Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Programming-1 CS112 **Arithmetic expressions** Dr. Amin Allam

1 Variables

A *variable* occupies a number of bytes in memory and can store *values*. There are several types of variables such as: *char*, *int*, *double*.

A *char* variable can store a character such as: `'A'` , `'('` , `'t'`, `'_'` , `'7'` , `'-'`.
An *int* variable can store an integer value such as: 14 , 0 , -3 , 1023.
A *double* variable can store a decimal value such as: 6.74 , 3.14 , -189.1 , 45.

The name of a variable is a sequence of letters such that:
• Each letter in the variable name can be:
(1) English letter (`'A'...'Z'`, `'a'...'z'`).
(2) Underscore (`'_'`).
(3) Digit (`'0'...'9'`).
• A variable name can not start with a digit.

```
1  int 3a, 9A, 8_u, a@b, ab!m;     ✖  // Incorrect variable names
2  int _3a, A_x, _Y4r, p_ta, O4;   ✔  // Correct variable names
```

2 Expressions

An *expression* consists of *operator* and *data* and performs a specific *operation* on the data. Data can be variables or values or both according to the used operator. Data associated to an operator are called *operands*. For example, the *assignment* (=) operator takes a variable or value on its right and assigns it to a variable on its left. Examples:

```
1  int a=17, b=5, c=8, x=6;   ✔  // Defines and initializes 4 integer variables
2
3  x = 5;                     ✔  // Stores 5 in x
4  x = a;                     ✔  // Stores 17 (the value of a) in x, does not change a
5  9 = 5;                     ✖  // Can not put value in the left of =
6  7 = a;                     ✖  // Can not put value in the left of =
7  x = 17 + 5;                ✔  // Stores 22 in x
8  x = a + b;                 ✔  // Stores 22 in x, does not change a or b
9  x = 17 + b;                ✔  // Stores 22 in x, does not change b
10 c = x + 5;                 ✔  // Stores 27 in c, does not change x
11 22 = x + 5;                ✖  // Can not put value in the left of =
```

The following examples illustrate the usage of various expressions:

```
1  int a=17, b=5, c=4, x;
2  double p=17, q=5, y;
3  x = 5 + 4;                      // Stores 5 + 4 = 9 in x
4  x = (5 + 4);                    // Stores 5 + 4 = 9 in x
5  x = b + c;                      // Stores 5 + 4 = 9 in x
6  x = 5 * 4;                      // Stores 5 × 4 = 20 in x
7  x = b * c;                      // Stores 5 × 4 = 20 in x
8  x = (b * c);                    // Stores 5 × 4 = 20 in x
9  x = 17 / 5;                     // Stores 17 div 5 = 3 in x
10 x = a / b;                      // Stores 17 div 5 = 3 in x
11 x = 17 % 5;                     // Stores 17 mod 5 = 2 in x
12 x = a % b;                      // Stores 17 mod 5 = 2 in x
13 y = 17 / 5;                     // Stores 17 div 5 = 3 in y
14 y = a / b;                      // Stores 17 div 5 = 3 in y
15 y = 17 % 5;                     // Stores 17 mod 5 = 2 in y
16 y = a % b;                      // Stores 17 mod 5 = 2 in y
17 y = 17.0 / 5.0;                 // Stores 17 ÷ 5 = 3.4 in y
18 y = 17.0 / 5;                   // Stores 17 ÷ 5 = 3.4 in y
19 y = 17 / 5.0;                   // Stores 17 ÷ 5 = 3.4 in y
20 y = p / q;                      // Stores 17 ÷ 5 = 3.4 in y
21 y = p % q;                   ✖  // Undefined operation
22 y = (double)a / (double)b;      // Stores 17 ÷ 5 = 3.4 in y
23 y = (double)a / b;              // Stores 17 ÷ 5 = 3.4 in y
24 y = a / (double)b;              // Stores 17 ÷ 5 = 3.4 in y
25 x = 17.0 / 5.0;                 // Evaluates 17 ÷ 5 = 3.4 then converts to integer 3
26 x = p / q;                      //   by discarding fractional part and stores 3 in x
27 x = 17 + 4 * 5;                 // Stores 17 + (4 × 5) = 37 in x
28 x = a + c * b;                  // Stores 17 + (4 × 5) = 37 in x
29 x = 17 + (4 * 5);               // Stores 17 + (4 × 5) = 37 in x
30 x = (17 + 4) * 5;               // Stores (17 + 4) × 5 = 105 in x
31 x = 4 + 17 / 5;                 // Stores 4 + (17 div 5) = 7 in x
32 x = c + (a / b);                // Stores 4 + (17 div 5) = 7 in x
33 x = (4 + 17) / 5;               // Stores (4 + 17) div 5 = 4 in x
34 y = 4.0 + 17 / 5;               // Stores 4 + (17 div 5) = 7 in y
35 y = 4 + 17.0 / 5;               // Stores 4 + (17 ÷ 5) = 7.4 in y
36 y = (double)(4 + 17) / 5;       // Stores (4 + 17) ÷ 5 = 4.2 in y
37 y = (4 + 17) / (double)5;       // Stores (4 + 17) ÷ 5 = 4.2 in y
38 y = (4 + 17) / 5.0;             // Stores (4 + 17) ÷ 5 = 4.2 in y
39 y = (c + p) / b;                // Stores (4 + 17) ÷ 5 = 4.2 in y
40 y = (c + (double)a) / b;        // Stores (4 + 17) ÷ 5 = 4.2 in y
41 x = a - b - c;                  // Stores (17 − 5) − 4 = 8 in x
42 x = (a - b) - c;                // Stores (17 − 5) − 4 = 8 in x
43 x = a - (b - c);                // Stores 17 − (5 − 4) = 16 in x
```

The operators in the previous examples has *order of precedence* as follows (ordered from highest precedence to lowest precedence):

Operator	Associativity	Type
`(type)`	right to left	Unary
`* / %`	left to right	Binary
`+ -`	left to right	Binary
`=`	right to left	Binary

A *unary* operator takes one operand, while a binary operator takes two *operands*.
If an expression contains sub-expressions with operators from the previous table, the association of operators to its operands is done in the following order:

- Associate what in parentheses `()`.
- Associate the type-cast `(type)` to its right operand. A type-cast constructs a temporary variable of the specified type and assigns to it the value of its right operand.
- Associate the `*`, `/` and `%` operators to its left and right operands,
- Associate the + and − operators to its left and right operands.
- Associate the = operators to its left and right operands.

If there are adjacent operators with the same order of precedence, they are associated in the order shown in the column "Associativity".

Assume the expression of the second line of the following code:

```
1 int a=17, b=5, c=4, d=8; double x;
2 x = d = a - b - (6 + b) * c / (double) d;
```

The expression may be evaluated internally by the compiler as follows:

```
1 int a=17, b=5, c=4, d=8; double x;
2 x = d = a - b - (6 + b) * c / (double) d; // Evaluate inside parentheses
3 x = d = a - b - 11 * c / (double) d;       // Type-cast operation
4 x = d = a - b - 11 * c / 8.0;              // Multiplication (left to right assoc)
5 x = d = a - b - 44 / 8.0;                  // Division
6 x = d = a - b - 5.5;                       // First subtraction (left to right assoc)
7 x = d = 12 - 5.5;                          // Subtraction
8 x = d = 6.5;          // Second assignment (right to left assoc), store the value 6 in d
9 x = 6;                // Assignment, store the value 6 in x
```

The following example explains the unary minus, increment and decrement operators:

```
1 int a=0, b=5, c=1;
2 a = -b;               // Store the value -5 in a (unary minus)
3 a = c + 1;            // Store the value 2 in a, does not change c
4 a + 1;                // This statement has no effect, does not change a
5 a = a + 1;            // Store the value 3 in a
6 a++;                  // Increase a by 1, store the value 4 in a
7 ++a;                  // Increase a by 1, store the value 5 in a
8 --a;                  // Decrease a by 1, store the value 4 in a
```

The following are more examples on the increment and decrement operators:

```
1 int a=4, b, c;
2 b = ++a;              // The same as {++a; b=a;} (a=5, b=5)
3 b = a++;              // The same as {b=a; a++;} (b=6, a=7)
4 c = b++ +a++;         // The same as {c=b+a; b++; a++;} (c=13, b=7, a=8)
5 c = ++b+ ++a;         // The same as {++b; ++a; c=b+a;} (b=8, a=9, c=17)
6 c = b++ + ++a;        // The same as {++a; c=b+a; b++;} (a=10, c=18, b=9)
7 c = a-- +b--;         // The same as {c=a+b; a--; b--;} (c=19, a=9, b=8)
8 c = --b+ ++a;         // The same as {--b; ++a; c=b+a;} (b=7, a=10, c=17)
9 c = b++ + --a;        // The same as {--a; c=b+a; b++;} (a=9, c=16, b=8)
```

The following are examples on the compound assignment operators:

```
1  int a=2, b=8, c=6;
2  a += 8;              // The same as {a=a+8;} (a=10)
3  a += b;              // The same as {a=a+b;} (a=18)
4  a -= c;              // The same as {a=a-c;} (a=12)
5  a /= c;              // The same as {a=a/c;} (a=2)
6  a *= b;              // The same as {a=a*b;} (a=16)
7  a %= c;              // The same as {a=a%c;} (a=4)
8  a += b-c;            // The same as {a=a+(b-c);} (a=6)
9  a -= b-c;            // The same as {a=a-(b-c);} (a=4)
10 a *= b-c;            // The same as {a=a*(b-c);} (a=8)
```

The following table summarizes properties of the previously studied operators (ordered in groups from highest precedence to lowest precedence):

Operator	Description	Associativity	Type	Example
`++ --`	Postfix increment and decrement	left to right	Unary	`a++`
`+ -`	Unary plus and minus	right to left	Unary	`-a`
`++ --`	Prefix increment and decrement	right to left	Unary	`++a`
`(type)`	C-style cast	right to left	Unary	`(double)a`
`* / %`	Multiplication, division, and remainder	left to right	Binary	`a*b`
`+ -`	Addition and subtraction	left to right	Binary	`a+b`
`=`	Assignment	right to left	Binary	`a=b`
`*= /= %=`	Compound assignment	right to left	Binary	`a*=b`
`+= -=`	Compound assignment	right to left	Binary	`a+=b`

Note that precedence and associativity are compile-time concepts. They are independent from order of evaluation, which is a run-time concept.