Operator Overloading

Department of Software Engineering,

National University of Computer & Emerging Sciences,

Islamabad Campus

contents

- Fundamentals of Operator Overloading, Overloading Binary Operators, Overloading the Binary Stream, Insertion and Stream Extraction Operators, Overloading Unary Operators, Overloading the Unary Prefix and Postfix ++ and -- Operators, Operators as Member Functions vs. Non-Member Functions, Converting between Types, explicit Constructors

Operator Overloading – Part 1

Operator Overloading

- **The method of defining additional meanings for operators is known as operator overloading**

- **Enables** an **operator** to **perform different operations depending** upon the **type of operands**

- The **basic operators** i.e. **+, -, *, / normally works** with **basic types** i.e. **double**, **float**, **int**, **long**. (defined in C++)

- So, **how can these operators be applied to user-defined data types?**

Operator Overloading Motivation

- **Don't want to create new operators** for **user-defined data types**
- **Conclusively, Operator overloading:**
  - **Enabling** C++'s **operators** to **work** with **class objects**
  - **Using traditional operators** with **user-defined objects**
  - **Requires great care**; **when overloading is misused, program difficult to understand**

  - Examples of **already overloaded operators**:
    - Operator **<<** is both the **stream-insertion** operator and the **bitwise left-shift operator**

Operator Overloading

- **Example (already overloaded operator):**

type `int` /type `int`

`9 / 5`

operator performs *int* division

type `long` /type `long`

`9L / 5L`

operator performs *long* division

type `double` /type `double`

`9.0 / 5.0`

operator performs *double* division

type `float` /type `float`

`9.0f / 5.0f`

operator performs *float* division

How to Overload an Operator?

- ***An operator can be overloaded by declaring a special member function in the class***

- **Name of the member function** is **operator** that is **followed by operator symbol** e.g., **operator+**, **operator/,** etc.

- Can be **independent function** (*except for the following operators:* ***( ), [ ], ->*** *or* ***any of the assignment operators***)

- Can be a class's **member function** (**must be non-static**)

How to Overload an Operator?

- **Member function:** If the **left operand** of that **particular class** is an **object of the same class**, then the **overloaded operator** is said to be **implemented by a member function**.

- **Non-member function:** If the **left operand of that particular class** is **an object of a different class**, then the **overloaded operator** is said to be **implemented by a non-member function**

Overload as Member or Non-Member Function

- If it is a ***unary operator***, implement it as a ***member* function**.

- If a **binary operator** **treats** ***both operands equally*** (it leaves them unchanged), implement this operator as a ***non-member* function**.

- If **a binary operator does *not* treat both** of its operands ***equally*** (usually it will change its left operand), it might be useful to make it a ***member* function of its left operand's type**

- the << operator (when used for stream output, not bit shifting) gets an ostream as its first parameter, so it can't be a member of your class.

Syntax to Overload an Operator

```
returnType operator opsymbol(parameters){  }
```

any type *keyword* *operator symbol* *function body*

- ***return-type*** may be **whatever** the **operator returns**
- ***Operator symbol*** may be any ***overloadable operator***

Example:

```
void   operator+ (parameters){  }
```

any type *keyword* *operator symbol* *function body*

Operator Overloading

- **Operators** are **really functions**
  - They have **arguments**, they **return values**
  - The **only difference** is that their **names** take on a **specific form**:

```
Operator+, operator[ ]
```

- **Overloading provides concise notation:**

```
// without operator overloading
object2 = object1.add(object2);

// with operator overloading
object2 = object2 + object1;
```

Restriction on Operator Overloading

- With **operator overloading** **we cannot change:**
  1. **How operators act on built-in data types:**
     - i.e., ***cannot change integer addition***
  2. **Precedence of operator (order of evaluation)**
     - **Use parentheses** to **force order-of-operations**
  3. **Association rules (*left-to-right* or *right-to-left* evaluation)**
  4. **Number of operands**
     - **i.e., &** is **unary**, only acts on **one operand**
  5. **Cannot create new operators**
  6. **Operators must be overloaded explicitly:**

     i.e., Overloading + , does not overload +=

Restriction on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

http://www.stroustrup.com/bs_faq2.html#overload-dot

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Operator =, operator &

- **Operator =** and **operator &** are **overloaded implicitly for every class**, so they can be used for each class objects.

- **operator = performs member-wise copy** of the **data members**.

- **operator & returns the address of the object** in **memory**.

Function Overloading

- An **overloaded function** is one which has the **same name** but **several different forms**.

- **For example**: we can **overload the constructor for the Date class:**

default	**Date d;**
initializing	**Date d(9,22,20);**
copy	**Date d1(d);**
other	**Date d(“Sept”,22,2020);**

Operator Overloading

- The **operator** “**+**” also has **different semantics depending** on the **type** of its “**arguments**”

- **Example**

```
int i, j;
double d, e;
i + j;    //add two int
i + d;    //add an int and a double
```

Operator Overloading Syntax

```
a = b + c;
```

```
datatype operator + (datatype) { … }
```

Second parameter (can be native data type or user defined data type)

Remember operator+ is a function, and it will be called with the help of any object, thus the first parameter is the calling object

return parameter

(can be native data type or user defined data type)

Operator Overloading Syntax

```
datatype operator+ (datatype) {   }
```

Example (1):

```
class myClass
{
   int operator+ ( int );
}

int main ( )
{
   int a, b;
   myClass object;
   a = object + b;
}
```

Operator Overloading Syntax

datatype operator+ (datatype)

Example (2):

```
class myClass
{
   int operator+ ( myClass &a );
}

int main ( )
{
   int a;
   myClass object1, object2;
   a = object1 + object2;
}
```

Operator Overloading Syntax

datatype operator+ (datatype)

Example (3):

```
class myClass
{       myClass operator+ ( int );    }


int main ( )
{
        int a = 5;
        myClass object1, object2;
        object2 = object1 + a;
}
```

Operator Overloading Syntax

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

Example (4):

```
class myClass
{
   myClass operator+ ( myClass &a );
}


int main ( )
{
      myClass object1, object2, object3;
      object3 = object1 + object2;
}
```

Implementing Overloaded Operators

- The **compiler** uses the **types of arguments** to **choose** the **appropriate overloading**.

```
int v1, v2;
v1 + v2; // int +

float s1, s2;
s1 + s2; // float+
```

Extended Example

Employee class and objects

```
class Employee
{
   private:
     int idNum;
     double salary;
   public:
     Employee(int  id, double salary);
     double addTwo (Employee& emp);
     double operator+ (Employee& emp);
     double getSalary() { return salary; }
};
```

The member functions ‘addTwo’ and operator+

```
//function notation
double Employee::addTwo(Employee& emp)
{
   double total;
   total = this->salary + emp.getSalary();
   return total;
}

//operator overloading notation
double Employee::operator+(Employee& emp)
{
   double total;
   total = this->salary + emp.getSalary();
   return total;
}
```

Using the Member Functions

```
double sum;
Employee Clerk (111, 10000), Driver (222, 6000);


// these three statements do the same thing
sum = Clerk.addTwo(Driver);
sum = Clerk.operator+(Driver);
sum = Clerk + Driver;


// the syntax for the last one is the most natural
// and is easy to remember because it is consistent
// with how the + operator works for everything else
```

Multiple Operators

- **Often**, you **may need to reference** an **operator more than once** in an expression:

  Example:

```
total = a + b + c;
```

- **But this can cause big problems when operator overloading is involved**

- See next example…

Client Code for Class Employee

```
void main()
{
    Employee Clerk(115, 20000.00);
    Employee Driver(256, 15500.55);
    Employee Secretary(567, 34200.00);
    double sum;

    sum = Clerk + Driver + Secretary;

    cout << “Sum is “ << sum;
}
```

The Problem

- **Operator +** is **left to right associative**, so **Clerk** and **Driver** are **added**. The **result is a double**.

- Now that **double** is on the **left** and an **Employee is on the right** (i.e., ***Secretary***)

- **BUT THE OPERATOR +** is only **defined for arguments** of **type Employee**, **not for double**

The Problem Gets Worse

- It would seem that all **we have to do** is **write another version** of the **overloaded operator** to work with the argument (double)

- **But…**
  - although **we could overload an operator** to work like this:

```
sum = Secretary + num;
```

- We **cannot overload (with member function)** one **like this**:

```
sum = num + Secretary;    // why not?
```

The Answer

- We **cannot overload +** for a **double (a native type)**

- The **real solution** is to **make sure** that your **operator+ function never returns a double (or any other native type)**.

- An operator to add **Employees should return an Employee** (see next slide)

Extended Example

Employee class and objects

```
class Employee
{
   private:
     int idNum;
     double salary;
   public:
     Employee(int  id, double salary);
     Employee operator+ (Employee& emp);
     double getSalary() { return salary; }
}
```

Solution Example

```
Employee Employee::operator+(Employee& emp)
{
    Employee total(999,0);  // dummy values
    total.salary = salary + emp.salary;
    return(total);
}
```

Client Code for Class Employee

```
void main()
{
    Employee Clerk(115, 20000.00);
    Employee Driver(256, 15500.55);
    Employee Secretary(567, 34200.00);
    Employee sum(0, 0.0);
    sum = Clerk + Driver + Secretary;
}
```

Invoking Objects

- If the operator is binary but there is only one explicit argument, the 'invoking instance' is assumed to be the one on the left hand side of the expression.

```
class Date
{
    public:  // member functions
    Date operator=(Date& d);
};


int main (void)
{
    s1 = s2;  // instead of s1.operator=(s2);
}
```

Non-member Operator Overloading Function

```
class myClass
{       private:
            int x;
        public:
        myClass(int x=0) { this->x=x; }

        //Getter function
        int getX(){
            return x;
        }

        //Setter function
        void setX(int x){
            this->x=x;
        }
};


myClass operator+ (myClass  &a, myClass  &b)
{
    myClass temp;
    temp.setX(a.getX()+b.getX());
    return temp;
}


int main ( )
{
        myClass object1,object2,object3;
        object1.setX(10);
        object2.setX(5);
        object3 = object1+object2;
        cout<<object3.getX();
}
```

The Answer (double+object)

- We **cannot overload +** for a **double (a native type)**
- The **real solution** is to **make sure** that your **operator+ function never returns a double** (**or any other native type**).
- An operator to add **Employees should return an Employee** (see next slide)
- **Solution 2: make a non-member operator overloading function.**

Assignment Operator =

- **Operator =** is **overloaded implicitly for every class**, so they **can be used** for **each class objects**.

- **operator = performs member-wise copy** of the **data members**.

- **However**, there is a **problem with implicitly overloaded operator**...(see next slide)

Using implicit Overloaded Assignment Operator

```
// A class without user defined assignment operator
class Test
{
    int *ptr;
public:
    Test (int i = 0)      { ptr = new int(i); }
    void setValue (int i) { *ptr = i; }
    void print()          { cout << *ptr << endl; }
};

int main()
{
    Test t1(5);
    Test t2;
    t2 = t1;
    t1.setValue(10);
    t2.print();
    return 0;
}
```

Output = 10

Operator Overloading – Review

- The **variables** of **native data types** can **perform a number of different operations** (functions) using operators (**+, - , / , *, etc**)
  - Example: **a + b * c**
  - Example: **if (a < b)**

- However, with **user defined (classes) objects** we **can not use operators**:
  - Example: **class obj1, obj2;**

    if (ob1 < obj2)

Operator Overloading – Review

- **To add operator functionality** in the class
- First **create a function** for the **class**
- Set the **name** of the **function** with **the operator name**

  operator + for the **addition operator** ‘**+**’

  operator > for the **comparison operator** ‘**>**’

Overloading > operator

```
bool Employee::operator>(Employee& e)
{
      return(seniority > e.seniority);
}
```

called from the program like this:

```
if (emp1 > emp2)
```

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 is 2, j is 2)
```

```
i = 1;
j = i++;
(i is 2, j is 1)
```

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;

        Inventory Item2 = ++someItem;
        //will this instruction work
// Will not work as the overloaded function does not return anything
}
```

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);
}
```

Using ++ (Prefix Notation)

```
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;
}
```

```
Item number: 56 sold 0 times
Item number: 999 sold 12 times
Item number: 999 sold 13 times
```

Problem

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

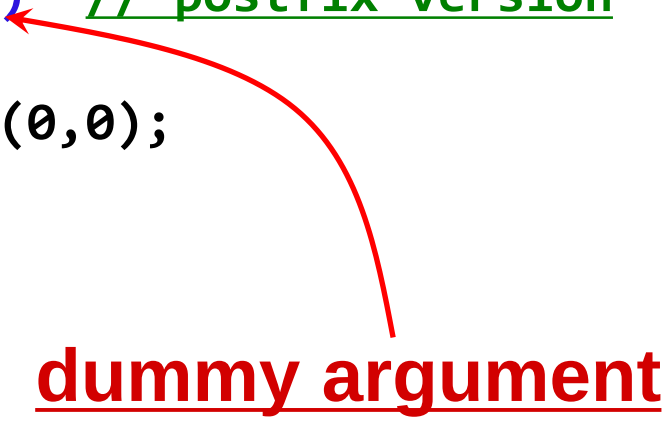
- But, **How will C++ be able to tell the difference between a prefix ++ operator and a postfix ++ operator?**

- **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);
}
```



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);
}
```

```
Item number:  sold 13 times
Item number:  sold 12 times
Item number:  sold 12 times
```

```
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 (=)

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

      Employee emp2;
      emp2 = emp1; // emp2 is calling object

}
```

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& opeator[] (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[ ]

- 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
}
```

```cpp
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
   private:
      int arr[SIZE];

   public:
      safearay() {
         register int i;
         for(i = 0; i < SIZE; i++) {
           arr[i] = i;
         }
      }

      int &operator[](int i) {
         if( i > SIZE ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
         }

         return arr[i];
      }
};

int main() {
   safearay A;

   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;

   return 0;
}
```

Calling an overloaded operator from native data types

- **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

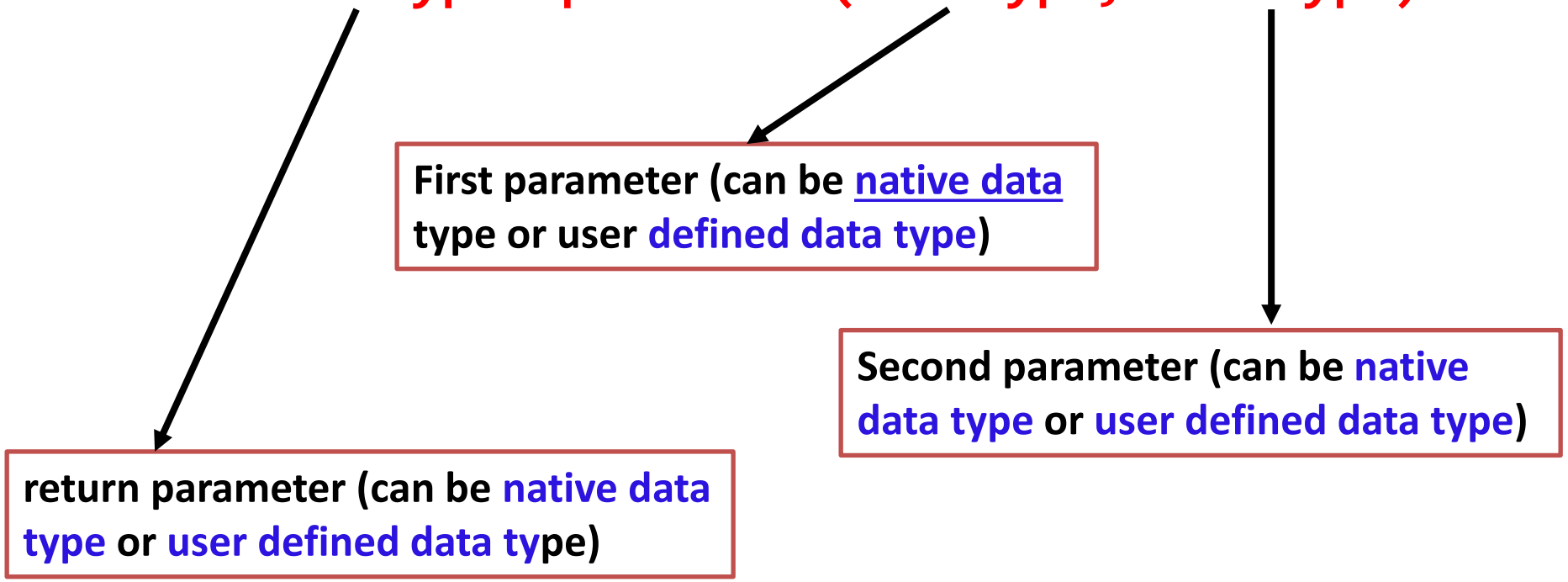
- **Friend functions:** can be **given special grant to access private** and **protected members**. A **friend function can be**:
  a) **method of another class**
  b) **global function**

- **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)
```



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

Overloading iostream operators >> and <<

- We can define the **prototype** of **iostream operators** (**>>** and **<<**) with the help of **Friend function**, and **then** **we do not need any object of a class for their calling**.

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;
}
```

Example

```
int main (void)
{
    cout << "Enter a point: " << endl;
    Point cPoint;
    cin >> cPoint;

    cout << "You entered: " << cPoint << endl;

}
```

Overloading iostream operators >> and <<

- But, what is the advantage of returning references of iostream objects

  friend ostream& operator<< (ostream &out, Point &cPoint);
  friend istream& operator>> (istream &in, Point &cPoint);

- In order to understand above, let take a look on the first and second parameters in case of >> and <<

```
Point cPoint;
cin >> cPoint; // cin is first parameter and
               // cPoint is second parameter
```

- Is above statement (cin >> cPoint) returning anything?

Overloading iostream operators >> and <<

- **Is** above **statement** (**cin >> cPoint**) **returning anything**?

- It is **returning reference of iostream object**, thus in above **statement** the **cin** reference is returned that can be further used for

```
Point cPoint1, cPoint2;
cin >> cPoint1 >> cPoint2;
```

- In above statement (**cin >> cPoint1**) **returns a reference** of **cin** which is **further used for** (**cin >> cPoint2**)

Data Conversion

- **Conversion** between **basic types**
- **Conversion** between **Objects** and **basic types**
- **Conversion** between **Objects** of **different classes**

Conversion b/w Basic Types

- When we use **two different Types**:

```
intvar=floatvar;  //the compiler calls a special routine
                  // that converts this value from
                  // floating point format to integer
                  // format.
```

- There are **many such conversion routines** build in C++ compiler and **called upon** when **any such conversion** is **required**.

Explicit Conversion

- **if we want to force compiler to convert data from one native type to other**, we can use **explicit casting,** `intvar=int(floatvar)`

- it is **obvious** in **listing** that **int( ) conversion function** will convert from **float to int**.

- This **explicit conversion uses same build in routines**.

Conversion Between Objects and Basic Types

- To convert from a **basic type** (i.e., ***float***) to **object types** (i.e., ***Distance***), **we use a constructor with one argument**.

```
Distance(float meters){ }
```

- This **function** is **called** when an **object of type Distance** is **created** with a **single argument**.

- This **conversion allows a floating value to be assigned to a Distance type object.**

```
Distance dist1=2.35; // constructor
```

- Above, **one argument constructor** will be **called**.

- **Same conversion** can be **achieved** by **providing overloaded ‘=‘ operator** which takes a **float value as argument**.

From User Defined to Basic

- What if **want to go from user-defined types(Distance)** to **native type(float)?**

- The trick here is to **overload the cast operator**, **creating something called** a "**Conversion function**".

```
operator float() {
  return floating_rep;
}
```

NOTE: the conversion function does not need return type

Conversion functions have no arguments, and the return type is implicitly the conversion type

From User Defined to Basic

- This **operator takes the value** of the **distance object** of which it is a **member**, **converts** this **value to a float value** and **returns this value**.

- **This operator can be called like this:**

```
float floatmtrs = float(dist2);
float floatmtrs = dist2;
```

both statements have exactly same effects.

From User Defined to Basic

```
class Employee
{  private:
      float salary;
    public:
      Employee ( float sal ) { salary = sal; }
      operator float();
}

Employee::operator float(  )
{
     return salary;
}
```

From User Defined to Basic

```
int main ( )
{
      Employee emp1(33.5);

      float value = float(emp1);
      cout << value; // 33.5
}
```

Conversion between Objects of Different Classes

- Both **methods shown before** can be applied to **conversion between objects** of **different basic types** (i.e., ***one argument constructor, and conversion function***).

Example

- There are **two classes**, **Polar** and **Rec**.

- We **want** to be able **to convert an object** of **type Polar** to **an object of type Rec**.

  i.e., `rec=pol;`

provide one argument constructor in class Rec.

```
Rec(Polar p){
    //process p's data and convert(assign)
    //it into object Rec.
}
```

```
   rec=pol;
  /*one argument constructor will be called to
perform the conversion*/
```

Pitfalls of Operator Overloading and Conversion

- With the **help of Operator overloading we can create entirely new language**.

- For example for **a = b + c** we can implement a new methodology on **user-defined types**.

- **But care should be taken as doing something different than native data types could make your code hard to read and understand**

Use Similar Meanings

- **Implement** the **operation** **of** **overloaded operator** **similar** to **native data types**.

- For example, ***adding two strings*** ***makes sense as we take adding as "concatenation" of two strings***

- but **adding two "Employees"** having ***personal data in them doesn't make much sense***.

Show Restraint

- Make sure that **user of your class** will **easily know the purpose** of **overloading an operator**.
- **Sometimes it make more sense** to **use functions**, as **their names may suggest what they are to perform**.
- **Use overloaded operator sparingly** and **only when the usage is obvious.**

Case Study: A `Date` Class

- The **following example** creates a **Date** class with
  - An **overloaded increment operator** to **change the day, month and year**
  - An **overloaded += operator**
  - A **function** to **test for leap years**
  - A **function** to **determine if a day is last day of a month**

```
1  // Fig. 8.6: date1.h
2  // Definition of class Date
3
4
5  #include <iostream>
6
7
8
9  class Date {
10    friend ostream &operator<<( ostream &, Date & );
11
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
14    void setDate( int, int, int ); // set the date
15    Date &operator++();            // preincrement operator
16    Date operator++( int );        // postincrement operator
17    Date &operator+=( int ); // add days, modify object
18    bool leapYear( int );    // is this a leap year?
19    bool endOfMonth( int );  // is this end of month?
20
21 private:
22    int month;
23    int day;
24    int year;
25
26    static int days[];       // array of days per month
27    void helpIncrement();          // utility function
28 };
29
30
```

```
31 // Fig. 8.6: date1.cpp
32 // Member function definitions for Date class
33 #include <iostream>
34 #include "date1.h"
35
36 // Initialize static member at file scope;
37 // one class-wide copy.
38  int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
39                           31, 31, 30, 31, 30, 31 };
40
41 // Date constructor
42 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
43
44 // Set the date
45 void Date::setDate( int mm, int dd, int yy )
46 {
47    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
48    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
49
50    // test for a leap year
51    if ( month == 2 && leapYear( year ) )
52       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
53    else
54       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
55 }
56
57 // Preincrement operator overloaded as a member function.
58 Date &Date::operator++()
59 {
60    helpIncrement();
61    return *this;  // reference return to create an lvalue
62 }
63
```

```
64 // Postincrement operator overloaded as a member function.
65 // Note that the dummy integer parameter does not have a
66 // parameter name.
67 Date Date::operator++( int )
68 {
69    Date temp = *this;
70    helpIncrement();
71
72    // return non-incremented, saved, temporary object
73    return temp;   // value return; not a reference return
74 }
75
76 // Add a specific number of days to a date
77 Date &Date::operator+=( int additionalDays )
78 {
79    for ( int i = 0; i < additionalDays; i++ )
80       helpIncrement();
81
82    return *this;    // enables cascading
83 }
84
85 // If the year is a leap year, return true;
86 // otherwise, return false
87 bool Date::leapYear( int y )
88 {
89    if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
90       return true;   // a leap year
91    else
92       return false;  // not a leap year
93 }
94
95 // Determine if the day is the end of the month
96 bool Date::endOfMonth( int d )
97 {
```

postincrement operator has a dummy **`int`** value.

```
98    if ( month == 2 && leapYear( year ) )
99       return d == 29; // last day of Feb. in leap year
100   else
101      return d == days[ month ];
102}
103
104// Function to help increment the date
105void Date::helpIncrement()
106{
107   if ( endOfMonth( day ) && month == 12 ) {  // end year
108      day = 1;
109      month = 1;
110      ++year;
111   }
112   else if ( endOfMonth( day ) ) {            // end month
113      day = 1;
114      ++month;
115   }
116   else       // not end of month or year; increment day
117      ++day;
118}
119
120// Overloaded output operator
121ostream &operator<<( ostream &output, Date &d )
122{
123   char *monthName[ 13 ] = { "", "January",
124      "February", "March", "April", "May", "June",
125      "July", "August", "September", "October",
126      "November", "December" };
127
128   output << monthName[ d.month ] << ' '
129          << d.day << ", " << d.year;
130
131   return output;   // enables cascading
132}
```

```
133// Fig. 8.6: fig08_06.cpp
134// Driver for class Date
135#include <iostream>
136
137
138
139
140#include "date1.h"
141
142int main()
143{
144   Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
145   cout << "d1 is " << d1
146        << "\nd2 is " << d2
147        << "\nd3 is " << d3 << "\n\n";
148
149   cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
150
151   d3.setDate( 2, 28, 1992 );
152   cout << "  d3 is " << d3;
153   cout << "\n++d3 is " << ++d3 << "\n\n";
154
155   Date d4( 3, 18, 1969 );
156
157   cout << "Testing the preincrement operator:\n"
158        << "  d4 is " << d4 << '\n';
159   cout << "++d4 is " << ++d4 << '\n';
160   cout << "  d4 is " << d4 << "\n\n";
161
162   cout << "Testing the postincrement operator:\n"
163        << "  d4 is " << d4 << '\n';
164   cout << "d4++ is " << d4++ << '\n';
165   cout << "  d4 is " << d4 << endl;
166
167   return 0;
168}
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```

String Library

- We will use **operator overloading to build String library**

- **Overloaded Operators**

  = (for text assignment)

  == (for comparison between two strings)

  ostream and **istream** (for cin and cout)

  + (for adding two strings)

  [ ] (for retrieving or changing single character in string)

String Library

```
class String
{
   private:
       char *text;
   public:
   String(char *str)
   {
       text = new char[strlen(str)];
       strcpy(text,str);
   }

   friend ostream& operator<<(ostream &,String &str);
   friend istream& operator>>(istream &,String &str);
   void operator= (char *str);

};
```

String Library

```
   String& operator+(String &str);
   String& operator+(char *str);

   bool operator==(String &str);
   bool operator==(char *str);

   char& operator[] (int Index);

};


bool String::operator == ( char *str )
{
   bool val;
   val = strcmp(text,str);
   if ( val == 0 )
       return true;
   else
       return false;
}
```

String Library

```
bool String::operator == ( String &par)
{
  bool val;
  val = strcmp(text,par.text);
  if ( val == 0 )
     return true;
  else
     return false;

}

void String::operator = (char *str)
{
  text = new char[ strlen(str) ];
  strcpy(text,str);
}
```

String Library

```
String& String::operator + (String &par)
{
  String iSt = "";
  int length = 0;
  length = strlen(text);
  length += strlen(par.text);
  iSt.text = new char[length];

  strcpy(iSt.text,text);
  strcat(iSt.text,par.text);

  return iSt;
}
```

String Library

```
String& String::operator + (char *str)
{
  String iSt = "";
  int length = 0;
  length = strlen(text);
  length += strlen(str);

  iSt.text = new char[length];

  strcpy(iSt.text,text);
  strcat(iSt.text,str);

  return iSt;
}
```

String Library

```
ostream& operator<< (ostream &out, String &str)
{
   out << str.text;
   return out;
}


istream& operator>> (istream &in, String &str)
{
   char temp[200];
   in >> temp;
   text = new char[strlen(temp)];
   strcpy(text,temp);

   return in;
}
```

String Library

```
char& String::[] (int Index)
{
      return text[Index];

}

//String string1 = “hello”;
// string1[0] = ‘a’;
//string1.text[0] = ‘a’;

// char c = string1[0];
```

String Library

```
int main ( )
{
        String string1 = “hello”;
        String string2 = “”;

        string1 = “hello world”;
        cout << “Enter string 2 text” << endl;
        cin >> string2;

        if ( string1 == string2 )
                cout << “Both strings are equal” << endl;

        string2[0] = ‘a’;
        string2[1] = ‘b’;
        cout << “The second string is “ << string2 << endl;

        cout  << the first character is “<< string1[0] << endl;
}
```