Expression and Operator

- Two types of Expression:
  - The expression is a function call: `printf("%d",x);`
  - The expressions is formed by data (variables/ constants) and operators:
    - `3 + 5; x=0; x=x+1;`
- An expression in C usually has an associated value
  - `3+5; //returns 8`
  - `x=0; //returns 0`
  - except for the function calls that return `void`.

Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Adds operands</td>
<td><code>x + y</code></td>
</tr>
<tr>
<td>-</td>
<td>Subtract second from the first</td>
<td><code>x - y</code></td>
</tr>
<tr>
<td>*</td>
<td>Multiply operands</td>
<td><code>x * y</code></td>
</tr>
<tr>
<td>/</td>
<td>Divides first by second</td>
<td><code>x / y</code></td>
</tr>
<tr>
<td>%</td>
<td>Remainder of divide operation</td>
<td><code>x % y</code></td>
</tr>
</tbody>
</table>

- `%` can not be applied to a float or double: compile time error
- `3.2 % 1.6 ??`
- Arithmetic operators associates left to right.
- `12 + 6 – 7 = 11`

Relational Operators

- Relational operators are used to compare data values. They return a 1 value for true and a 0 for false.
- `i < lim-1` is equivalent to `i < (lim - 1)`

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td><code>==</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Greater than</td>
<td><code>&gt;</code></td>
<td><code>x &gt; y</code></td>
</tr>
<tr>
<td>Less than</td>
<td><code>&lt;</code></td>
<td><code>x &lt; y</code></td>
</tr>
<tr>
<td>Greater/equal</td>
<td><code>&gt;=</code></td>
<td><code>x &gt;= y</code></td>
</tr>
<tr>
<td>Less than/equal</td>
<td><code>&lt;=</code></td>
<td><code>x &lt;= y</code></td>
</tr>
<tr>
<td>Not equal</td>
<td><code>!=</code></td>
<td><code>x != y</code></td>
</tr>
</tbody>
</table>

- There is no bool type in C. Instead, C uses:
  - `0` as false
  - Non-zero integer as true
- Relational operators have lower precedence than arithmetic operators:
Logical Operators

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>Any</td>
<td>false</td>
</tr>
</tbody>
</table>

| A   | B   | A || B |
|-----|-----|-------|
| true| Any| true  |
| false| Any| true  |
| false| false| false |

<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

lazy evaluation:

If A is false, returns false without testing B

If A is true, returns true without testing B

When there is need to evaluate both A and B, use & instead of &&, | instead of ||

```
if( (interest=amount*rate) > 100 || (payment=amount/12) > 200)
    balance=balance-interest-payment;
```

Exercise

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Write a loop equivalent to the for loop above without using && or ||

Type Conversion

- Implicit type conversions:
  - if a binary operator has operands of different types, the "lower" type is promoted to the "higher" type before the operation proceeds.
    - int + double // int is promoted to double
  - The value of the right side of = is converted to the type of the left, which is the type of the result.
    - int x = 0.2; // x has value 0
    - Java compiler would report an error.

Increment and Decrement

- Pre-increment and decrement operators:
  - increase/decrease the operand before it is used.
  - ++x is equivalent to x=x+1 (same applies to -x)
  - int count=0; printf("%d",++count); //add 1 to count, print 1
- Post-increment and decrement operators.
  - increase/decrease the operand after it is used.
  - x++ is equivalent to x=x+1 (same applies to x--)
  - int count=0; printf("%d",count++); //print 0, add 1 to count
Exercise

/* squeeze: delete all c from s */
void squeeze(char s[], char c)
{
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];  //s[++j]=s[i]?
    s[j] = '\0';
}

Bitwise Operators

- C allows you to operate on the bit representations of integer variables.
- All integers can be thought of in binary form.
  - For example, suppose ints have 16-bits
    - \( 65520_{10} = 1111 1111 1111 0000_2 = \text{FF00}_{16} = 17760_{8} \)
- In C, hexadecimal literals begin with 0x, and octal literals begin with 0.
  * x=65520; base 10
  * x=0xffff; base 16 (hex)
  * x=0177760; base 8 (octal)

Bitwise Shift Operators

- \( x << n \ (n \geq 0) \)
  - Shifts \( x \) \( n \) bit positions to the left, add 0 on the right.
  - If \( x = 1111 1111 1111 0000_2 \)
    - \( x << 1 \) equals \( 1111 1111 1111 1000_2 \)
- \( x >> n \ (n\geq0) \)
  - Shifts \( x \) \( n \) bit positions right.
    - add the sign if it is a signed integer (arithmetic shift)
    - add 0 if it is an unsigned integer
  - \( x >> 1 \) is \( 0111 1111 1111 1000_2 \) (unsigned)
  - \( x >> 1 \) is \( 1111 1111 1111 1000_2 \) (signed)

Shift, Multiplication and Division

- Multiplication and division are often slower than shift.
- Multiplying 2 can be replaced by shifting 1 bit to the left.
  \[ n = 10 \]
  \[ \text{printf}("%d = %d", n*2, n<<1); \]
  \[ \text{printf}("%d = %d", n*4, n<<2); \]
- Division by 2 can be replace by shifting 1 bit to the right.
  \[ n = 10 \]
  \[ \text{printf}("%d = %d", n/2, n>>1); \]
  \[ \text{printf}("%d = %d", n/4, n>>2); \]
Bitwise Logical Operators

• Work on all integer types
  – & Bitwise AND
    \[ x = 01001000 (72) \]
    \[ y = 10111000 (184) \]
    \[ x \& y = 00001000 (8) \]
  – | Bitwise Inclusive OR
    \[ x | y = 11111000 (248) \]
  – ^ Bitwise Exclusive OR
    \[ x ^ y = 11110000 \]
  – ~ The complement operator
    \[ \sim y = 01000111 \]
    – Complements all of the bits of \( x \)

Exercise

• int \( x = 1; \)
• int \( y = 2; \)
• \( x \& x \) ?
• \( x \& y \) ?
• \( x | | y \) ?
• \( x \& y \) ?
• \( \sim y \) ?

Assignment Operators

• C offers a short form for expression where the variable on the left side is repeated immediately on the right:
  variable = variable op expr => variable op= expr

Example

\[ /* \text{bitcount: count the number of 1 bits in } x */ \]
\[ \text{int bitcount(unsigned } x \text{)} \]
\[ \{ \]
\[ \text{int } b; \]
\[ \text{for } (b=0; x!-0; x >>= 1) \]
\[ \text{if } (x \& 1) \]
\[ \text{b++;} \]
\[ \text{return } b; \]
\[ \} \]
Conditional Expression

- **Generic Form**
  \[ \exp_1 \ ? \ \exp_2 : \exp_3 \]

- **Equivalent to**
  - if \( \exp_1 \) is true (non-zero)
    - value is \( \exp_2 \)
    - \( \exp_3 \) is not evaluated
  - else // \( \exp_1 \) is false (0),
    - value is \( \exp_3 \)
    - \( \exp_2 \) is not evaluated

- **Example:**
  \[ z = (x > y) \ ? \ x : y; \]
  \[ \text{This is equivalent to:} \]
  - if \( (x > y) \)
    - \( z = x; \)
  - else
    - \( z = y; \)

Comma Operator

- An expression can be composed of multiple subexpressions separated by commas.
  - Subexpressions are evaluated left to right.
  - The entire expression evaluates to the value of the **rightmost** subexpression.

- **Example:**
  ```java
  int a = 1, b=10;
  x = (a++, b++);
  // a is incremented
  // b is assigned to x
  // b is incremented
  ```
  - Parenthesis are required because the comma operator has a lower precedence than the assignment operator!

- The comma operator is often used in for loops.

Operator Precedence (decreasing order)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Left to right</td>
</tr>
<tr>
<td>~, ++, --</td>
<td>Right to left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Left to right</td>
</tr>
<tr>
<td>+, -</td>
<td>Right to right</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>Left to right</td>
</tr>
<tr>
<td>==, !=</td>
<td>Left to right</td>
</tr>
<tr>
<td>&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>=, +=, -=</td>
<td>Right to left</td>
</tr>
<tr>
<td>? :</td>
<td>Right to left</td>
</tr>
<tr>
<td>,</td>
<td>Left to right</td>
</tr>
</tbody>
</table>