Operators and Type Casting Revisisted

(CS 1002)

Dr. Mudassar, Ms. Uzma Bibi, Ms. Nimra Shahid

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>

INT_MIN INT_MAX
LONG_MIN LONG_MAX

//integer constants defined here

Lower and upper bounds for Integer types.

#include <cfloat>

FLT_MIN FLT_MAX
DBL_MIN DBL_MAX

// float constants defined here

Lower and upper bounds for Decimal types.

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;
}
```

String Input

- **>> operator** can **NEVER read strings** that **contain WHITESPACE**
  - **Skips** or **stops** on space, tab, end-of-line, end-of-file

- To read string S (which may contain whitespace)
  - string S;
  - getline(cin, S);

- How it works: **reads all characters** from cursor (5) to the **end-of-line** character (20), but **does not store the eoln character**.

  _ 1 2 3 _ T O M _ B R O W N _ 7 2 . 5 eol

  1 2 3 4 **5** 6 7 8 9 0 1 2 3 4 5 6 7 8 9 **20** ← position

  getline(cin, S); **S = " TOM BROWN 72.5"**

String Input

Reading in a string object

```
string str;

cin >> str;              // Reads in a string
                         // with no blanks

getline(cin, str);       // Reads in a string
                         // that may contain
                         // blanks
```

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`

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)
```

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…

```
a: 3, b: 9
a: 3, b: 2
a: 3, b: 6
a: 3, b: 3
```

```
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 i
EXAMPLE: `++a + ++a + ++a;`

```
++a + ++a + ++a =  a: 4, b: 10
a++ + a++ + a++ =  a: 4, b: 6
++a + a++ + a++ =  a: 4, b: 8
a++ + ++a + ++a =  a: 4, b: 8
a++ + a++ + ++a =  a: 4, b: 7
```

```
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:

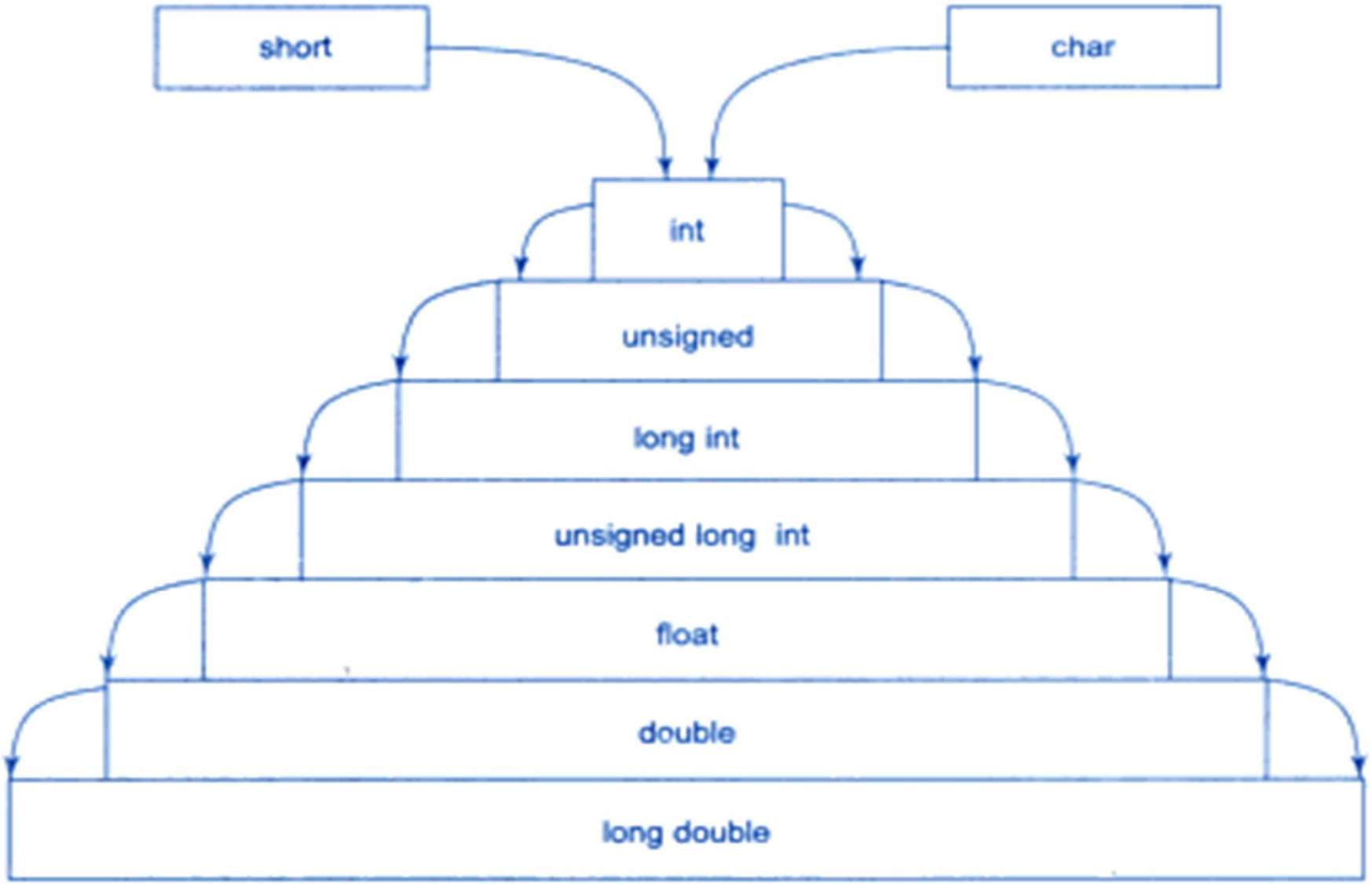
```
short i = 10;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
cout<<d;
```

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++



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;
}
```

Explicit Type Casting - Example

WARNING! In Program 3-9, the following statement would still have resulted in integer division:

```
perMonth = static_cast<double>(books / months);
```

The result of the expression `books / months` is 4. When 4 is converted to a `double`, it is 4.0. To prevent the integer division from taking place, one of the operands should be converted to a `double` prior to the division operation. This forces C++ to automatically convert the value of the other operand to a `double`.

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;
```

Any Questions!