location of the entire 2D array.

3. Knowing the address of the first byte of memory, the compiler can easily compute to find the memory location of any other elements in the 2D array provided the number of columns in the array is known.

   If each data value of the array requires B bytes of memory, and if the array has C columns, then the memory location of an element such as score[m][n] is (m*c+n)*B from the address of the first byte.

Source: http://cis.poly.edu/~mleung/CS1114/f04/ch10/MDmemory.htm

2D Array

4. Note that to find the memory location of any element, there is no need to know the total number of rows in the array, i.e. the size of the first dimension. Of course the size of the first dimension is needed to prevent reading or storing data that is out of bounds.

5. Again one should not think of a 2D array as just an array with two indexes. You should think of it as an array of arrays.

6. Higher dimensional arrays should be similarly interpreted. For example a 3D array should be thought of as an array of arrays of arrays. To find the memory location of any element in the array relative to the address of the first byte, the sizes of all dimensions other than the first must be known.

Source: http://cis.poly.edu/~mleung/CS1114/f04/ch10/MDmemory.htm

2D Array

int array1[3][2] = {{0, 1}, {2, 3}, {4, 5}};

In memory looks like this:

0 1 2 3 4 5

exactly the same as:

int array2[6] = { 0, 1, 2, 3, 4, 5 };

0	1
2	3
4	5

Source: http://cis.poly.edu/~mleung/CS1114/f04/ch10/MDmemory.htm

3D Array

array1[0][][] : {{0, 1}, {2, 3}, {4, 5}};

array1[1][][] : {{6, 7}, {8, 9}, {10, 11}};

0	1
2	3
4	5

6	7
8	9
10	11