CSE 220 – C Programming

Bitwise Operators

Integer types

- Whole numbers
- Signed/Unsigned
  - Signed: most significant bit denotes the sign:
    - 1 if –
    - 0 if +
  - By default, integers are signed
- Length (machine dependent):
  - int: 16/32bits
  - long int, long: 32/64bits
  - short int, short: 16bits

long int ≥ int ≥ short int

- sizeof operator: number of bytes:
  - sizeof(char): 1 sizeof(int): 4 sizeof(x): 4

Integer types

31 *30* *0*

- 111111111 00101000 00111000 00000110
  - signed int x: - or + ($2^{30} + 2^{29} +$ ….)
  - unsigned int x: 2^{31} + …
- Integer overflow:
  - 1111111111111111 + 0000000000000001
  - Result does not fit in data type (16 bits)
  - If signed: behavior undefined
  - If unsigned: correct answer modulo 2^n
    - n is the number of bits

Integer Constants

- C allows constants to be written in:
  - Decimal: base 10
    - Digits between 0 and 9, must not begin with 0
    - 34 199
  - Octal: base 8
    - Digits between 0 and 7, must begin with 0
    - 034 07777
  - Hexadecimal: base 16
    - Digits between 0 and 9, letters between a and f (case doesn't matter), must begin with 0x
    - 0xFA 0X2fCB 0xfddd

For the binary number: 111011

What is its Octal representation?

0111011

037

073

I don't know

For the binary number: 111011

What is its Hexadecimal representation?

0x73

0x3b

0x311

I don't know

Conversion Specification

% | - + 0 | m | .p | X

Start of the specifier

Flags

Minimum width of the field

Precision

Conversion type

- Conversion type:
  - c: a single character
  - s: string
  - d: integer
  - f: floating point notation
  - E,e: scientific notation
  - X,x: hexadecimal number
  - o: octal number

Bitwise Operators

- For bit manipulation:
  - Bitwise AND: &
  - Bitwise inclusive OR: |
  - Bitwise exclusive OR: ^
  - Bitwise complement: ~
  - Left shift: <<
  - Right shift: >>

Bitwise Operators

- Binary representation:

```
int i = 22;        /*    10110 */
int j = 91;        /* 1011011 */
```

- Division by 2

  $22 = 2*11 = 2 * (2 * 5 + 1)$

  $= 2 * (2 * (2*2*1 + 1) + 1) = 2^4 + 2^2 + 2^1$

- Comparison by powers of 2 (1, 2, 4, 8, 16, 32, 64, …)

  $22 = 16 + 4 + 2 = 2^4 + 2^2 + 2^1$

  $91 = 64 + 16 + 8 + 2 + 1 = 2^6 + 2^4 + 2^3 + 2^1 + 2^0$

- In binary: $1111 = 10000 – 1 = 2^4 - 1$

What do you think the following code outputs?

```
unsigned int a = 3;
unsigned int b = 2;
printf("%d", a && b);
```

0

1

2

3

Bitwise Operators

In this example:
- variables i and j are of type int
- are represented by 16 bits (2 bytes)

- Bitwise &:

  result = i & j;
  0000000000010110
  0000000001011011
  0000000000010010 ⇔ $2^1 + 2^4 = 18$

- Bitwise ~ (complement):

  result = ~i;
  0000000000010110
  1111111111101001 ⇔
  65,513

What do you think the following code outputs?

```
unsigned int a = 3;
unsigned int b = 2;
printf("%d", a & b);
```

0

1

2

3

Bitwise Operators

- Bitwise exclusive or ^: *1 if bits are different, 0 if the same*

  result = i ^ j;
  0000000000010110
  0000000001011011
  0000000001001101 ⇔ 77

- Bitwise inclusive or |: *1 if at least one of the bits is 1*

  result = i | j;
  0000000000010110
  0000000001011011
  0000000001011111 ⇔ 95

Bitwise Operators

- Left shift:

result = i << 3;
0000000000010110

0000000010110000 ⇔ 176

- Right shift:

result = i >> 2;
0000000000010110

0000000000000101⇔ 5

What is the difference between y and z?

```
unsigned int x = 56;
unsigned int y = x >> 1;
unsigned int z = x / 2;
```

No difference

y is a float

y is negative

I don't know

When To Use Bitwise Operators?

- When space efficiency is paramount.
  - If you need to pack a lot of data into a small space, you want to use each byte of memory available.
- When speed is paramount.
  - Bitwise operators take less time to run (generally) than all the other arithmetic operators.
- When you are forced to.
  - Following certain algorithms (often related to data compression or cryptography) or communicating with certain hardware (often microcontrollers or sensors) may require "unpacking" multiple pieces of information stored in a single int.

stdint.h

- ints, longs, and all the types in C can be different sizes depending on the compiler and the hardware.
- However, the stdint.h library allows you to get a data type that is big enough to hold a particular number of bits (8, 16, 32, or 64).
- We'll mostly be concerning ourselves with unsigned integers, primarily "uint16_t", which is the 16 bit unsigned integer.