Arithmetic Operators

- "Normal" binary arithmetic operators: + − ∗ /
- Modulus or 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 (see H&S, section 7.2.1)
  highest unary − +
  ∗ / %
  lowest + −
  so \(-2*a + b\) is parsed as \(((−2)*a) + b\)
- Associativity: left to right
  \(a + b + c\) is parsed as \(((a + b) + c)\)

Portability: Printing Numbers

- Print a number in decimal
  ```c
  void putd(int n) {
    if (n < 0) {
      putchar('-');
      n = -n;
    }
    if (n >= 10) putd(n/10);
    putchar(n%10 + '0');
  }
  ```
- Can this program print \(\text{INT\_MIN} == -2147483648?\)
Portability: Printing Numbers, Cont’d

• Convert to negative numbers

```c
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);
}
```

• \(n/10\) and \(n\%10\) are “implementation dependent” when \(n < 0\)

Portability, cont’d

• Remainder is a mess:

```c
int a, b, q, r;
q = a/b; r = a%b;
ANSI Standard guarantees only
q*b + r == a
|r| < |b|
r >= 0 when a > 0 && b > 0
r might be negative if a is
```

5/(-3) = -1.666...

| -2 | -1 | 0 | 1 | 2 |

if 5/(-3) == -2,
5%(-3) = 5 - (-2)(-3) = -1

if 5/(-3) == -1,
5%(-3) = 5 - (-1)(-3) = 2

• Check for sign of \(n\%10\), handle both

```c
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]);
}
```

if (-7)/10 == -1,
(-7)%10 = -7 - (-1)(10) = 3

if (-7)/10 == 0,
(-7)%10 = -7 - (0)(10) = -7
An Easier Way

- Use unsigned arithmetic

```c
#include <limits.h>
#include <stdio.h>

static void putu(unsigned n) {
    if (n > 10)
        putu(n/10);
    putchar("0123456789"[n%10]);
}

global 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** operator increments operand **before** returning the value

  ```c
  n = 5;
  x = ++n;
  x is 6, n is 6
  ```

- **Postfix** operator increments operand **after** returning the value

  ```c
  n = 5;
  x = n++;
  x is 5, n is 6
  ```

- Operands of `++` and `--` must be **variables**

  ```c
  ++1
  2 + 3++
  ```
  are illegal
Relational & Logical Operators

- Logical values are ints: 0 is false, !0 is true
- “Normal” relational operators:  >  >=  <  <=
- Equality operators:  ==  !=
- Unary logical negation:  !
- Logical connectives:  &&  ||

Evaluation rules: left-to-right; as far as to determine outcome
  && stops when the outcome is known to be 0
  || stops when the outcome is known to be !0

if (i >= 0 && i < 10 && a[i] == max)
    ++a[i];

- Associativity: left to right; precedence:
  highest
    !
  arithmetic operators
    < <= >= >
    == !=
    &&
  lowest
    ||

Bit Manipulation

- Bitwise logical operators apply to all the bits of an integer value:
  &  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
  unary ~  bitwise complement
  ~1=0  ~0=1

- The  |  operator can be used to “turn on” one or more bits
  #define BIT0 0x1
  #define BIT1 0x2
  #define BITS (BIT0 | BIT1)
  flags = flags | BIT0;

- the  &  operator can be used to “mask off” one or more bits
  test = flags & BITS;

- examples using 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, shift may be arithmetic or logical
  - if \( x \) is unsigned, shift is logical
  - arithmetic shift fills with sign bit
  - logical shift fills with 0

- When shifting left, the vacated bits are always filled with 0

- Examples using 16-bit quantities

  \[
  \begin{align*}
  \text{bits} & = 1100011100000001 \\
  \text{bits} << 2 & = 0001110000000100 \quad \text{(arithmetic, with sign extension)} \\
  \text{bits} >> 2 & = 1111000111000000 \quad \text{(logical)}
  \end{align*}
  \]

Assignment

- Assignment is an operator, not a statement

  \[
  \begin{align*}
  c & = \text{getchar();} \\
  \text{if } (c == \text{EOF}) \ldots \\
  \text{can be written as} \\
  \text{if } ((c = \text{getchar()}) == \text{EOF}) \ldots
  \end{align*}
  \]

- Watch out for “typos” like

  \[
  \begin{align*}
  \text{if } (c = \text{EOF}) \ldots \quad \text{probably meant } ==; \text{power tools can maim...}
  \end{align*}
  \]

- “Augmented” assignment combines + - * / % >> << & ^ | with =

  \[
  \begin{align*}
  i & = i + 2 \\
  \text{flags} & = \text{flags} \mid \text{BIT0} \\
  e_1 \text{ op } e_2 & = e_2 \\
  \quad \text{is the same as} \quad i & = i + 2 \\
  \quad \text{flags} & = \text{flags} \mid \text{BIT0;} \\
  \quad e_1 \text{ op } e_2 & = e_2 \\
  \quad \text{is the same as} \quad e_1 & = e_1 \text{ op } e_2 \\
  \quad \text{except that } e_1 & \text{ is evaluated once}
  \end{align*}
  \]

- Watch out for precedence

  \[
  \begin{align*}
  x & *= y + 1 \text{ means } x *= (y + 1) \\
  \text{not} \quad (x *= y) + 1 \quad (\text{which is also legal})
  \end{align*}
  \]
Conversions

- **Implicit** conversions occur in expressions and across assignments

- In expressions with mixed types, “Promote” to the “higher” type
  - `int + float → float + float`
  - `short + long → long + long`

- Watch out for sign extension! e.g. `char → int`
  - `char c = '\377'; int i = c;` is `i` equal to `0377` or `-1`? when in doubt, mask: `i = c&0377`

- Assigning a “big” `int` to a “small” `int`, causes the extra bits to be **discarded**

- Assigning a `float` or `double` to an `int` **truncates**
  - `int n = 2.5` **assigns** `2` to `n`

- **Explicit** conversions are specified with **casts**: `(type)expr`
  - `sqrt((double)n)`
  - `(int)1.5`

- study H&S, section 6.2 **carefully**

---

Evaluation Order

- Except for `&&` and `||`, the evaluation order of expressions is **undefined**;

- Avoid expressions whose outcome might depend on evaluation order
  - `x = f() + g();`
  - `a[i] = i++;`
  - `f(++n, g(n));`

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! &amp; * (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>/ %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;= &gt;= !== &amp;^</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>??</td>
<td></td>
</tr>
<tr>
<td>?: = += -= /= %= ^=</td>
<td>= &lt;&lt;= &gt;&gt;=</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>