C expressions

(Reek, Ch. 5)

Shift operations

- Left shift: `value << n`
  - discard the `n` leftmost bits, and add `n` zeroes to the right
- Right shift: `value >> n`
  - Two definitions:
  - logical version:

    discard the `n` rightmost bits, and add `n` zeroes to the left
    - for negative values, the sign bit is the leftmost bit – so logical right shift has the effect of making the value positive
  - arithmetic right shift:

    like logical right shift, but maintain sign bit
  - The distinction is only relevant for negative values

Right shift: example

```
INT_MAX                 01111111111111111111111111111111
2147483647
INT_MAX >> 16           00000000000000001111111111111111
32767

INT_MIN                 10000000000000000000000000000000
-2147483648
```

logical right shift:

```
INT_MIN >> 16           11111111111111110000000000000000
-32768
```

arithmetic right shift:

```
INT_MIN >> 16           00000000000000010000000000000000
32768
```

Bitwise operations

- Binary operators &, |, ^
  - perform bitwise and, or, xor on each bit of the operands

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

- Unary operator ~
  - perform one's complement of the operand: change each 0 to a 1 and vice versa

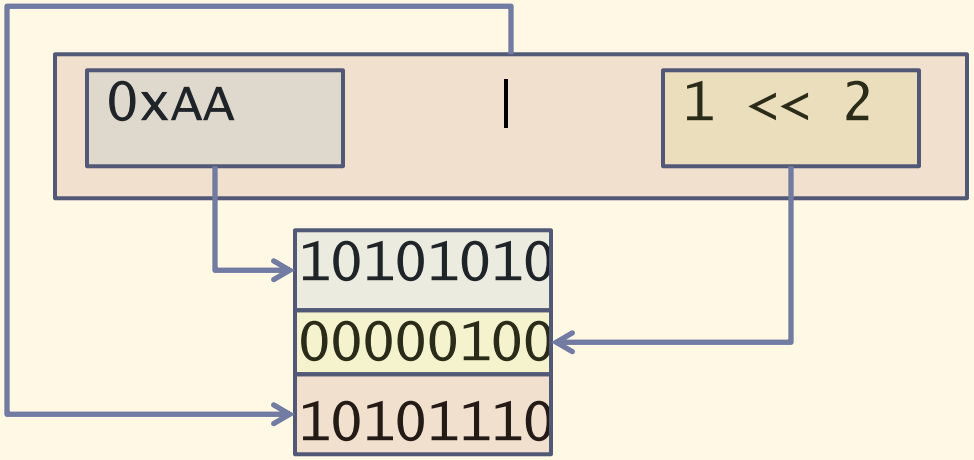
Bitwise operators: Example

```
a:     00101110 (0x2E)          b:     01011011 (0x5B)
~a:    11010001 (0xD1)          ~b:    10100100 (0xA4)

       00101110                00101110                00101110
       01011011                01011011                01011011
a&b    00001010     a|b        01111111     a^b        01110101
       (0x0A)                  (0x7F)                  (0x75)
```

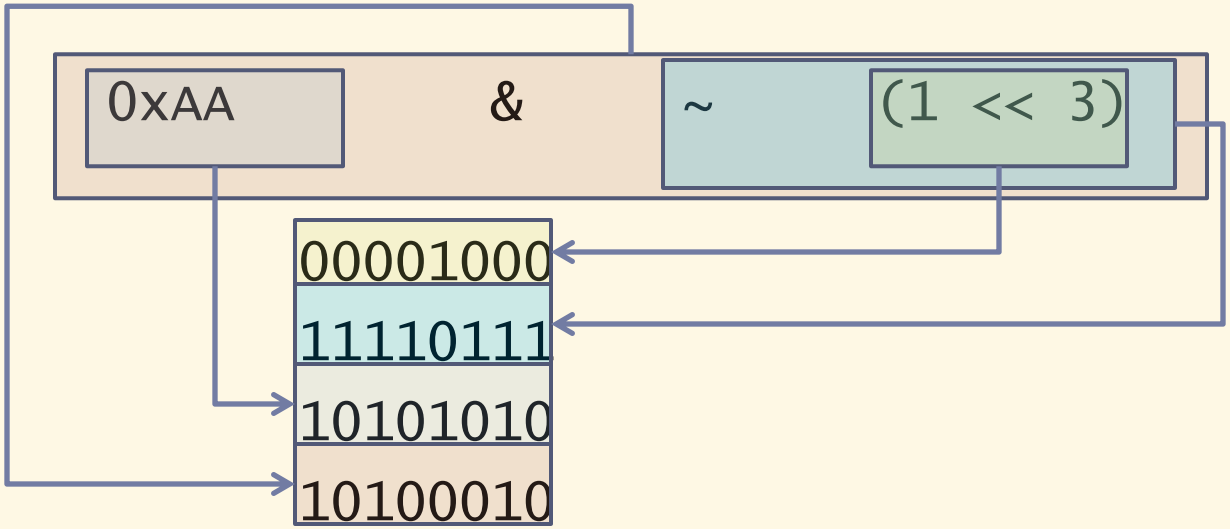
Example: setting a bit to one

```
value =     value | 1 << bit_number;
```



Example: setting a bit to zero

```
value =     value & ~ ( 1 << bit_number );
```



Assignment is an expression

- In many languages, expressions only compute values
- They never have side effects: lasting changes to the program state (e.g. update a variable, print a symbol).
- C (like Java and C++) does have side effects in expressions – notably, assignment expressions (e.g. ==, +=, ++)

Why does `x = y = z = 17` work?

- = is right-associative – so the expression is interpreted as: x = (y = (z = 17)

z is assigned 17;
return value is 17

y is assigned 17;
return value is 17

x is assigned 17;
return value is 17

Another example of = as subexpression

- The `stdio` library function `getchar()` returns a character value (read from the input), or the value `EOF` if end-of-file is detected.

```
int   ch;
while ( (ch = getchar()) != EOF) {
  ... Use the character stored in ch ...
}
```

- Note: `ch` is an `int` because `EOF` is larger than any `char` value

Precedence, associativity, evaluation order

- Precedence: given two operators of *different* types, which is evaluated first?
- Associativity: given a sequence of instances of the *same* operator, are they evaluated left-to-right or right-to-left?
- Evaluation order: given an operator with a number of operands, in what order are the operands evaluated?

Example: Expression evaluation

```
a * b + c * d + e * f
```

- `*` has precedence over `+`
  - So, multiplications are performed before additions
- This leaves us with a sequence of two additions
- `+` has left-to-right associativity, so do the leftmost addition first
- Note: the multiplications can be done in any order
  - `*` has left-to-right associativity, but it doesn't matter, since we don't have a sequence of multiplications

Side effects and evaluation order

```
int i = 10;

i = i-- - --i * ( i = -3 ) * i++ + ++i;
```

- What are the possibilities?
  - Note: `++i` increments `i` and returns the incremented value;
  - `i++` does the same but returns the value before the increment

&& and || have "short circuit" evaluation

- left operand evaluated first; right operand evaluated only if necessary
  - &&: if left operand evaluates to 0, immediately return 0
  - ||: if left operand evaluates to nonzero value, immediately return 1
- This allows you to write code like this with confidence:

```
if    (y == 0 || x / y > MAX) ...
```

Conditional operator: order of evaluation

- Ternary operator: `a ? b : c`
  - If `a` evaluates to nonzero, evaluate `b` and return its value
  - Else evaluate `c` and return its value
  - Only two of the three operands are evaluated at one time

```
if (a)
  result = b;
else
  result = c;
```

vs.

```
result = a ? b : c;
```

More concise, less room for typos – particularly if result is a complex expression

- But avoid abuse of conditional operator
  - e.g. nested conditionals – yikes!

Comma operator

- `a, b` : evaluate `a`, then evaluate `b` and return `b`'s value
- Useful only if `a` has a side effect