10 types of students

There are only 10 types of students ...

- those that understand binary
- those that don’t understand binary

Decimal Representation

- Can interpret decimal number 4705 as:
  \[4 \times 10^3 + 7 \times 10^2 + 0 \times 10^1 + 5 \times 10^0\]

- The base or radix is 10 ... digits 0 – 9

- Place values:

  \[
  \begin{array}{cccc}
  \ldots & 1000 & 100 & 10 & 1 \\
  \ldots & 10^3 & 10^2 & 10^1 & 10^0 \\
  \end{array}
  \]

- Write number as 4705_{10}
  - Note use of subscript to denote base
Represenation in Other Bases

- base 10 is an arbitrary choice
- can use any base
- e.g. could use base 7
- Place values:

<table>
<thead>
<tr>
<th>...</th>
<th>343</th>
<th>49</th>
<th>7</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>7^3</td>
<td>7^2</td>
<td>7^1</td>
<td>7^0</td>
</tr>
</tbody>
</table>

- Write number as 1216\_7 and interpret as:
  
  \[1 \times 7^3 + 2 \times 7^2 + 1 \times 7^1 + 6 \times 7^0 = 454\_10\]

Binary Representation

- Modern computing uses binary numbers
  - because digital devices can easily produce high or low level voltages which can represent 1 or 0.
- The base or radix is 2
- Digits 0 and 1
- Place values:

<table>
<thead>
<tr>
<th>...</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>2^3</td>
<td>2^2</td>
<td>2^1</td>
<td>2^0</td>
</tr>
</tbody>
</table>

- Write number as 1011\_2 and interpret as:
  
  \[1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11\_10\]

Hexadecimal Representation

- Binary numbers hard for humans to read — too many digits!
- Conversion to decimal awkward and hides bit values
- Solution: write numbers in hexadecimal!
- The base or radix is 16 ...
- Digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Place values:

<table>
<thead>
<tr>
<th>...</th>
<th>4096</th>
<th>256</th>
<th>16</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>16^3</td>
<td>16^2</td>
<td>16^1</td>
<td>16^0</td>
</tr>
</tbody>
</table>

- Write number as 3AF\_16 and interpret as:
  
  \[3 \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 1 \times 16^0 = 15089\_10\]
- in C, 0x prefix denotes hexadecimal, e.g. 0x3AF1
Octal & Binary C constants

- Octal (based 8) representation used to be popular for binary numbers
- Similar advantages to hexadecimal
- In C, a leading 0 denotes octal, e.g. 07563
- Standard C doesn’t have a way to write binary constants
- Some C compilers let you write 0b
  - OK to use 0b in experimental code but don’t use in important code

```
printf("%d", 0x2A); // prints 42
printf("%d", 052);  // prints 42
printf("%d", 0b101010); // might compile and print 42
```

Binary Constants

In hexadecimal, each digit represents 4 bits

```
0100 1000 1111 1010 1011 1100 1001 0111
0x  4     8    F   A  B  C  9  7
```

In octal, each digit represents 3 bits

```
01 001 000 111 110 101 011 110 010 010 111
0  1    1   0  7  6  5  3  2  2  7
```

In binary, each digit represents 1 bit

```
0b01001000111110101011110010010111
```

Binary to Hexadecimal

- Example: Convert 1011111000101001₂ to Hex:
  - 1011111000101001₂ = 0x7D9₁

- Example: Convert 10111101011100₂ to Hex:
- Reverse the previous process ...
- Convert each hex digit into equivalent 4-bit binary representation
- Example: Convert $AD_{16}$ to Binary:

Representing Negative Integers

- modern computers almost always use two's complement to represent integers
- positive integers and zero represented in obvious way
- negative integers represented in clever way to make arithmetic in silicon fast/simpler
- for an $n$-bit binary number the representation of $-b$ is $2^n - b$
- e.g. in 8-bit two’s complement $-5$ is represented as $2^8 - 5 = 11111011_2$

Code example: printing all 8 bit twos complement bit patterns

- Some simple code to examine all 8 bit twos complement bit patterns.

```c
for (int i = -128; i < 128; i++) {
    printf("%4d ", i);
    print_bits(i, 8);
    printf("\n");
}
```

$ dcc 8_bit_twos_complement.c print_bits.c -o 8_bit_twos_complement$
Code example: printing all 8 bit twos complement bit patterns

```bash
$ ./8_bit_twos_complement
-128 10000000
-127 10000001
-126 10000010
...
-3 11111101
-2 11111110
-1 11111111
0 00000000
1 00000001
2 00000010
3 00000011
...
125 01111101
126 01111110
127 01111111
```

Code example: printing bits of int

```c
int a = 0;
printf("Enter an int: ");
scanf("%d", &a);
// sizeof returns number of bytes, a byte has 8 bits
int n_bits = 8 * sizeof a;
print_bits(a, n_bits);
printf("\n");
```

```bash
$ dcc print_bits_of_int.c print_bits.c -o print_bits_of_int
$ ./print_bits_of_int
Enter an int: 42
00000000000000000000000000101010
$ ./print_bits_of_int
Enter an int: -42
11111111111111111111111111010110
```

Code example: printing bits of int

```bash
$ ./print_bits_of_int
Enter an int: 0
00000000000000000000000000000000
$ ./print_bits_of_int
Enter an int: 1
00000000000000000000000000000001
$ ./print_bits_of_int
Enter an int: -1
11111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: 2147483647
01111111111111111111111111111111
$ ./print_bits_of_int
Enter an int: -2147483648
10000000000000000000000000000000
$ ```
Many hardware operations work with bytes: 1 byte == 8 bits.

C's `sizeof` gives you the number of bytes used for a variable or type.

- `sizeof variable` returns the number of bytes to store `variable`.
- `sizeof (type)` returns the number of bytes to store `type`.

On CSE servers, C types have these sizes:

- `char` = 1 byte = 8 bits, 42 is `00101010`
- `short` = 2 bytes = 16 bits, 42 is `0000000000101010`
- `int` = 4 bytes = 32 bits, 42 is `00000000000000000000000000101010`
- `double` = 8 bytes = 64 bits, 42 = ?

- Above are common sizes but not universal on a small embedded CPU. `sizeof (int)` might be 2 (bytes).

---

### Code example: `integer_types.c` - exploring integer types

We can use `sizeof` and `limits.h` to explore the range of values which can be represented by standard C integer types on our machine...

```
$ dcc integer_types.c -o integer_types
$ ./integer_types
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>signed char</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int</td>
<td>-2147483648</td>
<td>2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>long</td>
<td>-922337203685477588</td>
<td>922337203685477587</td>
</tr>
<tr>
<td>unsigned long</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>long long</td>
<td>-922337203685477588</td>
<td>922337203685477587</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>

Source code for `integer_types.c`
#include <stdint.h>

- to get below integer types (and more) with guaranteed sizes
- we will use these heavily in COMP1521

```c
// range of values for type
// minimum          maximum
int8_t  i1;  // -128          127
uint8_t i2;  //  0           255
int16_t i3;  // -32768       32767
uint16_t i4; //  0            65535
int32_t i5;  // -2147483648  2147483647
uint32_t i6; //  0            4294967295
int64_t i7;  // -9223372036854775808 9223372036854775807
uint64_t i8; //  0          18446744073709551615
```

Source code for `stdint.c`

### Code example: char_bug.c

Common C bug:

```c
char c;  // c should be declared int (int16_t would work, int is better)
while ((c = getchar()) != EOF) {
    putchar(c);
}
```

Typically `stdio.h` contains:

```c
#define EOF -1
```

- most platforms: `char` is signed (-128..127)
  - loop will incorrectly exit for a byte containing 0xFF
- rare platforms: `char` is unsigned (0..255)
  - loop will never exit