COP3503

Bitwise Operations

The Basics of Bits and Bytes

- A **bit** (or binary digit) is a basic unit of data in computing.
- A bit can have a value of 0, or 1.
  - Unlike a decimal digit, which has a value of 0-9.
- A **byte** is made of 8 bits (commonly, though other sizes exist).
- Data types are (usually) made of multiple bytes.
- A `char` is 1 byte, or an 8-bit integer (everything is a number, either integral or floating-point value).
- A `short` is typically 2 bytes, or a 16-bit integer.
- An `int` is typically a 4 byte variable, or a 32-bit integer.

Calculating Decimal Values

- In a base 10, or decimal, numbering system, place values are based on powers of 10.
- Decimal value: 8,935

Thousands Place (10^3)	Hundreds Place (10^2)	Tens Place (10^1)	Ones Place (10^0)
8	9	3	5
8 x 1000	9 x 100	3 x 10	5 x 1

- Decimal value: 8,935

8000	+	900	+	30	+	5

- 8000 + 900 + 30 + 5 = 8935

Binary Numbering

- In a base 2, or binary, numbering system, place values are based on powers of 2.
- Binary value: 1011

Eight place (2^3)	Four Place (2^2)	Two Place (2^1)	One Place (2^0)
1	0	1	1
1 x 8	0 x 4	1 x 2	1 x 1

- The sum of the products of each place value and its corresponding digit is the final value.

8	+	0	+	2	+	1

- 8 + 0 + 2 + 1 = 11

Decimal and Binary Combined

10,000,000	1,000,000	100,000	10,000	1,000	100	10	1	Base 10
128	64	32	16	8	4	2	1	Base 2
1	1	1	1	1	1	1	1	Number

- 11111111 in decimal is… 11,111,111 — 10,000,000 + 1,000,000 + 100,000 + 10,000 + 1,000 + 100 + 10 + 1
- 11111111 in binary is… 255 — 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1

Our Code Modifies Bits Indirectly

- Code ultimately turns into bit operations. For example:

```
int x = 5;
```

- A compiler translates this to instructions:
  - Set some bits to the binary equivalent of 5
    - 0101, plus 28 leading zeroes for a 32-bit integer
- Whether you're aware of it or not, this is always happening.
- Most computational processing (i.e., our code) ultimately ends up as:
  - Set some bits to a value.
  - Check if these bits are equal to a value.
  - If so, set some bits, or check some other bits.

A little dull when you think of it this way!

Not nearly as interesting as
"draw this 3D model to the screen"
"open this file and read its contents"
"connect to this remote web server"

We Can Modify Bits Directly

- We typically write “high level” code.
- We can use bitwise operators to work on a lower level.
- This can give us more control over our computer.

Operator	Name
|	OR
&	AND
^	XOR
~	Negation
<<	Left shift
>>	Right shift
Operator	Name

Not to be confused with logical operators:

logical OR: ||
logical AND: &&

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result (ValueOne | ValueTwo)	?	?	?	?

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

At least one of these is “on”, so the final result is on, or 1

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	?	?	?

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

Neither of these bits are on, so the result is a 0, or “off” bit

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	0	?	?

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

At least one of these is on, so the final result is on

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	0	1	?

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

At least one of these is on, so the final result is on

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	0	1	$\boxed{1}$

| OR |

- Compares pairs of bits of two values
- If **EITHER** of the two bits are 1, the final result is 1.
- Used to combine groups of bits together, or turn bits ON

ValueOne (3)	0 0 1 1
ValueTwo (10)	1 0 1 0
Final result(11)	1 0 1 1

8+0+2+1

This process is combining **BITS**, not **VALUES**.

3 + 10 != 11

3 |= 10 == 11

AND &

- If **BOTH** bits are 1, the final result is 1.
- Useful for checking if a particular bit is turned on.

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result (ValueOne & ValueTwo)	?	?	?	?

AND &

- If **BOTH** bits are 1, the final result is 1.
- Useful for checking if a particular bit is turned on.

BOTH of these bits are not on, so the final bit is off

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	0	?	?	?

AND &

- If **BOTH** bits are 1, the final result is 1.
- Useful for checking if a particular bit is turned on.

BOTH of these bits are not on, so the final bit is off

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	0	0	?	?

AND &

- If **BOTH** bits are 1, the final result is 1.
- Useful for checking if a particular bit is turned on.

BOTH of these bits are turned on, so the result is 1

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	0	0	1	?

AND &

- If **BOTH** bits are 1, the final result is 1.
- Useful for checking if a particular bit is turned on.

Only one of these bits is on, so the resulting bit is 0

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	0	0	1	0

AND &

ValueOne (3)	0 0 1 1
ValueTwo (10)	1 0 1 0
Final result(2)	0 0 1 0

$\rightarrow$ 0+0+2+0

This can be used to check if a specific bit is turned on:

```
if ((someValue & 2) == 2)
{
    // 2 bit is on
}
```

someValue could be anything, but using AND with 2 means we ignore all but one specific bit:
`? ? 1 ?`

```
someValue: 1111
2:      0010
(someValue & 2) == 2 // true
```

```
someValue:     1101
2:      0010
(someValue & 2) == 2 // false
```

```
someValue:     1111
10:     1010
(someValue & 10) == 10 // true
```

Exclusive Or, XOR ^

- If **EXACTLY ONE** of the two bits are 1, the final result is 1.
- If both are 0, the result is 0.
- If both are 1, the result is 0.

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result (ValueOne ^ ValueTwo)	?	?	?	?

Exclusive Or, XOR ^

- If **EXACTLY ONE** of the two bits are 1, the final result is 1.
- If both are 0, the result is 0.
- If both are 1, the result is 0.

Only one of these two is on, so the final bit is on.

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	?	?	?

Exclusive Or, XOR ^

- If **EXACTLY ONE** of the two bits are 1, the final result is 1.
- If both are 0, the result is 0.
- If both are 1, the result is 0.

Is EXACTLY ONE of the two bits on? No? The final bit is 0.

ValueOne (3)	0 0 1 1
ValueTwo (10)	1 0 1 0
Final result	1 0 ? ?

Exclusive Or, XOR ^

- If **EXACTLY ONE** of the two bits are 1, the final result is 1.
- If both are 0, the result is 0.
- If both are 1, the result is 0.

Is EXACTLY ONE of the two bits on? No? The final bit is 0.

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	0	0	?

Exclusive Or, XOR ^

- If **EXACTLY ONE** of the two bits are 1, the final result is 1.
- If both are 0, the result is 0.
- If both are 1, the result is 0.

Only one of these bits is on, so the final result is 1.

ValueOne (3)	0	0	1	1
ValueTwo (10)	1	0	1	0
Final result	1	0	0	1

Exclusive Or, XOR ^

```
ValueOne (3)      0 0 1 1
ValueTwo (10)     1 0 1 0
---------------------------
Final result(9)   1 0 0 1
                  8+0+0+1
```

XOR doesn't have as many "common" uses as AND/OR.

There are a lot of "tricks" you can do with it.

For example, you can swap two values without a 3rd variable, by using 3 XOR operations.

```
int a = 5;  // 0101
int b = 10; // 1010
```

```
// This may seem like magic
a = a ^ b; // 1111 (15...?)
b = a ^ b; // 0101 (5!)
a = a ^ b; // 1010 (10!)
```

It's a neat trick, but not easy to read code. Many XOR uses are like this.

NEGATION ~

- An unary operator—it only has one operand

```
int value = 12;
cout << ~value;
int negated32 = ~32;
```

- Inverts all of the bits of a value – 0s become 1, and 1s become 0

Value (12)	1 1 0 0
~Value (3)	0 0 1 1
Zero	0 0 0 0
~Zero (15)	1 1 1 1

Left Shift << and Right Shift >>

- **Bit shifting** moves every bit in a value left or right a number of times specified by the right-hand operand.

```
int value = 1;
value = value << 2;   // Move every bit 2 spaces to the left
cout << (32 >> 3);// Move every bit 3 spaces to the right
```

- Left Shift Examples:

```
value (1)      0 0 0 1 == 1
value <<= 2    0 1 0 0 == 4
value <<= 1    1 0 0 0 == 8
```

Left-shifting is equivalent to multiplying by **2^N**, where **N** is the right-hand operand

<< 3 (multiply by 2^3, or 8)

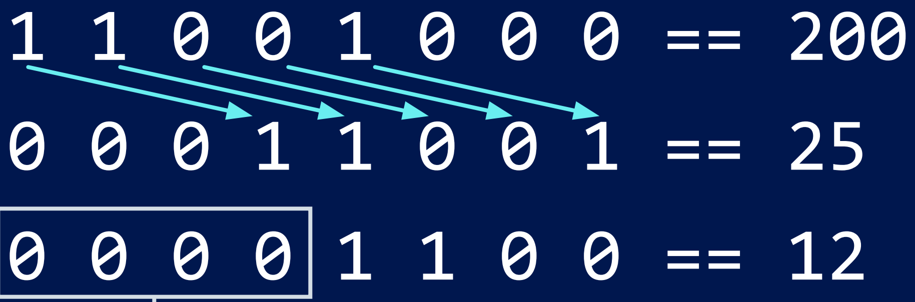
<< 6 (multiply by 2^6, or 64)

Right-shifting: Divide by 2^N

- The same concept as left-shift, in reverse!

```
int value = 200;
value = value >> 3;   // Shift 3 spaces right (divide by 2^3, or 8)
value >>= 1;        // Shift 1 space right (divide by 2)
```

- Left Shift Examples:



```
value (200)    1 1 0 0 1 0 0 0 == 200
value >>= 3    0 0 0 1 1 0 0 1 == 25
value >>= 1    0 0 0 0 1 1 0 0 == 12
```

Mathematically 25/2 == 12.5, but the bits don't lie! (Also, integers can't store floating-point numbers)

"new" bits are set to 0, and bits can "fall off the edge" of a value, and get discarded

Bit Fields

- A **bit field** is, generally, just a collection of bits that are stored together
- Basic data types can be used for this purpose (everything is made of bits!)

We often think of a variable storing ONE value, but what if we think of it as storing multiple?

```
unsigned char
```

1 byte, or 8 bits (a bit field with 8 values)

```
unsigned int
```

4 bytes, or 32 bits (a bit field with 32 values)

Goal: Store 8 Boolean Values

- 1 `bool` variable = 1 byte
- 8 `bool` variable = 8 bytes
- 1 `unsigned char` (made of 8 bits) = 1 byte
- One `unsigned char` as a **bit field** can store up to 8 `bool` values, for 1/8 the memory!

Why not just use a `bool` as a bitfield?

Compilers often treat Boolean data types a bit differently, and they may not work with other operations.

Example: Tracking a Video Game Character's Inventory

- A character can carry the following items:
  1. Flashlight
  2. Pistol
  3. Crowbar
  4. Shotgun
  5. Laser
  6. Machine gun
  7. Chainsaw
  8. Rocket launcher

How could we track what the character is holding?

```
// 8 true/false values
bool items[8];

// Also 8 true/false values
// (0/1 values, same thing... ish)
unsigned char bitItems;
```

Picking Up an Item

1. Flashlight
2. Pistol
3. Crowbar
4. Shotgun
5. Laser
6. Machine gun
7. Chainsaw
8. Rocket launcher

```
bool items[8];
unsigned char bitItems = 0;

items[0] = true;   // Flashlight
bitItems |= 1;     // turn on the

items[1] = true;   // Pistol
bitItems |= 2;     // turn on the 2 bit

items[2] = true;   // Crowbar
bitItems |= 4;     // turn on the 4 bit

items[6] = true;   // Chainsaw
bitItems |= 64;    // turn on the 64 bit

bitItems |= (1 << 6); // an alternative
```

Why the 4 bit? Because binary "places" increase by power of 2 values:

```
8 4 2 1
0 1 1 1
```

The bit for the second item has the equivalent value of 2^2

1 << 6 == 64

Dropping an Item

1. Flashlight
2. Pistol
3. Crowbar
4. Shotgun
5. Laser
6. Machine gun
7. Chainsaw
8. Rocket launcher

```
bool items[8];
items[0] = true; // Flashlight
items[2] = true; // Crowbar

// Flashlight (1) and crowbar (4)
unsigned char bitItems = 5;

// Drop the crowbar
items[2] = false;

// Turn off the 4 bit
bitItems &= ~4;
```

What sorcery is this?!

OR | is the “turn on” operator, but there’s no “turn off” operator.

The combination of & and ~ is how we turn a bit off.

Turning Off Bits Requires More Work

1. Start with the bit(s) you want to turn off (just the 4 bit, in this case, so a value of 4)

```
0 1 0 0
```

2. Negate that value (flip 0s to 1s and 1s to 0s)

```
 4 == 0 1 0 0
~4 == 1 0 1 1
```

Inventory
bitItems = 5
flashlight (1)
+
crowbar (4)
0 1 0 1

3. AND with the value we want to modify (AND requires BOTH bits to be “on”)

```
        bitItems == 0 1 0 1
              ~4 == 1 0 1 1
-----------------------------
bitItems &= ~4 == 0 0 0 1
```

Make this bit 0 in the final value.

The original value, minus a crowbar!

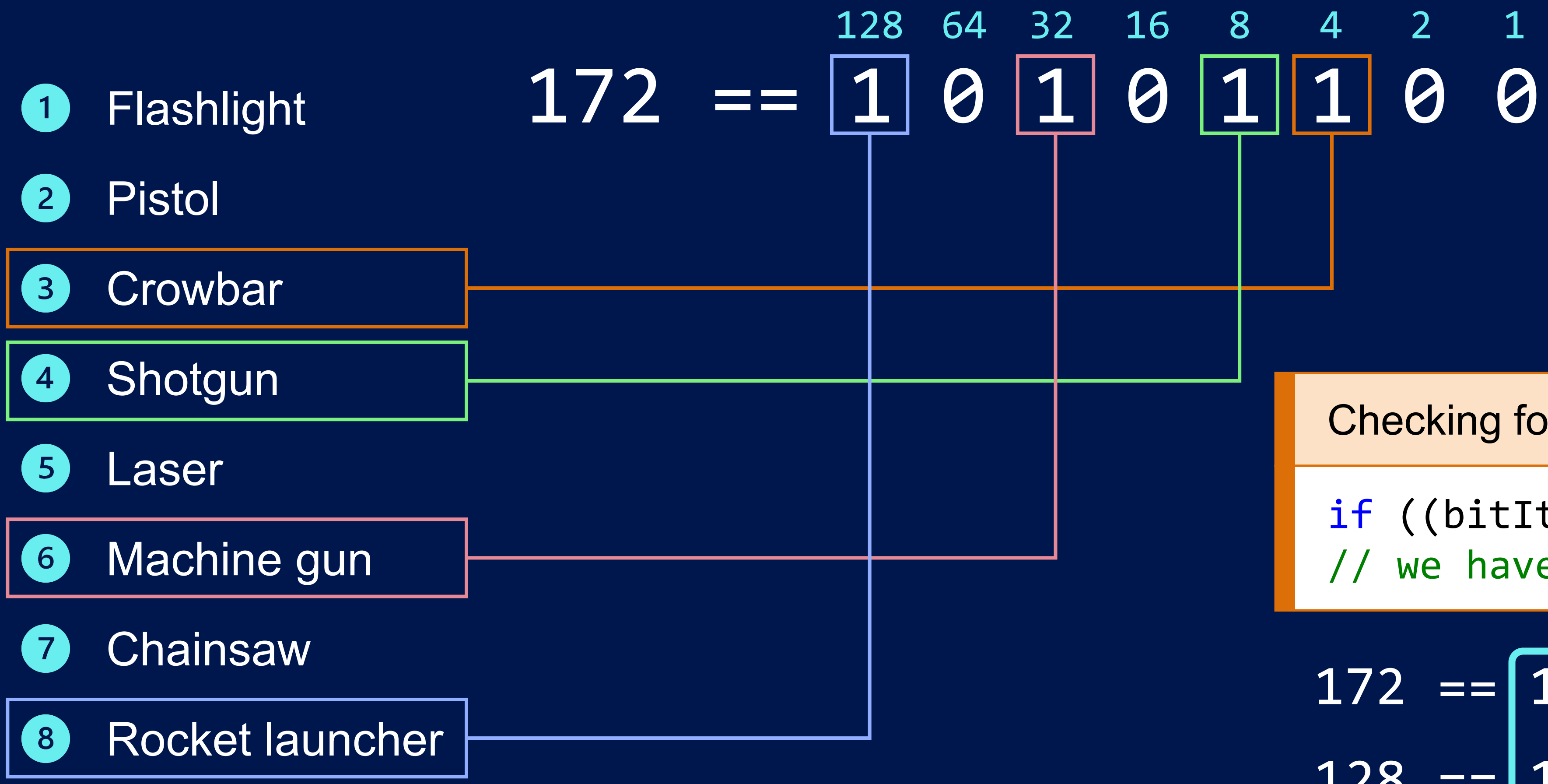
Generating a Random Inventory

```
bool items[8] = { false }; // Start empty
for (int i = 0; i < 8; i++)
{
    int roll = RandomNumber(0, 1); // "Flip a coin"
    if (roll == 1)
        items[i] = true;
}

// Bitfield version, just one random number
unsigned char bitItems = RandomNumber(0, 255);
```

How does this work? Let's say the number generated was **172**.

Dropping an Item



Checking for a specific item

```
if ((bitItems & 128) == 128)
// we have a rocket launcher
```

```
172 == 1 0 1 0 1 1 0 0
128 == 1 0 0 0 0 0 0 0
```

Only one bit is turned on from (172 & 128) – the 128 bit

Bit Fields and Enumerations

- An enumeration (enum) can help with identifying bits
- Instead of bit 4, bit 32, bit 128…

No more wondering what was item 16? Was the Flashlight 1, or 2?

```
enum Items {
    FlashLight = 1,
    Pistol = 2,
    Crowbar = 4,
    Shotgun = 8,
    Laser = 16,
    MachineGun = 32,
    Chainsaw = 64,
    RocketLauncher = 128 };
```

```
// Easy to pick up a laser
bitItems |= Items::Laser;
```

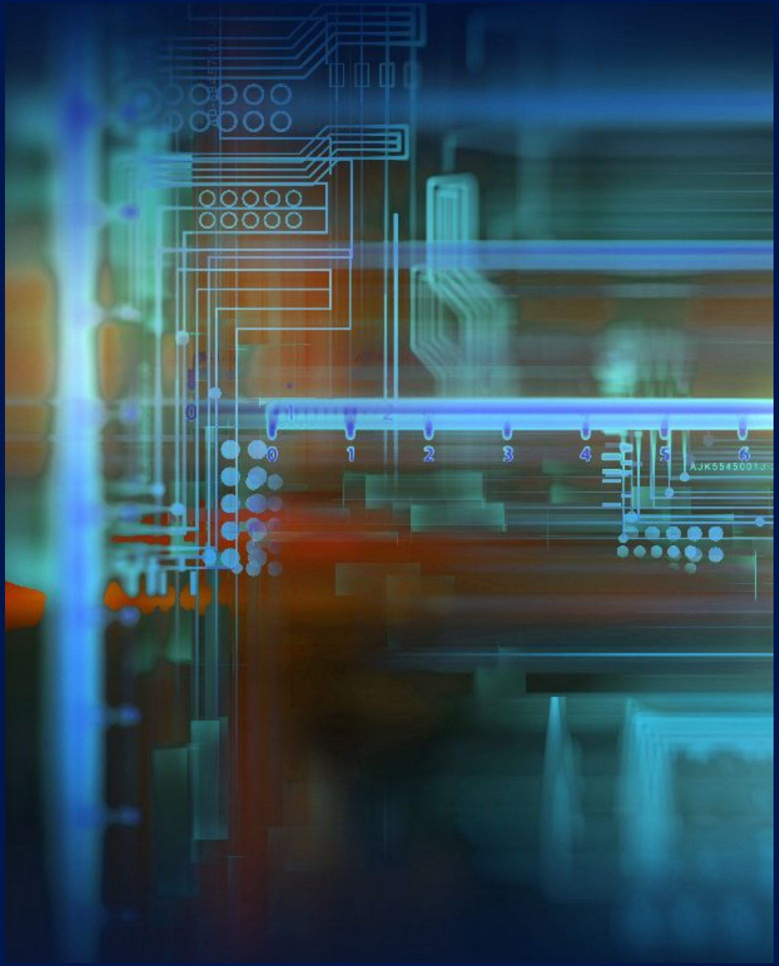
```
// A class provides a clean interface
void BitField::TurnOff(Items item)
{
    _bitItems &= ~item;
}

someBitField.TurnOff(Items::Crowbar);
```

Recap

- Fundamentally, **computers manipulate bits**.
  - Every program, every line of code, is some form of changing bits to 1s and 0s.
- **Bitwise operations** let you manipulate values on the bit level.
- These operations are more “computer language” than programming language.
- **Bit fields** allow us to store more values in a smaller space
  - We can do this for memory efficiency.
  - A single 4-byte integer can hold 32 bit values!
- Bitwise operations can be used in applications like cryptology and security, which go a bit beyond the scope of the course.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.