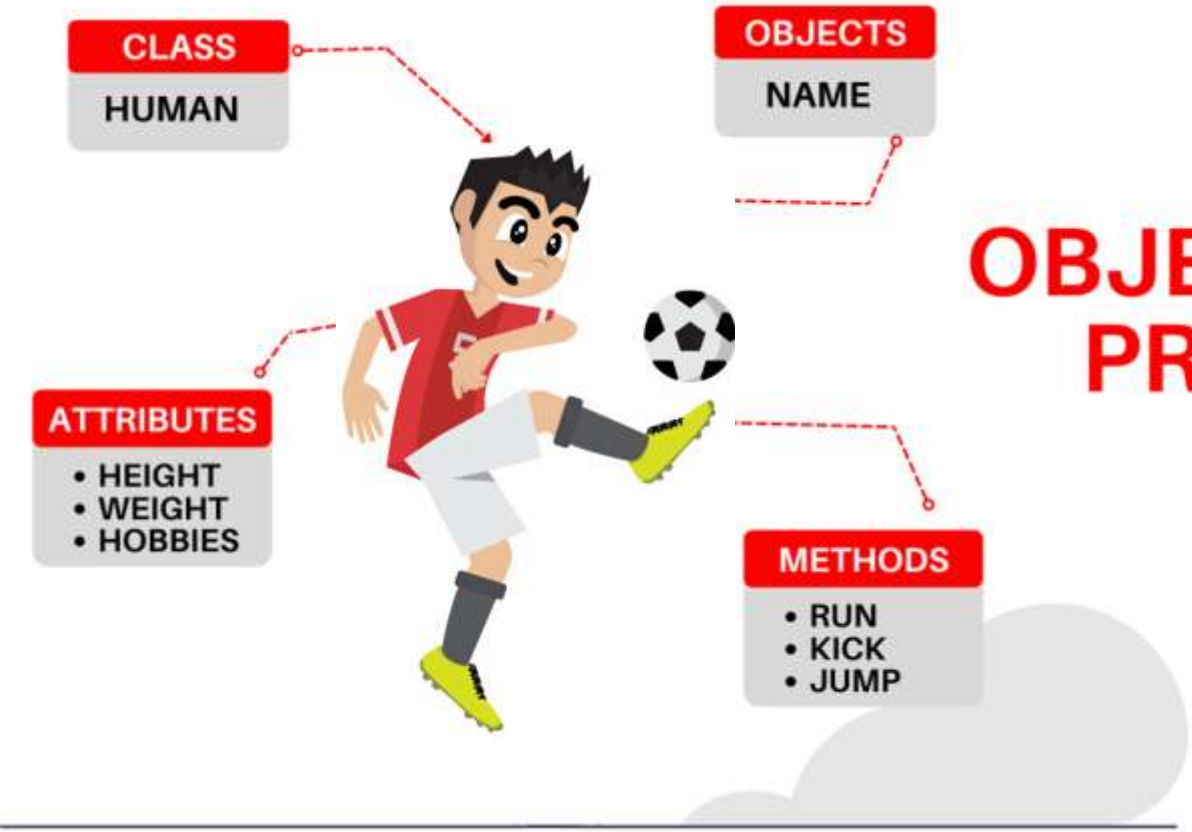
National University of Computer and Emerging Sciences



OBJECT-ORIENTED PRORAMMING

Summer 2023

Pir Sami Ullah Shah

Lecture # 7 Operator Overloading

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;
  - Parametrized (3 ints) — Date d(9,22,20);
  - copy — Date d1(d);
  - Parametrized (1 string, 2 ints) — Date d(“Sept”,22,2020);

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 all primitive types i.e. integers, double, float, int, long.

Operator Overloading

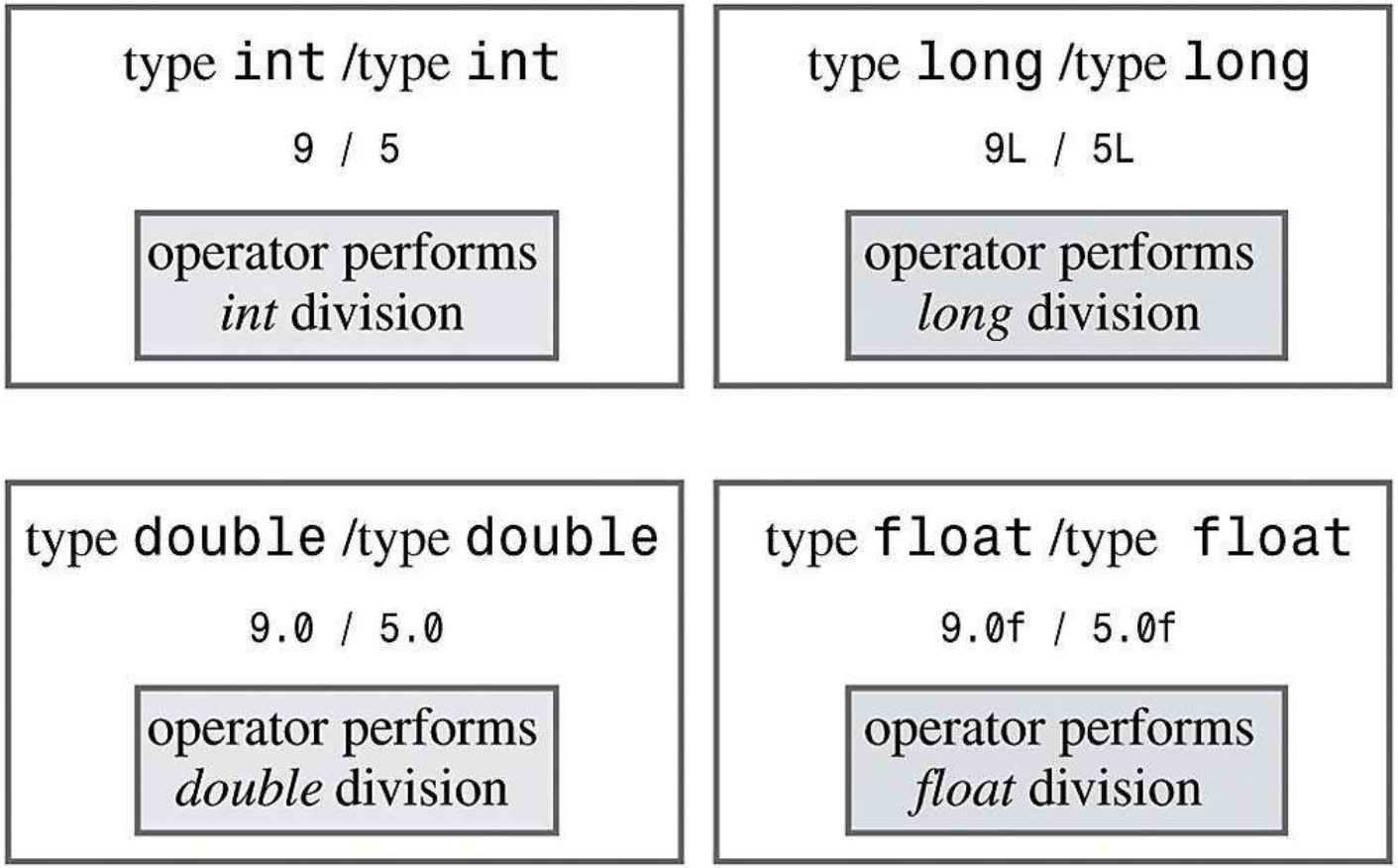
- The operator "+" also has different semantics depending on the type of its "arguments"

Example

```
int i=5, j=7;
i + j;   //add two int

double d=5.4, e=3.0;
i + d;   //add an int and a double

d + e; //add two doubles
```

Operator Overloading

- Example (already overloaded operator **/**):
- Same operator has different meaning for different operands



Operator Overloading Motivation

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

- Yes, using **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

How to Overload an Operator?

- An operator can be overloaded by declaring a special function
- Name of the function is *operator* followed by ***operator symbol*** e.g., **operator+**, **operator/**, etc.
- *operator* is a keyword here
- Can be a member function of the class (must be non-static)

Syntax to Overload an Operator

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

any type — keyword — operator symbol — function body

Example:

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

any type — keyword — operator symbol — function body

- return-type may be whatever the operator returns
- Operator symbol may be any over-loadable operator

Operator Overloading

- Operators are really functions
  - They have paremeters, they return values
  - The only difference is operator keyword is used in the function name e.g. 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:
  - How operators act on built-in data types:
    - i.e., cannot change integer addition
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Association rules (left-to-right evaluation)
  - Number of operands
    - i.e., & is unary, only acts on one operand
  - Cannot create new operators
  - Operators must be overloaded explicitly:
  - i.e., Overloading + , does not overload +=

Restriction on Operator Overloading

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

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

How to Overload an Operator?

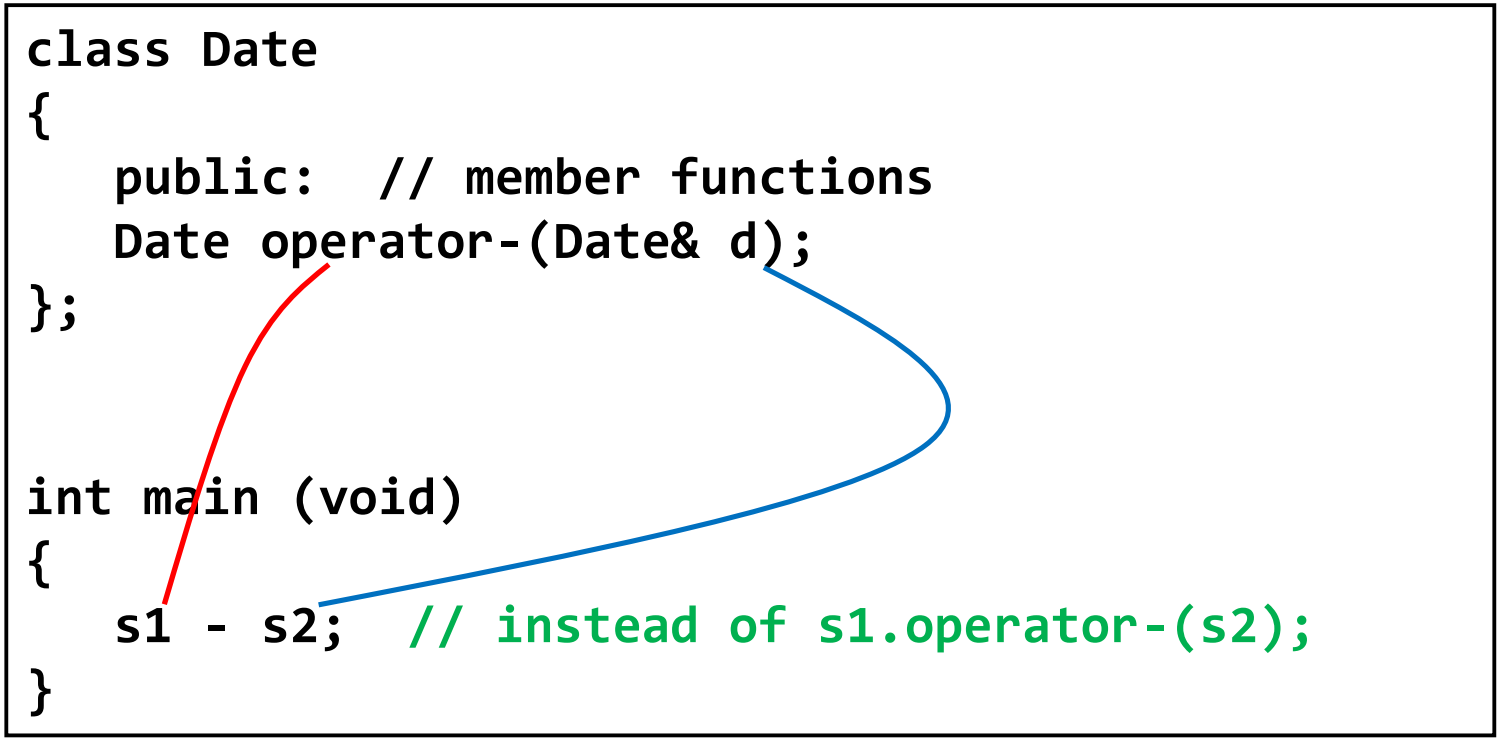
- **Member function:** If the left operand of that particular operator is an object of the same class, then the overloaded operator is said to be implemented by a member function.

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

Invoking Objects

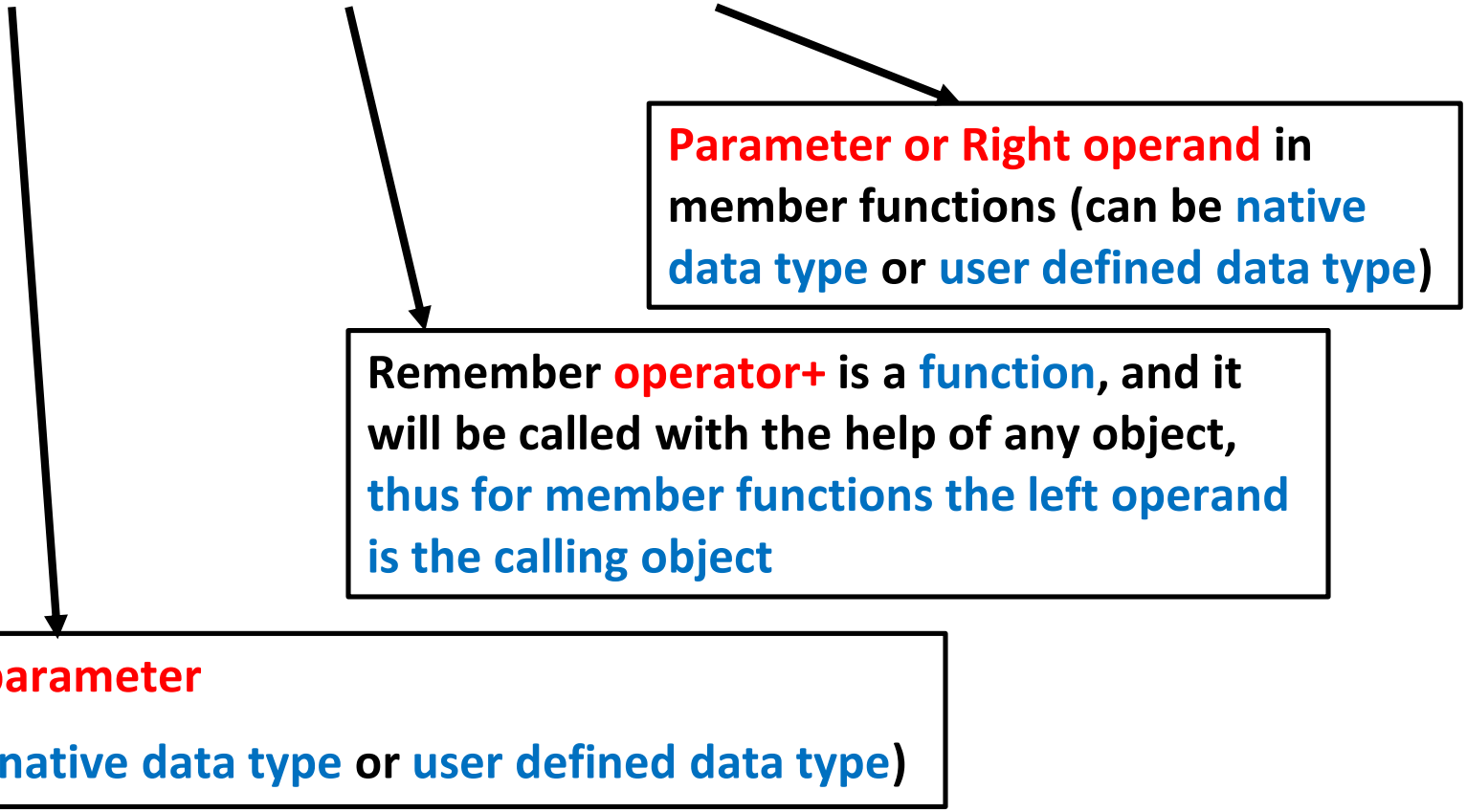
- If the operator is binary but there is only one explicit argument, the calling object is assumed to be the left operand

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


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



Operator Overloading Syntax

- `a = b + c;`
- `datatype operator+ (datatype) { … }`



Operator Overloading Syntax

datatype operator+ (datatype)
Example (1): parameter of primitive data type
return value of primitive data type

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

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

Operator Overloading Syntax

datatype operator+ (datatype)
Example (2): parameter of user defined data type
return value of primitive data type

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

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

Operator Overloading Syntax

datatype operator+ (datatype)
Example (3): parameter of primitive data type
return value of user defined data type

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


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

Operator Overloading Syntax

datatype operator+ (datatype)

Example (4): parameter of user defined data type

return value of user defined data type

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


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

Overload as Member or Non-Member Function

Rule of Thumb:

- 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 class

Operator Overloading

- Relational operators are also binary and can be overloaded in the same way

`<` , `>` , `<=` , `>=` , `==`

- 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.getSeniority());
}
```

called from the program like this:

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

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 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.***

```
double + Secretary; //ERROR
```

- The overloaded operator+ function is a member of the employee class.

- Left operand **MUST** be an object of Employee class.

The Problem Gets Wo

- A member function CANNOT ov

Primitive data type + User-define

double + Employee

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

- Whenever an operator is overloaded as a member function. The left operand must be of that SAME class.

- Here the left operand is a **double**, we cannot create an overloaded function in the double class

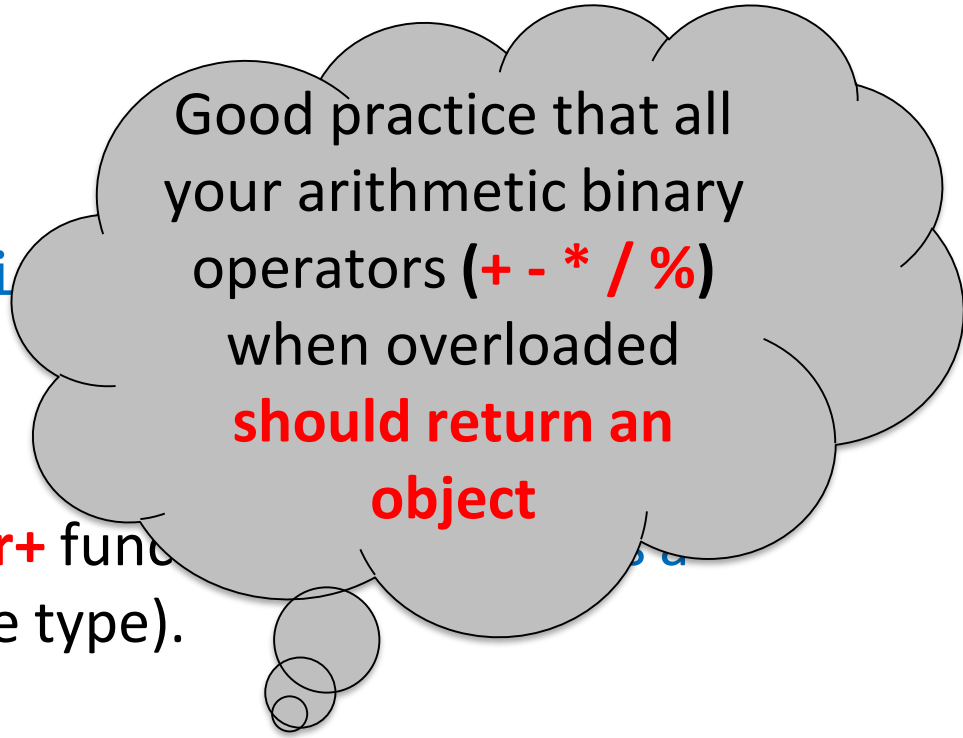
There are **two ways** to make this work for

1. **non-member function**
2. **Friend function**

To be discussed later…

Solution 1

```
sum = Clerk + Dri
```



- Make sure that your **operator+** func
  double (or any other primitive type).

- An operator to add Employees should return an Employee object.

Solution 1

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)
{
   double sal = salary + emp.salary;
   Employee total(0, sal);
   return total; //returns an object
}
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;
}
```

Solution 2

```
sum = Clerk + Driver + Secretary;
```

- Make a **non-member operator+** function that returns an Employee object.

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();
}
```

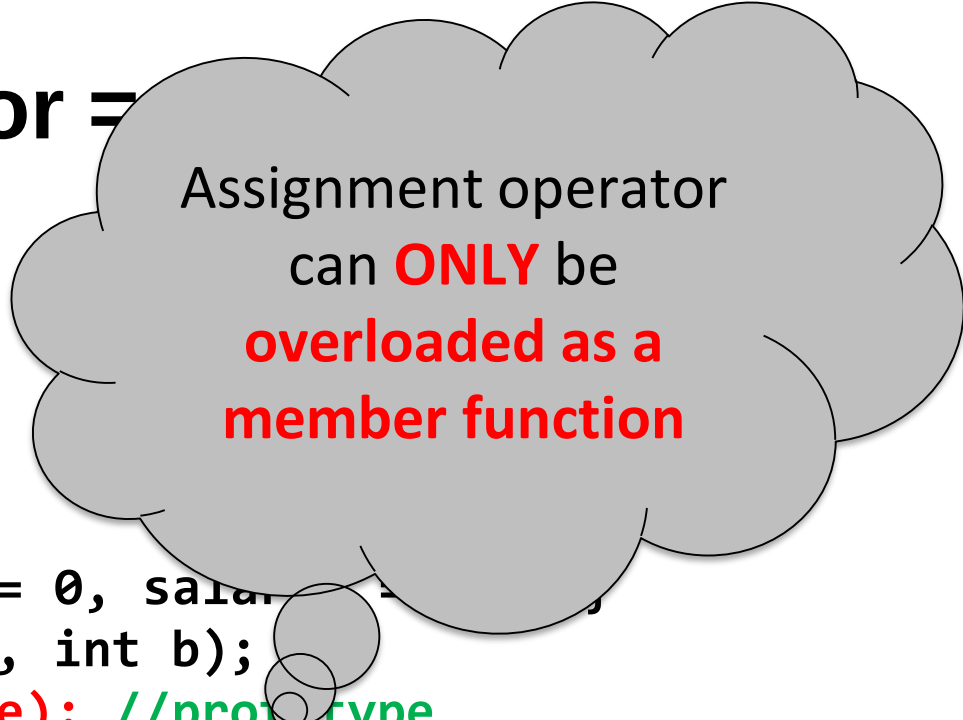
Assignment Operator =

- Operator **=** is overloaded implicitly for every class, so they can be used for each class objects.
- Recall default copy constructor
- operator = performs member-wise copy of the data members.
- However, there is a problem with implicitly overloaded operator=
- Recall **shallow copy**

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;
}
```

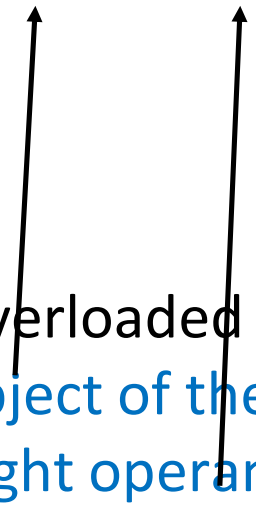
Assignment Operator =




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

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

}
```



- Overloaded as a member function so left operand **must** be an object of the same class
- Right operand a double (parameter of the overloaded = function)

Assignment Operator =

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

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


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

Assignment Operator =

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

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

}
```

```
class myClass {
    int a;
public:
    myClass(int x) {
        a = x;
    }
    myClass(myClass& m) {
        cout << "copy constructor"
        a = m.a;
    }
    int getA() {
        return a;
    }
    void setA(int aa) {
        a = aa;
    }
    void operator=(myClass& c) {
        cout << "\nOverloaded = function" << endl;

        this->a = c.a;
    }
};
```

```
int main()
{   myClass c1(-3);
    myClass c2(5);
//copy constructor will be called at
//object creation
    myClass c3 = c2;

    cout << c3.getA();

//overloaded = operator function called
    c1 = c2;
    cout << c1.getA();

    return 0;
}
```

```
copy constructor
5
Overloaded = function
5
```

```
class myClass {
    int a;
public:
    myClass(int x) {
        a = x;
    }
    myClass(myClass& m) {
        cout << "copy constructor"
        a = m.a;
    }
    int getA() {
        return a;
    }
    void setA(int aa) {
        a = aa;
    }
    myClass operator=( const myCla
        myClass m(0);
        cout << "\nOverloaded = function" << endl;
        m.a = c.a;
        return m;  }
  };
```

```
int main()
{   myClass c1(-3);
    myClass c2(5);
//copy constructor will be called at
//object creation
    myClass c3 = c2;

    cout << c3.getA();

//overloaded = operator function called
    c1 = c2;     //c1.operator=(c2);

//c1 is unchanged
    cout << c1.getA();

    return 0;
}
```

```
copy constructor
5
Overloaded = function
-3
```

Comparison Operator ==

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

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


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

Comparison Operator ==

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

}
```

Operator Overloading Syntax

- Syntax of an overloaded operator function:

  datatype operator+ (datatype)

- However, for some operators, this syntax will be slightly different:
  - **++**, **--** 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 processes the variable

- **Postfix** processes the variable, then 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 simple. 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 operator will 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);
}
```

```
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;
}
```

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 does this statement execute?

  semesterGPA[0] = 3.7;

- The **[ ]** has higher 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

}
```

```
#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;
}
```

Parenthesis operator ()

- Can only be overloaded as a member function
- Can have any return type
- Can have zero or more parameters
- Implement any logic in the function

```
class myClass {
    int a;
    char arr[15] = "something";
public:
    myClass(int x) {
        cout << "\nConstructor called" <
        a = x;
    }
    myClass(myClass& m) {
        cout << "copy constructor" << er
        a = m.a;
    }
    int getA() {
        return a;
    }
    void setA(int aa) {
        a = aa;
    }
    int operator()(char find ) {
        cout << "\nOverloaded () called" << endl;
        for (int i = 0; arr[i] != '\0'; i++) {
            if (arr[i] == find)
                return i;
        }
        return -1;
    }
};
```

```
int main()
{
    //constructor called
    myClass c1(15);

       //overloaded () called
    int ind = c1('m');

    cout<<"Index of 'm' is: "<<ind;

    return 0;
}
```

Constructor called

Overloaded () called

```
class myClass {
    int a;
    char arr[15] = "something";
public:
    myClass(int x) {
        cout << "\nConstructor called" << endl
        a = x;
    }
    myClass(myClass& m) {
        cout << "copy constructor" << endl;
        a = m.a;
    }
    int getA() {
        return a;
    }
    void setA(int aa) {
        a = aa;
    }
    int operator()(myClass &find) {
        cout << "\nOverloaded () called" << endl;
        return -1;
    }
};
```

```
int main()
{
    myClass c1(15),c3(0);
    //constructor called

    c1(c3);
    //overloaded () called

    myClass c2(c1);
    //copy constructor called

    return 0;
}
```

Constructor called

Constructor called

Overloaded () called

copy constructor

new vs new[ ] operator overloaded

```
class myClass {
    int a;
    char arr[15] = "something";
public:
    myClass(int x=0) {
        cout << "\nConstructor called" << endl;
    }
    myClass(const myClass& m) {
        cout << "copy constructor" << endl;
        a = m.a;
    }
    int getA() {
        return a;
    }
    void setA(int aa) {
        a = aa;
    }
    void* operator new(size_t s) {
        cout << "\nOverloaded new called" << endl;
        void* p =:: operator new(s); //:: calls standard new operator
        return p;
    }
    void* operator new[](size_t s) {
        cout << "\nOverloaded new [] called" << endl;
        void* p = ::operator new[s]; //:: calls standard new [] operator
        return p;
    }
};
```

```
int main()
{
    myClass *p=new myClass[5];
    //calls overloaded new []
    p = new myClass;
    //calls overloaded new
}
```

Calling an overloaded operator from native data types

```
int var;
Point object;
var = var + object;
```

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

- But in operator overloading we ***can't change the functionality of integer (or any primitive) data type***

Calling an overloaded operator from native data types

- ***Friend functions*** can help solve this problem.

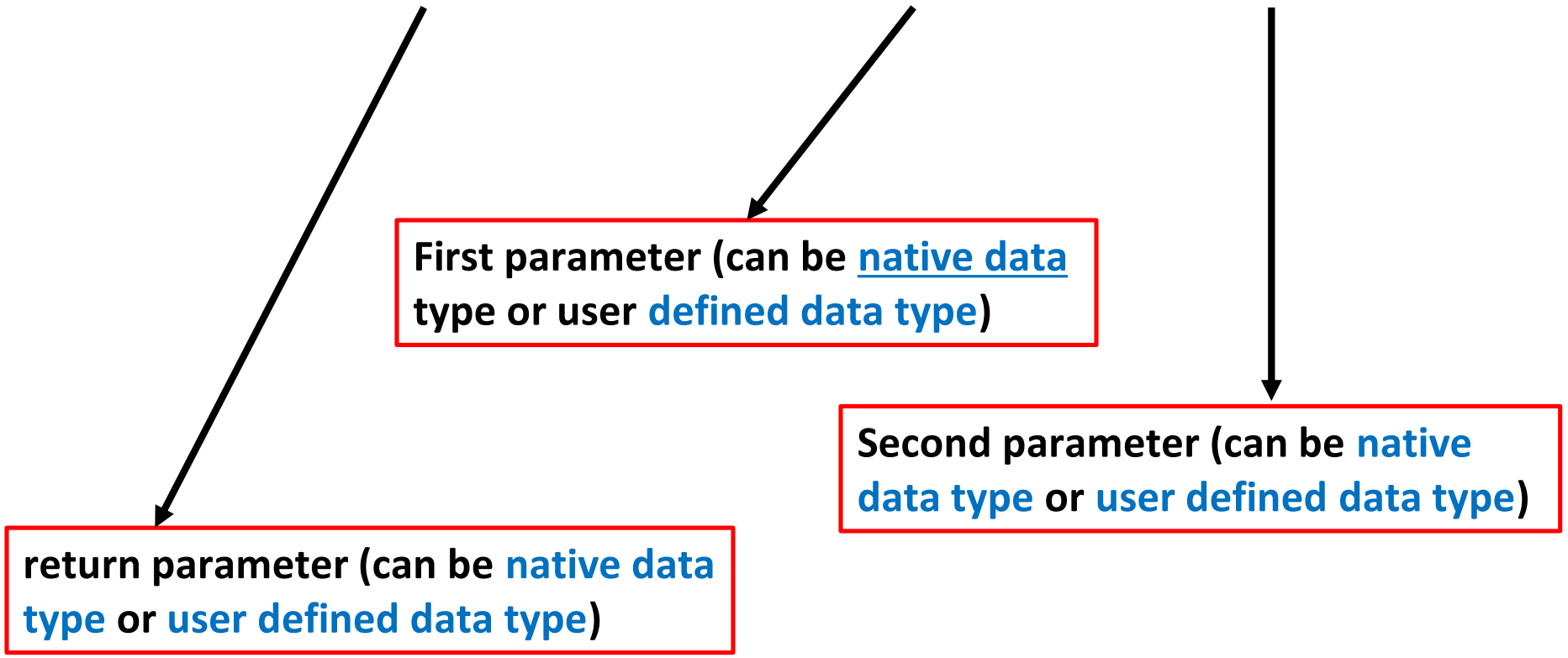
- A Friend function does not need an object of a class for its calling.

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

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, Point &);
};
```

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 <<

- stream insertion operator **<<** is used for output
- stream extraction operator **>>** is used for input

cout is an object of ostream class

cin is an object of istream class

These operators must be overloaded as a global function. And if we want to allow them to access private data members of the class, we must make them friend.

Overloading iostream operators >> and <<

- **Why these operators must be overloaded as *global*?**
- Left operand with << and >> is always going to be cin and cout
- if we want to make them a member function, then they must be made members of ostream and istream classes, not a good option.
- Therefore, these operators are overloaded as global functions with two parameters, istream/ostream and object of user-defined class.

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 **<<**) inside a class with the help of ***friend function***, and **then** ***we can access private data members.***

- We can also overload these operators without making friend functions

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;
}
```

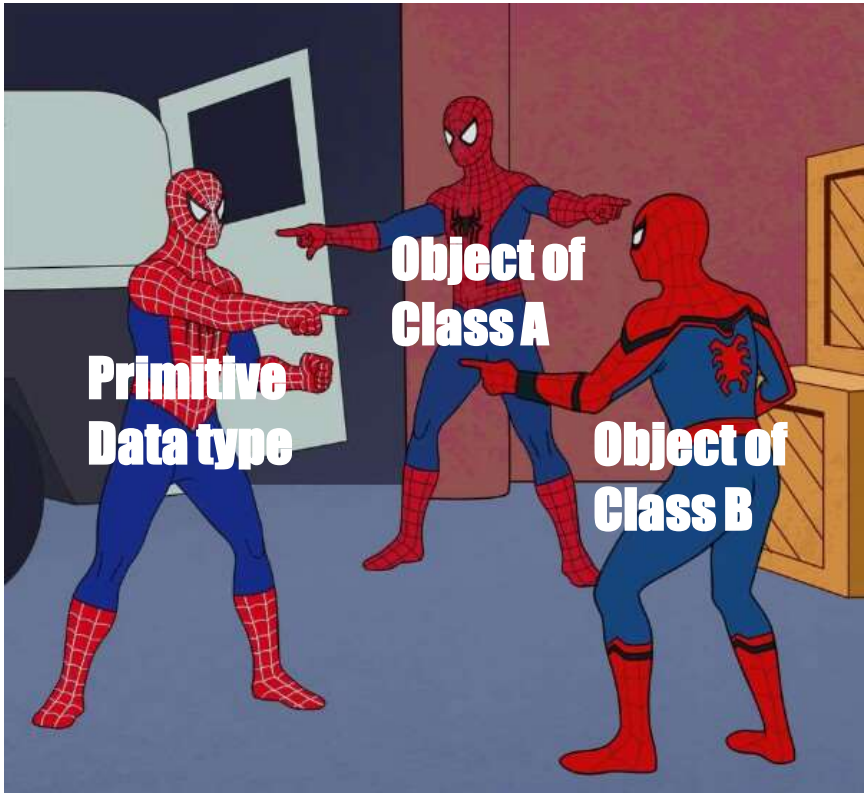
```
istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}
```

Example

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

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

}
```

Data Conversion

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



Implicit Conversion b/w Basic Types

- When we use two different Types:

```
intvar = floatvar; //assign float to integer
```

the compiler calls a special function that converts this value from floating point format to integer

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

Explicit Conversion b/w Basic Types

- if we want to force compiler to convert data from one native type to other, we can use explicit type 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.

Conversion From User Defined to Basic

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

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

```
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:

```
Distance dist2;
float floatmtrs = float(dist2); //explicit conversion
float floatmtrs = dist2;                    //implicit
conversion
```

- both statements have exactly the same effect.

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.

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);
        }

        bool operator==(String &str);
        bool operator==(char *str);
        String& operator+(String &str);
        String& operator+(char *str);
        void operator= (char *str);
        char& operator[] (int Index);
        friend ostream& operator<<(ostream &,String &str);
        friend istream& operator>>(istream &,String &str);
};
```

String Library

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

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

}
```

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

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


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

}

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

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

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

```
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;
}
```