Chapter 7
Expressions and Assignment Statements

**Chapter 7 Topics**

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment
Chapter 7
Expressions and Assignment Statements

Introduction
- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence of imperative languages is dominant role of assignment statements.

Arithmetic Expressions
- Their evaluation was one of the motivations for the development of the first programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in math.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.
- The operators can be unary, or binary. C-based languages include a ternary operator, which has three operands (conditional expression).
- The purpose of an arithmetic expression is to specify an arithmetic computation.
- An implementation of such a computation must cause two actions:
  - Fetching the operands from memory
  - Executing the arithmetic operations on those operands.
- Design issues for arithmetic expressions:
  1. What are the operator precedence rules?
  2. What are the operator associativity rules?
  3. What is the order of operand evaluation?
  4. Are there restrictions on operand evaluation side effects?
  5. Does the language allow user-defined operator overloading?
  6. What mode mixing is allowed in expressions?

Operator Evaluation Order
1. Precedence
   - The operator precedence rules for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated (“adjacent” means they are separated by at most one operand).
   - Typical precedence levels:
     1. parentheses
     2. unary operators
     3. ** (if the language supports it)
     4. *, /
5. +, -
- Many languages also include unary versions of addition and subtraction.
- Unary addition (+) is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
- In Java, unary plus actually does have an effect when its operand is short or byte. An implicit conversion of short and byte operands to int type takes place.
- Unary minus operator (-) Ex:

  ```
  A + (- B) * C // is legal
  A + - B * C // is illegal
  ```

2. Associativity
- The operator associativity rules for expression evaluation define the order in which adjacent operators with the **same precedence** level are evaluated. An operator can be either left or right associative.
- Typical associativity rules:
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., FORTRAN)
- Ex: (Java)

  ```
  a - b + c // left to right
  ```

- Ex: (Fortran)

  ```
  A ** B ** C // right to left
  (A ** B) ** C // In Ada it must be parenthesized
  ```

<table>
<thead>
<tr>
<th>Language</th>
<th>Associativity Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN</td>
<td>Left: * / + -</td>
</tr>
<tr>
<td></td>
<td>Right: **</td>
</tr>
<tr>
<td>C-BASED LANGUAGES</td>
<td>Left: * / % binary + binary -</td>
</tr>
<tr>
<td></td>
<td>Right: ++ -- unary – unary +</td>
</tr>
<tr>
<td>ADA</td>
<td>Left: all except **</td>
</tr>
<tr>
<td></td>
<td>Non-associative: **</td>
</tr>
</tbody>
</table>

- **APL** is different; all operators have **equal** precedence and all operators associate **right to left**.
- Ex:

  ```
  A X B + C // A = 3, B = 4, C = 5 \rightarrow 27
  ```

- Precedence and associativity rules can be **overridden with parentheses**.
3. Parentheses
- Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
- A parenthesized part of an expression has precedence over its adjacent un-parenthesized parts.
- Ex:

  \[(A + B) \times C\]

4. Conditional Expressions
- Sometimes if-then-else statements are used to perform a conditional expression assignment.

  ```
  if (count == 0)
    average = 0;
  else
    average = sum / count;
  ```

- In the C-based languages, this can be specified more conveniently in an assignment statement using a conditional expressions. Note that ? is used in conditional expression as a ternary operator (3 operands).

  ```
  expression_1 \, ? \, expression_2 \, : \, expression_3
  ```

- Ex:

  ```
  average = (count == 0) \, ? \, 0 \, : \, sum / count;
  ```

Operand evaluation order
- The process:
  1. Variables: just fetch the value from memory.
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction.
  3. Parenthesized expressions: evaluate all operands and operators first.

- Side Effects
  - A side effect of a function, called a functional side effect, occurs when the function changes either one of its parameters or a global variable.
  - Ex:

    ```
    a + fun(a)
    ```
If fun does not have the side effect of changing \( a \), then the order of evaluation of the two operands, \( a \) and \( \text{fun}(a) \), has no effect on the value of the expression.

However, if \( \text{fun} \) changes \( a \), there is an effect.

Ex:
Consider the following situation: \( \text{fun} \) returns the value of its argument divided by 2 and changes its parameter to have the value 20, and:

\[
\begin{align*}
\text{a} &= 10; \\
\text{b} &= \text{a} + \text{fun}(\text{a});
\end{align*}
\]

If the value of \( a \) is returned first (in the expression evaluation process), its value is 10 and the value of the expression is 15.

But if the second is evaluated first, then the value of the first operand is 20 and the value of the expression is 25.

The following shows a C program which illustrate the same problem.

```c
int a = 5;
int fun1() {
    a = 17;
    return 3;
}
void fun2() {
    a = a + fun1(); // C language a = 20; Java a = 8
}
void main() {
    fun2();
}
```

The value computed for \( a \) in \( \text{fun2} \) depends on the order of evaluation of the operands in the expression \( a + \text{fun1()} \). The value of \( a \) will be either 8 or 20.

Two possible solutions:
1. Write the language definition to disallow functional side effects
   - No two-way parameters in functions.
   - No non-local references in functions.
   - **Advantage:** it works!
   - **Disadvantage:** Programmers want the flexibility of two-way parameters (what about C?) and non-local references.
2. Write the language definition to demand that operand evaluation order be fixed
   - **Disadvantage:** limits some compiler optimizations

Java guarantees that operands are evaluated in **left-to-right order**, eliminating this problem. // C language a = 20; Java a = 8
Overloaded Operators

- The use of an operator for **more than one purpose** is operator overloading.
- Some are common (e.g., + for int and float).
- Java uses + for addition and for **string catenation**.
- Some are potential trouble (e.g., & in C and C++)

```
x = &y // as binary operator bitwise logical
     // AND, as unary it is the address of y
```

- Causes the address of y to be placed in x.
- Some loss of readability to use the same symbol for two completely unrelated operations.
- The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler “difficult to diagnose”.
- Can be avoided by introduction of new symbols (e.g., Pascal’s **div for integer division and / for floating point division**)
Type Conversions

- A **narrowing conversion** is one that converts an object to a type that cannot include all of the values of the original type e.g., `double` to `float`.
- A **widening conversion** is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`.

Coercion in Expressions

- A **mixed-mode expression** is one that has operands of different types.
- A **coercion** is an implicit type conversion.
- The disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- Language are not in agreement on the issue of coercions in arithmetic expressions.
- Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they eliminate the benefits of type checking.
- Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions.
- The issue is whether programmers should be concerned with this category of errors or whether the compiler should detect them.
- Java method Ex:

```java
void mymethod() {
   int a, b, c;
   float d;
   ...
   a = b * d;
   ...
}
```

- Assume that the second operand was supposed to be `c` instead of `d`.
- Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, `b` will be coerced to `float`.
Explicit Type Conversions

- Often called casts in C-based languages.
- Ex: Ada:

  \( \text{FLOAT(INDEX)} \rightarrow \text{INDEX is INTEGER type} \)

Java:

  \((\text{int})\text{speed} \quad /\text{speed is float type}/ \)

Errors in Expressions

- Caused by:
  - Inherent limitations of arithmetic e.g. division by zero
  - Limitations of computer arithmetic e.g. overflow or underflow
- Floating-point overflow and underflow, and division by zero are examples of run-time errors, which are sometimes called exceptions.
Relational and Boolean Expressions

- A relational operator: an operator that compares the values of its two operands.
- Relational Expressions: two operands and one relational operator.
- The value of a relational expression is Boolean, unless it is not a type included in the language.
  - Use relational operators and operands of various types.
  - Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)
- The syntax of the relational operators available in some common languages is as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ada</th>
<th>C-Based Languages</th>
<th>Fortran 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>=</td>
<td>==</td>
<td>.EQ. or ==</td>
</tr>
<tr>
<td>Not Equal</td>
<td>/=</td>
<td>!=</td>
<td>.NE. or &lt;&gt;</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>.GT. or &gt;</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>.LT. or &lt;</td>
</tr>
<tr>
<td>Greater than or equal</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>.GE. or &gt;=</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td>.LE. or &gt;=</td>
</tr>
</tbody>
</table>

Boolean Expressions

- Operands are Boolean and the result is Boolean.

<table>
<thead>
<tr>
<th>FORTRAN 77</th>
<th>FORTRAN 90</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>and</td>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>.OR.</td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NOT.</td>
<td>not</td>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

- Versions of C prior to C99 have no Boolean type; it uses int type with 0 for false and nonzero for true.
- One odd characteristic of C’s expressions: \( a < b < c \) is a legal expression, but the result is not what you might expect.
- The left most operator is evaluated first because the relational operators of C, are left associative, producing either 0 or 1.
- Then this result is compared with var c. There is never a comparison between b and c.
Short Circuit Evaluation

- A short-circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators.
  
  Ex:

  \[(13 \times a) \times (b/13 - 1) // is independent of the value (b/13 - 1) if a = 0, because 0\times x = 0.\]

  So when a = 0, there is no need to evaluate \((b/13 - 1)\) or perform the second multiplication.
  
  However, this shortcut is not easily detected during execution, so it is never taken.

  The value of the Boolean expression:

  \[(a \geq 0) && (b < 10) // is independent of the second expression if a < 0, because(F && x) is False for all the values of x.\]

  So when a < 0, there is no need to evaluate b, the constant 10, the second relational expression, or the && operation.

  Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.

  Short-circuit evaluation exposes the potential problem of side effects in expressions

  \[(a > b) || (b++ / 3) // b is changed only when a <= b.\]

  If the programmer assumed b would change every time this expression is evaluated during execution, the program will fail.

  C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (&& and ||), but also provide bitwise Boolean operators that are not short circuit (& and |)
Assignment Statements

Simple Assignments
- The C-based languages use == as the equality relational operator to avoid confusion with their assignment operator.
- The operator symbol for assignment:
  1. = FORTRAN, BASIC, PL/I, C, C++, Java
  2. := ALGOL, Pascal, Ada

Conditional Targets
- Ex:

  ```
  flag ? count 1 : count2 = 0; ⇔ if (flag)
      count1 = 0;
  else
      count2 = 0;
  ```

Compound Assignment Operators
- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment.
- The form of assignment that can be abbreviated with this technique has the destination var also appearing as the first operand in the expression on the right side, as in

  ```
  a = a + b
  ```

- The syntax of assignment operators that is the catenation of the desired binary operator to the = operator.

  ```
  sum += value; ⇔ sum = sum + value;
  ```

Unary Assignment Operators
- C-based languages include two special unary operators that are actually abbreviated assignments.
- They combine increment and decrement operations with assignments.
- The operators ++ and -- can be used either in expression or to form stand-alone single-operator assignment statements. They can appear as prefix operators:

  ```
  sum = ++ count; ⇔ count = count + 1; sum = count;
  ```
If the same operator is used as a postfix operator:

\[
\text{sum} = \text{count}++; \quad \Leftrightarrow \quad \text{sum} = \text{count}; \ \text{count} = \text{count} + 1;
\]

**Assignment as an Expression**

- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- Ex:

  ```c
  while ((ch = getchar())!=EOF)
  {
  \}  // why ( ) around assignment?
  ```

- The assignment statement must be parenthesized because the precedence of the assignment operator is lower than that of the relational operators.
- Disadvantage: Another kind of expression side effect which leads to expressions that are difficult to read and understand. For example

  \[
  a = b + (c = d / b++) - 1
  \]

  denotes the instructions
  - Assign b to temp
  - Assign b + 1 to b
  - Assign d / temp to c
  - Assign b + c to temp
  - Assign temp − 1 to a

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors.

```c
if (x = y) ...
```instead of

```c
if (x == y) ...
```
**Mixed-Mode Assignment**

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded.)
- In **Java**, only **widening** assignment coercions are done.
- In **Ada**, there is no assignment coercion.
- In all languages that allow mixed-mode assignment, the coercion takes place **only after** the right side expression has been evaluated. For example, consider the following code:

```c
int a, b;
float c;
...
c = a / b;
```

Because `c` is float, the values of `a` and `b` could be coerced to float before the division, which could produce a different value for `c` than if the coercion were delayed (for example, if `a` were 2 and `b` were 3).