MODULE 2
Expressions and Assignment Statements

Expressions and Assignment Statements

- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Arithmetic Expressions

- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
- The purpose of an arithmetic expression is to specify an arithmetic computation.
- An implementation of such a computation must cause two actions: fetching the operands, usually from memory, and executing arithmetic operations on those operands.

- **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 type mixing is allowed in expressions?

➢Operator Evaluation Order

- The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

➢ **Precedence**

- The value of an expression depends at least in part on the order of evaluation of the operators in the expression.
- Consider the following expression:
  - a + b * c
  - Suppose the variables a, b, and c have the values 3, 4, and 5, respectively.
  - If evaluated left to right (the addition first and then the multiplication), the result is 35. If evaluated right to left, the result is 23.

- The operator precedence rules of the common imperative languages are nearly all the same, because they are based on those of mathematics.
- In these languages, exponentiation has the highest precedence (when it is provided by the language), followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.

➢ Associativity

- Consider the following expression:

  a - b + c - d

  - If the addition and subtraction operators have the same level of precedence, as they do in programming languages, the precedence rules say nothing about the order of evaluation of the operators in this expression.

- When an expression contains two adjacent 2 occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity rules** of the language.
- An operator can have either **left or right associativity**, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.

- The associativity rules for a few common languages are given here:

Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +
Ada	Left: all except **
	Nonassociative: **

The ** operator is exponentiation.

- In APL, all operators have the same level of precedence.

Parentheses

- A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.
- For example, although multiplication has precedence over addition, in the expression

  (A + B) * C

  the addition will be evaluated first.

- The **operator precedence rules** for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated

  ("adjacent" means they are separated by at most one operand)
- Typical precedence levels:

  1. parentheses
  2. unary operators
  3. ** (if the language supports it)
  4. *, /
  5. +, -

- 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., FORTRAN)
- Precedence and associativity rules can be overriden with **parentheses**

➢ Conditional Expressions

- **if-then-else statements** can be used to perform a conditional expression assignment. For example, consider

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

- In the C-based languages, this code can be specified more conveniently in an assignment statement using a conditional expression, which has the form

  expression_1 ? expression_2 : expression_3

  - where expression_1 is interpreted as a Boolean expression. If expression_1 evaluates to true, the value of the whole expression is the value of expression_2; otherwise, it is the value of expression_3. For example, the effect of the example

- if-then-else can be achieved with the following assignment statement, using a conditional expression:

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

- In effect, the question mark denotes the beginning of the **then** clause, and the colon marks the beginning of the **else** clause. Both clauses are mandatory.
- Note that **?** is used in conditional expressions as a ternary operator.

➢Operand Evaluation Order

- Variables in expressions are evaluated by fetching their values 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

➢ **Side Effects**

- A side effect of a function, naturally called a **functional side effect**, occurs when the function changes either one of its parameters or a global variable.
- Consider the expression

  a + fun(a)
- Consider the following situation:

  fun returns 10 and changes the value of its parameter to 20.

  Suppose we have the following:

  a = 10;

  b = a + fun(a);

  Then, if the value of a is fetched first (in the expression evaluation process), its value is 10 and the value of the expression is 20. But if the second operand is evaluated first, then the value of the first operand is 20 and the value of the expression is 30.

- The following C program illustrates the same problem when a function changes a global variable that appears in an expression:

```
int a = 5;
int fun1() {
        a = 17;
        return 3;
} /* end of fun1 */
void main() {
        a = a + fun1();
} /* end of main */
```

- The value computed for a in main depends on the order of evaluation of the operands in the expression a + fun1().
- The value of a will be either 8 (if a is evaluated first) or 20 (if the function call is evaluated first).

- There are two possible solutions to the problem of operand evaluation order and side effects.
- First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects.
- Second, the language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order.

Referential Transparency and Side Effects

- The concept of referential transparency is related to and affected by functional 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.
- The value of a referentially transparent function depends entirely on its parameters.
- The connection of referential transparency and functional side effects is illustrated by the following example:

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

  temp = fun(a);

  result2 = (temp + b) / (temp - c);

  - If the function fun has no side effects, result1 and result2 will be equal, because the expressions assigned to them are equivalent.
  - However, suppose fun has the side effect of adding 1 to either b or c. Then result1 would not be equal to result2.
- So, that side effect violates the referential transparency of the program in which the code appears.

Overloaded Operators

- Arithmetic operators are often used for more than one purpose.
- For example, + usually is used to specify integer addition and floating-point addition. Some languages—Java, for example—also use it for string catenation.
- This multiple use of an operator is called **operator overloading**
- operator overloading was one of the C++ features
- C++ has a few operators that cannot be overloaded. Among these are the class or structure member operator (.) and the scope resolution operator (::)

- As an example of the **possible dangers of overloading**,
- consider the use of the **ampersand (&) in C++**.
- As a binary operator, it specifies a bitwise logical AND operation.
- As a unary operator, however, its meaning is totally different.
- As a unary operator with a variable as its operand, the expression value is the address of that variable. In this case, the ampersand is called the address-of operator. For example, the execution of

  ```
  x = &y;
  ```

  causes the address of y to be placed in x.
- There are two problems with this multiple use of the ampersand.
  - First, using the same symbol for two completely unrelated operations is detrimental to readability.
  - Second, the simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler, because it is interpreted as an address-of operator. Such an error may be difficult to diagnose.

Type Conversions

- Type conversions are either narrowing or widening.
- A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type.
  - For example, converting a **double to a float** in Java is a narrowing conversion, because the range of **double** is much larger than that of **float.**
- A **widening conversion** converts a value to a type that can include at least approximations of all of the values of the original type.
  - For example, converting an **int to a float** in Java is a widening conversion.
- Widening conversions are nearly always safe, meaning that the magnitude of the converted value is maintained.
- Narrowing conversions are not always safe—sometimes the magnitude of the converted value is changed in the process.
  - For example, if the floating-point value 1.3E25 is converted to an integer in a Java program, the result will be only distantly related to the original value.

- Although widening conversions are usually safe, they can result in reduced accuracy.
- In many language implementations, although integer-to-floating-point conversions are widening conversions, some precision may be lost.
- For example, in many cases, integers are stored in 32 bits, which allows at least nine decimal digits of precision. But floating-point values are also stored in 32 bits, with only about seven decimal digits of precision (because of the space used for the exponent). So, integer-to-floating-point widening can result in the loss of two digits of precision.

- Type conversions can be either implicit or explicit.

➢Coercion in Expressions

- Implicit conversion to the expected type is called a type coercion, is automatically done
- 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.
- When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and supply the code for that coercion.

- consider the following Java code:

```
int a;
float b, c, d;
. . .
d = b * a;
```

- Assume that the second operand of the multiplication operator was supposed to be c, but because of a keying error it was typed as a.
- Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error.
- It would simply insert code to coerce the value of the **int** operand, a, to **float.**
- If mixed-mode expressions were not legal in Java, this keying error would have been detected by the compiler as a type error.

Explicit Type Conversion

- **Explicit conversion**, which is also called **casting**, is performed by code instructions.
- In the C-based languages, explicit type conversions are called **casts.**
- To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

  (**int**) angle

- Example:

r=(float) n;//generates a code for run-time conversion

n=(int) r; //run-time conversion, no overflow check in C

Errors in Expressions

- A number of errors can occur during expression evaluation.
- If the language requires type checking, either static or dynamic, then operand type errors cannot occur.
- The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called **overflow or underflow,** depending on whether the result was too large or too small.
- One limitation of arithmetic is that division by zero is disallowed.
- Floating-point overflow, underflow, and division by zero are examples of run-time errors, which are sometimes called **exceptions.**

Relational and Boolean Expressions

- In addition to arithmetic expressions, programming languages support relational and Boolean expressions.

Relational Expressions

- A relational expression has two operands and one relational operator.
- A **relational operator** is an operator that compares the values of its two operands.
- The value of a relational expression is Boolean, except when Boolean is not a type included in the language.
- The relational operators always have lower precedence than the arithmetic operators, so that in expressions such as

  a + 1 > 2 * b

  the arithmetic expressions are evaluated first.
- The syntax of the relational operators for equality and inequality differs among some programming languages. For example, for inequality, the C-based languages use !=, Ada uses /=, Lua uses ~=, Fortran 95+ uses .NE. or <>, and ML and F# use <>.

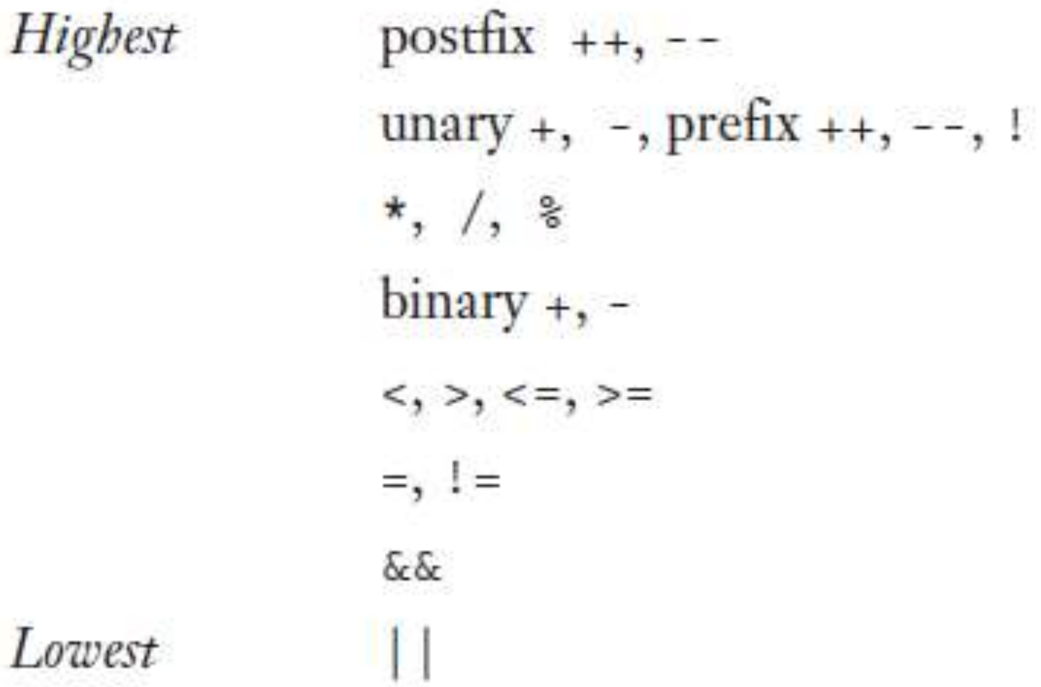
- One odd result of C's design of relational expressions is that the following expression is legal:

  a > b > c
- The leftmost relational operator is evaluated first because the relational operators of C are left associative, producing either 0 or 1.
- Then, this result is compared with the variable c.
- There is never a comparison between b and c in this expression.

Boolean Expressions

- Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators.
- The operators usually include those for the AND, OR, and NOT operations, and sometimes for exclusive OR and equivalence.
- Boolean operators usually take only Boolean operands and produce Boolean values.
- Ada's AND and OR operators have equal precedence. However, the C-based languages assign a higher precedence to AND than OR.

- The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows:



Highest	postfix ++, --
	unary +, -, prefix ++, --, !
	*, /, %
	binary +, -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

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.
- For example, the value of the arithmetic expression

  (13 * a) * (b / 13 - 1)

  is independent of the value of (b / 13 - 1) if a is 0, because 0 * x = 0 for any x.

  So, when a is 0, there is no need to evaluate (b / 13 - 1) or perform the second multiplication.
- However, in arithmetic expressions, this shortcut is not easily detected during execution, so it is never taken.

- The value of the Boolean expression

  (a >= 0) && (b < 10)

  is independent of the second relational expression if a < 0, because the expression (FALSE && (b < 10)) is FALSE for all values of b.

  - So, when a < 0, there is no need to evaluate b, the constant 10, the second relational expression, or the && operation.
- Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.

- In the **C-based languages**, the usual AND and OR operators, && and ||, respectively, are **short-circuit**. However, these languages also have bitwise AND and OR operators, & and |, respectively, that can be used on Boolean-valued operands and are **not short-circuit**.
- All of the logical operators of Ruby, Perl, ML, F#, and Python are **short-circuit** evaluated.
- **Ada** allows the programmer to specify **short-circuit** evaluation of the Boolean operators AND and OR by using the two-word operators **and then** and **or else**. Ada also has **non–short-circuit** operators, **and** and **or**.

Assignment Statements

- It provides the mechanism by which the user can dynamically change the bindings of values to variables.

➢ **Simple Assignments**

- Nearly all programming languages currently being used use the **equal sign** for the assignment operator.
- All of these must use something different from an equal sign for the equality relational operator to avoid confusion with their assignment operator.
- ALGOL 60 pioneered the use of := as the assignment operator, which avoids the confusion of assignment with equality. Ada also uses this assignment operator.

Conditional Targets

- Perl allows conditional targets on assignment statements.
- For example, consider

```
($flag ? $count1 : $count2) = 0;
```

which is equivalent to

```
if ($flag) {
        $count1 = 0;
} else {
        $count2 = 0;
}
```

Compound Assignment Operators

- The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

  a = a + b

- The syntax of these assignment operators is the catenation of the desired binary operator to the = operator.
- For example,

  sum += value;

  is equivalent to

  sum = sum + value;

Unary Assignment Operators

- The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments.
- They combine increment and decrement operations with assignment. The operators ++ for increment, and – – for decrement, can be used either in expressions or to form stand-alone single-operator assignment statements.
- They can appear either as prefix operators, meaning that they precede the operands, or as postfix operators, meaning that they follow the operands.
- In the assignment statement

  sum = ++ count;

  the value of count is incremented by 1 and then assigned to sum.

  - This operation could also be stated as

    count = count + 1;

    sum = count;

- If the same operator is used as a postfix operator, as in

  sum = count ++;

  - the assignment of the value of count to sum occurs first; then count is incremented.
  - The effect is the same as that of the two statements

    sum = count;

    count = count + 1;
- When two unary operators apply to the same operand, the association is right to left.
- For example, in

  - count ++

  - count is first incremented and then negated. So, it is equivalent to

    - (count ++)

Multiple Assignments

- Several recent programming languages, including Perl, Ruby, and Lua, provide multiple-target, multiple-source assignment statements.
- For example, in Perl one can write

  ($first, $second, $third) = (20, 40, 60);

  - The semantics is that 20 is assigned to $first, 40 is assigned to $second, and 60 is assigned to $third.
- If the values of two variables must be interchanged, this can be done with a single assignment, as with

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

Mixed-Mode Assignment

- One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types. Languages that allow such expressions, which are called **mixed-mode expressions.**
- Frequently, assignment statements also are mixed mode.
- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions; that is, many of the possible type mixes are legal, with coercion freely applied.
- Ada does not allow mixed-mode assignment.
- C++, Java and C# allow mixed-mode assignment only if the required coercion is widening. So, an **int** value can be assigned to a **float** variable, but not vice versa.