EEC-101
Programming with C++

Module-2.3 Expressions, Statements, and Operators

Ramanuja Panigrahi

Topics

- Concepts of algorithm & flow charts;
- Input/output,
- constants, variables
- operators;
- Naming conventions and styles;
- Conditions and selection statements;
- Looping and control structures (while, for, do-while, break and continue);
- File I/O, Header files, String processing;
- Pre-processor directives such as #include, #define, #ifdef, #ifndef;
- Compiling and linking.

Operators in C++

- Expressions
- Statements and block statements
- Operators
  - Assignment operators
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators

Assignment (=)

- A property that C++ has over other programming languages is that the assignment operation can be used as the rvalue (or part of an rvalue) for another assignment operation.

  For example:

  ```
  a = 2 + (b = 5);
  ```

  is equivalent to:

  ```
  b = 5;
  a = 2 + b;
  ```

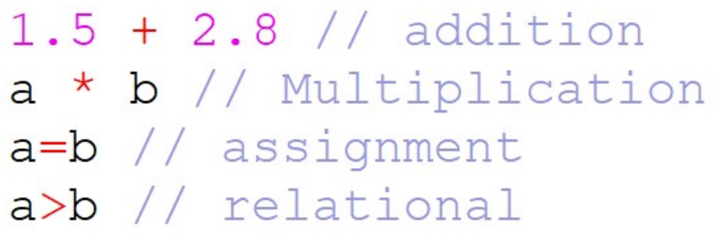
- Means first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignment of b (i.e. 5), leaving a with a final value of 7.

  The following expression is also valid in C++:

  ```
  a = b = c = 5;
  ```

Expression and Statements

- Expression is a sequence of operators and operands that specifies computation.
- Computes values from a number of Operands
-



```
1.5 + 2.8 // addition
a * b // Multiplication
a=b // assignment
a>b // relational
```

Statements

- A statement is a complete line of code that performs some action
- Usually terminated with a semicolon
- Usually contains expressions

```
int x;                                  // declaration
my_number=12;                          // assignment
x=2*5;                                // assignment
if (a>b) cout << "a bigger b"          // conditional
```

Operators

- C++ has a rich set of operators
    - Unary, binary, ternary
- Common C++ Operators are
    - Assignment operators
    - Arithmetic operators
    - Increment and decrement operators
    - Relational operators
    - Logical operators
    - Conditional operators
    - Bitwise operators
    - Special operators

Assignment Operator (=)

```
lhs=rhs;    // assignment operator
```

- rhs is an expression to be evaluated
- The value of rhs is stored in lhs
- ‘=‘ is called assignment operator (binary)
- The value of rhs must be type-compatible with lhs
- More than one variable can be assigned in one statement.

- A property that C++ has over other programming languages is that the assignment operation can be used in the rhs (or part of an rhs) for another assignment operation.
- For example:
  x = 3 + (y = 2);
  is equivalent to:
  y = 2;
  x = 2 + y;
  This means first assign 5 to variable y , then assign 2+y to variable x.

- The following expression is also valid in C++:   a = b = c = 5;

Compound Assignment (+=, -=, *=, /=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable, we can use compound assignment operators:

Expression	Equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
price *= units + 1;	price = price * (units + 1);

Arithmetic operators

- +, - Unary plus and minus
- +, - Binary addition and subtraction
- * Multiplication
- / Integer or floating-point division
- % Modulus Operator and the remainder of after an integer division

Relational operators

- == Equal to
- != Not equal to
- > Greater than
- < Less than
- >= Greater than equal to
- <= Less than or equal to

Integer and floating point division

- **If both of the operands are integers**, the division operator performs integer division.
- Integer division drops any fractions and returns an integer value. For example, 7 / 3 = 2 because the fraction is dropped.
- Note that integer division does not round. For example, 3 / 4 = 0, not 1.

- **If either or both of the operands are floating point values**, the division operator performs floating point division.
- Floating point division returns a floating point value, and the fraction is kept.
- For example, 7.0 / 3 = 2.333, 7 / 3.0 = 2.333, and 7.0 / 3.0 = 2.333.

- **Note that trying to divide by 0 (or 0.0) will generally cause your program to crash, as the result are undefined!**

Modulus (remainder)

- The **modulus operator** is also informally known as the **remainder operator.**

- The modulus operator only works on integer operands, and it returns the remainder after doing integer division.

- For example, 7 / 3 = 2 remainder 1, thus 7 % 3 = 1.
- As another example, 25 / 7 = 3 remainder 4, thus 25 % 7 = 4.

- Modulus is very useful for testing whether a number is evenly divisible by another number: if x % y == 0,

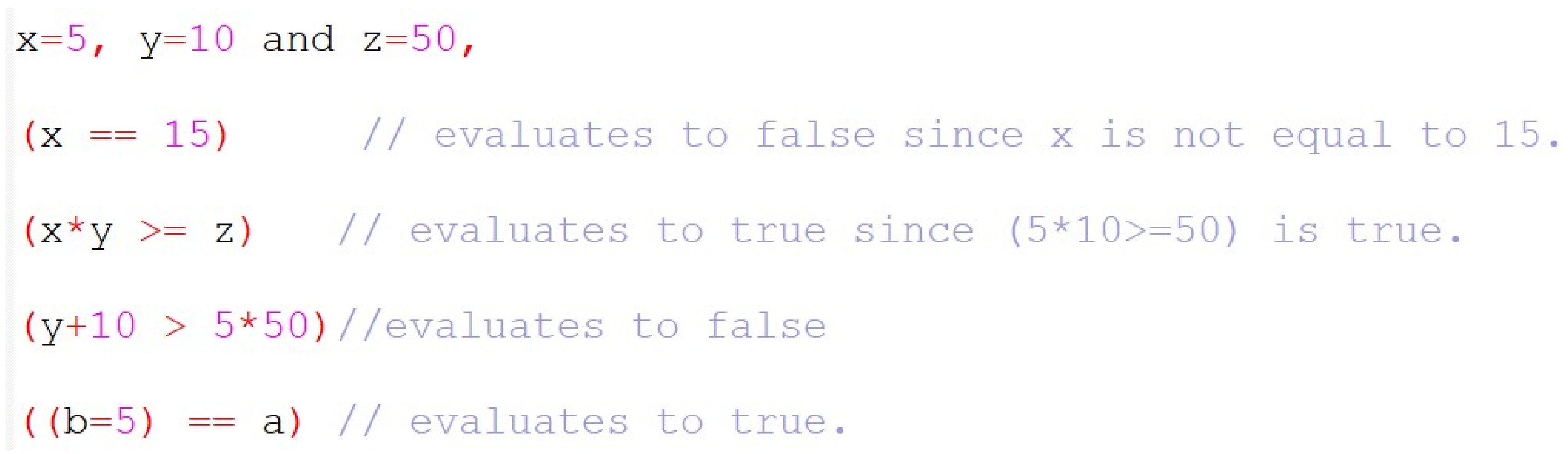
Relational and equality operators (==, !=, >, <, >=, <=)

- These are used to evaluate a comparison between two expressions.
- The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.
- We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is.

Examples of relational operators

```
(9 == 3)     // evaluates to false.
(100 > 4)      // evaluates to true.
(3 != 11)     // evaluates to true.
(6 >= 6)     // evaluates to true.
(5 < 5)      // evaluates to false.
```

Instead of using only numeric constants, one can use any valid expression, including variables.



```
x=5, y=10 and z=50,

(x == 15)      // evaluates to false since x is not equal to 15.

(x*y >= z)    // evaluates to true since (5*10>=50) is true.

(y+10 > 5*50)//evaluates to false

((b=5) == a) // evaluates to true.
```

- The operator = (one equal sign) is not the same as the operator == (two equal signs)

- **= is an assignment operator** assigns the value at its right to the variable at its left

- **== is the equality operator** that compares whether both expressions in the two sides of it are equal to each other.

- Example: ((b=2) == a),
  - first assigned the value 2 to b
  - then compared it to a, that also stores the value 2,
  - so the result of the operation is true.

Logical operators

- && AND
- || OR
- ! NOT

! Operator

The Operator ! is the C++ operator to perform the Boolean operation NOT, it returns the opposite Boolean value of evaluating its operand.

```
!(5 == 5)    // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)    // evaluates to true because (6 <= 4) would be false.
!true        // evaluates to false
!false       // evaluates to true.
```

- The logical operators && and || are used when evaluating two expressions to obtain a single relational result.

&& Operator

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

The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise.

|| operation.

- The operator || corresponds with Boolean logical operation OR.
- True if either one of its two operands is true.
- False only when both operands are false

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

- ((5 == 5) && (3 > 6)) // evaluates to false (true && false).

- ((5 == 5) || (3 > 6)) // evaluates to true (true || false).

Increment & decrement operators (unary)

- ++ adds one(1) to operand
- -- subtracts one(1) from operand

- There are two forms - prefix and postfix
  - ++k is prefix;
  - k++ is postfix;

- As standalone statements there is no difference in effect of the two.

- In an expression the effect of the two forms can be different.

Example 1

```
B=9;
A=++B;
// A contains 10, B contains 10
```

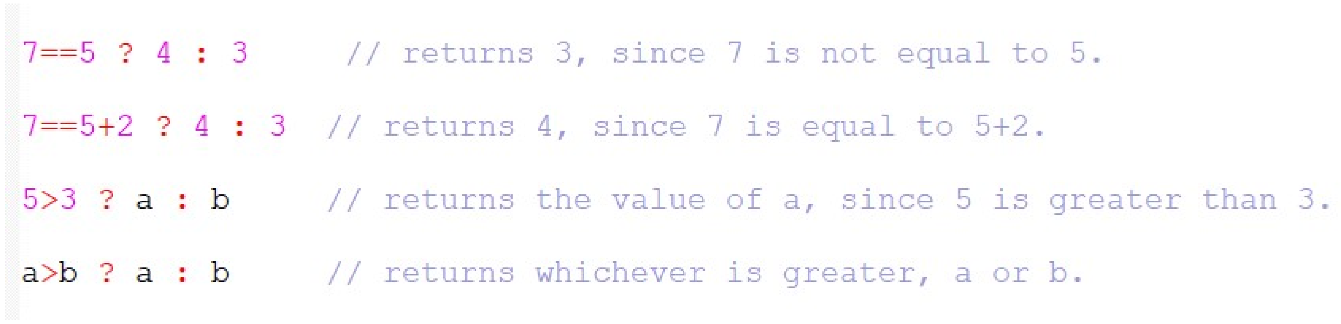
Example 2

```
B=3;
A=B++;
// A contains 3, B contains 4
```

Conditional operator (?)

condition ? result1 : result2

- If condition is true, then the expression will return result1.
- If condition is not true, then it will return result2.



```
7==5 ? 4 : 3      // returns 3, since 7 is not equal to 5.

7==5+2 ? 4 : 3  // returns 4, since 7 is equal to 5+2.

5>3 ? a : b     // returns the value of a, since 5 is greater than 3.

a>b ? a : b     // returns whichever is greater, a or b.
```

```
// conditional operator
#include <iostream>
using namespace std;
int main ()
{ int a,b,c;
a=2; b=7;
c = (a>b) ? a : b;
cout << c;
return 0; }
```

Result-----7

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

Operator	As equivalent	Description
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left
>>	SHR	Shift Right

C++ provides operators to work with the individual bits in int.

Integers are represented in binary.

For example:

$(3)_{10} = (011)_2$

$(5)_{10} = (101)_2$

Bitwise AND

- The bitwise AND operator is a single ampersand: &.
- Small version of the boolean AND, &&, works on smaller pieces (bits instead of bytes, chars, integers, etc).

- **For instance, working with a byte (the char type):**

```
        01001000 &
        10111000
=       --------
        00001000
```

- The most significant bit of the first number is 0, so we know the most significant bit of the result must be 0; in the second most significant bit, the bit of second number is zero, so we have the same result. The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left.
- **Example:** `72 & 184 = 8`

Example

```
//Bitwise operator &
#include<iostream>

using namespace std;

int main ()
{
  int a=72,b=184,c;
  c = a & b;
  cout << c<<"\n";
  return 0;
}
```

```
Result
8
```

Don't confuse && and &

- &&, is the short-circuit *logical AND*
- &, is the uncommon *bitwise AND*.

They may not produce the same result in a logical expression.

Bitwise OR

Bitwise OR works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1. (If both bits are a 1, the result will also have a 1 in that position.)

The symbol is a pipe: |.

This is similar to boolean logical operator, which is ||.

```
  01001000 |
  10111000
= --------
  11111000
```

and consequently

```
72 | 184 = 248
```

Bitwise Exclusive-Or (XOR)

The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. If both inputs are 1 or both inputs are 0, it returns 0.

Bitwise exclusive-or, with the operator of a carrot, ^, performs the exclusive-or operation on each pair of bits.
Exclusive-or is commonly abbreviated XOR.

```
01110010 ^
10101010
--------
11011000
```

Operator	Name	Description
a&b	and	1 if both bits are 1. 3 & 5 is 1.
a|b	or	1 if either bit is 1. 3 | 5 is 7.
a^b	xor	1 if both bits are different. 3 ^ 5 is 6.
~a	not	This unary operator inverts the bits. If ints are stored as 32-bit integers, ~3 is 11111111111111111111111111111100.
n<<p	left shift	shifts the bits of n left p positions. Zero bits are shifted into the low-order positions (fill from right with 0 bits)
n>>p	right shift	shifts the bits of n right p positions. If n is a 2's complement signed number, the sign bit is shifted into the high-order positions.

Explicit type casting operator

- Type casting operators allow to convert a datum of a given type to another.
- There are several ways to do this in C++.
- The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

- int i;
- float f = 3.14;
- i = (int) f;

- The previous code converts the float number 3.14 to an integer value (3), the remainder is lost.

- Here, the typecasting operator was (int).

- Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

  i = int (f);

Both ways of type casting are valid in C++.

Special Operators

- **size of()**
- returns the size of data type or variable in terms of byte occupied in memory
- **Comma**
- It separates the set of expressions

sizeof() Operator

- This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

  a = sizeof (char);

- This will assign the value 1 to a because char is a one-byte long type.

- The value returned by sizeof is a constant, so it is always determined before program execution.

Comma operator (,)

- The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected.
- When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

  For example, the following code:

  a = (b=3, b+2);

  - Would first assign the value 3 to b, and then assign b+2 to variable a.
  - So, at the end, variable a would contain the value 5 while variable b would contain value 3.

Operator precedence

operators have the same precedence as other operators in their group, and higher precedence than operators in lower groups

operator	**name**
!	boolean *not*
*	multiplication
/	division
%	mod
+	addition
-	subtraction
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to
&&	boolean *and*
||	boolean *or*
=	assignment
*=	multiply and assign

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left

- What does the following expression evaluate to? 6 + 5 * 4 % 3
- Because * and % have higher precedence than +, the + will evaluate last. We can rewrite our expression as 6 + (5 * 4 % 3). * and % have the same precedence, so we have to look at the associativity to resolve them. The associativity for * and % is left to right, so we resolve the left operator first. We can rewrite our expression like this: 6 + ((5 * 4) % 3).
- 6 + ((5 * 4) % 3) = 6 + (20 % 3) = 6 + 2 = 8