REMEMBER TO REGISTER FOR THE EXAM

http://tenta.angstrom.uu.se/tenta/
How are numbers actually stored?

Some performance consequences and tricks…
Encoding Byte Values

Byte = 8 bits

Binary  \(00000000_2\) to \(11111111_2\)

Decimal: \(0_{10}\) to \(255_{10}\)

Hexadecimal \(00_{16}\) to \(FF_{16}\)

- Base 16 number representation
- Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Write \(\text{FA1D37B}_{16}\) in C as \(0\text{xFA1D37B}\)
  - Or \(0\text{xfalld37b}\)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>
The computer has a “Word Size”

Nominal size of a “natural data unit”

- Including addresses
- Ideally register size == integer size == address size

Most older machines are 32 bits (4 bytes)

- Limits addresses to 4GB

New systems are 64 bits (8 bytes)

- Potentially address $1.8 \times 10^{10}$ GB

Computers can often support multiple data formats

- Fractions or multiples of word size
- Always integral number of bytes
## Data Representations

### Sizes of C Objects (in Bytes)

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>Compaq Alpha</th>
<th>Intel IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>void *</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

(Or any other pointer)

C standard defines “int” as “the most efficient integer type”

Use `sizeof(type)`
The Standard C library header <stdint.h> define types with a specific representation:

int8_t, int16_t, uint8_t, uint32_t, ..

The include file <limits.h> defines constants for the maximum and minimum values:

INT_MAX, INT_MIN, CHAR_MIN, CHAR_MAX, ...
• How should bytes within multi-byte word be ordered in memory?

```c
int64_t x = 0x26AC = C A 6 2 or 2 6 A C
```

little  BIG

• Conventions
  - SPARC, PowerPC tend to be “BigEndian”
  - x86 is “LittleEndian”
  - Alpha, most PowerPC, MIPS, ARM, IA64 are all “switchable endian”
  - Code with strong x86 heritage tends to use little endian
Byte Ordering Example

Big Endian

Least significant byte has highest address

Little Endian

Least significant byte has lowest address

Example

Variable x has 4-byte representation 0x01234567

Address given by &x is 0x100

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>23</td>
<td>45</td>
<td>67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67</td>
<td>45</td>
<td>23</td>
<td>01</td>
</tr>
</tbody>
</table>
The UNIX network header `<netinet/in.h>` contains functions for converting between “network order” (Big endian, TCP/IP standard) and “host order” (might be big or little endian)

```
unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

$ man htonl

`htonl` = “host to network long”
Casting in Endians

```c
int a = 4;
int* ptr = &a;
short b = *(short*) ptr;
if (a == b)
{
    printf("Little endian!" excellent);
}
// else big endian
```

A value that fits within a smaller type is represented correctly in LE.

Only applies to casting pointers. The compiler will always dereference a scalar cast "correctly".
## Encoding Integers

<table>
<thead>
<tr>
<th>signed int</th>
<th>msb</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>0</td>
<td>1111111</td>
</tr>
<tr>
<td>126</td>
<td>0</td>
<td>1111110</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0000010</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0000000</td>
</tr>
<tr>
<td>−1</td>
<td>1</td>
<td>1111111</td>
</tr>
<tr>
<td>−2</td>
<td>1</td>
<td>1111110</td>
</tr>
<tr>
<td>−127</td>
<td>1</td>
<td>0000001</td>
</tr>
<tr>
<td>−128</td>
<td>1</td>
<td>0000000</td>
</tr>
</tbody>
</table>
Integers example

```c
short int x = 15213;
short int y = -15213;
```

C short is 2 bytes long and is `signed`

Signed integers are represented in 2's complement

Most significant bit indicates sign

- 0 for nonnegative
- 1 for negative
Encoding Example (Cont.)

- 2's complement is the bitwise negation plus one.

\[
\begin{align*}
x &= 15213: \\
   &\quad 00111011 \ 01101101 \\
y &= -15213: \\
   &\quad 11000100 \ 10010011
\end{align*}
\]
Unsigned & Signed Numeric Values

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- Addition, subtraction, and multiplication can be done regardless of sign.

<table>
<thead>
<tr>
<th>X</th>
<th>B2U(X)</th>
<th>B2T(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>
Casting Signed to Unsigned

C Allows Conversions from Signed to Unsigned

```c
short int           x =  15213;
unsigned short int ux = (unsigned short) x;
short int           y = -15213;
unsigned short int uy = (unsigned short) y;
```

Resulting Value:

No change in bit representation

Nonnegative values unchanged

- \( ux == 15213 \)

Negative values change into (large) positive values

- \( uy == 50323 \)
#define Constants

By default considered to be signed integers

Unsigned if “U” suffix is used:

- 0U, 4294967259U

Explicit casting between signed & unsigned

- int tx, ty;
- unsigned ux, uy;
- tx = (int) ux;
- uy = (unsigned) ty;

Implicit casting also occurs via assignments and procedure calls

- tx = ux;
- uy = ty;
Why Should I Use Unsigned?

Don’t use just because you think a variable will never be negative.

- After all, it's “int main(int argc, char ** argv)”
- Unsigned ints in array subscripts may perform badly

Do Use When:

- Doing bit masks
- Performing modular arithmetic
- When you need the extra bit of range
Bit-Level Operations in C

Operations &, |, ^, ~ available in C

Apply to any “integral” data type

- long, int, short, char

View arguments as bit vectors, operations applied bit-wise

Examples (Char data type)

- \(~0x41\) \rightarrow 0xBE
- \(~01000001\) \rightarrow 10111110
- \(~0x00\) \rightarrow 0xFF
- \(~00000000\) \rightarrow 11111111
- \(0x69 \ & \ 0x55\) \rightarrow 0x41
- \(01101001 \ & \ 01010101\) \rightarrow 01000001
- \(0x69 \ | \ 0x55\) \rightarrow 0x7D
- \(01101001 \ | \ 01010101\) \rightarrow 01111101
Shift Operations

Left Shift: \( x \ll y \)

- Shift bit-vector \( x \) left \( y \) positions
- Throw away extra bits on left
- Fill with 0’s on right

Right Shift: \( x \gg y \)

- Shift bit-vector \( x \) right \( y \) positions
- Throw away extra bits on right

Logical shift
- Fill with 0’s on left

Arithmetic shift
- Replicate most significant bit on right
- Useful with two’s complement integer representation

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( \ll 3 )</th>
<th>( \gg 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{align*}01100010\end{align*}</td>
<td>\begin{align*}00010000\end{align*}</td>
<td>\begin{align*}00011000\end{align*}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument ( x )</th>
<th>( \ll 3 )</th>
<th>( \gg 2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{align*}10100010\end{align*}</td>
<td>\begin{align*}00010000\end{align*}</td>
<td>\begin{align*}00101000\end{align*}</td>
</tr>
</tbody>
</table>
Set bit \#n:

\[
\text{value} |= (1 \ll n);
\]

Swap bit \#n:

\[
\text{value} ^= (1 \ll n);
\]

Clear bit \#n:

\[
\text{value} &= -1 ^ (1 \ll n);
\]

-1 has all bits set
Swapping numbers

Should do:

```c
int tmp = a;
a = b;
b = tmp;
```

- Popularly believed "efficient" (no temp variable):

```c
a ^= b ^= a ^= b;
```

But this ruins pipelining, and the usual temporary variable is usually optimized away anyway.
## Binary fractions

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5+1/4</td>
<td>101.01</td>
</tr>
<tr>
<td>2+5/8</td>
<td>10.101</td>
</tr>
<tr>
<td>63/64</td>
<td>0.111111</td>
</tr>
</tbody>
</table>

### Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111… equal just below 1.0
  \[ \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \approx 1.0 \]
Limitation of binary representation:

Can only exactly represent numbers of the form $x/2^k$

Other numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/3</td>
<td>0.0101010101010101[01]...</td>
</tr>
<tr>
<td>1/5</td>
<td>0.0011001100110011[0011]...</td>
</tr>
<tr>
<td>1/10</td>
<td>0.00011001100110011[0011]...</td>
</tr>
</tbody>
</table>
IEEE Standard 754

Established in 1985 as uniform standard for floating point computation

Supported by all major CPUs

Established number formats and rules for how to treat the formats.

Can be hard to implement efficiently

- “Numerical Analysis – Computer Architecture: 1-0”
Floating Point Representation

Numerical Form: \(-1^s M \cdot 2^E\)

- Sign bit \(s\) determines whether number is negative or positive
- Significand \(M\) normally a fractional value in range \([1.0, 2.0)\).
- Exponent \(E\) weights value by power of two

Most Significant Bit is sign bit
- \(exp\) field encodes \(E\)
- \(frac\) field encodes \(M\)
Sizes:

Half precision: 5 exp bits, 10 frac bits

Single precision: 8 exp bits, 23 frac bits

Double precision: 11 exp bits, 52 frac bits

Quadruple precision: 15 exp bits, 112 frac bits
Double Precision

- Significand has 53 digits of accuracy *normally*
  - This extra digit = 1 unless exp = 0

- This gives 15 - 17 significant decimal digits precision
- Single precision gives 6-9 significant digits
- Quadruple precision gives 33-36 significant digits!!
What happens with very small values?

Exp = 000…0

Frac begins to have leading zeroes:

- Numbers very close to 0.0 ( < 10^{-308} for doubles )
- Significand loses precision as value gets smaller
- “Gradual underflow”
What happens with very large values?

\[ \exp = 111 \ldots 1 \]

Cases

- \[ \exp = 111 \ldots 1, \frac{}{} = 000 \ldots 0 \]
  - Represents value infinity, \( \infty \), inf
  - Operation that overflows
  - Both positive and negative
  - E.g. \( 1.0/0.0 = -1.0/-0.0 = +\infty \), \( -1.0/0.0 = -\infty \)

- \[ \exp = 111 \ldots 1, \frac{}{} \neq 000 \ldots 0 \]
  - Represents Not-a-Number, NaN
  - Represents case when no numeric value can be determined
  - E.g. \( \sqrt{-1} \), 0/0
Summary of IEEE Floating Point Encodings

Switch between normalized and denormalized
6-bit IEEE-like format

e = 3 exponent bits

f = 2 fraction bits

Bias is 3

Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

6-bit IEEE-like format

- $e = 3$ exponent bits
- $f = 2$ fraction bits
- Bias is 3
• Between $2^{52}=4,503,599,627,370,496$ and $2^{53}=9,007,199,254,740,992$ the representable numbers are exactly the integers.

• From $2^{53}$ to $2^{54}$, everything is multiplied by 2, so every other integer is representable.

• For the range from $2^{51}$ to $2^{52}$, the spacing is $\frac{1}{2}$ integers.

• The spacing as a fraction of the numbers in the range from $2^n$ to $2^{n+1}$ is $2^{n-52}$.

• The maximum relative rounding error when rounding a number to the nearest representable one (the machine epsilon) is $2^{-53} \approx 10^{-16}$. 
Velocity = \( \frac{x_2 - x_1}{t_2 - t_1} \)

Suppose \( t \) measured in seconds, and the machine is turned on for a week?
C Guarantees Two Levels

*float*  single precision  
*double*  double precision

Conversions

*Casting between*  *int*,  *float*,  *and*  *double*  *changes numeric values*

*double*  *or*  *float*  *to*  *int*

  Truncates fractional part

  Not defined when out of range

*int*  *to*  *double*

  Exact conversion, as long as  *int*  is  ≤  53 bits
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_MANT_DIG, FLT_RADIX</td>
<td>Number of digits in the mantissa, and exponent</td>
</tr>
<tr>
<td>FLT_MIN_EXP, FLT_MAX_EXP</td>
<td>Maximum and minimum values of exponent in binary</td>
</tr>
<tr>
<td>FLT_MIN_10_EXP, FLT_MAX_10_EXP</td>
<td>Maximum and minimum values of exponent in decimal</td>
</tr>
<tr>
<td>FLT_EPSILON</td>
<td>Machine epsilon</td>
</tr>
<tr>
<td>FLT_ROUNDS</td>
<td>Rounding mode</td>
</tr>
<tr>
<td>FLT_EVAL_METHOD (c99 only)</td>
<td>Intermediate representation used for arithmetic</td>
</tr>
</tbody>
</table>
-1  Indeterminable
0  Towards zero
1  To nearest
2  Positive infinity
3  Negative infinity
4-  Implementation dependent
-1 Indeterminable

0 Evaluate all operations and constant just to the precision of the type

1 Evaluate operations and constants of type float and double to the range and precision of the double type

2 Evaluate all operations and constants to the range and precision of the long double type

3 Implementation dependent
fenv.h (c99)

Defines the floating-point environment
Get/set rounding modes
Toggle floating point exceptions for division by zero, inexact, invalid, overflow and underflow
Floating-point division and square root are slow.

So are \textit{log, exp, pow, sin, cos}, ...

These tend to be implemented as Newton-Raphson or repeated ranges of polynomial approximations.

Tip: Use squared distances unless sqrt is \textit{absolutely} needed.
Super useful!

Normalize to the unit vector: \( \hat{\vec{r}} = \frac{\vec{r}}{|\vec{r}|} = \frac{\vec{r}}{\sqrt{x^2 + y^2}} \)

Also \( \text{rsqrt}(x) \times x = \text{sqrt}(x) \)

Compute by Newton-Raphson

What about the starting guess?

- Used to be from lookup table, now we use fancy trickery
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalves = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;                       // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalves - ( x2 * y * y ) );   // 1st iteration
    // y = y * ( threehalves - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
/* This is a very approximate but very fast version of rsqrt. It is just two integer instructions (shift right and subtract), plus instructions to load the constant.

The constant 0x5f37642f balances the relative error at \pm 0.034213. */

float rsqrt2(float x) {
    int i = *(int *)&x; // View x as an int.
    i = 0x5f37642f - (i >> 1); // Initial guess.
    x = *(float *)&i; // View i as float.
    return x;
}
Selecting FP type for performance

- **Doubles:**
  - Pro: Doubles incur less roundoff error
  - Pro: Doubles have a larger range
  - Con: Doubles take up $\frac{1}{2}$ of a 128-bit register
  - Con: Doubles require 2x bandwidth
  - Con: Doubles consume 2x cache

- Conclusion: SP is preferred where possible
- Consider Single Precision computation as an optimization method
Single Precision as optimization

- A basic program may use DP as default
- Then:
  - Convert performance-critical parts to SP, or
  - Use SP to get “rough” answer, then refine using DP, kind of like an iterative method
Intuitively, you can:

- Compute a 32 bit result,
- Calculate a correction to 32 bit result using selected higher precision and
- Perform the update of the 32 bit results with the correction using high precision.

Moreover, by limiting your use of SP you can be more aware and responsible about managing round-off error.