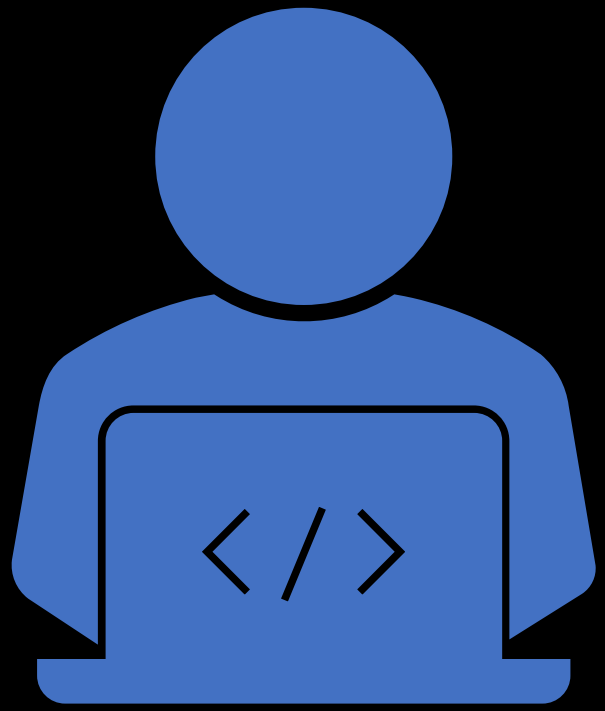


CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 12
Expressions and Assignment Statements

Lecture 12 Topics

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

Introduction

Introduction

- Expressions are the fundamental means of specifying computations in a programming language.
- It is crucial for a programmer to understand both the syntax and semantics of expressions of the language he or she uses.
- A formal mechanism (namely BNF) for describing the syntax of expressions was introduced before.
- In this chapter, the semantics of expressions are discussed.

Arithmetic Expressions

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 programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in mathematics.
- In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls.

```
a * (b + c) * factorial(c)
```

Arithmetic Expressions

- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
- Java examples:
  - **Unary** → `++i` /`i++` (can be prefix or postfix)
  - **Binary** → `i+j` (in most programming languages, binary operators are infix)
  - **Ternary** → `a ? i : j`

Operator Evaluation Order

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

Precedence

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

```
a + b * c
```

- Typical precedence levels (C-Based Languages)
  - parentheses
  - postfix `++, --`
  - prefix `++, --,` unary `+,-`
  - `*, /, %`
  - binary `+, -`

Associativity

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.
- 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.
- Associativity in common languages is left to right. In the Java expression

```
a - b + c
```

- the left operator ('–') is evaluated first.

Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
- 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.

Conditional Expressions

- In the C-based languages, an assignment statement code can be written using a conditional expression, which has the following form:

```
expression_1 ? expression_2 : expression_3
```

- **EXAMPLE**: `average = (count == 0) ? 0 : sum / count;`
- Evaluates as if written as follows:

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

Overloaded Operators

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 concatenation.
- This multiple use of an operator is called operator overloading and is generally thought to be acceptable, as long as neither readability nor reliability suffers.

Overloaded Operators

- 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.
- For example, the execution of

```
x = &y;
```

- causes the address of y to be placed in x.

Overloaded Operators

- 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.

Overloaded Operators

- When sensibly used, user-defined operator overloading can aid readability.
- For example, if `+` and `*` are overloaded for a matrix abstract data type and `A, B, C,` and `D` are variables of that type, then

```
A * B + C * D
```

- can be used instead of

```
MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
```

Overloaded Operators

- On the other hand, user-defined overloading can be harmful to readability.
- For one thing, nothing prevents a user from defining + to mean multiplication.
- Furthermore, seeing an * operator in a program, the reader must find both the types of the operands and the definition of the operator to determine its meaning. Any or all of these definitions could be in other files.

Type Conversions

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 approximate magnitude of the converted value is maintained.

Type Conversions

- 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 "`1.3`" is converted to an integer in a Java program, the result will not be in any way 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.

Type Conversions

- Type conversions of nonprimitive types are, of course, more complex.
- Type conversions can be either **explicit** or **implicit**.
- A **coercion** is an implicit type conversion (initiated by the compiler or runtime system).
- Type conversions explicitly requested by the programmer are referred to as explicit conversions, not coercions.
- 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 /(Student)st1
```

Relational and Boolean Expressions

Relational and Boolean Expressions

- In addition to arithmetic expressions, programming languages support relational and Boolean expressions.
- A **relational operator** is an operator that compares the values of its two operands.
- A **relational expression** has two operands and one relational operator.
- The value of a relational expression is **Boolean**.
- C has no Boolean type--it uses int type with 0 for false and nonzero for true (decreases readibility).

Relational and Boolean Expressions

- The syntax of the relational operators for equality and inequality differs among some programming languages.
- For example, the C-based languages use "==" for equality and "!=" for inequality.
- The relational operators always have lower precedence than the arithmetic operators, so that in expressions such as

```
a + 1 > 2 * b
```

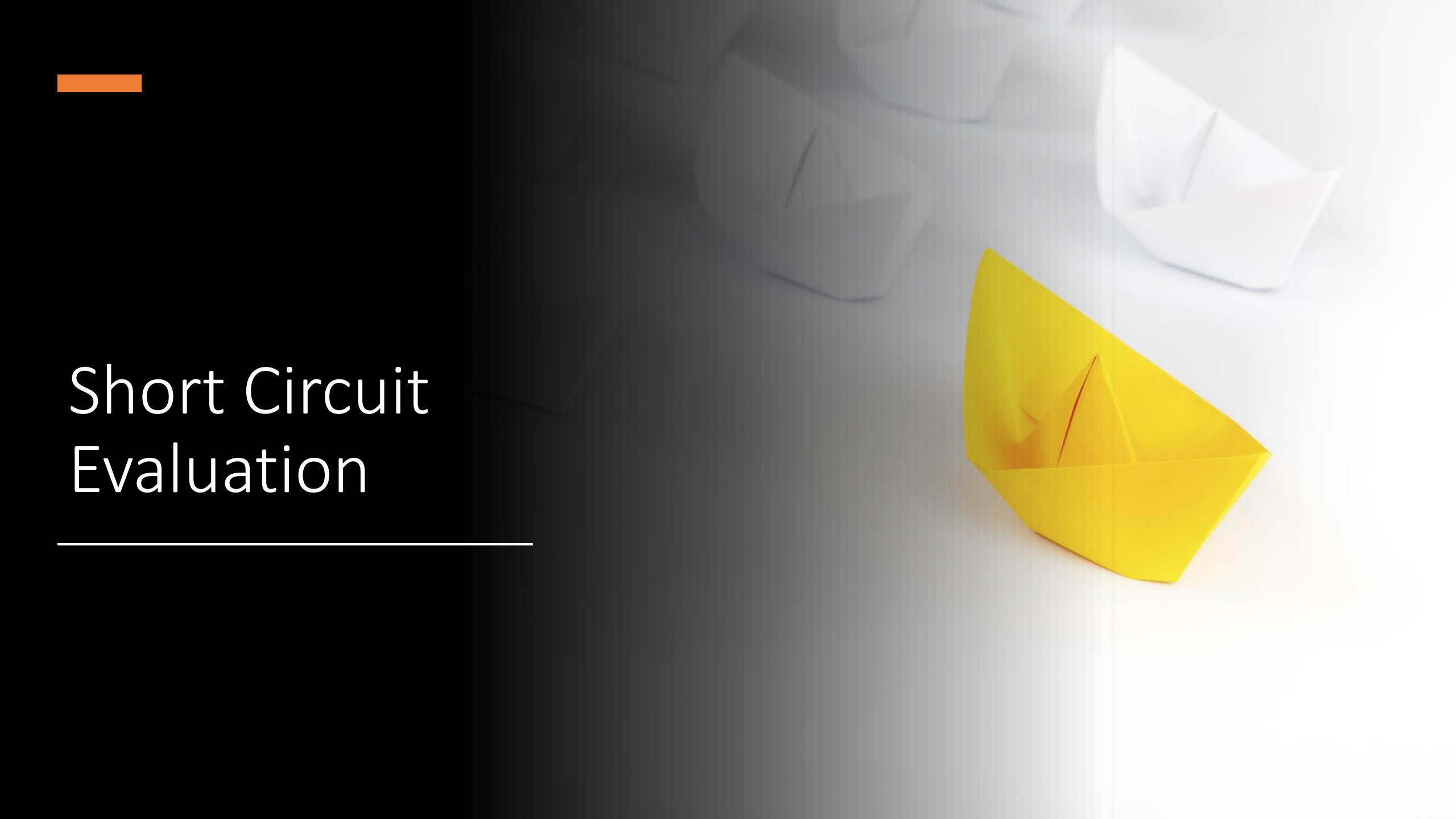
- the arithmetic expressions are evaluated first.

Relational and Boolean Expressions

- In the mathematics of Boolean algebras, the OR and AND operators must 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 +, unary -, prefix ++, --, !
	*, /, %
	binary +, binary -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

Short Circuit Evaluation



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.

Short Circuit Evaluation

- 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` is less than zero, 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.

Short Circuit Evaluation

- To illustrate a potential problem with non-short-circuit evaluation of Boolean expressions, suppose Java did not use short-circuit evaluation.
- A table lookup loop could be written using the while statement.
- One simple version of Java code for such a lookup, assuming that list, which has `listlen` elements, is the array to be searched and key is the searched-for value, is

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

Short Circuit Evaluation

- If evaluation is not short-circuit, both relational expressions in the Boolean expression of the while statement are evaluated, regardless of the value of the first.
- The same iteration that has `index == listlen` will reference `list[listlen]` , which causes the indexing error because list is declared to have `listlen-1` as an upper-bound subscript value.
- Thus, if `key` is not in `list`, the program will terminate with a subscript out-of-range exception.

Short Circuit Evaluation

- A language that provides short-circuit evaluations of Boolean expressions and also has side effects in expressions allows subtle errors to occur.
- Suppose that short-circuit evaluation is used on an expression and part of the expression that contains a side effect is not evaluated; then the side effect will occur only in complete evaluations of the whole expression.
- If program correctness depends on the side effect, short-circuit evaluation can result in a serious error.
- For example, consider the Java expression

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

Short Circuit Evaluation

- In this expression, `b` is changed (in the second arithmetic expression) only when `a <= b`.
- If the programmer assumed `b` would be changed every time this expression is evaluated during execution (and the program's correctness depends on it), the program will fail.

Assignment Statements

Assignment Statements

- The assignment statement is one of the central constructs in imperative languages.
- It provides the mechanism by which the user can dynamically change the bindings of values to variables.
- **The general syntax**

<target_var> <assign_operator> <expression>

- **The assignment operator**

= Fortran, BASIC, the C-based languages

:= Ada, Pascal

Assignment Statements: Compound Assignment Operators

- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment.
- 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
```

- This statement can be written as

```
a += b
```

Assignment Statements: 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.

Assignment Statements: Unary Assignment Operators

- An example of the use of the unary increment operator to form a complete assignment statement is

```
count++;
```
(or `++count;`)

- which simply increments `count`.
- It does not look like an assignment, but it certainly is one. It is equivalent to the statement

```
count = count + 1;
```

Assignment Statements: Unary Assignment Operators

- 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;
```

Assignment Statements: Unary Assignment Operators

- 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++)
```

- rather than

```
(-count)++
```

Assignment Statements: Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result, which is the same as the value assigned to the target.
- It can therefore be used as an expression and as an operand in other expressions.
- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- For example, in C, it is common to write statements such as

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

- In this statement, the next character from the standard input file, usually the keyboard, is gotten with `getchar` and assigned to the variable `ch`.
- The result, or value assigned, is then compared with the constant `EOF`. If `ch` is not equal to `EOF`, the compound statement `{...}` is executed.

Assignment Statements: Assignment as an Expression

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors.
- In particular, if we type

```
if (x = y) ...
```

- instead of

```
if (x == y) ...
```

- which is an easily made mistake, it is not detectable as an error by the compiler.