Expression and Operator
Expressions and Operators

• Examples:
  
  3 + 5;
  x;
  x=0;
  x=x+1;
  printf("%d",x);

• Two types:
  – Function calls
  – The expressions formed by data and operators

• An expression in C usually has a value
  – except for the function call that returns `void`. 
# Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>Adds operands</td>
<td>x + y</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>Subs second from first</td>
<td>x - y</td>
</tr>
<tr>
<td>Negation</td>
<td>-</td>
<td>Negates operand</td>
<td>-x</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>Multiplies operands</td>
<td>x * y</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>Divides first by second</td>
<td>x / y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(integer quotient)</td>
<td></td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>Remainder of divide op</td>
<td>x % y</td>
</tr>
</tbody>
</table>
Assignment Operator

◆ x=3
  – = is an operator
  – The value of this expression is 3
  – = operator has a side effect -- assign 3 to x

◆ The assignment operator =
  – The side-effect is to assign the value of the right hand side (rhs) to the left hand side (lhs).
  – The value is the value of the rhs.

◆ For example:

  
x = ( y = 3 ) +1; /* y is assigned 3 */
  /* the value of (y=3) is 3 */
  /* x is assigned 4 */
## Compound Assignment Operator

- Often we use “update” forms of operators
  - \( x = x + 1, \ x = x * 2, \ldots \)
- C offers a short form for this:
  - Generic Form
    \[
    \text{variable op= expr} \quad \text{equivalent to} \quad \text{variable = variable op expr}
    \]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Equivalent to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x *= y )</td>
<td>( x = x * y )</td>
</tr>
<tr>
<td>( y -= z + 1)</td>
<td>( y = y - (z + 1) )</td>
</tr>
<tr>
<td>( a /= b )</td>
<td>( a = a / b )</td>
</tr>
<tr>
<td>( x += y / 8 )</td>
<td>( x = x + (y / 8) )</td>
</tr>
<tr>
<td>( y %= 3 )</td>
<td>( y = y % 3 )</td>
</tr>
</tbody>
</table>

- Update forms have value equal to the final value of expr
  - i.e., \( x=3; \ y= (x+=3); \ /* \ x \ and \ y \ both \ get \ value \ 6 */ \)
Increment and Decrement

- Other operators with side effects are the pre- and post-increment and decrement operators.
  - **Increment**: `++`, `++x`, `x++`
    - `++x` is the same as: `(x = x + 1)`
    - Has value `x_{old} + 1`
    - Has side-effect of incrementing `x`
    - `x++`
    - Has value `x_{old}`
    - Has side-effect of incrementing `x`
  - **Decrement**: `--`, `--x`, `x--`
    - Similar to `++`
Relational Operators

- Relational operators allow you to compare variables.
  - They return a 1 value for true and a 0 for false.

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

- There is no `bool` type in C. Instead, C uses:
  - 0 as false
  - Non-zero integer as true
Logical Operators

- `&&` AND
- `||` OR
- `!` NOT

`!((a>1) && (a<10)) || ((a<-1) && (a>-10))`
C allows you to operate on the bit representations of integer variables.

- Generally called bit-wise operators.

All integers can be thought of in binary form.

- For example, suppose ints have 16-bits

\[ 65520_{10} = 1111 \ 1111 \ 1111 \ 0000_2 = FFF0_{16} = 177760_8 \]

In C, hexadecimal literals begin with \( 0x \), and octal literals begin with \( 0 \).

- \( x=65520; \)  \hspace{1cm} \text{base 10} \\
- \( x=0xffff0; \)  \hspace{1cm} \text{base 16 (hex)} \\
- \( x=0177760; \)  \hspace{1cm} \text{base 8 (octal)}
Operating on Bits (2)

Bitwise operators

- The shift operator:
  - \( x << n \)
    - Shifts the bits in \( x \) \( n \) positions to the left, shifting in zeros on the right.
    - If \( x = 1111\ 1111\ 1111\ 0000_2 \)
      - \( x << 1 \) equals \( 1111\ 1111\ 1110\ 0000_2 \)
  - \( x >> n \)
    - Shifts the bits in \( x \) \( n \) positions right.
      - shifts in the sign if it is a signed integer (arithmetic shift)
      - shifts in 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)
Operating on Bits (3)

- Bitwise logical operations
  - Work on all integer types
    - & Bitwise AND
      \[ x = 0xFFF0 \]
      \[ y = 0x002F \]
      \[ x \& y = 0x0020 \]
    - | Bitwise Inclusive OR
      \[ x | y = 0xFFFF \]
    - ^ Bitwise Exclusive OR
      \[ x ^ y = 0xFFDF \]
    - ~ The complement operator
      \[ \sim y = 0xFFD0 \]
      - Complements all of the bits of X
Shift, Multiplication and Division

◆ Don’t try this at home!
◆ Multiplication and division is often slower than shift.
◆ Multiplying 2 can be replaced by shifting 1 bit to the left.

```
n = 10
printf("%d = %d" , n*2, n<<1);
printf("%d = %d" , n*4, n<<2);
......
```

◆ Division by 2 can be replace by shifting 1 bit to the right.

```
n = 10
printf("%d = %d" , n/2, n>>1);
printf("%d = %d" , n/4, n>>2);
```

An optimising compiler will make this transformation automatically.
### Operator Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence level</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>1</td>
</tr>
<tr>
<td>~, ++, --, unary -</td>
<td>2</td>
</tr>
<tr>
<td>*, /, %</td>
<td>3</td>
</tr>
<tr>
<td>+, -</td>
<td>4</td>
</tr>
<tr>
<td>&lt;&lt;=, &gt;&gt;=</td>
<td>5</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>6</td>
</tr>
<tr>
<td>==, !=</td>
<td>7</td>
</tr>
<tr>
<td>&amp;</td>
<td>8</td>
</tr>
<tr>
<td>^</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=, -=, etc.</td>
<td>14</td>
</tr>
</tbody>
</table>

◆ We’ll be adding more to this list later on...
An Example

- What is the difference between the two lines of output?

```c
#include <stdio.h>

int main ()
{
    int w=10,x=20,y=30,z=40;
    int temp1, temp2;
    temp1 = x * x /++y + z / y;
    printf("temp1= %d;\nw= %d;\nx= %d;\ny= %d;\nz= %d\n",
        temp1, w,x,y,z);
    y=30;
    temp2 = x * x /y++ + z / y;
    printf("temp2= %d;\nw= %d;\nx= %d;\ny= %d;\nz= %d\n",
        temp2, w,x,y,z);
    return 0;
}
```
Conditional Operator

- The conditional operator essentially allows you to embed an “if” statement into an expression

- **Generic Form**
  \[ \text{exp1} \ ? \ \text{exp2} : \ \text{exp3} \]
  
  - if \( \text{exp1} \) is true (non-zero)
  - value is \( \text{exp2} \)
  - (\( \text{exp3} \) is not evaluated)
  
  - if \( \text{exp1} \) is false (0),
  - value is \( \text{exp3} \)
  - (\( \text{exp2} \) is not evaluated)

- **Example:**
  
  \[ z = (x > y) \ ? \ x : y; \]
  
  - 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:
  ```
  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.
Example:

```c
int i, sum;
for (i=0, sum=0; i<100; i++) {
    sum += i;
}
printf("1+...+100 = %d", sum);
```