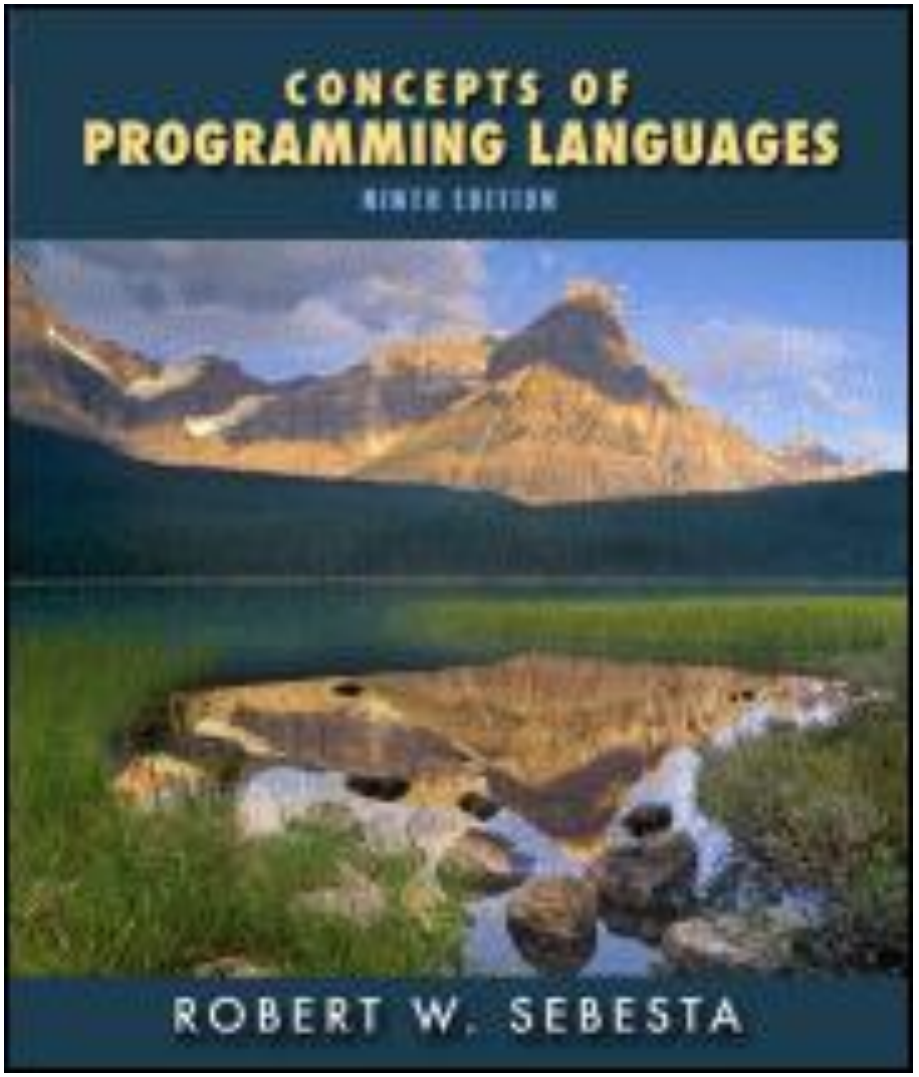
Chapter 7

Expressions and Assignment Statements



Ch07 – Expressions and Assignment Statements

7.1 Introduction*

7.2 Arithmetic Expressions

7.3 Overloaded Operators*

7.4 Type Conversions*

7.5 Relational and Boolean Expressions*

7.6 Short-Circuit Evaluation*

7.7 Assignment statements*

7.8 Mixed-mode Assignment*

7.2 Arithmetic Expressions

- Side effects (7.2.2)
  - Assignment and I/O expressions yield values and side effects

	value	Side effect
x = 7	7	Store 7 in x's location
printf("Snoopy")	6	Snoopy is displayed

  - In a boarder sense, assignment and I/O statements also yield side effects, e.g.

    int f() { static int x = 0; x++; return x; }

    The call f() yields a side effect, due to the assignment statement x++.
    Such a side effect is called a functional side effect.

7.2 Arithmetic Expressions

- Referential transparency and opaque
  - Referential transparency

    Contexts do not affect the meanings of expressions
  - Referential opaque

    Contexts do affect the meanings of expressions
  - Side effects make a language referential opaque.
  - For example, f() yields different values in different places

    cout << f();

    cout << f();

    cout << f();

    In a referential opaque language, exp==exp may not be true, e.g f()==f()

7.2 Arithmetic Expressions

- Con for side effects
  - Hard to read
  - Hard to prove – recall { x=5 } y = x++ { x=6 }
  - Prohibit optimization

    exp+exp cannot be optimized to 2*exp, if exp yields side effects.
  - Make parallel processing difficult

    Suppose exp1 reads from x and exp2 writes to x, what will be the value of exp1+exp2, if exp1 and exp2 are evaluated in parallel?
  - Affect portability

    due to side effects + unspecified operand evaluation order

7.2 Arithmetic Expressions

- Argument (operand) evaluation order
  - exp1+exp2*exp3

    Every language specifies operator evaluation order, e.g. * has a higher precedence than +.

    But, what is the evaluation order of the three operands exp1, exp2, and exp3?
  - f(exp1,exp2)

    What is the evaluation order of the two arguments exp1 and exp2?
  - Does a language have to specified argument (or operand) evaluation order?

7.2 Arithmetic Expressions

- Side effects and argument evaluation order
  - Without side effects
    - Argument evaluation order is immaterial
    - Purely functional languages won't specify argument evaluation order, e.g. Haskell.
  - With side effects
    - Argument evaluation order is important
    - Some specify argument evaluation order, e.g. Java l-2-r
    - Most languages don't, e.g. C, C++, Scheme
      (N.B. In C/C++, ?:, &&, ||, and , have required order.)
- Pro for specifying argument evaluation order
  - Increase portability

7.2 Arithmetic Expressions

- Pro for not specifying argument evaluation order
  - Increase efficiency – leave a room for the compiler writer
    - Optimization
      With required order, e+e cannot be optimized to 2*e
    - Parallel processing
      With required order, exp1+exp2 cannot be evaluated in parallel
    - Intelligent compilation
      With required order, 5/n must be evaluated in
      n = 0; … n*(5/n) …
      But, a compiler may choose not to evaluate 5/n at all, for the reason that 0×anything=0

7.2 Arithmetic Expressions

- Example

  Given

  ```
  int x=3;
  int f(int y) { x++; return x+y; }
  ```

  What might be the value of the expression x*f(x)+f(x)?

Operand evaluation order	x*f(x)+f(x)
1 2 3	3*7+9=30
1 3 2	3*9+7=34
2 1 3	4*7+9=37
2 3 1	4*9+7=43
3 1 2	5*7+9=44
3 2 1	5*9+7=52

7.2 Arithmetic Expressions

- Example (cont'd)

  Instead of fetching the value of x from memory thrice, some compilers generate code to fetch the value of x once and store it in a register. That is, the compiled code reads as

  ```
  reg = x;
  reg*f(reg)+f(reg)
  ```

  Since the value 3 stored in the register doesn't change, the compiled code has the same effect as

  ```
  3*f(3)+f(3)
  ```

  Finally, depending on the order of evaluating the two calls, we have

7.2 Arithmetic Expressions

- Example (cont'd)

Operand evaluation order	3*f(3)+f(3)
1 2	3*7+8=29
2 1	3*8+7=31

Lesson

Never write code that depends on argument evaluation order.