Operators

(CS 1002)

Dr. Mudassar Aslam

Cybersecurity Department

National University of Computer & Emerging Sciences,
Islamabad Campus

Arithmetic Operators

- **Used** for **performing numeric calculations**

- C++ has **unary**, **binary**, and **ternary** operators
  - **unary** (**1 operand**) `-5`
  - **binary** (**2 operands**) `13 - 7`
  - **ternary** (**3 operands**) `exp1 ? exp2 : exp3`

Binary Arithmetic Operators

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

Remainder operator **is also known as** ***modulus operator***

Integer and Real Division

```
float result = 5 / 2;          // ➔ result equal to 2

float result = 5.0 / 2;    // ➔ result equal to 2.5
```

- If any of the operand is a real value (float or double) the division will be performed as ***"Real Division"***

Remainder/Modulus operator

- Operands of **modulus** operator **must be integers**
  - **34 % 5** (**valid**, result → **4**)
  - **-34 % 5** (**valid**, result → **-4**)
  - **34 % -5** (**valid**, result → **4**)
  - **-34 % -5** (**valid**, result → **-4**)

NOTE: 34 % 1.2 **is an Error**

Arithmetic Expressions

- **Convert following expression into C++ code**

$$\text{result} = \frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

is translated to:

```
result = (3+4*x)/5 – (10*(y-5)*(a+b+c))/x  + 9*(4/x + (9+x)/y)
```

Example: Converting Temperatures

- Write a program that converts a Fahrenheit to Celsius using the formula:

$$celsius = (\frac{5}{9})(\ fahrenheit - 32)$$

Multiple Assignment

- The **assignment operator (=)** can be **used more than 1 time** in an **expression**

```
x = y = z = 5;
```

- Associates right to left

```
x = (y = (z = 5));
```

Done 3rd

Done 2nd

Done 1st

Combined Assignment

- Also consider it "**arithmetic" assignment**
- **Updates** a **variable** by **applying an arithmetic** operation to a variable
- Operators: **+= -= *= /= %=**

- Example:

`sum += amt;` is short for `sum = sum + amt;`

`p += 3 + y;` means `p = p + (3+y);`

More Examples

```
x += 5;   means   x = x + 5;
x -= 5;   means   x = x - 5;
x *= 5;   means   x = x * 5;
x /= 5;   means   x = x / 5;
x %= 5;   means   x = x % 5;
```

RULE: The right hand side is evaluated first, then the combined assignment operation is done.

```
x *= a + b;  means  x = x * (a + b);
```

Type Casting

Type Coercion

- **Coercion**: **automatic conversion** of an **operand** to **another data type**

- **Promotion**: **converts** to a **higher type**

  float p; p = 7; ➔ 7 (int) converted to **float 7.0**

- **Demotion**: **converts** to a **lower type**

  int q; q = 3.5; ➔ 3.5 (float) converted to **int 3**

Coercion Rules

1) **char, short, unsigned short are automatically promoted to int**

2) **When operating on values of different data types, the lower one is promoted to the type of the higher one.**

3) For the **assignment operator =** the type of **expression on right will be converted to** the **type of variable on left**

Typecasting

- A **mechanism** by which **we can change** the data **type** of a **variable** (**no matter how it was originally defined**)

- **Two ways:**
  1. **Implicit type casting** (*done by compiler*)
  2. **Explicit type casting** (*done by programmer*)

Implicit type casting

- **As seen in previous examples:**

```
void main( )
{
      char c = 'a';
      float f = 5.0;
      float d = c + f;
       cout<<d<<“ “<<sizeof(d)<<endl;
      cout<<sizeof(c+f);
}
```

Numeric Type Conversion

Consider the following statements:

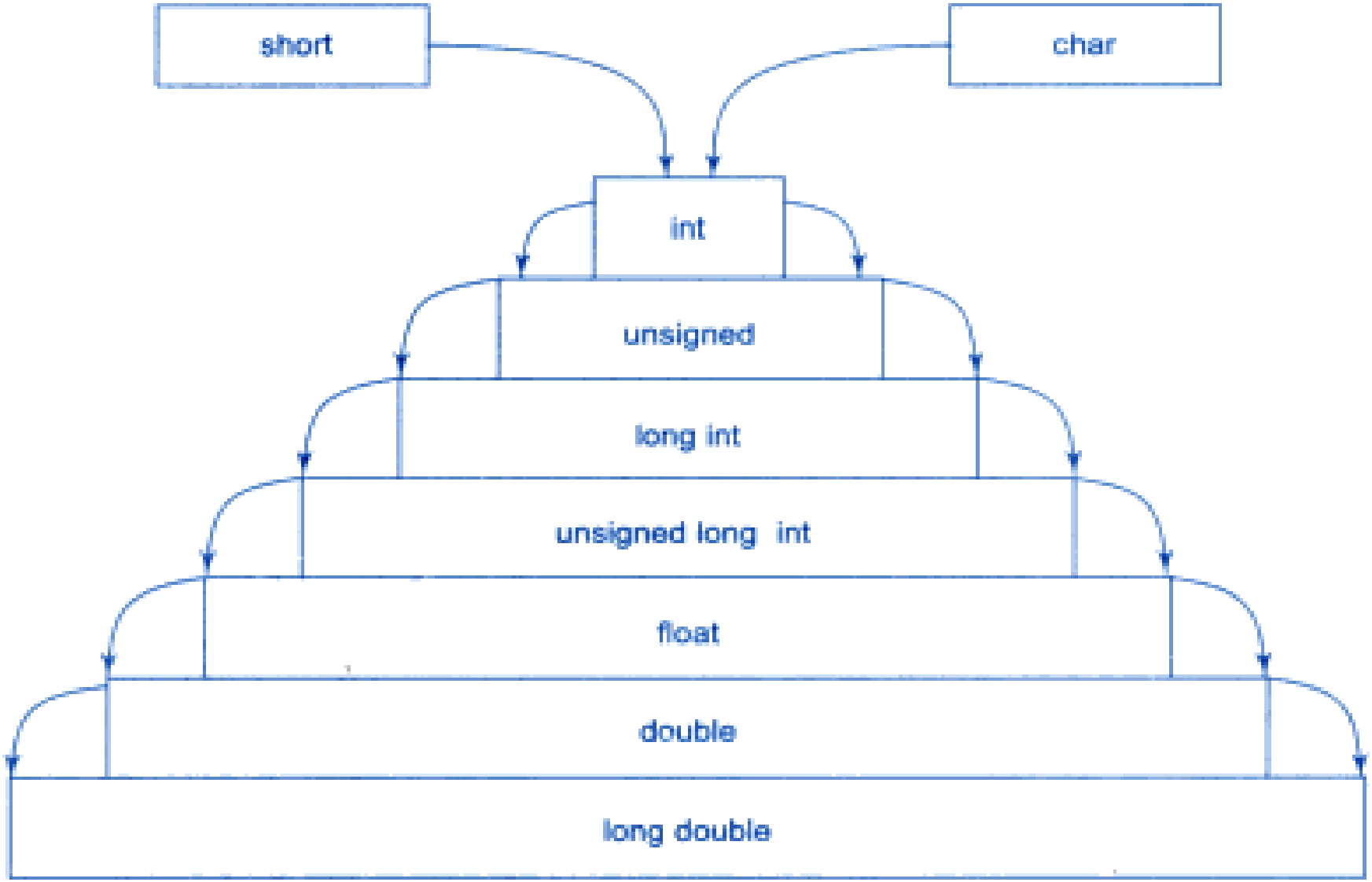
```
short i = 10;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
cout<<d;
```

Type Conversion Rules

Auto Conversion of Types in C++

1. If one of the operands is **long double**, the other is **converted into long double**
2. Otherwise, if one of the operands is **double**, the other is **converted into double**.
3. Otherwise, if one of the operands is **unsigned long**, the other is **converted into unsigned long**.
4. Otherwise, if one of the operands is **long**, the other is **converted** b**long**.
5. Otherwise, if one of the operands is **unsigned int**, the other is **converted into unsigned int**.
6. Otherwise, both operands are converted into **int**.

Implicit Type Conversion in C++



Overflow and Underflow

- When a variable is assigned a value that is too large or too small in range:
  - Overflow
  - Underflow

  - After overflows/underflow values wrap around the maximum or minimum value of the type

Example

```
// testVar is initialized with the maximum value for a short.
short int testVar = 32767;

// Display testVar.
cout <<"\nOrignal value: "<<testVar <<endl;

// Add 1 to testVar to make it overflow.
testVar = testVar + 1;
cout <<"\nValue Overflow +1: "<<testVar << endl;

// Subtract 1 from testVar to make it underflow.
testVar = testVar - 1;
cout <<"\nValue underflow -1: "<<testVar << endl;
```

Explicit type casting

- Explicit casting performed by programmer. It is performed by using cast operator

```
float a=5.0, b=2.1;
int  c = a%b;   //  → ERROR
```

- **Three Styles**

```
int c = (int) a  %  (int) b;        //C-style cast
int c = int(a)   %   int(b);        // Functional notation
int c = static_cast<int>(a) % static_cast<int>(b);
cout<<c;
```

Explicit Type Casting

- **Casting does not change the variable being cast.**

For example, **d** is not changed after casting in the following code:

```
double d = 4.5;
int j = (int) d;   //C-type casting
int i = static_cast<int>(d);  // d is not changed
cout<<j<<" "<<d;
```

Explicit Type Casting - Example

Program Output with Example Input Shown in Bold

```
How many books do you plan to read? 30 [Enter]
How many months will it take you to read them? 7 [Enter]
That is 4.28571 books per month.
```

division.

```
int main()
{
    int books;          // Number of books to read
    int months;         // Number of months spent reading
    double perMonth;    // Average number of books per month

    cout << "How many books do you plan to read? ";
    cin >> books;
    cout << "How many months will it take you to read them? ";
    cin >> months;
    perMonth = static_cast<double>(books) / months;
    cout << "That is " << perMonth << " books per month.\n";
    return 0;
}
```

Widening type casting

- A "**widening**" cast is a cast from one type to another, where the "**destination**" type has a **larger range** or **precision** than the "**source**“

Example:

```
int  i = 4;
double d = i;
```

Narrowing type casting

- A “**narrowing**" cast is a cast from one type to another, where the "**destination**" type has a **smaller range** or **precision** than the "**source**“

Example:

```
double d = 787994.5;
int j = (int) d;

// or

int i = static_cast<int>(d);
```

Casting between char and Numeric Types

```
int i = 'a';        // Same as    int i = (int) 'a';

char c = 97;    // Same as    char c = (char)97;
```

Using ++, -- on “char” type

- The **increment** and **decrement** operators can also be applied on **char** type variables:

Example:

```
char ch = 'a';
cout << ++ch;
```

Mathematical Expressions

- An **expression** can be a **constant**, a **variable**, or a **combination of constants and variables** combined with operators

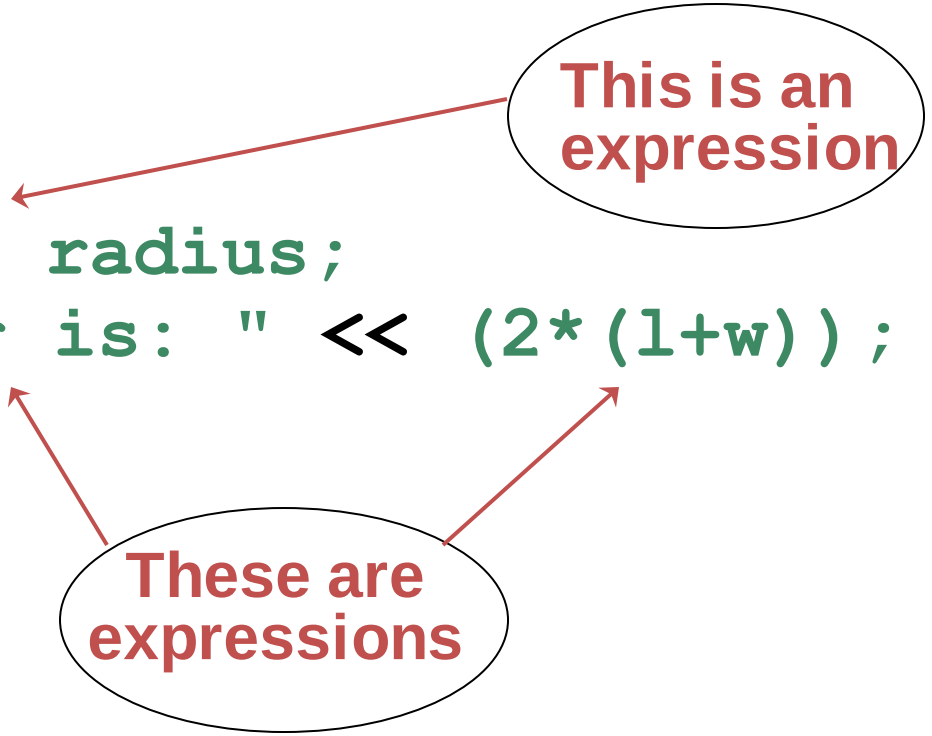
- Can create **complex expressions** using **multiple mathematical operators**:

```
 2
height
a + b / c
```

Using Mathematical Expressions

- Can be used in assignment statements, with **`cout`**, and in other types of statements
- Examples:

```
area = 2 * PI * radius;
cout << "border is: " << (2*(l+w));
```



Precedence Rules

Priority	Operators	Ass.	Associativity
high	! ~ ++ -- + - (Unary Operators)	⇐	right to left
	* / % (Arithmetic Operators)	⇒	left to right
	+ - (Arithmetic Operators)	⇒	left to right
	<< >> (Bitwise shift operators)	⇒	left to right
	< <= > >= (Relational operators)	⇒	left to right
	== != (Equality operators)	⇒	left to right
	&	⇒	left to right
	^	⇒	left to right
	|	⇒	left to right
	&&	⇒	left to right
	||	⇒	left to right
	? :	⇐	right to left
	= += -= *= /= %= &= ^= |= <<= >>=	⇐	right to left

Order of Operations

- In an **expression** with **more than one operator**, **evaluate** in this **order**

- In the expression `2 + 2 * 2 - 2`

Associativity of Operators

- **Implied grouping/parentheses**

  Example: - (unary negation) *associates right to left*

```
-5 ➔ -5;
--5 ➔ - (-5) ➔ 5;
---5 = -(-(-5)) = - (+5) ➔ -5
```

Associativity of Operators

- * / % + - all **associate left to right**

3 + 2 + 4 + 1 = (3 + 2) + 4 + 1 = ((3+2)+4)+1 =(((3+2)+4) + 1)

- **parentheses ( )** can be used to **override** the **order of operations**

```
 2 + 2  *  2 - 2  = 4
(2 + 2) *  2 - 2  = 6
 2 + 2  * (2 - 2) = 2
(2 + 2) * (2 - 2) = 0
```

Algebraic Expressions

- **Multiplication requires** an **operator**

  Area = lw is written as **Area = l * w;**

- There is **no exponentiation operator**

  Area = s^2 is written as **Area = pow(s, 2);**

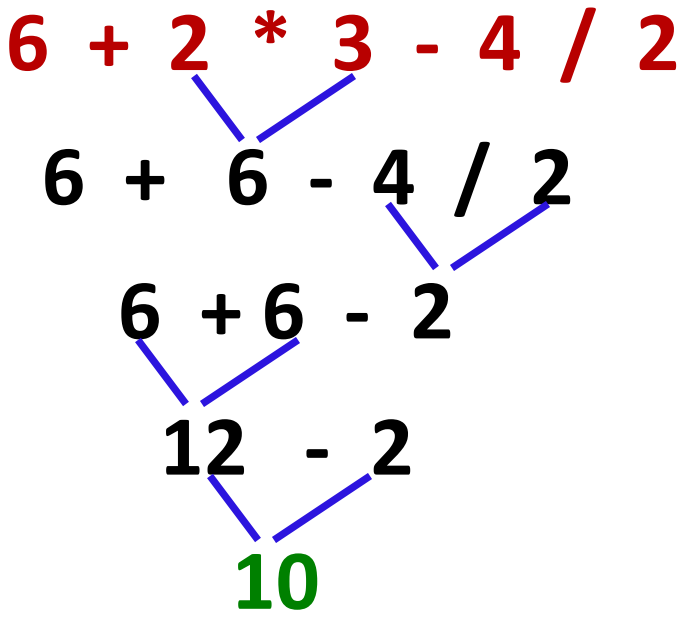
  (note: **pow** requires the **cmath** header file) OR

  Area = s*s;

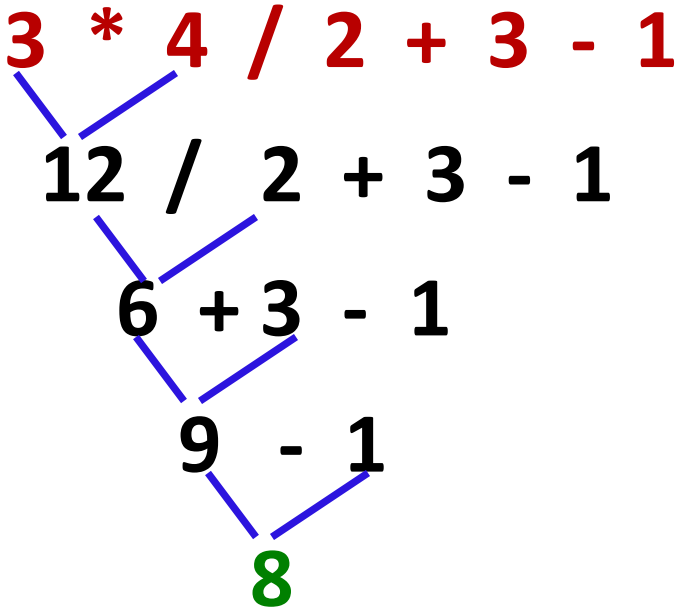
- **Parentheses** may be **needed** to **maintain order of operations**

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$ is written as: **m = (y2-y1)/(x2-x1);**

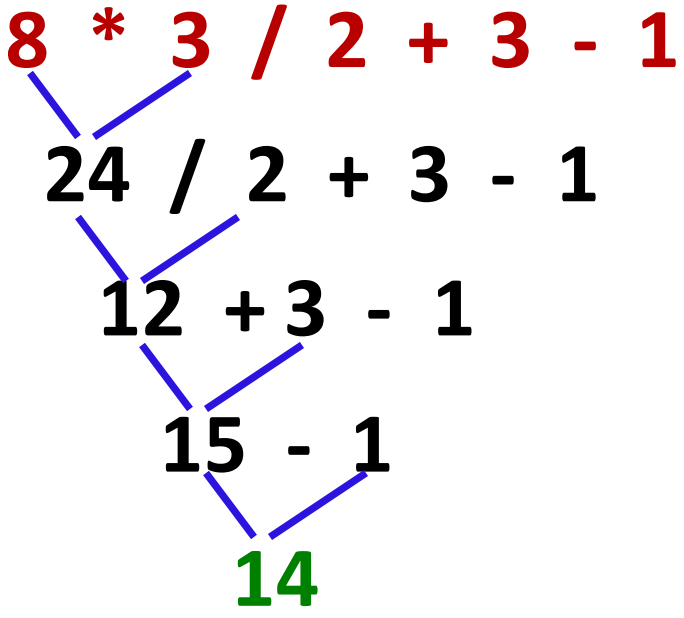
Precedence Rules – Example 1

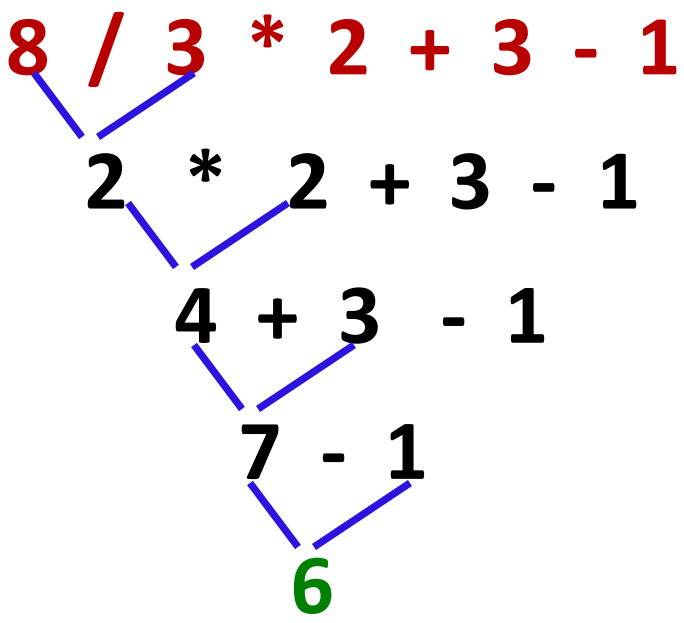


Precedence Rules – Example 2



Precedence Rules – Example 3





Precedence Rules (overriding)

- For example: **x = 3 * a - ++b % 3;**

- If we intend to have the statement evaluated differently from the way specified by the precedence rules, we need to specify it using parentheses ( )

- Using parenthesis:

  x = 3 * ((a - ++b)%3);

Pre and Post Increment Operators

will be used frequently in loops

Increment and Decrement Operators

Operator	Name	Description
++var	**pre-increment**	The expression (++var) increments var by 1 and evaluates to the *new* value in var *after* the increment.
var++	**post-increment**	The expression (var++) evaluates to the *original* value in var and increments var by 1.
--var	**pre-decrement**	The expression (--var) decrements var by 1 and evaluates to the *new* value in var *after* the decrement.
var--	**post-decrement**	The expression (var--) evaluates to the *original* value in var and decrements var by 1.

Increment and Decrement Operators

- Evaluate the followings:

```
int val = 10;
int result = 10 * val++;
cout<<val<<“  “<<result;
```

```
int val = 10;
int result = 10 * ++val;
cout<<val<<“ “<<result;
```

Increment and Decrement Operators

- Output of the following code:

```
int x = 5, y = 5, z;
x = ++x;
y = --y;
z = x++ + y--;
cout << z;
```

Increment and Decrement Operators

- Output of the following code:

```
int num1 = 5;
int num2 = 3;
int num3 = 2;
num1 = num2++;
num2 = --num3;
cout << num1 << num2 << num3 <<endl;
```

Any Questions!