CS61B Lecture #14: Integers

Integer Types and Literals

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

No	`'a' // (char) 97`
	`'\n' // newline ((char) 10)`
	`'\t' // tab ((char) 8)`
	`'\\' // backslash`
	`'A', '\101', '\u0041' // == (char) 65`

Yes	`123`
	`0100 // Octal for 64`
	`0x3f, 0xffffffff // Hexadecimal 63, -1 (!)`

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

nerals are just negated (positive) literals.

ns that there are 2^N integers in the domain of the type:

range of values is $-2^{N-1} .. 2^{N-1}-1$.

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

Modular Arithmetic

w do we handle overflow, such as occurs in 10000*10000*10000?

ges throw an exception (Ada), some give undefined re-
)

the result of any arithmetic operation or conversion
pes to "wrap around"—*modular arithmetic.*

"next number" after the largest in an integer type is
(like "clock arithmetic").

```
128 == (byte) (127+1) == (byte) -128
```

sult of some arithmetic subexpression is supposed to
T, an n-bit integer type,

ompute the real (mathematical) value, x,

a number, x', that is in the range of T, and that is
to x modulo 2^n.

ns that $x - x'$ is a multiple of 2^n.)

Modular Arithmetic: Examples

8) yields 0, since $512 - 0 = 2 \times 2^8$.

2) and (byte) (127+1) yield -128, since $128 - (-128) =$

*99) yields 15, since $9999 - 15 = 39 \times \cdot 2^8$.

*13) yields 122, since $-390 - 122 = -2 \times 2^8$.

yields $2^{16} - 1$, since $-1 - (2^{16} - 1) = -1 \times 2^{16}$.

Modular Arithmetic and Bits

ound?

tion is the natural one for a machine that uses binary

consider bytes (8 bits):

Decimal	Binary
101	1100101
×99	1100011
9999	100111|00001111
− 9984	100111|00000000
15	00001111

it n, counting from 0 at the right, corresponds to 2^n.

he left of the vertical bars therefore represent multi-
256.

them away is the same as arithmetic modulo 256.

Negative numbers

presentation for -1?

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

n a byte, so bit 8 falls off, leaving 0.

ed bit is in the 2^8 place, so throwing it away gives an
modulo 2^8. All bits to the left of it are also divisible

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

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

Promotion

perations (+, *, . . .) *promote* operands as needed.

just implicit conversion.

perations,

rand is **long**, promote both to **long**.

promote both to **int**.

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

ely,

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

mple:

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


Bit twiddling

C++) allow for handling integer types as sequences of
version to bits" needed: they already are.

nd their uses:

Set	Flip	Flip all
00101100	00101100	
| 10100111	^ 10100111	~ 10100111
10101111	10001011	01011000

	Arithmetic Right	Logical Right
1 << 3	10101101 >> 3	10101100 >>> 3
0	11110101	00010101

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

Bit twiddling

C++) allow for handling integer types as sequences of
version to bits" needed: they already are.

nd their uses:

Set	Flip	Flip all
00101100	00101100	
| 10100111	^ 10100111	~ 10100111
10101111	10001011	01011000

	Arithmetic Right	Logical Right
1 << 3	10101101 >> 3	10101100 >>> 3
0	11110101	00010101

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

Conversion

va will silently convert from one type to another if this
and no information is lost from value.

ast explicitly, as in (byte) x.

```
e; char aChar; short aShort; int anInt; long aLong;
```

```
aByte; anInt = aByte; anInt = aShort;
Char; aLong = anInt;
```

```
, might lose information:
Long; aByte = anInt; aChar = anInt; aShort = anInt;
aChar; aChar = aShort; aChar = aByte;
```

```
special dispensation:
3;    // 13 is compile-time constant
2+100 // 112 is compile-time constant
```


Bit twiddling

C++) allow for handling integer types as sequences of
version to bits" needed: they already are.

nd their uses:

Set	Flip	Flip all
00101100	00101100	
| 10100111	^ 10100111	~ 10100111
10101111	10001011	01011000

	Arithmetic Right	Logical Right
1 << 3	10101101 >> 3	10101100 >>> 3
0	11110101	00010101

1) >>> 29?
<< n?
>> n?
>>> 3) & ((1<<5)-1)?

Bit twiddling

C++) allow for handling integer types as sequences of
version to bits" needed: they already are.

nd their uses:

Set	Flip	Flip all
00101100	00101100	
| 10100111	^ 10100111	~ 10100111
10101111	10001011	01011000

	Arithmetic Right	Logical Right
1 << 3	10101101 >> 3	10101100 >>> 3
0	11110101	00010101

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

Bit twiddling

C++) allow for handling integer types as sequences of
version to bits" needed: they already are.

nd their uses:

	Set	Flip	Flip all
	00101100	00101100	
	| 10100111	^ 10100111	~ 10100111
	10101111	10001011	01011000

	Arithmetic Right	Logical Right
1 << 3	10101101 >> 3	10101100 >>> 3
0	11110101	00010101

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