
MONTANA
STATE UNIVERSITY

# Bit Manipulation In C

Working with Bitmasks

# Binary Data

- Today we are going to review how to read and manipulate binary data
- In particular we are going to look at unsigned 64 bit integers and how to use bitmasks to store non-integer representations in them
- Does this sound like it might be useful to you?
    - Of course it does :)

# Binary Data

- Why use binary representations of data?
  - Space efficient
  - CPU efficient
  - It is low level hacking that is good for you, like kale

# Binary Data - Storage

- Recall that we are going to use a few unsigned 64 bit integers to store board state for our game
- Declaration & literal form:

```
// variable declaration
unsigned long long my_unsigned_int_64 = 0;

// literal
my_unsigned_int_64 = 1ull;
```

# Binary Data - Storage

- Why are we using a 64 bit integer?
- Why are we using an *unsigned* 64 bit integer?

```
// variable declaration
unsigned long long my_unsigned_int_64 = 0;

// literal
my_unsigned_int_64 = 1ull;
```

# Binary Data - Storage

- Why are we using a 64 bit integer?
- Why are we using an *unsigned* 64 bit integer?

# Binary Data - Manipulation

- Binary manipulation (bit masks) are based on simple boolean logic
- Boolean - true or false
- Binary Equivalent - 1 or 0
- Boolean Logic Operators:
  - NOT
  - AND
  - OR
  - XOR

# Boolean Logic - NOT

- Unary operator
- Inverts a single value
  - true → false
  - false → true
  - 0 → 1
  - 1 → 0

| A | !A |
|---|---|
| 1 | 0 |
| 0 | 1 |

# Boolean Logic - AND

- Binary Operator
  - If both values are true, true
  - Else false

| A | B | A&B |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 0 |

# Boolean Logic - OR

- Binary Operator
  - If either value is true, true
  - Else false

| A | B | A\|B |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |

# Boolean Logic - XOR

- Binary Operator
  - If either value is true and the other is not true, true
  - Else false

- *XOR is a very important logical operator, we will discuss it more when we discuss CPU design*

| A | B | A^B |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 0 | 0 |

# Boolean Logic - C Equivalents

- C has *bitwise* equivalents of these operators
- Allow you to apply boolean logic, treating each bit of two numbers as pairwise booleans
- Consider two for bit numbers:

10 decimal → binary 1010
5 decimal → binary 0101

# Boolean Logic - C Equivalents

- Consider two for bit numbers:

  10 decimal → binary 1010
  5 decimal → binary 0101

- The NOT operator (~)

  10 decimal → binary 1010 → ~ → 0101 → 5 in decimal!

# Boolean Logic - C Equivalents

- Consider two for bit numbers:

  10 decimal → binary 1010
  5 decimal → binary 0101

- The AND bitwise operator (&)

  binary 1010 & binary 0101 -> binary 0000 -> 0

# Boolean Logic - C Equivalents

- Consider two for bit numbers:

  10 decimal → binary 1010
  5 decimal → binary 0101

- The OR bitwise operator (|)

  binary 1010 | binary 0101 -> binary 1111 -> 15

# Boolean Logic - C Equivalents

- Consider two for bit numbers:

  10 decimal → binary 1010
  5 decimal → binary 0101

- The OR bitwise operator (|)

  binary 1010 | binary 0101 -> binary 1111 -> 15

# Boolean Logic - C Equivalents

- Consider two for bit numbers:

  12 decimal → binary 1100
  5 decimal → binary 0101

- The XOR bitwise operator (^)

  binary 1100 ^ binary 0101 -> binary 1001 -> 9

# Bit Masking

- In the project and in life in general, you will often be interested in a specific bit
- To get the value of a single bit, you will need to create a *bit mask*
- A bit mask is the number 1 that has been *shifted* to the correct bit position

# Bit Masking

- The left bit shift operator in C is <<
- This moves all the bits n slots to the left in the number
- Consider binary 1010 (decimal 10)

1010 << 1 = 0100 (decimal 4)
1010 << 2 = 1000 (decimal 8)
1010 << 3 = 0000 (decimal 0)

# Bit Masking

- Getting an bit mask for the Nth bit

```
unsigned int n = 10;
unsigned int mask = 1u << n;
```

# Bit Masking

- Testing the Nth bit of a value

```c
unsigned int value = 1024;
unsigned int n = 10;
unsigned int mask = 1u << n;
if (mask & value) {
    printf( format: "Yep! That bit was 1\n");
}
```

# Bit Masking

- Setting the Nth bit of a value

```
unsigned int value = 1023;
unsigned int n = 10;
unsigned int mask = 1u << n;
value = value | mask;
```

# Bit Masking

- Clearing the Nth bit of a value

```
unsigned int value = 1025;
unsigned int n = 10;
unsigned int mask = ~(1u << n);
value = value & mask;
```

# Review

- It is possible to work with individual bits in integer values
- This can be used for efficient representation of data
- Operators at the bit level are based on boolean logic
- The core operators are NOT, AND, OR, and XOR
- By using the shift operator, it is possible to construct bit masks that can be used for reading and manipulating individual bits


MONTANA
STATE UNIVERSITY