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
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: Design Issues

• Arithmetic evaluation was one of the motivations for development of the first programming languages
• Arithmetic expressions consist of operators, operands, parentheses, and function calls
• Design issues for arithmetic expressions
  – operator precedence rules
  – operator associativity rules
  – order of operand evaluation
  – operand evaluation side effects
  – operator overloading
  – mode mixing expressions
Arithmetic Expressions: Operator Precedence Rules

• Operators in arithmetic expressions
  – A unary operator has one operand
  – A binary operator has two operands
  – A ternary operator has three operands

• Operator precedence rules
  – The rules for expression evaluation. They define the order in which “adjacent” operators of different precedence levels are evaluated
    • E.g., $3 + 4 \times 5$
  – Typical precedence levels
    • parentheses
    • unary operators
    • ** (exponentiation, if the language supports it)
    • *, /
    • +, -
Arithmetic Expressions: Operator Associativity Rule

• **Operator associativity rules**
  – The rules for expression evaluation. They define the order in which adjacent operators with *the same precedence* level are evaluated
  – Typical associativity rules
    • Left to right, except **, which is right to left
      – E.g., a – b + c
    • Sometimes unary operators associate right to left (e.g., in FORTRAN)
  – APL is different; all operators have equal precedence and all operators associate right to left (an expression must be parenthesized to show the desired order).
  – Precedence and associativity rules can be overridden with parentheses
Arithmetic Expressions: Conditional Expressions

• Conditional Expressions
  – C-based languages (e.g., C, C++)
  – An example:
    • ternary operator ?:
    • Expression_1 ? Expression_2 : Expression_3
    • E.g.,
      \[
      \text{average} = (\text{count} == 0)\ ? \ 0 \ : \ \text{sum} / \ \text{count}
      \]
    • Evaluates as if written like
      \[
      \text{if (count} == 0) \ \text{average} = 0 \\
      \text{else average} = \text{sum} / \text{count}
      \]
Arithmetic Expressions: Operand Evaluation Order

- **Operand evaluation order**
  1. Variables: fetch the value from memory
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
  3. Parenthesized expressions:
     - All operators the parenthesis contains must be evaluated before its value can be used as an operand.
     - Operand evaluation order is important when the evaluation of an operand have *side effects*. 
Arithmetic Expressions: Potentials for Side Effects

• **Functional side effects**
  – when a function changes one of its parameters (two-way parameter) or a global variable
  – Problem with functional side effects:
    • When a function referenced in an expression alters another operand of the expression;
    • e.g., for a parameter change:
      
      ```
      a = 10;
      /* assume fun returns the value of its argument divided by 2 and changes its parameter to 20*/
      b = a + fun(a);
      
      – If value of a is fetched first, b is 15.
      – If fun(a) is evaluated first, b is 25.
      ```
Functional Side Effects

• Two possible solutions to the problem

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:** inflexibility of two-way parameters and non-local references

2. Write the language definition to demand that operand evaluation order be fixed
   • **Disadvantage:** limits some compiler optimizations which need reordering operand evaluation.
Overloaded Operators

• **Operator overloading**
  – Use of an operator for more than one purpose
  – Some are common (e.g., + for `int` and `float`)
  – Some are potential trouble
    • e.g., `&` in C and C++:
      – As a binary operator: bitwise logical AND operation
      – As a unary operator: address-of operator
    • Loss of compiler error detection
      – omission of an operand (keying error) for a bitwise AND operation will be undetected by the compiler.
    • Some loss of readability
    • Can be avoided by introduction of new symbols
      – e.g., Pascal’s `div` for integer division, `/` for float division
Overloaded Operators (continued)

• C++ and Ada allow user-defined overloaded operators
• Potential problems:
  – Users can define nonsense operations
  – Readability may suffer, even when the operators make sense
    • The reader must find both the types of the operands and the definition of the operator to determine its meaning. These definition could be in other files.
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
  – Dangerous because the converted type cannot store even approximations of all of the values of original type.

• 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 original type
  – e.g., int to float
  – Usually it is safe, but still can result in reduced accuracy (float use some space to store exponential part).
Type Conversions: Mixed Mode

• A **mixed-mode expression** is one that has operands of different types

• A **coercion** is an implicit type conversion
  
  — Disadvantage of coercions:
    
    • decrease in the type error detection for compiler (keying error might not be detected)
  
  — In most languages, all numeric types are coerced in expressions, using widening conversions
  
    • In java, **byte** and **short** are integer types that are smaller than **int**. They are coerced to **int** first before be manipulated.
  
    • E.g.,  
      
      ```java
      byte a, b, c;
      a = b + c;
      ```
      
      b and c are coerced to **int** and an **int** addition is used. The sum is then converted to byte and put in a.
Explicit Type Conversions

• Explicit Type Conversions
• Called *casting* in C-based language
• Examples
  – C: (int) angle
  – Ada: Float (sum)

*Note that Ada’s syntax is similar to function calls*
Relational and Boolean Expressions

• Relational Expressions
  – Use relational operators and operands of various types
  – Evaluate to some Boolean representation
  – Operator symbols used vary somewhat among languages (>, <, !=, /=, .NE., <>, #)
  – E.g.,
    a + 1 > 2 * b
  – Relational operators always have lower precedence than arithmetic operators. So, a+1 and 2*\( b \) are evaluated first than operator \( > \).
Relational and Boolean Expressions

- **Boolean Expressions**
  - Operands are Boolean and the result is Boolean
  - Example operators

<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>

- C has no Boolean type—it uses *int* type with 0 for false and nonzero for true
- One odd expression in C: a < b < c is a legal expression, but the result is not what you might expect:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., c)
Short Circuit Evaluation

• An expression in which the result is determined without evaluating all of the operands and/or operators
  – Example 1: \((13*a) * (b/13-1)\)
    If \(a\) is zero, there is no need to evaluate \((b/13-1)\)
  – Example 2: \((a >= 0) && (b < 0)\)
    If \(a < 0\), no need to evaluate if \(b < 0\) or not

• Problem with non-short-circuit evaluation
  
  ```
  index = 0;
  while (index < length) && (LIST[index] != value)
    index++;
  ```
  
  – When `index=length`, `LIST [index]` will cause an indexing problem
    (assuming `LIST has length -1 elements`)
Short Circuit Evaluation (continued)

• Short-circuit evaluation exposes the potential problem of side effects in expressions
  e.g. \((a > b) \ |\ | (b++ / 3)\)
  – With short circuit, \(b++/3\) is evaluated only when \(a \leq b\).

• C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (\&\& and | |),
  – But bitwise Boolean operators & and | are not short circuit

• Ada: programmer can specify either short-circuit evaluation or not
Assignment Statements

• The general syntax
  `<target_var>  <assign_operator>  <expression>`

• The assignment operator
  `=  FORTRAN, BASIC, PL/I, C, C++, Java (and use `==`
     for equality relational operator)`
  `:=  ALGOLs, Pascal, Ada (and use `=` for equality
       relational operator)`

• = can be bad when it is overloaded for the
  relational operator for equality
Assignment Statements: Conditional Targets

• Conditional targets (C, C++, and Java)

\[(\text{flag})? \text{ total : subtotal} = 0\]

Which is equivalent to

\[
\begin{align*}
\text{if (flag)} \\
\quad \text{total} & = 0 \\
\text{else} \\
\quad \text{subtotal} & = 0
\end{align*}
\]
Assignment Statements: Compound Operators

• A shorthand method of specifying a commonly needed form of assignment
• Introduced in ALGOL; adopted by C
• Example

\[ a = a + b \]

is written as

\[ a += b \]
Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment

- Examples
  
  \[
  \text{sum} = \text{++count} \quad (\text{count incremented, assigned to sum})
  \]
  
  \[
  \text{sum} = \text{count++} \quad (\text{assigned to sum, count incremented})
  \]
  
  \[
  \text{count++} \quad (\text{count incremented})
  \]
  
  \[
  -\text{count++} \quad (\text{count incremented then negated})
  \]
Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands.
- An example:

```c
while ((ch = getchar()) != EOF) {...}
```

The `ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the `while` statement.
Mixed-Mode Assignment

• Assignment statements can also be mixed-mode, for example

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

• In Pascal, integer variables can be assigned to real variables, but real variables cannot be assigned to integers

• In Java, only widening assignment coercion are done

• In Ada, there is no assignment coercion
Summary

• Expressions
• Operator precedence and associativity
• Operator overloading
• Mixed-type expressions
• Various forms of assignment