LECTURE-16

Conversions

In a mixed expression constants and variables are of different data types. The assignment operations cause automatic type conversion between the operand as per certain rules. The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left. Consider the following example:

int x;

float y = 20.123;

x=y ;

These converts float variable y to an integer before its value assigned to x. The type conversion is automatic as far as data types involved are built in types.

```
#include <iostream>
using namespace std;
main()
{
  int int_var = 50;
  char char_var = 'A';
  int_var = int_var + char_var;
  // char_var is implicitly converted to the integer ASCII of 'A'.
  // ASCII of 'A' is 65.
  cout << "The value of (50 + 'A') is: " << int_var << endl;
  // Now, converting int_var to a float (implicitly).
  float float_var = int_var * 1.5;
  cout << "The value of float_var is: " << float_var << endl;
}
```

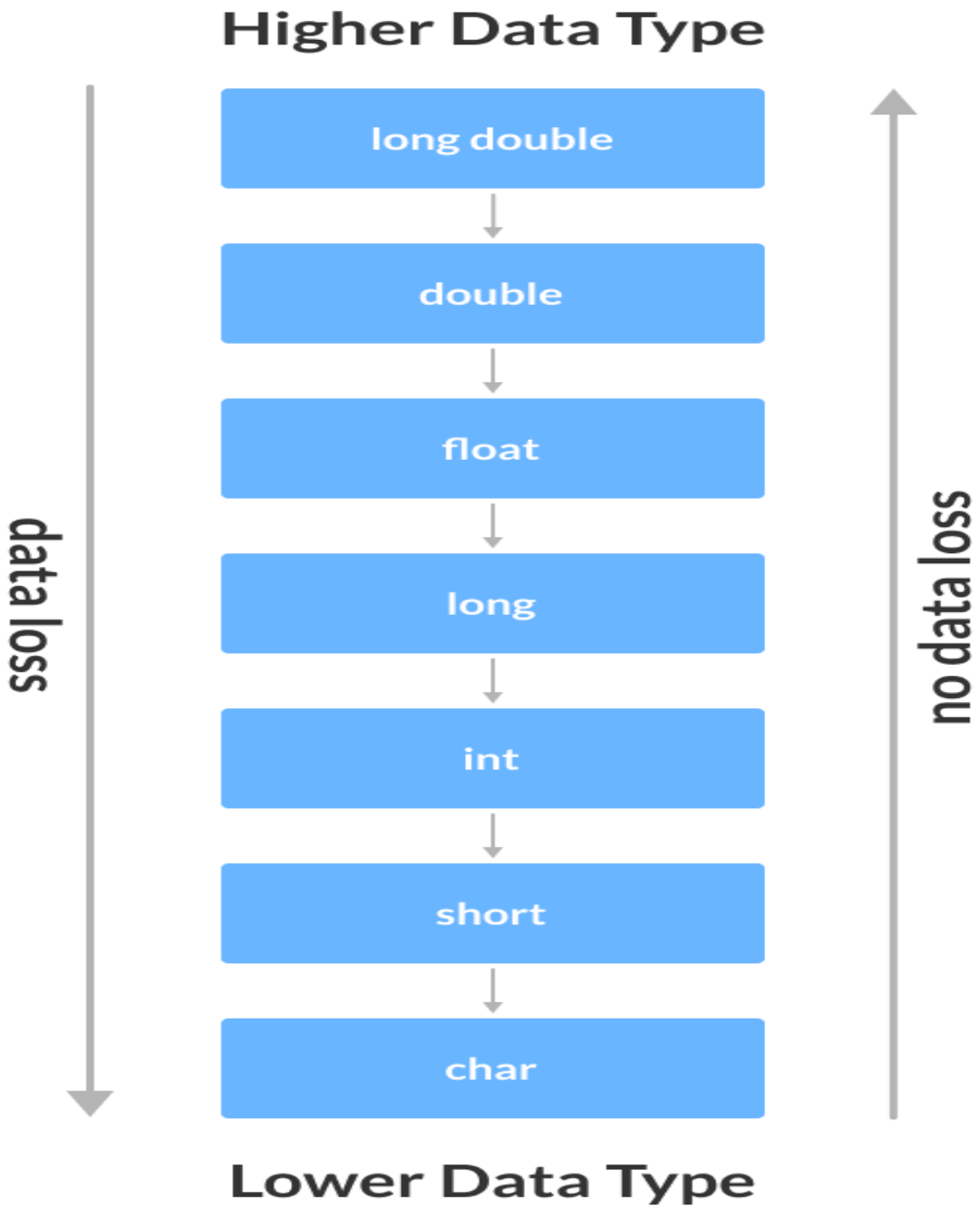
Example

```
#include <iostream>
using namespace std;
main()
{
  char char_var = 'a';
  int int_var;
  int_var = (int) char_var; // Using cast notation.
  cout << "The value of char_var is: " << char_var << endl;
  cout << "The value of int_var is: " << int_var << endl;
}
```

Example

```
#include <iostream>
using namespace std;
main() {
  int int_var = 17;
  float float_var;
  float_var = float(int_var) / 2;
  // Function converting an int to a float.
  cout << "The value of float_var is: " << float_var << endl;
}
```

Conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type.


Higher Data Type
long double
double
float
long
int
short
char
data loss
no data loss
Lower Data Type

Explicit Conversions (Casting)

1. Static cast allows explicit conversion between compatible types. It performs compile-time checks when possible and is safer than C-style cast.

```
double numDouble = 3.14;
int numInt = static_cast<int>(numDouble);
```

2. The dynamic cast can only be used with pointers and references to classes (or void*).

```
class Base { virtual void foo() {} };
class Derived : public Base {};
Base* basePtr = new Derived;
Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
```

3. Const cast is used to add or remove const qualification from variables.

```
const int numConst = 10;
int* numPtr = const_cast<int*>(&numConst);
```

4. Reinterpret cast converts any pointer type to any other pointer type, even of unrelated classes. It's typically used for low-level operations.

```
int* numPtr = new int(5);
double* doublePtr = reinterpret_cast<double*>(numPtr);
```

We can also use the assignment operator in case of objects to copy values of all data members of right-hand object to the object on left hand. The objects in this case are of same data type. But of objects are of different data types we must apply conversion rules for assignment. There are three types of situations that arise where data conversion is between incompatible types.

1. Conversion from built in type to class type.
2. Conversion from class type to build in type.
3. Conversion from one class type to another.

Basic to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defector type conversion from the argument's type to the constructor's class type. Consider the following constructor:

```
string:: string (char*a)
{
        length = strlen (a);
        name=new char[len+1];
        strcpy (name,a);
}
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the constructor in the class string, it can be used for conversion from char* type to string type.

```
string s1  , s2;
char* namel = "Good Morning";
char* name2 = " STUDENTS" ;
s1 = string(namel);
s2 = name2;
```

The program statements

S1 = string (namel);

first converts name 1 from char* type to string type and then assigns the string type values to the object s1. The statement

s2 = name2;

performs the same job by invoking the constructor implicitly. Consider the following example

```
class  time
{
  int hours; int minutes;
  public:
         time (int t) // constructor
         {
                hours = t  /  60;    //t  is  inputted in minutes
minutes  =  t   %  60;
         }
};
```

In the following conversion statements:

time Tl; //object Tl created int period = 160;

Tl = period; //int to class type

The object Tl is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours of Tl will have value 2 arid minutes will have a value of 40 denoting 2 hours and 40 minutes. Note that the constructors used for the type conversion take a single argument whose type is to be converted. In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded = operator.

```
#include <iostream>
using namespace std;
class MyNumber {
private:
    int value;
public:
    // Constructor to initialize MyNumber object from an integer
    MyNumber(int val) : value(val) {}
    // Function to retrieve the value
    int getValue()
       {
        return value;
    }
};
main() {
    int intValue = 100;
    // Implicit conversion from int to MyNumber using constructor
    MyNumber num = intValue;
    cout << "Value stored in MyNumber object: " << num.getValue() <<
endl;
}
```

Example

```
#include <iostream>
using namespace std;
class MyNumber {
private:
    int value;
public:
    // Constructor to initialize MyNumber object from an integer
    MyNumber(int val) : value(val) {}
    // Conversion operator to convert MyNumber to int
    operator int()
       {
        return value;
    }
};
```

```
main() {
    MyNumber num(200);
    // Explicit conversion from MyNumber to int using conversion
operator
    int intValue = num; // static_cast<int>(num);
    cout << "Value stored in intValue: " << intValue << endl;
}
```

Example

```
#include <iostream>
using namespace std;
class MyNumber {
private:
    int value;
public:
    MyNumber(int val) : value(val) {}
    // Overloading the assignment operator to assign integer values to
MyNumber objects
    MyNumber& operator=(int val) {
        value = val;
        return *this;
    }
    // Function to retrieve the value
    int getValue()
       {
        return value;
    }
};
main() {
    int intValue = 30;
    MyNumber num(0);
    // Assigning an integer value to a MyNumber object using the
overloaded assignment operator
    num = intValue;
    cout << "Value stored in MyNumber object: " << num.getValue() <<
endl;
 }
```

Class to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator TypeName ()
{
    //Program statements.
}
```

This function converts a class type data to TypeName. For example, the operator double() converts a class object to type double, in the following conversion function:

```
vector:: operator double ( )
{
    double sum = 0 ;
    for(int I = 0; Size of;
        sum = sum + v[i] * v[i ] ;      //scalar magnitude return
    sqrt(sum);
}
```

The casting operator should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion

inside the function belongs to the object that invoked the function. As a result, function does not need an argument.

In the string example discussed earlier, we can convert the object string to **char*** as follows:

```
string:: operator char*( )
{
        return (str) ;
}
```

Let assume we have a class representing a temperature in Celsius, and we want to convert it to a double representing the temperature in Fahrenheit.

```
#include <iostream>
using namespace std;
class Celsius {
private:
    double temperature;
public:
    Celsius(double temp) : temperature(temp) {}
    // Conversion operator to convert Celsius to double (Fahrenheit)
    operator double()  {
        return (temperature * 9 / 5) + 32; // Conversion formula from
Celsius to Fahrenheit
    }
};
main() {
    Celsius celsiusTemp(25.0); // Temperature in Celsius
    // Implicit conversion from Celsius to double (Fahrenheit)
    double fahrenheitTemp = celsiusTemp;
    cout << "Temperature in Celsius: " << celsiusTemp << " degrees"
<<endl;
    cout << "Temperature in Fahrenheit: " << fahrenheitTemp << "
degrees" <<endl;
}
```

Example

```
#include <iostream>
using namespace std;
class Meters {
    double distance;
public:
    Meters(double dist) : distance(dist) {}
    // Conversion operator to convert Meters to double (kilometers)
    operator double()  {
        return distance / 1000.0; // Conversion from meters to
kilometers
    }
};
main() {
    Meters metersDistance(5000.0); // Distance in meters
    // Implicit conversion from Meters to double (kilometers)
    double kilometersDistance = metersDistance;
    cout << "Distance in Meters: " << metersDistance << " meters"
<<endl;
    cout << "Distance in Kilometers: " << kilometersDistance << "
kilometers" << endl;
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type. **Example**

Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.

Objl is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Objl. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a

constructor or a conversion function. Which form to use, depends upon where we want the type- conversion function to be located, whether in the source class or in the destination class.

We studied that the casting operator function Operator typename( )

Converts the class object of which it is a member to TypeName. The type name may be a built-in type or a user defined one (another class type). In the case of conversions between objects, TypeName refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source class and the result is given to the destination class object.

Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. The argument belongs to the source class and is passed to the destination class for conversion. Therefore, the conversion constructor must be placed in the destination class.

Conversion	**Conversion takes place in**	
	Source class	**Destination class**
Basic to class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class. Consider the following example of an inventory of products in a store. One way of keeping record of the details of the products is to record their code number, total

items in the stock and the cost of each item. Alternatively, we could just specify the item code and the value of the item in the stock. The following program uses classes and shows how to convert data of one type to another.

```
#include <iostream>
#include<conio.h>
using namespace std;
class stock2;
class stock1
{
	int code, item;
	float price;
	public:
		stock1 (int a, int b, float c)
		{
			code=a;
			item=b;
			price=c;
		}
		void disp()
		{
			cout<<"code"<<code <<endl;
			cout<<"Items"<<item <<endl;
			cout<<"Price per item Rs . "<<price <<endl;
		}
		int getcode()
			{return code; }
		int getitem()
			{return item; }
		int getprice()
		{return price;}
		operator float()
		{
			return ( item*price );
		}
};
class stock2
{
	int code;
```

```
        float val;
        public:
                stock2()
                {
                        code=0;
                        val=0;
                }
                stock2(int x, float y)
                {
                        code=x; val=y;
                }
                void disp()
                {
                        cout<< "code"<<code << endl;
                        cout<< "Total Value Rs . " <<val <<endl;
                }
                stock2 (stock1 p)
                {
                        code=p . getcode ();
                        val=p.getitem() * p. getprice ();
                }
};
main ()
{
        stock1 i1(101, 10,125.0);
        stock2 i2;
        float tot_val;
        tot_val=i1 ;
        i2=i1 ;
        cout<<"Stock Details-stockl-type"<<endl;
        i1.disp ();
        cout<<"Stock value"<<endl;
        cout<< tot_val<<endl;
        cout<<"tock Details-stock2-type"<<endl;
        i2.disp();
        getch ();
}
```

Converting one class type to another class type in C++ often involves defining a conversion constructor or a conversion operator. Let assume we have a class representing a distance in meters, and we want to convert it to a class representing the same distance in feet:

```
#include <iostream>
using namespace std;
class Meters {
private:
    double distanceMeters;
public:
    Meters(double meters) : distanceMeters(meters) {}
    double getDistanceMeters() {
        return distanceMeters;
    }
};
class Feet {
private:
    double distanceFeet;
public:
    Feet(double feet) : distanceFeet(feet) {}
    // Conversion constructor to convert Feet to Meters
    Feet(class Meters meters) {
        distanceFeet = meters.getDistanceMeters() * 3.28084;
    }
    double getDistanceFeet() {
        return distanceFeet;
    }
};
main() {
    Meters metersDistance(10.0);
    Feet feetDistance = metersDistance; // Implicit conversion from
Meters to Feet
    cout << "Distance in Meters: " <<
metersDistance.getDistanceMeters() << " meters" <<endl;
    cout << "Distance in Feet: " << feetDistance.getDistanceFeet() <<
" feet" <<endl;}
```