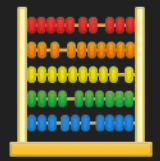
Week 5 

Numeric Types

We’ve seen some different numeric types before:

- **char**
  - 1 byte => 2^8 = 256 different values
- **short**
  - 2 bytes
- **int**
  - Usually 4 bytes
  - Only guaranteed to be at least 2 bytes
- **long**
  - Usually 8 bytes
  - Only guaranteed to be at least 4 bytes

Numeric Types

We also have these:

- When should we use them?
- Why do they have different ranges?
- What does signed / unsigned mean?

```
#include <stdint.h>

                // 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
```

Bases!

2. How can you tell if an integer constant in a C program is decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2)?

Bases!

2. How can you tell if an integer constant in a C program is decimal (base 10), hexadecimal (base 16), octal (base 8) or binary (base 2)?

```
int dec = 116;
int hex = 0x74;
int oct = 0164;
int bin = 0b11101;
```

Bases!

- How does hexadecimal work (base 16)

DEC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

- Working with bases: [Bases Spreadsheet]

- Let’s write a program to do this for us

Bitwise Operations

3. Assume that we have the following 16-bit variables defined and initialised:

```
uint16_t a = 0x5555, b = 0xAAAA, c = 0x0001;
```

What are the values of the following expressions:

a. `a | b` (bitwise OR)
b. `a & b` (bitwise AND)
c. `a ^ b` (bitwise XOR)
d. `a & ~b` (bitwise AND)
e. `c << 6` (left shift)
f. `a >> 4` (right shift)
g. `a & (b << 1)`
h. `b | c`
i. `a & ~c`

Bitwise Operations

3. Assume that we have the following 16-bit variables defined and initialised:

```
uint16_t a = 0x5555, b = 0xAAAA, c = 0x0001;
```

What are the values of the following expressions:

a. `a | b` (bitwise OR)
b. `a & b` (bitwise AND)
c. `a ^ b` (bitwise XOR)
d. `a & ~b` (bitwise AND)
e. `c << 6` (left shift)
f. `a >> 4` (right shift)
g. `a & (b << 1)`
h. `b | c`
i. `a & ~c`

Answers:

a. `a | b == 0xFFFF`
b. `a & b == 0x0000`
c. `a ^ b == 0xFFFF`
d. `a & ~b == 0x5555`
e. `c << 6 == 0x0040`
f. `a >> 4 == 0x0555`
g. `a & (b << 1) == 0x5554`
h. `b | c == 0xAAAB`
i. `a & ~c == 0x5554`

Using bit operations

7. Given the following type definition

```
typedef unsigned int Word;
```

Write a function

```
Word reverseBits(Word w);
```

... which reverses the order of the bits in the variable `w`.

For example: If `w == 0x01234567`, the underlying bit string looks like:

```
0000 0001 0010 0011 0100 0101 0110 0111
```

which, when reversed, looks like:

```
1110 0110 1010 0010 1100 0100 1000 0000
```

which is `0xE6A2C480` in hexadecimal.

Lab Notes

- **sixteen_in.c**

```c
//
// given a string of binary digits ('1' and '0')
// return the corresponding signed 16 bit integer
//
int16_t sixteen_in(char *bits) {

    // PUT YOUR CODE HERE

    return 0;
}
```

- **sixteen_out.c**

```c
// given a signed 16 bit integer
// return a null-terminated string of 16 binary digits ('1' and '0')
// storage for string is allocated using malloc
char *sixteen_out(int16_t value) {

    // PUT YOUR CODE HERE

}
```