哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

高级语言程序设计
High-level Language Programming

Lecture 3 Operators

Yitian Shao (shaoyitian@hit.edu.cn)
School of Computer Science and Technology

Operators

Course Overview

- Basic arithmetic operators
- Operator precedence
- Assignment operators
- Increment and decrement
- Constants
  - Marco
  - Const

Try it yourself on https://www.onlinegdb.com

Why we need operators

- Need for computation (heavy work load)

- C++ Operators

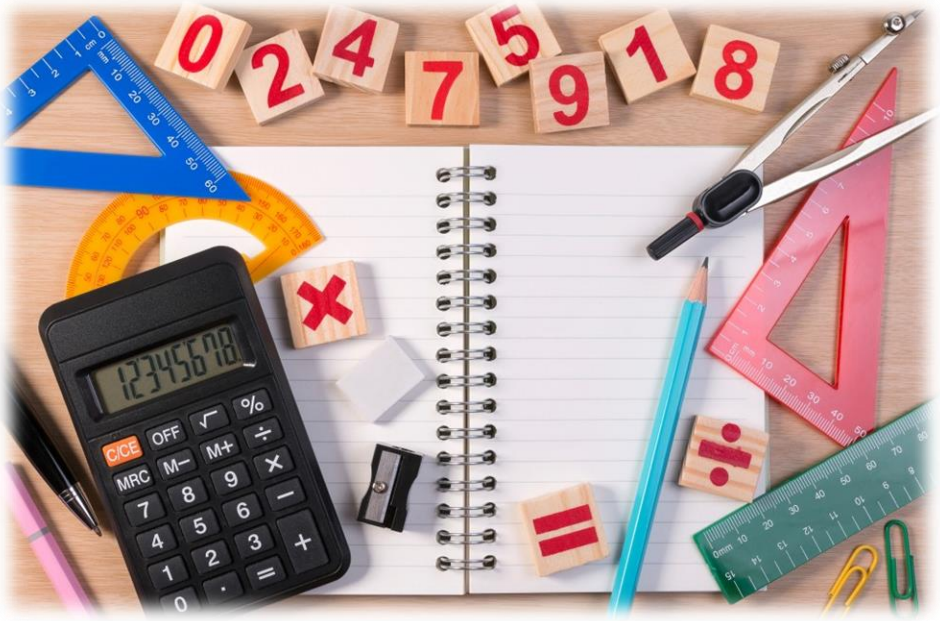
Try it yourself on https://www.onlinegdb.com

Type of operators

- Arithmetic operators
  - Unary arithmetic operators
- Assignment operators
  - Compound-assignment operators
- Logical and relational operators

  (Lecture 5)
- Member and pointer operators

  (Lecture 9 and 10)



Syntax

- Operator and operand

Example: 2 + 3 = ?

```
int a = 2, b = 3, c;
c = a + b;
```

Operator

```
int a = 2, b = 3, c;
c = a + b;
```

Operand

Arithmetic operators

Addition	**+**
Subtraction	**-**
Multiplication	*****
Division	**/**
Modulus	**%**

Division /

Example: 5 / 2 = ?

main.cpp

```cpp
#include <iostream>

int main()
{
    double a = 5, b = 2;
    std::cout << a/b << std::endl;

    return 0;
}
```

Ln: 5, Col: 11

Run | Share | Command Line Arguments

```
2.5
```

main.cpp

```cpp
#include <iostream>

int main()
{
    int a = 5, b = 2;
    std::cout << a/b << std::endl;

    return 0;
}
```

Ln: 5, Col: 22

Run | Share | Command Line Arguments

```
2
```

Why their results are different?

Division /

Example: 5 / 2 = ?

Beware of the data type!

main.cpp

```cpp
#include <iostream>

int main()
{
    double a = 5, b = 2;
    std::cout << a/b << std::endl;

    return 0;
}
```

Floating Division

Ln: 5, Col: 11

Run | Share | Command Line Arguments

```
2.5
```

main.cpp

```cpp
#include <iostream>

int main()
{
    int a = 5, b = 2;
    std::cout << a/b << std::endl;

    return 0;
}
```

Integer Division

Ln: 5, Col: 22

Run | Share | Command Line Arguments

```
2
```

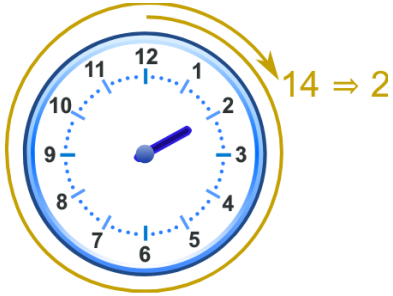
Both operands are floating-point

Both operands are integer

Modulus %

- Example of modulo operation

Time format conversion from 24:00H to 12:00H



$14 \% 12 = (12 + 2) \% 12 = 2$

Dividend = **Divisor** $*$ **K** + **Remainder** , where **K** must be an integer

14 o'clock becomes 2 o'clock

main.cpp

```
#include <iostream>

int main()
{
    int a = 14, b = 12;
    std::cout << a%b << std::endl;

    return 0;
}
```

Ln: 6, Col: 35

Run | Share | Command Line Arguments

```
2
```

Modulus %

- Note that the operands of modulus must be **integers**
- Negative operands

$$14 \,\%\, 5 = (5 * 2 + 4) \,\%\, 5 = 4$$

$$14 \,\%\, (-5) = \big((-5) * (-2) + 4\big) \,\%(-5) = 4$$

$$(-14)\%\, 5 = \big(5 * (-2) + (-4)\big) \,\%\, 5 = -4$$

Operator Precedence

- Arithmetic expression containing more than one operators

  Example: 5 + 8 / 2 = ?

Operator Precedence

- Arithmetic expression containing more than one operators

  Example: 5 + 8 / 2 = 9

- Computation order
  - Determined by the order of precedence
  - From left to right for operators with the same precedence

Operator Precedence

C++ Operator Precedence [cppreference.com]

Precedence	Operator	Description
3	+a	unary plus
	-a	unary minus
5	a*b	multiplication
	a/b	division
	a%b	modulus
6	a+b	addition
	a-b	subtraction
16	=	direct assignment

Precedence	Operator	Description
1	::	Scope resolution
2	a++ a--	Suffix/postfix increment and decrement
	type() *type*{}	Functional cast
	a()	Function call
	a[]	Subscript
	. ->	Member access
3	++a --a	Prefix increment and decrement
	+a -a	Unary plus and minus
	! ~	Logical NOT and bitwise NOT
	(*type*)	C-style cast
	*a	Indirection (dereference)
	&a	Address-of
	sizeof	Size-of[note 1]
	co_await	await-expression (C++20)
	new new[]	Dynamic memory allocation
	delete delete[]	Dynamic memory deallocation
4	.* ->*	Pointer-to-member
5	a*b a/b a%b	Multiplication, division, and remainder
6	a+b a-b	Addition and subtraction
7	<< >>	Bitwise left shift and right shift
8	<=>	Three-way comparison operator (since C++20)
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively
10	== !=	For equality operators = and ≠ respectively
11	a&b	Bitwise AND
12	^	Bitwise XOR (exclusive or)
13	|	Bitwise OR (inclusive or)
14	&&	Logical AND
15	||	Logical OR
16	a?b:c	Ternary conditional[note 2]
	throw	throw operator
	co_yield	yield-expression (C++20)
	=	Direct assignment (provided by default for C++ classes)
	+= -=	Compound assignment by sum and difference
	*= /= %=	Compound assignment by product, quotient, and remainder
	<<= >>=	Compound assignment by bitwise left shift and right shift
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR
17	,	Comma

Operator Precedence

Examples: a = ?

```
int a = 2 + 5 * 2;

int a = 2 + 2 * 5 % 3;

int a = - 2 * - 5;

int a = 2 - - 2 * 5;
```

Precedence	Operator	Description
3	+a	unary plus
	-a	unary minus
5	a*b	multiplication
	a/b	division
	a%b	modulus
6	a+b	addition
	a-b	subtraction
16	=	direct assignment

Operator Precedence

Examples: a = ?

```
int a = 2 + 5 * 2;
```

$$a = 2 + (5 * 2) = 2 + 10 = 12$$

```
int a = 2 + 2 * 5 % 3;
```

$$a = 2 + \big((2 * 5)\ \%\ 3\big) = 2 + (10\ \%\ 3) = 2 + 1 = 3$$

```
int a = - 2 * - 5;
```

$$a = (-2) * (-5) = 10$$

```
int a = 2 - - 2 * 5;
```

$$a = 2 - \big((-2) * 5\big) == 2 - (-10) = 12$$

Precedence	Operator	Description
3	+a	unary plus
	-a	unary minus
5	a*b	multiplication
	a/b	division
	a%b	modulus
6	a+b	addition
	a-b	subtraction
16	=	direct assignment

Compound assignment

- Simple assignment

```
int a;
a = 100;
```

- Multiple assignment

```
int a, b, c;
a = b = c = 100;
a = (b = (c = 100));
```

Associativity: right-to-left

- Compound assignment

```
int a = 100;
a += 5;
```

Compound assignment

- Add operands using simple assignment

```
int a = 100;
a = a + 5;
```

- Add operands using compound assignment

```
int a = 100;
a += 5;
```

Shorthand assignment operator

Compound assignment

- Note that **no space** is allowed between the two symbols of a compound operator (for example, no space between '+' and '=')

Compound operator	Equivalent arithmetic operation
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

15	`||`	Logical OR
	`a?b:c`	Ternary conditional[note 2]
	`throw`	throw operator
	`co_yield`	yield-expression (C++20)
16	`=`	Direct assignment (provided by default for C++ classes)
	`+=` `-=`	Compound assignment by sum and difference
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR
17	`,`	Comma

Precedence

- Using compound operator to improve code **readability** and **compile efficiency**

Increment and Decrement

- Unary operators
  (works on a single operand)

a = a + 1

- Increment
  a++

a = a - 1

- Decrement
  a--

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	
2	`a++ a--`	Suffix/postfix increment and decrement	left-to-right
	`type``()` *`type`*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*`type`*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	

Increment and Decrement

- Unary operators
  (works on a single operand)

  a = a + 1

- Increment
  a++ ++a

  a = a - 1

- Decrement
  a-- --a

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	
2	`a++ a--`	Suffix/postfix increment and decrement	left-to-right
	`type``()` *`type`*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*`type`*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	

Increment and Decrement

- Post- and pre-increment
  a++ ++a

- Post- and pre-decrement
  a-- --a

Value increased/decreased by 1 **after** the other operations are done

Precedence	Operator	Description
1	`::`	Scope resolution
2	`a++` `a--`	Suffix/postfix increment and decrement
	`type``()` *`type`*`{}`	Functional cast
	`a()`	Function call
	`a[]`	Subscript
	`.` `->`	Member access
3	`++a` `--a`	Prefix increment and decrement
	`+a` `-a`	Unary plus and minus
	`!` `~`	Logical NOT and bitwise NOT
	`(`*`type`*`)`	C-style cast
	`*a`	Indirection (dereference)
	`&a`	Address-of
	`sizeof`	Size-of[note 1]
	`co_await`	await-expression (C++20)
	`new` `new[]`	Dynamic memory allocation
	`delete` `delete[]`	Dynamic memory deallocation

Increment and Decrement

- Post- and pre-increment
  a++ ++a

- Post- and pre-decrement
  a-- --a

Value increased/decreased by 1 **before** executing the other operations

Precedence	Operator	Description
1	`::`	Scope resolution
2	`a++` `a--`	Suffix/postfix increment and decrement
	`type``()` *`type`*`{}`	Functional cast
	`a()`	Function call
	`a[]`	Subscript
	`.` `->`	Member access
3	`++a` `--a`	Prefix increment and decrement
	`+a` `-a`	Unary plus and minus
	`!` `~`	Logical NOT and bitwise NOT
	`(`*`type`*`)`	C-style cast
	`*a`	Indirection (dereference)
	`&a`	Address-of
	`sizeof`	Size-of[note 1]
	`co_await`	await-expression (C++20)
	`new` `new[]`	Dynamic memory allocation
	`delete` `delete[]`	Dynamic memory deallocation

Increment and Decrement

- Pre-increment

```
int a = 3;
int b = ++a;
```

a = 4, b = 4

Value of **a** increased by 1 **before** assigning it to **b**

- Post-increment

```
int a = 3;
int b = a++;
```

a = 4, b = 3

Value of **a** increased by 1 **after** assigning it (originally **a** = 3) to **b**

Precedence	Operator	Description
1	`::`	Scope resolution
2	`a++ a--`	Suffix/postfix increment and decrement
	`type()` *`type{}`*	Functional cast
	`a()`	Function call
	`a[]`	Subscript
	`. ->`	Member access
3	`++a --a`	Prefix increment and decrement
	`+a -a`	Unary plus and minus
	`! ~`	Logical NOT and bitwise NOT
	`(`*`type`*`)`	C-style cast
	`*a`	Indirection (dereference)
	`&a`	Address-of
	`sizeof`	Size-of[note 1]
	`co_await`	await-expression (C++20)
	`new new[]`	Dynamic memory allocation
	`delete delete[]`	Dynamic memory deallocation

Exercise (**Try it yourself!**)

```
int a = 2;
int b = ++a - 2;
```

a = ?
b = ?

```
int a = 2;
int b = a++ - 2;
```

a = ?
b = ?

```
int a = 2;
int b = a+ + - 2;
```

a = ?
b = ?

a++ and a–
Value increased/decreased by 1 after the other operations are done

++a and –a
Value increased/decreased by 1 before executing the other operations

Precedence	Operator
2	a++
	a--
3	++a
	--a
	+a
	-a
5	a*b
	a/b
	a%b
6	a+b
	a-b
16	=
	+=
	-=
	*=
	/=
	%=

Exercise (Solution)

```
int a = 2;
int b = ++a - 2;
```

(a = 2+1)

b = (++a) – 2 = 3 – 2 = 1

```
int a = 2;
int b = a++ - 2;
```

b = (a++) – 2 = 2 – 2 = 0

a = 3

```
int a = 2;
int b = a+ + - 2;
```

b = a + (+(-2)) = a + (-2) = a – 2 = 2 – 2 = 0

a = 2

a++ and a–

Value increased/decreased by 1 after the other operations are done

++a and –a

Value increased/decreased by 1 before executing the other operations

Precedence	Operator
2	a++
	a--
3	++a
	--a
	+a
	-a
5	a*b
	a/b
	a%b
6	a+b
	a-b
16	=
	+=
	-=
	*=
	/=
	%=

Increment and Decrement

- Example usage

main.cpp

```cpp
#include <iostream>

int main()
{
    for(int i = 0; i < 5; ++i)
    {
        std::cout << i << std::endl;
    }

    return 0;
}
```

Ln: 1, Col: 20

Run | Share | Command Line Arguments

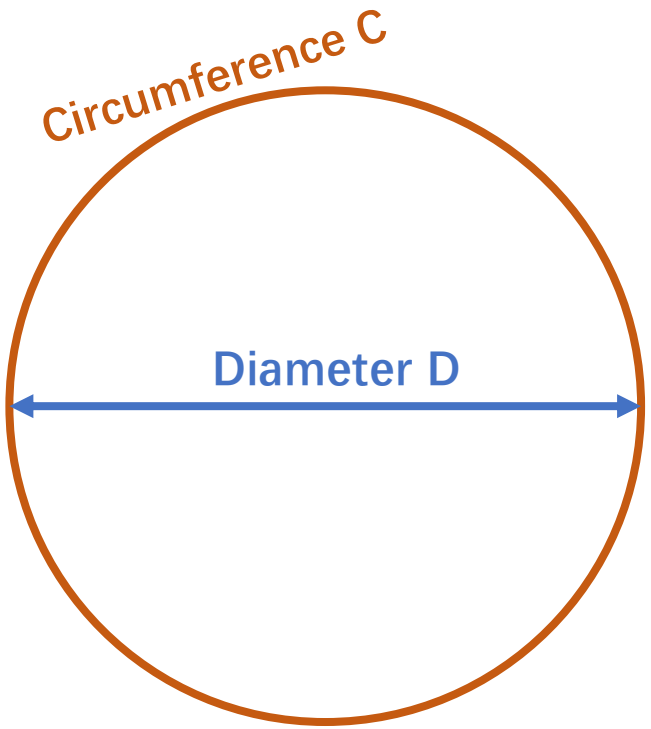
```
0
1
2
3
4
```

- **Avoid** using **more than one increment/decrement operations** in the **same line** of code

Constants

Consider the task: Writing a program that can calculate a tire circumference based on measured diameter

$C = \pi * D$ $\pi = 3.1415926$



main.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Circumference of tire 1 = " << 3.1415926 * 19 << std::endl;
    std::cout << "Circumference of tire 2 = " << 3.1415926 * 24 << std::endl;
    std::cout << "Circumference of tire 3 = " << 3.1415926 * 28 << std::endl;

    return 0;
}
```

Ln: 1, Col: 1

Run Share Command Line Arguments

```
Circumference of tire 1 = 59.6903
Circumference of tire 2 = 75.3982
Circumference of tire 3 = 87.9646
```

Constants

- Should avoid creating magic numbers
  - Code readability
  - Prone to mistakes, especially when the numbers need to be changed

How to avoid magic numbers?
Define constants: **marco** or **const variable**

main.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Circumference of tire 1 = " << 3.1415926 * 19 << std::endl;
    std::cout << "Circumference of tire 2 = " << 3.1415926 * 24 << std::endl;
    std::cout << "Circumference of tire 3 = " << 3.1415926 * 28 << std::endl;

    return 0;
}
```

Ln: 1, Col: 1

Avoid magic numbers in your code!

Run | Share | Command Line Arguments

```
Circumference of tire 1 = 59.6903
Circumference of tire 2 = 75.3982
Circumference of tire 3 = 87.9646
```

Macro constants

- Syntax

#define MACRO_NAME replacement_value

```
#define PI 3.1415926
```

- In C++, the preprocessor directives are special commands that are used to instruct the preprocessor. It begins with a **'#'** symbol and tells the preprocessor to the modify source code before compilation.

- When the preprocessor encounters the macro name in our code, it will be replaced by the specified value

- Generally we capitalize the entire marco name

Macro constants

preprocessor replace macro with the specified value

main.cpp

```cpp
#include <iostream>

#define PI 3.1415926
#define D1 19
#define D2 24
#define D3 28

int main()
{
    std::cout << "Circumference of tire 1 = " << PI * D1 << std::endl;
    std::cout << "Circumference of tire 2 = " << PI * D2 << std::endl;
    std::cout << "Circumference of tire 3 = " << PI * D3 << std::endl;

    return 0;
}
```

Ln: 1, Col: 20

Run Share Command Line Arguments

```
Circumference of tire 1 = 59.6903
Circumference of tire 2 = 75.3982
Circumference of tire 3 = 87.9646
```

main.cpp

```cpp
#include <iostream>

int main()
{
    std::cout << "Circumference of tire 1 = " << 3.1415926 * 19 << std::endl;
    std::cout << "Circumference of tire 2 = " << 3.1415926 * 24 << std::endl;
    std::cout << "Circumference of tire 3 = " << 3.1415926 * 28 << std::endl;

    return 0;
}
```

Equivalent code

Ln: 1, Col: 1

Run Share Command Line Arguments

```
Circumference of tire 1 = 59.6903
Circumference of tire 2 = 75.3982
Circumference of tire 3 = 87.9646
```

Macro constants

main.cpp

```cpp
#include <iostream>

#define PI 3.1415926
#define D1 19
#define D2 24
#define D3 28

int main()
{
    std::cout << "Circumference of tire 1 = " << PI * D1 << std::endl;
    std::cout << "Circumference of tire 2 = " << PI * D2 << std::endl;
    std::cout << "Circumference of tire 3 = " << PI * D3 << std::endl;

    return 0;
}
```

Where to define them

Ln: 1, Col: 20

Run | Share | Command Line Arguments

```
Circumference of tire 1 = 59.6903
Circumference of tire 2 = 75.3982
Circumference of tire 3 = 87.9646
```

main.cpp

```cpp
#include <iostream>

#define PI 3.1415926;
#define D1 19;
#define D2 24;
#define D3 28;

int main()
```

Must not use semicolons

Const variable

- Can use the **const** keyword instead of the #define preprocessor directive **to define constant values.**
- The **const** keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it

Can define the data type of the constant

Values defined with const are subject to type checking

main.cpp

```cpp
#include <iostream>

int main()
{
    const double PI = 3.1415926;
    const int D1 = 19;
    const int MESSAGE_LENGTH = 27;
    const char MESSAGE_STR[MESSAGE_LENGTH] = "Circumference of tire 1 = ";

    std::cout << MESSAGE_STR << PI * D1 << std::endl;

    return 0;
}
```

Ln: 1, Col: 20

Run | Share | Command Line Arguments

```
Circumference of tire 1 = 59.6903
```

Const variable

Note that you can specify the size of an array with a const variable (or marco)

main.cpp

```cpp
#include <iostream>

int main()
{
    const double PI = 3.1415926;
    const int D1 = 19;
    const int MESSAGE_LENGTH = 27;
    const char MESSAGE_STR[MESSAGE_LENGTH] = "Circumference of tire 1 = ";

    std::cout << MESSAGE_STR << PI * D1 << std::endl;

    return 0;
}
```

Ln: 1, Col: 20

Run | Share | Command Line Arguments

```
Circumference of tire 1 = 59.6903
```

Input/Output (I/O)

- Output
  std::cout << *a_string_constant*
  std::cout << *a_variable*

  - Concatenate multiple outputs

  - Change line:
  std::endl

- Input
  std::cin >> *a_variable*

main.cpp

```cpp
#include<iostream>

[redacted]

int main()
{
    int age;
    std::cout << "Please input your age and press Enter: " << std::endl;

    std::cin >> age;

    std::cout << "Your age is: " << age << std::endl;
}
```

Ln: 14, Col: 1

Run | Share | Command Line Arguments

```
Please input your age and press Enter:
12
Your age is: 12
```

A Hard Challenge (No grading)

(This challenge is not part of the homework! Try it yourself! Not required for any submission)

Exercise (**Try it yourself!**)

```
int a = 3;
a += a -= a * -a;
```

a = ?

Precedence	Operator	Associativity
3	+a	right-to-left
	-a	
5	a*b	left-to-right
	a/b	
	a%b	
6	a+b	left-to-right
	a-b	
16	=	right-to-left
	+=	
	-=	
	*=	
	/=	
	%=	

Exercise (Solution)

```
int a = 3;
a += a -= a * -a;
```

Value stored in a

$$a \mathrel{+}= (a \mathrel{-}= (a * (-a)))$$

3

$$a \mathrel{+}= (a \mathrel{-}= (a * (-3)))$$

3

$$a \mathrel{+}= (a \mathrel{-}= (3 * (-3)))$$

3

$$\boxed{\begin{array}{c} a \mathrel{+}= (a \mathrel{-}= (-9)) \\ a \mathrel{+}= (a = a - (-9)) \end{array}}$$

3

a -= (-9) is equivalent to a = a – (-9)

$$a \mathrel{+}= (a = 3 - (-9))$$

12

Value of a is updated! Because of the '=' operation

$$\boxed{\begin{array}{c} a \mathrel{+}= 12 \\ a = a + 12 \end{array}}$$

12

a += 12 is equivalent to a = a + 12

$$a = 24$$

24

Value updated!

Precedence	Operator	Associativity
3	+a	right-to-left
	-a	
5	a*b	left-to-right
	a/b	
	a%b	
6	a+b	left-to-right
	a-b	
16	=	right-to-left
	+=	
	-=	
	*=	
	/=	
	%=	

HOMEWORK

Homework 3

- 1. Convert the following mathematical equations into valid C++ statements :

(a) $m = \dfrac{y_1 - y_2}{x_1 - x_2}$

(b) $y = mx + c$

(c) $a = \dfrac{b}{c} - \dfrac{d}{e}$

(d) $C = \dfrac{5(F - 32)}{9}$

(e) $s = ut + \frac{1}{2}at^2$

- 2.Assuming the following, `int a = 1, b = 2, c = 3 ;` what is the value of a, b and c after each of the following statements?

(a) `a += b ;`

(b) `a /= 3 ;`

(c) `a *= c ;`

(d) `a %= 2 ;`

(e) `a += b+1 ;`

(f) `a += ++b ;`

Homework 3

- 3.Assuming the following, `int a = 12, b = 0, c = 3, d ;` what is the value of a, b, c and d after each of the following statements?

(a) `a++ ;`

(b) `b-- ;`

(c) `d = ++c ;`

(d) `d = c-- ;`

(e) `d = a++ - 2 ;`

(f) `d = a++ + b++ - c-- ;`

Homework 3

- 4. Place parentheses around the following expressions to indicate the order of evaluation :

```
(a) a = 1 - 2 * 3 + 4 / 5 ;
(b) a = 5 % b + c - d / 10 ;
(c) a = ++b * -10 / 5 ;
```

- 5. Write a program to compute (a) the volume and (b) the surface area of a box with a height of 10 cm, a length of 11.5 cm, and a width of 2.5 cm.