Chapter7 Expression and Assignment Statement

- Arithmetic Expressions
- Overloaded Operators
- Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- Fundamental means of specifying computations in a programming language
- Arithmetic expressions consist of operators, operands, parentheses, and function calls

Arithmetic Expressions: Operators

- A unary operator has one operand

  ! expression
  ** expression*
  ++ expression
  -- expression

- A binary operator has two operands
- A ternary operator has three operands // average = (count ==0) ? 0 : sum/count;

```
using System; /*Learning C# */
public class ThreeInputValues {
   static void Main()    {
      int valueOne = 10;
      int valueTwo = 20;
      int maxValue = valueOne > valueTwo ?  valueOne : valueTwo;
      Console.WriteLine(valueOne, valueTwo, maxValue);    } }
```

Precedence of Operators

w - x / y - z or w / x - y / z

4*3**2

- Typical precedence levels
  - parentheses
  - unary operators
  - ** (exponentiation)
  - *, /, %
  - +, -

Arithmetic Expressions: Operator Associativity Rule

Determines the order of operation between operators of the same precedence

–In most cases, left to right

x - y - z or x / y / z

```
b = 9;
c = 5;
a = b = c;
```

5 is assigned to both a and b because of the right-to-left associativity of the = operator.

In Fortran, the exponentiation is right asssociative.

```
2**3**4
```

In Ada, The exponentiation operator is non-associative. If you wish to do more than one exponentiation in an expression, you must use parentheses.

```
(2^3)^4;
```

or `2^(3^4);`

APL is different; all operators have equal precedence and all operators associate right to left

```
A*B+C
```

/* demonstrate various operator precedence and associativity

```
#include <stdio.h>
int fun( int * );

int main() {
   int a, b, c, d;
   int n[5] = {0,2,4,6,8};
   int *ip = &n[0];
   printf ("%p %d \n", ip, *(ip + 1) );
   printf ("%p %d \n", ip, *ip + 1 );
   a = 7;
   b=3;
   c = a % b * 4;

   int count = 5;
   int count1= - count ++; //-5
   printf ("%d \n", count1);
   int count1= - ++count; //-6
   printf ("%d \n", count1);}
```

Functional side effects:

A side effect of a function occurs when the function changes **parameters** or a **global variable.**

a + fun(a)

If fun does not have the side effect of changing a, then no effect on the value of the expression.

However if fun changes a, there is an effect.

(1) Function changes parameters

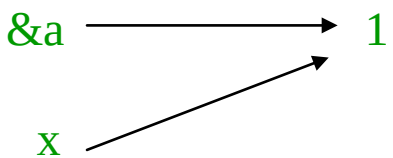
```
int f1(int *x) {
        (*x)++;
        return *x;
}
int f2(int *x) {
        (*x) *= (*x);
        return *x;}
int main() {
        int a = 1;
        printf("%d\n",f2(&a) + f1(&a));
}
```

left-to-right:

```
  f2: set a to 1;
      return 1
  f1: set a to 2
      return 2;
  print 3
```

right-to-left:

```
  f1: set a to 2
      return 2
  f2: set a to 4
      return 4
  print 6
```



(2) Function changes global variables

```
public class OperandOrder {
    public static int x;
    public static int f(int y) {
        x++;
        return y;
     }

public static void main (String[] args) {
     int result;
     x = 3;
     result = x + f(5);
     System.out.println("First evaluation = " + result);
     x = 3;
     result = f(5) + x;
     System.out.println("Second evaluation = " + result);
     } // end of main ()
}// OperandOrder
```

```
First evaluation = 8
Second evaluation = 9
```

Possible solutions to the problem

Write the language definition to demand that operand evaluation order be fixed

Java guarantees that the operands of operators appear to be evaluated from left to right.

Overloaded Operators

An operator is used for more than one purpose

+ is compiled into one prodecure for ints, another for floats, and another for Strings.

- C++ and Ada allow user-defined overloaded operators

```
#include <iostream>
using namespace std;

class complx {
    double real, imag;
public:
    complx( double real = 0., double imag = 0.); // constructor
    complx operator+(const complx&) const;       // operator+() };
complx::complx( double r, double i ){
    real = r; imag = i;}
// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const {
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result; }
int main() {
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()}
```

- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

- Boolean Expressions

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not

No Boolean Type in C

- C has no Boolean type--it uses `int` type with 0 for false and nonzero for true

One odd result of C's design is that the expression

a>b>c

is legal.

Short Circuit Evaluation

- The result of an expression is determined without evaluating all of the operands and/or operators

To evaluate X && Y, first evaluate X. If X is false then stop: the whole expression is false. Otherwise, evaluate Y then AND the two values.

- goal: efficiency

```
int a = 0;
  if (a && myfunc(b)) {
      do_something();
  }
```

- In C-based languages: && and || are short-circuited.

Assignment Statements

- The assignment operator
  - = FORTRAN, BASIC, PL/I, C, C++, Java
  - := ALGOLs, Pascal, Ada

Assignment Statements: Compound Operators

- A shorthand method of specifying a commonly needed form of assignment

- Introduced in ALGOL; adopted by C

a = a + b

is written as a += b

Unary Assignment Operators

If the operator is used as a prefix operator

Sum = ++ count;

This operation could also be stated as:

Count = count +1;

Sum = count;

If the operator is used as a postfix operator

Sum = count++;

This operation could also be stated as:

Sum = count;

Count = count +1;

```
sum += ++count
```

```
sum += count++
```

Assignment as an Expression

- makes it possible to write compact loops:

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

```
while ((ch = getchar())!= EOF){…}
```

```
void strcpy(char *q, char *p) {  // copies a string into another
        while (*q++ = *p++);
}
```

Mixed-Mode Assignment

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

```
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 c-based languages, only widening assignment coercions are done.

- In Ada, there is no assignment coercion.