Expressions and Data Types (continued)
Logical Expressions

• Relation operators: <, <=, >=, >
• Equality operators: ==, !=
• Logical operators: !, &&, ||
• Produce 0 or 1

(1 <= j) && (j <= 100)

!answer || count >= 5
Logical Expressions

• !(a || b) == !a && !b
• !(a && b) == !a || !b

!answer || count >= 5
!(answer && count < 5)
Bitwise Operators
Bitwise Operators

1 1 1 1 1 1 1 1 1 1 1 1
Bitwise Operators

0 1 0 0 0 0 0 0 0 0 0
Bitwise Operators: Example

• Binary representation:

```java
int i = 22; /* 10110 */
int j = 91; /* 1011011 */
```

• From binary to decimal

\[10110_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0\]

\[= 22\]
Bitwise Operators: Example

• Binary representation:
  
  int i = 22;       /* 10110 */
  int j = 91;       /* 1011011 */

• From decimal to binary ?
From Decimal to Binary (I)

(http://www.wikihow.com/Convert-from-Decimal-to-Binary)
From Binary to Decimal

http://www.wikihow.com/Convert-from-Binary-to-Decimal
From Binary to Decimal

\[
\begin{align*}
128 & \quad 64 & \quad 32 & \quad 16 & \quad 8 & \quad 4 & \quad 2 & \quad 1 \\
1 & \quad 0 & \quad 0 & \quad 1 & \quad 1 & \quad 0 & \quad 1 & \quad 1 \\
\hline
128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 &= 155
\end{align*}
\]
Hexadecimal Representation

\[ \text{mask} = \underbrace{1000}_{0x} \underbrace{0000}_{0x} \underbrace{0000}_{0x} \underbrace{0000}_{0x} = \]

\[ 0x8000 \]

Prefix "0x" means this is a hexadecimal representation
Hexadecimal Representation

- Exercise: what's the hexadecimal representation for
  \[
  \begin{array}{ccccccc}
  1 & 0 & 1 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]
  \[
  0x\ 10 \ 0 \ 0 \ 0 \\
  \]
  \[
  0x\ A \ 0 \ 0 \ 0 \\
  \]

- In hexadecimal representation, we use A ~ F to represent 10 ~ 15
Hexadecimal Representation

- From hexadecimal to decimal

\[ 0xA123 = 3 \times 16^0 + 2 \times 16^1 + 1 \times 16^2 + ? \times 16^3 \]

\[ = 3 \times 16^0 + 2 \times 16^1 + 1 \times 16^2 + 10 \times 16^3 \]

\[ = 41251 \]
Bitwise Operators
Bitwise Operators

• For bit manipulation:
  – Bitwise AND: &
  – Bitwise inclusive OR: |
  – Bitwise exclusive OR: ^
  – Bitwise One's complement: ~
  – Left shift: <<
  – Right shift: >>
Bitwise AND: &

AND gate:

A and B are single bits (0 or 1)

<table>
<thead>
<tr>
<th>2 Input AND gate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

The bitwise AND (&) should not be confused with the logical AND (&&) operator. For example:

1 & 4 evaluates to 0 (0001 & 0100 = 0000) while
1 && 4 evaluates to true [True && True = True]
The `|` (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one bitwise inclusive OR operator.

The bitwise OR (|) should not be confused with the logical OR (||) operator. For example:

1 | 4 evaluates to 5 (0001 | 0100 = 0101) while 1 || 4 (True || True = True) evaluates to true
Bitwise complement: ~

NOT gate:

<table>
<thead>
<tr>
<th>A</th>
<th>(\bar{A})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The ~ (bitwise negation) operator yields the bitwise (one) complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must have an integral type. The result has the same type as the operand but is not an lvalue (left value). The symbol used called tilde.
Bitwise exclusive OR (XOR): ^

XOR gate:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ⊕ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Bitwise XOR Operator

- The bitwise exclusive OR operator (^) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.
## Bitwise shift operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Left shift operator, shift their first operand left (<code>&lt;&lt;</code>) by the number of positions specified by the second operand.</td>
<td><code>nbits &lt;&lt; nshiftSize</code></td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Right shift operator, shift their first operand right (<code>&gt;&gt;</code>) by the number of positions specified by the second operand.</td>
<td><code>nbits &gt;&gt; nshiftSize</code></td>
</tr>
</tbody>
</table>

- For left-shift operators without overflow, the statement:
  
  ```
  expression1  << expression2
  ```

is equivalent to multiplication by \(2^{expression2}\).

- For right-shift operators:
  
  ```
  expression1  >> expression2
  ```

is equivalent to division by \(2^{expression2}\) if `expression1` is unsigned or has a nonnegative value.
Bitwise Operators: Example

• Bitwise &: 

\[
\text{result} = i \& j; \\
00010110 \\
01011011 \\
\downarrow \\
00010010 \Leftrightarrow 2^1 + 2^4 = 18
\]

• Bitwise ~ (complement): 

\[
\text{result} = \sim i; \\
000000000000010110 \\
1111111111101001 \Leftrightarrow \\
65,513
\]
Bitwise Operators: Example

• Bitwise exclusive or ^:

\[
\text{result} = i \ ^\ ^\ j;
\]

0000000000010110
0000000001011011
\[\overline{0000000001011011}\]
00000000001001101 \[\equiv 77\]

• Bitwise inclusive or |

\[
\text{result} = i \ | \ j;
\]

0000000000010110
0000000001011011
\[\overline{0000000001011011}\]
00000000001011111 \[\equiv 95\]
Bitwise Operators: Example

• Left shift:

```
result = i << 3;
0000000000010110
```

```
0000000010110000 ⇔ 176
```

• Right shift:

```
result = i >> 2;
0000000000010110
```

```
0000000000000101 ⇔ 5
```
# Precedence and Associativity of C Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type of Operation</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] ( ) . → postfix ++ and postfix --</td>
<td>Expression</td>
<td>Left to right</td>
</tr>
<tr>
<td>prefix ++ and prefix -- sizeof &amp; * + – ~ !</td>
<td>Unary</td>
<td>Right to left</td>
</tr>
<tr>
<td>typecasts</td>
<td>Bondary</td>
<td>Right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplicative</td>
<td>Left to right</td>
</tr>
<tr>
<td>+ –</td>
<td>Additive</td>
<td>Left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Bitwise shift</td>
<td>Left to right</td>
</tr>
<tr>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Relational</td>
<td>Left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>Equality</td>
<td>Left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise-AND</td>
<td>Left to right</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise-exclusive-OR</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise-inclusive-OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical-AND</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>Conditional-expression</td>
<td>Right to left</td>
</tr>
<tr>
<td>= *= /= %=</td>
<td>Simple and compound</td>
<td>Right to left</td>
</tr>
<tr>
<td>+= – = &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
<td>Assignment</td>
<td>Right to left</td>
</tr>
<tr>
<td>,</td>
<td>Sequential evaluation</td>
<td>Left to right</td>
</tr>
</tbody>
</table>
Associativity

• When there are multiple operators with the same precedence, which one to execute first

E.g., \( a \times b \div c \rightarrow (a \times b) \div c \)

\[
a = b \mathrel{+}= c \rightarrow a = (b \mathrel{+}= c)
\]
Expression Evaluation

• **Postfix** operator have higher precedence

• Expressions are evaluated according to precedence order of operators

\[ a = b+c++-d+-e/-f \]
Expression Evaluation

- **Postfix** operator have higher precedence

- Expressions are evaluated according to precedence order of operators

  \[ a = b+=(c++) - d + (--e) / (-f) \]
  \[ a = b+ = (c++) - d + ((--e) / (-f)) \]
  \[ a = b+ = ((c++) - d + ((--e) / (-f))) \]
  \[ a = (b+ = (c++) - d + ((--e) / (-f)))) \]