Ch. 6: Expressions & Assignment

I. Introduction
- Expressions are the building blocks for various parts of statements
- Assignment statement changes the value of a variable, which is the conceptual basis for imperative languages.

II. Arithmetic Expressions
- models mathematical expressions.
- unary & binary operators mostly

A. Operator Evaluation Order
1. Precedence
   a. e.g. consider A + B * C. Order of evaluation of subexpressions changes result.
   b. unary operators may need to be parenthesized: e.g. A + - B * C is: A + ( - B) * C
   c. [Chart] on operator precedence in various languages
2. Associativity: usually left-to-right (left-associative)
   a. Given the expression: A + B - C will it be done right-to-left or left-to-right? This may not
      seem like an issue at this point, but when we see that sub expressions can have side-
      effects then it becomes important, e.g. A + (B++) + B
   b. exponentiation is right-associative, e.g. A ** B ** C
   c. [Chart] shows associativity rules for a few languages
   d. APL is always right-associative (doesn’t follow mathematical conventions)
   e. Compiler may reorder order of evaluation in some instances
      E.g. A + B - C where if addition is done first we get overflow, but if subtraction is done
      first we don’t
3. Parenthesis
   a. Used to alter precedence rules, e.g. (A + B) * C
   b. APL expressions must be fully parenthesized

B. Operand Evaluation Order
1. If there are no side effects, then operand evaluation order within same class (e.g. +) doesn’t mat-
   ter.
2. E.g. where it does matter:
   ```c
   int a = 5;
   int fcn1() {
       a = 17;
       return 3;
   } // End fcn1
   void fcn2() {
       a = a + fcn1(); // is 'a' fetched first, or is fcn1 called?
   } // End fcn2
   void main() {
       fcn2();
   }
   ```
   Due to compiler optimization, evaluation order is implementation-dependent (incompatibilities...)

C. Conditional Expressions
An assignment statement can be implemented in C using a conditional expression as follows:
```c
average = (count == 0) ? 0 : sum / count;
```
III. Overloaded Operators
- Arithmetic Operators can be used for more than one purpose, e.g. + used for int & real
  A. E.g. "&" in C
    - As a binary operator (a & b) it specifies bitwise logical AND.
    - As a unary operator &b it specifies the address of b
    - (In C++ it is also used to specify that a parameter is a constant pointer)
  Distinct symbols increase readability
B. Is "-" binary or unary? Can be hard to tell
C. Real vs. Integer Division
  1. E.g. in FORTRAN, given AVG as FLOAT, SUM & COUNT as INTEGER:
     \[\text{AVG} = \frac{\text{SUM}}{\text{COUNT}}\]
     The answer is incorrect, as the integer division is truncated before being converted to FLOAT for
     storage in AVG
  2. PASCAL uses two different symbols for this reason: DIV and "/"
D. "Incorrect" overloading
  In languages that allow user-specified overloading (which enhances readability), "+" could be
  overloaded to actually do multiplication in some contexts. (C++, Ada, FORTRAN 90)

IV. Type Conversions
- Narrowing conversion: converts an object to a type that cannot include all the original values
- Widening (promoting) conversions: converts an object to a type that includes approximations of all
  the original values
A. Widening is usually safe, but not always. Exception:
  VAX stores int as 32 bits (9 decimal digits of precision), but also stores float in 32-bits (only 7
decimal digits of precision), so converting a large int to a float can mean the loss of two digits
  of precision (the least significant two digits)
B. Mixed-mode expressions
  coercion: implicit conversions
  cast: explicit conversions
  1. [FORTRAN coercians]
  2. Original C: promotes combination of float and int to float. float and short int are always promoted
to double and int, respectively.
  3. Coercion prevents type-checking, e.g.
     void main() {
       int a, b, c;
       float d;
       ...
       a = b * d; //"d" mistakenly typed rather than "c". No error found.
       ...
     }
  4. PL/I: tries to intelligently coerce a character string into a numeric value when used in an expre-
sion with an integer. If a decimal point is found, it is assumed to be a float.
C. Explicit Type Conversion
  1. In FORTRAN: AVG := FLOAT(SUM) / FLOAT(COUNT)
  2. In C: (int) angle; The parenthesis needed since some types are two words (e.g. short int)
V. Relational and Boolean Expressions

A. Relational Expressions:
   1. Definitions:
      a. A relational operator compares the values of its two operands.
      b. A relational expression has two operands and one relational operator, resulting in a Boolean
   2. [Chart] shows common relational operators.
   3. These have lower precedence than arithmetic operators, so * and + are done first in:
      \[ a + 1 > 2 * b \]
      When parenthesized is:
      \[ (a + 1) > (2 * b) \]

B. Boolean Expressions
   1. A boolean expression consists of boolean variables (e.g. x, y), boolean constants (e.g. T, F),
      relational expressions (e.g. >, <), and boolean operators (AND, OR, NOT, XOR).
   2. There is often the precedence of: 1. NOT, 2. AND, OR & XOR
   3. We must order precedence of relational, arithmetic, and boolean expressions
   [Examples in FORTRAN 77, Ada, C]
      a. Note that Ada’s boolean operators are on same level, requiring parenthesis to evaluate:
      \[ a > b \text{ and } a < c \text{ or } d = 0 \]
      which could become:
      \[ (a > b \text{ and } a < c) \text{ or } d = 0 \]
      b. C uses integers to represent boolean values, allowing expressions like:
      \[ a > b > c \]
      c. In Pascal the boolean operators have higher precedence than relational operators, so
      \[ a > 5 \text{ or } a < 0 \]
      is illegal and must be written as:
      \[ (a < 5) \text{ or } (a < 0) \]

VI. Short-Circuit evaluation

A. Example
   \[ (13 * a) * (b/13 - 1) \]
   Need not be evaluated any further than "13 * a" if a is 0. Similarly for
   A and B
   when A is false.

B. Example from Pascal, which does not have short-circuit evaluation:
   \[ \text{index} := 1; \]
   \[ \text{WHILE (index <= listLength) AND (list[index] <> key) DO} \]
   \[ \text{index := index + 1;} \]
   If key is not found, we get subscript out-of-range error. (Turbo Pascal does have short circuit, as does C)

C. On the other hand, allowing short-circuit evaluation has problem of side effects in expressions
   \[ (a > b) || (b++ / 3) \]
   // b not incremented if short-circuited

D. Ada: allows specifying short-circuit evaluation or not
   "and then", "or else" specify short circuit evaluation, otherwise not. E.g.
   \[ \text{INDEX := 1;} \]
   \[ \text{while (INDEX <= LISTLEN) and then (LIST (INDEX) /= KEY) } \]
   \[ \text{loop} \]
   \[ \text{INDEX := INDEX + 1;} \]
   \[ \text{end loop;} \]
VII. Assignment Statements

General syntax:
\[
<\text{target}> \ <\text{assignment\_operator}> \ <\text{expression}>
\]

A. Simple Assignments
1. PL/I:
   \[
   A = B = C
   \]
   sets A to the result of the relational expression \(B = C\)
2. ALGOL 60: used ";=" rather than "="

B. Multiple Targets
PL/I: \(\text{SUM, TOTAL} = 0\)
C: \(a = b = c = 0;\)

C. Conditional Targets in C:
\[
\text{flag} \ ? \ \text{count1} : \ \text{count2} = 0
\]
which is equivalent to
\[
\text{if flag then count1 = 0 else count2 = 0;}
\]

D. Compound Assignment Operators: shorthand for common assignments. Example:
\[
\text{sum} = \text{sum} + \text{a};
\]
which can be written in C as:
\[
\text{sum} += 2;
\]

E. Unary Assignment Operators in C & C++
1. increment (++) and decrement (--) operators can be combined with assignment
2. We can rewrite
   \[
   \text{count} = \text{count} + 1;
   \]
   \[
   \text{sum} = \text{count};
   \]
   as
   \[
   \text{sum} = ++\text{count};
   \]
3. We can rewrite
   \[
   \text{sum} = \text{count};
   \]
   \[
   \text{count} = \text{count} + 1;
   \]
   as
   \[
   \text{sum} = \text{count}++;
   \]
4. We can simply have: \(\text{count}++;\)

F. Assignment as an Expression
1. Since C assignments return a value, we could have:
   \[
   \text{while } ((\text{ch} = \text{getchar()} ) \neq \text{EOF}) \{ ...\}
   \]
2. The disadvantage of this is the confusion that can develop from:
   \[
   \text{a} = \text{b} + (\text{c} = \text{d} / \text{b}++) - 1
   \]
   where the first b is the "original" b