Operator Overloading Syntax

- **Although,** the **syntax of defining prototype**:

```
datatype operator+ (datatype)
```

- **However**, for **some operators**, there is **little bit change** in the above **syntax**:

  ++, **--** operators

  >>, **<<** operators

  & and **[ ]** operators

Overloading ++ and --

- **Operator ++ and -- are different to other operators of C++**

- **We can call them:**
  - either in the form of **prefix** (**++i**) **before an object**
  - or in the form of **postfix** (**i++**) **after an object**
  - But in **both cases**, the **calling object will be i**.

i++ and ++i ?

- **Prefix makes the change**, and then it **processes the variable**
- **Postfix processes the variable**, then it **makes the change.**

```
i = 1;
j = ++i;
```

```
i = 1;
j = i++;
```

Overloaded ++

```
class Inventory
{
   private:
      int stockNum;
      int numSold;
   public:
      Inventory(int stknum, int sold);
       void operator++();
};



void Inventory::operator++()
{
       numSold++;
}
```

Use of the operator ++

```
int main ( )
{
        Inventory someItem(789, 84);
        // the stockNum is 789
        // the numSold is 84

        ++someItem;



}
```

Use of the operator ++

```
int main ( )
{
        Inventory someItem(789, 84);
        // the stockNum is 789
        // the numSold is 84

        ++someItem;

        Inventory Item2 = ++someItem;
        //will this instruction work

}
```

Overloaded ++

```
class Inventory
{
   private:
      int stockNum;
      int numSold;
   public:
      Inventory(int stknum, int sold);
       Inventory& operator++();
};


Inventory& Inventory::operator++()
{
   Inventory *object = new Inventory(0,0);
   numSold++;
   object->numSold = numSold;
   return(*object);
}
```

```
class Inventory   {
   private:
      int stockNum;  int numSold;
   public:
      Inventory(int stknum, int sold) {
          this->stockNum= stknum;
          this->numSold = sold;
      }
     Inventory operator++();

    void Display() {
        cout<<"\n Item number: "<<stockNum<<" sold "<<numSold<<" times";
    }
};
Inventory Inventory::operator++() {
    numSold++;
    Inventory temp(999,numSold);
    return temp;
}

int main() {
    Inventory v(55,11);
    Inventory v2(56,0);
    v2.Display();
    v2=++v;
    v2.Display();
    ++v2;
    v2.Display();
    return 0;
}
```

Using ++
(Prefix Notation)

Problem

- The **definition** of the **prefix operator is easy enough**. It **increments** the **value before any other operation**.

- But, **How will C++ be able to tell the difference between a prefix ++ operator and a postfix ++ operator?**

- **Answer: overloaded postfix operators take a dummy argument** ***(just for differentiation between postfix and prefix)***.

Postfix operator

```
Inventory& Inventory::operator++()  // prefix version
{
    Inventory *object = new Inventory(0,0);
    numSold++;
    object->numSold = numSold;
    return(*object);
}
```

```
Inventory& Inventory::operator++(int)  // postfix version
{
    Inventory *object = new Inventory(0,0);
    object->numSold = numSold;
    numSold++;
    return(*object);
}
```

dummy argument

Postfix and Prefix ++

```
class Inventory   {
   private:
      int stockNum;  int numSold;
   public:
      Inventory(int stknum, int sold) {
          this->stockNum= stknum;
          this->numSold = sold;
      }

      Inventory& operator++();  // prefix version
      Inventory& operator++(int);  // postfix version

    void Display() {
        cout<<"\n Item number: "<<stockNum<<" sold "<<numSold<<" times";
    }
};

Inventory& Inventory::operator++()  // prefix version
{
    Inventory *object = new Inventory(0,0);
    numSold++;
    object->numSold = numSold;
    return(*object);
}
Inventory& Inventory::operator++(int)  // postfix version
{
    Inventory *object = new Inventory(0,0);
    object->numSold = numSold;
    numSold++;
    return(*object);
}
```

```
int main() {
    Inventory v1(55,11);
    Inventory v2 = ++v1;
    Inventory v3 = v1++;
    v1.Display();
    v2.Display();
    v3.Display();
    return 0;
}
```

Postfix and Prefix ++

```
class Inventory   {
   private:
      int stockNum;  int numSold;
   public:
      Inventory(int stknum, int sold) {
          this->stockNum= stknum;
          this->numSold = sold;
      }

      Inventory& operator++();  // prefix version
      Inventory& operator++(int);  // postfix version

    void Display() {
        cout<<"\n Item number: "<<stockNum<<" sold "<<numSold<<" times";
    }
};

Inventory& Inventory::operator++()  // prefix version
{
    Inventory *object = new Inventory(0,0);
    numSold++;
    object->numSold = numSold;
    return(*object);
}
Inventory& Inventory::operator++(int)  // postfix version
{
    Inventory *object = new Inventory(0,0);
    object->numSold = numSold;
    numSold++;
    return(*object);
}
```

```
int main() {
    Inventory v1(55,11);
    Inventory v2 = ++v1;
    Inventory v3 = v1++;
    v1.Display();
    v2.Display();
    v3.Display();
    return 0;
}
```

Assignment Operator (=)

```
class Employee
{     private:
        int idNum;
        double salary;
    public:
         Employee ( ) { idNum = 0, salary  = 0.0; }
         void setValues (int a, int b);
         void operator= (double );
}


void Employee::setValues ( int idN , double sal )
{
        salary = sal;        idNum  = idN;
}

void Employee::operator = (double sal)
{       salary = sal;         }
```

Assignment Operator (=)

```
int main ( )
{
        Employee emp1;
        emp1.setValues(10,33.5);

        Employee emp2;
        emp2 = 44.6; // emp2 is calling object


}
```

Assignment Operator (=)

```
class Employee
{    private:
       int idNum;
       double salary;
   public:
       Employee ( ) { idNum = 0, salary  = 0.0; }
       void setValues (int a, int b);
       void operator= (Employee &emp );
}

void Employee::setValues ( int idN , double sal )
{      salary = sal;        idNum  = idN;        }


void Employee::operator = (Employee &emp)
{      salary = emp.salary;          }
```

Assignment Operator (=)

```
class Employee
{    private:
       int idNum;
       double salary;
   public:
       Employee ( ) { idNum = 0, salary  = 0.0; }
       void setValues (int a, int b);
       void operator= (Employee &emp );
}

void Employee::setValues ( int idN , double sal )
{      salary = sal;        idNum  = idN;        }


void Employee::operator = (Employee &emp)
{      salary = emp.salary;          }
```

Comparison Operator (==)

```
class Employee
{     private:
        int idNum;
        double salary;
    public:
         Employee ( ) { idNum = 0, salary  = 0.0; }
         void setValues (int a, int b);
         bool operator== (Employee &emp );
}

void Employee::setValues ( int idN , double sal )
{       salary = sal;          idNum  = idN;          }


bool Employee::operator == (Employee &emp)
{       return (salary == emp.salary);          }
```

Comparison Operator (==)

```
int main ( )
{
        Employee emp1;
        emp1.setValues(10,33.5);

        Employee emp2;
        emp2.setValues(10,33.1);

        if ( emp2 == emp1 )
            cout <<“Both objects have equal value”;
        else
             cout <<“objects do not have equal value”;

}
```

Subscript operator [ ]

- With the help of **[ ] operator**, we can **define array style syntax** for **accessing** or **assigning individual elements** of **classes**

```
Student semesterGPA;
semesterGPA[0] = 3.5;
semesterGPA[1] = 3.3;
```

Subscript operator[ ]

```
class Student
{   private:
       double gpa[8];
    public:
       Student ()
       {     gpa[0]=3.5;  gpa[1]=3.2;   gpa[2]=4;    gpa[3]=3.3;
             gpa[4]=3.8;  gpa[5]=3.6;   gpa[6]=3.5;  gpa[7]=3.8;
       }
       double& operator[] (int Index);
}


double& Student::operator [ ] (int Index)
{
       return gpa[Index];
}
```

Subscript operator[ ]

```
class Student
{   private:
        double gpa[8];
    public:
        Student ()
        {    gpa[0]=3.5;  gpa[1]=3.2;  gpa[2]=4;    gpa[3]=3.3;
             gpa[4]=3.8;  gpa[5]=3.6;  gpa[6]=3.5;  gpa[7]=3.8;
        }
        double& operator[] (int Index);
}

double& Student::operator [ ] (int Index)
{
        return gpa[Index];
}
```

Subscript operator[ ]

```
int main ( )
{
      Student semesterGPA;
      semesterGPA[0] = 3.7;

     double gpa = semesterGPA[4];

}
```

Subscript operator[ ]

- **How the statement executes?**

  semesterGPA[0]=3.7;

- The **[ ]** has **highest priority** than the **assignment operator**, therefore **semesterGPA[0]** is **processed first**.

- **semesterGPA[0]** calls **operator [ ]**, which then **return** a **reference** of **semesterGPA.gpa[0]**.

Subscript operator[ ]

```
int main ( )
{
      Student semesterGPA;
      semesterGPA[0] = 3.7;

      double gpa = semesterGPA[4];

}
```

Subscript operator[ ]

- The **return value** is **reference** to **semesterGPA.gpa[0]**, and the **statement** semesterGPA[0] = 3.7 is actually **integer assignment**.

```
int main ( )
{
      Student semesterGPA;
      semesterGPA[0] = 3.7;
      // the above statement is processed like as
      semesterGPA.gpa[0] = 3.7
}
```

Calling an overloaded operator from native data types

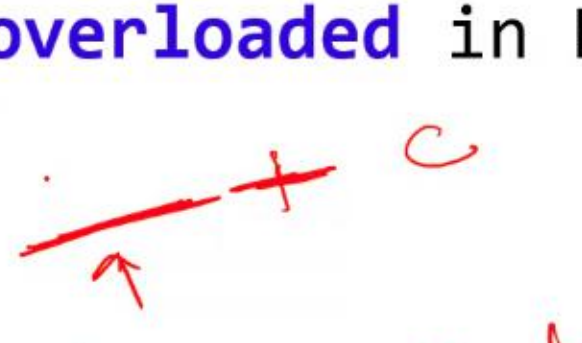
- In previous lectures, we were calling an overloaded operator of a class only with the help of its object (instance)

```
Point a, b, c;
// where + is overloaded in Point class
a = b + c;
```

Calling an overloaded operator from native data types

- In **previous lectures**, we were **calling an overloaded operator** of a **class only with the help of its object (instance)**

```
Point a, b, c;
// where + is overloaded in Point class
a = b + c;
```



Calling an overloaded operator from native data types

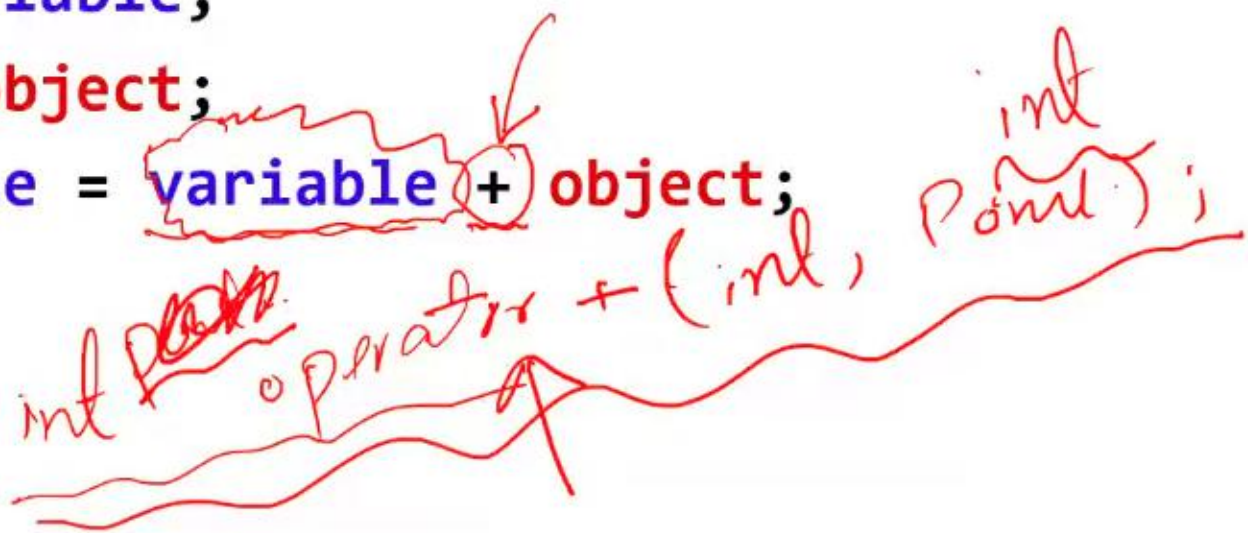
- But, **Can we call an overloaded operator of a class from the variables of native data types?**

```
int variable;
Point object;
variable = variable + object;
```

Calling an overloaded operator from native data types

- But, **Can we call an overloaded operator of a class from the variables of native data types?**

```
int variable;
Point object;
variable = variable + object;
```



Calling an overloaded operator from native data types

- But, **Can we call an overloaded operator of a class from the variables of native data types?**

```
int variable;
Point object;
variable = variable + object;
```

- In above example, **it seems that we need to overload + operator** for **int (native-data type)**.

- **But in operator overloading we can't change the functionality of int data type**

Calling an overloaded operator from native data types

- **Friend functions** can **help us** in **solving this problem.**

- **Friend Function:** ***A Friend function does not need an object of a class for its calling***.

- Thus, **with a simple trick** we can **set parameter1** of an **overloaded object** to **native data type** and **parameter2** to **class object**.

Friend Functions

- **Friend functions:** can be **given special grant to access private** and **protected members**. A **friend function can be**:
  a) **method of another class**
  b) **global function**

- **Friends** should be **used only for limited purpose**, too many **functions declared as friends** with protected or private data access, **lessens the value of encapsulation**

Calling an overloaded operator from native data types

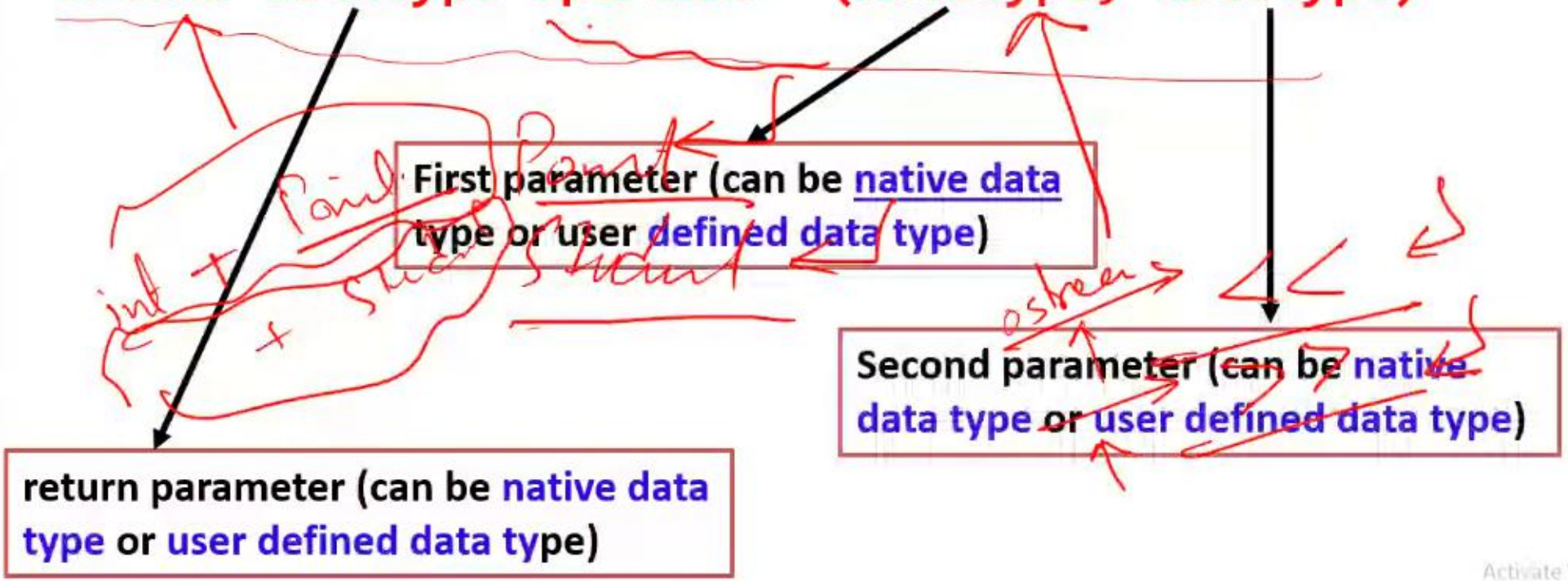
- **For friend function the syntax is changed**, the **first operator** is moved from **calling object** to **first parameter of function**.

```
friend datatype operator+ (datatype, datatype)
```

Calling an overloaded operator from native data types

- **For friend function the syntax is changed**, the **first operator** is moved from **calling object** to **first parameter of function**.

```
friend datatype operator+ (datatype, datatype)
```



Example

```
class Point
{
    private:
        float m_dX, m_dY, m_dZ;
    public:
        Point(float dX, float dY, float dZ)
        {
                m_dX = dX;
                m_dY = dY;
                m_dZ = dZ;
         }
        friend float operator+ (float var1, Point &p);
};
```

Example

```
float operator+(float var1, Point &p)
{
      return ( var1 + p.m_dX);
}

int main (void)
{
      float variable = 5.6;
      Point cPoint ( 2, 9.8, 3.3 );
      float returnVar;
      returnVar = variable + cPoint;
      cout << returnVar; // 7.6
      return 0;
}
```

Overloading iostream operators >> and <<

- We can **use friend function** for **overloading iostream operators** (>> or <<).

- Usually **iostream operators** (>> or <<) are **not called** from an **object of the class**

```
Point p;
cin  >> p;
cout << p;
```

where ***cin*** and ***cout*** are object of **iostream** class

Example

```
class Point
{
   private:
        float m_dX, m_dY, m_dZ;
   public:
        Point(float dX, float dY, float dZ)
        {
                m_dX = dX;
                m_dY = dY;
                m_dZ = dZ;
         }
      friend ostream& operator<< (ostream &out, Point &cPoint);
      friend istream& operator>> (istream &in, Point &cPoint);
};
```

Example

```
ostream& operator<< (ostream &out, Point &cPoint)
{
    out << "(" << cPoint.m_dX << ", " <<
    cPoint.m_dY << ", " << cPoint.m_dZ <<")";
    return out;
}


istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}
```