Data-Types & Operators

(CS 1002)

Dr. Muhammad Aleem,

Department of Computer Science,

National University of Computer & Emerging Sciences,
Islamabad Campus

Constants (named)

- **Named constants** are **declared** and **referenced** by **identifiers**:

```
        const  int   MAX_MARKS = 100;
        const  string   UNIVERSITY = “FAST";
        const  double   PI = 3.141592654;
        const  char   TAB = '\t';
```

- Constants **must** be **initialized** in their **declaration**

- **No further assignment possible within program**

C++ Standard Constants

```
#include <climits>        //integer constants defined here

INT_MIN      INT_MAX
LONG_MIN     LONG_MAX
```

Lower and upper bounds for Integer types.

```
#include <cfloat>         // float constants defined here

FLT_MIN      FLT_MAX
DBL_MIN      DBL_MAX
```

Lower and upper bounds for Decimal types.

Types

- **C++ provides a set of types**
  - E.g. **bool**, **char**, **int**, **double** called **“built-in types”**
- C++ **programmers** can **define new types**
  - Called ***“user-defined types”***
- The **C++ standard library** provides a **set of types**
  - E.g. **string, vector, ..**
  - (for vector type → #include<vector>)

Data Types

Three basic PRE-DEFINED data types:

1. To **store whole numbers**
   - **int**, **long int**, **short int**, **unsigned int**

2. To **store real numbers**
   - **float**, **double**

3. **Characters**
   - **char**

Types and Literals

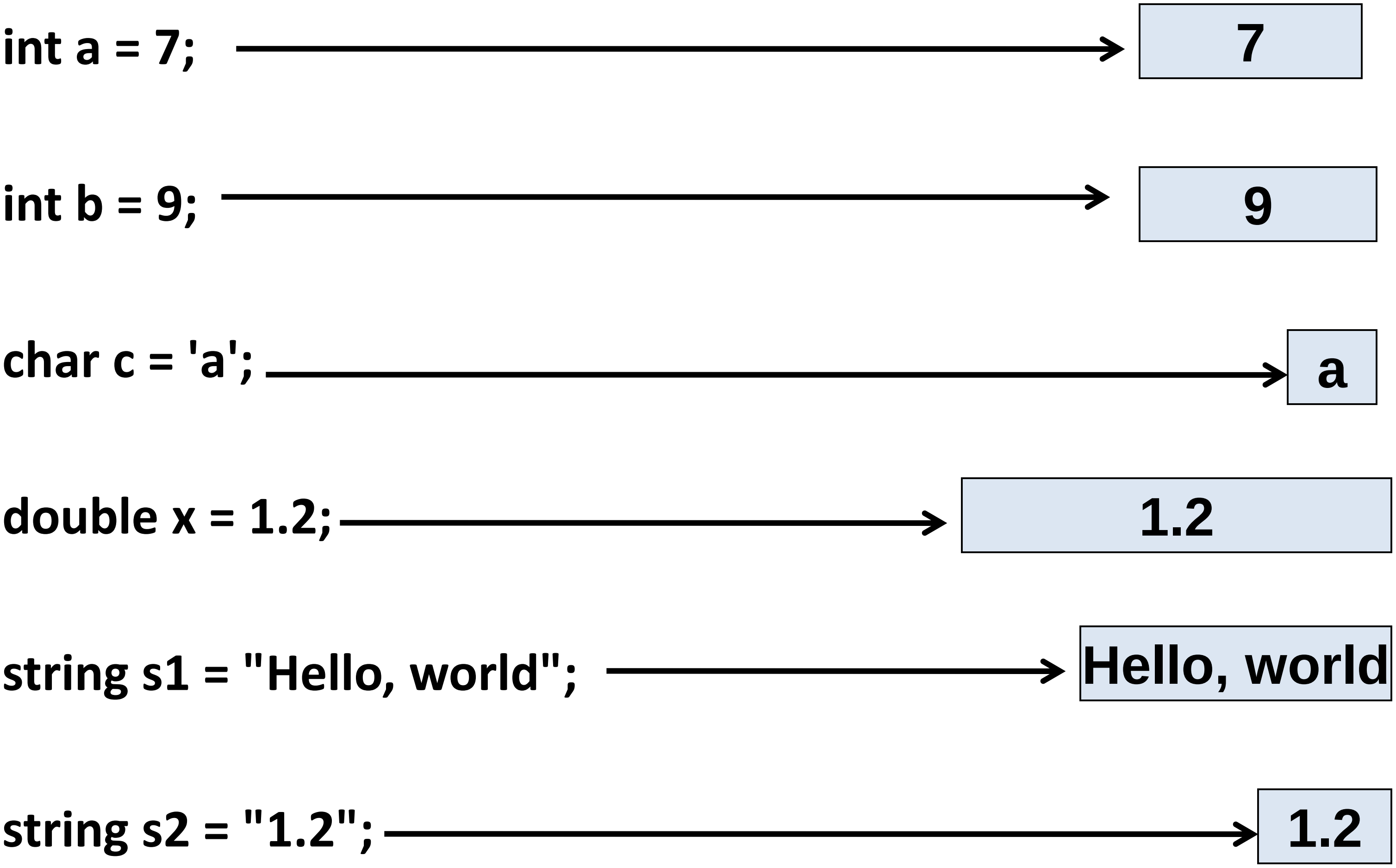
- **Built-in types**
  - Boolean type
    - **bool**
  - Character types
    - **char**
  - Integer types
    - **int**
      - **and** **short** and **long**
  - Floating-point types
    - **double**
      - and **float**

- **Standard-library types**
  - **string**

Literals

- Boolean: **true**, **false**
- Character literals
  - **'a'**, **'x'**, **'4'**, **'\n'**, **'$'**
- Integer literals
  - **0**, **1**, **123**, **-6**,
- Floating point literals
  - **1.2**, **13.345**, **0.3**, **-0.54**,
- String literals
  - **"asdf"**, **“Hello”**, **“Pakistan”**

Declaration and initialization



int a = 7; → 7

int b = 9; → 9

char c = 'a'; → a

double x = 1.2; → 1.2

string s1 = "Hello, world"; → Hello, world

string s2 = "1.2"; → 1.2

char type

- Reserves **8 bits** or **1 byte** of memory
- A char variable may represent:
  - ASCII character 'A‘, 'a‘, '1‘, '4‘, '*‘
  - signed integers **127** to **-128** (Default)
  - unsigned integer in range **255** to **0**

Examples:

- char grade;
- unsigned char WeekNumber= 200;
- char cGradeA = 65;
- char cGradeAA = ‘A';

char type

- Example program…

Special characters

- Text string special characters (Escape Sequences)
  - **\n** = newline
  - **\r** = carriage return
  - **\t** = tab
  - **\"** = double quote
  - **\?** = question
  - \\ = backslash
  - **\'** = single quote

- **Examples:**

```
cout << "Hello\t" << "I\’m Ali\n";
cout << "123\nabc “;
```

Escape Sequence

- **Example Program:**

int type

32 bits (4 bytes) on Win32 /Linux 32-bit system

- int -2,147,483,648 to 2,147,483,647
- unsigned int 0 to 4,294,967,295

- **Examples:**

```
int  earth_diameter;
int seconds_in_week= 604800;
unsigned int Height = 100;
unsigned int Width = 50000;
```

int type (long and short)

- **long int**
  - reserves 64 bits (8 bytes) of memory
  - signed long -2,147,483,648 to 2,147,483,647
  - unsigned long int 0 to 4,294,967,295

- **short int**
  - reserves 16 bits (2 bytes) of memory
  - signed short int -32,768 to 32,767
  - unsigned short int 0 to 65,535

int (long and short)

- **Examples:**

```
long int  light_speed=186000;
unsigned long int seconds= 604800;
short int Height = 30432;
unsigned short int Width = 50000;
```

Check Bytes in Memory – Whole Numbers

- **Check how many bytes following types occupy in memory:**

  int

  short

  long int

  short int

  char

- Use (cout << sizeof(intVar);) operator to get this information, Example:…

Real Values

- **float**
  - Reserves 32 bits (4 bytes) of memory
  - **$\pm$ 1.180000x10$^{\pm 38}$** , 7-digit precision
  - *Example*: float radius= 33.4221;

- **double**
  - Reserves 64 bits (8 bytes) of memory
  - **$\pm$ 1.79000000000000x10$^{\pm 308}$** , 15-digit precision
  - *Example:* double Distance = 257.5434342;

- **long double**
  - Reserves **128 bits** (16 bytes) of memory , 18-digit precision
  - *Example:* long double EarthMass = 25343427.53434233;

Check Bytes in Memory – Real Numbers

- get information for following data types:

  float

  double

  long double

- Use (**cout << sizeof(floatVar);**) operator to get this information, Example:…

bool Type

- Only **1 bit of memory required**
  - Generally, **1 byte** is reserved
- **Literal values:**
  - **true**
  - **false**
- Can be used in **logical conditions**:
  - Examples:

```
bool RainToday=false;
bool passed;
passed = GetResult(80);
```

string type

- **Special data type** supports working with “*strings*”

```
 #include <string>

      string <variable_name> = “string literal”;
```

- string type variables in programs:

```
      string firstName, lastName;
```

- Using with assignment operator:

```
     firstName = “Umer";
     lastName = “Arshad";
```

- Display using cout

```
 cout << firstName << " " << lastName;
```

Working with Characters and String Objects

- char: **holds** a **single character**
- string: **holds** a **sequence of characters**
- Both can be used in assignment statements
- Both can be displayed with **cout** and **<<**

Other Input Functions

- **>> operator DOES NOT read WHITESPACE**
  - **Skips** or **stops** on **space**, **tab**, **end-of-line**,
  - **Skips** over **leading white space**;
  - **Stops** on **trailing white space**.

- To **read** any **single char V** (***incl. whitespace***)
  - **cin.get(V)**

Character Input

- To skip input characters:

```
cin.ignore( );    // one character.
cin.ignore(n);  // n characters.
```

Reading in a character

```
char ch;
cin >> ch;         // Reads in any non-blank char
cin.get(ch);      // Reads in any char
cin.ignore();     // Skips over next char in
                  // the input buffer
```

Cin.ignore Example

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
    int empID=-1;
    string empName="";
    int empSalary=-1;

    cout<<"\nEnter employee ID:";
    cin>>empID;
    cin.ignore(1000,'\n');

    cout<<"\nEnter employee Name:";
    cin>>empName;
    //getline(cin,empName,'$');
    //cin.ignore(1000,'\n');

    cout<<"\nEnter Employee Salary:";
    cin>>empSalary;
    //cin.ignore(1000,'\n');

    cout<<endl;
    cout<<"\n=========== Data Entered by the User ================";
    cout<<"\nEmployee ID:"<<empID;
    cout<<"\nEmployee Name:"<<empName;
    cout<<"\nEmployee Salary:"<<empSalary;
    cout<<endl;
    return 0;
}
```

Operators

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:

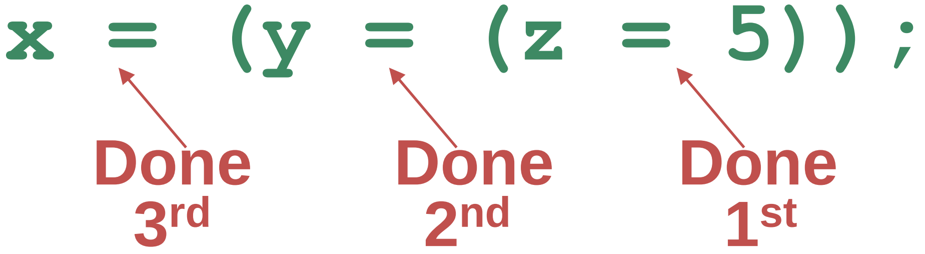
$$celsius = (\tfrac{5}{9})(fahrenheit - 32)$$

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);`

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;
```

Examples…

```
int a=1;
int b;
b = ++a * ++a;
cout<<a: "<<a<<", b: "<<b; cout<<endl;

a=1; b = a++ * a++;
cout<<"a: "<<a<<", b: "<<b; cout<<endl;

a=1; b = ++a * a++;
Cout<<"a: "<<a<<", b: "<<b; cout<<endl;

a=1; b = a++ * ++a;
cout<<"a: "<<a<<", b: "<<b; cout<<endl;
```

Examples…

When we have three operands it is evaluated as two at a time:

EXAMPLE: ++a + ++a + ++a; ➔ ((++a + ++a)   + ++a)

```
a=1; b = ++a + ++a + ++a;
cout<<"++a + ++a + ++a =  "<<"a: "<<a<<", b: "<<b<<endl;a=1;

b = a++ + a++ + a++;
cout<<"a++ + a++ + a++ =  "<<"a: "<<a<<", b: "<<b<<endl;a=1;

b = ++a + a++ + a++;
cout<<"++a + a++ + a++ =  "<<"a: "<<a<<", b: "<<b<<endl;a=1;

b = a++ + ++a + ++a;
cout<<"a++ + ++a + ++a =  "<<"a: "<<a<<", b: "<<b<<endl;a=1;

b = a++ + a++ + ++a;
cout<<"a++ + a++ + ++a =  "<<"a: "<<a<<", b: "<<b<<endl;a=1;
```

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:

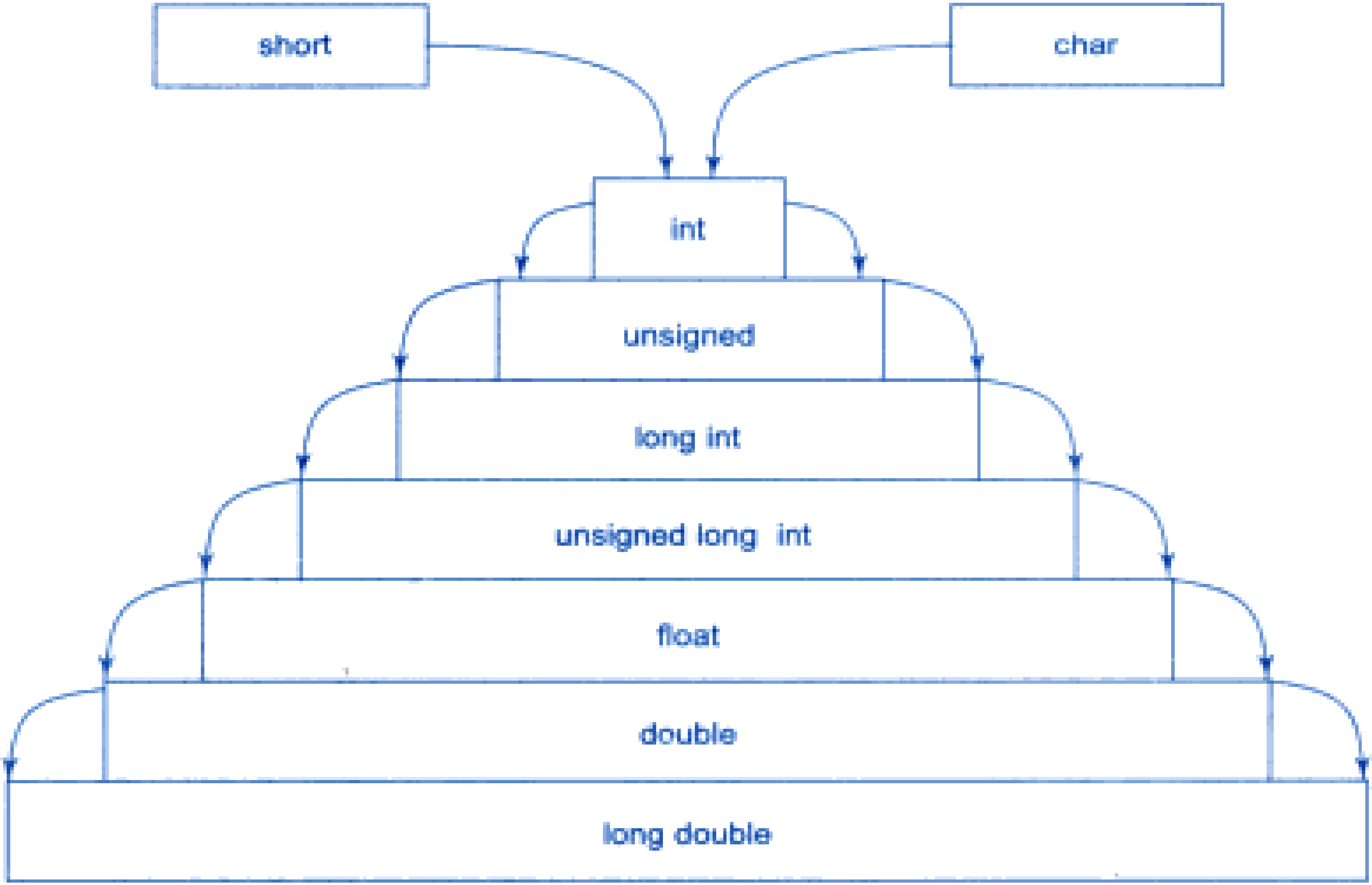
```
short i = 10;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
cout<<d;
```

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++



short

char

int

unsigned

long int

unsigned long int

float

double

long double

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;
```

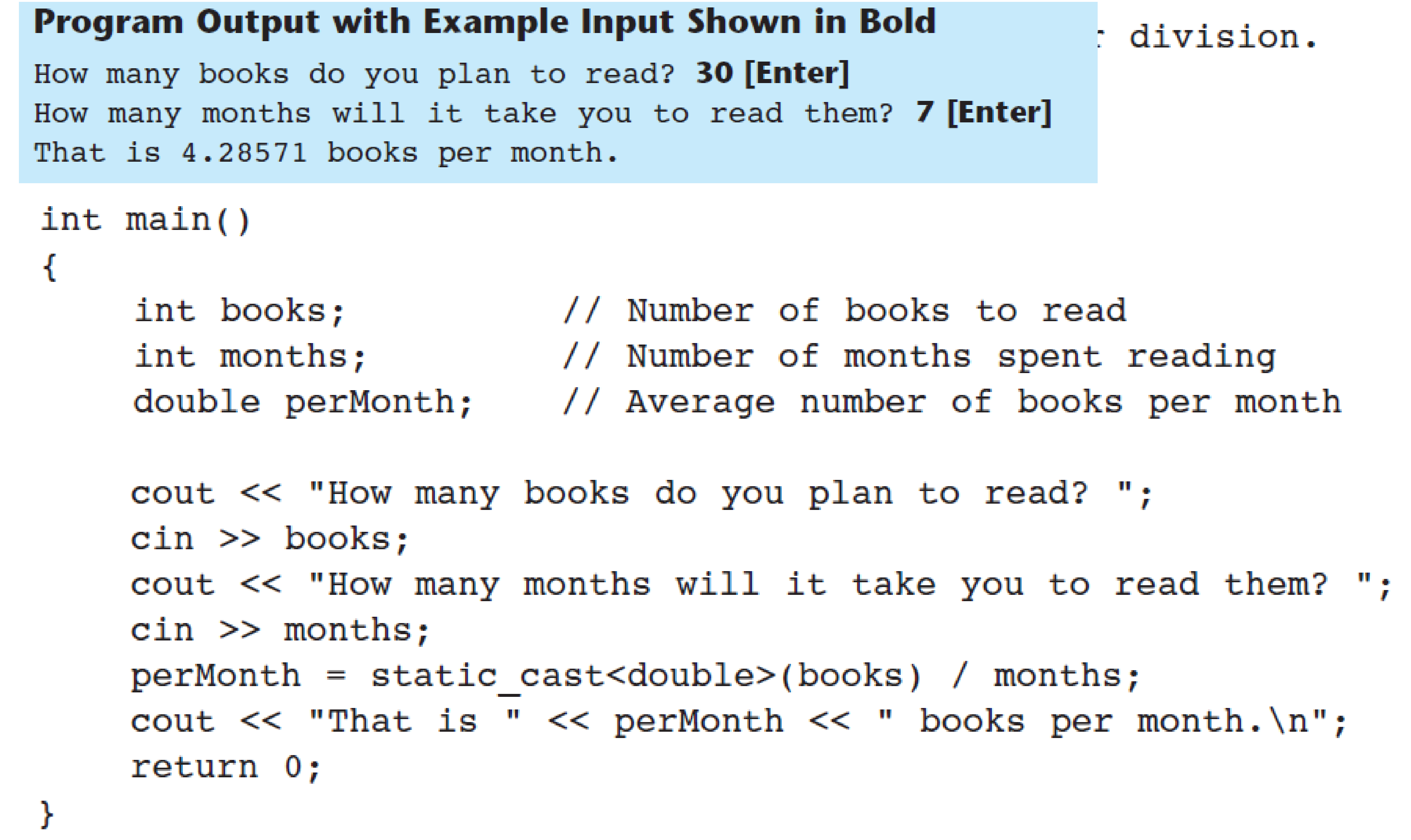
Explicit Type Casting

- **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;
```

int to string Conversion

- - C style (we will study in pointers topic)
- - C++ style:

```
#include<sstream>


void main()  {
        int val=0;
        stringstream ss;
        cout<<“Enter Value: “;   cin>>val;
        ss << val;     //Using stream insertion op.
        string str_val= ss.str();
        cout<<“\n Output string is: “<<str_val;
}
```

Equality and Relational Operators

- Equality Operators:

Operator	Example	Meaning
==	x == y	x is equal to y
!=	x != y	x is not equal to y

- Relational Operators:

Operator	Example	Meaning
>	x > y	x is greater than y
<	x < y	x is less than y
>=	x >= y	x is greater than or equal to y
<=	x <= y	x is less than or equal to y

Logical Operators

- Logical operators are useful when we want to test multiple conditions

- Three types:
  1. boolean AND
  2. boolean OR
  3. boolean NOT

Boolean AND or logical AND

- Symbol: **&&**

- **All** the conditions must be true for the whole expression to be true

  – Example:

```
if ( (a == 10) && (b == 10) && (d == 10) )
        cout<<“a, b, and d are all equel to 10”;
```

Boolean OR / Logical OR

- Symbol: ||

- **ANY** condition is sufficient to be true for the whole expression to be true

  – Example:

```
if (a == 10 || b == 9 || d == 1)
        // do something
```

Boolean NOT/ Logical NOT

- Symbol: **!**

- **Reverses the meaning of the condition** (makes a true condition false, OR a false condition true)

  – Example:

  ```
  if ( ! (marks > 90) )
          // do something
  ```

Bitwise Operators (integers)

- Bitwise "and" operator **&**
- Bitwise "or" operator **|**
- Bitwise "exclusive or" operator **^**
  - **(0 on same bits, 1 on different bits)**
- Bitwise "ones complement" operator **~**
- **Shift left <<**
- **Shift right >>**

Bitwise Operators (Example)

int i = 880;	→	1 1 0 1 1 1 0 0 0 0	
int j = 453;	→	0 1 1 1 0 0 0 1 0 1	
i & j	(320)	0 1 0 1 0 0 0 0 0 0	
i | j	(1013)	1 1 1 1 1 1 0 1 0 1	
i ^ j	(693)	1 0 1 0 1 1 0 1 0 1	
~j	(-454)	1 0 0 0 1 1 1 0 1 0	
i = i<<1;	(1760)	1 0 1 1 1 0 0 0 0 0	unsigned int
i = i>>1;	(440)	0 1 1 0 1 1 1 0 0 0	

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

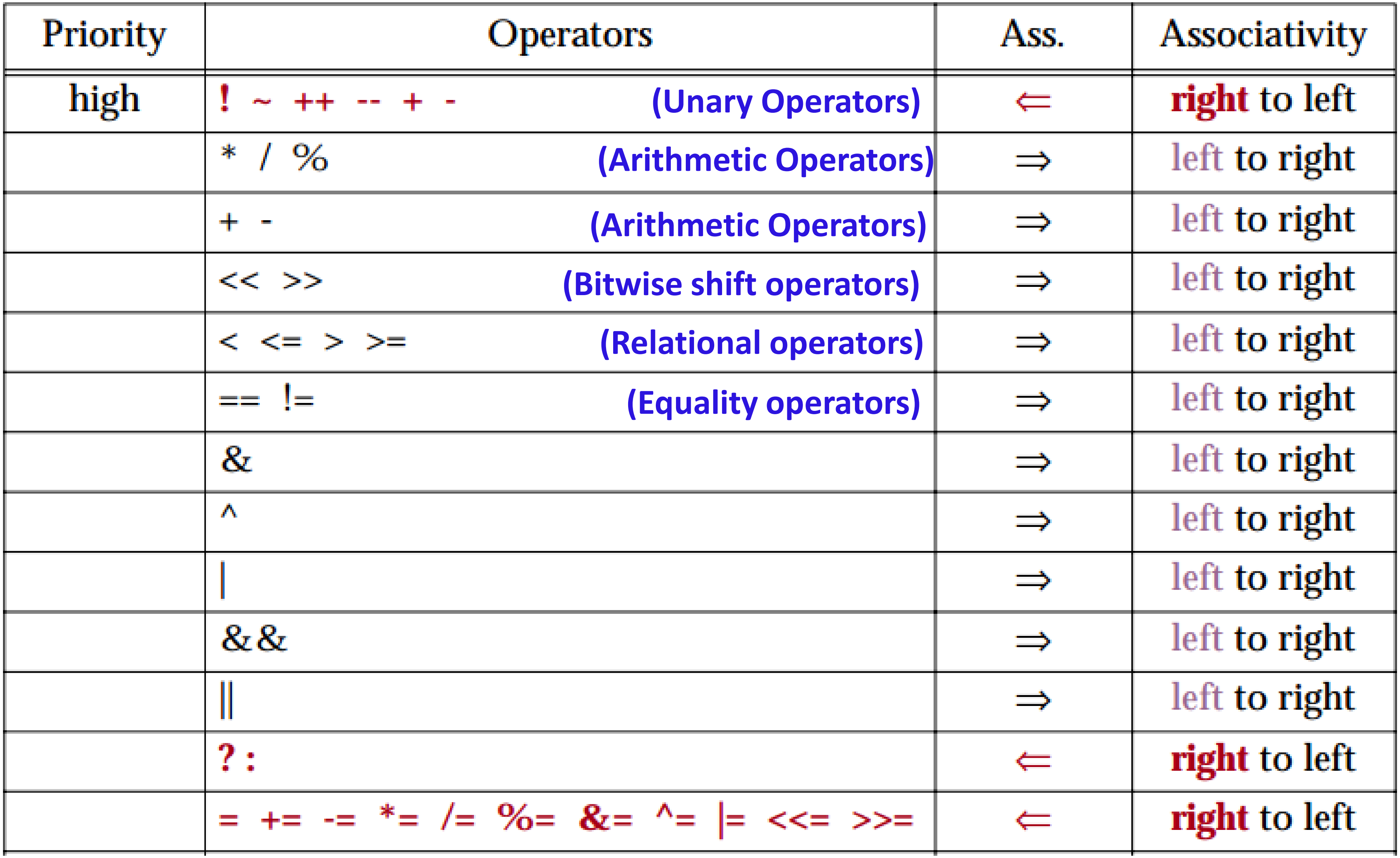
- Can be used in assignment statements, with **`cout`**, and in other types of statements
- Examples:

This is an expression

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

These are expressions

Precedence Rules

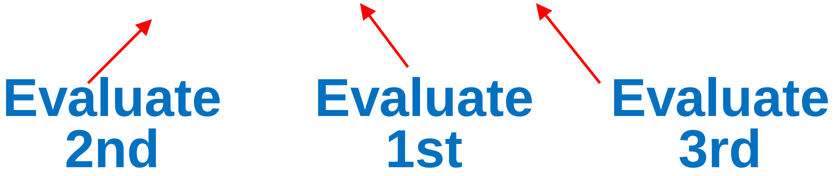


Priority	Operators	Ass.	Associativity
high	! ~ ++ -- + - (Unary Operators)	⇐	right to left
	* / % (Arithmetic Operators)	⇒	left to right
	+ - (Arithmetic Operators)	⇒	left to right
	<< >> (Bitwise shift operators)	⇒	left to right
	< <= > >= (Relational operators)	⇒	left to right
	== != (Equality operators)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



Evaluate 2nd

Evaluate 1st

Evaluate 3rd

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);**

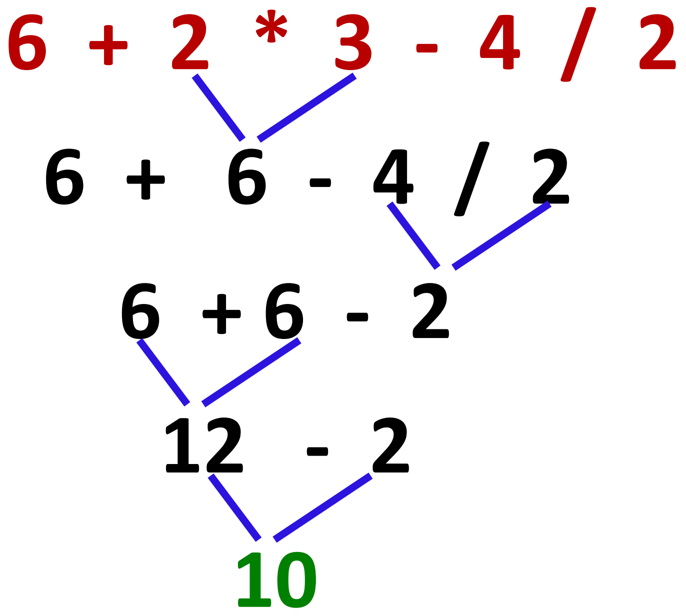
  (note: **pow** requires the **cmath** header file) OR

  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



6 + 2 * 3 - 4 / 2

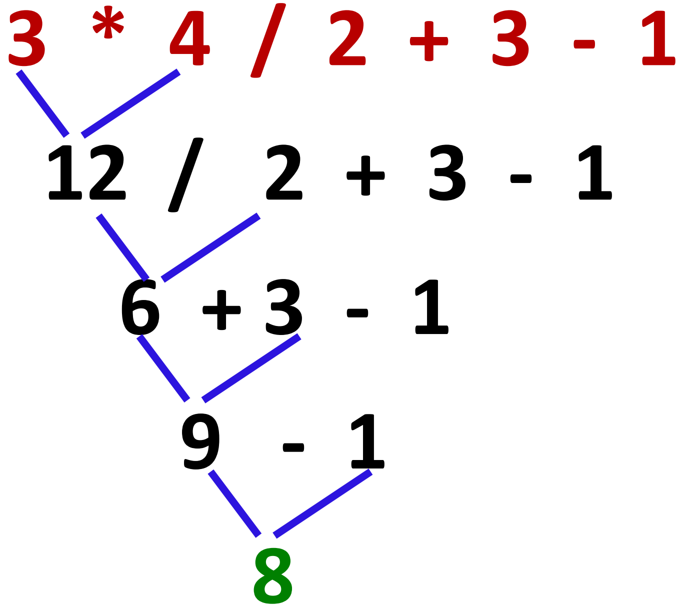
6 + 6 - 4 / 2

6 + 6 - 2

12 - 2

10

Precedence Rules – Example 2



3 * 4 / 2 + 3 - 1

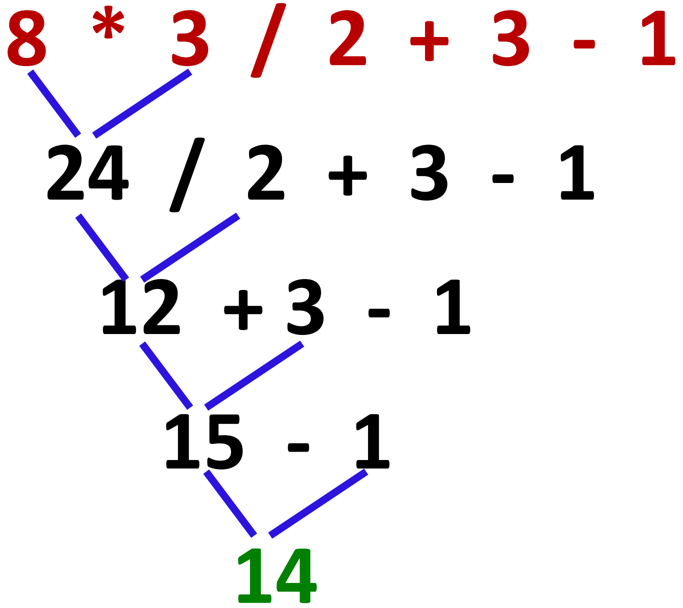
12 / 2 + 3 - 1

6 + 3 - 1

9 - 1

8

Precedence Rules – Example 3



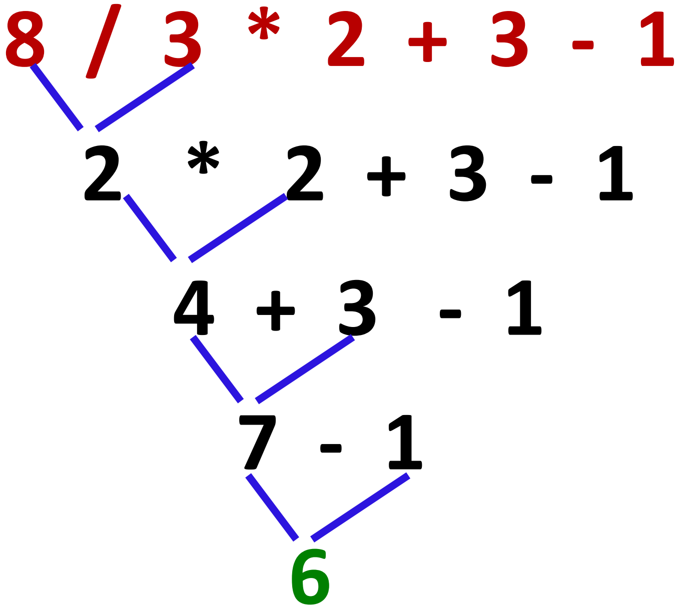
8 * 3 / 2 + 3 - 1

24 / 2 + 3 - 1

12 + 3 - 1

15 - 1

14



8 / 3 * 2 + 3 - 1

2 * 2 + 3 - 1

4 + 3 - 1

7 - 1

6

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);
```

Any Questions!