CSE 452: Programming Languages

Expressions and Control Flow

Outline of Today's Lecture

- Expressions and Assignment Statements
  - Arithmetic Expressions
  - Overloaded Operators
  - Type Conversions
  - Relational and Boolean Expressions
    - Short-circuit evaluation
  - Assignment Statements
    - Mixed mode assignment

Expressions

- Expressions are the fundamental means of specifying computations in a programming language
- Types:
  - Arithmetic
  - Relational/Boolean

Arithmetic Expressions

- Consist of operators, operands, parentheses, and function calls
- Design issues for arithmetic expressions:
  - What are the operator precedence rules?
  - What are the operator associativity rules?
  - What is the order of operand evaluation?
  - Are there restrictions on operand evaluation side effects?
  - Does the language allow user-defined operator overloading?
  - What mode mixing is allowed in expressions?

Arithmetic Expressions

- Types of operators
  - A unary operator has one operand:

    - x
  - A binary operator has two operands:

    x + y
    - Infix: operator appears between two operands
    - Prefix: operator precede their operands
  - A ternary operator has three operands:

    (x > 10)? 0 : 1
- Evaluation Order
  - Operator evaluation order
  - Operand evaluation order

Operator Evaluation Order

- Four rules to specify order of evaluation for operators
  1. Operator precedence rules
     - Define the order in which the operators of different precedence levels are evaluated (e.g., * vs.+)
  2. Operator associativity rules
     - Define the order in which adjacent operators with the same precedence level are evaluated (e.g., left/right associative)
  3. Parentheses
     - Precedence and associativityrules can be overriden with parentheses
  4. Conditional Expressions (?: operator in C/C++/Perl)
     - Equivalent to if-then-else statement

Operand Evaluation Order

- When do we evaluate operand?
  - Variables are evaluated by fetching their values from memory
  - Constants
    - Sometimes, constants are evaluated by fetching its value from memory;
    - At other times, it is part of the machine language instruction
  - Parenthesized expressions
    - If operand is a parenthesized expression, all operators it contains must be evaluated before its value can be used as an operand
  - Function calls
    - Must be evaluated before its value can be used as an operand

Operand Evaluation Order

- Functional Side Effects
  - When function changes one of its params/global variable

    `a + fun(a)`
    - If `fun` does not have the side effect of changing `a`, then the order evaluation of the two operands, `a` and `fun(a)` , does not matter
    - If fun does have the side effect of changing `a`, order of evaluation matters
  - Two Possible Solutions :
    - Disallow functional side effects in the language definition
      - No two-way parameters in functions
      - No non-local references in functions
      - Advantage: it works!
      - Disadvantage: No more flexibility
    - Write language definition to demand fixed operand evaluation order
      - Disadvantage: limits some compiler optimizations

Overloaded Operators

- Multiple use of an operator
  - E.g., use + for integer addition and floating-point addition
- Some drawbacks of operator overloading
  - May affect readability
    - E.g., the ampersand (&) operator in C is used to specify
      - Bitwise logical AND operation
      - Address of a variable
  - May affect reliability
    - Program does not behave the way we want
    - `int x, y; float z;  z = x / y`
    - Problem can be avoided by introducing new symbols (e.g., Pascal's div for integer division and / for floating point division)
- C++ and Ada allow user-defined overloaded operators
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense
      - E.g. use + to mean multiplication