How are floats/doubles represented?

- Each number has three parts:
  - its sign \( s \), which is 0 for positive numbers, and 1 for negative numbers
  - a mantissa \( m \), which represents a number between 0 and 1
    - it's represented as a binary number, i.e., \( \frac{1}{2} = 0.1 \)
    - it's normalized into \([1,2)\) (the exponent is adjusted as needed)
  - an exponent \( e \), which designates the position of the decimal point
- A number is \((-1)^s \times m \times r^e\), where \( r \) is the radix
  - the number 6132.789\(_{10} = 1 \times 6.132789 \times 10^3 \) (the radix is 10 for this example)
  - the number 0.05\(_{10} = 1 \times 5.0 \times 10^{-2} \) (radix is also 10)
  - the number -1001.1110\(_2 = -1 \times 1.001110 \times 2^3 \) (here the radix is 2)

- This is much like scientific notation, with the addition of the sign as a factor, and the ability to use a base other than 10
Floating point representation, cont.

- A number can be expressed as \(-1^s \times m \times r^e\), where \(r\) is the radix
- Computers normally use a radix of 2
- Examples of floating point numbers
  
  \[10.5_{10} = 1010.1_2 = .10101 \times 2^4\]
  
  \[7.4375_{10} = 111.0111_2 = .1110111 \times 2^3\]

- Decimal/binary points:

  \[
  \begin{array}{cccccccc}
  10^3 & 10^2 & 10^1 & 10^0 & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} \\
  2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \\
  \end{array}
  \]

The IEEE 754 floating point standard

- The IEEE 754 floating point standard has different sizes for values:
  
  - 32 bit floating point (C float):
    - 1 sign bit, 8 bits exponent, 23 bits mantissa
    - the range of representable values is approximately \(2^{-126} \ldots 2^{128}\), which is approximately \(1.2 \times 10^{-38} \ldots 3.4 \times 10^{38}\)
  
  - 64 bit floating point (C double):
    - 1 sign bit, 11 bits exponent, 52 bits mantissa
    - this is the precision most commonly used for real applications
    - the range of representable values is approximately \(2^{-1022} \ldots 2^{1024}\), which is approximately \(2.2 \times 10^{-308} \ldots 1.8 \times 10^{308}\)
  
  - 128 bit floating point (quad):
    - 1 sign bit, 15 bits exponent, 112 bits of mantissa
    - this is not commonly used

More about IEEE 754 floating-point numbers

- The leading 1 of the mantissa isn't stored:
  - the binary point (like a decimal point) is moved just to the right of the leftmost nonzero digit
  - but in binary, the leftmost nonzero digit must be a 1, so there's no need to actually store it, giving one more bit of precision in the mantissa for free

- The exponent:
  - uses a bias, rather than two's complement, for storing negative as well as positive exponents. The bias is added to the exponent's value.
  - the bias is 127 for single-precision IEEE numbers (C float's), and 1023 for double-precision numbers (C double's)

- The use of a bias allows the representation of the number zero to be all zeros; in fact, an exponent of all 1s or all 0s represents a special number
  - 0, infinities, NaN, denormalized numbers

Example IEEE floating-point number

- Here's how the example number -25.625 is represented in IEEE floating point (single precision):
  
  - The sign bit (one bit) is 1, since the number is negative; we compute the absolute value of the number below
  
  - To compute the mantissa (23 bits):
    - write the number in binary, with a binary point:
      - \(25_{10} = 11001_2\)
      - \(.625_{10} = 1/2 + 1/8\), which is \(.101_2\)
      - so \(25.625_{10} = 11001.101_2\)
    - move the binary point right after the first nonzero digit, giving \(1.1001101\) (moved 4 places to the left)
    - drop the leading 1 (and the binary point), giving 1001101
    - add zeros to the right to get 23 bits (here 16 zeros are needed)
    - so the mantissa is \(10011010000000000000000\)
Example IEEE floating-point number, cont.

- Recall the example number is -25.625
  - To determine the exponent (8 bits):
    - in the previous step, we moved the binary point 4 places to the left to place it to the right of the first nonzero digit, so the exponent value is 4
    - to bias the exponent, we add 127; 127 + 4 = 131, so the value of the exponent field is 131
    - 131 in binary is 10000011
  - Putting it all together, the number is represented as (-1)\(^1\) \times 1.1001101 \times 2^{4} = -1.6015625 \times 16 = -25.625
  - And the number is stored in memory as

```
1 1 0 0 0 0 0 1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
```

Imprecision with real numbers

- The real numbers are dense (unlike the integers), but anything in computer memory has to be stored in a finite bit representation; this causes imprecision
- First consider an analogy with decimal numbers:
  - There are some numbers that can't be represented exactly in a finite number of digits - they require an infinite number of repeating digits
  - Example: 1/3 = .3333333333333...
  - Suppose we have only a fixed number of decimal digits in which to express 1/3, say for example 8 digits. The closest we can get is .33333333. But notice this is .0000000033333... away from the actual number 1/3
  - The next representable number (if we only have 8 digits) is .33333334, and any number between these two can only be approximated as one or the other of these two values- there are no values between them

Imprecision with real numbers, cont.

- In binary there are also real numbers (not necessarily the same ones as in decimal) that can't be represented in a finite number of (binary) digits
  - Example: (1/3)\(_{10}\) = .010101010101010101...
  - Another example: (1/5)\(_{10}\) = .00110011001100110011...
  - If we have only four binary digits, the closest we can come to representing (1/5)\(_{10}\) is .0011 (1/8 + 1/16 = .1875)
  - If we have eight binary digits, we can come closer to representing (1/5)\(_{10}\): .00110011 (1/8 + 1/16 + 1/128 + 1/256 = .19921875). The more digits we have, the closer we can come to representing it
  - But we'll never get exactly to 0.2\(_{10}\), if we only have a fixed number of binary digits in which to represent the number

Imprecision with real numbers, cont.

- The IEEE representation of 1/5, with a 23-digit mantissa, is 0011110010110011001100101, which works out to 0.20000000298023223876953125
- The next smaller bit pattern (only one bit different) is 001111001011001100110011001, which works out to 0.199999988079071044921875000
- There is no (single-precision) IEEE 754 float between these two values because, with a fixed 23 digits of mantissa, there is no bit pattern between them
- If you try to compute or store values between these, such as 0.19999998825, 0.19999998850, 0.19999998875, etc., they'll all be represented as 0011110010011001100110011001100, which is 0.199999988079071044921875000
Another example (a large number)

• The IEEE float 375207,297024.0 is represented as 010100101011101101100000110010
• The next bit pattern is 010100101011101101100000110011, which is the float 375207,329792.0
• These two numbers are 32,768 apart, yet there is no IEEE 754 float value between them

An example 32-bit number

• On a 32-bit machine, consider the bit pattern 11001101011101011100000110010:
  – as an unsigned integer, this bit pattern represents the value 3445054489
  – as a two’s complement signed integer, this bit pattern represents the value -849912807
  – and as a single-precision IEEE float, this bit pattern represents the value -225821072.0
• A pattern of bits can represent lots of different things - we need to know what kind of thing they're supposed to represent to make sense of them
• Given the information above, what does the following code print?
  unsigned int num = 3445054489;
  printf("%f\n", * (float *) &num);

Time

• On a computer there are many ways to measure time
• Conceptual difference:
  – wall time is always running
  – process time is the time your program was running
  • process time doesn't count:
    – the time when your program was stopped for others
    – the time when your program stopped to wait for I/O
  • is composed of:
    – user time: when the OS is running your code
    – system time: when the OS is running system code (handling system calls)
• Difference in how to measure
  – interval time
  – clock cycles
• What time to use depends on what you are measuring
Timing a program

• To time an entire program in the shell:
  – `time program-name program-arguments`
  – runs the program and prints its execution time in the form (tcsh)
    
    2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w
    
    • the first two numbers are user and system time
    • the third number is wall time
    • the fourth is percentage of wall time: (user+system)/wall
    • the remainder are paging and I/O statistics

• This provides some idea about timing
  – but it's hard to know what to do about it - what functions are taking most of the time?

Representing time-of-day

• How to deal with
  – time zones
  – daylight saving time rules
  – computers moving between time zones
  – leap seconds?

• Answer:
  – UNIX keeps time internally as seconds and fractions of a second
    • $2^{32}$ bit values work nicely for 1ns accuracy for > 100 years
  – use a reference starting point
    • called the epoch - midnight Jan. 1, 1970
  – keep all time in UTC form until printed
    • no time zones or daylight saving time to deal with

Date and time functions

• There are several functions in `<time.h>` for working with times.
  – most use a type `time_t` that contains an encoded representation of a time
  – several use the following `tm` structure defined in `<time.h>` that has fields that can be extracted from a `time_t` variable:
    ```c
    struct tm {
      int tm_sec;   /* seconds */
      int tm_min;   /* minutes */
      int tm_hour;  /* hours */
      int tm_mday;  /* day of the month */
      int tm_mon;   /* month */
      int tm_year;  /* year */
      int tm_wday;  /* day of the week */
      int tm_yday;  /* day in the year */
      int tm_isdst; /* daylight saving time */
    }
    ```

• Some common functions:
  ```c
  clock_t clock(void);
  -- returns the process time since the start of program execution
  -- to convert to time, divide by `CLOCKS_PER_SEC` (also in time.h)
  time_t time(time_t *val);
  -- fills `val` with the current time (in an implementation-dependent format)
  char *ctime(time_t *val);
  -- returns a character representation of the passed time
  double difftime(time_t time1, time_t time2);
  -- returns the number of seconds between `time1` and `time2`
  struct tm *gmtime(time_t val);
  -- converts a time to UTC or local time, in the form of a `struct tm`
  struct tm *localtime(time_t val);
  ```
Adding timing calls to your program

- Wall time
  ```c
  int gettimeofday(struct timeval *tv,  
                  struct timezone *tz);
  ```
  - `tv` is a structure of time `tv_sec` and `tv_usec` (10^-6 seconds)
  - `tz` is no longer used (just pass NULL)
- Process time
  ```c
  int getrusage(int who, struct rusage *usage);
  ```
  - `who` is `RUSAGE_SELF` or `RUSAGE_CHILDREN`
    - `RUSAGE_CHILDREN` is all terminated children
  - `rusage` contains fields for
    ```c
    struct timeval ru_utime; /* user time used */  
    struct timeval ru_stime; /* system time used */
    ```
    - and fields for various other OS statistics

Adding timing calls, cont.

- Include `<sys/time.h>` to use `gettimeofday()`
- Include `<sys/time.h>`, `<sys/resource.h>`, and `<unistd.h>` to use `getrusage()`

Example measuring time

```c
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int main() {
  struct rusage start_ru, end_ru;
  struct timeval start_wall, end_wall;
  
  gettimeofday(&start_wall, NULL);
  getrusage(RUSAGE_SELF, &start_ru);
  /* code to time */
  gettimeofday(&end_wall, NULL);
  getrusage(RUSAGE_SELF, &end_ru);
  /* compute difference */
  return 0;
}
```

Calculating the difference of 2 times

- Not trivial, as two fields are involved in each `struct timeval`, but not too complicated
- Example (calculating `end - start`):
  ```c
  struct timeval tv_delta(struct timeval start,  
                         struct timeval end)  
  {
    struct timeval delta = end;
    delta.tv_sec -= start.tv_sec;
    delta.tv_usec -= start.tv_usec;
    if (delta.tv_usec < 0) {
      delta.tv_usec += 1000000;
      delta.tv_sec--;
    }
    return delta;
  }
  ```