C Expressions, Operators, and Flow of Control

C Programming and Software Tools

N.C. State Department of Computer Science
Outline

• Expressions

• Operators
  – Single operand
  – Two operands
  – Relational
  – Logical
  – Assignment

• Statement Separation

• C Operator Precedence and Order of Evaluation

• Flow of Control
Expressions

• Most statements in a C program are *expressions*
• *Evaluating* an expression means doing the computation according to the definition of the operations specified
• *Results* of expression evaluation
  – the *value* returned (and assigned); and/or
  – *side effects* (other changes to variables, or output, along the way)

```c
j = k + 3 * m++;  // Example expression
```
Comparison: C vs. Java Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associates</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>access class feature</td>
<td>left-to-right</td>
</tr>
<tr>
<td>a[]</td>
<td>array index</td>
<td></td>
</tr>
<tr>
<td>fn()</td>
<td>function call</td>
<td></td>
</tr>
<tr>
<td>++ --</td>
<td>post-inc/dec</td>
<td>left-to-right</td>
</tr>
<tr>
<td>++ --</td>
<td>pre-inc/dec</td>
<td>right-to-left</td>
</tr>
<tr>
<td>~</td>
<td>bitwise not</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
<td></td>
</tr>
<tr>
<td>- +</td>
<td>unary -/+</td>
<td></td>
</tr>
<tr>
<td>&amp; *</td>
<td>address/dereference</td>
<td></td>
</tr>
<tr>
<td>(type)</td>
<td>cast</td>
<td></td>
</tr>
<tr>
<td>new</td>
<td>object allocation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associates</th>
</tr>
</thead>
<tbody>
<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;&gt;</td>
<td>left, right shift</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;= &gt; &gt;=</td>
<td>relational</td>
<td>left-to-right</td>
</tr>
<tr>
<td>== !=</td>
<td>equality/ineq.</td>
<td>left-to-right</td>
</tr>
<tr>
<td>instanceof</td>
<td>test object type</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
<td>left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise xor</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise 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>assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>*= /= %=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;= ^=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- C operators not found in Java:
  - pointer operations (->, &, *)
  - sizeof
  - sequential evaluation (,)
What Are the C Operators?

- There are approximately 50 of them
- Most operators do the same thing in Java and C
- Categories of operators
  1. “other”
  2. arithmetic
  3. logical and relational
  4. assignment
  5. bit operators
Other Operators

- Array indexing (\texttt{x[]})
- Function calls (\texttt{f()})
- Address-of (\texttt{&x}) operator, and pointer dereferencing (\texttt{*x})
  - and effect of other operators on pointers
- Member (of \texttt{struct}) specification
  - direct (\texttt{x.y}) and indirect (\texttt{x->y})
- The \texttt{sizeof()} operator
- casting: (\texttt{type}) operand

Later

Covered

Later

Covered

Covered
Arithmetic: Ops on a Single Operand

Unary plus (+a): no effect

```
a = +b;
```

Unary minus (−b): change sign of operand

```
a = −b;
```

Increment (++) and decrement (--) operators

- operand type must be modifiable (not a constant)
- these operators have side effects!

```
a = ++b / c--; 
```
prefix: side effect takes place first, then expression value is determined

```c
int i = 1, j = 8;
printf("%d %d\n", ++i, --j);
printf("%d %d\n", i, j);
```

postfix: expression uses old operand value first, then side effect takes place

```c
int i = 1, j = 8;
printf("%d %d\n", i++, j--);
printf("%d %d\n", i, j);
```
Arithmetic on Two Operands

• Multiply (*), Quotient (/), Remainder (%), Add (+), Subtract (−)
  – possibility of underflow and overflow during expression evaluation, or assignment of the results

• Divide by zero
  – causes program execution failure if the operands are integer type
  – generates a special value (inf) and continues execution if the operands are IEEE floating point
Arithmetic on Two Operands

• Modulus operator (\%) operands must have type integer, should both be positive*

\begin{verbatim}
printf("%d", (37 % 3));
\end{verbatim}

\begin{verbatim}
printf("%d", (-37 % 3));
\end{verbatim}

• Result of a\%b is program exception if b == 0

* If one operand is negative, result depends on the language. To check your language, consult this handy table**

** Wait, let’s just never do that.

<table>
<thead>
<tr>
<th>Language</th>
<th>Operator</th>
<th>Result has the same sign as</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionScript</td>
<td>%</td>
<td>Dividend</td>
</tr>
<tr>
<td>Ada</td>
<td>mod</td>
<td>Divisor</td>
</tr>
<tr>
<td></td>
<td>rem</td>
<td>Dividend</td>
</tr>
</tbody>
</table>
Relational and Logical Operators

Used in evaluation conditions

```c
if (expression evaluates to TRUE)
  ...do something...
```

What is TRUE (in C)?

- 0 means FALSE
- anything else (1, −96, 1.414, ‘F’, inf) means TRUE
- ???

```c
float f = 9593.264;
if (f)
  ...do something...
```
Relational Operators

Six comparison operators:  \( <, >, ==, !=, >=, <= \)

\[
\begin{align*}
\text{if} & \; (a < b) \; \ldots \\
\text{if} & \; (x >= y) \; \ldots \\
\text{if} & \; (q == r) \; \ldots \\
\end{align*}
\]

- Operands must be numbers (integer or floating point), result type is **int**
  - i.e., cannot use to compare structs, functions, arrays, etc.
- If relation is true, result is 1, else result is 0

```c
float f = 9593.264;
if (f != 0)  
   ...do something...
```

same meaning as previous slide
Relational Operators (cont’d)

• Most common mistake in C (in my experience)
  
  `==` is relational comparison for equality

  `=` is assignment!

Example: some strategic defense code...

```c
if (enemy_launch = confirmed)
    retaliate();
```

Oops... sorry!
Logical Operators

Logical operators allow construction of complex (compound) conditions

Operands must be (or return) numbers (integer or floating point), result type is int

Logical NOT (!) operator

- result: 1 (TRUE) if operand was 0 (FALSE), otherwise 0

```c
int j = ...;
if (! j)
    ... do something ...

float f = ..., g = ...;
if (! (f < g) )
    ... do something ...
```
Logical ... (cont’d)

• AND (&&):
  – evaluate first operand, if 0, result is 0; else,
  – evaluate second operand, if 0, result is 0; else,
  – result is 1

```c
if (x && (y > 32))
    ... do something ...
```
Logical... (cont’d)

• Condition evaluation stops as soon as truth value is known
  – i.e., order of the operands is significant
• Relied on by many programs!

```c
if ((b != 0) && ((a / b) > 5))
  printf(“quotient greater than 5
”);
```

what’s the difference???

```c
if (((a / b) > 5) && (b != 0))
  printf(“quotient greater than 5
”);
```
Logical... (cont’d)

- **OR (||) operator**
  - evaluate *first* operand, if *not* 0, result is 1;
  - otherwise, evaluate *second* operand, if *not* 0, result is 1;
  - otherwise, result is 0

- **There is no logical XOR in C**
  - but \((a \text{ XOR } b) \Rightarrow (a \text{ && } (! b)) \text{ || } ((! a) \text{ && } b)\)
A Strange Idea?

• Mixing relational, bit-wise, and arithmetic operations into a single expression

```c
unsigned char g, h;
int a, b;
float e, f;

... 
if ((a < b) && (e * f || (g ^ h)))
  ...do something here...
```

```c
int a = -4;
char c = 'D';
float e = 0.0, f = 22.2, g;

... 
g = (c == 'D') + (e || f) * a;
```
Assignment Operators

• \( a = b \) assigns the value of \( b \) to \( a \)
  
  – \( a \) must be a reference and must be *modifiable* (not a function, not an entire array, etc.)

• Both \( a \) and \( b \) must be one of the following
  
  – numbers (integer or floating), or
  
  – structs or unions of the same type, or
  
  – pointers to variables of the same type

---

**OK**

\[
\begin{align*}
\text{float } & \ a; \\
\text{int } & \ b = 25; \\
\text{a } & = \ b;
\end{align*}
\]

**Not OK**

\[
\begin{align*}
\text{float } & \ a[2]; \\
\text{int } & \ b[2] = \{25, 15\}; \\
\ \ & \ a = \ b;
\end{align*}
\]
Assignment Operators (cont’d)

• \( a \ op= b \)
  – where \( op \) is one of *, /, %, +, -, <<, >>, &, ^, |
  – “shorthand” for \( a = a \ op b \)

```c
int i = 30, j = 40, k = 50;
i += j;  // same as \( i = i + j \)
k %= j;  // same as \( k = k \ % j \)
j *= k;  // same as \( j = j \ * k \)
```
Statement Termination and the “,”

• Normally, statements are executed sequentially and are separated by ;

• Another separator: ‘,’ (e.g., \( j = k++ \), \( i = k \);):
  1. evaluate expressions left to right
  2. complete all side effects of left expression before evaluating right expression
  3. result is value of the right expression

• More shorthand?
Constant Expressions

• Constant-valued expressions are used in...
  – case statement labels
  – array bounds
  – bit-field lengths
  – values of enumeration constants
  – initializers of static variables

\[
\text{static int } a = 35 + (16 \% (4 | 1)) ;
\]

(static: variable’s value is initialized only once, no matter how many times the block in which it is defined is executed)
Constant Expressions... (cont’d)

• **Cannot** contain assignments, increment or decrement operators, function calls, ...
  – see a C reference manual for all the restrictions
  – basically: nothing that has to be evaluated at run-time

```c
static int b = a++ - sum();
```

error
# C Operator Precedence

<table>
<thead>
<tr>
<th>Tokens</th>
<th>Operator</th>
<th>Class</th>
<th>Prec.</th>
<th>Associates</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a[k]</code></td>
<td>subscripting</td>
<td>postfix</td>
<td></td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>f(...)</code></td>
<td>function call</td>
<td>postfix</td>
<td></td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>.</code></td>
<td>direct selection</td>
<td>postfix</td>
<td></td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>indirect selection</td>
<td>postfix</td>
<td></td>
<td>left to right</td>
</tr>
<tr>
<td><code>++ --</code></td>
<td>increment, decrement</td>
<td>postfix</td>
<td></td>
<td>left-to-right</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>size</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>~</code></td>
<td>bit-wise complement</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>!</code></td>
<td>logical NOT</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>- +</code></td>
<td>negation, plus</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>address of</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Indirection (dereference)</td>
<td>unary</td>
<td></td>
<td>right-to-left</td>
</tr>
<tr>
<td>(type)</td>
<td>casts</td>
<td>unary</td>
<td>14</td>
<td>right-to-left</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------</td>
<td>-----------</td>
<td>-----</td>
<td>---------------</td>
</tr>
<tr>
<td>* / %</td>
<td>multiplicative</td>
<td>binary</td>
<td>13</td>
<td>left-to-right</td>
</tr>
<tr>
<td>+ -</td>
<td>additive</td>
<td>binary</td>
<td>12</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left, right shift</td>
<td>binary</td>
<td>11</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>relational</td>
<td>binary</td>
<td>10</td>
<td>left-to-right</td>
</tr>
<tr>
<td>== !=</td>
<td>equality/ineq.</td>
<td>binary</td>
<td>9</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
<td>binary</td>
<td>8</td>
<td>left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>bitwise xor</td>
<td>binary</td>
<td>7</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise or</td>
<td>binary</td>
<td>6</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>binary</td>
<td>5</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
<td>binary</td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td>ternary</td>
<td>3</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+= -=</td>
<td>assignment</td>
<td>binary</td>
<td>2</td>
<td>right-to-left</td>
</tr>
<tr>
<td>*= /= %=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;= ^=</td>
<td>=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>sequential eval.</td>
<td>binary</td>
<td>1</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>
Precedence rules of thumb

- Increment/decrement are ultra sticky
- Unary operators are very sticky
- Math is math, and it’s pretty sticky
- Comparisons are not very sticky
- Bitwise and logic are very unsticky
- Assignment is positively repellant

- Anything else? Not sure? USE PARENTHESES!!!!
  - Parentheses never hurt!!!
Order of Evaluation in Compound Expressions

• Which operator has higher precedence?
• If two operators have equal precedence, are operations evaluated left-to-right or right-to-left?
• Ex:

```
a += b = q - ++ r / s && ! t == u ;
```

what gets executed first, second, ...?

One solution: use parentheses to force a specific order

```
t = (u + v) * w;
```
Order of Evaluation in Compound Expressions

• **Common mistake**: overlooking precedence and associativity (l-to-r or r-to-l)

```c
t = u+v * w;
```

Advice: either...

- force order of evaluation when in doubt by **using parentheses**
- or (even better) write one large expression as sequence of several **smaller expressions**
Evaluating Expressions... (cont’d)

• Instead of...

```c
a+=b=q-++r/(s^!t==u);
```

Or...

```c
a+=(b=(q-((++r)/(s^((!t)==u)))));
```

Better:

```c
tmp1 = s ^ ( (!t) == u );
tmp2 = (++r) / tmp1;
b = q - tmp2;
a += b;
```
Exercise 05a

Operators

1. What does the following output?

```c
int a = 32, b = 5;
printf("%d %d\n", a--, ++b);
printf("%d %d\n", --a, --b);
```

2. What is the value of a after executing the following, and is the condition TRUE or FALSE?

```c
int a = 32, b = 5, c = 8, d = 4, e = 12;
if (a -= ((b > c) || (e / d)) + 6)
  ...do something...
```
Flow of control

• Flow-of-control statements in C
  – if-then-else
  – while and do-while
  – for
  – continue and break
  – switch-case
  – goto
  – conditional operator (?:)

• Same set in java, except for goto
  – Which is bad anyway
    (unless you’re a super kernel hacker, then go nuts)
The C Conditional Operator

• A terse way to write if-then-else statements

\[ c = (a > b) \ ? \ d : e; \]

• This is equivalent to (shorthand for)

```c
if (a > b)
    c = d;
else
    c = e;
```

common source of bugs
complex conditional statements
Combining Assignment and Condition Checking

Why write this...

```
c = getchar();
while (c != \n) {
    ...do something...
    c = getchar();
}
```

does the same thing!

...when you can write this instead?

```
while ((c = getchar()) != \n) {
    ...do something...
}
```
The value of the counter after the loop is exited is valid and can be tested or used.

- C99: you can declare your counter in the for loop

```c
for ( i = 0; i < 10; i++ )
    b *= 2;
printf("b was doubled %d times\n", i);
```

- Some parts of the expression can be missing; default to null statement

```c
for ( ; i < 10; i++ )
    b *= 2;
```

no initialization, i’s value determined before the loop is executed
**break Statement**

- Terminates execution of closest enclosing *for*, *while*, *do*, or *switch* statement

```c
b = 0;
for (i = 0; i < 10; i++) {
    for (j = 0; j < 5; j++) {
        if (a[i][j] > 100)
            break;
        b += a[j];
    }
    printf("b = %d\n", b);
}
```

which loop(s) does this exit?

Unlike *Java*, there is **no labeled break**

See: [http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html](http://download.oracle.com/javase/tutorial/java/nutsandbolts/branch.html) for example of a labeled break in Java.
**continue Statement**

- For bypassing *1* iteration of the innermost loop
  - but **not** exiting the loop altogether

- Example

```c
b = 0;
for ( i = 0; i < 10; i++ ) {
    for ( j = 0; j < 5; j++ ) {
        if (a[i][j] > 100)
            continue;
        b += a[i][j];
    }
    printf("b = %d\n", b);
}
```
The **goto**

- Add symbolic labels (**thisisalabel:**) to arbitrary points in your program
- **goto** `<label>`; transfers control to that point
goto... (cont’d)

• General consensus: avoid using goto’s

```c
label6: ...code here...
  if (something) goto label4;
label3: ...code here...
  if (something) goto label2;
label4: ...code here...
  if (something) goto label3;
label2: ...code here...
  if (something) goto label5;
...
```
goto... (cont’d)

• Common exception: use for global exits (program termination)

```c
for (...) {
    for (...) {
        ... 
        if (disaster)
            goto whoops;
    }
}
... 
whoops:
    /* clean up the mess here and abort execution */
```
Exercise 05b

Control flow

1. What are \(d\) and \(g\) equal to after...

\[
\begin{align*}
    \text{int } d &= 11, \quad g = 12; \\
    \text{int } e &= 13, \quad f = 14; \\
    \text{int } h &= 15; \\
    \text{int } a &= 2, \quad b = 3; \\
    \text{int } x &= 40, \quad y = 30; \\
    \text{if } (a < b) & \quad \{ \quad \\
        d &= e; \\
        \text{if } (x > y) & \quad g = h; \\
    \} \\
    \text{else } & \quad d = f;
\end{align*}
\]

2. Write an equivalent switch statement

\[
\begin{align*}
    \text{unsigned int } a; & \quad \ldots \\
    \text{if } ((a > 1) \&\& (a \leq 3)) & \quad \text{printf(“process now
”);} \\
    \quad \text{else if } (a == 5) & \quad \text{printf(“defer til later
”);} \\
    \quad \text{else if } (a < 7) & \quad ; \\
    \quad \text{else } & \quad \text{printf(“invalid code
”);} \\
\end{align*}
\]

Reminder: Go to course web page for link to exercise form. Paste code into ideone.com and submit the link.