CSE 374: Programming Concepts and Tools

Spring 2026

Instructor: Megan Hazen

Today's Goals

Subject Matter

- Some details about
  - Data types
  - Memory management
- Numerical challenges

Your Goals

- Keep working on
  - Debugging
  - HW4 (due TONIGHT)
  - HW5 (due FRIDAY)

Memory Model

- Each byte of memory has an address
- Lowest memory stores program instructions, then global variables (static constants, string literals)
- ‘Heap’ holds dynamically allocated variables (‘new’ or ‘malloc’ variables)
- ‘Stack’ holds current instructions, each function in a frame
  - ‘Stack’ memory implies that a frame is added, and then the last frame added is removed first

Memory (zoomed in)

char – 1 byte int – 4 bytes

- Each byte of memory has an address
- Different data types need different amount of memory to store them
- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.

Memory (zoomed in even more)

0	1	1	0	0	0	0	1

char – 'a' in binary

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	1	0	0	0	0	1

int – 97 in binary

- All data types are stored as binary code: each of the 8 bits in a byte is a zero or one.

Number representation

Base 10

- (10 fingers)
- `234`
- `2x100 + 3x10 + 4x1`
- $2\times10^2 + 3\times10^1 + 4\times10^0$

Base 2

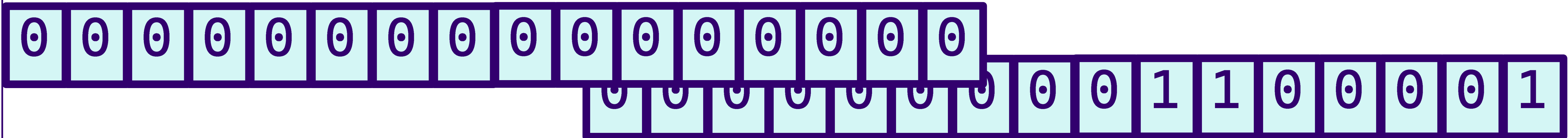
- (binary/digital)
- `234 = 0b11101010`
- $1\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4$ $1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

3 digits in base 10, 8 digits in base 2, 2 digits in base 16

Base 16

- (hexadecimal)
- `234 = 0xEA`
- $14\times16^1 + 10\times16^0$
- Need 16 digits, so use [0-9A-F]

Integer representation

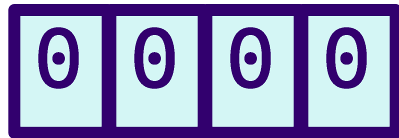


- Hardware (and C) support two different kinds of integers
  - Unsigned (only non-negatives)
  - Signed (both negative and non-negatiave)

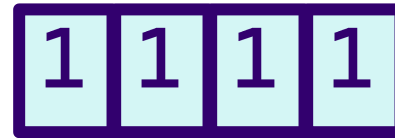
- There are only 2W distinct bit patterns, for W bits, so
  - Can not represent all numbers
  - Unsigned = 0… $2^{W}-1$
  - Signed values = -2^{W-1}… $2^{w-1}-1$

- Left-most == highest-order == most significant
  Right-most == lowest-order == least signficant

4 bit examples



= 0



= 15

- Hardware (and C) support two different kinds of integers
  - Unsigned (only non-negatives)
  - Signed (both negative and non-negatiave)
- There are only 2W distinct bit patterns, for W bits, so
  - Can not represent all numbers
  - Unsigned = $0 \ldots\ 2^{W}-1$
  - Signed values = $-2^{W-1} \ldots\ 2^{w-1}-1$
- Left-most == highest-order == most significant
  Right-most == lowest-order == least signficant

Signed Ints

`0 0 0 1` = 1

`1 0 0 1` = −1

- Obvious solution – reserve most significant bit for the sign
  range = $[-2^{W-1} - 1,\ 2^{W-1} - 1]$
- Math: `0000=0 & 1000=-0`
  Unsigned is normal, signed is…. tricky

```
  0 1 0 1
+ 0 0 1 1
= 1 0 0 0
```

5+3=8

```
  0 1 0 0
+ 1 0 1 1
= 1 1 1 1
```

4+−3=−7 ?

Signed Ints Fixed : twos-complement

`0 0 1 1` = 3

`1 0 1 1` = −5

- Still reserve most significant bit for the sign
  But calculate with 'subtract most significant value'
  range = $[-2^{W-1},\ 2^{W-1}-1]$
- Math: only 1 zero, and addition is easier

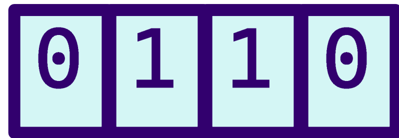
```
  0 1 0 0
- 0 0 1 1
---------
= 0 0 0 1
```

4−3=1

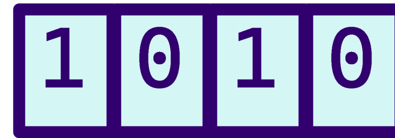
```
  0 1 0 0
+ 1 1 0 1
---------
= 0 0 0 1
```

4+−3=1

Twos-complement



= 6



= −6

- Negate a number by flipping the bits & adding 1
- Use the same algorithm for addition, so we don't need special hardware
  - (Use the 'adder' to do both addition and subtraction – just add a negative #)

```
    0 0 0 1
  + 1 0 1 0
  ---------
  = 1 0 1 1
```

1+−6=−5

Multiplication – bit shifting

0	1	0	1

= 5

0	0	1	1

= 3

- Multiply numbers by bit shifts
  - `Move every bit to the left == multiply by 2`
  - `1x4 == 1<<2      (0 0 0 1) ⇒ (0 1 0 0)`
- For `5x3`, decompose `3` into powers of two:
  `5x3 = ` $5 \times 2^1 + 5 \times 2^0$

```
  0 1 0 1      5<<0 = 5
+ 1 0 1 0      5<<1 = 10
---------
= 1 1 1 1           = 15
```

What about 'overflow'

- Overflow: have numbers too big or small for your number of digits.
- (Remember, using 4 bits, unsigned = [0,15] and signed [-8,7]
- 6+4 = ? (signed) 15+2 = ? (unsigned)
- -6 - 6 = ? (signed) 12-14 = ? (unsigned)
- *Notes: You may get a warning for overflow with two-complement numbers, but probably not with unsigned numbers.*

```
  0 1 1 0
+ 0 1 0 0
---------
= 1 0 1 0   =−6
```

```
  1 1 1 1
+ 0 0 1 0
---------
= 0 0 0 1   =1
```

```
  1 0 1 0
+ 1 0 1 0
---------
= 0 1 0 0   =4
```

```
  1 1 0 0
− 1 1 1 0
---------
= 1 1 1 0   =14
```

Type-casting

Convert one type to another

- Syntax: `(t)e` where `t` is a type and `e` is an expression (same as Java)
- If e is a numeric type and t is a numeric type, this is a conversion
  - To wider type, get same value
  - To narrower type, may not (will get mod)
  - From floating-point to integer, will round (may overflow)
  - From integer to floating-point, may round (but int to double is exact on most machines)

```
main() {
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count;
    printf("Mean val: %f\n", mean );
}
```

C: 'int' and 'unsigned'

```
int tx, ty;
unsigned ux, uy;
```

- Explicit casting between signed & unsigned:
```
tx = (int) ux;
uy = (unsigned) ty;
```
- Implicit casting also occurs via assignments and function calls:
```
tx = ux;
uy = ty;
```
- The gcc flag `-Wsign-conversion` produces warnings for implicit casts, but `-Wall` does not!

- Explicit casting - doesn't change underlying bits, they just get interpreted differently! This is NOT taking the absolute value.
- Note: C doesn't dictate the integer representation method, the compiler does. Casting an integer to unsigned will result in different values depending on that choice.
- Note: in C, constants are assumed to be signed, unless the 'U' suffix is used: 15U -> 15 unsigned

Floating point numbers

- Fractional binary numbers work in the same fashion as fractional decimal numbers
  - $1.25 = 1 \cdot 10^{0} + 2 \cdot 10^{-1} + 5 \cdot 10^{-2}$
  - $0b1.01 = 1 \cdot 2^{0} + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + 1/4 = 1.25$
- can have repeating just like decimal
  - `1/10 = 0b0.0001100110011[0011 ]…`
- floating point values only represent numbers that can be written $x \cdot 2^{y}$
- like scientific notation
  - not `0b0.000101` but $1.01 \cdot 2^{-4}$
- Floating point standard established
  - 1985, IEEE 754 - before that every system had a different approach

Floating point representation

- Numerical form: $V_{10} = (-1)^s * M * 2^E$
  - Sign bit **s** determines whether number is negative or positive
  - Significand (mantissa) **M** normally a fractional value in range [1.0,2.0)
  - Exponent **E** weights value by a (possibly negative) power of two

Floating point representation 2

Numerical form: V_{10} = **$(-1)^s * M * 2^E$**

Single precision (32 bits/ 4 bytes) : s = 1 bit, E = 8 bits, M = 23 bits

Double precision (64 bits) : s = 1 bit, E = 11 bits, M = 52 bits

Due to finite number of bits, some numbers approximated

Special values (these can pollute numerical computation):

zero : s=0, E=0, M=0
+/1 inf: E = all ones, M = 0
NaN : E = all ones, M != 0

Floating point rules

- Understand limited number of bits
  - Can get overflow just like the integers
- Some simple factions have no exact representation (such a `0.2`. Why?)
- You can lose precision – every operation may introduce some small error
- Mathematically equivalent expressions may compute different results
- Violates associativity / distributivity
- **Never test floating point numbers for equality**
- **Careful when converting between floats and ints**

Floating points in C

- Two levels of precision
  - Float (4 bytes)
  - Double (8 bytes)
- You must use `#include <math.h>` to get `INF` and `NaN`
  - (you'll need to explicitly link at compile time: `gcc -lm module.c`
- Avoid (`==`) comparisons

Implicit Casting

- Compiler converts one type to another
  - During arithmetic
  - Convert R-value to L-value
- Type promotion – convert a 'lower' type to a 'higher' type
  - int-> float : may be rounded
  - int -> double or float -> double EXACT
- Demotion – convert a 'higher' type to a 'lower' type
  - May lose some information
  - Floats are *truncated* to ints

Type ranks:

1. **`Char`**
2. **`Short`**
3. **`Int`**
4. **`Unsigned Int`**
5. **`Long`**
6. **`Unsigned Long`**
7. **`Float`**
8. **`Double`**
9. **`Long double`**

Remember Hexadecimal?

Base 10

- (10 fingers)
- `234`
- `2x100 + 3x10 + 4x1`
- $2x10^2 + 3x10^1 + 4x10^0$

Base 2

- (binary/digital)
- `234 = 0b11101010`
- $1x2^7 + 1x2^6 + 1x2^5 + 0x2^4\ 1x2^3 + 0x2^2 +1x2^1 + 0x2^0$

3 digits in base 10, 8 digits in base 2, 2 digits in base 16

Base 16

- (hexadecimal)
- `234 = 0xEA`
- $14x16^1 + 10x16^0$
- Need 16 digits, so use [0-9A-F]

Hexadecimal in C

- There is no unique type for hexadecimal in C. We use '`unsigned int`' or '`unsigned char`'.
  Remember, `sizeof(int) = 4` [bytes]
  and `sizeof(char) = 1` [byte] => **2 hex digits**
- An unsigned char can hold values up to 255 or 0xFF (maximum two digit hex value)

```
unsigned char ahexvalue = 0xFE;
uintptr_t mymem = (uintptr_t) malloc (16);
for (int i = 0; i < 16; i++) {
    *((unsigned char*)(mymem+i)) = 0xFE;
}
```

Wait, what was that code?

```
unsigned char ahexvalue = 0xFE;
uintptr_t mymem = (uintptr_t) malloc (16);
for (int i = 0; i < 16; i++) {
    *((unsigned char*)(mymem+i)) = 0xFE;
}
```

We can use '`uintptr_t`' as a type to hold a memory address as an integer, and add or subtract bytes. The loop moves forward one byte at a time.

Defined as an integer type that can hold an address pointer – convert from a void pointer, and then be converted back with no loss of information.

Will port across systems, with `#include <stdint.h>`

Derived Types and User-Defined Types

- Pointer types
  - Require space to hold an address (so, 8 bytes)
  - But negative addresses don't make sense (so, unsigned)
  - Useful type defined to meet these qualifications: `uintptr_t`
- User-defined types (such as structs)
  - Require space to hold multiple primitive (or other user-defined) types
  - We can calculate byte-requirements
  - Memory manager places some restrictions that affect actual space usage

Struct Memory Alignment

- Structs are allotted contiguous memory.
- Position in memory dictated by order of declaration
- HOWEVER, it is more efficient to align addresses with multiples of type widths.
  - ints - address multiple of 4
  - doubles - address multiple of 8
  - Pointers - address multiple of 8
- Entire struct size guided by largest data type it contains

```
struct studenta {
    char *name;
    char section;
    int late_days;
    double grade;
};
struct studentb {
    char *name;
    char section;
    double grade;
    int late_days;
};
```

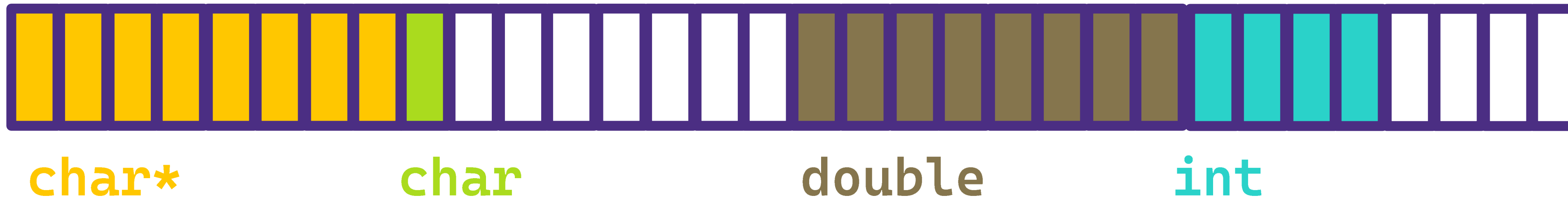
Struct Memory Alignment studenta

char* char int double

studenta requires a total of 24 bytes

```
struct studenta {
      char *name;
      char section;
      int late_days;
      double grade;
};
struct studentb {
      char *name;
      char section;
      double grade;
      int late_days;
};
```

Struct Memory Alignment studentb



studentb requires a total of 32 bytes

always use sizeof() !

```
struct studentb {
      char *name;
      char section;
      double grade;
      int late_days;
};
```