Recap of yesterday's lecture

Why do we care about data representation?

- Information = Data + Representation
  - Without the data, there's obviously no information at all
  - But without knowing the exact representation, who knows what the data could even mean?

Data ambiguity example

What does `0b10100011` mean?
- Does it mean $-93$? (signed 1-byte integer)
- Does it mean $163$? (unsigned 1-byte integer)
- Does it mean something else?

What does `0b01110011_01110010_01101001_00000000` mean?
- Does it mean $1936877824$? ([un]signed 4-byte int)
- Does it mean $-1.9205 \times 10^{31}$? (IEEE-754 single-precision floating point)
- ... or could it mean "sri"? (null-terminated ascii string)
Consider file permissions in the Unix file system.

Each file has three sets of “flags” defining its permissions:

```bash
$ ls -l foo.c
-rwxrw-r-- 1 sri group 486 4 May 12:34 foo.c
```

In this example:

- `rwx` gives permissions for the owner of the file
- `rw-` gives permissions for group members
- `r--` gives permissions for everyone else

How can we represent this information efficiently?

We could use:

```c
// 10 * 1 byte = 10 bytes
c char permissions[10] = "rwxrw-r--";
```

Or possibly:

```c
// 9 * 4 bytes = 36 bytes
int permissions[9] = {1, 1, 1, 1, 1, 0, 1, 0, 0};
```

Stop and think - can we make a more efficient representation?

Since each permission is only a true or false boolean value, we can take advantage of this and use only a single bit for each permission.

This allows us to represent the entire data in just 2 bytes!

```c
//
unsigned short permissions = 0b111110100;
```

This is much more efficient, but how are we able to work with individual bits in C?
Sometimes we want to work with individual bits inside a larger value.

Fortunately, everything in C really is just 1's and 0's under the hood!

- eg. the number 42 is 0b00101010
- eg. the ascii character ' #' is 0b00100001
- eg. the floating point 3.14 is 0b01000000000100100011

C provides special operators to read/write individual bits:

- & = bitwise AND
- | = bitwise OR
- ~ = bitwise NOT
- ^ = bitwise XOR
- << = left shift
- >> = right shift

**Bitwise AND:** &

- Takes two values (1, 2, 4, 8 bytes), treats as sequence of bits
- Performs logical AND on each corresponding pair of bits
- Result contains same number of bits as inputs

Example:

```
00100111 AND 11100011
```

```
& 11100011 ----|------
-------- 0 | 0 0
00100011 1 | 0 1
```

Used for e.g. checking whether a bit is set

**Checking for Odd Numbers**

The obvious way to check for odd numbers in C

```c
int isOdd(int n) {
    return n % 2 == 1;
}
```

We can use & to achieve the same thing:

```c
int isOdd(int n) {
    return n & 1;
}
```
Bitwise OR: 

The | operator
- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical OR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
00100111 OR | 0 1
| 11100011 ----|------
-------- 0 | 0 1
11100111 1 | 1 1
```

Used for e.g. ensuring that a bit is set

Bitwise NEG: ~

The ~ operator
- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- performs logical negation of each bit
- result contains same number of bits as input

Example:

```
~ 00100111 NEG | 0 1
-------- ----|------
11011000 | 1 0
```

Used for e.g. creating useful bit patterns

Bitwise XOR: ^

The ^ operator
- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical XOR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
00100111 XOR | 0 1
^ 11100011 ----|-----
-------- 0 | 0 1
11000100 1 | 1 0
```

Used in e.g. generating hashes, graphic operation, cryptography
Left Shift: «

The « operator

- takes a single value (1, 2, 4, 8 bytes), treats as sequence of bits
- also takes a small positive integer \( x \)
- moves (shifts) each bit \( x \) positions to the left
- left-end bit vanishes; right-end bit replaced by zero
- result contains same number of bits as input

Example:

\[
\begin{array}{c|c}
00100111 & \ll 2 \\
---------- & ---------- \\
10011100 & 00000000 \\
\end{array}
\]

Right Shift: »

The » operator

- takes a single value (1, 2, 4, 8 bytes), treats as sequence of bits
- also takes a small positive integer \( x \)
- moves (shifts) each bit \( x \) positions to the right
- right-end bit vanishes; left-end bit replaced by zero (*)
- result contains same number of bits as input

Example:

\[
\begin{array}{c|c}
00100111 & \gg 2 \\
---------- & ---------- \\
00001001 & 00000000 \\
\end{array}
\]

- shifts involving negative values are not portable (implementation defined)
- common source of bugs in COMP1521 and elsewhere
- always use unsigned values/variables to be safe/portable.

bitwise.c: showing results of bitwise operation

$ dcc bitwise.c print_bits.c -o bitwise
$ ./bitwise
Enter a: 23032
Enter b: 12345
Enter c: 3
a = 0101100111111000 = 0x59f8 = 23032
b = 0011000000111001 = 0x3039 = 12345
~a = 1010011000000111 = 0xa607 = 42503
a & b = 0001000000111000 = 0x1038 = 4152
a | b = 01111001111111001 = 0x79f9 = 31225
a ^ b = 0110100111000001 = 0x69c1 = 27073
a >> c = 0001011100000001 = 0x8b3f = 2879
a << c = 1100111111000000 = 0xcfc0 = 53184

source code for bitwise.c
source code for print_bits.c source code for print_bits.h
```c
uint16_t a = 0;
printf("Enter a: ");
scanf("%hd", &a);
uint16_t b = 0;
printf("Enter b: ");
scanf("%hd", &b);
printf("Enter c: ");
int c = 0;
scanf("%d", &c);
print_bits_hex(" a = ", a);
print_bits_hex(" b = ", b);
print_bits_hex(" ~a = ", ~a);
print_bits_hex(" a & b = ", a & b);
print_bits_hex(" a | b = ", a | b);
print_bits_hex(" a ^ b = ", a ^ b);
print_bits_hex("a >> c = ", a >> c);
print_bits_hex("a << c = ", a << c);
```

```c
int n = strtol(argv[1], NULL, 0);
uint32_t power_of_two;
int n_bits = 8 * sizeof(power_of_two);
if (n >= n_bits) {
    fprintf(stderr, "n is too large\n");
    return 1;
}
power_of_two = 1;
power_of_two = power_of_two << n;
printf("2 to the power of %d is %u\n", n, power_of_two);
printf("In binary it is: ");
print_bits(power_of_two, n_bits);
printf("\n");
```
The bottom 3 bits of 7 are ones:
00000000000000000000000000000111

The bottom 19 bits of 524287 are ones:
00000000000111111111111111111111

The bottom 29 bits of 536870911 are ones:
00011111111111111111111111111111

int n = strtol(argv[1], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof mask;
assert(n >= 0 && n < n_bits);
mask = 1;
mask = mask << n;
mask = mask - 1;
printf("The bottom %d bits of %u are ones:\n", n, mask);
print_bits(mask, n_bits);
printf("\n");

The bottom 0 to 7 of 255 are ones:
00000000000000000000000011111111

The bottom 8 to 15 of 65280 are ones:
00000000000001111111111000000000

The bottom 8 to 23 of 16776960 are ones:
00000001111111111111100000000000

The bottom 1 to 30 of 2147483646 are ones:
01111111111111111111111111111110

$ dcc set_low_bits.c print_bits.c -o n_ones
$ ./set_low_bits 3

$ dcc set_bit_range.c print_bits.c -o set_bit_range
$ ./set_bit_range 0 7

Bits 0 to 7 of 255 are ones:
00000000000000000000000011111111

Bits 8 to 15 of 65280 are ones:
00000000000001111111111000000000

Bits 8 to 23 of 16776960 are ones:
00000001111111111111100000000000

Bits 1 to 30 of 2147483646 are ones:
01111111111111111111111111111110

source code for set_low_bits.c
```c
int low_bit = strtol(argv[1], NULL, 0);
int high_bit = strtol(argv[2], NULL, 0);
uint32_t mask;
int n_bits = 8 * sizeof mask;

int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
printf("Bits %d to %d of %u are ones:\n", low_bit, high_bit, mask);
print_bits(mask, n_bits);
printf("\n");
```

---

```c
$ dcc extract_bit_range.c print_bits.c -o extract_bit_range
$ ./extract_bit_range 4 7 42
Value 42 in binary is:
000000000000000000000000101010
Bits 4 to 7 of 42 are:
0010
$ ./extract_bit_range 10 20 123456789
Value 123456789 in binary is:
00000111010110111100110100010101
Bits 10 to 20 of 123456789 are:
11011110011
```

---

```c
int mask_size = high_bit - low_bit + 1;
mask = 1;
mask = mask << mask_size;
mask = mask - 1;
mask = mask << low_bit;
// get a value with the bits outside the range low_bit..high_bit set to zero
uint32_t extracted_bits = value & mask;
// right shift the extracted_bits so low_bit becomes bit 0
extracted_bits = extracted_bits >> low_bit;
printf("Value %u in binary is:\n", value);
print_bits(value, n_bits);
printf("\n");
printf("Bits %d to %d of %u are:\n", low_bit, high_bit, value);
print_bits(extracted_bits, mask_size);
printf("\n");
```

---
void print_bits(uint64_t value, int how_many_bits) {
    // print bits from most significant to least significant
    for (int i = how_many_bits - 1; i >= 0; i--) {
        int bit = get_nth_bit(value, i);
        printf("%d", bit);
    }
}

int get_nth_bit(uint64_t value, int n) {
    // shift the bit right n bits
    // this leaves the n-th bit as the least significant bit
    uint64_t shifted_value = value >> n;
    // zero all bits except the the least significant bit
    int bit = shifted_value & 1;
    return bit;
}

int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    printf("%u = 0x", a);
    print_hex(a);
    printf("\n");
    return 0;
}
print_int_in_hex.c: print_hex - extracting digit

```c
// sizeof returns number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);
// print hex digits from most significant to least significant
for (int which_digit = n_hex_digits - 1; which_digit >= 0; which_digit--) {
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit;
    uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bites
    int hex_digit = shifted_value & 0xF;
    // hex digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values (+ a '\0'
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    putchar(hex_digit_ascii);
}
```

source code for print_int_in_hex.c

https://www.cse.unsw.edu.au/~cs1521/21T2/ COMP1521 21T2 — Bitwise Operators 28 / 49

int_to_hex_string.c: convert int to a string of hex digits

- Write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function snprintf to do this.

```bash
$ dcc int_to_hex_string.c -o int_to_hex_string
$ ./int_to_hex_string
$ ./int_to_hex_string
Enter a positive int: 42
42 = 0x0000002A
$ ./int_to_hex_string
Enter a positive int: 65535
65535 = 0x0000FFFF
$ ./int_to_hex_string
Enter a positive int: 3735928559
3735928559 = 0xDEADBEEF
$ ...
```

source code for int_to_hex_string.c

https://www.cse.unsw.edu.au/~cs1521/21T2/ COMP1521 21T2 — Bitwise Operators 29 / 49

int_to_hex_string.c: main

```c
int main(void) {
    uint32_t a = 0;
    printf("Enter a positive int: ");
    scanf("%u", &a);
    char *hex_string = int_to_hex_string(a);
    // print the returned string
    printf("%u = 0x%s\n", a, hex_string);
    free(hex_string);
    return 0;
}
```

source code for int_to_hex_string.c

https://www.cse.unsw.edu.au/~cs1521/21T2/ COMP1521 21T2 — Bitwise Operators 30 / 49
int_to_hex_string.c: convert int to a string of hex digits

```c
// sizeof returns number of bytes in n's representation
// each byte is 2 hexadecimal digits
int n_hex_digits = 2 * (sizeof n);
// allocate memory to hold the hex digits + a terminating 0
char *string = malloc(n_hex_digits + 1);
// print hex digits from most significant to least significant
for (int which_digit = 0; which_digit < n_hex_digits; which_digit++) {
    // shift value across so hex digit we want
    // is in bottom 4 bits
    int bit_shift = 4 * which_digit;
    uint32_t shifted_value = n >> bit_shift;
    // mask off (zero) all bits but the bottom 4 bites
    int hex_digit = shifted_value & 0xF;
    // hex digit will be a value 0..15
    // obtain the corresponding ASCII value
    // "0123456789ABCDEF" is a char array
    // containing the appropriate ASCII values
    int hex_digit_ascii = "0123456789ABCDEF"[hex_digit];
    string[which_digit] = hex_digit_ascii;
}
// 0 terminate the array
string[n_hex_digits] = 0;
return string;
```

hex_string_to_int.c: convert hex digit string to int

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <hexadecimal-number>
", argv[0]);
        return 1;
    } char *hex_string = argv[1];
    uint32_t u = hex_string_to_int(hex_string);
    printf("%s hexadecimal is %u base 10\n", hex_string, u);
    return 0;
}
```

hex_string_to_int.c: main

As an exercise write C to convert an integer to a string containing its hexadecimal digits.

Could use the C library function `strtol` to do this.

```bash
$ dcc hex_string_to_int.c -o hex_string_to_int
$ ./hex_string_to_int 2A 2A hexadecimal is 42 base 10
$ ./hex_string_to_int FFFF FFFF hexadecimal is 65535 base 10
$ ./hex_string_to_int DEADBEEF DEADBEEF hexadecimal is 3735928559 base 10
```

hex_string_to_int.c: main
```c
uint32_t hex_string_to_int(char *hex_string) {
    uint32_t value = 0;
    for (int which_digit = 0; hex_string[which_digit] != 0; which_digit++) {
        int ascii_hex_digit = hex_string[which_digit];
        int digit_as_int = hex_digit_to_int(ascii_hex_digit);
        value = value << 4;
        value = value | digit_as_int;
    }
    return value;
}
```

```c
int hex_digit_to_int(int ascii_digit) {
if (ascii_digit >= '0' && ascii_digit <= '9') {
    // the ASCII characters '0' .. '9' are contiguous
    // in other words they have consecutive values
    // so subtract the ASCII value for '0' yields the corresponding integer
    return ascii_digit - '0';
}
if (ascii_digit >= 'A' && ascii_digit <= 'F') {
    // for characters 'A' .. 'F' obtain the
    // corresponding integer for a hexadecimal digit
    return 10 + (ascii_digit - 'A');
}
    fprintf(stderr, "Bad digit \%c\n", ascii_digit);
    exit(1);
}
```

```c
// int16_t is a signed type (-32768..32767)
// below operations are undefined for a signed type
int16_t i;
    i = -1;
    i = i >> 1; // undefined - shift of a negative value
    printf("%d\n", i);
    i = -1;
    i = i << 1; // undefined - shift of a negative value
    printf("%d\n", i);
    i = 32767;
    i = i << 1; // undefined - left shift produces a negative value
uint64_t j;
    j = 1 << 33; // undefined - constant 1 is an int
    j = ((int64_t)1) << 33; // ok
```

```c
int xor_value = strtol(argv[1], NULL, 0);
if (xor_value < 0 || xor_value > 255) {
    fprintf(stderr, "Usage: %s <xor-value>\n", argv[0]);
    return 1;
}
int c;
while ((c = getchar()) != EOF) {
    // exclusive-or
    // ^ | 0 1
    // ----|-----
    // 0 | 0 1
    // 1 | 1 0
    int xor_c = c ^ xor_value;
    putchar(xor_c);
}
```

Source code for xor.c

```
$ echo Hello Andrew|xor 42
bOFFE
kDNXO$ $ echo Hello Andrew|xor 42|cat -A
bOFFE$
kDNXO$
$ echo Hello |xor 42
bOFFE $ echo -n 'bOFFE '|xor 42
Hello
$ echo Hello|xor 123|xor 123
Hello
$ 
```

---

### pokemon.c: using an int to represent a set of values

```c
#define FIRE_TYPE 0x0001
#define FIGHTING_TYPE 0x0002
#define WATER_TYPE 0x0004
#define FLYING_TYPE 0x0008
#define POISON_TYPE 0x0010
#define ELECTRIC_TYPE 0x0020
#define GROUND_TYPE 0x0040
#define PSYCHIC_TYPE 0x0080
#define ROCK_TYPE 0x0100
#define ICE_TYPE 0x0200
#define BUG_TYPE 0x0400
#define DRAGON_TYPE 0x0800
#define GHOST_TYPE 0x1000
#define DARK_TYPE 0x2000
#define STEEL_TYPE 0x4000
#define FAIRY_TYPE 0x8000
```

Source code for pokemon.c
**pokemon.c: using an int to represent a set of values**

- Simple example of a single integer specifying a set of values
- Interacting with hardware often involves this sort of code

```c
uint16_t our_pokemon = BUG_TYPE | POISON_TYPE | FAIRY_TYPE;
```

// Example code to check if apokemon is of a type:
```c
if (our_pokemon & POISON_TYPE) {
    printf("Poisonous\n"); // prints
}
if (our_pokemon & GHOST_TYPE) {
    printf("Scary\n"); // does not print
}
```

**bitset.c: using an int to represent a set of values**

```c
$ dcc bitset.c print_bits.c -o bitset
$ ./bitset
Set members can be 0-63, negative number to finish
Enter set a: 1 2 4 8 16 32 -1
Enter set b: 5 4 3 33 -1
a = 000000000000000000000000000000010000000000000010000000100010110 = 0x100010116 = 4295033110
b = 000000000000000000000000000000100000000000000000000000000011100 = 0x200000038 = 8589934648
a = \{1,2,4,8,16,32\}
b = \{3,4,5,33\}
a union b = \{1,2,3,4,5,8,16,32,33\}
a intersection b = \{4\}
cardinality(a) = 6
is_member(42, a) = 0
```
printf("Set members can be 0-%d, negative number to finish\n", MAX_SET_MEMBER);
set a = set_read("Enter set a: ");
set b = set_read("Enter set b: ");
print_bits_hex("a = ", a);
print_bits_hex("b = ", b);
set_print("a = ", a);
set_print("b = ", b);
set_print("a union b = ", set_union(a, b));
set_print("a intersection b = ", set_intersection(a, b));
printf("cardinality(a) = %d\n", set_cardinality(a));
printf("is_member(42, a) = %d\n", (int)set_member(42, a));
Exercise: Bitwise Operations

Given the following variable declarations:

```c
// a signed 8-bit value
unsigned char x = 0x55;
unsigned char y = 0xAA;
```

What is the value of each of the following expressions:

- `(x & y)`  
- `(x ^ y)`  
- `(x << 1)`  
- `(y << 1)`  
- `(x >> 1)`  
- `(y >> 1)`
Exercise: Bit-manipulation

Assuming 8-bit quantities and writing answers as 8-bit bit-strings:

What are the values of the following:

- 25, 65, ~0, ~1, 0xFF, ~0xFF
- (01010101 & 10101010), (01010101 | 10101010)
- (x & ~x), (x | ~x)

How can we achieve each of the following:

- ensure that the 3rd bit from the RHS is set to 1
- ensure that the 3rd bit from the RHS is set to 0