# Operators

CS 217



## Arithmetic Operators

- Binary arithmetic operators: `+ - * /`
- Modulus (remainder) operator: `%`
  - `x%y` is the remainder when `x` is divided by `y`
  - well defined only when `x > 0` and `y > 0`
- Unary operators: `- +`
- Precedence: unary higher than binary
  - `-2*a+b` is parsed as `(((-2)*a)+b`
- Associativity: left to right
  - `a+b+c` is parsed as `((a+b)+c)`



## Portability

- Print a number in decimal

```
void putd(int n) {
   if (n < 0) {
      putchar('-');
      n = -n;
   }
   if (n >= 10)
      putd(n/10);
   putchar(n%10 + '0');
}
```

- Can this program print

```
INT_MIN == -2147483648
```

## Machine Arithmetic

- Computer values are of fixed-length (32-bits)
- For example, with 6-bits (0..5, right to left)
  - largest number: $111111_2 = 63_{10} = 2^6 - 1$
  - smallest number: $000000_2 = 0$
- What is 50 + 20?

```
  110010
+ 010100
 1000110
```

- Spilling over the lefthand side is overflow



## Machine Arithmetic (cont)

- Sign-magnitude notation
  - bit $n$-1 is the sign; 0 for **+**, 1 for **–**
  - bits $n$-2 through 0 hold an unsigned number
  - largest number: $01111\ldots111_2 = 2^{n-1} - 1$
  - smallest number: $11111\ldots111_2 = -(2^{n-1} - 1)$
  - addition and subtraction are complicated when signs differ, so sign-magnitude is rarely used



## Machine Arithmetic (cont)

- One's-complement notation
  - bit $n$-1 is the sign; bits $n$-2..0 hold an unsigned number
  - bits $n$-2..0 hold the complement of negative numbers
  - $-k = (2^n - 1) - k = 1111\ldots111 - k$
  - largest number: $01111\ldots111_2 = 2^{n-1} - 1$
  - smallest number: $10000\ldots000_2 = -(2^{n-1} - 1)$
  - addition and subtraction are easy, but there are two representations for 0

## Machine Arithmetic (cont)

- Two's-complement notation

  bit $n$-1 is the sign; bits $n$-2..0 hold an unsigned number

  bits $n$-2..0 hold the complement of negative numbers +1

  $-k = (2^n - k) = (2^n - 1) - k + 1$

  largest number: $01111\ldots111_2 = 2^{n-1} - 1$

  smallest number: $10000\ldots000_2 = -2^{n-1}$

  6-bit examples: "complement and increment" to negate

| | | | | |
|---|---|---|---|---|
| +6 | 000110 | 111001 | 111010 | -6 |
| -6 | 111010 | 000101 | 000110 | +6 |
| +0 | 000000 | 111111 | 000000 | +0 |
| +31 | 011111 | 100000 | 100001 | -31 |
| -32 | 100000 | 011111 | 100000 | -32 |



## Machine Arithmetic (cont)

- To add 2's-complement number, ignore signs and add the unsigned bit strings

```
  +20   010100      -20   101100
+ - 7 + 111001    + + 7 + 000111
  +13   001101      -13   110011

  +20   010100      -20   101100
+ + 7 + 000111    + - 7 + 111001
  +27   011011      -27   100101
```

- Signed overflow occurs if the carry into the sign bit differs from the carry out of the sign bit

```
  +20   010100      -20   101100
+ +17 + 010001    + -17 + 101111
  -27   100101      +27   011011
```



## Return to **`putd`** Example

- Convert negative numbers

```
static void putneg(int n) {
   if (n <= -10)
      putneg(n/10);
   putchar("0123456789"[-(n%10)]);
}
void putd(int n) {
   if (n < 0) {
      putchar('-');
      putneg(n);
   } else
   putneg(-n);
}
```

## Portability (cont)

- **`n/10`** and **`n%10`** are implementation dependent when **`n < 0`**

```
int a, b, q, r;
q = a/b; r = a%b;
```

ANSI Standard guarantees only that

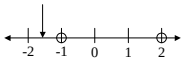
**`q*b + r == a`**
**`|r| < |b|`**
**`r >= 0`** when **`a >= 0 && b > 0`**

**`r`** might be negative if **`a`** is negative



## Portability (cont)

**`5/(-3) = -1.666…`**




```
if 5/(-3) == -2
   5%(-3) = 5 – (-2)(-3) = -1
if 5/(-3) == -1
   5%(-3) = 5 – (-1)(-3) = 2
```



## Portability (cont)

- Check for sign of **`n%10`**; handle both cases

```
static void putneg(int n) {
   int q = n/10, r = n%10;
   if (r > 0) {
      r -= 10;
      q++;
   }
   if (n <= -10)
      putneg(q);
   putchar("0123456789"[-r]);
}
```

## An Easier Way

```
#include <limits.h>
#include <stdio.h>
static void putu(unsigned n) {
   if (n > 10)
      putchar("0123456789"[n%10]);
}
void putd(int n) {
   if (n == INT_MIN) {
      putchar('-');
      putu((unsigned)INT_MAX+1);
   } else if (n < 0) {
      putchar('-');
      putu(-n);
   } else
      putu(n);
}
```



## Increment/Decrement

- Prefix ops increment before returning value

  ```
  n = 5;
  x = ++n;
  ```

  **x** is **6**, **n** is **6**
- Postfix ops increment after returning value

  ```
  n = 5;
  x = n++;
  ```

  **x** is **5**, **n** is **6**
- Operands of **++** and **--** must be variables



## Relational & Logical Ops

- Logical values are **ints**: **0** is false **!0** is true
- Relational ops: **> >= < <=**
- Equality ops: **== !=**
- Unary logical negation: **!**
- Logical connectives: **&& ||**
- Evaluation is left-to-right as far as needed
  - **&&** stops when outcome known to be **0**
  - **||** stops when outcome known to be **!0**

## Bit Operations

- Bitwise logical operations apply to all integers

```
&  bitwise AND           1&1=1 0&1=0
|  bitwise inclusive OR  1|0=1 0|0=0
^  bitwise exclusive OR  1^1=0 1^0=1
~  bitwise complement    ~1=0 ~0=1
```

- The **|** operation is used to "turn on" bits

```
#define BIT0 0x1
#define BIT1 0x2
#define BITS (BIT0 | BIT1)
flags = flags | BIT0;
```

- The **&** op is used to "mask off" bits

```
test = flags & BITS;
```



## Bit Operations (cont)

- Assuming 16-bit quantities

```
BIT0 =          0000000000000001
BIT1 =          0000000000000010
BITS =          0000000000000011
flags =         0100011100000001
flags | BITS =  0100011100000011
flags & BITS =  0000000000000001
```



## Shifting

- Shift operators: **<<  >>**
  - **x<<y** shifts **x** left **y** bit positions
  - **x>>y** shifts **x** right **y** bit positions
- When shifting right:
  - if **x** is signed, may be logical or arithmetic
  - if **x** is unsigned, shift is always logical
  - arithmetic shift fills with sign bit
  - logical shift fills with **0**
- When shifting left: always fill with **0**

## Shifting (cont)

- Assuming 16-bit quantities

```
bits =      110001110000001
bits << 2 = 000111000000100
bits >> 2 = 111100011100000 (arithmetic)
bits >> 2 = 001100011100000 (logical)
```

- Which do you get?

  implementor's choice (i.e., not portable)



## Assignment

- Assignment is an operator, not a statement

```
c = getchar();
if (c == EOF) . . .
```

  can be written as

```
if ((c = getchar()) == EOF) . . .
```

- Watch out for typos

```
if (c = EOF) . . .
```

- Combine assignment with other operators

```
i = i + 2;      is the same as  i += 2;
f = f | BITS    is the same as  f |= BITS;
```