Expressions

Based on slides from K. N. King
Bryn Mawr College
CS246 Programming Paradigm

Operators

• C emphasizes expressions rather than statements.
• Expressions are built from variables, constants, and operators.
• C has a rich collection of operators, including
  o arithmetic operators
  o relational operators
  o logical operators
  o assignment operators
  o increment and decrement operators
  and many others

Arithmetic Operators

• C provides five binary arithmetic operators:
  +  addition
  −  subtraction
  *  multiplication
  /  division
  %  remainder
  
  An operator is binary if it has two operands.
  There are also two unary arithmetic operators:
  +  unary plus
  −  unary minus

Unary Arithmetic Operators

• The unary operators require one operand:
  i = +1;
  j = -i;

  The unary + operator does nothing. It’s used primarily to emphasize that a numeric constant is positive.

Binary Arithmetic Operators

• The value of i % j is the remainder when i is divided by j.
  10 % 3 has the value 1, and 12 % 4 has the value 0.
  Binary arithmetic operators—with the exception of %—allow either integer or floating-point operands, with mixing allowed.
  When int and float operands are mixed, the result has type float.
  9 + 2.5f has the value 11.5, and 6.7f / 2 has the value 3.35.

The / and % Operators

• The / and % operators require special care:
  o When both operands are integers, / “truncates” the result. The value of 1 / 2 is 0, not 0.5.
  o The % operator requires integer operands; if either operand is not an integer, the program won’t compile.
  o Using zero as the right operand of either / or % causes undefined behavior.
Operator Precedence

- The arithmetic operators have the following relative precedence:
  - Highest:  \( +, - \text{ (unary)} \)
  - \( \times, /, \% \)
  - Lowest:  \( +, - \text{ (binary)} \)

- Examples:
  - \( i + j \times k \) is equivalent to \( i + (j \times k) \)
  - \( -i \times -j \) is equivalent to \( (-i) \times (-j) \)
  - \( +i + j / k \) is equivalent to \( (+i) + (j / k) \)

Operator Associativity

- **Associativity** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be left associative if it groups from left to right.
- The binary arithmetic operators (\( \times, /, \% \)) are all left associative, so
  - \( i - j - k \) is equivalent to \( (i - j) - k \)
  - \( i \times j / k \) is equivalent to \( (i \times j) / k \)

Assignment Operators

- **Simple assignment**: used for storing a value into a variable
- **Compound assignment**: used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment \( v = e \) is to evaluate the expression \( e \) and copy its value into \( v \).
- \( e \) can be a constant, a variable, or a more complicated expression:
  - \( i = 5; \quad /* i \text{ is now 5 */} \)
  - \( j = i; \quad /* j \text{ is now 5 */} \)
  - \( k = 10 \times i + j; \quad /* k \text{ is now 55 */} \)

- If \( v \) and \( e \) don’t have the same type, then the value of \( e \) is converted to the type of \( v \) as the assignment takes place:
  - \( int i; \)
  - \( float f; \)
  - \( i = 72.99f; \quad /* i \text{ is now 72 */} \)
  - \( f = 136; \quad /* f \text{ is now 136.0 */} \)
- In C, assignment is an operator, just like +.
- The value of an assignment \( v = e \) is the value of \( v \) after the assignment.
  - The value of \( i = 72.99f \) is 72 (not 72.99).
Side Effects

- An operator that modifies one of its operands is said to have a **side effect**.
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression \( i = 0 \) produces the result 0 and—as a side effect—assigns 0 to \( i \).

Side Effects

- Since assignment is an operator, several assignments can be chained together: \( i = j = k = 0 \);
- The \( = \) operator is right associative, so this assignment is equivalent to \( i = (j = (k = 0)) \);

Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:
  ```
  int i;
  float f;
  f = i = 33.3f;
  i is assigned the value 33, then f is assigned 33.0 (not 33.3).
  ```

Side Effects

- An assignment of the form \( v = e \) is allowed wherever a value of type \( v \) would be permitted:
  ```
  i = 1;
  k = 1 + (j = i);
  printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
  ```
- “Embedded assignments” can make programs hard to read.
- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an **lvalue** as its left operand.
- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.
- Variables are lvalues; expressions such as 10 or \( 2 \times i \) are not.

Lvalues

- Since the assignment operator requires an lvalue as its left operand, it’s illegal to put any other kind of expression on the left side of an assignment expression:
  ```
  12 = i;     /*** WRONG ***/
  i + j = 0;  /*** WRONG ***/
  -i = j;     /*** WRONG ***/
  ```
- The compiler will produce an error message such as "invalid lvalue in assignment."
Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.
- Example:
  \[ i = i + 2; \]
- Using the `+=` compound assignment operator, we simply write:
  \[ i += 2; /* same as i = i + 2; */ \]

There are nine other compound assignment operators, including the following:

- `-=` subtracts the value of `e` from `v`, storing the result in `v`.
- `*=` multiplies `v` by `e`, storing the result in `v`.
- `/=` divides `v` by `e`, storing the result in `v`.
- `%=` computes the remainder when `v` is divided by `e`, storing the result in `v`.

When using the compound assignment operators, be careful not to switch the two characters that make up the operator.

Although `i += j` will compile, it is equivalent to `i = (i + j)`, which merely copies the value of `j` into `i`.

Increment and Decrement

- Two of the most common operations on a variable are “incrementing” (adding 1) and “decrementing” (subtracting 1):
  \[ i = i + 1; \]
  \[ j = j - 1; \]
- Incrementing and decrementing can be done using the compound assignment operators:
  \[ i += 1; \]
  \[ j -= 1; \]
Increment and Decrement

• Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:
  ```
i = 1;  
printf("i is %d\n", ++i); /* prints "i is 2" */  
printf("i is %d\n", i); /* prints "i is 2" */  
```

• Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:
  ```
i = 1;  
printf("i is %d\n", i++); /* prints "i is 1" */  
printf("i is %d\n", i); /* prints "i is 2" */  
```

• The `--` operator has similar properties:
  ```
i = 1;  
printf("i is %d\n", --i); /* prints "i is 0" */  
printf("i is %d\n", i); /* prints "i is 0" */  
```

Increment and Decrement

• When `++` or `--` is used more than once in the same expression, the result can often be hard to understand.

Example:
  ```
i = 1;  
j = 2;  
k = ++i + j++;  
```
The last statement is equivalent to
  ```
i = i + 1;  
k = i + j;  
j = j + 1;  
The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.
```

Expression Evaluation

• Table of operators discussed so far:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Name</th>
<th>Symbol(s)</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>increment (postfix)</td>
<td><code>++</code></td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>decrement (postfix)</td>
<td><code>--</code></td>
<td>left</td>
</tr>
<tr>
<td>2</td>
<td>increment (prefix)</td>
<td><code>++</code></td>
<td>right</td>
</tr>
<tr>
<td></td>
<td>decrement (prefix)</td>
<td><code>--</code></td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>unary plus</td>
<td><code>+</code></td>
<td>right</td>
</tr>
<tr>
<td></td>
<td>unary minus</td>
<td><code>−</code></td>
<td>left</td>
</tr>
<tr>
<td>3</td>
<td>multiplicative</td>
<td><code>*</code>, <code>/</code></td>
<td>left</td>
</tr>
<tr>
<td>4</td>
<td>additive</td>
<td><code>+</code>, <code>−</code></td>
<td>left</td>
</tr>
<tr>
<td>5</td>
<td>assignment</td>
<td><code>=</code>, <code>/=</code>, <code>==</code>, <code>!=</code>, <code>+-</code></td>
<td>right</td>
</tr>
</tbody>
</table>

Expression Evaluation

• The table can be used to add parentheses to an expression that lacks them.

• Starting with the operator with highest precedence, put parentheses around the operator and its operands.

Example:
  ```
a = b + c + d - e / f  
a = b = c++ d = --a / -f  
a = b += (c++) - d + (-a) / (-f)  
a = b += ((c++) - d) + ((-a) / (-f))  
a = b += (((c++) - d) + ((-a) / (-f)))  
```

• In contrast, executing the statements
  ```
i = 1;  
j = 2;  
k = i++ + j++;  
```
will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.
Order of Subexpression Evaluation

• To prevent problems, it’s a good idea to avoid using the assignment operators in subexpressions.
• Instead, use a series of separate assignments:
  
  \[
  \begin{align*}
  a &= 5; \\
  b &= a + 2; \\
  a &= 1; \\
  c &= b - a;
  \end{align*}
  \]

  The value of \( c \) will always be 6.
• Besides the assignment operators, the only operators that modify their operands are increment and decrement.
• When using these operators, be careful that an expression doesn’t depend on a particular order of evaluation.

Example:

\[
\begin{align*}
  i &= 2; \\
  j &= i \times i++; \\
  \end{align*}
\]

It’s natural to assume that \( j \) is assigned 4. However, \( j \) could just as well be assigned 6 instead:
1. The second operand (the original value of \( i \)) is fetched, then \( i \) is incremented.
2. The first operand (the new value of \( i \)) is fetched.
3. The new and old values of \( i \) are multiplied, yielding 6.

Undefined Behavior

• Statements such as \( c = (b - a + 2) - (a - 1); \) and \( j = i \times i++; \) cause **undefined behavior**.
• Possible effects of undefined behavior:
  o The program may behave differently when compiled with different compilers.
  o The program may not compile in the first place.
  o If it compiles it may not run.
  o If it does run, the program may crash, behave erratically, or produce meaningless results.
• Undefined behavior should be avoided.

Expression Statements

• In C, any expression can be used as a statement.
• Example:
  
  \[
  \begin{align*}
  &++i; \\
  \end{align*}
  \]

  \( i \) is first incremented, then the new value of \( i \) is fetched but then discarded.
• Since its value is discarded, there’s little point in using an expression as a statement unless the expression has a side effect:
  
  \[
  \begin{align*}
  &i = 1; \quad /* useful */ \\
  &i--; \quad /* useful */ \\
  &i \times j - 1; \quad /* not useful */
  \end{align*}
  \]