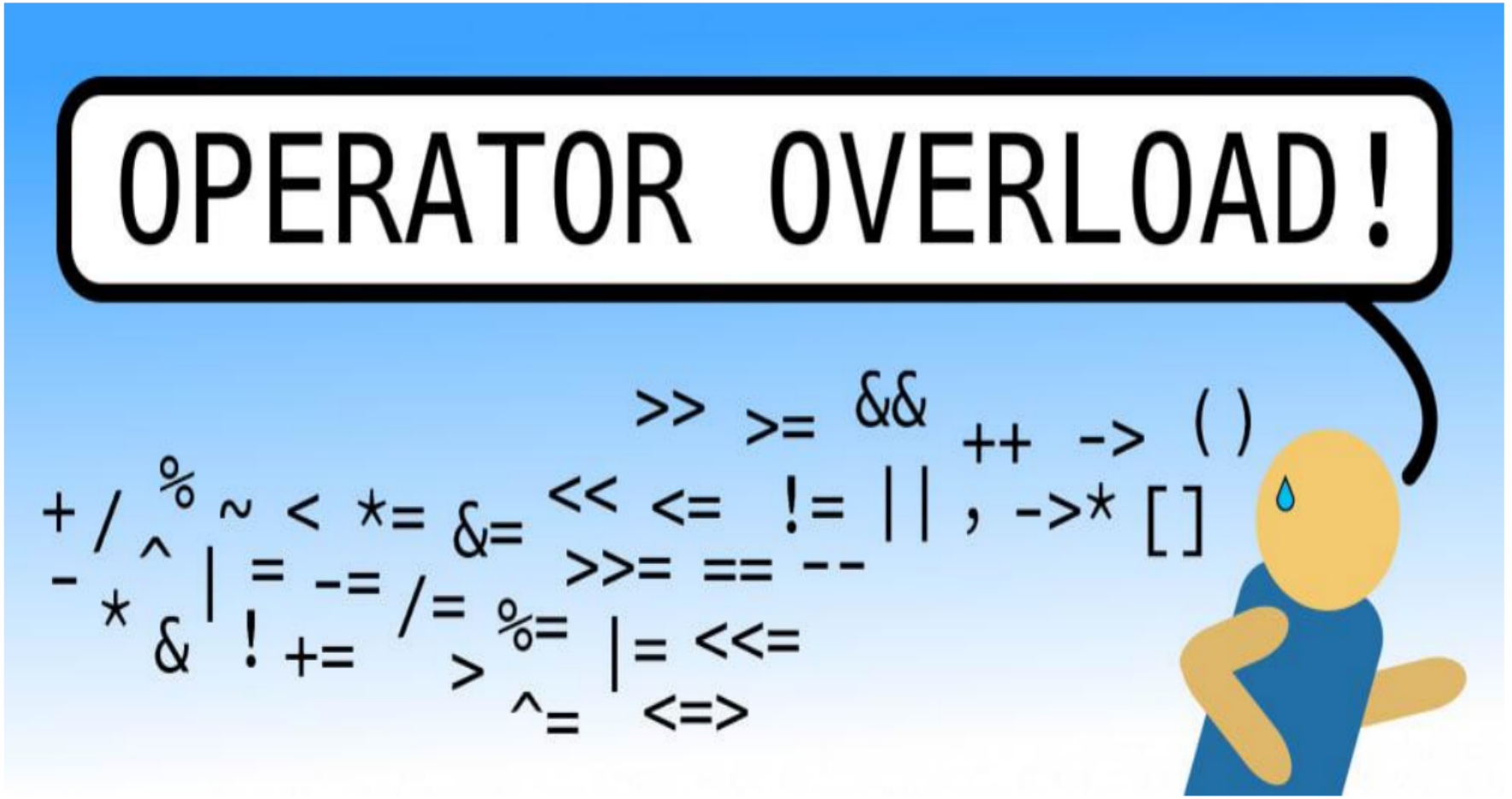
Unit-3

Operator overloading,

Inheritance


OPERATOR OVERLOAD!
>> >= && ++ -> ( )
+ / % ~ < *= &= << <= != || , ->* []
- * ^ | = -= /= %= |= <<=
& ! += > ^= <=>

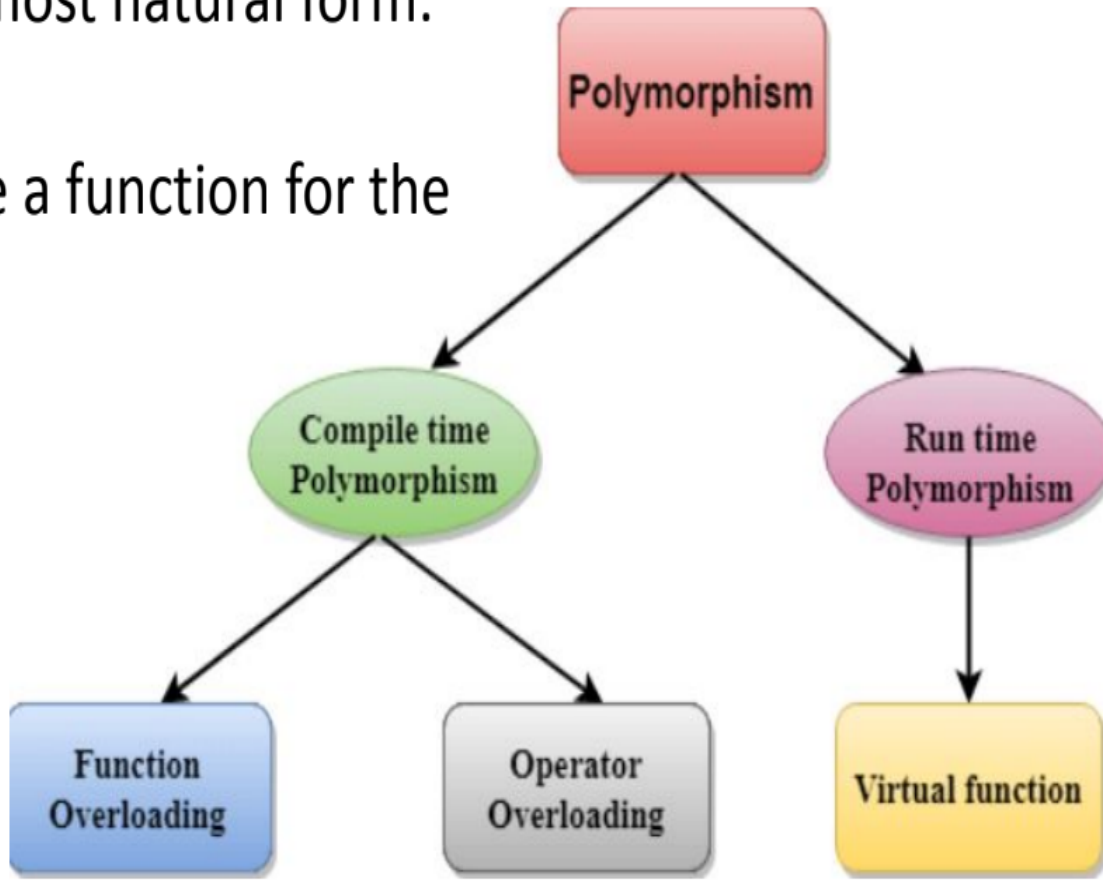
What is operator overloading?

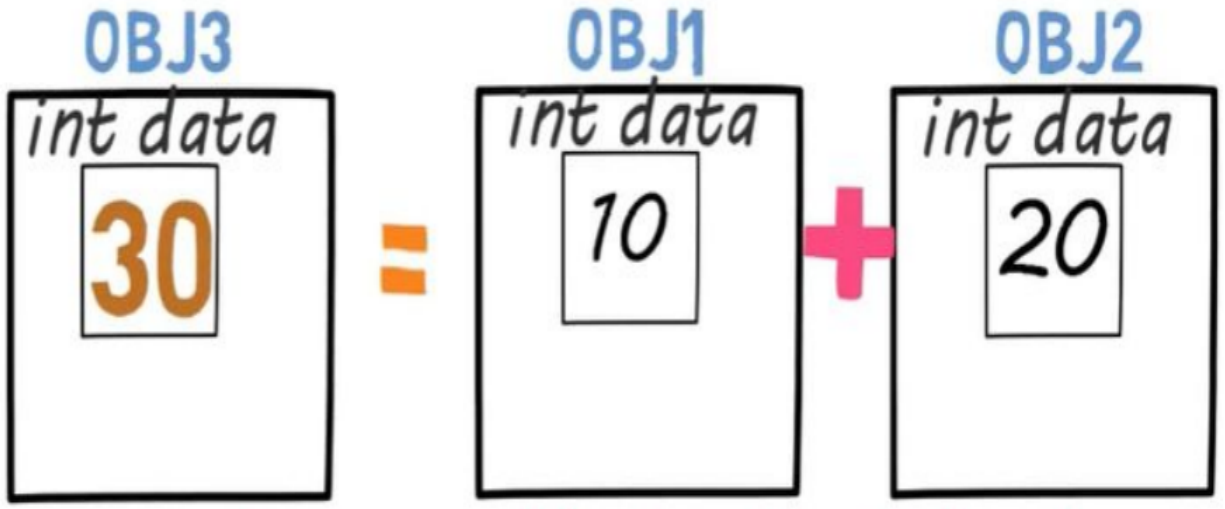
- Changing the definition of an operator so it can be applied on the objects of a class is called operator overloading.

- The basic purpose of operator overloading is used to provide facility to the programmer, to write expressions in the most natural form.

- To overload an operator, we need to write a function for the operator we are overloading.

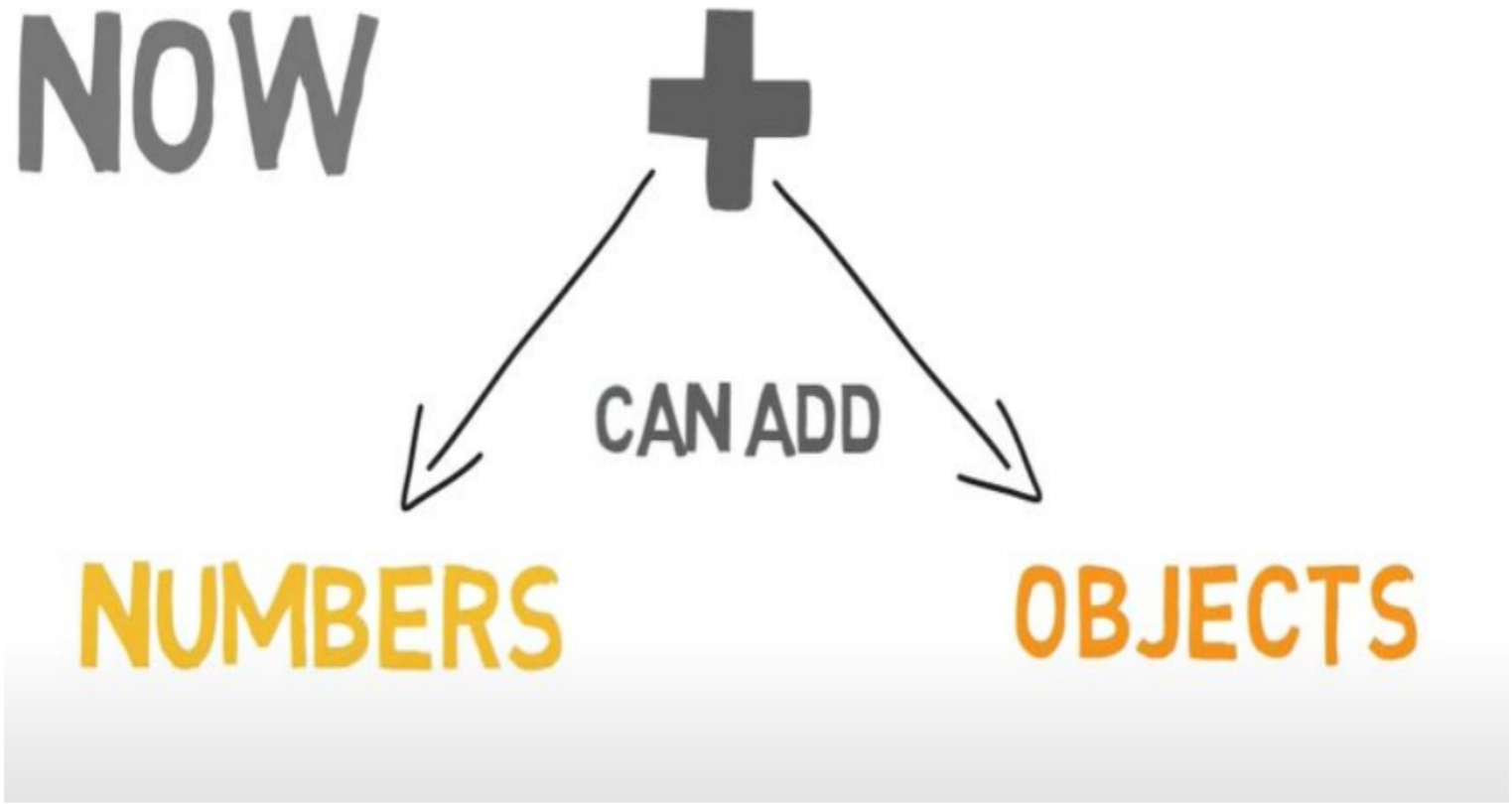
- It is a type of polymorphism.





Compiler throws error!

- for addition of objects OBJ1 and OBJ2, we need to define operator + (plus).
- Redefining the operator plus does not change its natural meaning.
- It can be used for both variables of built-in data type and objects of user-defined data type and this is called as operator overloading.

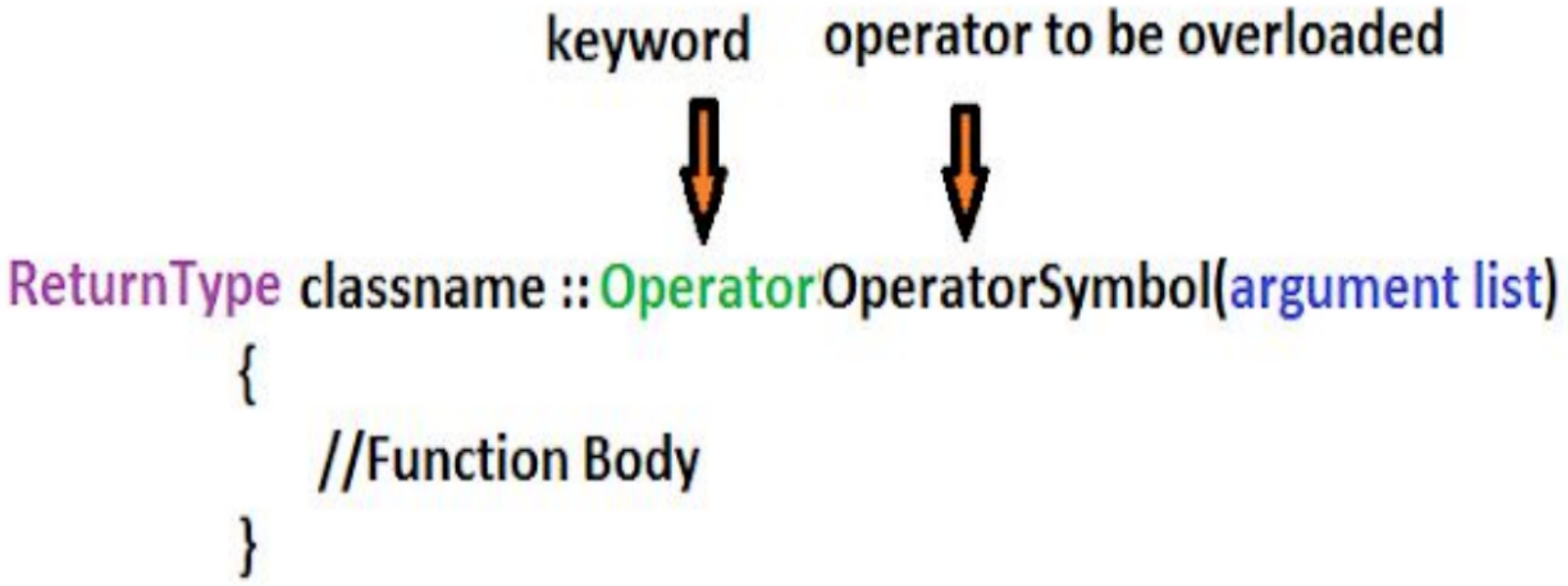

NOW
+
CAN ADD
NUMBERS
OBJECTS

Introduction to Operator overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`Operator +`** used to overload the addition operator (**+**)

- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator **(=)** or the address operator **(&)**
    - Assignment operator by default performs member wise assignment
    - Address operator (&) by default returns the address of an object

Operator Overloading Syntax:



Restrictions on Operator Overloading

- C++ operators that can be overloaded

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

- C++ Operators that cannot be overloaded

Operators that cannot be overloaded				
.	.*	::	?:	sizeof
Class membership operator	pointer to member operator	Scope resolution operator	ternary or conditional operator	

Where a friend function cannot be used	
=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
→	Class member access operator

Rules for Operator Overloading in C++

1. No new operators can be created, only existing operators can be overloaded.

2. The overloaded operator must have at least one operand that is of user-defined type

3. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc. We cannot change the basic meaning of an operator (+ for addition only)

4. Overloaded operators follow the syntax rules of the original operator. They cannot be overridden.

Rules for Operator Overloading in C++

5. There are some operators that cannot be overloaded.

6. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

7. We cannot use *friend function* to overload certain operators. However, the member function can be used to overload those operators.

8. When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

Rules for Operator Overloading in C++

9. When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

10. When using binary operators overloaded through a member function, left hand operand must be an object of the relevant class

11. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own meaning.

- Operator function must be either member functions (non – static) or friend functions.
- Arguments may be passed either by value or by reference.
- Operator functions are declared in the class using prototypes as follows:

vector operator+(vector);	// vector addition
vector operator-();	// unary minus
friend vector operator+(vector,vector);	// vector addition
friend vector operator – (vector);	// unary minus
vector operator- (vector &a);	// subtraction
int operator ==(vector);	// comparison
friend int operator==(vector,vector)	// comparison

Operator to Overload	Arguments passed to the Member Function	Arguments passed to the Friend Function
Unary Operator	**No**	**1**
Binary Operator	**1**	**2**

Overload the unary operator

```
#include <iostream>
using namespace std;
class Test  {
 private:         int num;
 public:
        Test(){
            cout<<"enter a number  ";
            cin>>num;
        }
    void operator ++()     {
        num = num+2;
    }
    void Print() {     cout<<"The Count is:
"<<num;
    }
};
```

```
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
    }
```

Overload a unary plus operator using a friend function

```
friend void operator++(Test &s); // declaration
void operator++(Test &s)// definition
   {
       s.num = s.num+2;
   }
```

```
int main()
{
   Test tt;
   ++tt;
   tt.Print();
   return 0;
   }
```

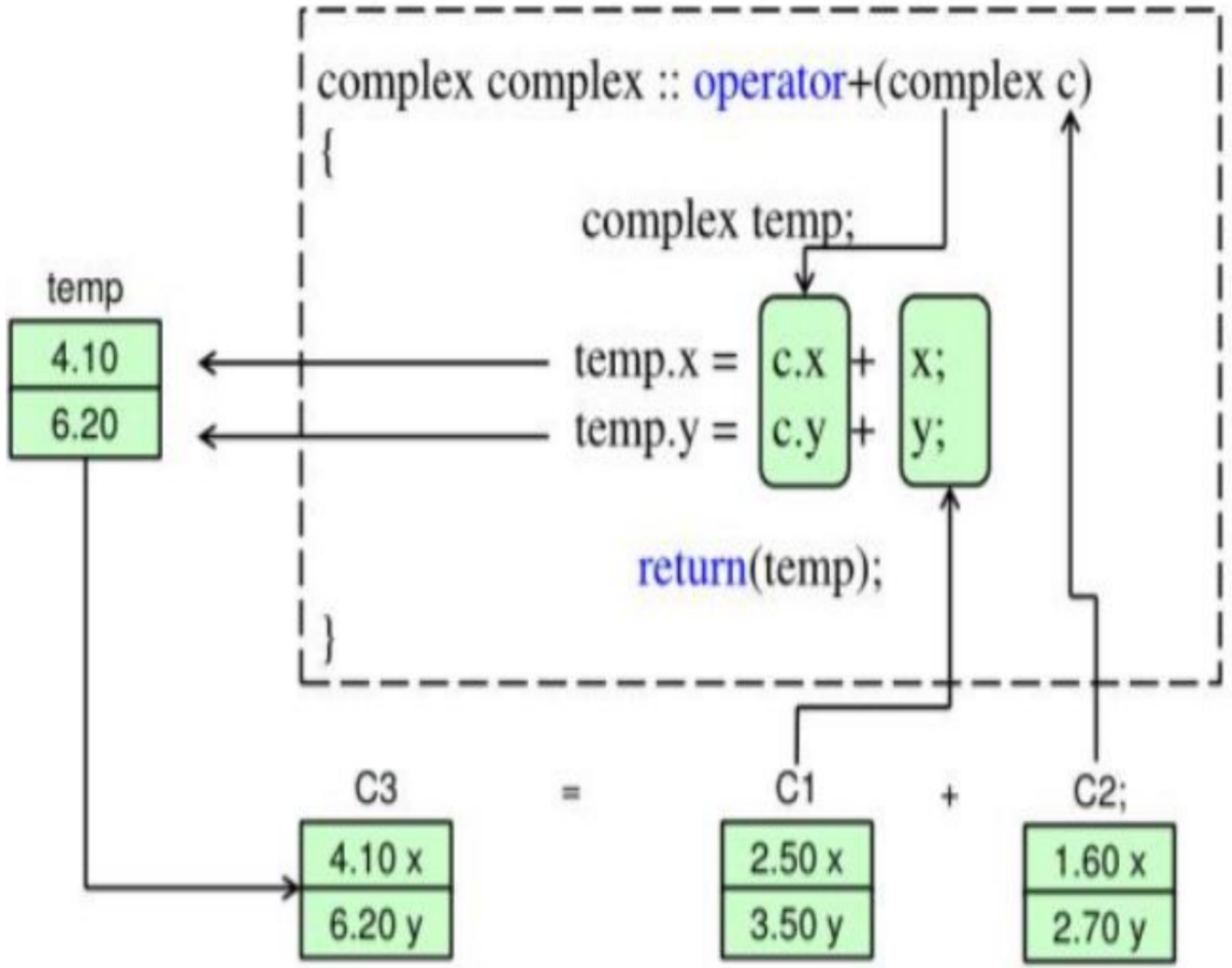
Overload the binary operator:

overloading + Operator to add two complex numbers

```
class complex
{
float x; // real part
float y; // imaginary part
public : complex() { } // constructor 1
complex(float real, float imag) // constructor 2
{
   x = real;
   y = imag;
}
complex operator + (complex);
void display(void);
};
```

```
complex complex :: operator+(complex c)
{
complex temp; //temporary
temp.x = x + c.x; // these are
temp.y = y + c.y; // statement for float additions
return(temp);
}
void complex :: display(void)
{
std::cout << x << " + i" << y << "\n";
}
```

```
int main()
{ complex c1, c2, c3; // invokes constructor 1
c1 = complex (2.5, 3.5); // invokes constructor 2
c2 = complex (1.6, 2.7);
c3 = c1 + c2;  // invokes operator+() function
std::cout << "c1 = "; c1.display();
std::cout << "c2 = "; c2.display();
std::cout << "c3 = "; c3.display();
return 0;
}


// invokes operator+() function
//  C3= c1.operator+(c2)
```


complex complex :: operator+(complex c)
{
complex temp;
temp
4.10
6.20
temp.x = c.x + x;
temp.y = c.y + y;
return(temp);
}
C3 = C1 + C2;
4.10 x
6.20 y
2.50 x
3.50 y
1.60 x
2.70 y

Overload the == operator in the complex class to directly compare two objects of complex class.

Creating prefix & postfix forms of ++ & -- operators

if ++ precedes its operand (++ob) operator ++() // function is valid

if ++ follows its operand (ob++) operator++(int x)// function is called,

//where x is a dummy argument and has the value 0.

prefix

```
type operator ++ ( )
{
        -----
}
```

Postfix

```
type operator ++ (int x)
{
        -----
}
```

Overloading short hand operators like +=, -=

```
loc loc::operator += (loc op)
{
        longitude = op.longitude + longitude;
         latitude = op.latitude + latitude;
        return *this;
}
```

```
#include <iostream>
using namespace std;
class Test  {
 private:         int num;
 public:
       Test(){
          cout<<"enter a number  ";
          cin>>num;
       }
   void operator ++()    {
      num = num+2;
   }

   void operator ++(int)    {
      num++;

   }

   void Print() {    cout<<"The Count is: "<<num;
   }
};
```

```
int main()
{
   Test tt;
   cout << "Pre-increment:";
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();


   cout << "\nPost-increment:";
   tt++; // calling of the post-increment operator
   tt.Print();
   return 0;
}
```

Overloading binary operators using friends

- In c++, friend functions may be used in the place of member functions for overloading a binary operator,
- so the only difference being that a friend function requires 2 (two arguments) to be explicitly passed to it,
- while a member function requires only one.

Replace the member function with **friend function** in ***complex class***

1). Replace the member declaration by the friend function declaration.

```
friend complex operator+(complex,complex);
```

2) Redefine the operator function as follows:

```
Complex operator+(complex a, complex b)
{
return complex (( a.x + b.x), (a.y+b.y));
}
```

```
Complex operator+(complex a, complex b){
complex temp;
temp.x = a.x + b.x;
temp.y = a.x + b.y;
return temp; }
```

In this case, the statement

```
C3 = C1+ C2;
```

Is equivalent to

```
C3 = operator+(C1 , C2);
```

we get the same results by the use of either a member function or a friend function. Why then an alternative is made available?

Where friend is a must ?

A= B +2; (or A = B * 2;)
Where in the above example A and B are objects of the same class. it will work for a member function but the statement

A = 2 + B; (or A = 2 * B) will not work.

- It is because the left-hand operand which is responsible for invoking the member function should be an object of the same class.
- However friend function allows both approaches. How?
- An object need to be used to invoke a friend function but can be passed as an argument.
- Thus, we can use a friend function or(method) with a built in type data as the left hand operand and an object as the right hand operand.

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

class MyClass {
   int value;

public:
   MyClass() {value=0;}
   MyClass(int val) {value=val;}

   void display() const {
      cout << "Value: " << value << endl;
   }

   // Friend function declarations
   friend MyClass operator+(const MyClass& lhs, int rhs);
   friend MyClass operator+(int lhs, const MyClass& rhs);
   friend MyClass operator*(const MyClass& lhs, int rhs);
   friend MyClass operator*(int lhs, const MyClass& rhs);
};
```

```
// Friend function definitions for addition
MyClass operator+(const MyClass& lhs, int rhs) {
   return lhs.value + rhs;
}

MyClass operator+(int lhs, const MyClass& rhs) {
   return lhs + rhs.value;
}

// Friend function definitions for multiplication
MyClass operator*(const MyClass& lhs, int rhs) {
   return lhs.value * rhs;
}

MyClass operator*(int lhs, const MyClass& rhs) {
   return rhs.value * lhs;
}
```

```
int main() {
    MyClass A, B(5);

    A = B + 2;
    cout << "A = B + 2: ";    A.display();

    A = 2 + B;
    cout << "A = 2 + B: ";    A.display();

    A = B * 2;
    cout << "A = B * 2: ";    A.display();

    A = 2 * B;
    cout << "A = 2 * B: ";    A.display();

    return 0;
}
```

```
A = B + 2: Value: 7
A = 2 + B: Value: 7
A = B * 2: Value: 10
A = 2 * B: Value: 10
```

overloading << Operator to print Class Object

```
class Time {
   int hr, min, sec;

public:   // Default constructor
   Time() {
      hr = 0;      min = 0;      sec = 0;
   }
   // Overloaded constructor
   Time(int h, int m, int s) {
      hr = h;      min = m;      sec = s;
   }
   // Overloading '<<' operator
   friend ostream& operator<<(ostream &out, Time &tm);
};
```

overloading << Operator to print Class Object

```
// Define the overloaded function
ostream& operator<<(ostream &out, Time &tm) {
    out << "Time is: " << tm.hr << " hour : " << tm.min << " min : " << tm.sec << " sec";
    return out;
}

void main() {
    Time tm(3, 15, 45);
    cout << tm;
}
```

// the time information is formatted
and sent to the output stream (`out`).

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as **+= , –=,** and the like.

```
loc loc::operator+=(loc op2)
{
longitude = op2.longitude + longitude;
latitude = op2.latitude + latitude;
return *this;
}
```

```
#include <iostream>
using namespace std;
class loc {    int longitude, latitude;
   public:       loc() {}
      loc(int lg, int lt) {
         longitude = lg;
         latitude = lt;
      }
      void show() {
         cout << longitude << " ";
         cout << latitude << "\n";
      }
      loc operator+(loc op2);
      loc operator-(loc op2);
      loc operator=(loc op2);
      loc operator++();
      loc operator+=(loc op2);
};
```

```
loc loc::operator+=(loc op2)
{
   longitude = op2.longitude + longitude;
   latitude = op2.latitude + latitude;
   return *this;
}

loc loc::operator+(loc op2)
{
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude;
   return temp;
}
```

```
loc loc::operator-(loc op2)
{
   loc temp;
   // notice order of operands
   temp.longitude = longitude - op2.longitude;
   temp.latitude = latitude - op2.latitude;
   return temp;
}
loc loc::operator=(loc op2)
{
   longitude = op2.longitude;
   latitude = op2.latitude;
   return *this; // i.e., return object that generated call
}
loc loc::operator++()
{
   longitude++;
   latitude++;
   return *this;
}
```

```
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show();       // displays 11 21
ob2 = ++ob1;
ob1.show();       // displays 12 22
ob2.show();       // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show();       // displays 90 90
ob2.show();       // displays 90 90
ob2+=ob3;
ob2.show();       // displays 180 180
return 0;
}
```

C++ Manipulation of string using operators

- ANSI C implements string character arrays, pointers and string functions.
- There are no operators for manipulation the strings.
- string can be defined as class objects which can be then manipulated like the built in types.
- since the strings vary greatly in size, we use new operator to allocate memory for each string and a pointer variable to point to the string array.

For example, we shall be able to use statements like

```
string3=string1+string2;
if(string1>=string2) string =string1;
```

A typical string class

```
class string
{
    char *p; // pointer to string
    int len; // length of string
public:
    ……. // member functions
    …… // to initialize and
    ……. // manipulate strings
};
```

```
#include <string.h>
#include <iostream>
class myString {
   char *p;
   int len;
public :
   myString()   {len=0; p=nullptr;};
   myString(const char *s); // create string from arrays
   myString(const myString &s); // copy constructor
   ~myString()   {delete[] p;}

friend myString operator+(const myString &s,const myString &t);
friend int operator<=(const myString &s, const myString &t);
friend void show(const myString s);
};
```

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

myString::myString(const myString &s)
{
   len=s.len;
   p=new char[len+1];
   strcpy(p,s.p);
}
```

```
// overloading + operator
myString operator +(const myString &s, const myString &t)
{
   myString temp;
   temp.len=s.len+t.len;
   temp.p=new char[temp.len+1];
   strcpy(temp.p,s.p);
   strcat(temp.p,t.p);
   std::cout << temp.p;
   temp.p[temp.len] = '\0';
   return(temp);
}
```

```
// overloading <= operator
int operator<=(const myString &s, const myString &t)
{
   int m=strlen(s.p);
   int n=strlen(t.p);
   if (m<=n)
      return (1);
   else
      return (0);
}
void show(const myString s)
{
std::cout<<s.p;
}
```

```
int main()
{
myString s1="new";
myString s2="york";
myString s3="delhi";
myString t1,t2,t3;
t1=s1;
t2=s2;
t3=s1+s3;
std::cout << "\n t1="; show(t1);
std::cout << "\n t2="; show(t2);
std::cout << "\n";
std::cout << "\nt3="; show(t3);
std::cout << "\n\n";
```

```
if (t1<=t3)
{
   show(t1);
   std::cout << " Smaller than ";
   show(t2);
   std::cout << "\n";
}
else
{
   show(t3);
   std::cout << "smaller than ";
   show(t1);
   std::cout << "\n";
}
return 0;
} ;
```

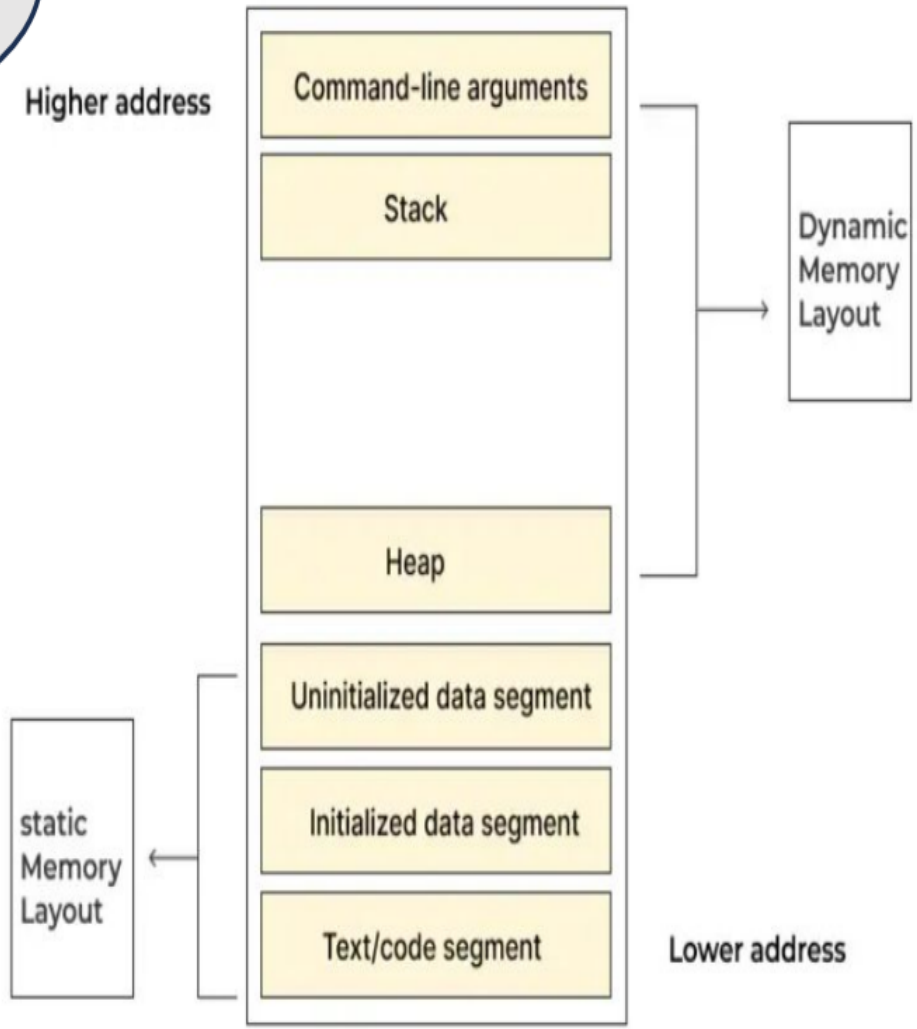
t1=New
t2=york
t3=new delhi
new smaller than new delhi

Overloading new and delete

- overload new and delete to use some special allocation method
- For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted

The type size_t is a defined type capable of containing the largest single piece of memory that can be allocated

```
// Allocate an object.
void *operator new(size_t si
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```



```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
   int longitude, latitude;
   public:       loc() {}
      loc(int lg, int lt) {
         longitude = lg;
         latitude = lt;
      }
      void show() {
         cout << longitude << " ";
         cout << latitude << "\n";
      }
      void *operator new(size_t size);
      void operator delete(void *p);
};
```

```
void *loc::operator new(size_t size)
{
   void *p;
   cout << "In overloaded new.\n";
   p = malloc(size);
   if(!p) {
      bad_alloc ba;
      throw ba;
   }
   return p;
}

void loc::operator delete(void *p)
{
   cout << "In overloaded delete.\n";
   free(p);
}
```

```
int main() {
   loc *p1, *p2;
   try {
      p1 = new loc (10, 20);
   }
   catch (bad_alloc xa) {
      cout << "Allocation error for p1.\n";
      return 1;
   }
   try {
      p2 = new loc (-10, -20);
   }
   catch (bad_alloc xa) {
      cout << "Allocation error for p2.\n";
      return 1;;
   }
 float *f = new float;
   p1->show();
   p2->show();
   delete p1;
   delete p2;
   return 0;
}
```

In C++, bad_alloc is an exception class that is typically thrown by the new operator when it fails to allocate memory. It is part of the <new> header.

```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

If we add,
float *f = new float; // uses default new

overload new and delete globally

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
   int longitude, latitude;
   public:
      loc() {}
      loc(int lg, int lt) {
         longitude = lg;
         latitude = lt;
      }
      void show() {
         cout << longitude << " ";
         cout << latitude << "\n";
      }
};
```

```
void *operator new(size_t size)
{
   void *p;
   cout << "In overloaded new.\n";
   p = malloc(size);
   if(!p) {
      bad_alloc ba;
      throw ba;
   }
   return p;
}

void operator delete(void *p)
{
   cout << "In overloaded delete.\n";
   free(p);
}
```

```
int main() {
    loc *p1, *p2;
    try {
        p1 = new loc (10, 20);
    }
    catch (bad_alloc xa) {
        cout << "Allocation error for p1.\n";
        return 1;
    }
    try {
        p2 = new loc (-10, -20);
    }
    catch (bad_alloc xa) {
        cout << "Allocation error for p2.\n";
        return 1;;
    }
```

```
try {
    f = new float; // uses overloaded new, too
    } catch (bad_alloc xa) {
        cout << "Allocation error for f.\n";
        return 1;;
        }
    *f = 10.10F;
    cout << *f << "\n";
    p1->show();
    p2->show();
    delete p1;
    delete p2;
    return 0;
}
```

```
In overloaded new.
In overloaded new.
In overloaded new.
10.1
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

Overloading Some Special Operators

- C++ defines array subscripting, function calling, and class member access as operations.

- The operators that perform these functions are the [ ], ( ), and –>, respectively.

- One important restriction applies to overloading these three operators:
- They must be nonstatic member functions. They cannot be friends.

Overloading [] Operator

In C++, the [ ] is considered a binary operator when you are overloading it.

Therefore, the general form of a member operator[ ]( ) function is as shown here:

```
type class-name::operator[](int i)
{
  // . . .
}
```

Given an object called O, the expression O[3]

translates into this call to the operator[ ]( ) function: O.operator[](3)

Overloading [] Operator

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

class atype {
     int a[3];
   public:
     atype(int i, int j, int k) {
                a[0] = i;
                a[1] = j;
                a[2] = k;
     }
   int operator[](int i) { return a[i]; }
};
```

```
int main()
{
   atype ob(1, 2, 3);
   cout << ob[1]; // displays 2
   return 0;
}
```

Overloading [] Operator

- You can design the operator[ ]( ) function in such a way that the [ ] can be used on both the left and right sides of an assignment statement.
- To do this, simply specify the return value of operator[ ]( ) as a reference.

```
#include <iostream>
using namespace std;
class atype {
    int a[3];
  public:
    atype(int i, int j, int k) {
            a[0] = i;
            a[1] = j;
            a[2] = k;
    }
  int &operator[](int i) { return a[i]; }
};
```

```
int main()
{
atype ob(1, 2, 3);
cout << ob[1];          // displays 2
cout << " ";
ob[1] = 25;         // [ ] on left of =
cout << ob[1];          // now displays 25
return 0;
}
```

Overloading ( )

- Creating an operator function that can be passed an arbitrary number of parameters.
- When we use the ( ) operator in your program, the arguments specified are copied to those parameters.
- The object that generates the call is pointed to by the this pointer.
- When overloading ( ), we can use any type of parameters and return any type of value.
- Can also specify default arguments.

```
#include <iostream>
using namespace std;
class loc {
   int longitude, latitude;
public: loc() {}
      loc(int lg, int lt) {
         longitude = lg;
         latitude = lt;
      }
      void show() {
         cout << longitude << " ";
         cout << latitude << "\n";
      }
      loc operator+(loc op2);
      loc operator()(int i, int j);
};
```

```
loc loc::operator()(int i, int j){
   longitude = i;
   latitude = j;
   return *this;
}
loc loc::operator+(loc op2){
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude;
   return temp;
}
int main() {
   loc ob1(10, 20), ob2(1, 1);
   ob1.show();    // 10 20
   ob1(7, 8); // can be executed by itself
   ob1.show();    // 7 8
   ob1 = ob2 + ob1(10, 10); // can be used in expressions
   ob1.show();    return 0;  }     // 11 11
```

Overloading –>

- The –> pointer operator, also called the class member access operator.
- It is considered a unary operator when overloading.
- Its general usage is : object->element;
- Here, object is the object that activates the call.
- The operator–>( ) function must return a pointer to an object of the class that operator–>( ) operates upon.
- The element must be some member accessible within the object.

```
#include <iostream>
using namespace std;
class myclass {
   public:
   int i;
   myclass *operator->()
    {
    return this;
    }
};
int main()
{
myclass ob;
ob->i = 10;  // same as ob.i
cout << ob.i << " " << ob->i;       // 10 10
return 0;
}
```

Overloading the Comma Operator

- The comma is a binary operator, and like all overloaded operators.
- We can make an overloaded comma perform any operation.
- The rightmost value becomes the result of the comma operation.
- This is the way the comma works by default in C++.

```
#include <iostream>
using namespace std;
class loc {
   int longitude, latitude;
public:
   loc() {}
   loc(int lg, int lt) {
      longitude = lg;
      latitude = lt;
   }
   void show() {
      cout << longitude << " ";
      cout << latitude << "\n";
   }
   loc operator+(loc op2);
   loc operator,(loc op2);
};
```

```
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}

loc loc::operator,(loc op2)
{
loc temp;
temp.longitude = op2.longitude;
temp.latitude = op2.latitude;
cout<<"in comma operator\n";
cout << op2.longitude << " " << op2.latitude
<< "\n";
return temp;
}
```

```
int main()
{
loc ob1(10, 20), ob2( 5, 30),
ob3(1, 1);
ob1.show();
ob2.show();
ob3.show();
cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1,
the value of ob3
return 0;
}
```

```
10 20
5 30
1 1

in comma operator
10 60
in comma operator
1 1
1 1
```

Special cases

- The assignment operator (=) may be used with every class without explicit overloading.
- The default behavior of the assignment operator is a *memberwise assignment* of the data members of the class.
- The address operator (&) may also be used with objects of any class without explicit overloading.
- It returns the address of the object in memory.

TYPE CONVERSIONS

```
int m;
float x = 3.14159;
m = x; // automatic type conversion take place
cout<<m; // 3 will be displayed
```

- An assignment operation causes the automatic type conversion.
- The type of data to the right of an assignment operator is automatically converted to the type of variable on the left.

TYPE CONVERSIONS

Time T1;

int m;

m=T1; Class type will not be converted to basic type

OR

T1=m; basic type will not be converted to Class type

automatically

TYPE CONVERSIONS

- For user defined datatypes programmers have to convert it by using constructor or by using casting operator.

- There are three types of situations that arise where data conversion are between incompatible types.

1. Conversion from built in type to class type.(using constructor)
2. Conversion from class type to built in type.(using casting operator)
3. Conversion from one class type to another.(using constructor & using casting operator)

1. Basic to Class Type

```
#include <iostream>
using namespace std;
class sample
{
    int a;
    public:
    sample(){}
    sample(int x){
        a=x;
    }
    void disp()
    {cout<<"the value of a is = "<<a;
}
};
```

```
int main()
{
    int m=10;
    sample s;
    s=m;
    s.disp();
    return 0;
}
```

1. Basic to Class Type

```
class time1
{
int hours;
int minutes;
public:
time1 (int t) // constructor
{
    hours = t / 60; //t is inputted in minutes
    minutes = t % 60;
}
};
```

```
time1 TI; //object TI created
int period = 160;
TI = period; //int to class type
```

The left-hand operand of = operator is always a class object.

Hence, we can also accomplish this conversion using an overloaded = operator.

1. Basic to Class Type -constructor with single argument

```
Class string {
char *p;
int len;
string (char *a);
}
string :: string (char *a) {
length = strlen (a);
p = new char [length + 1];
strcpy(p,a);
}
```

This constructor builds a string type object from a char* type variable a.

```
string s1 , s2;
char* name1 = "Good Morning";
char* name2 = " STUDENTS" ;
s1 = string(name1);  //first converts name 1 from char*
                     type to string type and then assigns the
                     string type values to the object s1.
s2 = name2;    //performs the same job by invoking the
               constructor implicitly.
```

2. Class to Basic Type

-overloading casting function/ converts an operator

- The constructor functions do not support conversion from a class to basic type.

- C++ allows us to define a overloaded casting operator that convert a class type data to basic type.

- The general form:

```
operator typename ( )
{
   //Program statements
   return
}
```

2. Class to Basic Type

-overloading casting function/ converts an operator

- The casting operator should satisfy the following conditions.

1. It must be a class member.
2. It must not specify a return type.
3. It must not have any arguments.

Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function.

As a result function does not need an argument.

2. Class to Basic Type

```
#include <iostream>
using namespace std;
class sample
{   float a;
    public:
    sample()   {      a=23.45;   }

    operator int()
    {
        int x;
        x=a;
        return x;
    }
};
```

```
 int main()
{
    sample s;
    int y=s;
    cout<<" the value of y is ="<<y;
    return 0;
}
```

```
#include <iostream>
using namespace std;
class time1
{
int hours;
int minutes;
public:
   time1 (int h,int m) // constructor
   {   hours = h;    minutes = m;   }

   void display()
   {
      cout<<"hrs = "<<hours;
      cout<<"\nminutes= "<<minutes;
   }
      operator int()
   {      return hours*60+minutes;   }
};
```

```
int main() {
   int dur;
   time1 t(2,40);
   t.display();
   dur=t;
   cout<<"\nTime in interger value is = "<<dur;
   return 0;
}
```

```
const size = 3;
class vector {
int v[size];
operator double();
}
//scalar magnitude of a vector is calculated as the square root of the sum of the squares of its components
vector :: operator double() {
    double sum = 0;
    for (int i =0;i<size; i++)
        sum = sum + v[i]* v[i];
    return sqrt(sum);
}
double length = v1; // calls operator double ()
double length ;
length = v1; // calls operator double()
length = (double) v1; // calls operator double()
length = v1;
```

// valid , first it will look for appropriately overloaded assignment operator and if it is not found it will call any function (constructor or conversion function) to achieve the conversion although some books appreciate the syntax as given : static_cast<double>(v1);

- we can convert the object string to char* as follows:

```
string:: operator char*( )
{
return (str) ;
}
```

3. One Class to Another Class Type

constructor in destination class / conversion operation in source class

- Conversion between objects of different classes can be carried out by either a
  - constructor or
  - conversion function.
- Which form to use, depends upon where we want the type conversion function to be located, whether in the source class or in the destination class.

In case of conversion between objects constructor function is applied to destination class.

Conversion function

- The casting operator function:

  operator typename( )

- Converts the class object of which it is a member to typename.
- The type name may be a built-in type or a user defined one(another class type)
- In the case of conversions between objects, typename refers to the destination class.
- Therefore, when a class needs to be converted, a casting operator function can be used.
- The conversion takes place in the source class and the result is given to the destination class object.

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

class alpha
{
    int a;
    public:
        alpha(){}
        alpha(int x)
        {
            a=x;
        }
        //operator function
        operator beta();
};
```

```
class beta
{
    int b;
    public:
        beta(){}
        beta(int x)
        {
            b=x;
        }

        void disp()
        {
            cout<<" value is "<< b;
        }
};
```

```
alpha::operator beta()
{
    return beta(a);
}
int main()
{
    alpha obja(10);
    beta objb=obja;
    objb.disp();
    return 0;
}
```

@Source

Constructor

- Consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member.

- The argument belongs to the source class and is passed to the destination class for conversion.

- Therefore the conversion constructor must be placed in the destination class.

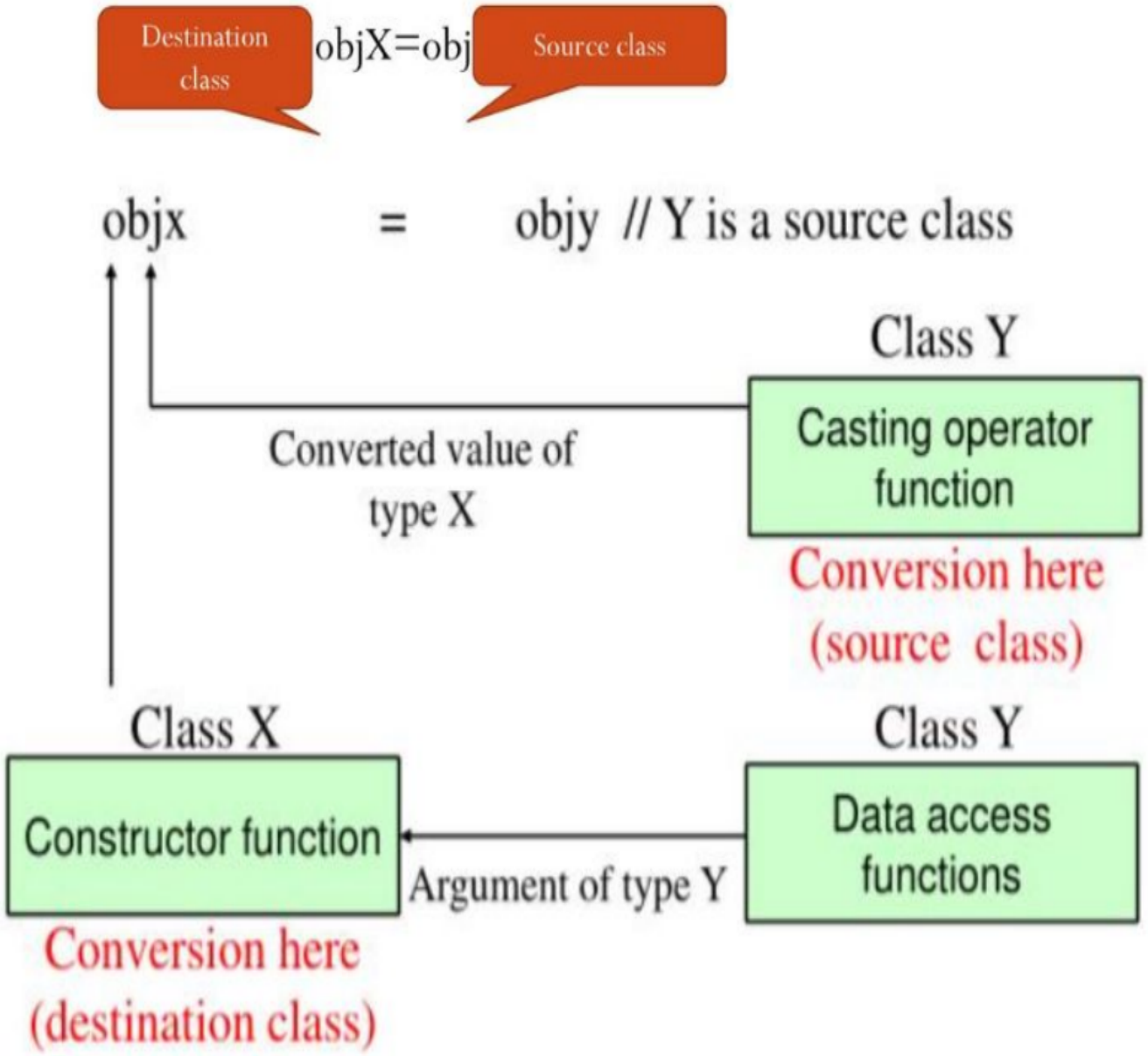
```
#include <iostream>
using namespace std;
class alpha
{
    int a;
    public:
        alpha(){}
        alpha(int x)
{       a=x;    }

        int getvalue()
        {           return a;
}
};
```

```
class beta
{
    int b;
    public:
        beta(){}
        beta(int x)
        {
            b=x;
        }
        //Constructor
        beta(alpha temp)
        {
            b=temp.getvalue();
        }
        void disp()
        {
            cout<<" value is "<< b;
        }
};
```

```
int main()
{
    alpha obja(10);
    beta objb=obja;
    objb.disp();
    return 0;
}
```

@Destination

In case of conversion between objects constructor function is applied to destination class.



```
#include <iostream>
using namespace std;
class stock2;
class stock1
{
   int code, item;
   float price;
   public:
   stock1(int a, int b, float c)
   {
      code=a; item=b; price=c;
   }
   void disp( )
   {
      cout<<"code"<<code <<"\n";
      cout<<"Items"<<item <<"\n";
      cout<<"Price per item Rs . "<<price <<"\n";
   }
   int getcode( ) {return code; }
   int getitem( ) {return item; }
   int getprice( ) {return price;}
   operator float( )    {return ( item*price );}
};
```

```
class stock2
{
   int code;
   float val;
   public:
   stock2()
   {
      code=0; val=0;
   }
   stock2(int x, float y)
   {
      code=x; val=y;
   }
   void disp( )
   {
      cout<< "code "<<code  << "\n";
      cout<< "Total Value Rs . "<<val <<"\n";
   }
   stock2(stock1 p)
   {
      code=p.getcode( ) ;
      val=p.getitem( ) * p.getprice ( ) ;
   }
};
```

```
int main ( ) {
   stock1 i1(101, 10, 125.0);
   stock2 i2;
   float tot_val;

   tot_val=i1 ; // Stock1 to float
   i2=i1;  // Stock1 to Stock2

   cout<<" stock1-type" <<"\n";
   i1.disp( ) ;
   cout<<"\n Stock value"<<"\n";
   cout<< tot_val<<"\n";
   cout<<"\n tock2-type"<< "\n";
   i2.disp( ) ;
return 0;
}
```

```
stock1-type
code101
Items10
Price per item Rs . 125

 Stock value
1250

stock2-type
code 101
Total Value Rs . 1250
```

Type Conversions

Conversion required	Conversion take place in	
	Source class	Destination class
Basic->class	NA	constructor
Class->basic	Casting operator	NA
Class->class	Casting operator	constructor