CS61B Lecture #14: Integers

Integer Types and Literals

Type	Bits	Signed?	Literals
byte	8	Yes	Cast from **int**: (byte) 3
short	16	Yes	None. Cast from **int**: (short) 4096

char	16	No	`'a' // (char) 97`
			`'\n'  // newline ((char) 10)`
			`'\t'  // tab ((char) 8)`
			`'\\'  // backslash`
			`'A', '\101', '\u0041'  // == (`

int	32	Yes	`123`
			`0100  // Octal for 64`
			`0x3f, 0xffffffff // Hexadecima`

long	64	Yes	`123L, 01000L, 0x3fL` `1234567891011L`

- Negative numerals are just negated (positive) literals.
- "N bits" means that there are 2^N integers in the domain of the type:
  - If signed, range of values is $-2^{N-1} \mathinner{..} 2^{N-1}-$

1.

- If unsigned, only non-negative numbers, and range is $0..2^N - 1$.

Overflow

- **Problem:** How do we handle overflow, such as occurs in `10000*10000*10000`?
- Some languages throw an exception (Ada), some give undefined results (C, C++)
- Java *defines* the result of any arithmetic operation or conversion on integer types to "wrap around"—*modular arithmetic.*
- That is, the "next number" after the largest in an integer type is the smallest (like "clock arithmetic").
- E.g., `(byte) 128 == (byte) (127+1) == (byte) -128`
- In general,
  - If the result of some arithmetic subexpression is supposed to have type T, an n-bit integer type,
  - then we compute the real (mathematical) value, x,

- and yield a number, x', that is in the range of T, and that is equivalent to x modulo 2^n.
- (That means that $x - x'$ is a multiple of 2^n.)

Modular Arithmetic

- Define $a \equiv b \pmod{n}$ to mean that $a - b = kn$ for some integer k.
- Define the binary operation $a \bmod n$ as the value b such that $a \equiv b \pmod{n}$ and $0 \leq b < n$ for $n > 0$. (Can be extended to $n \leq 0$ as well, but we won't bother with that here.) This is *not* the same as Java's % operation.
- Various facts: (Here, let a' denote $a \bmod n$).

$$
\begin{aligned}
a'' &= a' \\
a' + b'' &= (a' + b)' = a + b' \\
(a' - b')' &= (a' + (-b)')' = (a - b)' \\
(a' \cdot b')' &= a' \cdot b' = a \cdot b' \\
(a^k)' &= ((a')^k)' = (a \cdot (a^{k-1})')', \text{ for } k > 0.
\end{aligned}
$$

Modular Arithmetic: Examples

- `(byte) (64*8)` yields 0, since $512 - 0 = 2 \times 2^8$.
- `(byte) (64*2)` and `(byte) (127+1)` yield -128, since $128 - (-128) = 1 \times 2^8$.
- `(byte) (101*99)` yields 15, since $9999 - 15 = 39 \times \cdot 2^8$.
- `(byte) (-30*13)` yields 122, since $-390 - 122 = -2 \times 2^8$.
- `(char) (-1)` yields $2^{16} - 1$, since $-1 - (2^{16} - 1) = -1 \times 2^{16}$.

Modular Arithmetic and Bits

- Why wrap around?
- Java's definition is the natural one for a machine that uses binary arithmetic.
- For example, consider bytes (8 bits):

Decimal	Binary
101	1100101
$\times$99	1100011
9999	100111|00001111
$-$ 9984	100111|00000000
15	00001111

- In general, bit n, counting from 0 at the right, corresponds to 2^n.
- The bits to the left of the vertical bars therefore represent multiples of $2^8 = 256$.
- So throwing them away is the same as arithmetic modulo 256.

Negative numbers

- Why this representation for -1?

$$\begin{array}{rr|l} & 1 & \;\;00000001_2 \\ + & -1 & \;\;11111111_2 \\ = & 0 & 1|00000000_2 \end{array}$$

  Only 8 bits in a byte, so bit 8 falls off, leaving 0.

- The truncated bit is in the 2^8 place, so throwing it away gives an equal number modulo 2^8. All bits to the left of it are also divisible by 2^8.

- On unsigned types (**char**), arithmetic is the same, but we choose to represent only non-negative numbers modulo 2^{16}:

$$\begin{array}{rr|l} & 1 & \;\;0000000000000001_2 \\ + & 2^{16}-1 & \;\;1111111111111111_2 \\ = & 2^{16}+0 & 1|0000000000000000_2 \end{array}$$

Conversion

- In general Java will silently convert from one type to another if this makes sense and no information is lost from value.
- Otherwise, cast explicitly, as in `(byte) x`.
- Hence, given

```
byte aByte; char aChar; short aShort; int anInt; long aLong;

// OK:
aShort = aByte; anInt = aByte; anInt = aShort;
anInt = aChar; aLong = anInt;

// Not OK, might lose information:
anInt = aLong; aByte = anInt; aChar = anInt; aShort = anInt;
aShort = aChar; aChar = aShort; aChar = aByte;
```

```
  // OK by special dispensation:
  aByte = 13;    // 13 is compile-time
constant
  aByte = 12+100 // 112 is compile-time
constant
```

Promotion

- Arithmetic operations (+, *, ...) *promote* operands as needed.
- Promotion is just implicit conversion.
- For integer operations,
  - if any operand is **long**, promote both to **long**.
  - otherwise promote both to **int**.
- So,

```
  aByte + 3 == (int) aByte + 3   // Type
int
  aLong + 3 == aLong + (long) 3  // Type
long
  'A' + 2 == (int) 'A' + 2       // Type
int
  aByte = aByte + 1              // ILLEGAL (w
```

- But fortunately,

```
aByte += 1;      // Defined as aByte
= (byte) (aByte+1)
```

- Common example:

```
// Assume aChar is an upper-case letter
char lowerCaseChar = (char) ('a' + aChar
- 'A'); // why cast?
```

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical R
10101101 << 3	10101101 >> 3	1010110
01101000	11110101	0001010

- What is:
  - (-1) >>> 29?
  - x << n?
  - x >> n?
  - (x >>> 3) & ((1<<5)-1)?

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	**Set**	**Flip**	**Flip all**
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	**Arithmetic Right**	**Logical**
10101101 << 3	10101101 >> 3	1010110
01101000	11110101	0001010

- What is:

(-1) >>> 29?	$= 7.$
x << n?	
x >> n?	
(x >>> 3) & ((1<<5)-1)?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.

- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical
10101101 << 3	10101101 >> 3	1010110
01101000	11110101	0001010

- What is:

(-1) >>> 29?	$= 7.$
x << n?	$= x \cdot 2^n.$
x >> n?	
(x >>> 3) & ((1<<5)-1)?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical I
10101101 << 3	10101101 >> 3	1010110
01101000	11110101	0001010

- What is:

(-1) >>> 29?	$= 7.$
x << n?	$= x \cdot 2^n.$
x >> n?	$= \lfloor x/2^n \rfloor$ (i.e., ro
(x >>> 3) & ((1<<5)-1)?	

Bit twiddling

- Java (and C, C++) allow for handling integer types as sequences of bits. No "conversion to bits" needed: they already are.
- Operations and their uses:

Mask	Set	Flip	Flip all
00101100	00101100	00101100	
& 10100111	| 10100111	^ 10100111	~ 10100111
00100100	10101111	10001011	01011000

- Shifting:

Left	Arithmetic Right	Logical I
10101101 << 3	10101101 >> 3	1010110
01101000	11110101	0001010

- What is:

(-1) >>> 29?	$= 7.$
x << n?	$= x \cdot 2^n.$
x >> n?	$= \lfloor x/2^n \rfloor$ (i.e., ro
(x >>> 3) & ((1<<5)-1)?	5-bit integer, bi