Data Structures and Object Oriented Programming

Lecture 15

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Typecasting

(Type Conversion)

Typecasting (Type conversion)

- The process of converting one predefined type into another is called as type conversion

- C++ facilitates the type conversion into the following two forms for **built-in** data types:

  - ❑ Implicit Type Conversion
  - ❑ Explicit Type Conversion

Implicit Type Conversion

- Conversion performed by the compiler without programmer's intervention whenever differing data types are intermixed in an expression

- The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable)

- Example:

```
int main()
{
    short int x = 1417;
    char ch;
    ch = x;       // where ch is char (1 byte) and x is int (2 bytes)
    return 0;
}
```

Implicit Type Conversion

- **x** was having value 1417 (whose binary equivalent is 0000010110001001)

- **ch** will have lower 8-bits i.e., 10001001 resulting in loss of information.

Implicit Type Conversion

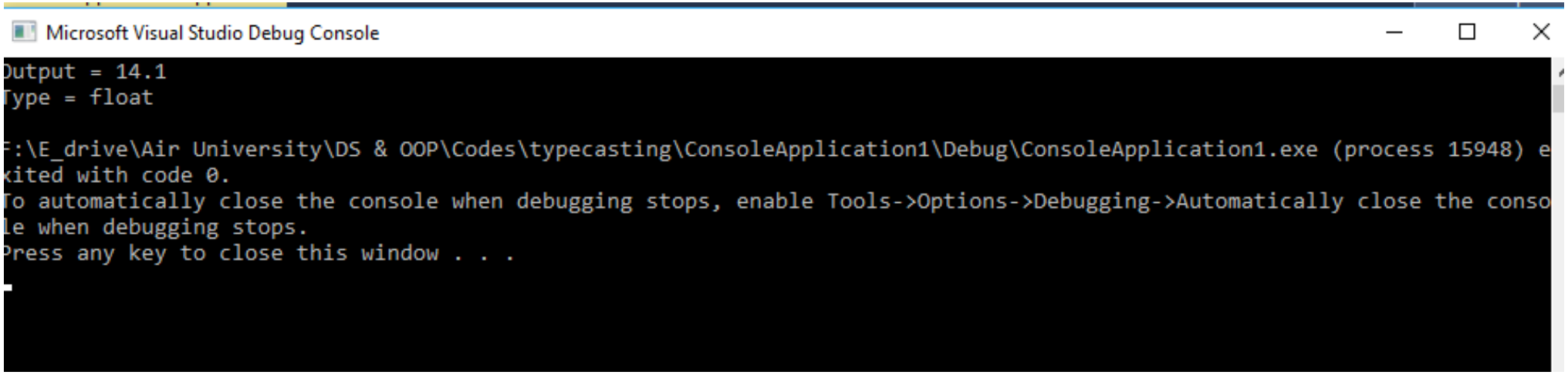
- Another example

```
int main()
{
    int x = 10;

    float y = 4.1;

    cout << "Output = " << x + y << endl;

    cout << "Type = " << typeid(x + y).name() << endl;
}
```



Explicit Type Conversion

- User-defined conversion that forces an expression to be of specific type

```
int main()
{
    int y = 3;
    cout << (float)(y) / 2;

}
```

Output= 1.5

```
int main()
{
    int y = 3;
    cout << (y) / 2;

}
```

Output= 1

Type Conversion for User-defined types

- Now what about **User-defined** data types?
- For user defined classes, there are two types of conversions
  - From any other type to current type
  - From current type to any other type

Type Conversion

- Conversion from any other type to current type:
  - Requires a **constructor** with a single parameter
- Conversion from current type to any other type:
  - Requires an overloaded operator

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Area is defined to take an argument that is a **Circle**

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Area is defined to take an argument that is a **Circle**

However, in main **Area(x)** is called with an argument that is a **double**

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Area is defined to take an argument that is a **Circle**

However, in main **Area(x)** is called with an argument that is a **double**

So C++ tries to convert the argument to a **Circle**. It notices that the constructor for **Circle** essentially converts a **double** into a **Circle**. So it uses the constructor to do the type conversion

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Area is defined to take an argument that is a **Circle**

However, in main **Area(x)** is called with an argument that is a **double**

Implicit type conversion

So C++ tries to convert the argument to a **Circle**. It notices that the constructor for **Circle** essentially converts a **double** into a **Circle**. So it uses the constructor to do the type conversion

Type Conversion – Consider another example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

class AnotherCircle
{
    double radius;
    public:
    AnotherCircle(double x) //constructor
    {
        radius = x;
    }
};
```

```
void Area(Circle N)
{
    cout << "In Area function" << endl;
}

void Area(AnotherCircle N)
{
    cout << "In Area2 function" << endl;
}


int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Type Conversion – Consider another example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

class AnotherCircle
{
    double radius;
    public:
    AnotherCircle(double x) //constructor
    {
        radius = x;
    }
};
```

```
void Area(Circle N)
{
    cout << "In Area function" << endl;
}

void Area(AnotherCircle N)
{
    cout << "In Area2 function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

To implement **Area(x)** compiler could convert **x** to a **Circle** and use **Area(Circle N)**, or to **AnotherCircle** and use **Area(AnotherCircle N)**.

Type Conversion – Consider another example

```
class Circle
{
    double radius;
    public:
    Circle(double x) //constructor
    {
        radius = x;
    }
};

class AnotherCircle
{
    double radius;
    public:
    AnotherCircle(double x) //constructor
    {
        radius = x;
    }
};
```

```
void Area(Circle N)
{
    cout << "In Area function" << endl;
}

void Area(AnotherCircle N)
{
    cout << "In Area2 function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

To implement **Area(x)** compiler could convert **x** to a **Circle** and use **Area(Circle N)**, or to **AnotherCircle** and use **Area(AnotherCircle N)**.

With no way to resolve the ambiguity, the compiler won't compile the program.

Type Conversion – Consider another example

```
class Circle
{
    double radius;
    public:
    explicit Circle(double x) //constructor
    {
        radius = x;
    }
};

class AnotherCircle
{
    double radius;
    public:
    AnotherCircle(double x) //constructor
    {
        radius = x;
    }
};
```

```
void Area(Circle N)
{
    cout << "In Area function" << endl;
}

void Area(AnotherCircle N)
{
    cout << "In Area2 function" << endl;
}


int main()
{
    double x=10;
    Area(Circle(x));
    return 0;
}
```

To resolve the ambiguity, we add an explicit call to the intended constructor

Type Conversion – Consider another example

```
class Circle
{
    double radius;
    public:
    explicit Circle(double x) //constructor
    {
        radius = x;
    }
};

class AnotherCircle
{
    double radius;
    public:
    AnotherCircle(double x) //constructor
    {
        radius = x;
    }
};
```

And **explicit** keywork here

```
void Area(Circle N)
{
    cout << "In Area function" << endl;
}

void Area(AnotherCircle N)
{
    cout << "In Area2 function" << endl;
}


int main()
{
    double x=10;
    Area(Circle(x));
    return 0;
}
```

To resolve the ambiguity, we add an explicit call to the intended constructor

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    explicit Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

Note:

If you do not want the constructor to be used implicitly as a conversion operator, then declare the constructor **explicit**. An **explicit** constructor will be invoked only explicitly and implicit conversion will be suppressed.

Type Conversion – Consider this example

```
class Circle
{
    double radius;
    public:
    explicit Circle(double x) //constructor
    {
        radius = x;
    }
};

void Area(Circle N)
{
    cout << "In Area function" << endl;
}

int main()
{
    double x=10;
    Area(x);
    return 0;
}
```

error: no implicit double -> Circle conversion

Note:

If you do not want the constructor to be used implicitly as a conversion operator, then declare the constructor **explicit**. An **explicit** constructor will be invoked only explicitly and implicit conversion will be suppressed.

Type Conversion

- There is another method for type conversion:
  - “Operator overloading”
  - (*Converting from current type to any other type*)

Type Conversion

- General Syntax:
  - **`TYPE`$_1$`::Operator TYPE`$_2$`();`**

- Must be a member function
- NO return type and arguments are specified
- Return type is implicitly taken to be **`TYPE`$_2$** by compiler

Type Conversion

```
class Circle
{
    double radius;
public:
    Circle(double x) //constructor
    {
        radius = x;
    }
    operator double() //conversion operator overloading
    {
        return radius;
    }
};

int main()
{
    Circle C(10);
    double x=C;
    cout << x;
    return 0;
}
```

Implicit type conversion

Type Conversion

```
class Circle
{
    double radius;
public:
    Circle(double x) //constructor
    {
        radius = x;
    }
    explicit operator double() //conversion operator overloading
    {
        return radius;
    }
};

int main()
{
    Circle C(10);
    double x=C;
    cout << x;
    return 0;
}
```

error: no implicit **Circle -> double** conversion

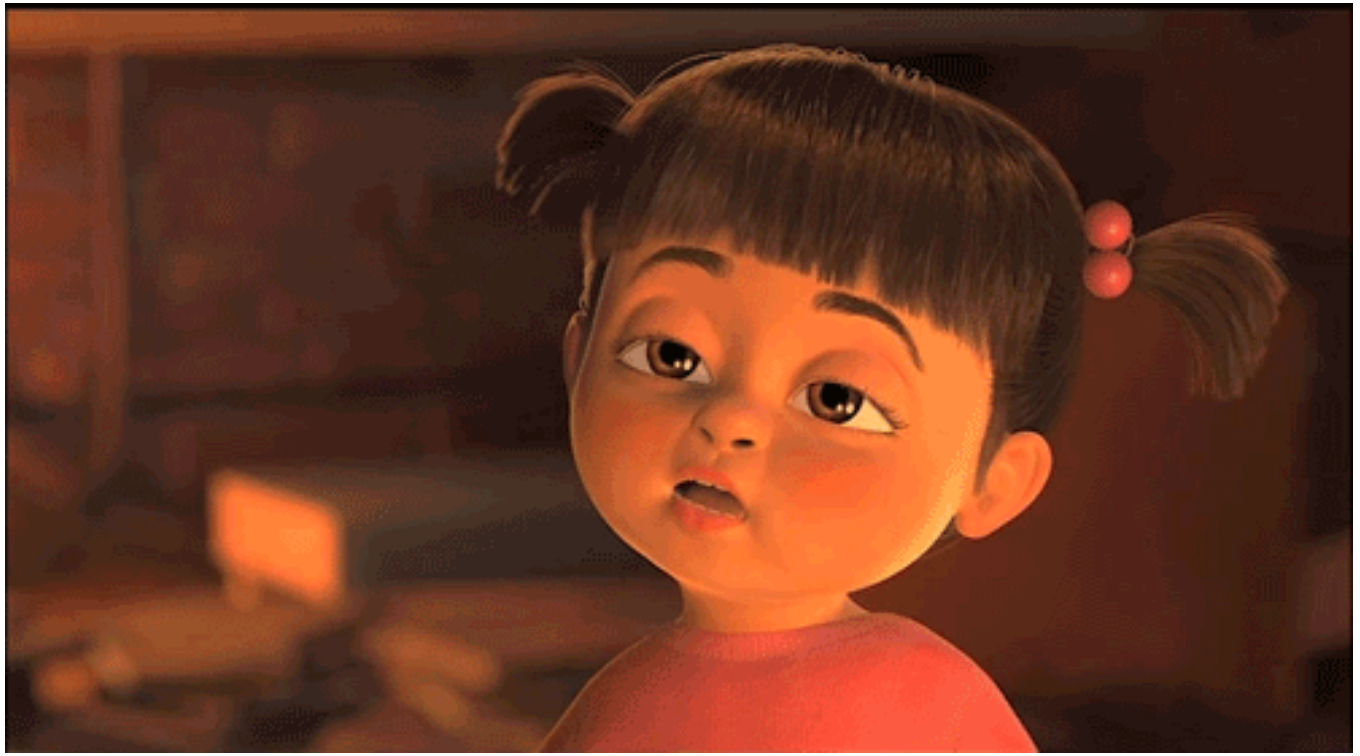
Type Conversion

```
class Circle
{
    double radius;
public:
    Circle(double x) //constructor
    {
        radius = x;
    }
    explicit operator double() //conversion operator overloading
    {
        return radius;
    }
};

int main()
{
    Circle C(10);
    double x=double(C);
    cout << x;
    return 0;
}
```

Explicit type conversion

Thanks a lot



If you are taking a Nap, **wake up**.....Lecture Over