Lecture Three

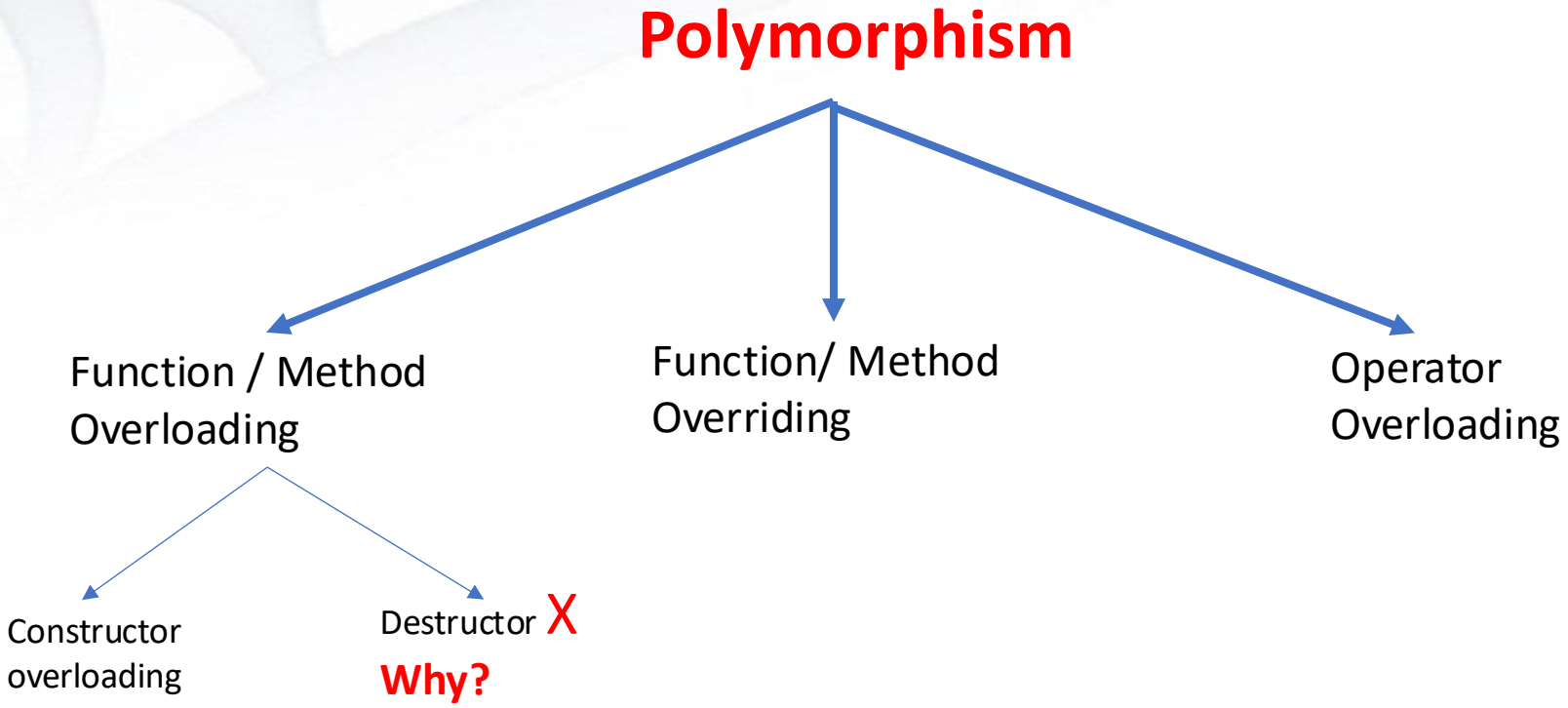
# Polymorphism

© **Dr. Mohammad Mahfuzul Islam, PEng**
Professor, Dept. of CSE, BUET

# Polymorphism

**Poly** = many

**Morphism** = shape, form or structure



# Constructor Overloading in C++

- It is common to overload a class's constructor function.
- It is not possible to overload a destructor function.
- Three main reasons to overload constructor function:
  - to gain flexibility (discussed in Lecture 1)
  - to support arrays and
  - to create copy constructors (discussed in Lecture 2)
- If a program attempts to create an object for which no matching constructor is found, a compile-time error occurs.

**Example for supporting array**

```
#include <iostream>
using namespace std;

class myclass {
    int x;
public:
    myclass() { x = 0;}
    myclass(int n) { x = n;}
    int getx() {return x;}
};

int main() {
    myclass o1[5];
    myclass o2[5] = {1, 2, 3, 4, 5};
    for(int i = 0; i < 5; i++){
       cout << o1[i].getx() << " ";
       cout << o2[i].getx() << endl;
    }
    return 0;
}
```

**OUTPUT:**

```
0 1
0 2
0 3
0 4
0 5
```

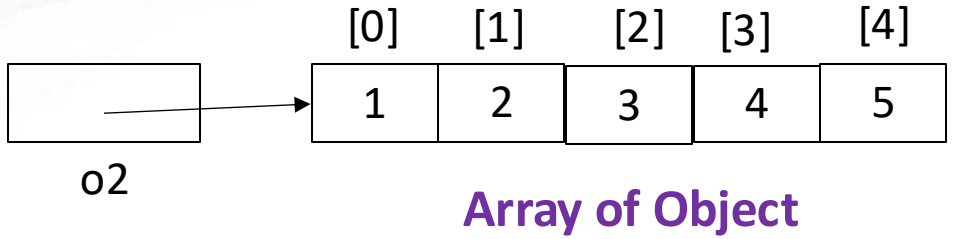
# Constructor Overloading in Java

```
class MyClass{
    private int x;

    MyClass(){ x = 0;}
    MyClass(int n){ x = n;}
    public int getX(){ return x;}
}

public class Main {
    public static void main(String[] args) {
        MyClass[] ob = new MyClass[5];

        for(int i = 0; i <ob.length; ++i){
            if (i % 2 != 0)
                ob[i] = new MyClass();
            else ob[i] = new MyClass(i+1);
        }

        for(int i = 0; i < ob.length; ++i){
            System.out.print(ob[i].getX()+" ");
        }
    }
}
```

**OUTPUT:**
1 0 3 0 5

**C++**

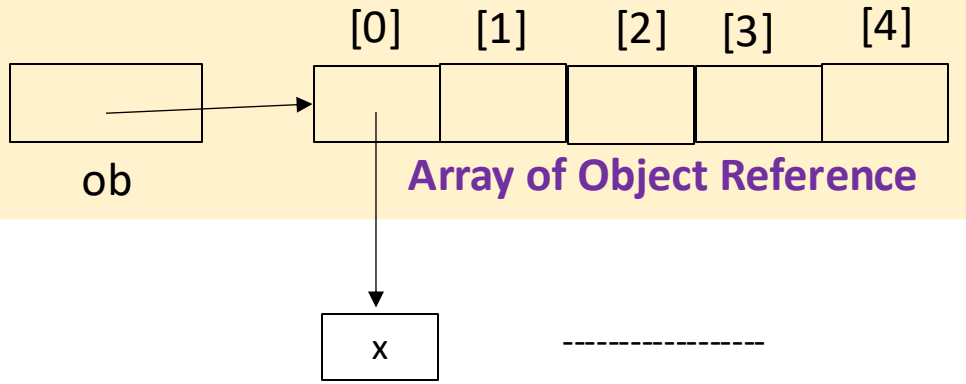
```
myclass o2[5] = {1, 2, 3, 4, 5};
```




Array of Object

Array of Object
Why?

**Java**



# Copy Constructor

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype{
    char *p;
public:
    strtype(const char *s){
        int l;
        l = strlen(s) + 1;
        p = new char[l];
        if (!p){
            cout << "Allocation error\n";
            exit(1);
        }
        strcpy( p, s );
    }
    ~strtype() { delete [] p;}
    char *get() { return p; }
};
```

```
void show(strtype x){
    char *s;

    s = x.get();
    cout << s << endl;
}
int main(){
    strtype a("Hello"), b("There");

    show(a);
    show(b);
    return 0;
}
```

What is the problem of the program?

**OUTPUT:**

```
Hello
There
CopyConstructor2(1240,0x10d697600) malloc: *** error for object
0x7f8d22f05a00: pointer being freed was not allocated
CopyConstructor2(1240,0x10d697600) malloc: *** set a breakpoint in
malloc_error_break to debug
zsh: abort      ./"CopyConstructor2"
```

# Copy Constructor

➢The copy constructor is invoked when a function generates the temporary object.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype{
    char *p;
public:
    strtype(const char *s){
       int l;
       l = strlen(s) + 1;
       p = new char[l];
       strcpy( p, s );
       cout << "Constructing normally\n";
   }
   strtype(const strtype &s);
   ~strtype() { delete [] p;}
   char *get() { return p; }
};
```

No Copy Constructor in Java

Why?

```
strtype::strtype(const strtype &s){
    int l;
    l = strlen(s.p) + 1;
    p = new char[l];
    strcpy( p, s.p );
    cout << "Constructing copy\n";
}

void show(strtype x){
    char *s;
    s = x.get();
    cout << s << endl;
}

int main(){
    strtype a("Hello"), b("There");
    show(a);
    show(b);
    return 0;
}
```

**OUTPUT:**

```
Constructing normally
Constructing normally
Constructing copy
Hello
Constructing copy
There
```

# Default Argument

- The defaults can be specified either in function prototype or in its definition.
- The defaults cannot be specified in both the prototype and the definition.
- All default parameters must be to the right of any parameters that do not have defaults.

```
#include <iostream>
using namespace std;

void f(int a = 0, int b = 0){
    cout << a << " " << b << endl;
}

int main(){
    f();
    f(10);
    f(10, 99);
}
```

**OUTPUT:**

```
0 0
10 0
10 99
```

```
#include <iostream>
using namespace std;

void f(int a = 0, int b);

int main(){
    f(10);
    f(10, 99);
}
void f(int a, int b){
    cout << a << " " << b << endl;
}
```

What's Wrong?

# Default Argument

➢ **Default argument can be used instead of function overload**

```
#include <iostream>
using namespace std;

double rect_area( double length, double width = 0){
      if (!width) width = length;
      return length*width;
}

int main(){
     cout << rect_area(10.0, 5.8) << endl;
     cout << rect_area(10.0) << endl;
     return 0;
}
```

**OUTPUT:**
58
100

➢ **Copy constructors may take additional arguments, as long as the additional arguments have default values.**

```
myclass( const myclass &obj, int x = 0){
            //body of constructor
}
```

```
#include <iostream>
using namespace std;

class myclass {
    int x;
public:
    myclass(int n = 0) { x = n; }
    int getx() { return x; }
};

int main(){
    myclass o1(10);
    myclass o2;
    cout << o1.getx() << endl;
    cout << o2.getx() << endl;
    return 0;
}
```

**OUTPUT:**
10
0

**Java doesn't support default argument.**

# Ambiguity with Function Overloading

➢ **Automatic type conversion** rule cause an ambiguous situation.

```
#include <iostream>
using namespace std;

float f(float i){
    return i / 2.0;
}

double f(double i){
    return i / 3.0;
}
```

```
int main(){
    float x = 10.09;
    double y = 10.09;

    cout << f(x) << endl;
    cout << f(y) << endl;
    cout << f(10) << endl;

    return 0;
}
```

**OUTPUT:**

error: call to 'f' is ambiguous

cout << f(10) << endl;

➢ **Wrong type of arguments** causes an ambiguous situation.

```
#include <iostream>
using namespace std;

void f(unsigned char c){
    cout << c;
}

void f(char c){
    cout << c;
}
```

```
int main(){
    f('c');
    f(86);
    return 0;
}
```

**OUTPUT:**

error: call to 'f' is ambiguous

f(86);

Theoretically, Java should have both Type ambiguities. But intelligent platform like Intellij resolves such ambiguities based on-

- Close proximity
- Specialization

# Ambiguity with Function Overloading

- **Call by value** and **call by reference** cause an ambiguous situation.

```
#include <iostream>
using namespace std;

int f(int a, int b){
   return a+b;
}

int f(int a, int &b){
   return a-b;
}
```

```
int main(){
   int x = 1, y = 2;

   cout << f(x, y); // which f() is called?
   return 0;
}
```

- **Default argument** causes an ambiguous situation.

```
#include <iostream>
using namespace std;

int f(int a){
   return a*a;
}

int f(int a, int b = 0){
   return a*b;
}
```

```
int main(){
   cout << f(10, 2);
   cout << f(10);  // which f() is called?
   return 0;
}
```

Java doesn't have these two ambiguities.

# Finding address of an Overloaded Function

- A function address is obtained by putting its name on the right side of an assignment statement without any parenthesis or arguments.

To assign p the address of zap(),

```
p = zap;
```

- What about overloaded function????

```
#include <iostream>
using namespace std;

void space(int count){
   for( ; count; count--) cout << '_';
}

void space(int count, char ch){
   for( ; count; count--) cout << ch;
}
```

**OUTPUT:**

```
__________
xxxxxxxxxx
```

```
int main(){
   void (*fp1)(int);
   void (*fp2)(int, char);

   fp1 = space;
   fp2 = space;

   fp1(10);
   cout << endl;
   fp2(10, 'x');
   cout << endl;

   return 0;
}
```

**Java**

- Java doesn't provide address level access of any code due to security reason.
- Hash Code of an object can be found using object.hashCode() method.

**A hash value is a numeric value of a fixed length that uniquely identifies data. Mainly used for digital signature.**

# Method Overriding

```
#include <iostream>
Using namespace std;

class Figure {
    double dim1, dim2;
    Figure(double a, double b){ dim1 = a; dim2 = b;}
    virtual double area() = 0;  // Pure virtual function
    virtual void show(){ cout << "Abstract";}
}

class Rectangle:  public Figure {
    Rectangle(double a, double b) { super(a, b);}
    double area(){ return dim1*dim2;}
    void show(){cout << "Rectangle Area: " << area();}
}

class Triangle: public Figure {
    Triangle(double a, double b) {super(a, b);}
    double area(){ return 0.5*dim1*dim2;}
    void show(){cout << "\nTriangle Area: " << area());
    }
}
```

C++ Code

```
int main() {
    Rectangle r(4,5);
    Triangle t(4, 3);
    Figure *figref;

    figref = r;
    figref.show();
    figref = t;
    figref.show();

    return 0;
}
```

**OUTPUT:**

```
Rectangle Area: 20.0
Triangle Area: 6.0
```

# Method Overriding

```
abstract class Figure {
    double dim1, dim2;
    Figure(double a, double b){ dim1 = a; dim2 = b;}
    abstract double area();
    void show(){
        System.out.println("Abstract");
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) { super(a, b);}
    double area(){ return dim1*dim2;}
    void show(){
        System.out.println("Rectangle Area: "+area());
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {super(a, b);}
    double area(){ return 0.5*dim1*dim2;}
    void show(){
        System.out.println("Triangle Area: "+area());
    }
}
```

Java Code

```
public class Main {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(4,5);
        Triangle t = new Triangle(4, 3);
        Figure figref;

        figref = r;
        figref.show();
        figref = t;
        figref.show();
    }
}
```

**OUTPUT:**

```
Rectangle Area: 20.0
Triangle Area: 6.0
```

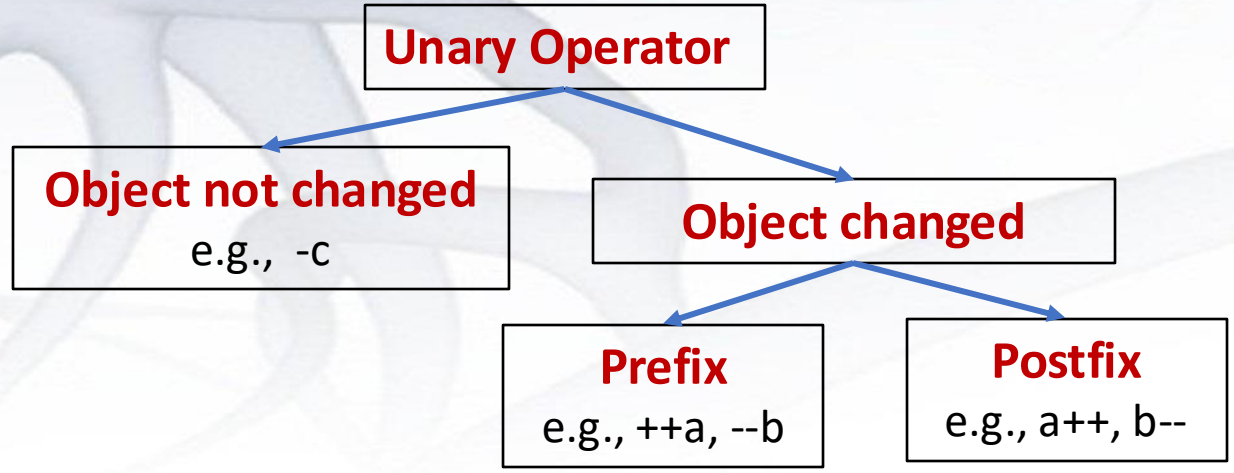
# Operator Overloading in C++

- When an operator is overloaded, that operator loses none of its original meaning; instead, it gains additional meaning relative to the class.
- Operator can be overloaded by creating either a member operator function or a friend operator function.
- The general form of member operator function:

```
return-type class-name::operator#(arg-list){
        // operation to be performed
}
```

- Two important restrictions of operator overloading:
  (1) the precedence of the operator cannot be changed;
  (2) the number of operands that an operator takes cannot be altered.
- Most C++ operators can be overloaded. Only the following operators cannot be overloaded-
  (1) Preprocessor operator (2) . (3) :: (4) .* (5) ?
- Except for the =, operator functions are inherited by any derived class.

# Unary Operator Overloading in C++




## When Object Not Changed:

**No Change of Object (-ob)**

```
Coord Coord:: operator +(){
        Coord temp;
        temp.x = -x;
        temp.y = -y;
        return temp;
}
```

Member of Object

## Prefix and Postfix Unary operations:

| Object before operator (ob++) | No object before operator (++ob) |
|---|---|
| Coord Coord:: operator ++(int notused){<br>x++;<br>y++;<br>return *this;<br>}<br>Member of Object | Coord Coord:: operator ++(){<br>++x;<br>++y;<br>return *this;<br>}<br>Member of Object |

Only (int) can be used instead of (int notused). "notused" passed 0.

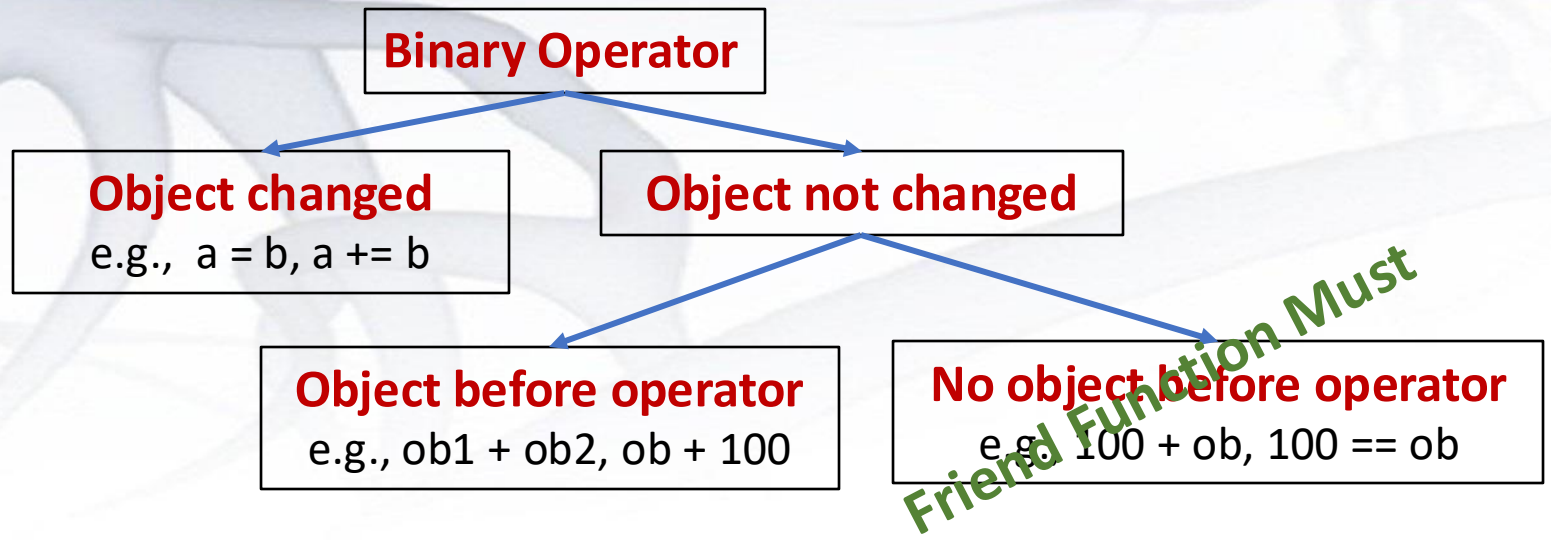
## When Object Changed:

**Prefix Unary Operation:**
**++a; --b; -c**

**Postfix Unary Operation:**
**a++; b--**

**To make differences, Postfix Unary operation is assumed as**
**a ++ (notused)**

# Binary Operator Overloading in C++




**Binary operation when object changed:**

**Object before operator (ob1 += ob2)**

```
Coord Coord:: operator += (Coord ob){
        x += ob.x;
        y += ob.y;
        return *this;
}
```

Member of Object

**Binary operation without changing object:**

| Object before operator (ob1*ob2) | No object before operator (100+ob) |
|---|---|
| Coord Coord:: operator * (Coord ob){<br>Coord temp;<br>temp.x = x * ob.x;<br>temp.y = y * ob.y;<br>return temp;<br>}<br>**Member of Object** | Coord operator + (int x, Coord ob){<br>Coord temp;<br>temp.x = x + ob.x;<br>temp.y = x + ob.y;<br>return temp;<br>}<br>**Friend Function Must** |

# Operator Overloading in C++

```
#include <iostream>
using namespace std;

class Coord{
      int x, y;
public:
      Coord(int a=0, int b=0){ x = a; y = b;}
      void getxy(int &i, int &j){ i = x; j = y;}

      Coord operator + (Coord ob);
      Coord operator + (int i);
      Coord operator++(int); // Postfix increment
      Coord operator++(); // Prefix increment
      Coord operator * (Coord ob);
      Coord operator = (Coord ob);

      friend bool operator ==(int x, Coord ob);
      friend Coord operator + (int x, Coord ob);
};
```

```
Coord Coord:: operator + (Coord ob){
       Coord temp;
       temp.x = x + ob.x;
       temp.y = y + ob.y;
       return temp;
}

Coord Coord:: operator + (int i){
      Coord temp;
      temp.x = x + i;
      temp.y = y + i;
      return temp;
}

Coord Coord:: operator ++(int notused){
      x++;
      y++;
      return *this;
}

Coord Coord:: operator ++(){
      ++x;
      ++y;
      return *this;
}
```

- For Unary operator overloading, the operand is passed implicitly to the function (identified by *this object).
- When a binary operator is overloaded, the left operand is passed implicitly to the function (identified by *this object) and the right operand is passed as an argument.

# Operator Overloading in C++

```
Coord Coord:: operator * (Coord ob){
        Coord temp;
        temp.x = x * ob.x;
        temp.y = y * ob.y;
        return temp;
}

bool operator ==(int x, Coord ob){
         return (x == ob.x && x == ob.y);
}

Coord operator + (int x, Coord ob) {
        Coord temp;
        temp.x = x + ob.x;
        temp.y = x + ob.y;
        return temp;
}

Coord Coord:: operator = (Coord ob){
       x = ob.x + 100; // Not true, but just for testing
       y = ob.y + 100; // Not true, but just for testing
       return *this;
}
```

```
int main(){
        Coord a(10, 20), b(4, 4), c;
        int x, y;

        c = a++;
        c.getxy(x, y);
        cout << x << " " << y << endl;
        a.getxy(x, y);
        cout << x << " " << y << endl;

        c = ++a;
        c.getxy(x, y);
        cout << x << " " << y << endl;
        a.getxy(x, y);
        cout << x << " " << y << endl;

        (a*b).getxy(x, y);
        cout << x << " " << y << endl;

        if (4 == b) cout << "Equal" << endl;
        else cout << "Not equal" << endl;
        (20+a).getxy(x, y);
        cout << x << " " << y << endl;

        (a + b + c + 100).getxy(x, y);
        cout << x << " " << y << endl;

        c = a++;
        c.getxy(x, y);
        cout << x << " " << y << endl;

        return 0;
}
```

**OUTPUT:**

```
110 120
11 21
112 122
12 22
48 88
Equal
32 42
228 248
112 122
```

# Passing Reference Parameter for Operator Overloading in C++

- Passing a reference parameter has two advantages-

  (1) passing the address of an object is always quick and efficient.

  (2) to avoid the trouble caused when a copy of an operand is destroyed.

```
#include <iostream>
using namespace std;

class coord {
        int x, y;
public:
        coord(int i = 0, int j = 0) { x = i; y = j;}
        void getxy(int &i, int &j) { i = x; j = y;}
        coord operator + (coord &ob){
                coord temp;
                temp.x = x + ob.x;
                temp.y = y + ob.y;
                return temp;
        }
        coord operator = (const coord &ob){
                x = ob.x;
                y = ob.y;
                return *this;
        }
};
```

```
int main(){
        coord a(10, 20), b(30, 40), c;
        int x, y;

        (a+b).getxy(x, y);
        cout << x << " " << y << endl;

        c = a = b;
        c.getxy(x, y);
        cout << x << " " << y << endl;
        a.getxy(x, y);
        cout << x << " " << y << endl;

        return 0;
}
```

**OUTPUT:**

```
40  60
30  40
30  40
```

- ✓ Changes of object impacted the calling function.
- ✓ Better to use const if object changed is not required.

**Note:**

- (a + b).getxy(x,y);
  - ✓ a & b are not changed.
  - ✓ A temporary object is created to return the object, and is destroyed after the execution of (a + b).getxy(x,y);

# Returning Reference for Operator Overloading in C++

**(1) Assignment Operator:**

- ✓ **Returning Reference** is built-in for **assignment operators** (=, +=, -=, *=, /=).
- ✓ Hence, to maintain consistency, returning reference should be used for assignment operator for **better understanding**.

**(2) Increment (++) / Decrement (--) Operator:**

- ✓ **Prefix version** (++ob, --ob) should return a **reference** to the modified object.
- ✓ **Postfix version** (ob++, ob++) should return a **copy of the object** before modification.

**(3) Input / Output Stream Operator ( >> or <<):**

- ✓ Return **a reference** to the **ostream or istream** object.

# Returning Reference for Operator Overloading in C++

```
#include <iostream>
using namespace std;

class Coord{
    int x, y;
public:
    Coord(int a=0, int b=0){ x = a; y = b;}

    Coord &operator += (Coord &ob);
    friend ostream &operator<<(ostream &out, const Coord &ob);
    friend istream &operator>>(istream &in, Coord &ob);

    Coord &operator ++(){
        ++x;
        ++y;
        return *this;
    }

    Coord operator ++(int){
        x++;
        y++;
        return *this;
    }
};
```

```
Coord &Coord:: operator += (Coord &ob){
        x += ob.x;
        y += ob.y;
        return *this;
}

ostream &operator<<(ostream &out, const Coord &ob) {
       out << "(" << ob.x << ", " << ob.y << ")";
       return out;
}

istream &operator>>(istream &in, Coord &ob) {
      cout << "Enter coordinates (x y): ";
      in >> ob.x >> ob.y;
      return in;
}
```

```
int main(){
       Coord a(10, 20), b;
        int x, y;

        cin >> b;
        cout << b << endl;
        a += b;
        cout << a << endl;

        return 0;
}
```

**OUTPUT:**

```
Enter coordinates (x y): 3 4
(3, 4)
(13, 24)
```

# Problem with Returning Reference for Operator Overloading in C++

- **Arithmetic Operator** (+, -, *, /, %) typically **creates new object** as the result of the operation and should return the new **object by value**.
- Returning **a Reference** for arithmetic operator would be **problematic** because the refenced object would likely to be a **temporary** that goes **out of scope**.

```
#include <iostream>
using namespace std;

class Coord{
    int x, y;
public:
    Coord(int a=0, int b=0){ x = a; y = b;}
    void getxy(int &i, int &j){ i = x; j = y;}

    Coord &operator + (Coord &ob){
        Coord temp;
        temp.x = x + ob.x;
        temp.y = y + ob.y;
        return temp;
    }
};
```

**OUTPUT:**

```
-1121524120 32759 //Why
40 60
10 20
30 40
```

```
int main(){
    Coord a(10, 20), b(30, 40), c;
    int x, y;

    (a+b).getxy(x, y);
    cout << x << " " << y << endl;

    c = a + b;
    c.getxy(x, y);
    cout << x << " " << y << endl;
    a.getxy(x, y);
    cout << x << " " << y << endl;
    b.getxy(x, y);
    cout << x << " " << y << endl;

    return 0;
}
```

# All Operator Overloading in C++ using Friend Function

**Consider the overloaded operator function,**

```
ob1 = ob2 + 100;     // can be implemented as a member or friend
ob1 = 100 + ob2;     //Only be implemented using a friend method.
```

- **Using friend operator function, flexibility can be added.**
- **A friend function does not have a "*this*" pointer.**
- **All the operands are passed explicitly to the friend method.**
- **Any modifications inside the friend method will not affect the object that is passed during the call. To ensure changes, reference parameter is used, if necessary.**

# Friend Function for Operator Overloading in C++

```
#include <iostream>
using namespace std;

class Coord{
      int x, y;
public:
     Coord(int x = 0, int y = 0) : x(x), y(y) {}
     void show() const { cout << "(" << x << ", " << y << ")" << endl; }

     friend Coord operator+(const Coord& c1, const Coord& c2);
     friend Coord operator+(const Coord& c, int n);
     friend Coord operator+(int n, const Coord& c);
     friend Coord operator++(Coord& c);
     friend Coord operator++(Coord& c, int);
};

Coord operator+(const Coord& c1, const Coord& c2) {
     return Coord(c1.x + c2.x, c1.y + c2.y);
}

Coord operator+(const Coord& c, int n) {
    return Coord(c.x + n, c.y + n);
}
```

```
Coord operator+(int n, const Coord& c) {
      return Coord(n + c.x, n + c.y);
}

Coord operator++(Coord& c) {
      ++c.x;
      ++c.y;
      return c;
}

Coord operator++(Coord& c, int) {
      ++c.x;
      ++c.y;
      return c;
}
```

**OUTPUT:**

```
(4, 6)
(6, 7)
(6, 7)
(2, 3)
(3, 4)
```

```
int main(){
      Coord c1(1, 2), c2(3, 4);
      Coord c3 = c1 + c2;
      c3.show();

      Coord c4 = c1 + 5;
      c4.show();

      Coord c5 = 5 + c1;
      c5.show();

      ++c1;
      c1.show();

      c1++;
      c1.show();

      return 0;
}
```

# Assignment Operator

➢By default, when an assignment operator applied to an object, a bitwise copy is made. So, there is no need to write own assignment operator.

➢In case of dynamic memory allocation, bitwise copy is not desirable and still need to write assignment operator.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype{
      char *p;
      int len;
public:
      strtype(char *s);
      ~strtype() { delete [] p;}
      char *get() { return p; }
      strtype &operator = (strtype &ob);
};
```

```
strtype:: strtype(char *s){
      int l;
      l = strlen(s) + 1;
      p = new char[l];
      strcpy( p, s );
      len = l;
}

strtype &strtype:: operator = (strtype &ob){
      p = new char[ob.len];
      len = ob.len;
      strcpy(p, ob.p);
      return *this;
}
```

```
int main(){
      strtype a("Hello"), b("There");

      cout<< a.get()<< " " << b.get()<< endl;
      a = b;
      cout<< a.get()<< " " << b.get()<< endl;

      return 0;
}
```

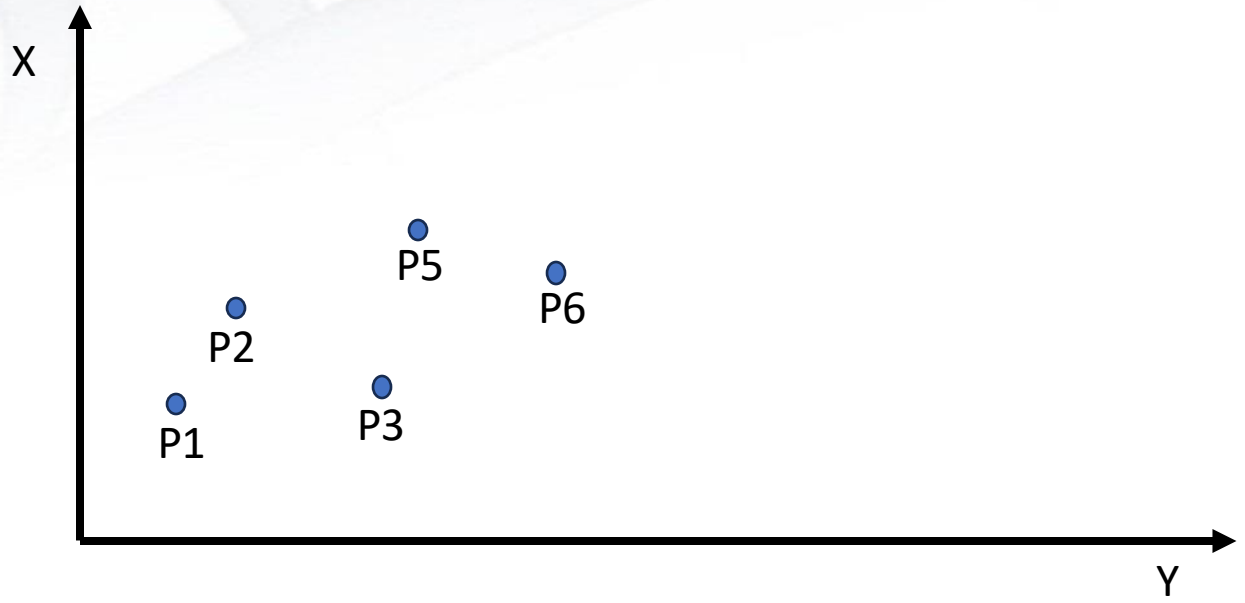
**OUTPUT:**

```
Hello There
There There
```

# Overloading Array Subscript Operator []

- **The general format of array subscript operator is as follows:**

```
int &operator [] (int i);
```



# Overloading Array Subscript Operator []

```
#include <iostream>
using namespace std;
const int arraySize = 2;

class Point {
    int *arr;
public:
    Point(int x = 0, int y = 0){
        arr = new int[arraySize];
        arr[0] = x;
        arr[1] = y;
    }

    int& operator[](int pos){
        if (pos < arraySize) return arr[pos];
        else{
            cout << "Out of bound" << endl;
            exit(0);
        }
    }

    void print(){
        cout << "(" << arr[0] << "," << arr[1] << ")" << endl;
    }
};
```

```
int main(){
    Point p1(3,4);

    p1.print();
    p1[0] = 6;
    p1[1] = 8;
    p1.print();

    return 0;
}
```

**OUTPUT:**

```
(3,4)
(6,8)
```

# Type Conversion in C++

- **Type conversion** from the **type of argument** to the **type of class** is of two types:
  - ✓ **implicit** and
  - ✓ **explicit**.
- **Type Conversion:**

```
myclass ob(4); //supported by both type of conversion
myclass ob = 4; //not supported by explicit conversion
```

```
#include <iostream>
#include <cstdlib>
using namespace std;

class Myclass{
   int a;
public:
   explicit Myclass(int x){ a=x; }
   Myclass(char *str){ a=atoi(str); }
   int getA(){ return a; }
};
```

```
int main(){
   Myclass ob1(10);
   // Myclass ob2 = 20; Error, Why?
   Myclass ob3("40");
   Myclass ob4 = "60";  // Ok, Why?

   cout << "ob1: " << ob1.getA() << endl;
   cout << "ob3: " << ob3.getA() << endl;
   cout << "ob4: " << ob4.getA() << endl;

   return 0;
}
```

**OUTPUT**:

```
ob1: 10
ob3: 40
ob4: 60
```

# Conversion Function in C++

- A conversion function automatically converts an object into a compatible value.

```
operator type() { return value;}
```

```
#include <iostream>
#include <cstring>
using namespace std;

class Rectangle{
    char name[20];
    int length;
    int wide;
public:
    Rectangle(char *name, int length, int wide){
        strcpy(this->name, name);
        this->length = length;
        this->wide = wide;
    }
    operator double(){
        return length * wide;
    }
    operator char*(){
        return name;
    }
};
```

```
int main(){
    Rectangle r("Rectangle", 5, 10);
    double area = r;
    cout << "Area: " << area << endl;
    char* name = r;
    cout << "Name: " << name << endl;
    return 0;
}
```

**OUTPUT**:

```
Area: 50
Name: Rectangle
```

# Overloading in Java

**Java doesn't support customized operator overloading.**

**Conversion Function in C++ =>**
**Auto Unboxing in Java**

# Auto-Boxing and Auto Unboxing in Java

- **Java wrapper wraps primitive types into Objects. The available wrappers are-**

  `Double, Float, Long, Integer, Short, Byte, Character and Boolean`

- **Auto Boxing wraps primitive types into the respective Objects and unboxing unwraps Objects into primitive types.**

- **Two type of methods:**

`valueOf()` **-> convert value to object.**

`typeValue()` **-> convert object to primitive type.**

```
public class Main {
    public static void main(String[] args) {
        Integer iOb1 = Integer.valueOf(100);
        Integer iOb2 = 150;
        double num1 = iOb1;
        int num2 = iOb2.intValue();
        double sum1 = num1 + num2;
        int sum2 = iOb1 + iOb2;
        double num = iOb1.doubleValue();

        System.out.println("Primitive Sum: "+ sum1);
        System.out.println("Object sum: " + sum2);
        System.out.println("Primitive Sum: "+ num1 + num2);
        System.out.println("Object sum: " + iOb1 + iOb2);
        System.out.println(num);
    }
}
```

**OUTPUT:**

```
Primitive Sum: 250.0
Object sum: 250
Primitive Sum: 100.0150
Object sum: 100150
100.0
```