C++ Program Design -- Operators

Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

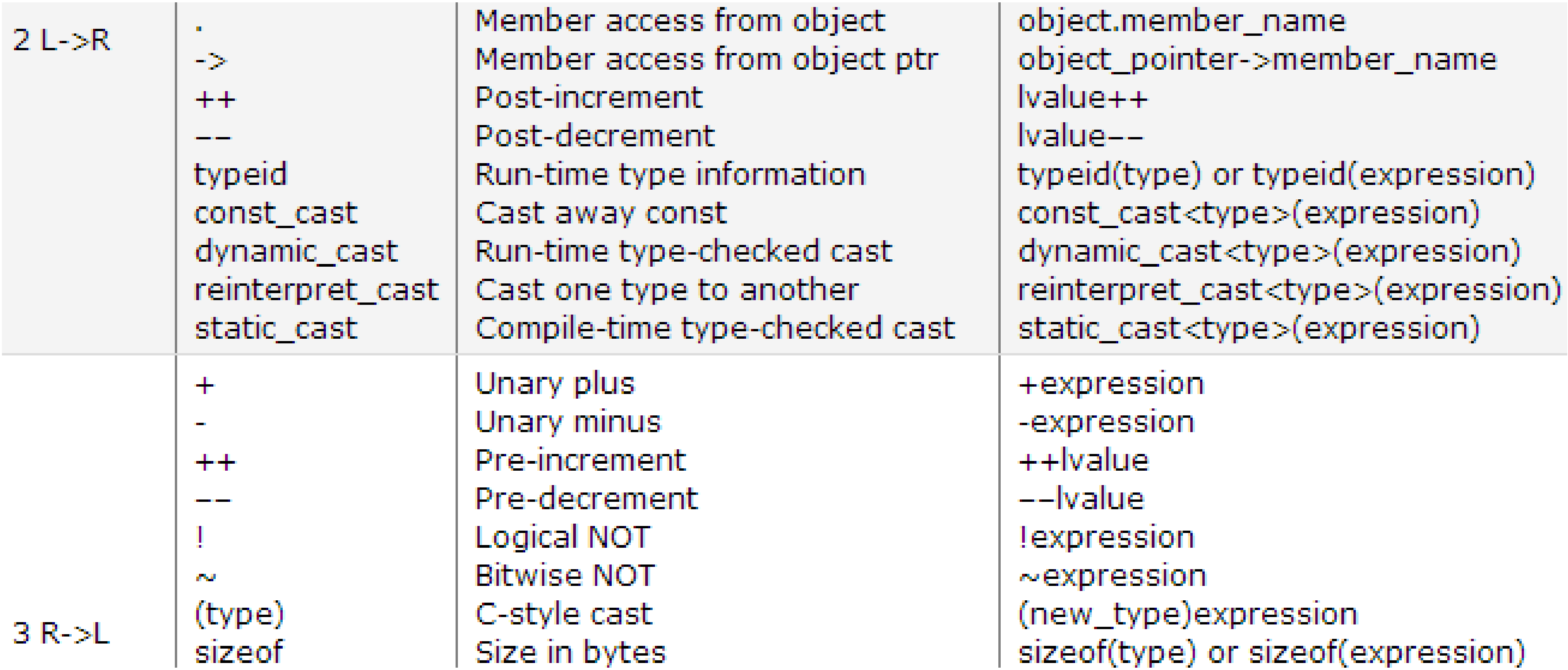
Operator precedence and associativity

Operator precedence and associativity

- 4 + 2 * 3 => 4 + (2 * 3)
- **Precedence**: The order in which operators are evaluated in a compound expression
  - Obey normal mathematical precedence rules
- **Associativity**:
- If two operators with the same precedence level are adjacent, it tell compiler whether to evaluate the operators from left to right or from right to left.
- 3 * 4 / 2 => (3 * 4)/ 2

Table of operators

- Precedence level 1 is the highest precedence level, and level 17 is the lowest. Operators with a higher precedence level get evaluated first.
- L->R means left to right associativity.
- R->L means right to left associativity.

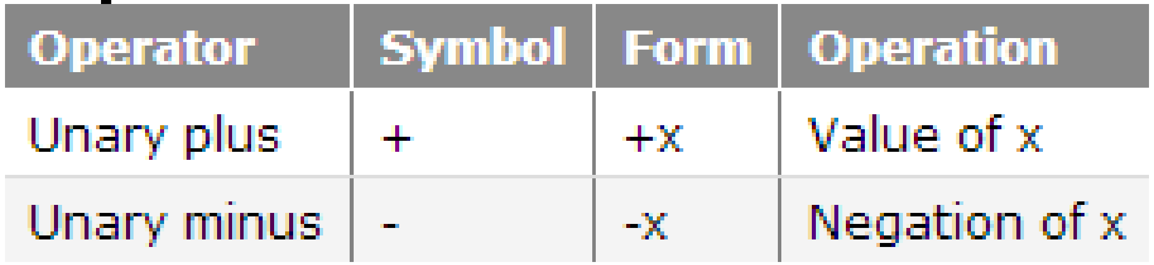


2 L->R	.	Member access from object	object.member_name
	->	Member access from object ptr	object_pointer->member_name
	++	Post-increment	lvalue++
	--	Post-decrement	lvalue--
	typeid	Run-time type information	typeid(type) or typeid(expression)
	const_cast	Cast away const	const_cast<type>(expression)
	dynamic_cast	Run-time type-checked cast	dynamic_cast<type>(expression)
	reinterpret_cast	Cast one type to another	reinterpret_cast<type>(expression)
	static_cast	Compile-time type-checked cast	static_cast<type>(expression)
3 R->L	+	Unary plus	+expression
	-	Unary minus	-expression
	++	Pre-increment	++lvalue
	--	Pre-decrement	--lvalue
	!	Logical NOT	!expression
	~	Bitwise NOT	~expression
	(type)	C-style cast	(new_type)expression
	sizeof	Size in bytes	sizeof(type) or sizeof(expression)

Arithmetic operators

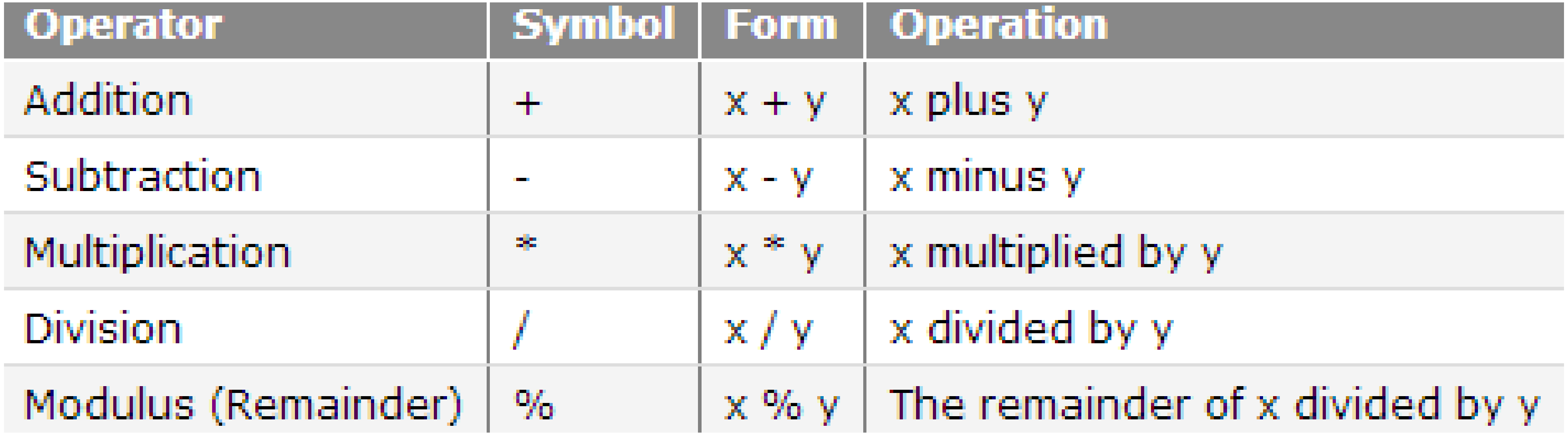
Arithmetic operators

- **Unary arithmetic operators**



Operator	Symbol	Form	Operation
Unary plus	+	+x	Value of x
Unary minus	-	-x	Negation of x

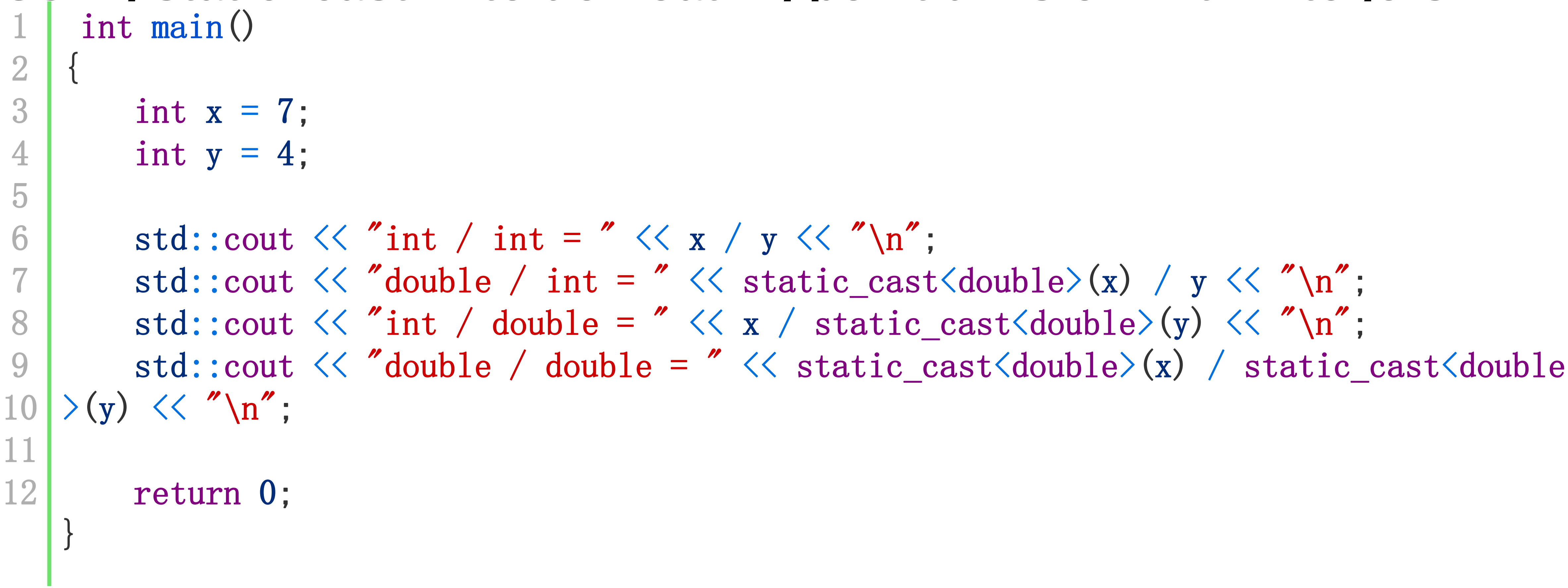
- **Binary arithmetic operators**



Operator	Symbol	Form	Operation
Addition	+	x + y	x plus y
Subtraction	-	x - y	x minus y
Multiplication	*	x * y	x multiplied by y
Division	/	x / y	x divided by y
Modulus (Remainder)	%	x % y	The remainder of x divided by y

Integer and floating point division

- two different “modes”
    - fraction is dropped: 7 / 4 = 1
    - floating point division: 7.0 / 3 = 2.333, 7 / 3.0 = 2.333
- **Using static_cast<> to do floating point division with integers**



```
 int main()
{
    int x = 7;
    int y = 4;

    std::cout << "int / int = " << x / y << "\n";
    std::cout << "double / int = " << static_cast<double>(x) / y << "\n";
    std::cout << "int / double = " << x / static_cast<double>(y) << "\n";
    std::cout << "double / double = " << static_cast<double>(x) / static_cast<double
>(y) << "\n";

    return 0;
}
```

Modulus (remainder)

- 7 / 4 = 1 remainder 3, thus 7 % 4 = 3

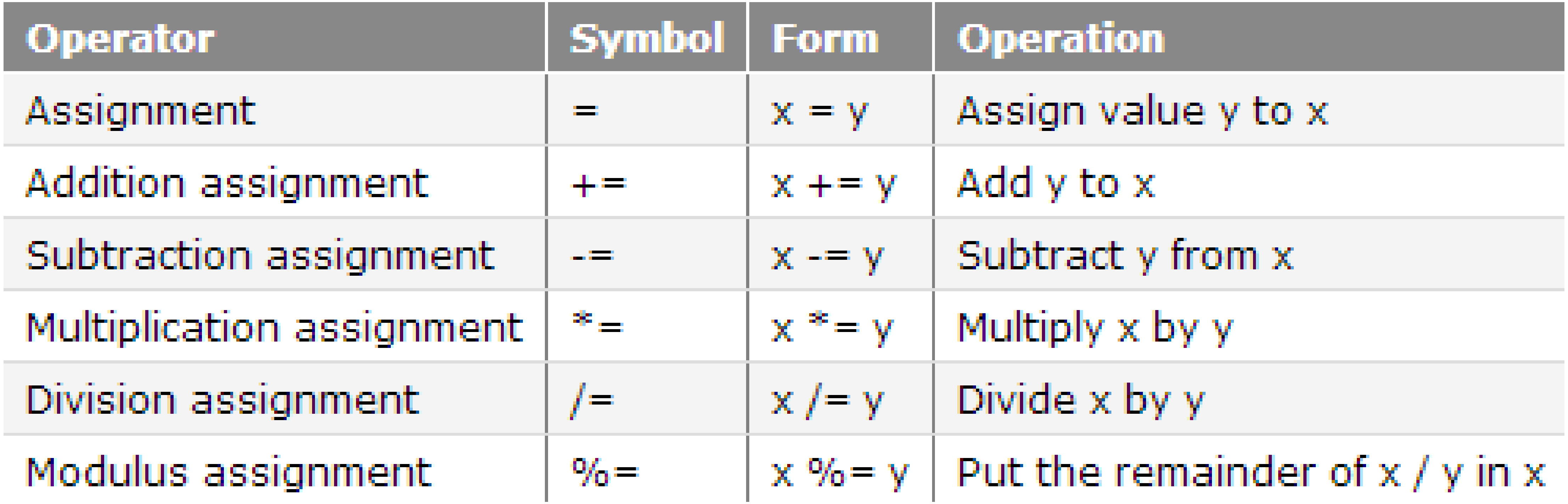
```
int main()
{
    int count = 1; //count holds the current number to print, start at 1
    // Loop continually until we pass number 100
    while (count <= 100)
    {
        std::cout << count << ” ”; // print the current number

        // if count is evenly divisible by 20, print a new line
        if (count % 20 == 0)
            std::cout << ”\n”;

        count = count + 1; // go to next number
    } // end of while

    return 0;
} // end of main()
```

integer division and modulus with negative numbers prior to C++11

- Before: -5 / 2 = -3 or -2, depending on the compiler
- Now: truncate towards 0 => -2

- Before: -5 % 2 = 1 or -1.
- Now: a % b always resolves to the sign of a.

Arithmetic assignment operators



Operator	Symbol	Form	Operation
Assignment	=	x = y	Assign value y to x
Addition assignment	+=	x += y	Add y to x
Subtraction assignment	-=	x -= y	Subtract y from x
Multiplication assignment	*=	x *= y	Multiply x by y
Division assignment	/=	x /= y	Divide x by y
Modulus assignment	%=	x %= y	Put the remainder of x / y in x

- x = x + 5 ⬄ x += 5

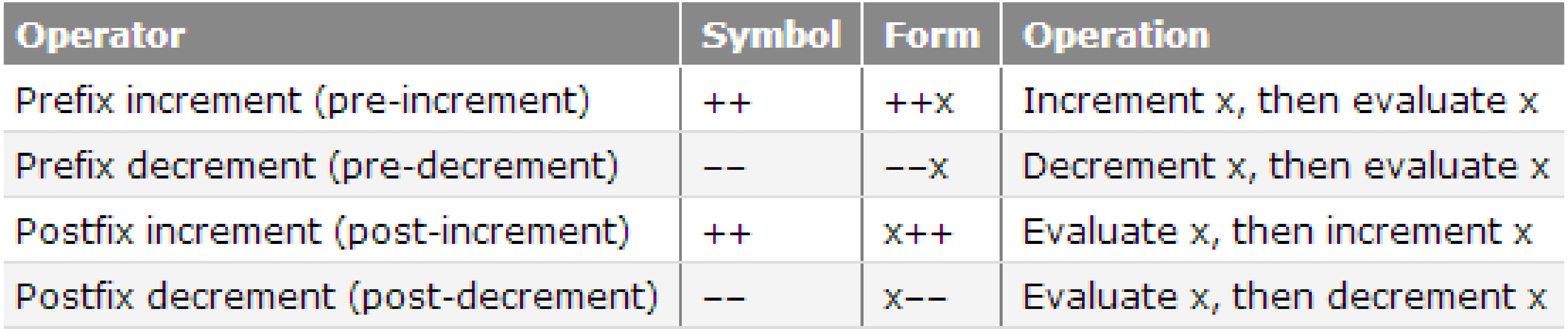
Quiz

- What does the following expression evaluate to? 6 + 5 * 4 % 3
  - Because * and % have higher precedence than +
  - 6 + (5 * 4 % 3)
  - * and % have the same precedence, then look at associativity: left to right
  - 6 + ((5 * 4) % 3).

- Write a program that asks the user to input an integer, and tells the user whether the number is even or odd.
  - Write a function called isEven() that returns true if an integer passed to it is even.
  - Use the modulus operator to test whether the integer parameter is even.

Increment/decrement operators, and side effects

Increment/decrement operators, and side effects



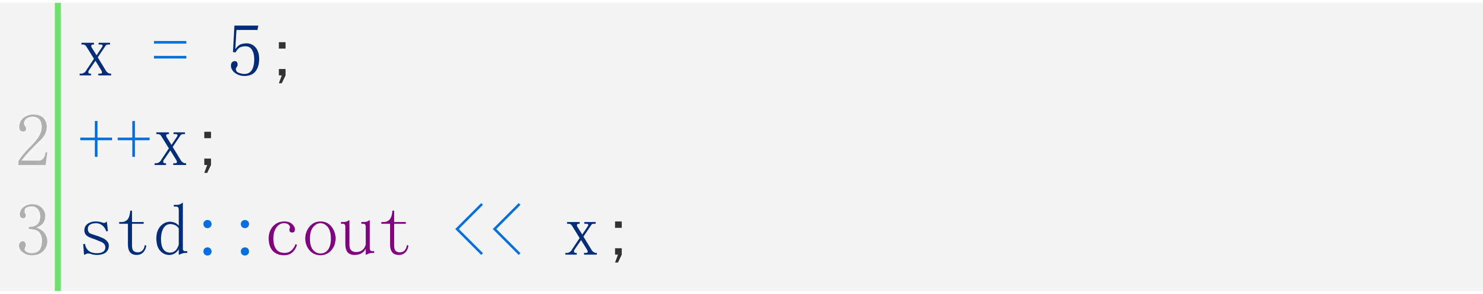
Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then evaluate x
Prefix decrement (pre-decrement)	--	--x	Decrement x, then evaluate x
Postfix increment (post-increment)	++	x++	Evaluate x, then increment x
Postfix decrement (post-decrement)	--	x--	Evaluate x, then decrement x

```
int x = 5;
int y = ++x; // x is now equal to 6, and 6 is assigned to y
```

```
int x = 5;
int y = x++; // x is now equal to 6, and 5 is assigned to y
```

Side effects

- A function or expression is said to have a **side effect** if it modifies some state (e.g. any stored information in memory)



```
x = 5;
++x;
std::cout << x;
```

- side effects can also lead to unexpected results:

```
int main(){
    int x = 1;
    x = x++;
    std::cout << x;

    return 0;
}
```

Undefined
C++ does not define the order of = and ++

side effects can also lead to unexpected results

- C++ does not define the order in which function arguments are evaluated.

```
int add(int x, int y){    return x + y;}

int main(){
    int x = 5;
    int value = add(x, ++x); // is this 5 + 6, or 6 + 6?  It d
epends on what order your compiler evaluates the function argu
ments in

    std::cout << value; // value could be 11 or 12, depending
on how the above line evaluates!
    return 0;
}
```

- ***Rule: Don’t use a variable that has a side effect applied to it more than once in a given statement.***

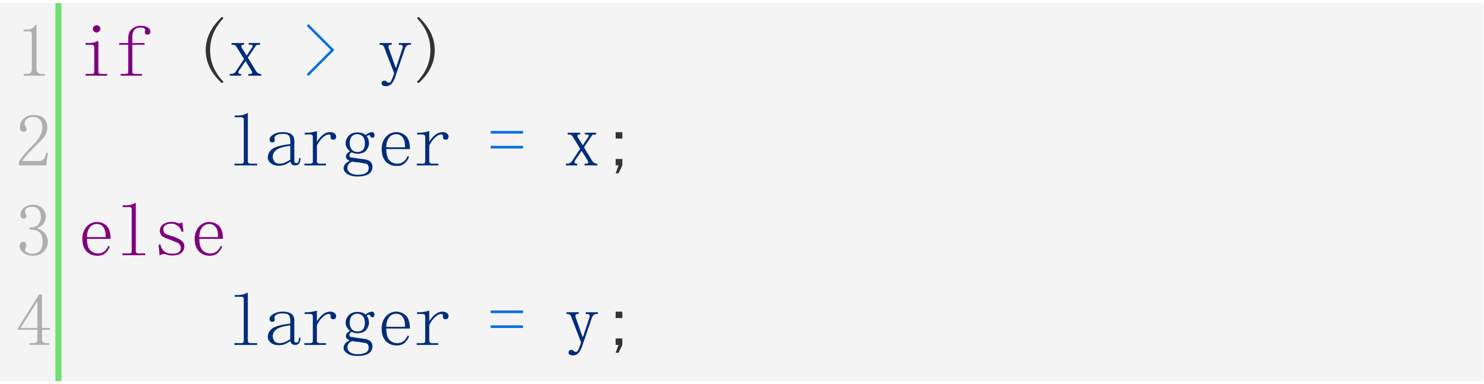
Conditional operator

Conditional operator

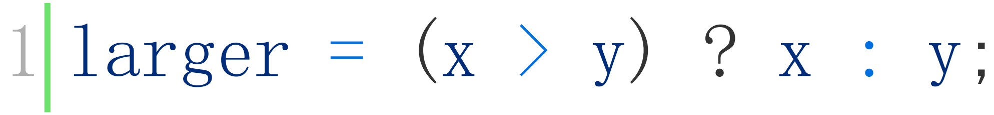
- ?: operator provides a shorthand method for doing a particular type of if/else statement.

  (condition) ? expression : other_expression;

- easier to read than if else



```
if (x > y)
    larger = x;
else
    larger = y;
```



```
larger = (x > y) ? x : y;
```

Precedence of Conditional operator

- put the conditional part inside of parenthesis,
  - easier to read
  - make sure the precedence is correct

```
larger = (x > y) ? x : y;
```

- ?: operator has a very low precedence

```
cout << (x > y) ? x : y; =>
(cout << (x > y)) ? x : y;
Print 0 or 1;
return x since cout << (x>y) is amlost always successful.
cout << ((x > y) ? x : y);
```

Conditional operator

- *Rule: Only use the conditional operator for simple conditionals where it enhances readability.*

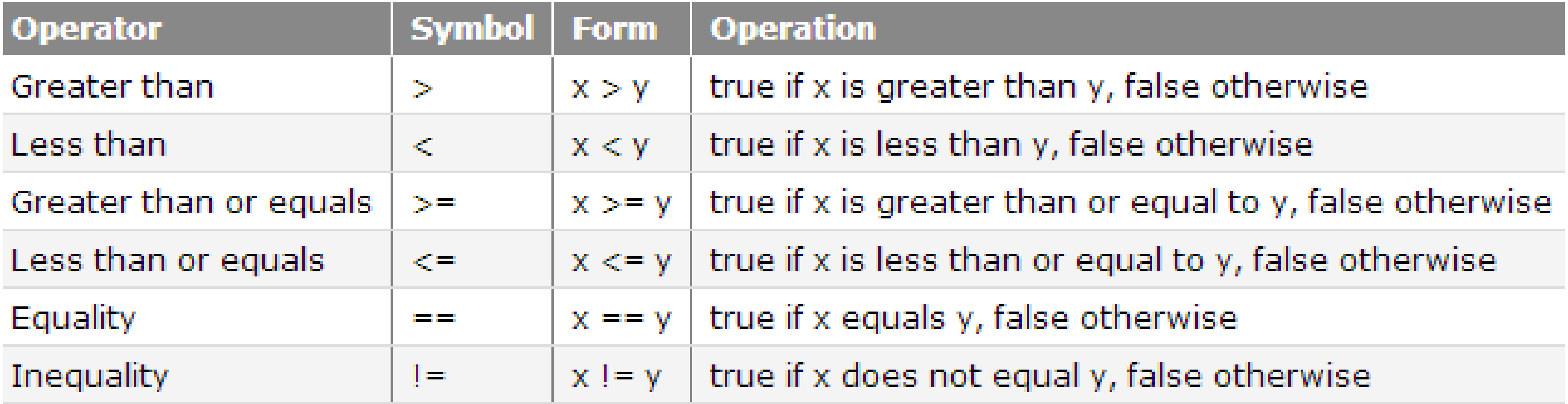
- **The conditional operator evaluates as an expression**

```
bool inBigClassroom = false;
const int classSize = inBigClassroom ? 30 : 20;
```

There’s **no satisfactory if/else statement for this**, since const variables must be initialized when defined, and the initializer can’t be a statement.

Relational operators

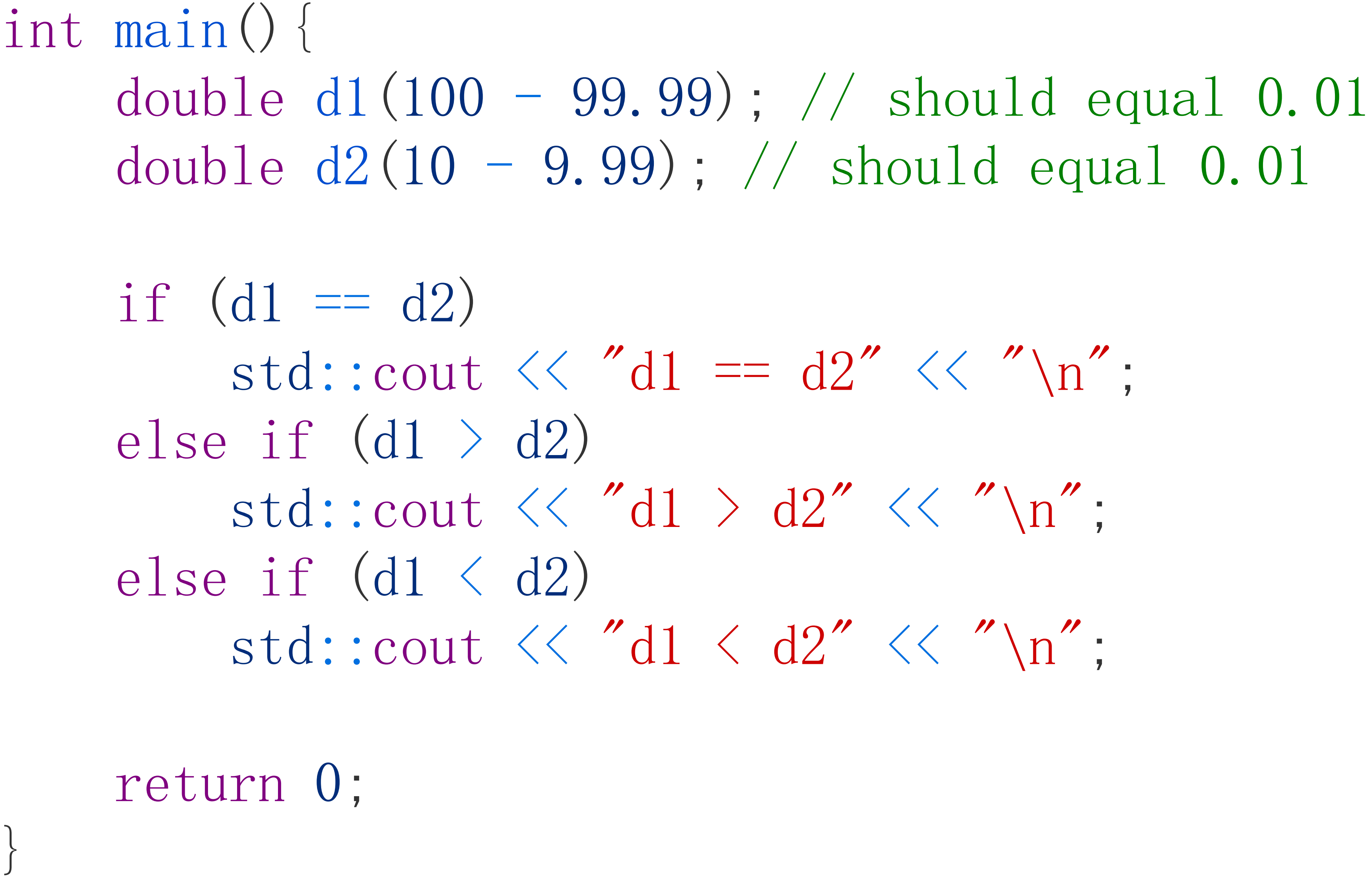
Relational operators (comparisons)



Operator	Symbol	Form	Operation
Greater than	>	x > y	true if x is greater than y, false otherwise
Less than	<	x < y	true if x is less than y, false otherwise
Greater than or equals	>=	x >= y	true if x is greater than or equal to y, false otherwise
Less than or equals	<=	x <= y	true if x is less than or equal to y, false otherwise
Equality	==	x == y	true if x equals y, false otherwise
Inequality	!=	x != y	true if x does not equal y, false otherwise

Comparison of floating point values

- Print d1>d2
- rounding errors
- Dangerous



```cpp
int main(){
    double d1(100 - 99.99); // should equal 0.01
    double d2(10 - 9.99); // should equal 0.01

    if (d1 == d2)
        std::cout << "d1 == d2" << "\n";
    else if (d1 > d2)
        std::cout << "d1 > d2" << "\n";
    else if (d1 < d2)
        std::cout << "d1 < d2" << "\n";

    return 0;
}
```

d1 = 0.010000000000005116 and d2 = 0.0099999999999997868.

floating point comparisons 1

```
#include <cmath> // for fabs()
bool isAlmostEqual(double a, double b, double epsilon)
{
    // if the distance between a and b is less than epsilon,
then a and b are "close enough"
    return fabs(a - b) <= epsilon;
}
```

- An epsilon of 0.00001 is good for inputs around 1.0, too big for numbers around 0.0000001, and too small for numbers like 10,000.

floating point comparisons 2

- Donald Knuth, “The Art of Computer Programming, Volume II: Seminumerical Algorithms (Addison-Wesley, 1969)”:

```
#include <cmath>
// return true if the difference between a and b is within epsilon percent of the larger of a and b
bool approximatelyEqual(double a, double b, double epsilon){
    return fabs(a - b) <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * epsilon);
}
```

floating point comparisons 3

```
int main(){
    // a is really close to 1.0, but has rounding errors, so it’s slightly smaller than 1.0
    double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

    // First, let’s compare a (almost 1.0) to 1.0.
    std::cout << approximatelyEqual(a, 1.0, 1e-8) << ”\n”;

    // Second, let’s compare a-1.0 (almost 0.0) to 0.0
    std::cout << approximatelyEqual(a-1.0, 0.0, 1e-8) << ”\n”;
}
```

```
1
0
```

- it is not perfect, especially as the numbers approach zero

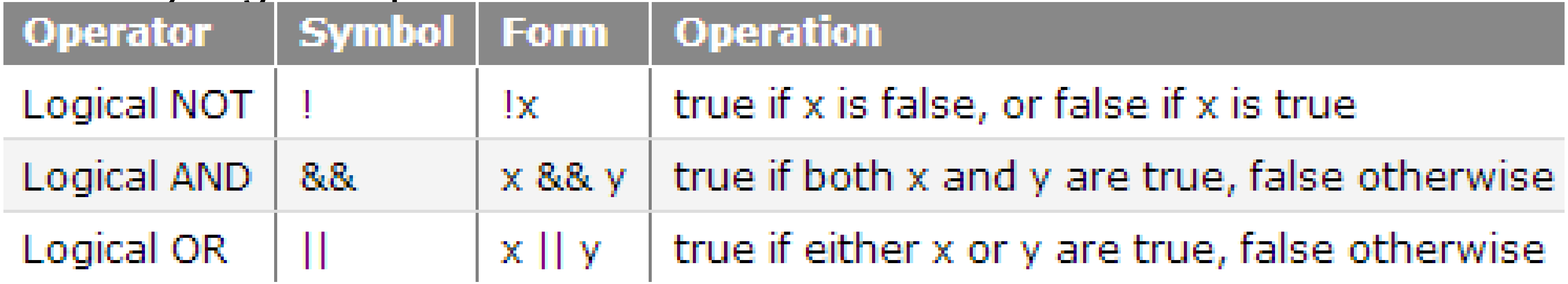
floating point comparisons 4

```
// return true if the difference between a and b is less than absEpsilon, or within relEpsilon percent of the larger of a and b
bool approximatelyEqualAbsRel(double a, double b, double absEpsilon, double relEpsilon){
    // Check if the numbers are really close -- needed when comparing numbers near zero.
    double diff = fabs(a - b);
    if (diff <= absEpsilon)
        return true;
    // Otherwise fall back to Knuth’s algorithm
    return diff <= ( (fabs(a) < fabs(b) ? fabs(b) : fabs(a)) * relEpsilon);
}
```

Logical operators

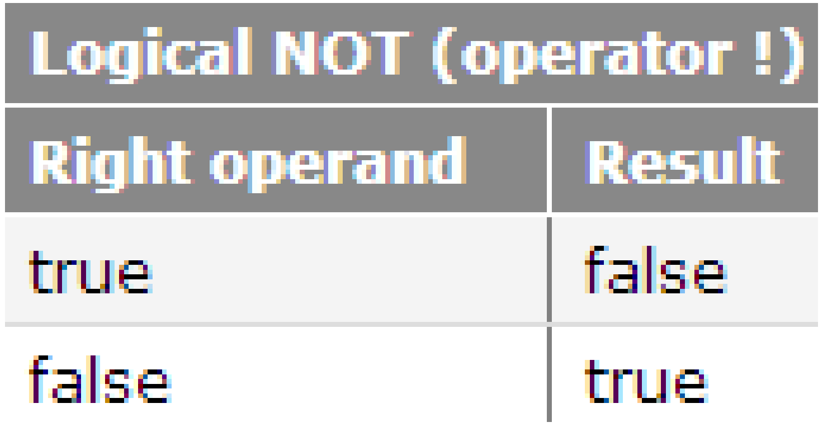
Logical operators

- 3 binary logical operators



Operator	Symbol	Form	Operation
Logical NOT	!	!x	true if x is false, or false if x is true
Logical AND	&&	x && y	true if both x and y are true, false otherwise
Logical OR	||	x || y	true if either x or y are true, false otherwise

- 1 unary logical operator



Logical NOT (operator !)	
Right operand	**Result**
true	false
false	true

Logical NOT has a very high level of precedence

```
int x = 5;
int y = 7;

if (! x == y)
   cout << ″x does not equal y″;
else
   cout << ″x equals y″;
```

- Reminder: any non-zero integer value evaluates to *true* when used in a boolean context.
- “x equals y”!

Logical NOT has a very high level of precedence

- *Rule: If logical NOT is intended to operate on the result of other operators, the other operators and their operands need to be enclosed in **parenthesis**.*

- *Rule: It’s a good idea to always use **parenthesis** to make your intent clear -- that way, you don’t even have to remember the precedence rules.*

Logical OR ||, Logical AND &&

```
if (value == 0 || value == 1 || value == 2 || value == 3)
     cout << ”You picked 0, 1, 2, or 3” << endl;



if (value > 10 && value < 20 && value != 16)
    // do something
else
    // do something else
```

Short circuit evaluation

- In order for logical AND to return true, both operands must evaluate to true.

- If the first operand evaluates to false, logical AND will go ahead and return false immediately
- without even evaluating the second operand!

```
if (x == 1 && y++ == 2)
    // do something
```

- if x does not equal 1, y++ never gets evaluated!
- which is probably not what the programmer intended!

Mixing ANDs and ORs

- logical AND has higher precedence than logical OR
- *use parenthesis*

These will be ignored here

- Converting between binary and decimal
- Bitwise operators
- Bit flags and bit masks