Syrian Private University

Introduction to Algorithms and Programming

Instructor: Dr. Mouhib Alnoukari

Expressions

Introduction

- Expressions are the fundamental means of specifying computations in a programming language
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation
- Essence of imperative languages is dominant role of assignment statements

Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- **If either or both operands used by an arithmetic operator are floating point, then the result is a floating point**

Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

```
14 / 3      equals      4

8 / 12      equals      0
```

  - **The remainder operator (%) returns the remainder after dividing the second operand into the first**

```
14 % 3      equals      2

8 % 12      equals      8
```

Operator Precedence

- Operators can be combined into complex expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order

Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
  1   2   3   4
```

```
a + b * c - d / e
  3   1   4   2
```

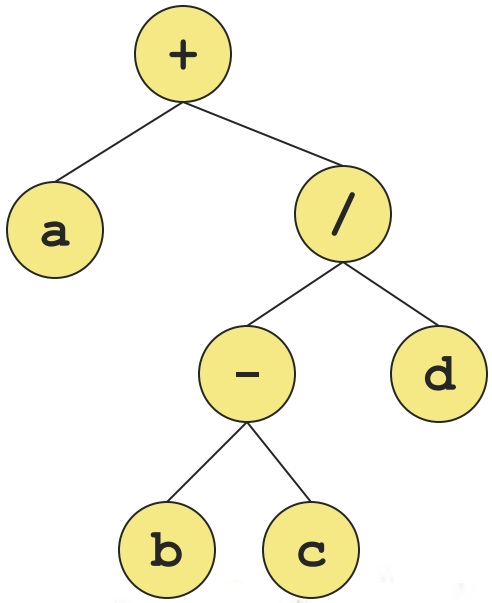
```
a / (b + c) - d % e
  2    1     4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Expression Trees

- The evaluation of a particular expression can be shown using an *expression tree*
- The operators lower in the tree have higher precedence for that expression

```
a + (b - c) / d
```



Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

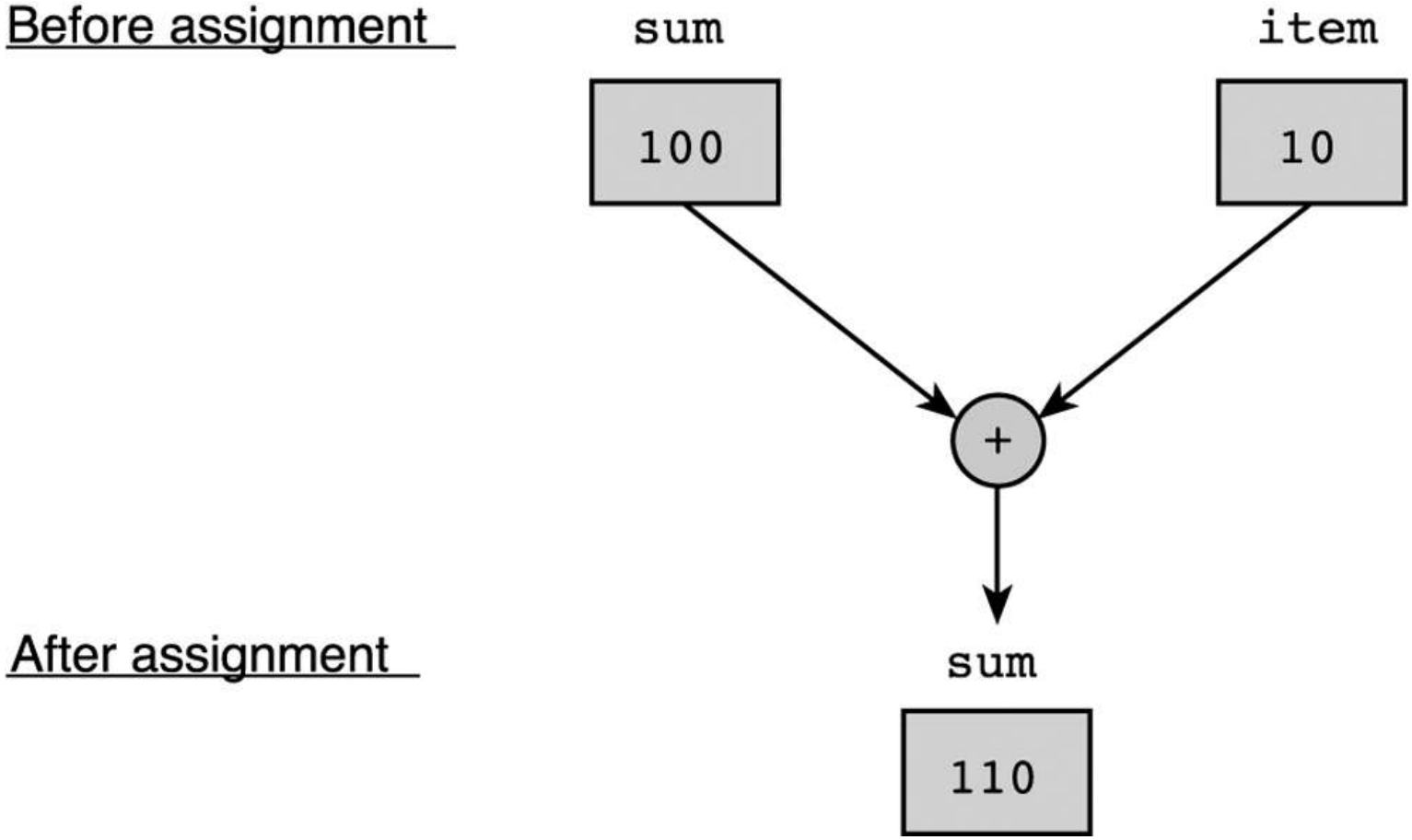
First the expression on the right hand side of the = operator is evaluated

```
answer  =  sum / 4 + MAX * lowest;
```

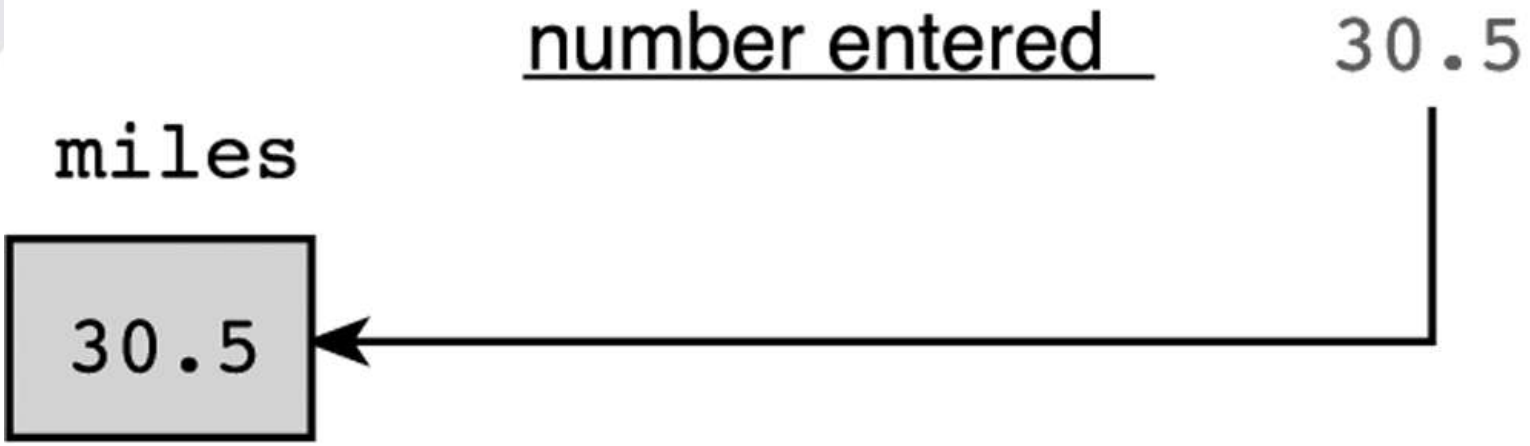
4 1 3 2

Then the result is stored in the variable on the left hand side

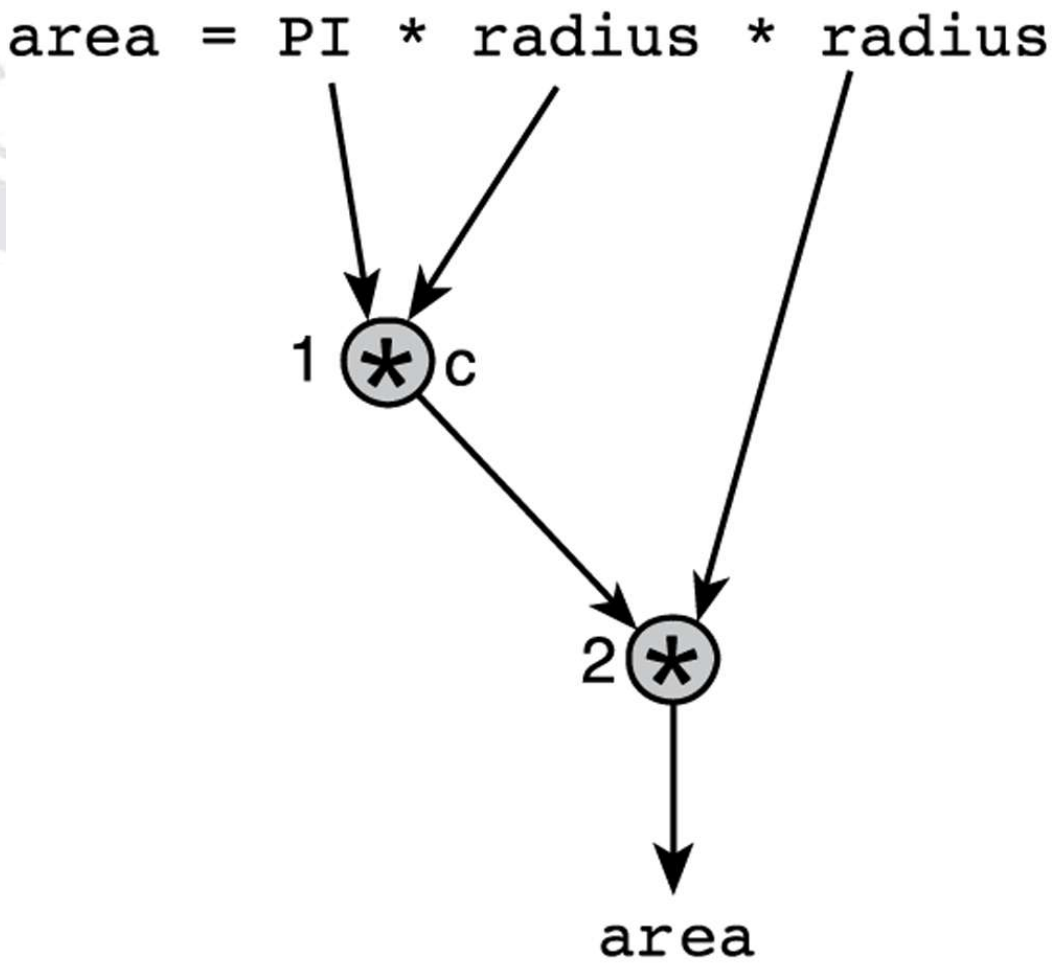
Effect of sum = sum + item;



Effect of scanf("%lf", &miles);



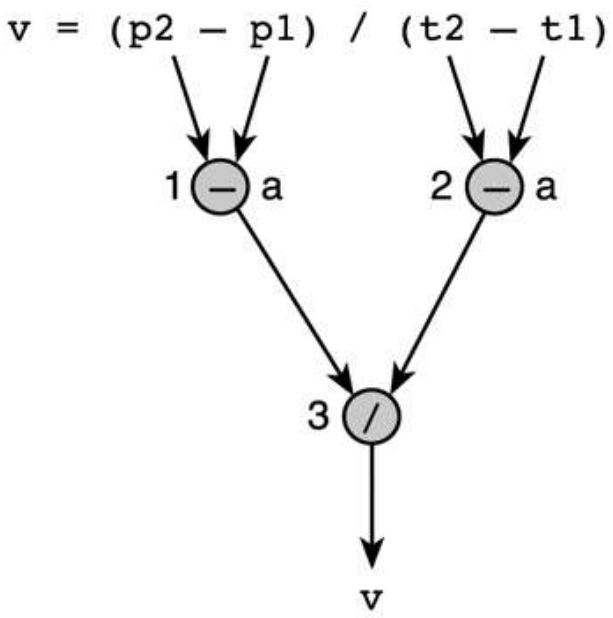
Evaluation Tree for
area = PI * radius * radius;

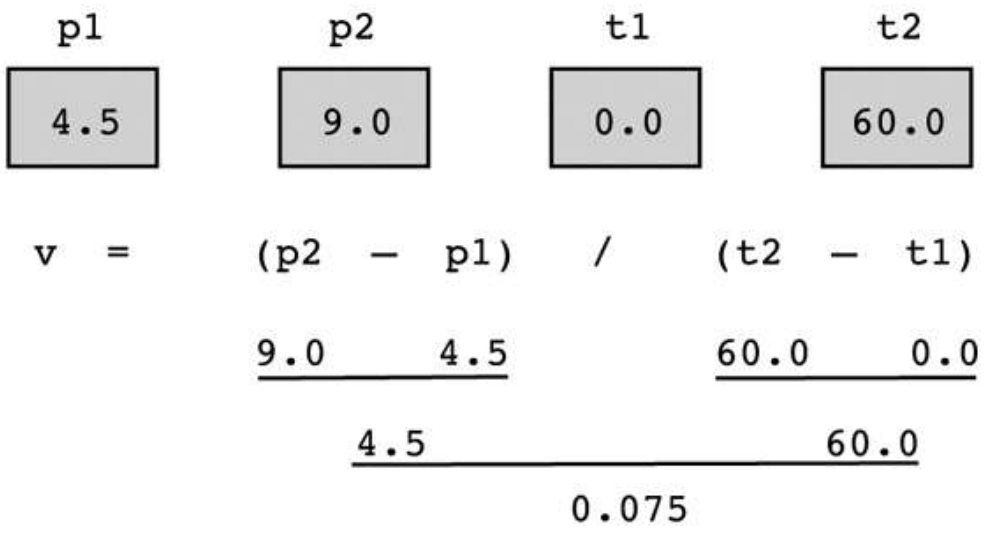


Step-by-Step Expression Evaluation

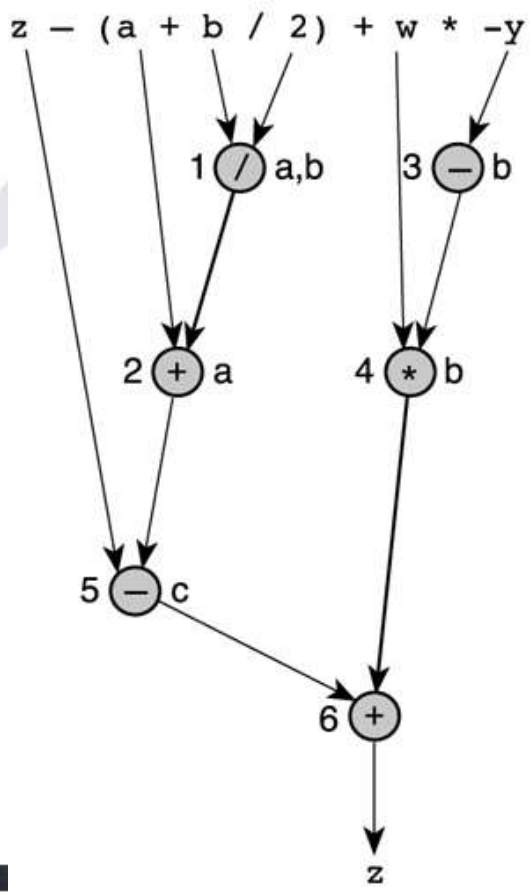
```
area    =      PI    *   radius   *   radius
           3.14159         2.0          2.0
                  6.28318
                                12.56636
```

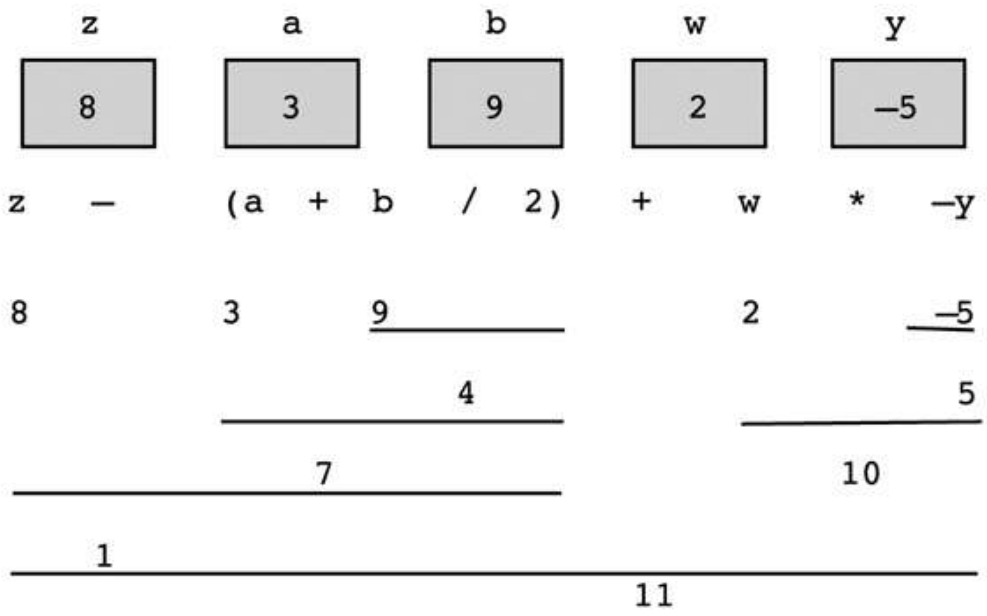
Evaluation Tree and Evaluation for v = (p2 - p1) / (t2 - t1);





Evaluation Tree and Evaluation for
z - (a + b / 2) + w * -y





Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable

First, one is added to the original value of count

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

Then the result is stored back into count (overwriting the original value)

Increment and Decrement

- The increment and decrement operators use only one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement

```
count++;
```

  is functionally equivalent to

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

Increment and Decrement

- The increment and decrement operators can be applied in *postfix form*:

```
count++
```

- or *prefix form*:

```
++count
```

- When used as part of a larger expression, the two forms can have different effects
- Because of their subtleties, the increment and decrement operators should be used with care

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- C provides *assignment operators* to simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```

Assignment Operators

- There are many assignment operators in C, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Boolean Expressions

- A Boolean expression is an expression that has relational and/or logical operators operating on Boolean variables.
- A Boolean expression evaluates to either *true* or *false*.

Boolean Operators

- The operators used with the `boolean` data type fall into two categories: relational operators and logical operators.
- There are six relational operators that compare values of other types and produce a `boolean` result:

`==`	Equals	`!=`	Not equals
`<`	Less than	`<=`	Less than or equal to
`>`	Greater than	`>=`	Greater than or equal to

For example, the expression `n <= 10` has the value `true` if `x` is less than or equal to 10 and the value `false` otherwise.

- There are also three logical operators:

`&&`	Logical AND	`p && q` means both `p` and `q`
`||`	Logical OR	`p || q` means either `p` or `q` (or both)
`!`	Logical NOT	`!p` means the opposite of `p`

Logical Operators

- C defines the following *logical operators*:

!	Logical NOT
&&	Logical AND
||	Logical OR

- Logical NOT is a unary operator (it operates on one operand)

- Logical AND and logical OR are binary operators (each operates on two operands)

Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some condition `a` is true, then `!a` is false; if `a` is false, then `!a` is true
- Logical expressions can be shown using a *truth table*

a	!a
true	false
false	true

Logical AND and Logical OR

- The *logical AND* expression

```
a && b
```

  is true if both `a` and `b` are true, and false otherwise

- The *logical OR* expression

```
a || b
```

  is true if `a` or `b` or both are true, and false otherwise

Logical Operators

- Expressions that use logical operators can form complex conditions

```
if (total < MAX+5 && !found)
   printf ("Processing…");
```

- **All logical operators have lower precedence than the relational operators**
- **Logical NOT has higher precedence than logical AND and logical OR**

Logical Operators

- A truth table shows all possible true-false combinations of the terms
- Since `&&` and `||` each have two operands, there are four possible combinations of conditions `a` and `b`

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Boolean Expressions

- Specific expressions can be evaluated using truth tables

`total < MAX`	`found`	`!found`	`total < MAX && !found`
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Boolean Expressions in C

- C does not have a Boolean data type.
- Therefore, C compares the values of variables and expressions against 0 (zero) to determine if they are true or false.
- If the value is 0 then the result is implicitly assumed to be false.
- If the value is different from 0 then the result is implicitly assumed to be true.
- Java have Boolean data types.

Equality Inequality Operator - Exercise

```
#include <stdio.h>

void main(void)
{
		int inum1 = 3, inum2 = 7, inum3 = 3;

		if(inum1 < inum2)
				printf("inum1 is less than inum2 (%d < %d)\n", inum1, inum2);
		else
				printf("inum1 is greater than inum2 (%d > %d)\n", inum1, inum2);

		if(inum2 > inum1)
				printf("inum2 is greater than inum1 (%d > %d)\n", inum2, inum1);
		else
				printf("inum2 is less than inum1 (%d < %d)\n", inum2, inum1);

		if(inum1 == inum3)
				printf("inum1 is equal to inum3 (%d == %d)\n", inum1, inum3);
		else
				printf("inum1 is not equal to inum3 (%d != %d)\n", inum1, inum3);

/* assignment operator, store/assign new value to inum1 variable */
		inum1 = inum2 + inum3;
		printf("inum1 = %d + %d = %d\n", inum2, inum3, inum1);

}
```

Multiplicative Additive Operator - Exercise

```
#include <stdio.h>

void main(void)
{
                /* declare 2 variables of type integer */
  int iaNumber = 400;
  int ianotherNum = 21;

  /* do the addition, subtraction, multiplication, division
     and modulus operations */
  printf("iaNumber = %d,  ianotherNum = %d\n", iaNumber,ianotherNum);
  printf("iaNumber + ianotherNum = %d\n", iaNumber + ianotherNum);
  printf("iaNumber - ianotherNum = %d\n", iaNumber - ianotherNum);
  printf("iaNumber * ianotherNum = %d\n", iaNumber * ianotherNum);
  /* using /, the fraction part will be truncated */
  printf("iaNumber / ianotherNum = %d\n", iaNumber / ianotherNum);
  /* cast to double to retain the fraction part */
  printf("(double)iaNumber / ianotherNum = %f\n", (double)iaNumber / ianotherNum);
  printf("iaNumber %% ianotherNum = %d\n", iaNumber % ianotherNum);
}
```