Week 02, Monday

Memory and Data

The C View of Data

2/33

A C program sees data as a collection of *variables*

```
int g = 2;

int main(void)
{
    int   i;
    int   v[5]
    char *s = "Hello";
    int  *n = malloc(sizeof(int));
    ...
}
```

Each variable has a number of properties (e.g. name, type, size)

... The C View of Data

3/33

Variables are examples of *computational objects*

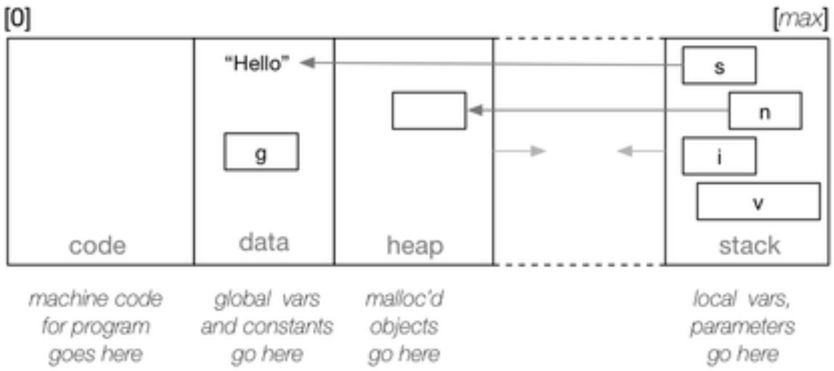
Each computational object has

- a *location* in memory
- a *value* (ultimately just a bit-string)
- a *name* (unless created by `malloc()`)
- a *type*, which determines ...
  - its *size* (in units of whole bytes, `sizeof`)
  - how to *interpret* its value
  - what *operations* apply to the value
- a *scope* (where it's visible within the program)
- a *lifetime* (during which part of program execution it exists)

... The C View of Data

4/33

C allocates data objects to various well-defined regions of memory
during program execution



Exercise 1: Properties of Variables

5/33

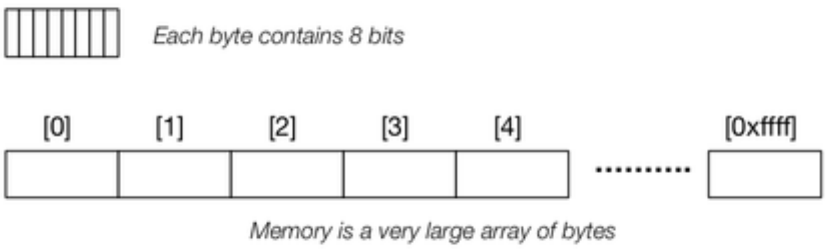
Identify the properties of each of the named objects in the following:

```
int a;          // global int variable

int main(void) {
   int  b;      // local int variable
   char c;      // local char variable
   char d[10];  // local char array
   ...
}

int e;          // global? int variable

int f(int g) {  // function + parameter
   double h;    // local double variable
   ...
}
```

The Physical View of Data

6/33

Memory = indexed array of bytes



Indexes are "memory addresses" (a.k.a. pointers)

Data can be fetched in chunks of 1,2,4,8 bytes

Memory

7/33

Also called: RAM, main memory, primary storage, ...

Technology: semiconductor-based

Distinguishing features

- relatively large (e.g. 2^{28} bytes)
- any byte can be fetched with same cost
- cost of fetching 1,2,4,8 bytes is small (ns)

Two properties related to data persistence

- *volatile* (e.g. DRAM) ... data lost when powered off
- *non-volatile* (e.g. EEPROM) ... data stays when powered off

---

... Memory

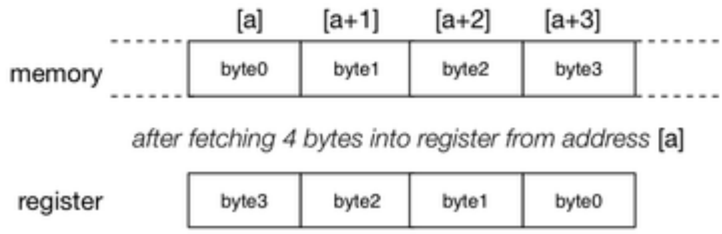
8/33

When addressing objects in memory

- any byte address can be used to fetch 1-byte object
- byte address for *N*-byte object must be divisible by *N*

Data is fetched into *N*-byte CPU registers for use

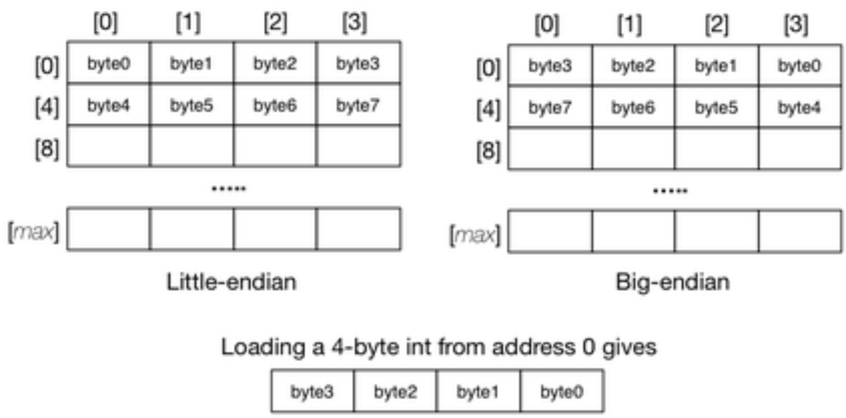
Data bytes in registers may be in different order to memory, e.g.



... Memory

9/33

Memories can be categorised as *big-endian* or *little-endian*



Data Representation

Data Representation

11/33

Ultimately, memory allows you to

- load bit-strings of sizes 1,2,4,8 bytes
- from *N*-byte boundary addresses
- into registers in the CPU

What you are presented with is a string of 8,16,32,64 bits

Need to *interpret* this bit-string as a meaningful value

Data representations provide a way of assigning meaning to bit-strings

---

Character Data

12/33

Character data has several possible representations (encodings)

The two most common:

- ASCII (ISO 646)
  - 7-bit values, using lower 7-bits of a byte (top bit always zero)
  - can encode roman alphabet, digits, punctuation, control chars
- UTF-8 (Unicode)
  - 8-bit values, with ability to extend to multi-byte values
  - can encode all human languages plus other symbols

  (e.g. √ ∑ ∀ ∃ or 🤑 🤗 🤓 😎 🤡 🤠)

ASCII Character Encoding

13/33

Uses values in the range `0x00` to `0x7F` (0..127)

Characters partitioned into sequential groups

- control characters (0..31) ... e.g. `'\0'`, `'\n'`
- punctuation chars (32..47,91..96,123..126)
- digits (48..57) ... `'0'..'9'`
- upper case alphabetic (65..90) ... `'A'..'Z'`
- lower case alphabetic (97..122) ... `'a'..'z'`

In C, can map between char and ascii code by e.g. `((int)'a')`

Sequential nature of groups allow for e.g. `(ch - '0')`

... ASCII Character Encoding

14/33

Hexademical ASCII char table (from `man 7 ascii`)

```
00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

`0x0a` = `'\n'`, `0x20` = `' '`, `0x09` = `'\t'`, but note no `EOF`

Exercise 2: Using 'a'..'z' as indexes

15/33

Write C code that allows you to treat an array like

```
int freq[26];
```

as if it were indexed by `'a'..'z'`

Sample usage

```
for (char c = 'a'; c <= 'z'; c++)
   freq[XXX] = 0;
...
for (char c = 'a'; c <= 'z'; c++)
   printf("%s has freq %d\n", c, freq[XXX]);
```

In other words, replace the `XXX` by an index calculation

UTF-8 Character Encoding

16/33

UTF-8 uses a variable-length encoding as follows

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	`0xxxxxxx`	-	-	-
2	11	`110xxxxx`	`10xxxxxx`	-	-
3	16	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	-
4	21	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

The 127 1-byte codes are compatible with ASCII

The 2048 2-byte codes include most Latin-script alphabets

The 65536 3-byte codes include most Asian languages

The 2097152 4-byte codes include symbols and emojis and ...

... UTF-8 Character Encoding

17/33

UTF-8 examples

ch	unicode	bits	simple binary	UTF-8 binary
$	U+0024	7	`010 0100`	`00100100`
¢	U+00A2	11	`000 1010 0010`	`11000010 10100010`
€	U+20AC	16	`0010 0000 1010 1100`	`11100010 10000010 10101100`
☐	U+10348	21	`0 0001 0000 0011 0100 1000`	`11110000 10010000 10001101 10001000`

Unicode strings can be manipulated in C (e.g. "안녕하세요")

Like other C strings, they are terminated by a 0 byte (i.e. `'\0'`)

Exercise 3: Measuring UTF-8 Strings

18/33

Write C functions that count

- the number of bytes in a Unicode string
- the number of "symbols" in a Unicode string

Use the function templates

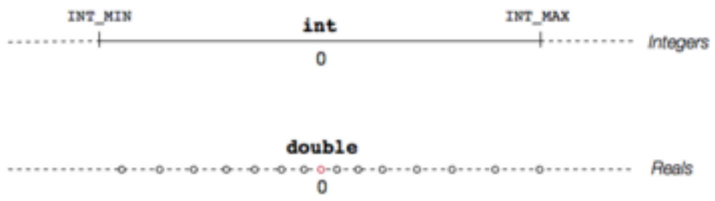
```
int unicodeNbytes(char *str) { ... }

int unicodeNsymbols(char *str) { ... }
```

Numeric Data

19/33

Numeric data comes in two major forms

- integer ... subset (range) of the mathematical integers
- floating point ... subset of the mathematical real numbers



Integer Constants

20/33

Three ways to write integer constants in C

- `42` ... signed decimal (`0..9`)
- `0x2A` ... unsigned hexadecimal (`0..F`)
- `052` ... signed octal (`0 ..7`)

Variations

- `123U` ... `unsigned int` value (typically 32 bits)
- `123L` ... `long int` value (typically 64 bits)
- `123S` ... `short int` value (typically 16 bits)

Invalid constants lie outside the range for their type, e.g.

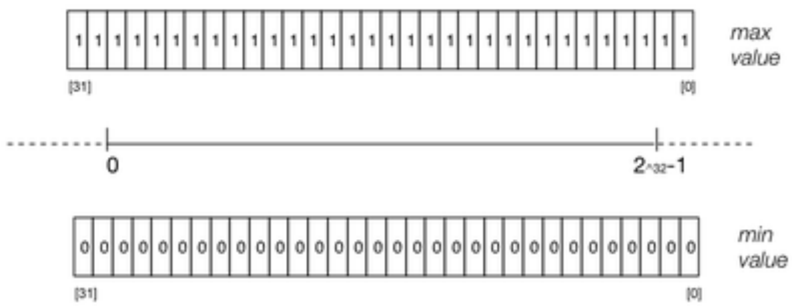
- `4294967296, -1U, 666666S, 078`

Unsigned integers

21/33

The **`unsigned int`** data type

- commonly 32 bits, storing values in the range 0 .. 2^{32}-1



... Unsigned integers

22/33

Value interpreted as binary number

E.g. consider an 8-bit unsigned int

`01001101` = $2^6 + 2^3 + 2^2 + 2^0$ = 64 + 8 + 4 + 1 = 77

Addition is bitwise with carry

```
  00000001     00000001     01001101     11111111
+ 00000010   + 00000011   + 00001011   + 00000001
  --------     --------     --------     --------
  00000011     00000100     01011000     00000000
```

Most machines will also flag the *overflow* in the fourth example

Exercise 4: Binary↔decimal Conversion

23/33

Convert these 8-bit binary numbers to hexadecimal:

- `00001001`, `00001101`, `00101010`, `00110011`, `11001100`

Convert these 8-bit binary numbers to decimal:

- `00001001`, `00001101`, `00101010`, `00110011`, `11001100`

Convert the following decimal numbers to 8-bit binary:

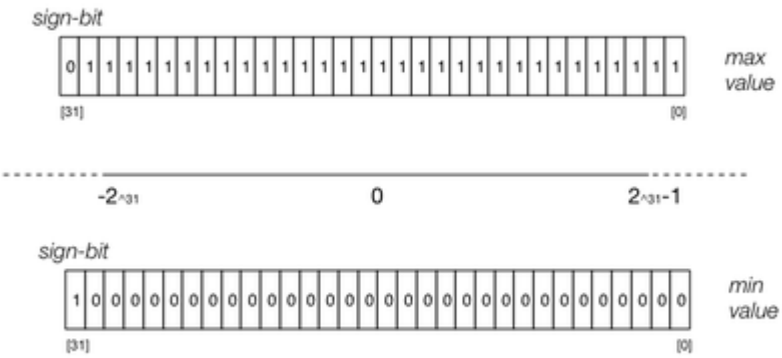
- 15, 64, 99, 200, 256

Signed integers

24/33

The **`int`** data type

- commonly 32 bits, storing values in the range -2^{31} .. $2^{31}-1$



... Signed integers

25/33

Several possible representations for negative values

- signed magnitude ... first bit is sign, rest are magnitude
- ones complement ... form *-N* by inverting all bits in *N*
- twos complement ... form *-N* by inverting *N* and adding 1

In all representations, +ve numbers have 0 in leftmost bit

Examples: representations of (8-bit) -5 (where 5 is `00000101`)

- `10000101` ... signed magnitude
- `11111010` ... ones complement
- `11111011` ... twos complement

... Signed integers

26/33

Signed magnitude: Easy to form *-X* from *X* ... OR in high-order bit

A problem (using 8-bit `int`s) ...

- what do these numbers represent? `00000000`, `10000000`

Two zeroes ... one positive, one negative

Another problem: $x + -x \neq 0$ (mostly) with simple addition

```
  00000011   3     00101010   42      01111111   127
+ 10000011  -3   + 10101010  -42    + 11111111  -127
  --------         --------           --------
  10000110  !0     11010100   !0      01111110    !0
```

To fix requires extra hardware in ALU

... Signed integers

27/33

Ones complement: Easy to form *-X* from *X* ... NEG all bits

A problem (using 8-bit `ints`) ...

- what do these numbers represent? `00000000, 11111111`

Two zeroes ... one positive, one negative

At least *x + -x* is equal to one of the zeroes with simple addition

```
  00000011  3     00101010  42      01111111
+ 11111100 -3   + 11010101 -42    + 10000000
  --------        --------          --------
  11111111 !0     11111111  !0      11111111 -0
```

... Signed integers

28/33

Twos complement: to form *-X* from *X* ... NEG all bits, then add 1

Now have only one representation for zero (`00000000`)

- `-0 = ~00000000+1 = 11111111+1 = 00000000`

Only one zero value. Also, *-(-x) = x*

Even better, *x + -x = 0* in all cases with simple addition

```
  00000011  3     00101010  42      01111111
+ 11111101 -3   + 11010110 -42    + 10000001
  --------        --------          --------
  00000000  0     00000000   0      00000000  0
```

Always produces an "overflow" bit, but can ignore this

Exercise 5: Binary↔decimal Conversion

29/33

What decimal numbers do these 8-bit twos complement numbers represent:

- `10001001, 10001101, 10101010, 10110011, 11001100`

Convert the following decimal numbers to 8-bit binary:

- 15, 64, 99, 127, 128

Show signed magnitude, 1's complement and 2's complement

Demonstrate the addition of *x + -x*, where *x* is

- 5, 20, 64, 99, 127

Exercise 6: Integer Powers

30/33

C does not have a power operator (e.g. like `x**y` = x^y)

Write a function to compute x^y

```
int raise(int x, int y) { ... }
```

Write a specialised version to compute 2^y

```
int powOf2(int y) { ... }
```

Pointers

31/33

Pointers represent memory addresses/locations

- number of bits depends on memory size, but typically 32-bits
- data pointers reference addresses in *data*/*heap*/*stack* regions
- function pointers reference addresses in *code* region

Many kinds of pointers, one for each data type

- `sizeof(int *) = sizeof(char *)`
  `= sizeof(double *) = sizeof(struct X *)`

Pointer values must be appropriate for data type, e.g.

- `(char *)` ... can reference any byte address
- `(int *)` ... must have *addr* `%4 == 0`
- `(double *)` ... must have *addr* `%8 == 0`

... Pointers

32/33

Can "move" from object to object by *pointer arithmetic*

For any pointer *T* `*p;`, `p++` increases `p` by `sizeof(`*T*`)`

Examples (assuming 16-bit pointers):

```
char   *p = 0x6060;  p++;  assert(p == 0x6061)
int    *q = 0x6060;  q++;  assert(q == 0x6064)
double *r = 0x6060;  r++;  assert(r == 0x6068)
```

A common (efficient) paradigm for scanning a string

```
char *s = "a string";
char *c;
// print a string, char-by-char
for (c = s; *c != '\0'; c++) {
   printf("%c", *c);
}
```

Exercise 7: Sum an array of `ints`

33/33

Write a function

```
int sumOf(int *a, int n)  { ... }
```

to sum the elements of array `a[]` containing `n` values.

Implement it two ways:

- using the "standard" approach with an index
- using a pointer that scans the elements

Produced: 3 Aug 2017