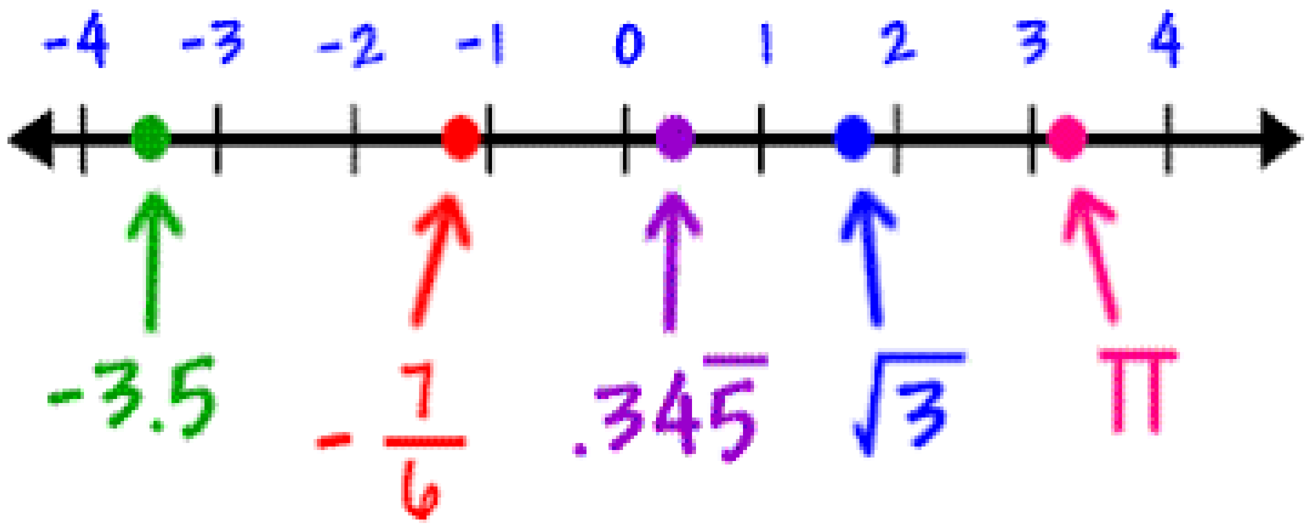
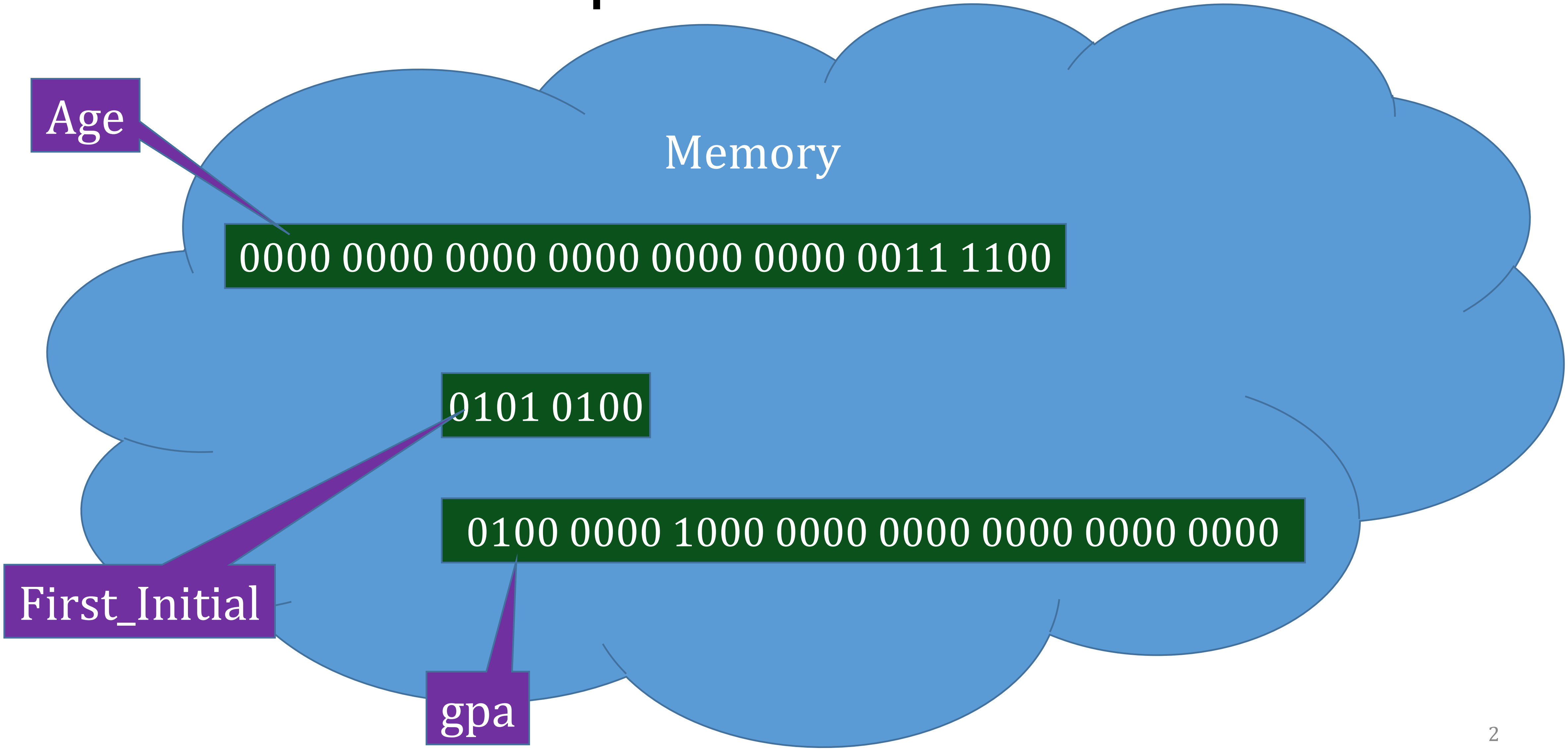
Data in C: Floating Point



Variable Concept



C Built-in Types

Numbers

Integer
Binary, 2's complement

Real
IEEE 754

char 8 bit	short 16 bit	int 32 bit	long 64 bit	float 32 bit	double 64 bit

Abstraction

Chap 2.4

Anything that is not an integer can be thought of as
<int>.<decimal>

e.g. 391.8125

Or can be thought of as

<int> + <numerator>/<denominator>

e.g.

391 + 8125/10000

or 391 13/16

Leak 1: **Floats are approximations!**

Numbers may not be exactly precise!

1/3 != 0.3333333333333333333333

$6.02214129 \times 10^{23}$ is not an exact Avogadro's constant

3.14159265358979323846264338327950288419716939937510
is not exactly π

Concrete: Floats are Binary

On computers, fractional numbers must be represented by bits

Implies base 2

Implies “binary point”

	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}	…
	4	2	1	.	1/2	1/4	1/8	1/16	…
…	1	0	1	.	1	0	0	1	…

Concrete: Floats are "Normalized"

Almost infinite number of ways to represent floating point numbers

- Implied binary point: 1101 1010 = 1101 10.10 = 54.5
- int numerator, int denominator: 0110 1101 / 0000 00010 = 109/2
- Scientific notation with int integer, int fraction, int exponent
  0000 0101 / 0111 0011 / 0000 0001
  $= (5+1/4 +1/8 +1/16 +1/128 + 1/256) \times 10^{1} =54.4921875$
- …

Scientific Notation (Base 10)

- Represent numbers as <Integer>.<Fraction> x $10^{\text{<exp>}}$
  - For instance: c = 2.99792458 x 10^{8} m/s; e=1.602 x 10^{-19} C
- Many different representations of the same number
  - c = 299,792,458 x 10^{0} m/s= 299.792458 x 10^{6} m/s
- "Normalization" : <1-9>.<Fraction> x $10^{\text{<exp>}}$
  - Special case for zero: 0.0 x 10^{0}

Scientific Notation Precision

- Numbers in scientific notation are often approximations
- Number of fractional digits indicates precision
  - $1/3 = 3.333333 \times 10^{-1}$ to within 7 digits (+/- 10^{-7})
  - $1000/3 = 3.333333 \times 10^{2}$ to within 7 digits (+/- 10^{-5})
- "Precision" is not the same as "tolerance"
  - Precision – relative accuracy - is independent of exponent
  - Tolerance - absolute accuracy – depends on exponent

Scientific Notation (Base Two)

- Start with a base two rational number… <0/1>*.<0/1>*

2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}
1	0	0	0	0	.	0	1	1

$= 16 + 1/4 + 1/8 = 16.375$

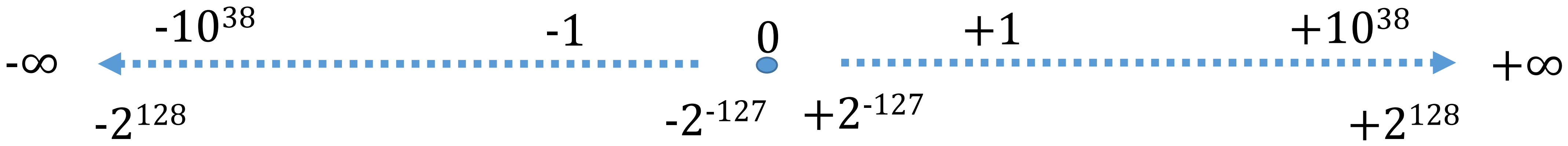
- Express normalized as 1.<FRAC> x $2^{\text{<exp>}}$
  - e.g. 1.0000011 x 2^4
  - Special case for zero: 0.0 x 2^0

IEEE Standard (32 bit float)

- First convert the number to the form:

$$value = -1^{S} \times FRAC \; \times \; 2^{exp}$$

- S = 0 (positive) or 1 (negative)
- $1 \leq FRAC < 2$ (except for special cases like 0 or +/- ∞)
- $-126 \leq \exp \leq 126$



Standard: IEEE 754

- Value Representation:
  - Decimal: [+/-]<digit>.<fraction> x $10^{<\text{exponent}>}$ e.g. 6.022 x 10^{23}
  - Binary: [+/-]1.<fraction> x $2^{<\text{exponent}>}$ e.g 1.11111110000101… x 2^{78}
  - Special case for 0, +/- ∞ (INFINITY), “Not a Number” (NAN)

- Bit Representation (32 bit float)

S	EXP								FRAC																						
b_{31}	b_{30}	b_{29}	b_{28}	b_{27}	b_{26}	b_{25}	b_{24}	b_{23}	b_{22}	b_{21}	b_{20}	b_{19}	b_{18}	b_{17}	b_{16}	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

8 bits

23 bits

IEEE 754 Special Cases

- +/-0

S	EXP								FRAC																						
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- +/-∞ (INFINITY)

S	EXP								FRAC																						
S	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Not a Number (NAN)

S	EXP								FRAC																						
S	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Note: math.h contains definitions of constants INFINITY and NAN, and also isnan(x) and isinf(x)

IEEE 754 Value vs. Bits

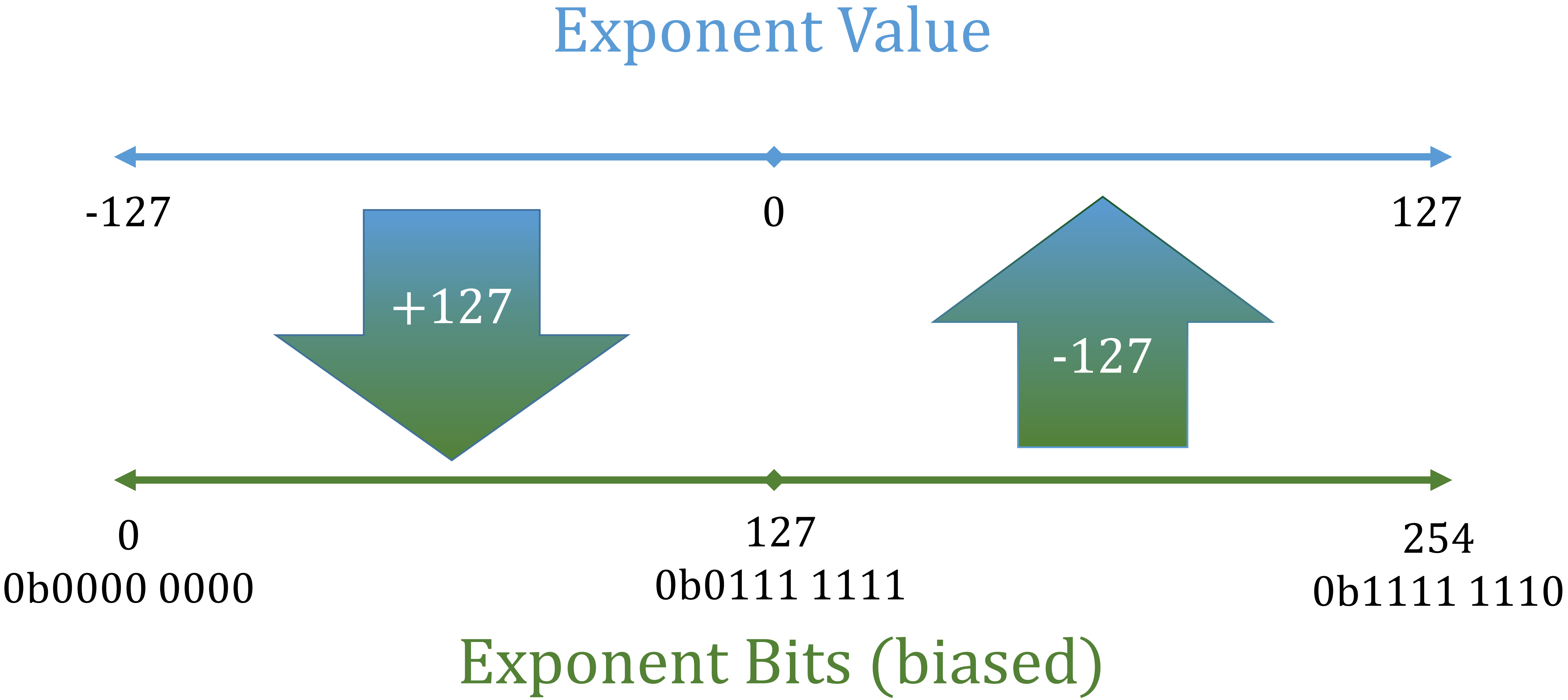
VALUE

- Decimal: -729.6
- Binary: 10 1101 1001.1001 1…
- B/Norm: 1.0110 1100 … x 2^{9}
- Exponent: 9

BITS

- Sign bit 0=+, 1=- : 1
- FRAC: 0110 1100 1100…
- Biased Exponent: 9+127=136 = 1000 1000

Biased Exponent



Bias Value

- Unsigned 8 bits holds values from 0 to 255 = 0b11111111 = 2^8-1
- 0b11111111 special case reserved for infinity and/or NAN
- Most useful bias is 127
  - enables both negative and positive exponents with the highest values
  - 127 = 0b01111111 = 2^7-1
- In general, if we have n bits for exponent, bias=$2^{(n-1)}$-1
  - For double, n=11, so bias=2^{10}-1 = 1023

IEEE 754 Value vs. Bits

value

- Sign bit is 0 for positive, 1 for negative… mathematically $(-1)^S$
- Except in special cases, $fraction = 1.FRAC$
bits
implicit
- Value must be **normalized** before it can be converted to bits
  - Normalization: moving binary point right of first 1 and adjusting exponent
  - E.g. 0b100110.1010 x 2^5 = 1.001101010 x 2^{10}
  - E.g. 0b0.0001010110 [x 2^0] = 1.010110 x 2^{-4}
- In bit form, exponent is >= 0

value
bits

  - Abstract exponent value is **biased** – add a constant
  - $EXP = exponent + 127 = exponent + 2^7 - 1 = \text{0x80} + exponent - 1$
  - $exponent = EXP - 127$ where EXP is 8 bit unsigned binary

value
bits

"Denormal" or "Subnormal" Numbers

- If EXP bits are zero,
  - Do NOT assume *fraction* starts with 1.<FRAC>
  - Assume it starts with 0.<FRAC> AND exponent is -126
- Allows numbers smaller than 2^{-126} to be represented
- Smallest Normal: $1.0 \times 2^{-126} = 1.17549435 \times 10^{-38}$

S	EXP								FRAC																						
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Biggest Denormal: $0.11111\ldots \times 2^{-126} = 1.1754942 \times 10^{-38}$

S	EXP								FRAC																						
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

- Smallest Denormal: $0.00\ldots01 \times 2^{-126} = 1.40129 \times 10^{-45}$

S	EXP								FRAC																						
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Example: Value to Bits

- Value: 3.1416
  - Convert to binary: 11.0010010000111l1....
  - Normalize: 1.100100100001111... x 2^1
- Bits:
  - S = 0 (positive)
  - EXP = 1 + 127 = 128 = 0b1000 0000
  - $FRAC$ = 100100100001111....

S	EXP								FRAC																						
0	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	1
4				0				4				9				0				F				F				9			

  - Bits=0x40490FF9

See Also: xmp_float

Example: Bits to Value

- Bits: 0x6703ece6

S	EXP								FRAC																						
0	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1	0	0	1	1	0
6				7				0				3				E				C				E				6			

  - $S = 0$, +
  - $EXP = 0xCE = 206$, $exponent = 206 - 127 = 79$
  - $fraction = 1.0000011111011...$
- Value: + 0b1.0000011111011.. x $2^{79} = 6.23 \times 10^{23}$

What happens when types are mixed?

- Mixed Type Expressions

  `float x; double y; x=y*x;`

- Assignment Statements

  `int x; float y; x=y*3.0;`

- Argument Evaluation

  `int myfn(float x); int y=myfn(3);`

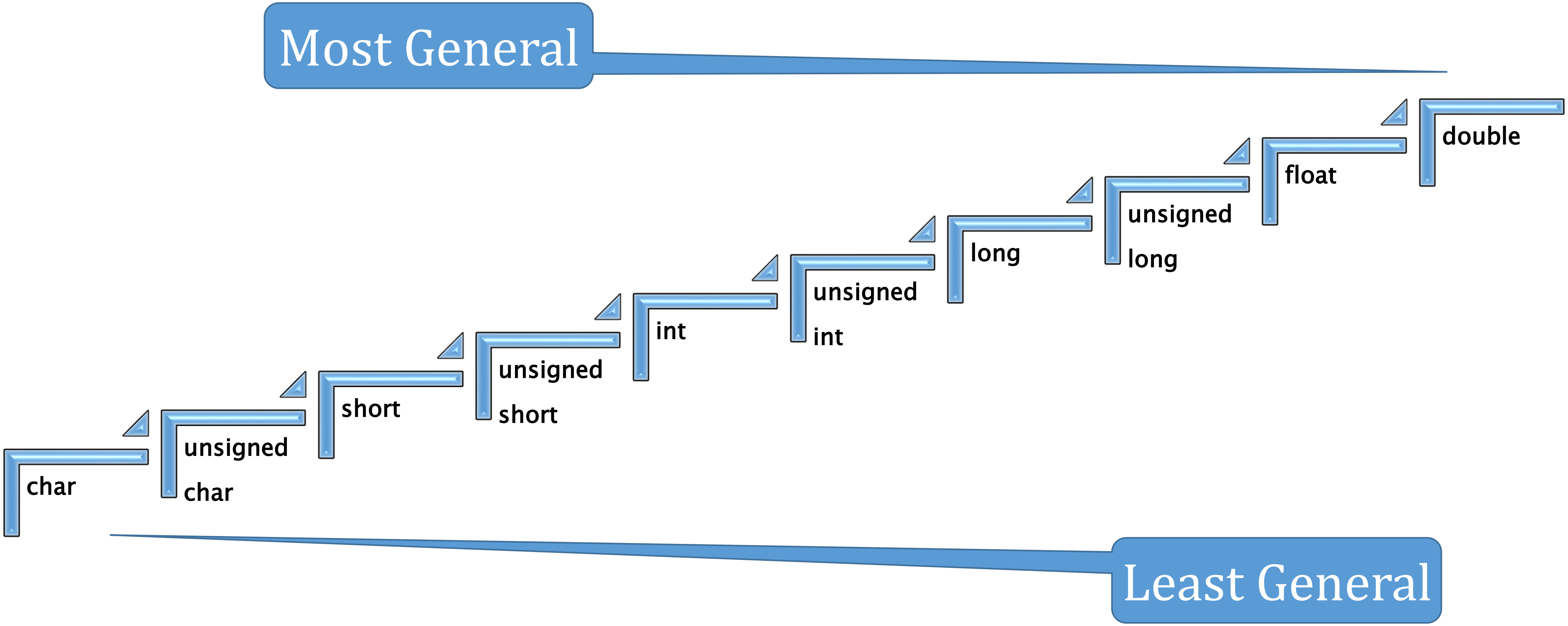
- Explicit Casting

  `int x=7; float y = ((float)x)/3;`

C Automatic type conversion rules

- In an expression (or part of an expression), C converts all components in that expression to the most "general" type, and then evaluates the expression using that general type
- In an assignment (or argument evaluation), C converts the value of the expression to the type of the receiver
- C converts expressions with a valid explicit cast

Generality of Numeric Types



Changing Floating Point Size

- Truncate Fraction or Pad Fraction with 0 on right
- Re-bias exponent
  - If exponent overflows, convert to infinity
- Special case for denormalized numbers!
  - May be denormal in float, but not in double
  - May be denormal in double, but 0 in float

```
float x = 234.34;
double y=x;
```

```
double w = 234.34;
float z=w;
```

x=z=0x436A570A y=0x406D4AE140000000 w=0x406D4AE147AE147B

Integer to Float

- Add .0 and convert to nearest floating point representation

```
int x = 1331254215;
float y=x;
printf("y=%f\n",y);
// prints y=1331254272.000000
```

Float to Integer

- Truncate at the decimal point (round towards zero)

```
float w=-374289.74112;
int z=w;
printf("z=%d\n",z);
// prints z=-374289
```

Conversion Errors

- When C truncates decimals

```
float x=2.7; int y=x; printf(“y = %d\n”,y); // 2
```

- When C approximates floating point

```
float x = 0.2; printf(“x=%.10f\n”,x); //0.2000000029
```

Integer Division Pitfall

```
int atBats = atoi(argv[1]);
int hits = atoi(argv[2]);

float battingAverage = (hits/ atBats) * 1000;

printf("Everybody has a zero batting average?\n");
```

Leak: Associativity

- Law of Associativity: $(A + B) + C = A + (B + C)$
- Floating point approximations can violate associativity!

```
float a=6.022e23;
float b=-a;
float z=3.14;
float p1=(a+b)+c;
float p2=a+(b+c);
if (fabs(p1-p2)<0.5) printf("Associativity holds!");
```

Leak: Multiplication is Repeated Addition

- Abstract: $x \times y = \sum_{i=0}^{y-1} x$ for integer y
- Leak: Rounding error compounds at each operation!

```
float onethird=1.0/3.0;
float sum = 0.0;
int mult=atoi(argv[1]);
for(int i=0;i<mult;i++) sum+=onethird;
float prod = mult * onethird;
printf("Sum is %f, product is %f\n",sum,prod);
```

Resources

- The C Programming Language, (K&R) Sections 2.2 and 2.7
- Wikipedia Type Conversion: https://en.wikipedia.org/wiki/Type_conversion
- C Tutorial – Cast operator: http://www.crasseux.com/books/ctutorial/The-cast-operator.html#The%20cast%20operator
- C-FaQ Floating Point Section: http://c-faq.com/fp/index.html