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:

<table>
<thead>
<tr>
<th></th>
<th>1000</th>
<th>100</th>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>10^3</td>
<td>10^2</td>
<td>10^1</td>
<td>10^0</td>
</tr>
</tbody>
</table>

- Write number as $4705_{10}$
  - Note use of subscript to denote base
• base 10 is an arbitrary choice

• can use any base

• e.g. could use base 7

• Place values:

\[
\begin{array}{cccc}
\ldots & 343 & 49 & 7 & 1 \\
\ldots & 7^3 & 7^2 & 7^1 & 7^0 \\
\end{array}
\]

• Write number as \(1216_7\) and interpret as:

\[
1 \times 7^3 + 2 \times 7^2 + 1 \times 7^1 + 6 \times 7^0 = 454_{10}
\]
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:

\[
\begin{array}{cccc}
\cdots & 8 & 4 & 2 & 1 \\
\cdots & 2^3 & 2^2 & 2^1 & 2^0 \\
\end{array}
\]

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

- Write number as $3AF1_{16}$ 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
- Binary constants were only recently added to C - some C compilers will not recognize them

```c
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 6 2 2 7

In binary, each digit represents 1 bit

0b0100100011111010101110010010111
• Example: Convert $101111000101001_2$ to Hex:

• Example: Convert $10111101011100_2$ to Hex:
Hexadecimal to Binary

- Reverse the previous process ...
- Convert each hex digit into equivalent 4-bit binary representation
- Example: Convert $AD5_{16}$ 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 $2^8 - 5 = 1111011_2$
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");
}
```

```sh
$ 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");
```

source code for `print_bits_of_int.c`

```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
1111111111111111111111111010110
```
Code example: printing bits of int

$ ./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 number of bytes used for variable or type

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

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>signed char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>
```

[https://www.cse.unsw.edu.au/~cs1521/23T3/](https://www.cse.unsw.edu.au/~cs1521/23T3/)
<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>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>unsigned long</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>long long</td>
<td>-9223372036854775808</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>0</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>
#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

https://www.cse.unsw.edu.au/~cs1521/23T3/
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
The bytes of a multi-byte (2 byte, 4 byte, ...) quantity can be stored in various orders.

* **Endian-ness** is the order.
* Two common orders: big-endian & little-endian
  * **big-endian** - most significant byte at the smallest memory address.
  * **little-endian** - least significant byte at the smallest memory address.
* Most modern general-purpose computers little-endian
* Endian-ness configurable on some architectures e.g ARM
Testing Endian-ness

C

```c
uint8_t b;
uint32_t u;

u = 0x03040506;
// load first byte of u
b = *(uint8_t *)&u;
// prints 6 if little-endian
// and 3 if big-endian
printf("%d\n", b);
```

MIPS

```mips
li $t0, 0x03040506
la $t1, u
sw $t0, 0($t1) # u = 0x03040506;
lb $a0, 0($t1) # b = *(uint8_t *)&u;
li $v0, 1 # printf("%d", a0);
syscall
li $a0, '\n' # printf("%c", '\n');
li $v0, 11
syscall
li $v0, 0 # return 0
jr $ra
```

.u:

.data

```
source code for endian.c
```

```
source code for endian.s
```