Simple Data Types in C

Alan L. Cox
alc@rice.edu

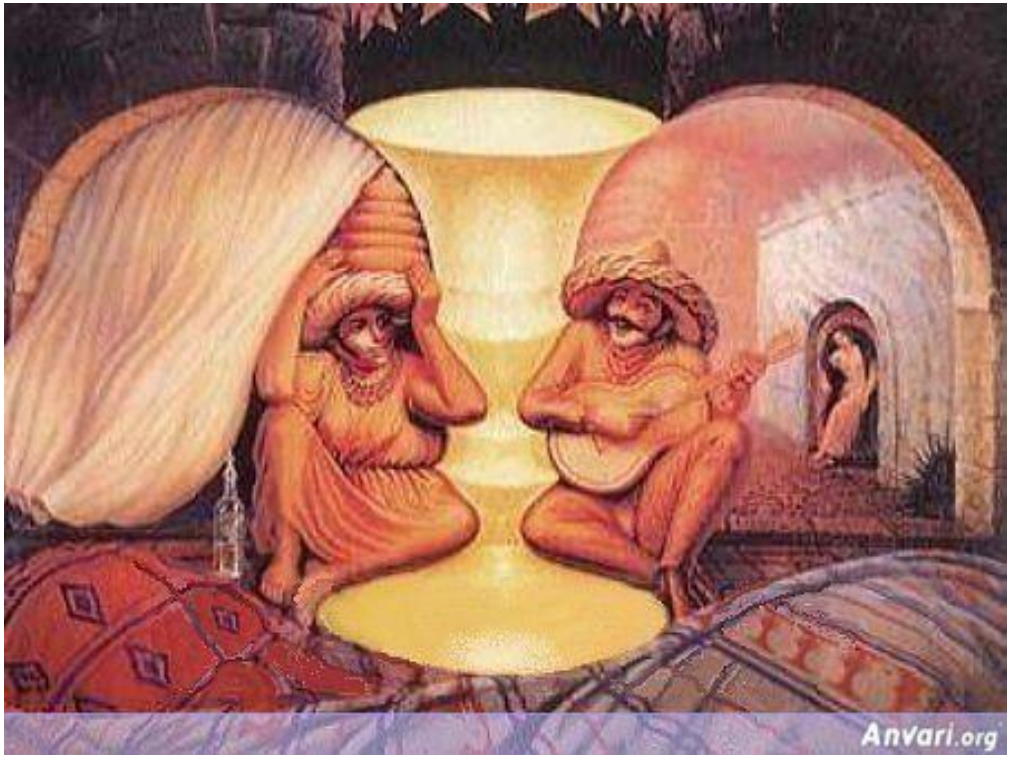
Objectives

Be able to explain to others what a data type is

Be able to use basic data types in C programs

Be able to see the inaccuracies and limitations introduced by machine representations of numbers

What do you see?



What do you see?

Last one

Everything is Just a Bunch of Bits

Bits can represent many different things

- **Depends on interpretation**

<u>You</u> and <u>your program</u> must keep track of what kind of data is at each location in the computer's memory

- **E.g., program data types**

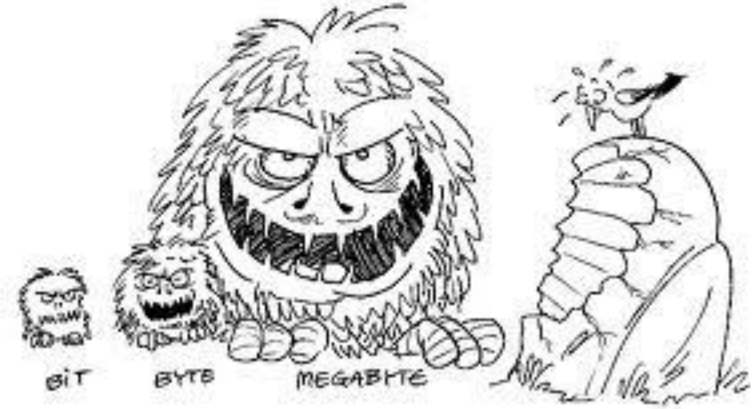
Big Picture

Processor works with <u>finite-sized</u> data

<u>All data</u> implemented as a sequence of *bits*

- **Bit = 0 or 1**
- **Represents the level of an electrical charge**

***Byte* = 8 bits**



***Word* = largest data size handled by processor**

- **32 bits on most older computers**
- **64 bits on most new computers**

Data types in C

Only really four basic types:

- char
- int (short, long, long long, unsigned)
- float
- double

Size of these types on CLEAR machines:

Type	Size (bytes)
char	1
int	4
short	2
long	8
long long	8
float	4
double	8

Sizes of these types *vary* from one machine to another!

Characters (char)

Roman alphabet, punctuation, digits, and other symbols:

- Encoded within one byte (256 possible symbols)
- ASCII encoding (`man ascii` for details)

In C:

```
char  a_char         = 'a';
char  newline_char   = '\n';
char  tab_char       = '\t';
char  backslash_char = '\\';
```

ASCII

Special *control* characters

From “**`man ascii`**”:

```
|  0 NUL|  1 SOH|  2 STX|  3 ETX|  4 EOT|  5 ENQ|  6 ACK|  7 BEL|
|  8 BS |  9 HT | 10 NL | 11 VT | 12 NP | 13 CR | 14 SO | 15 SI |
| 16 DLE| 17 DC1| 18 DC2| 19 DC3| 20 DC4| 21 NAK| 22 SYN| 23 ETB|
| 24 CAN| 25 EM | 26 SUB| 27 ESC| 28 FS | 29 GS | 30 RS | 31 US |
| 32 SP | 33  ! | 34  " | 35  # | 36  $ | 37  % | 38  & | 39  ' |
| 40  ( | 41  ) | 42  * | 43  + | 44  , | 45  - | 46  . | 47  / |
| 48  0 | 49  1 | 50  2 | 51  3 | 52  4 | 53  5 | 54  6 | 55  7 |
| 56  8 | 57  9 | 58  : | 59  ; | 60  < | 61  = | 62  > | 63  ? |
| 64  @ | 65  A | 66  B | 67  C | 68  D | 69  E | 70  F | 71  G |
| 72  H | 73  I | 74  J | 75  K | 76  L | 77  M | 78  N | 79  O |
| 80  P | 81  Q | 82  R | 83  S | 84  T | 85  U | 86  V | 87  W |
| 88  X | 89  Y | 90  Z | 91  [ | 92  \ | 93  ] | 94  ^ | 95  _ |
| 96  ` | 97  a | 98  b | 99  c |100  d |101  e |102  f |103  g |
|104  h |105  i |106  j |107  k |108  l |109  m |110  n |111  o |
|112  p |113  q |114  r |115  s |116  t |117  u |118  v |119  w |
|120  x |121  y |122  z |123  { |124  | |125  } |126  ~ |127 DEL|
```

Characters are just numbers

What does this function do?

return type

procedure name

local variable type and name

```
char
fun(char c)
{
  char new_c;

  if ((c >= 'A') && (c <= 'Z'))
    new_c = c - 'A' + 'a';
  else
    new_c = c;

  return (new_c);
}
```

argument type and name

comparisons with characters!

Math on characters!

Integers

Fundamental problem:

- Fixed-size representation can't encode all numbers

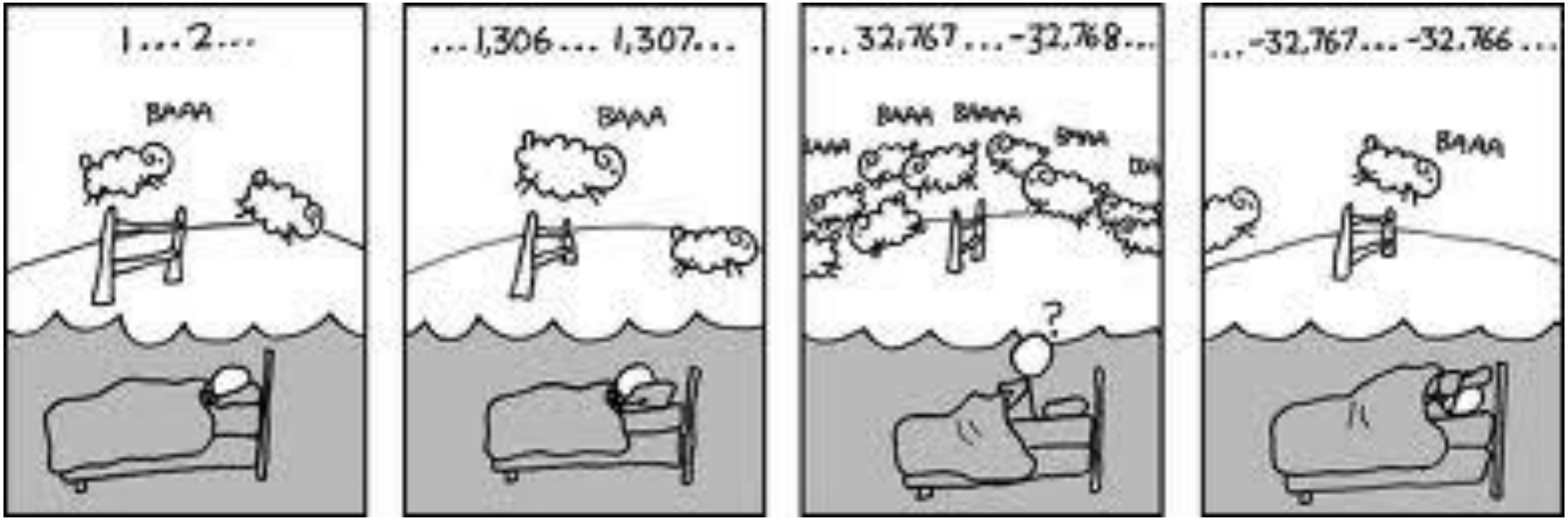
Standard low-level solution:

- Limit number range and precision
  - Usually sufficient
  - Potential source of bugs

Signed and unsigned variants

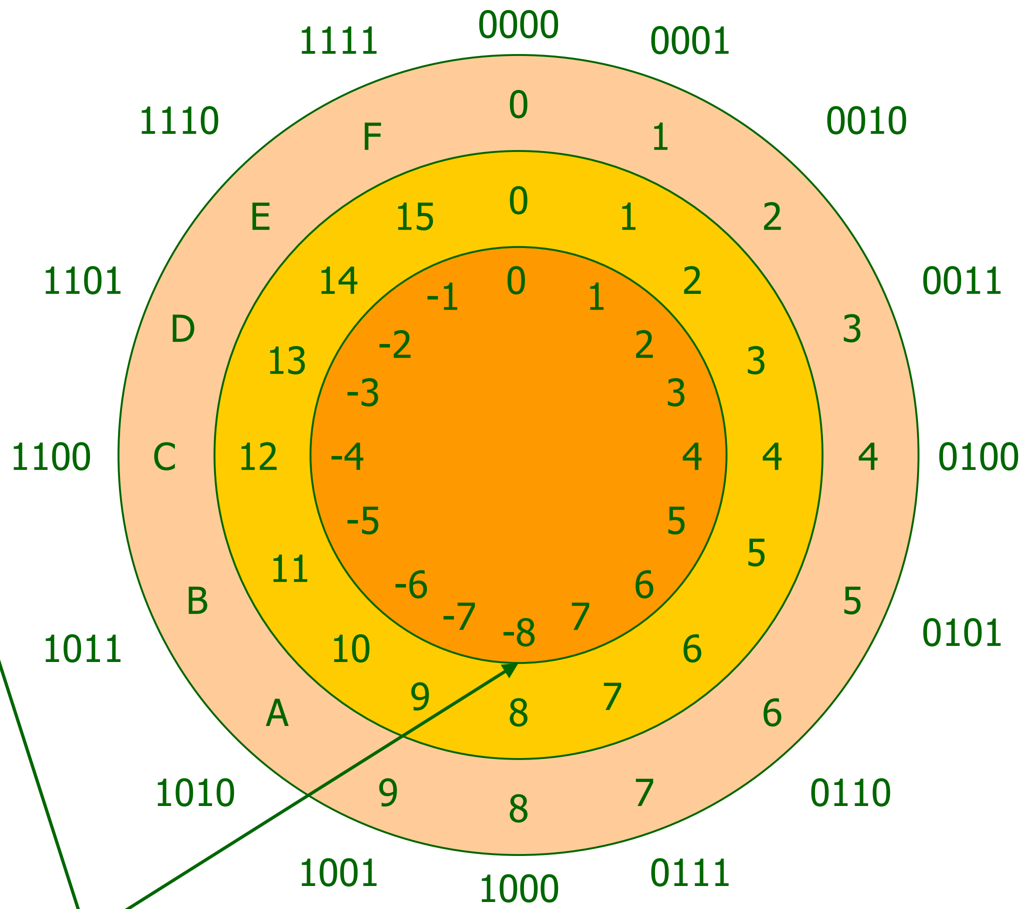
- unsigned modifier can be used with any sized integer (short, long, or long long)

Signed Integer



Integer Representations

Base 2	Base 16	Unsigned	2's Comp.
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1



Why one more negative than positive?

Integer Representations

Base 2	Base 16	Unsigned	2's Comp.
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	-8
1001	9	9	-7
1010	A	10	-6
1011	B	11	-5
1100	C	12	-4
1101	D	13	-3
1110	E	14	-2
1111	F	15	-1

Math for *n* bits:

Define $\vec{x} = x_{n-1} \cdots x_0$

$$B2U(\vec{x}) = \sum_{i=0}^{n-1} 2^i x_i$$

$$B2T(\vec{x}) = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i$$

sign bit
0=non-negative
1=negative

Integer Ranges

Unsigned
$UMin_n$ … $UMax_n$ = 0 … 2^n-1:

32 bits: 0 … 4,294,967,295 `unsigned int`
64 bits: 0 … 18,446,744,073,709,551,615 `unsigned long int`

2's Complement
$TMin_n$ … $TMax_n$ = -2^{n-1} … 2^{n-1}-1:

32 bits: -2,147,483,648 … 2,147,483,647 `int`
64 bits: -9,223,372,036,854,775,808 … 9,223,372,036,854,775,807 `long int`

Note: C numeric ranges are platform dependent!
`#include <limits.h>` **to define** `ULONG_MAX`, `UINT_MIN`, `INT_MAX`, **…**

Detecting Overflow in Programs

Some high-level languages (ML, Ada, …):

- **Overflow causes *exception* that can be *handled***

C:

- **Overflow causes no special event**
- **Programmer must check, if desired**

E.g., given a, b, and $c=UAdd_n(a,b)$ – overflow?

Claim: Overflow iff $c < a$ (Or similarly, iff $c < b$)

Proof: Know $0 \leq b < 2^n$

If no overflow, $c = (a + b) \bmod 2^n = a + b \geq a + 0 = a$

If overflow, $c = (a + b) \bmod 2^n = a + b - 2^n < a$

Overflow

```
unsigned int x = 2123456789U;
unsigned int y = 3123456789U;
unsigned int z;

z = x + y;
```

z is 951,946,282 not 5,246,913,578 as expected

```
int x = 2123456789;
int y = 3123456789;
int z;

z = x + y;
```

y is not a valid positive number (sign bit is set)! It's -1,171,510,507

z is still 951,946,282

However, …

```
#include <assert.h>

void
procedure(int x)
{
…
        assert(x + 10 > x);
```

Should this assertion ever fail?

Do a web search for "GCC bug 30475".

The C language definition does not assume 2's complement as the underlying implementation.

The behavior of signed integer overflow is undefined.

You say, "What should I do?"

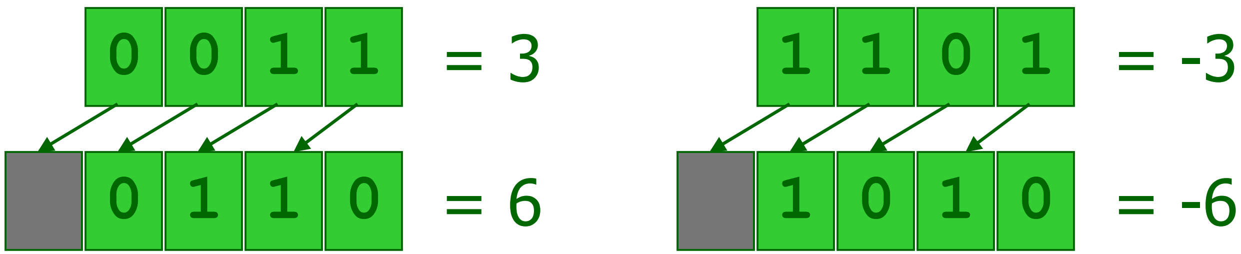
```
#include <limits.h>

/* Safe, signed add.  Won't overflow. */
int
safe_add(int x, int y)
{

      if (y < 0)
            return (safe_sub(x, -y));
      if (INT_MAX - y < x)
            return (INT_MAX); /* Don't overflow! */
      return (x + y);
}
```

Bit Shifting as Multiplication

Shift left (`x << 1`) multiplies by 2:



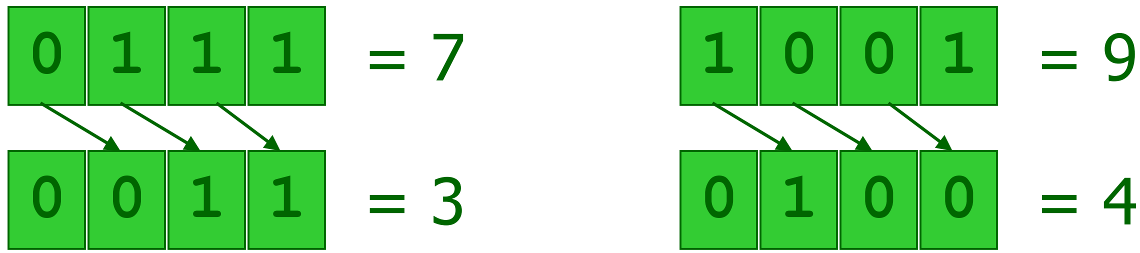
Works for unsigned, 2's complement

Can overflow

In decimal, same idea multiplies by 10: e.g., 42 $\rightarrow$ 420

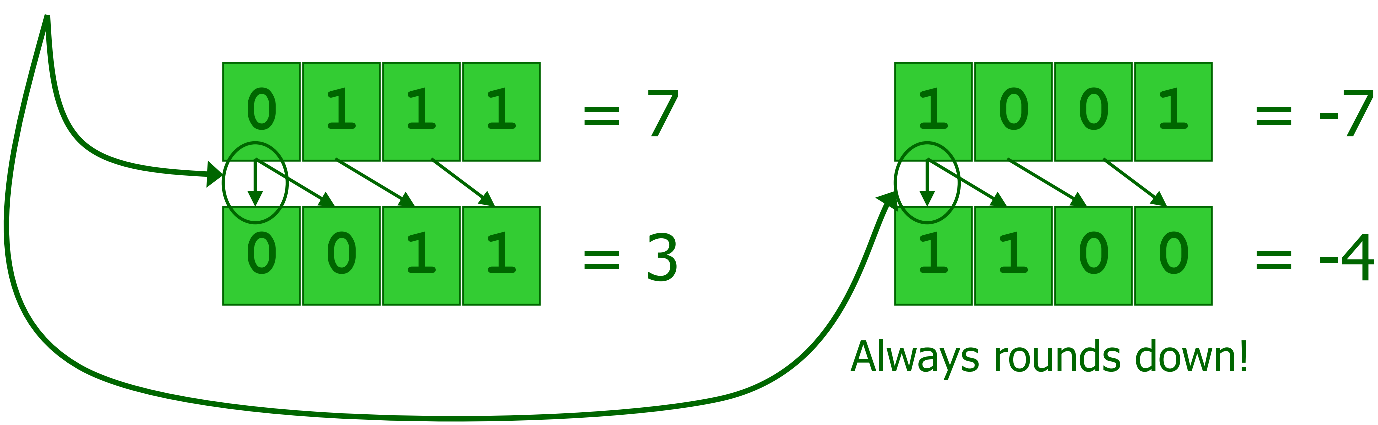
Bit Shifting as Division

Logical shift right (`x >> 1`) divides by 2 for unsigned:



Always rounds down!

Arithmetic shift right (`x >> 1`) divides by 2 for 2's complement:



Always rounds down!

Bit Shifting for Multiplication/Division

Why useful?

- Simpler, thus faster, than general multiplication & division
- Standard compiler optimization

Can shift multiple positions at once:

- Multiplies or divides by corresponding power-of-2
- `a << 5`   `a >> 5`

A Sampling of Integer Properties

For both unsigned & 2's complement:

Mostly as usual, e.g.:

- **0 is identity for +, -**
- **1 is identity for ×, ÷**
- **+, -, × are associative**
- **+, × are commutative**
- **× distributes over +, -**

Some surprises, e.g.:

- **÷ doesn't distribute over +, -**
- **$\neg$ (a,b > 0 $\Rightarrow$ a + b > a)**

<u>Why should you care?</u>

- Programmer should be aware of behavior of their programs
- Compiler uses such properties in optimizations

Beware of Sign Conversions in C

Beware implicit or explicit conversions between unsigned and signed representations!

One of many common mistakes:

```
unsigned int  u;
…
if (u > -1) …
```

What's wrong?

Always false(!) because -1 is converted to unsigned, yielding $UMax_n$

Non-Integral Numbers: How?

Fixed-size representations

- Rational numbers (i.e., pairs of integers)
- Fixed-point (use integer, remember where point is)
- Floating-point (scientific notation)

Variable-size representations

- Sums of fractions (e.g., Taylor-series)
- Unbounded-length series of digits/bits

Floating-point

Binary version of scientific notation

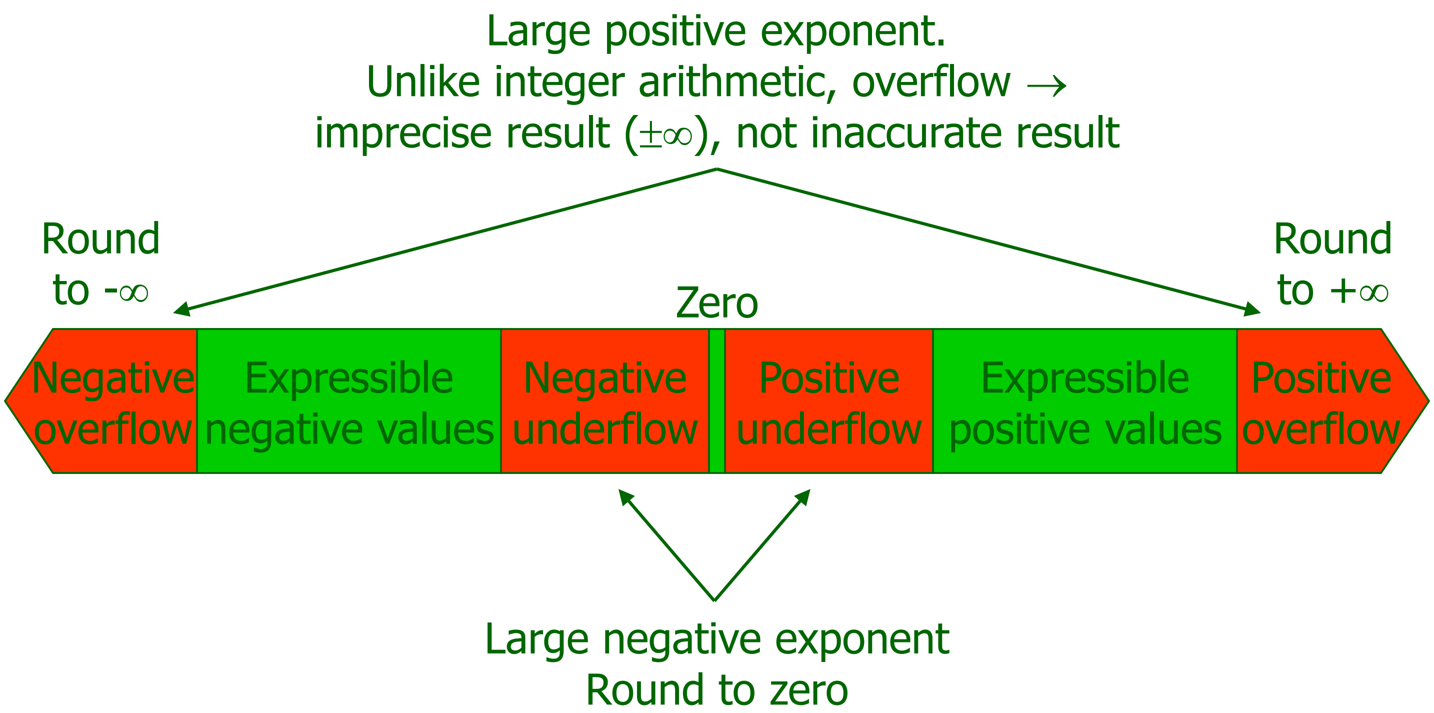
1.001101110×2^5 $= 100110.1110$
$= 32 + 4 + 2 + {}^1/_2 + {}^1/_4 + {}^1/_8$
$= 38.875$

-1.011×2^{-3} $= -.001011$
$= -\left({}^1/_8 + {}^1/_{32} + {}^1/_{64}\right)$
$= -.171875$

binary point

FP Overflow & Underflow

Fixed-sized representation leads to limitations



FP Representation

$$\underbrace{1.001101110}_{\text{significand}} \times 2^{\overset{\uparrow}{5}}$$

exponent

Fixed-size representation

- **Using more significand bits $\Rightarrow$ increased precision**
- **Using more exponent bits $\Rightarrow$ increased range**

Typically, fixed # of bits for each part, for simplicity

FP Representation: IEEE 754

Current standard version of floating-point

Single-precision (float)

- **One word: 1 sign bit, 23 bit fraction, 8 bit exponent**
- **Positive range:** $1.17549435 \times 10^{-38} \ldots 3.40282347 \times 10^{+38}$

Double-precision (double)

- **Two words: 1 sign bit, 52 bit fraction, 11 bit exponent**
- **Positive range:** $2.2250738585072014 \times 10^{-308} \ldots 1.7976931348623157 \times 10^{+308}$

Lots of details in B&O Chapter 2.4

IEEE 754 Special Numbers

+0.0, -0.0

+∞, -∞

NaN: “Not a number”

$(+1.0 \times 10^{+38})^2$	$= +\infty$	$+0.0 \div +0.0$	$=$ NaN
$+1.0 \div +0.0$	$= +\infty$	$+\infty - +\infty$	$=$ NaN
$+1.0 \div -0.0$	$= -\infty$	$\sqrt{-1}$	$=$ NaN

FP vs. Integer Results

```
int    i = 20 / 3;
float  f = 20.0 / 3.0;
```

True mathematical answer: $20 \div 3 = 6\ 2/3$

`i` = ?	6	Integer division ignores remainder
`f` = ?	6.666667	FP arithmetic rounds result

FP vs. Integer Results

```
int     i = 1000 / 6;
float   f = 1000.0 / 6.0;
```

True mathematical answer: $1000 \div 6 = 166\ ^2/_3$

`i` = ?	166	Integer division ignores remainder
`f` = ?	166.666672	FP arithmetic rounds result

Surprise!

Arithmetic in binary, printing in decimal – doesn't always give expected result

FP $\leftrightarrow$ Integer Conversions in C

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

int
main(void)
{
    unsigned int ui = UINT_MAX;
    float f = ui;
    printf("ui: %u\nf: %f\n", ui, (double)f);
}
```

Surprisingly, this program print the following. Why?

```
ui: 4294967295
f: 4294967296.000000
```

FP $\leftrightarrow$ Integer Conversions in C

```
int    i = 3.3 * 5;
float  f = i;
```

True mathematical answer: $3.3 \times 5 = 16\ ½$

`i` = ? 16 Converts $5 \rightarrow 5.0$ – Truncates result $16\ ½ \rightarrow 16$

`f` = ? 16.0

<u>integer $\rightarrow$ FP:</u>

Can lose precision

Rounds, if necessary

32-bit int fits in double-precision FP

<u>FP $\rightarrow$ integer:</u>

Truncate fraction

If out of range, undefined – not error

FP Behavior

Programmer must be aware of accuracy limitations!

Dealing with this is a subject of classes like CAAM 453

$(10^{10} + 10^{30}) + -10^{30}$	$=?$	$10^{10} + (10^{30} + -10^{30})$
$10^{30} - 10^{30}$	$=?$	$10^{10} + 0$
0	$\neq$	10^{10}

Operations not associative!

$(1.0 + 6.0) \div 640.0$	$=?$	$(1.0 \div 640.0) + (6.0 \div 640.0)$
$7.0 \div 640.0$	$=?$	$.001563 + .009375$
$.010937$	$\neq$	$.010938$

$\times, \div$ not distributive across +,-

What about other types?

Booleans

- A late addition to C

Strings

- We' ll cover these in a later class

Enumerated types

- A restricted set of integers

Complex Numbers

- Not covered

Booleans

One bit representation

- 0 is false
- 1 is true

One byte or word representation

- Inconvenient to manipulate only one bit
- Two common encodings:

0000…0000	is false
0000…0001	is true
all other words	are garbage

0000…0000	is false
all other words	are true

- Wastes space, but space is usually cheap

Booleans in C

```
#include <stdbool.h>

bool  bool1  = true;
bool  bool2  = false;
```

Important!
Compiler needs this or it won't know about "bool"!

`bool` **added to C in 1999**

Many programmers had already defined their own Boolean type

- **To avoid conflict** `bool` **is disabled by default**

C's Common Boolean Operations

C extends definitions to integers

- **Booleans are encoded as integers**
  - **0 == false**
  - **non-0 == true**
- **Logical AND:** `0 && 4 == 0` `3 && 4 == 1` `3 && 0 == 0`
- **Logical OR:** `0 || 4 == 1` `3 || 4 == 1` `3 || 0 == 1`
- **Logical NOT:** `! 4 == 0` `! 0 == 1`

&& and || short-circuit

- **Evaluate 2nd argument only if necessary**
- **E.g.,** `0 && error-producing-code == 0`

Enumerated Types

E.g., a Color = red, blue, black, or yellow

- Small (finite) number of choices
- Booleans & characters are common special cases
- Pick arbitrary bit patterns for each

Not enforced in C

- Actually just integers
- Can assign values outside of the enumeration
- Could cause bugs

Enumerated Types in C

```
enum Color { RED, WHITE, BLACK, YELLOW };
enum Color  my_color = RED;
```

The new type name is “`enum Color`”

Alternative style:

```
enum AColor { COLOR_RED, COLOR_WHITE,
    COLOR_BLACK, COLOR_YELLOW };
typedef  enum AColor  color_t;
color_t  my_color = COLOR_RED;
```

Pre-C99 Boolean definition:

```
enum Bool { false = 0, true = 1 };
typedef  enum Bool  bool;
bool  my_bool = true;
```

First Assignment

Carefully read the assignments web page

- Honor code policy
- Slip day policy

All assignments will be posted on that web page

Assignments are due at 11:55PM on the due date, unless otherwise specified

Assignments must be done on CLEAR servers

Next Time

Arrays and Pointers in C