Chapter 7 Expressions and Assignment statements

7.1 Introduction

- Expressions are the fundamental means of specifying computations in a programming language
  - Semantics of expressions are discussed in this Chapter
  - To understand the expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation
  - The essence of the imperative programming languages is the dominant role of assignment statements

7.2 Arithmetic Expressions

- Automatic evaluation of arithmetic expressions similar to those found in mathematics, science, and engineering was one of the primary goals of the first high-level programming language.
- Arithmetic expressions consist of
  - Operator, operands, parentheses, and function calls

7.2 Arithmetic Expressions

- Design issues for arithmetic expressions
  - Operator precedence rules?
  - Operator associativity rules?
  - Order of operand evaluation?
  - Operand evaluation side effects?
  - Operator overloading?
  - Type mixing in expressions?

7.2.1 Operator Evaluation Order

- The *operator precedence rules* for expression evaluation define the order in which "*adjacent*" operators of different precedence levels are evaluated
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (if the language supports it)
  - *, /
  - +, -

7.2.1 Operator Evaluation Order (Cont'd)

- The *operator associativity rules* for expression evaluation 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
  - 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
- Precedence and associativity rules can be overridden with parentheses

7.2.1.6 Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:

```
average = (count == 0)? 0 : sum / count
```

  - Evaluates as if written as follows:

```
if (count == 0)
   average = 0
else
   average = sum /count
```

7.2.2 Operand Evaluation Order

- Variables
  - Fetch the value from memory
- Constants:
  - Sometimes a fetch from memory; sometimes the constant is in the machine language instruction
- Parenthesized expressions:
  - evaluate all operands and operators first
- The most interesting case is when an operand is a function call

7.2.2.1 Side Effects

- A **side effect** of a function occurs when the function changes either one of its parameters or a global variable

7.2.2.1 Side Effects (Cont'd)

- 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 that fun changes its parameter */
b = a + fun(&a);
```

- The following program compiled with gcc version 4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4). The execution result is "`a=20`".

```
int a=5;
int fun1() {
  a=17;
  return 3;
}
void main() {
   a=a+fun1();
   printf("a=%d\n",a);
}
```

7.2.2.1 Side Effects (Cont’d)

- Note that functions in mathematics do not have side effects, because there is no notion of variables in mathematics.

7.2.2.1 Side Effects (Cont'd)

- 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 one-way parameters and lack of non-local references
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations
     - Java requires that operands appear to be evaluated in left-to-right order

7.2.2.2 Referential Transparency and Side Effects

- A program has the property of *referential transparency* if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If `fun` has no side effects, `result1 = result2`

Otherwise, not, and referential transparency is violated

7.2.2.2 Referential Transparency and Side Effects

- Advantage of referential transparency
  - Semantics of a program is much easier to understand if it has referential transparency
- Because they do not have variables, programs in pure functional languages are referentially transparent
  - Functions cannot have state, which would be stored in local variables
  - If a function uses an outside value, it must be a constant (there are no variables). So, the value of a function depends only on its parameters

7.3 Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*
  - Some are common (e.g., + for **`int`** and **`float`**)
  - It is generally thought to be acceptable, as long as neither readability nor reliability suffers

7.3 Overloaded Operators (Cont'd)

- Some are potential trouble
  - E.g.
    - `*` in C and C++
    - `x=&y; c=a&b;`
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability

7.3 Overloaded Operators (Cont'd)

- Some languages that support abstract data types, for example, C++, C#, and F#, allow the programmer to further overload operation symbols
  – See next slice
- C++ has a few operators that cannot be overloaded.
  – Structure member operator (.) and scope resolution operation (::)
- Interestingly, operator overloading was one of the C++ features that was not copied in to Java
  – However, it did reappear in C#

```cpp
#include <iostream.h>

class Complex
   {
   public:
      Complex(double=0.0,double=0.0);
      Complex operator +(Complex);
      Complex add(Complex);
      void Print();
   private:
      double Real;
      double Imag;
   };


//Constructor
Complex::Complex(double r, double i)
   {
   Real = r;
   Imag = i;
   }
```

```cpp
// implementation of addition operator
Complex Complex::operator +(Complex CNum)
   {
   Complex C;
   C.Real = Real + CNum.Real;
   C.Imag = Imag + CNum.Imag;
   return C;
   }
Complex Complex::add(Complex CNum)
   {
   Complex C;
   C.Real = Real + CNum.Real;
   C.Imag = Imag + CNum.Imag;
   return C;
   }

// implementation of print function
//---------------------------------
void Complex::Print()
   {
   cout << "Complex Number= "<<Real<<"+i"<<Imag<<endl;
   }

// simple main program
//--------------------
int main()
   {
   // Declare objects of complex class

   Complex x(22,2), y(11,3),z;

   z=x+y;
}
```

7.4 Type Conversions

- Type conversions are either *narrowing* or *widening*
  - 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., **`float`** to **`int`**
  - 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 the original type
    - e.g., **`int`** to **`float`**

7.4 Type Conversions (Cont'd)

- Widening conversions are *nearly* always safe, meaning that the magnitude of the converted value is maintain
  - It can result in reduced accuracy
    - 32-bit integer allows at least nine decimal digits of precision
    - 32-bit float-point values are with only about seven decimal digits of precision

7.4.1 Coercion in Expressions

- One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types
  - **Mixed-mode expression**
  - Must define conversions for implicit operand type conversions
    - Because computers do not have binary operations that take operands of different types

7.4.1 Coercion in Expressions (Cont'd)

- **Mixed-mode expression**
  - For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands
  - Language designers are not in agreement on the issue of coercions in arithmetic expressions.
    - Reduce the benefits of type checking

7.4.1 Coercion in Expressions (Cont'd)

- `int a;`
- `float b, c, d;`
- …
- `d=b*a; //a is a keying error`

- Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error
- F# and ML do not allow

7.4.2 Explicit Type Conversion

- Most languages provide some capabiity for doing explicit conversions,
  - Widening and narrowing
    - Warning messages may be produced
- Called *casting* in C-based languages
  - Examples
    - `C: (`**`int`**`)angle`
    - `F#: `**`float`**`(sum)`

7.4.3 Errors in Expressions

- If the language requires type checking, then operand type errors cannot occur
- Other kinds of errors:
  - Inherent limitations of arithmetic
    e.g., division by zero
  - Limitations of computer arithmetic
    e.g. overflow
- Often ignored by the run-time system

7.6 Short-Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators
- Example: `(13 * a) * (b / 13 – 1)`
  - If `a` is zero, there is no need to evaluate `(b/13 - 1)`
- However, in arithmetic expressions, this shortcut is not easily detected, so it is never taken

7.6 Short-Circuit Evaluation (Cont'd)

- Unlike the case of arithmetic expressions, the shortcut of Boolean expression can be easily discovered.
  - `(a>=0) && (b<10)`

7.6 Short-Circuit Evaluation (Cont'd)

- Problem with non-short-circuit evaluation
  - SCE and non-SCE are with different execution results

```
index = 0;
while ((index<=listlen) && (list[index]!= key)
 index=index+1;
```

- A language that provides SCEs of Boolean expressions and also has side effects in expressions allows subtle errors to occur

```
(a>b)||((b++)/3)
```

7.6 Short-Circuit Evaluation (Cont'd)

- Ada solution: by using two-word operations to activate SCE (The best solution)
  - "`and then`", "`or else`"
- In C-based language, the usual AND and OR operations, `&&` and `||`, respectively, are short-circuit.

7.7 Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`

- The assignment operator

  = Fortran, BASIC, the C-based languages

  := Ada, Pascal

- = can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)

7.7.2 Conditional Targets

- Conditional targets (Perl)
```
($flag ? $total : $subtotal) = 0
```

Which is equivalent to

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

7.7.3 Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and the C-based languaes
  - Example

```
a = a + b
```

can be written as

```
a += b
```

7.7.4 Unary Assignment Operators

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

  `sum = ++count` (`count` incremented, then assigned to `sum`)

  `sum = count++` (`count` assigned to `sum`, then incremented

  `count++` (`count` incremented)

  `-count++` (`count` incremented then negated)

7.7.5 Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

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

`ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the **`while`** statement

- Disadvantage: another kind of expression side effect

7.7.6 Multiple Assignments

- Perl, Ruby, and Lua allow multiple-target multiple-source assignments

```
($first, $second, $third)=(20, 30, 40);
```

- Also, the following is legal and performs an interchange:

```
($first, $second)=($second, $first);
```

7.8 Mixed-Mode Assignment

- Assignment statements can also be mixed-mode
- In Fortran, C, Perl, and C++, any numeric type value can be assigned to any numeric type variable
- In Java and C#, only widening assignment coercions are done
- In Ada, there is no assignment coercion

Summary

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