Operator Overloading

Operator Overloading

```
int a=5, b=10,c;
c = a + b;
```

Operator **+** performs **addition** of **integer operands** a, b

```
time t1,t2,t3;
t3 = t1 + t2;
```

Operator **+** performs **addition** of **objects** of type time

```
string str1=“Hello”
string str2=“Good Day”;
string str3;
str3 = str1 + str2;
```

Operator **+ concatenates** two strings str1,str2

Operator overloading

- **Function overloading** allow you to use same function name for different definition.
- **Operator overloading** extends the overloading concept to operators, letting you assign multiple meanings to C++ operators
- **Operator overloading** giving the normal C++ operators such as +, * and == additional meanings when they are applied with **user defined data types**.

Some of C++ Operators are already overloaded

Operator	Purpose
*	As pointer, As multiplication
<<	As insertion, As bitwise shift left
&	As reference, As bitwise AND

Operator Overloading

- Specifying more than one definition for an **operator** in the same scope, is called **operator overloading**.
- You can overload operators by creating ***“operator functions”***.

Syntax:

```
Return_type operator op_symbol(argument_list)
{
  // statements
}
```

Keyword

substitute the operator

Example:

```
void operator + (arguments);
int operator - (arguments);
Class_name operator / (arguments);
float operator * (arguments);
```

Rules for operator overloading

- Only existing operator can be overloaded.
- The overloaded operator must have at least one operand that is user defined type.
- We cannot change the basic meaning and syntax of an operator.

Rules for operator overloading (Cont…)

- When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
- We cannot overload following operators.

Operator	Name
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

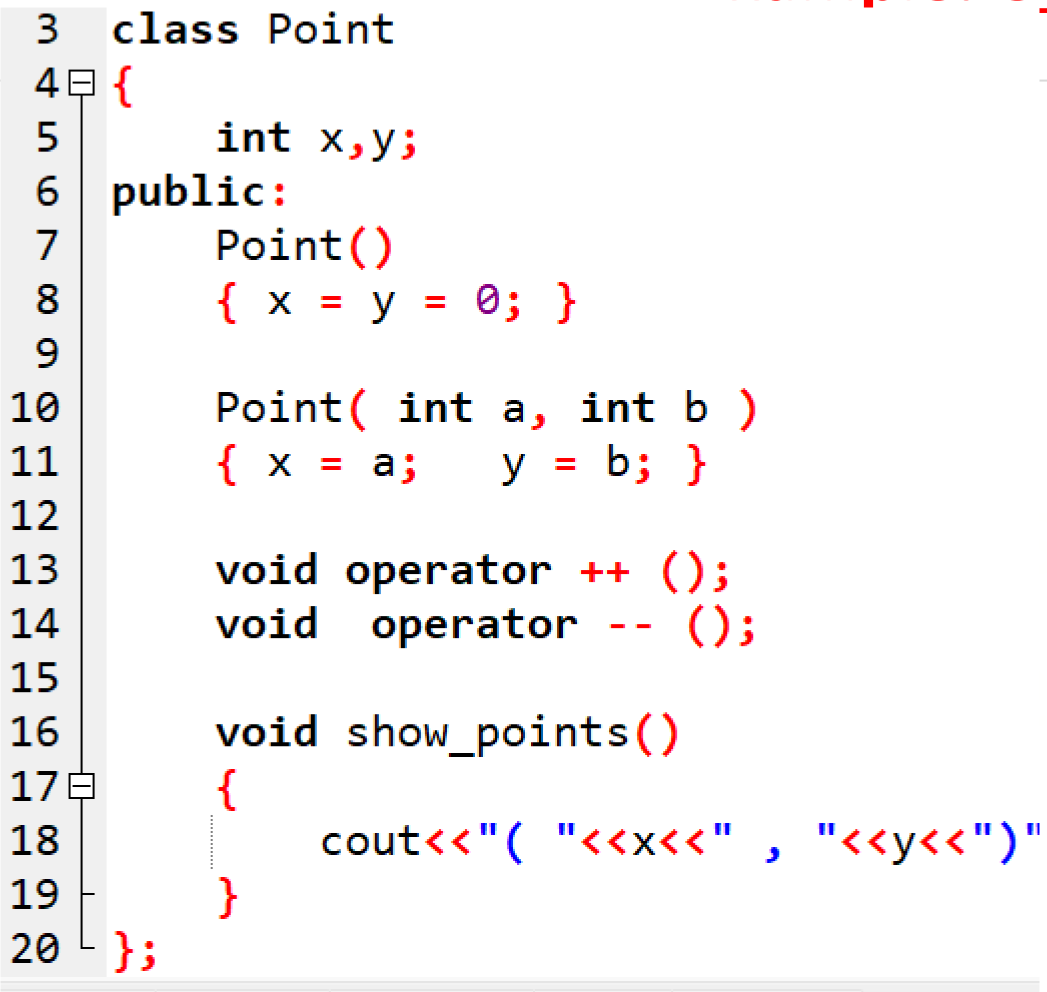
Overloading Unary Operator

Overloading Unary operator --

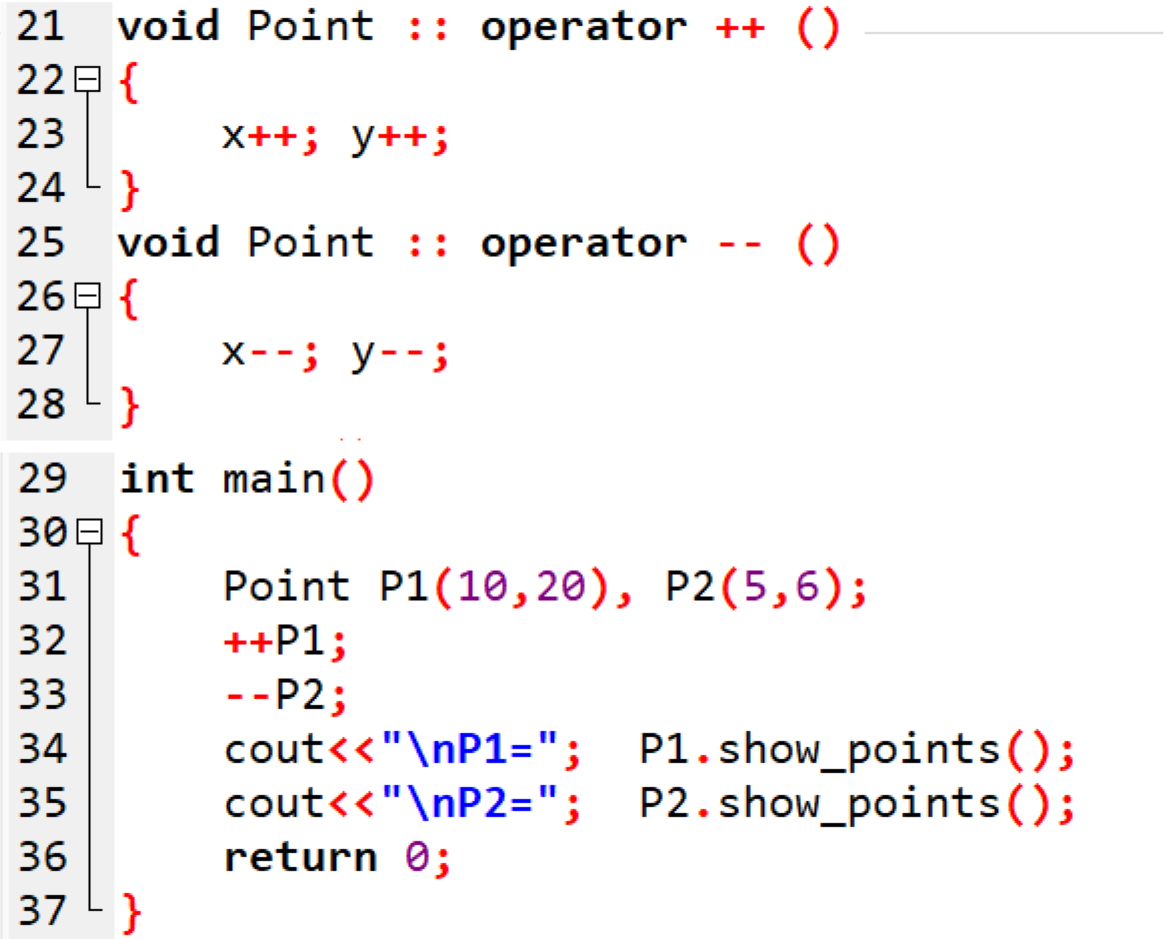
```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator--();
};
void space::operator--() {
  x--;
  y--;
  z--;
}
```

```
int main()
{
  space s1(5,4,3);
  s1.display();
  --s1;
  s1.display();
  return 0;
          }
```

Example: Unary operator overloading

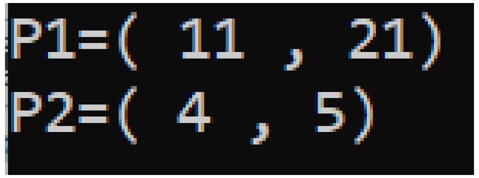


```cpp
class Point
{
    int x,y;
public:
    Point()
    { x = y = 0; }

    Point( int a, int b )
    { x = a;   y = b; }

    void operator ++ ();
    void  operator -- ();

    void show_points()
    {
        cout<<"( "<<x<<" , "<<y<<")"
    }
};
```



```cpp
void Point :: operator ++ ()
{
    x++; y++;
}
void Point :: operator -- ()
{
    x--; y--;
}
int main()
{
    Point P1(10,20), P2(5,6);
    ++P1;
    --P2;
    cout<<"\nP1=";  P1.show_points();
    cout<<"\nP2=";  P2.show_points();
    return 0;
}
```

OUTPUT

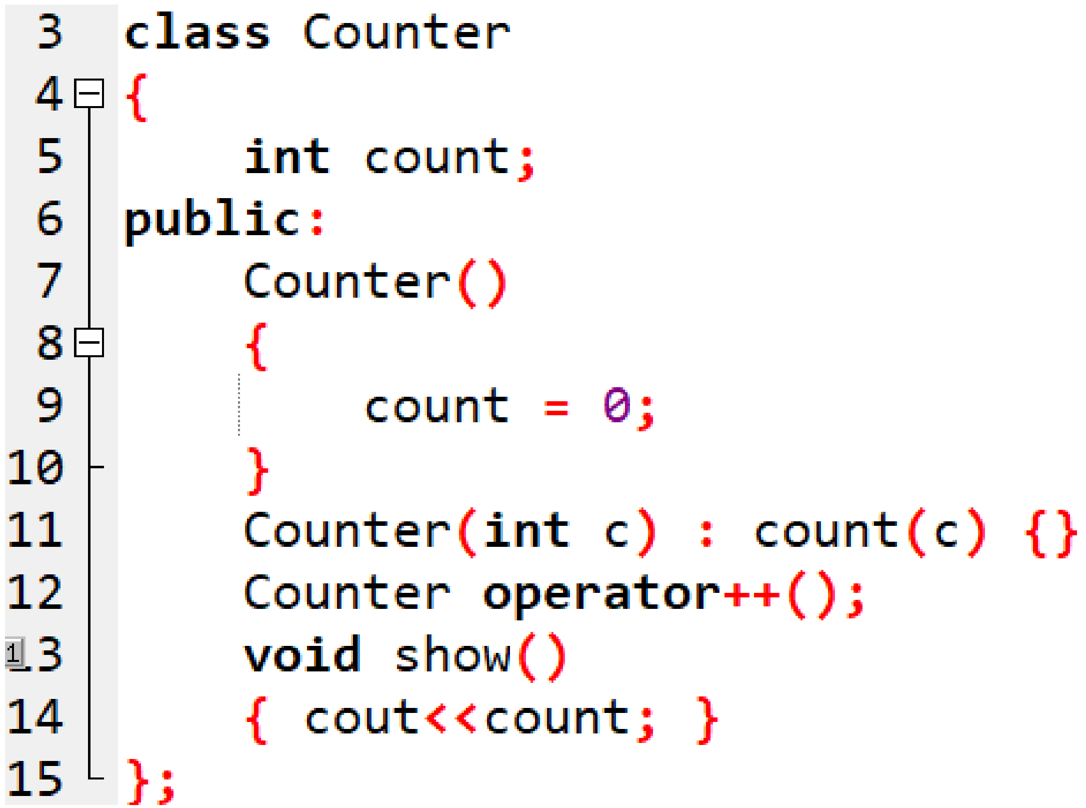


```
P1=( 11 , 21)
P2=( 4 , 5)
```

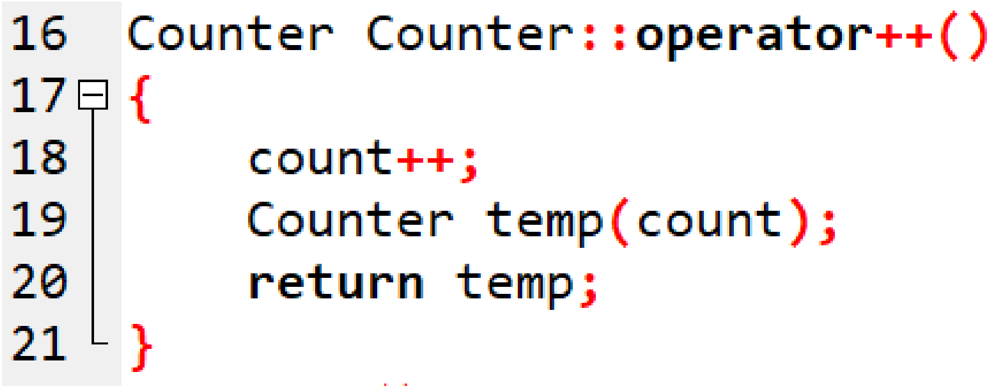
Overloading Unary operator –

```
class space {
  int x,y,z;
  public:
  space(){
    x=y=z=0;}
  space(int a, int b,int c){
    x=a; y=b; z=c; }
  void display(){
   cout<<"\nx="<<x<<",y="<<y<<",z="<<z;
  }
  void operator-();
};
void space::operator-() {
  x=-x;
  y=-y;
  z=-z;
}
```

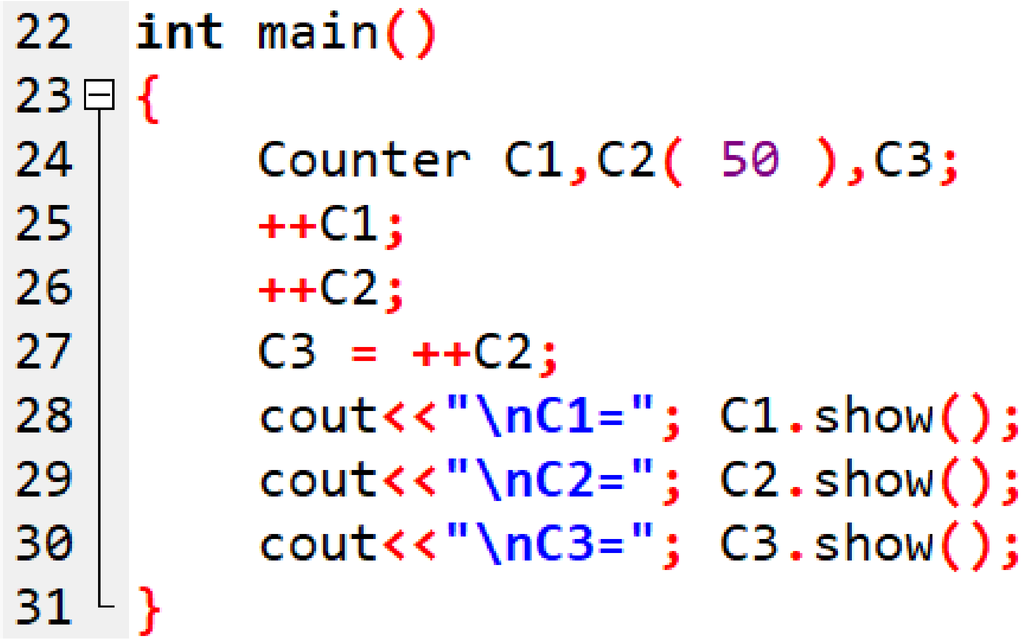
```
int main()
{
  space s1(5,4,3);
  s1.display();
  -s1;
  s1.display();
  return 0;
        }
```



```cpp
class Counter
{
    int count;
public:
    Counter()
    {
        count = 0;
    }
    Counter(int c) : count(c) {}
    Counter operator++();
    void show()
    { cout<<count; }
};
```

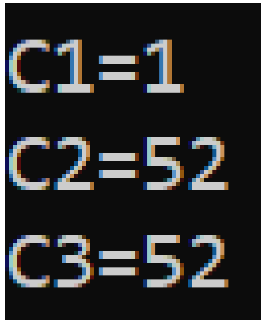


```cpp
Counter Counter::operator++()
{
    count++;
    Counter temp(count);
    return temp;
}
```



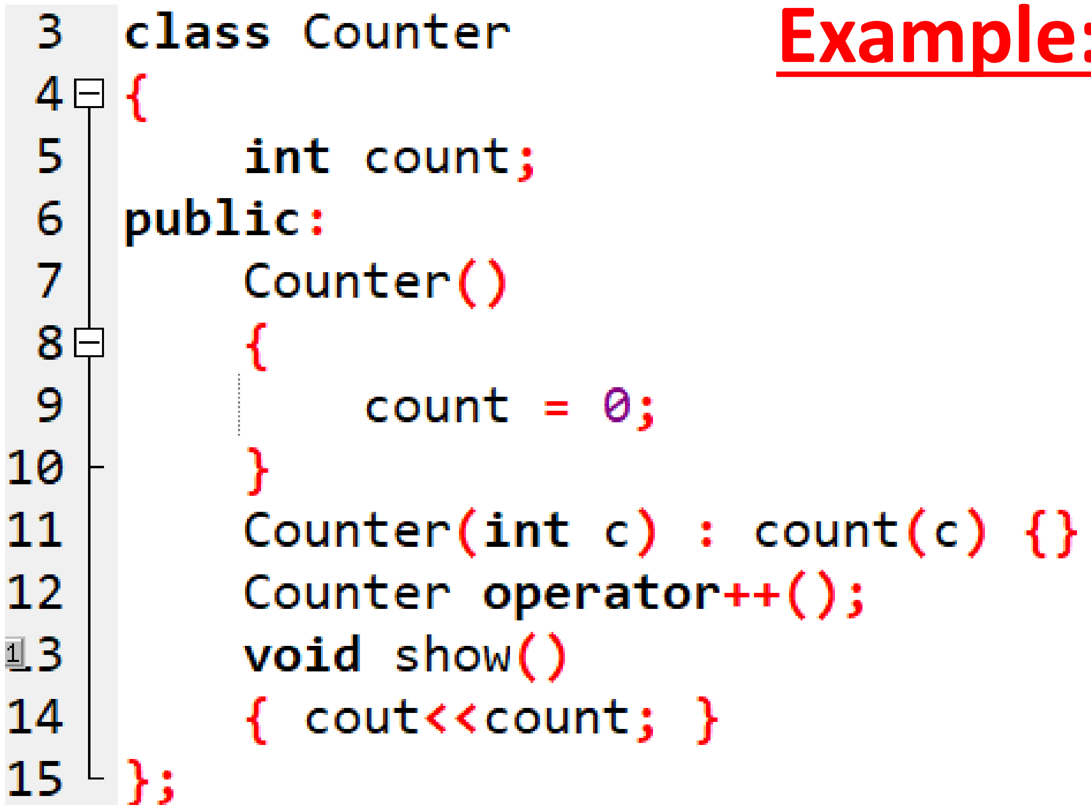
```cpp
int main()
{
    Counter C1,C2( 50 ),C3;
    ++C1;
    ++C2;
    C3 = ++C2;
    cout<<"\nC1="; C1.show();
    cout<<"\nC2="; C2.show();
    cout<<"\nC3="; C3.show();
}
```

OUTPUT:

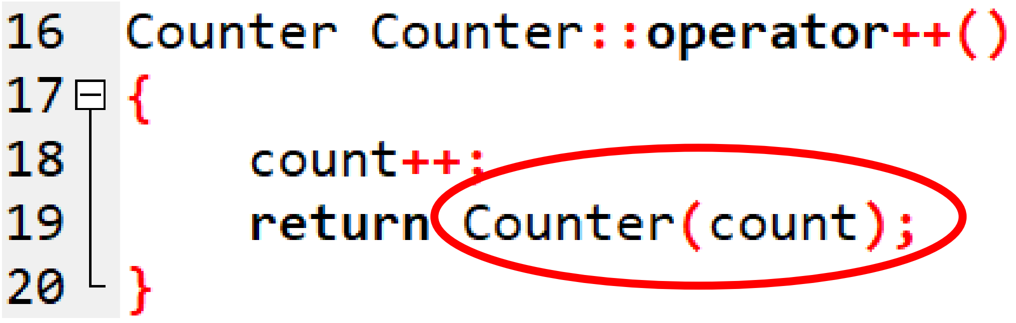


```
C1=1
C2=52
C3=52
```

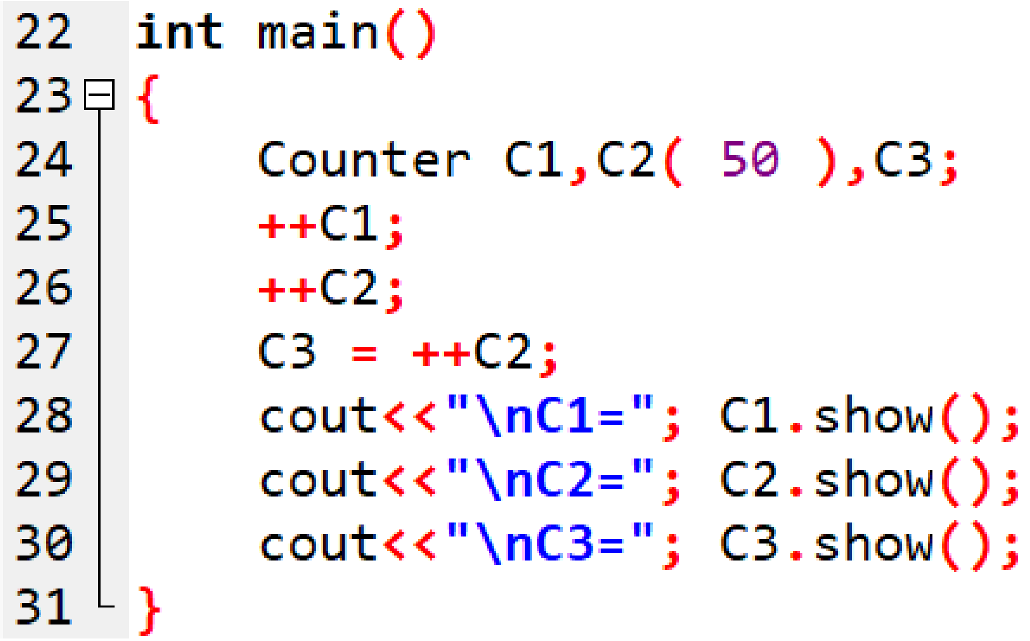
Example: Nameless Temporary Objects



```cpp
class Counter
{
    int count;
public:
    Counter()
    {
        count = 0;
    }
    Counter(int c) : count(c) {}
    Counter operator++();
    void show()
    { cout<<count; }
};
```

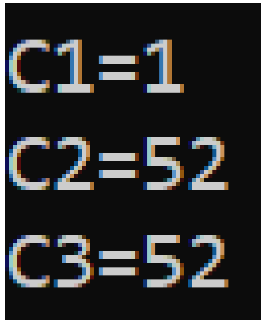


```cpp
Counter Counter::operator++()
{
    count++;
    return Counter(count);
}
```



```cpp
int main()
{
    Counter C1,C2( 50 ),C3;
    ++C1;
    ++C2;
    C3 = ++C2;
    cout<<"\nC1="; C1.show();
    cout<<"\nC2="; C2.show();
    cout<<"\nC3="; C3.show();
}
```

OUTPUT:



```
C1=1
C2=52
C3=52
```

Overloading Prefix and Postfix operator

```
class demo
{
    int m;
    public:
     demo(){ m = 0;}
     demo(int x)
     {
       m = x;
     }
     void operator ++()
     {
        ++m;
        cout<<"Pre Increment="<<m;
     }
     void operator ++(int)
     {
        m++;
        cout<<"Post Increment="<<m;
     }
};
```

```
int main()
{
    demo d1(5);
    ++d1;
    d1++;
}
```

Overloading Binary operator +

```
class complex{
  int real,imag;
  public:
    complex(){
     real=0; imag=0;
    }
    complex(int x,int y){
     real=x; imag=y;
    }
    void disp(){
     cout<<"\nreal value="<<real<<endl;
     cout<<"imag value="<<imag<<endl;
    }
    complex operator + (complex);
};
complex complex::operator + (complex c){
  complex tmp;
  tmp.real = real + c.real;
  tmp.imag = imag + c.imag;
  return tmp;
}
```

```
int main()
{
  complex c1(4,6),c2(7,9);
  complex c3;
  c3 = c1 + c2;
  c1.disp();
  c2.disp();
  c3.disp();
  return 0;
}
```

Similar to function call

`c3=c1.operator +(c2);`

Binary Operator Arguments

```
result = obj1.operator symbol (obj2);//function notation
```

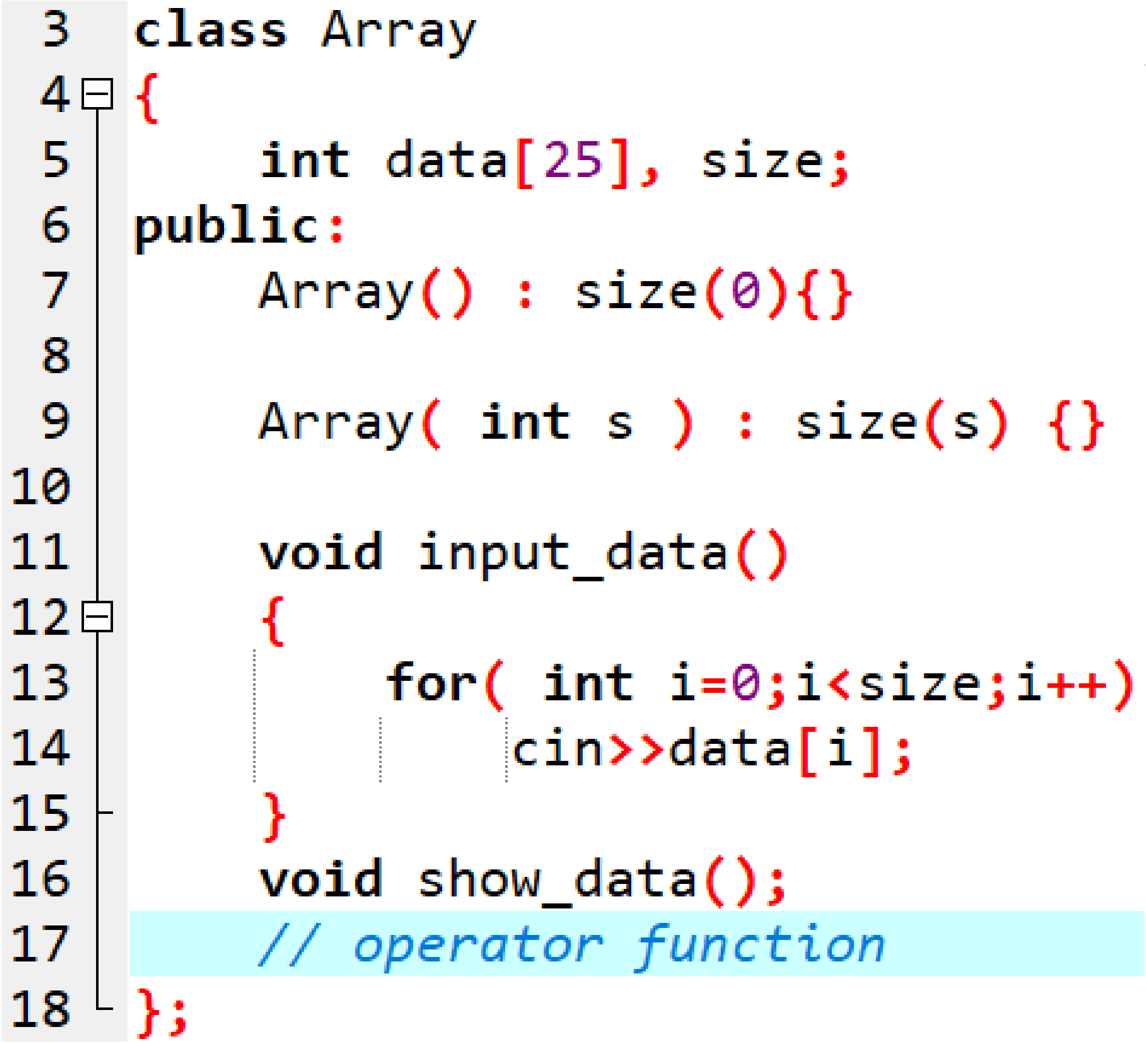
```
result = obj1 symbol obj2;           //operator notation
```

```
complex operator + (complex x)
{
   complex tmp;
   tmp.real = real + x.real;
   tmp.imag = imag + x.imag;
   return tmp;
}
```

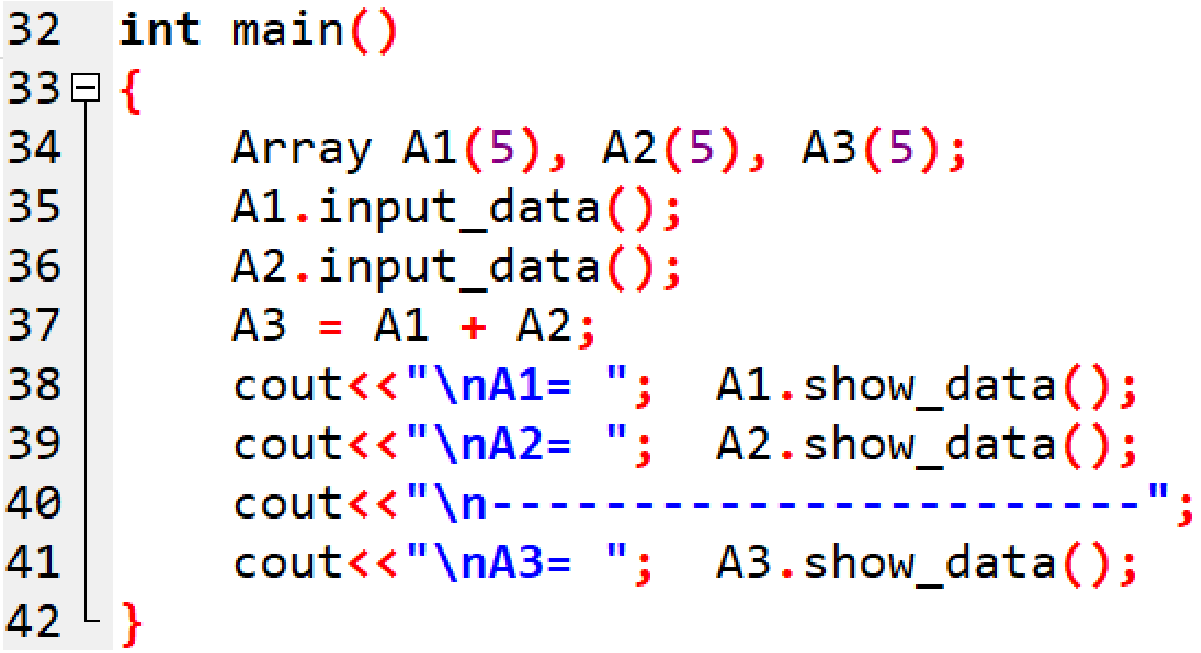
```
result = obj1.display();
```

```
void display()
{
   cout<<"Real="<<real;
   cout<<"Imaginary="<<imag;
}
```

Question: Define the operator function for the following

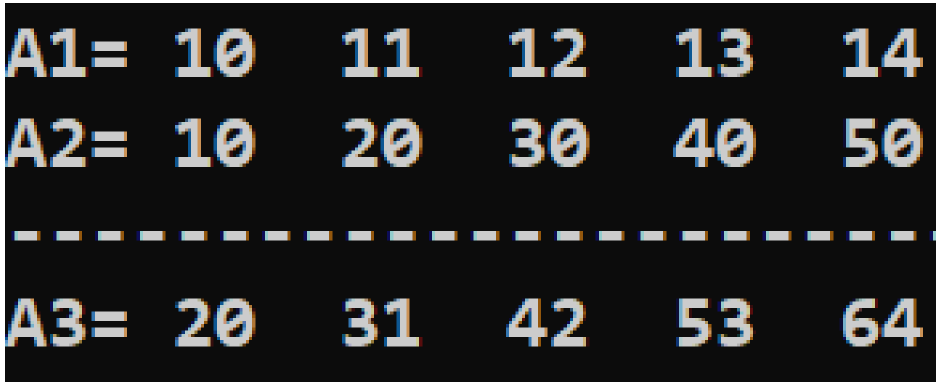


```
class Array
{
    int data[25], size;
public:
    Array() : size(0){}

    Array( int s ) : size(s) {}

    void input_data()
    {
        for( int i=0;i<size;i++)
            cin>>data[i];
    }
    void show_data();
    // operator function
};
```

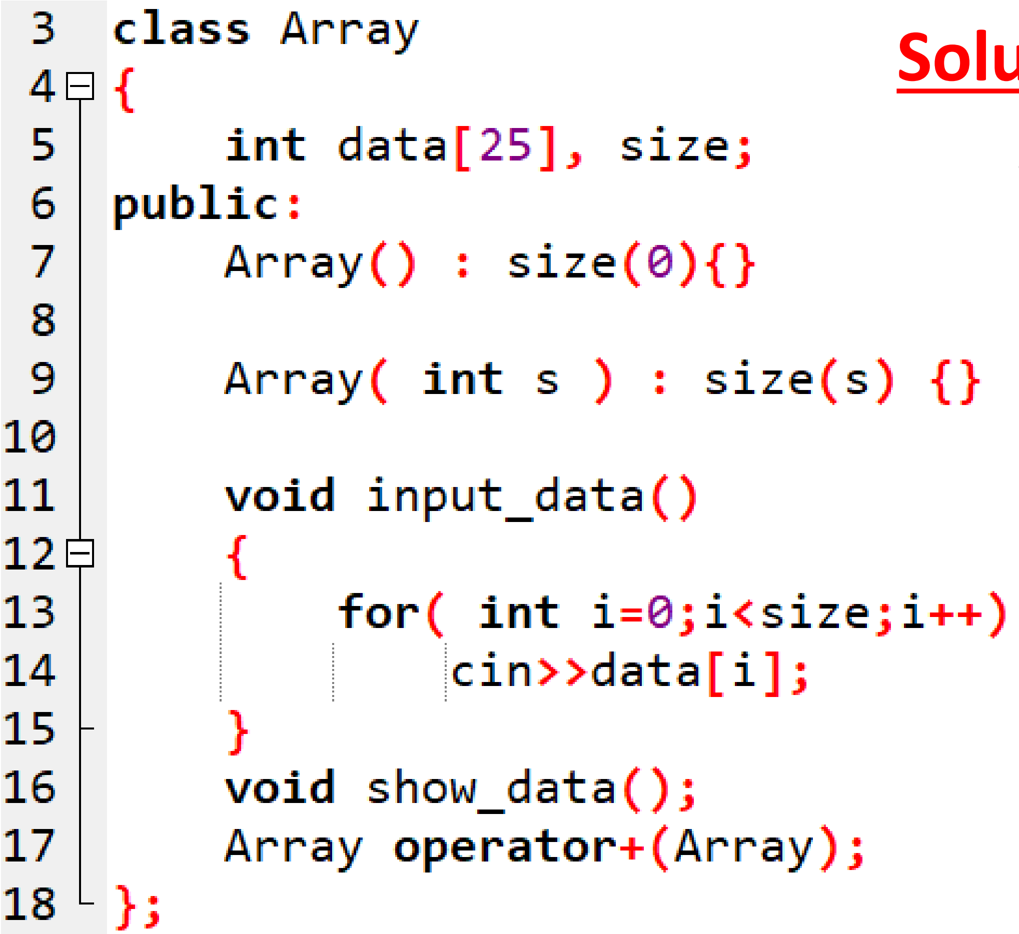


```
int main()
{
    Array A1(5), A2(5), A3(5);
    A1.input_data();
    A2.input_data();
    A3 = A1 + A2;
    cout<<"\nA1= ";  A1.show_data();
    cout<<"\nA2= ";  A2.show_data();
    cout<<"\n------------------------";
    cout<<"\nA3= ";  A3.show_data();
}
```

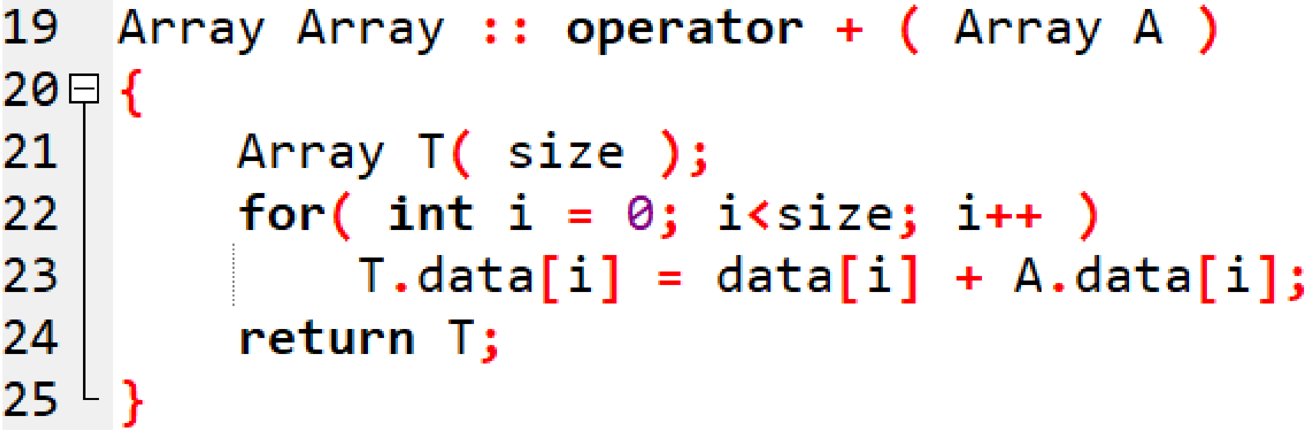
OUTPUT



```
A1= 10   11   12   13   14
A2= 10   20   30   40   50
------------------------
A3= 20   31   42   53   64
```

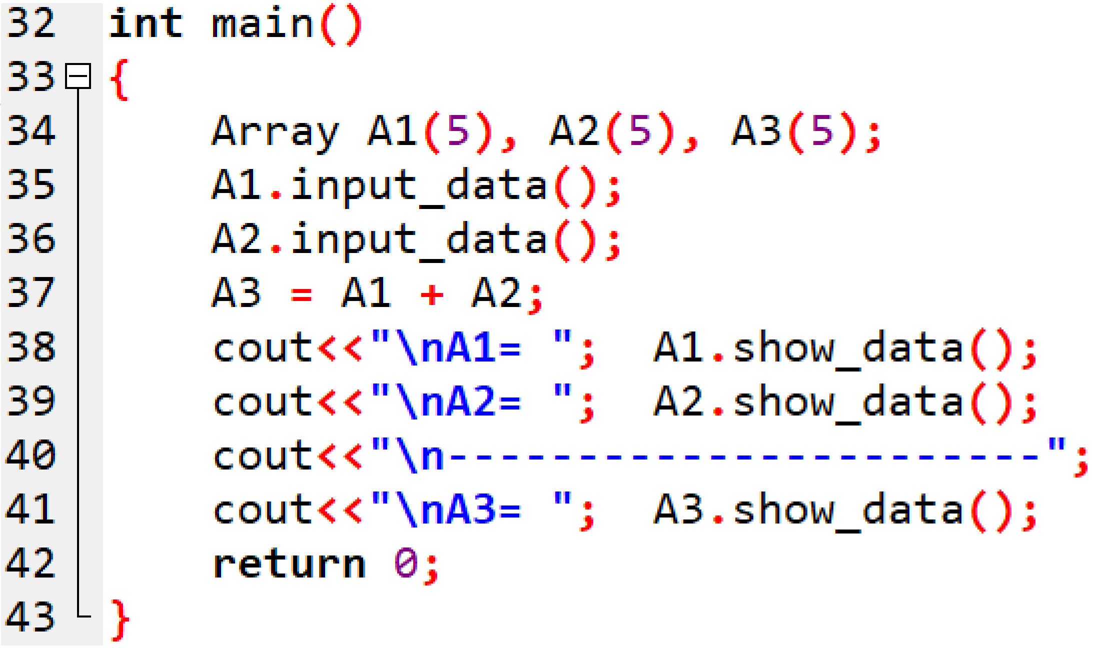


```cpp
class Array
{
    int data[25], size;
public:
    Array() : size(0){}

    Array( int s ) : size(s) {}

    void input_data()
    {
        for( int i=0;i<size;i++)
            cin>>data[i];
    }
    void show_data();
    Array operator+(Array);
};
```



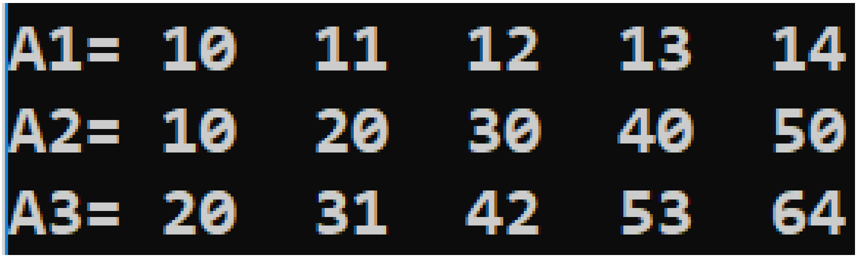
```cpp
Array Array :: operator + ( Array A )
{
    Array T( size );
    for( int i = 0; i<size; i++ )
        T.data[i] = data[i] + A.data[i];
    return T;
}
```

Solution:



```cpp
int main()
{
    Array A1(5), A2(5), A3(5);
    A1.input_data();
    A2.input_data();
    A3 = A1 + A2;
    cout<<"\nA1= ";  A1.show_data();
    cout<<"\nA2= ";  A2.show_data();
    cout<<"\n------------------------";
    cout<<"\nA3= ";  A3.show_data();
    return 0;
}
```

OUTPUT



```
A1= 10   11   12   13   14
A2= 10   20   30   40   50
A3= 20   31   42   53   64
```

Operator Overloading

- **Operator overloading** is compile time polymorphism.
- You can overload most of the built-in operators available in C++.

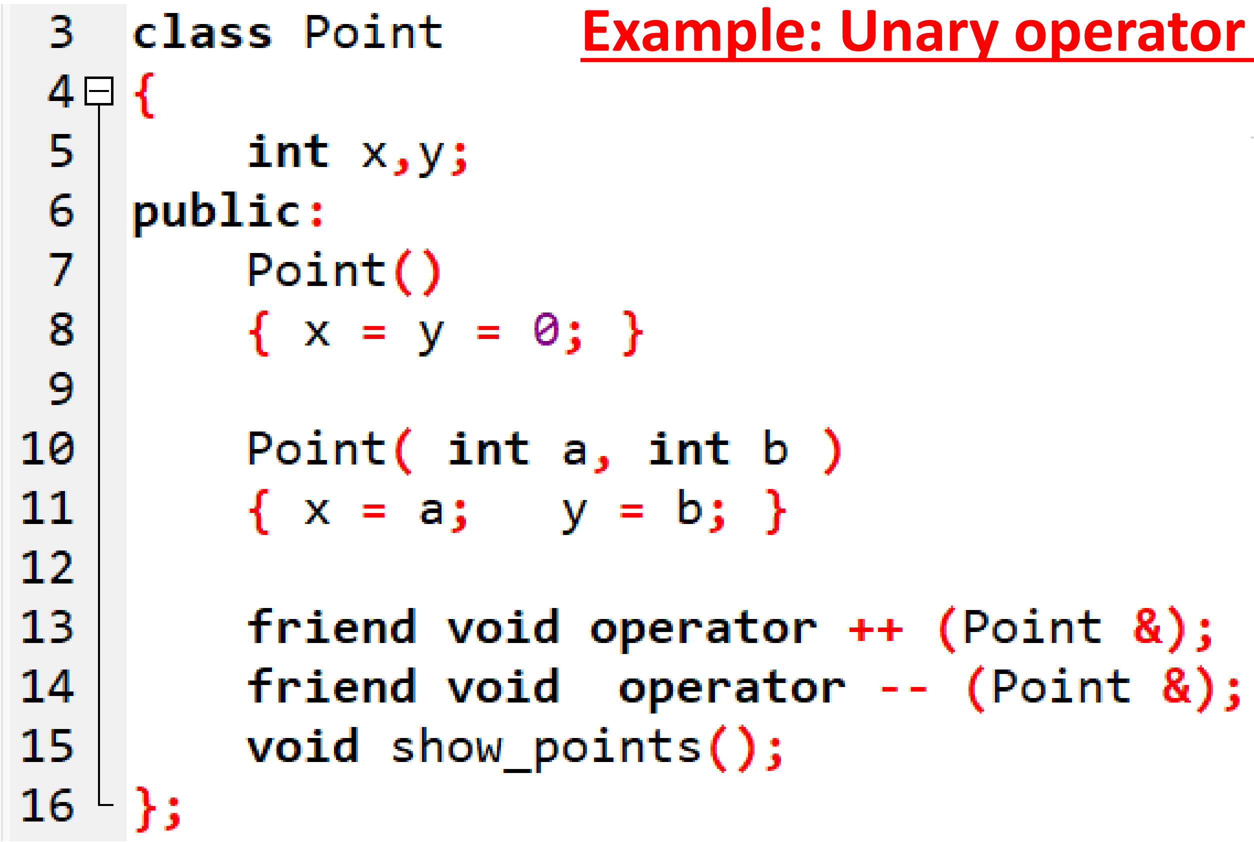
<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr>
<tr><td>&</td><td>|</td><td>~</td><td>!</td><td>,</td><td>=</td></tr>
<tr><td><</td><td>></td><td><=</td><td>>=</td><td>++</td><td>--</td></tr>
<tr><td><<</td><td>>></td><td>==</td><td>!=</td><td>&&</td><td>||</td></tr>
<tr><td>+=</td><td>-=</td><td>/=</td><td>%=</td><td>^=</td><td>&=</td></tr>
<tr><td>|=</td><td>*=</td><td><<=</td><td>>>=</td><td>[]</td><td>()</td></tr>
<tr><td>-></td><td>->*</td><td>new</td><td>new []</td><td>delete</td><td>delete []</td></tr>
</table>

Operator Overloading using Friend Function

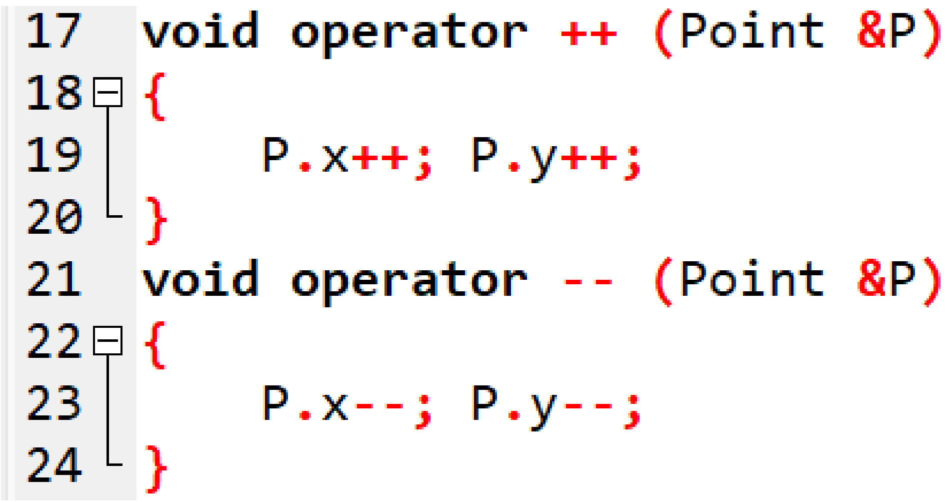
Operator overloading using friend function

```
class  Some_class
{

        . . . . . . . . . .

        friend    Ret_type    operator    op   (  parameters  );

};


Ret_type    operator    op   (parameters  )

{

        . . . . . . . . . .

}
```

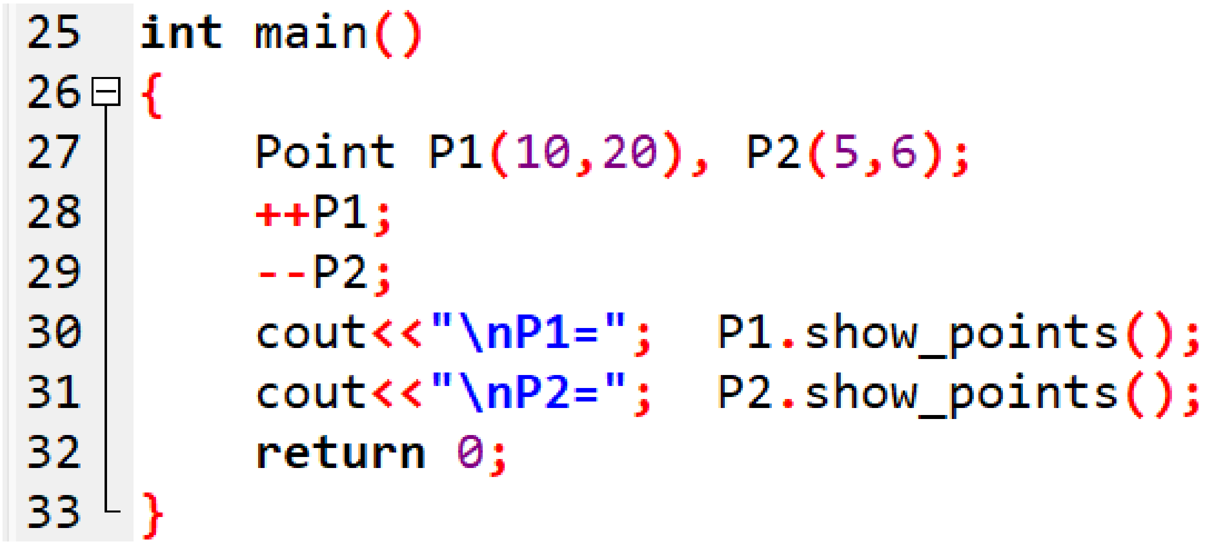
Example: Unary operator overloading using friend function



```cpp
class Point
{
    int x,y;
public:
    Point()
    { x = y = 0; }

    Point( int a, int b )
    { x = a;   y = b; }

    friend void operator ++ (Point &);
    friend void  operator -- (Point &);
    void show_points();
};
```

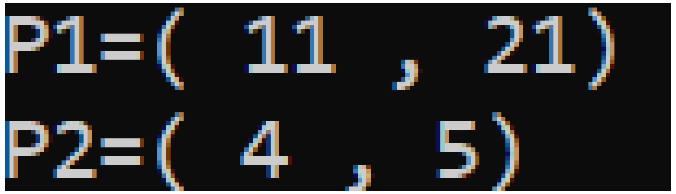


```cpp
void operator ++ (Point &P)
{
    P.x++; P.y++;
}
void operator -- (Point &P)
{
    P.x--; P.y--;
}
```



```cpp
int main()
{
    Point P1(10,20), P2(5,6);
    ++P1;
    --P2;
    cout<<"\nP1=";  P1.show_points();
    cout<<"\nP2=";  P2.show_points();
    return 0;
}
```

OUTPUT



```
P1=( 11 , 21)
P2=( 4 , 5)
```

Invoke Friend Function in operator overloading

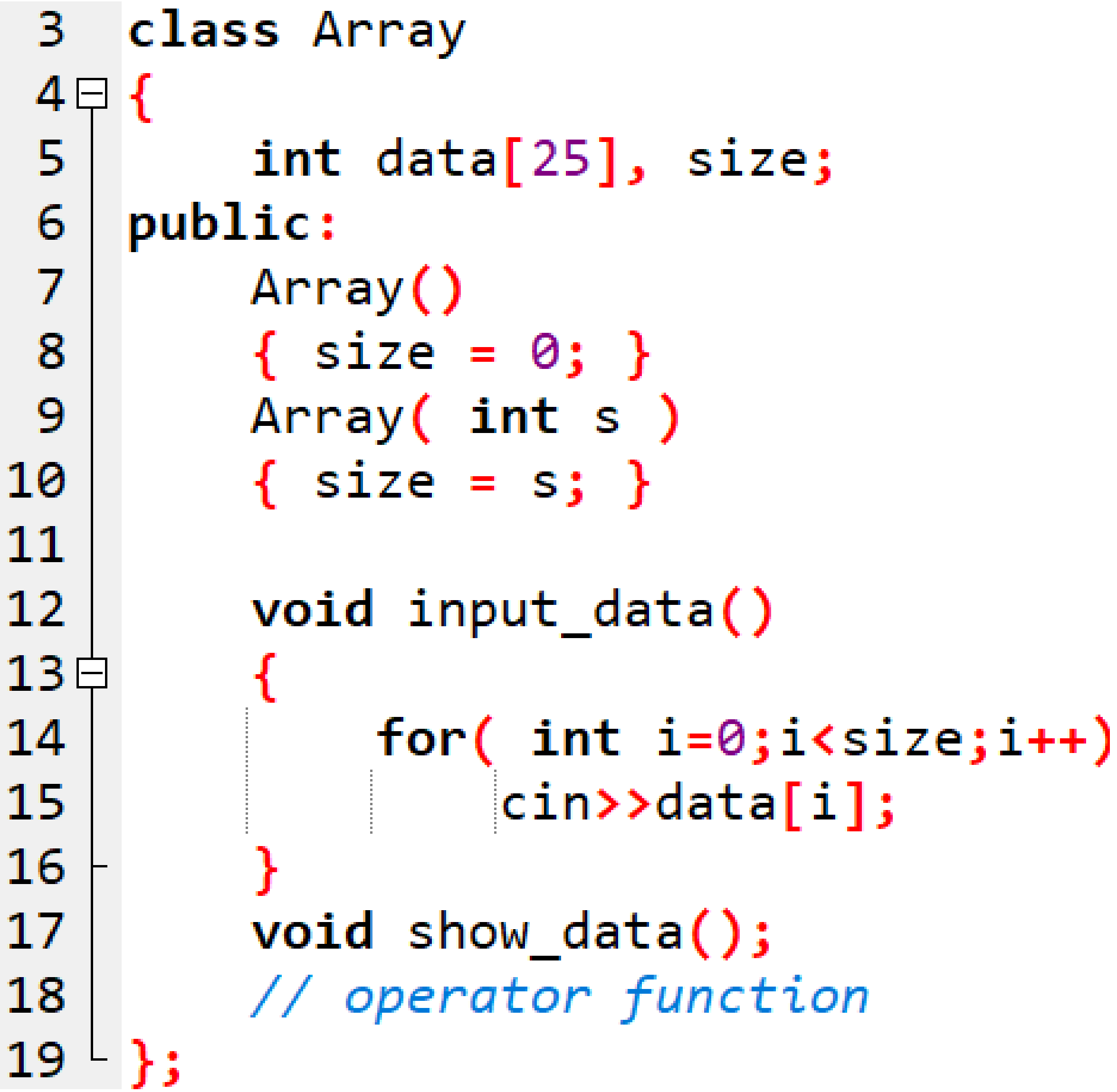
```
result = operator symbol (obj1,obj2);//function notation
```

```
result = obj1 symbol obj2;           //operator notation
```

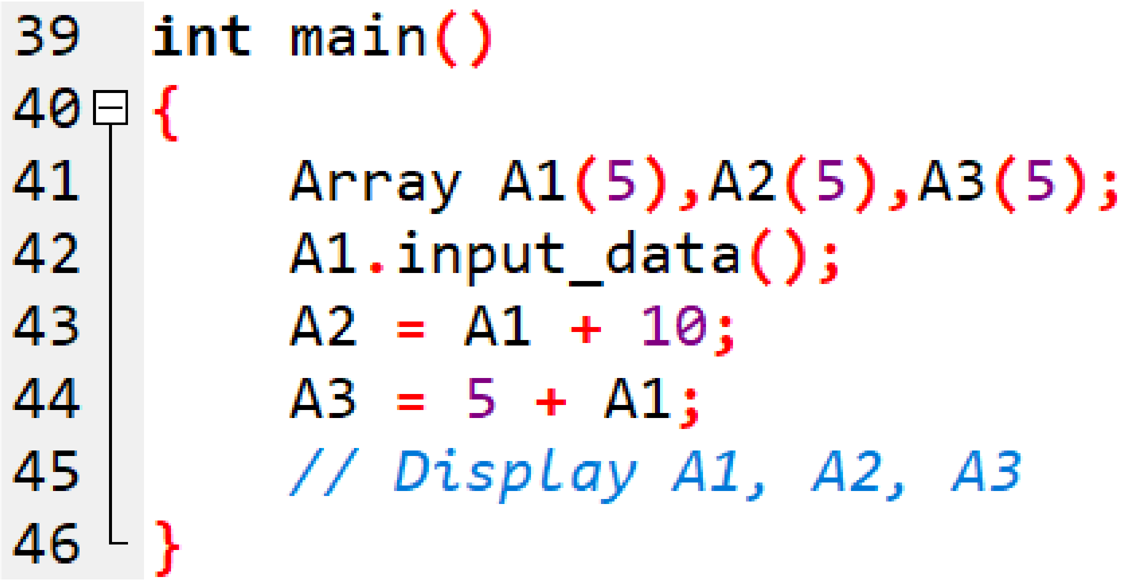
```
friend complex operator +(complex c1,complex c2)
{
  complex tmp;
  tmp.r=c1.r+c2.r;
  tmp.i=c1.i+c2.i;
  return tmp;
}
```

```
int main()
{
  complex c1(4,7),c2(5,8);
  complex c3;
  c3 = c1 + c2;
  c3 = operator +(c1,c2);
}
```

Question: Define the operator function for the following

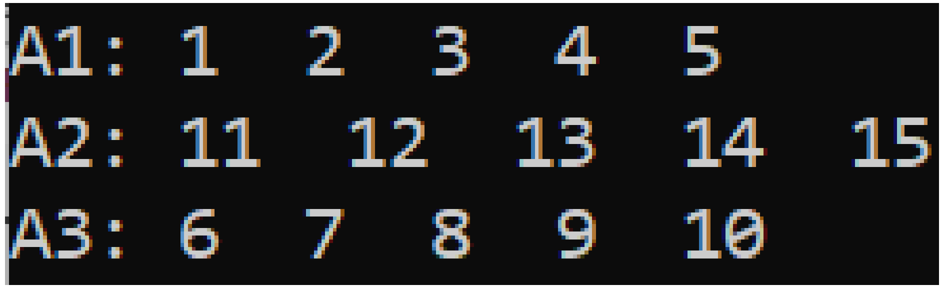


```cpp
class Array
{
    int data[25], size;
public:
    Array()
    { size = 0; }
    Array( int s )
    { size = s; }

    void input_data()
    {
        for( int i=0;i<size;i++)
            cin>>data[i];
    }
    void show_data();
    // operator function
};
```

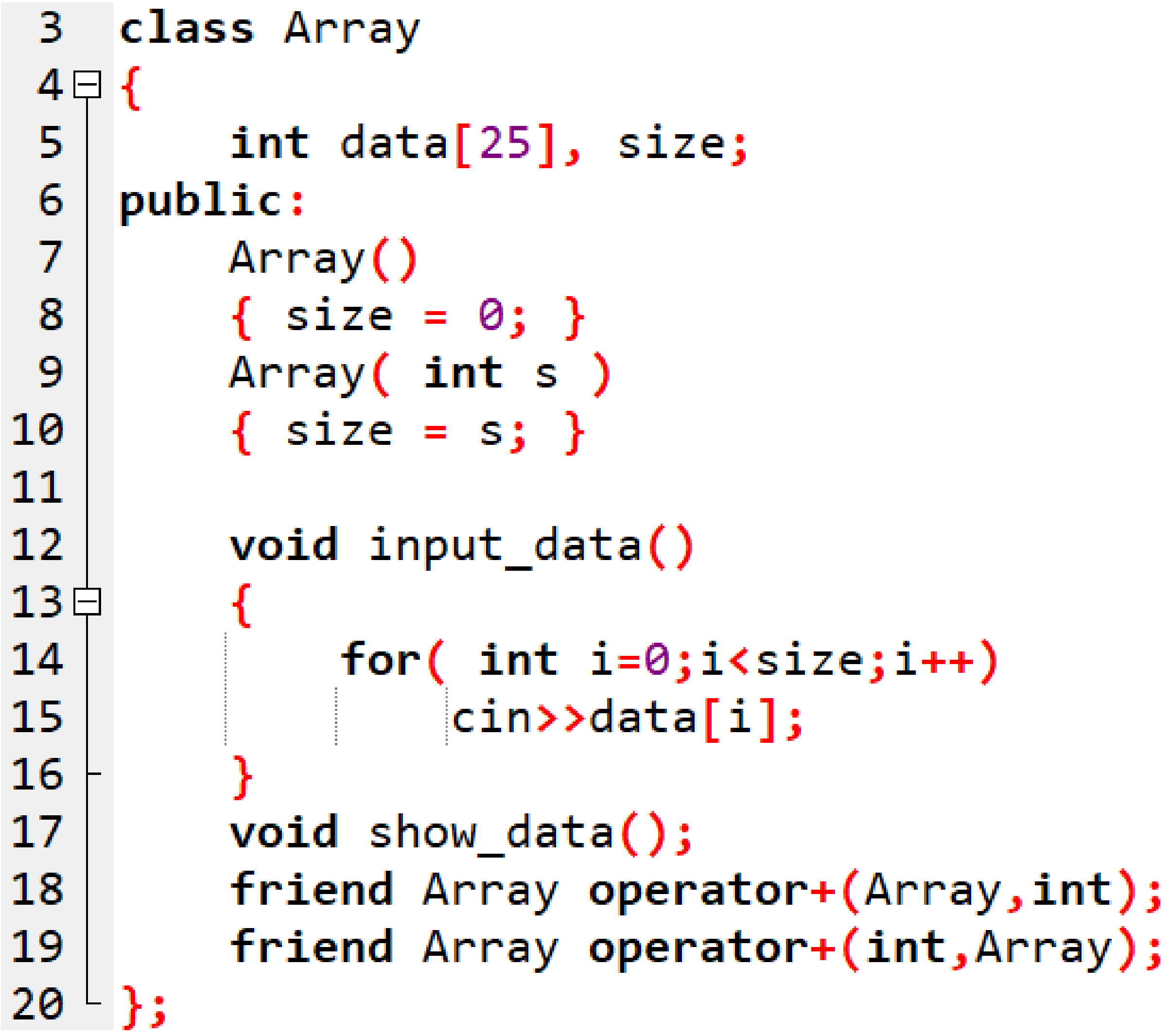


```cpp
int main()
{
    Array A1(5),A2(5),A3(5);
    A1.input_data();
    A2 = A1 + 10;
    A3 = 5 + A1;
    // Display A1, A2, A3
}
```

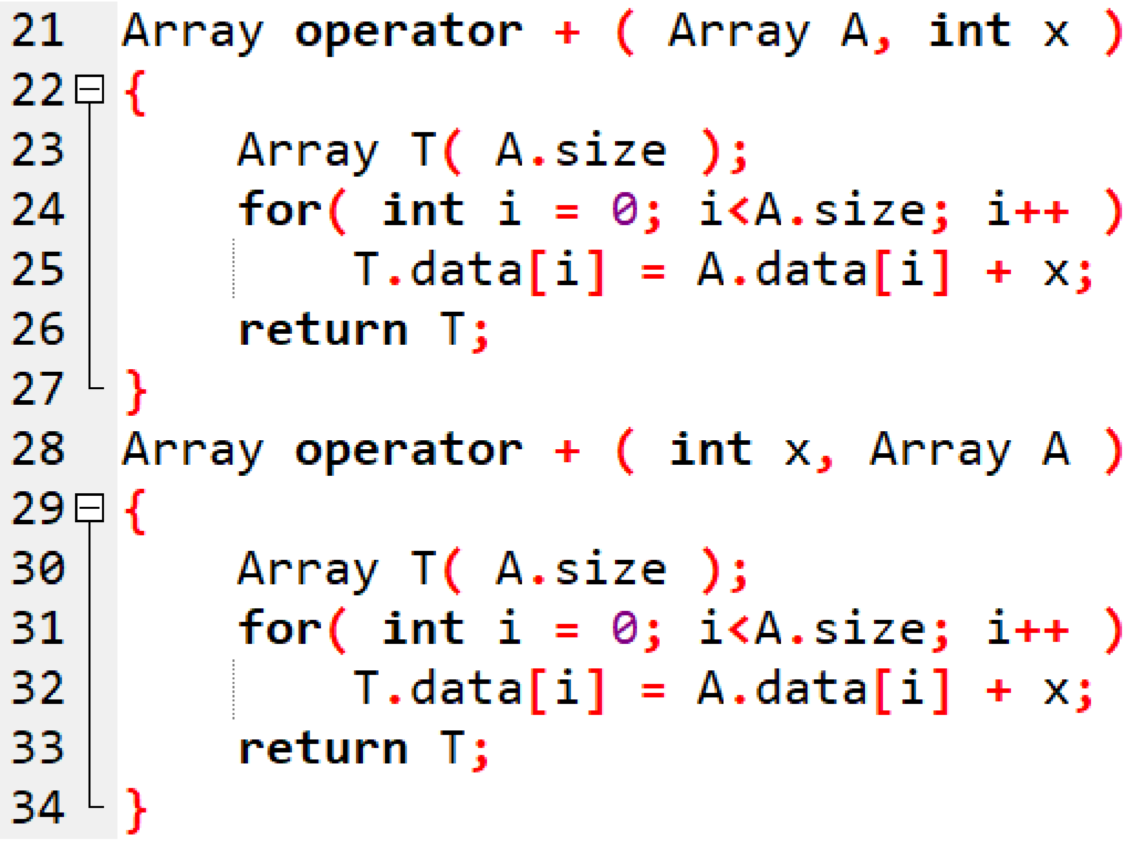
OUTPUT



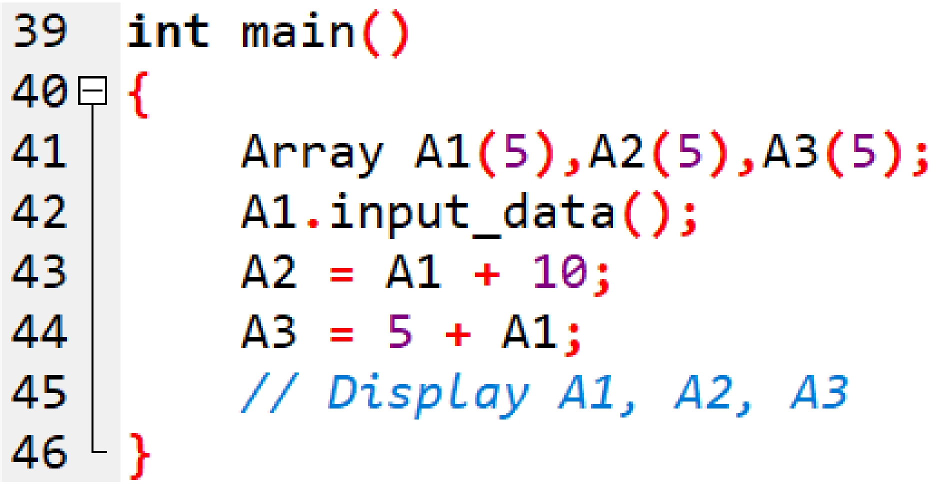
```
A1: 1   2   3   4   5
A2: 11   12   13   14   15
A3: 6   7   8   9   10
```



```cpp
class Array
{
    int data[25], size;
public:
    Array()
    { size = 0; }
    Array( int s )
    { size = s; }

    void input_data()
    {
        for( int i=0;i<size;i++)
            cin>>data[i];
    }
    void show_data();
    friend Array operator+(Array,int);
    friend Array operator+(int,Array);
};
```



```cpp
Array operator + ( Array A, int x )
{
    Array T( A.size );
    for( int i = 0; i<A.size; i++ )
        T.data[i] = A.data[i] + x;
    return T;
}
Array operator + ( int x, Array A )
{
    Array T( A.size );
    for( int i = 0; i<A.size; i++ )
        T.data[i] = A.data[i] + x;
    return T;
}
```

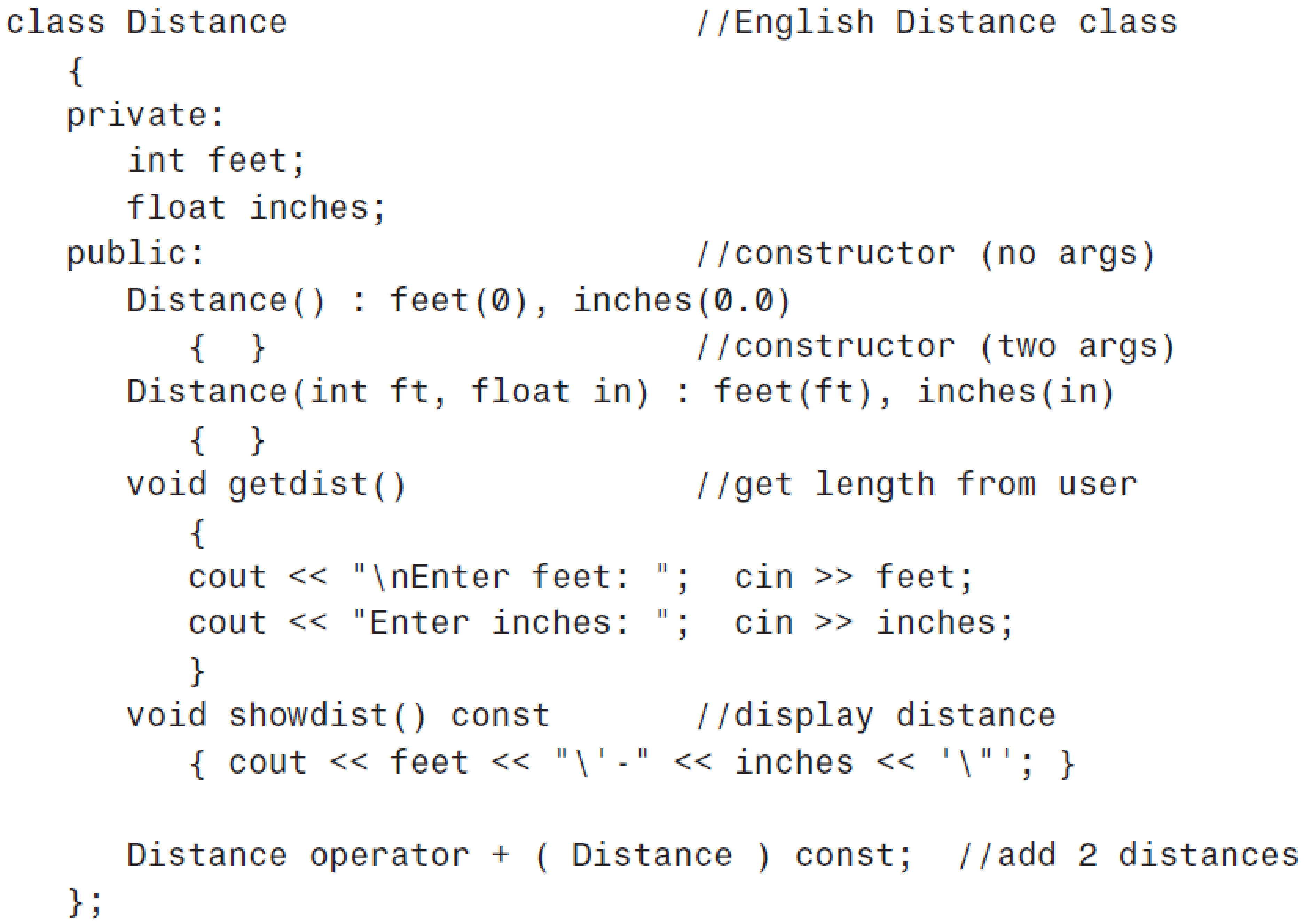


```cpp
int main()
{
    Array A1(5),A2(5),A3(5);
    A1.input_data();
    A2 = A1 + 10;
    A3 = 5 + A1;
    // Display A1, A2, A3
}
```

Overloading Binary operator ==

```
class complex{
  int r,i;
  public:
  complex(){
    r=i=0;}
  complex(int x,int y){
    r=x;
    i=y;}
  void display(){
   cout<<"\nreal="<<r<<endl;
   cout<<"imag="<<i<<endl;}
   int operator==(complex);
};
int complex::operator ==(complex c){
  if(r==c.r && i==c.i)
    return 1;
  else
    return 0;}
```

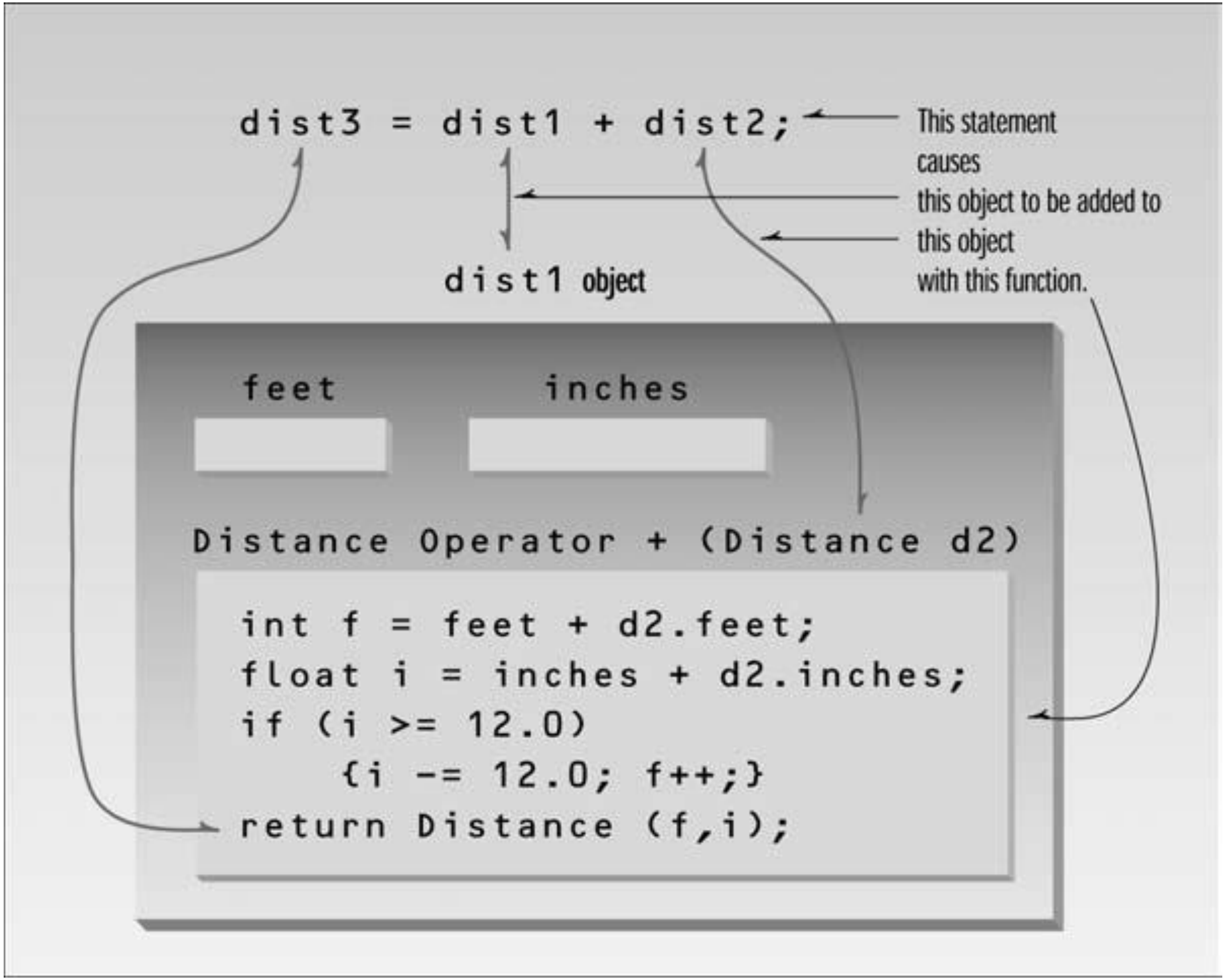
```
int main()
{
  complex c1(5,3),c2(5,3);
  if(c1==c2)
    cout<<"objects are equal";
  else
    cout<<"objects are not equal";
    return 0;
      }
```



```
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                     //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      Distance operator + ( Distance ) const;  //add 2 distances
   };
```

```
Distance Distance::operator + (Distance d2) const  //return sum
   {
   int f = feet + d2.feet;        //add the feet
   float i = inches + d2.inches;  //add the inches
   if(i >= 12.0)                  //if total exceeds 12.0,
      {                           //then decrease inches
      i -= 12.0;                  //by 12.0 and
      f++;                        //increase feet by 1
      }                           //return a temporary Distance
   return Distance(f,i);          //initialized to sum
   }

 int main()
    {
    Distance dist1, dist3, dist4;   //define distances
    dist1.getdist();                //get dist1 from user

    Distance dist2(11, 6.25);       //define, initialize dist2

    dist3 = dist1 + dist2;          //single '+' operator

    dist4 = dist1 + dist2 + dist3;  //multiple '+' operators
```

```
class Distance
   {
   private:
      int feet;
      float inches;

     ...........
```

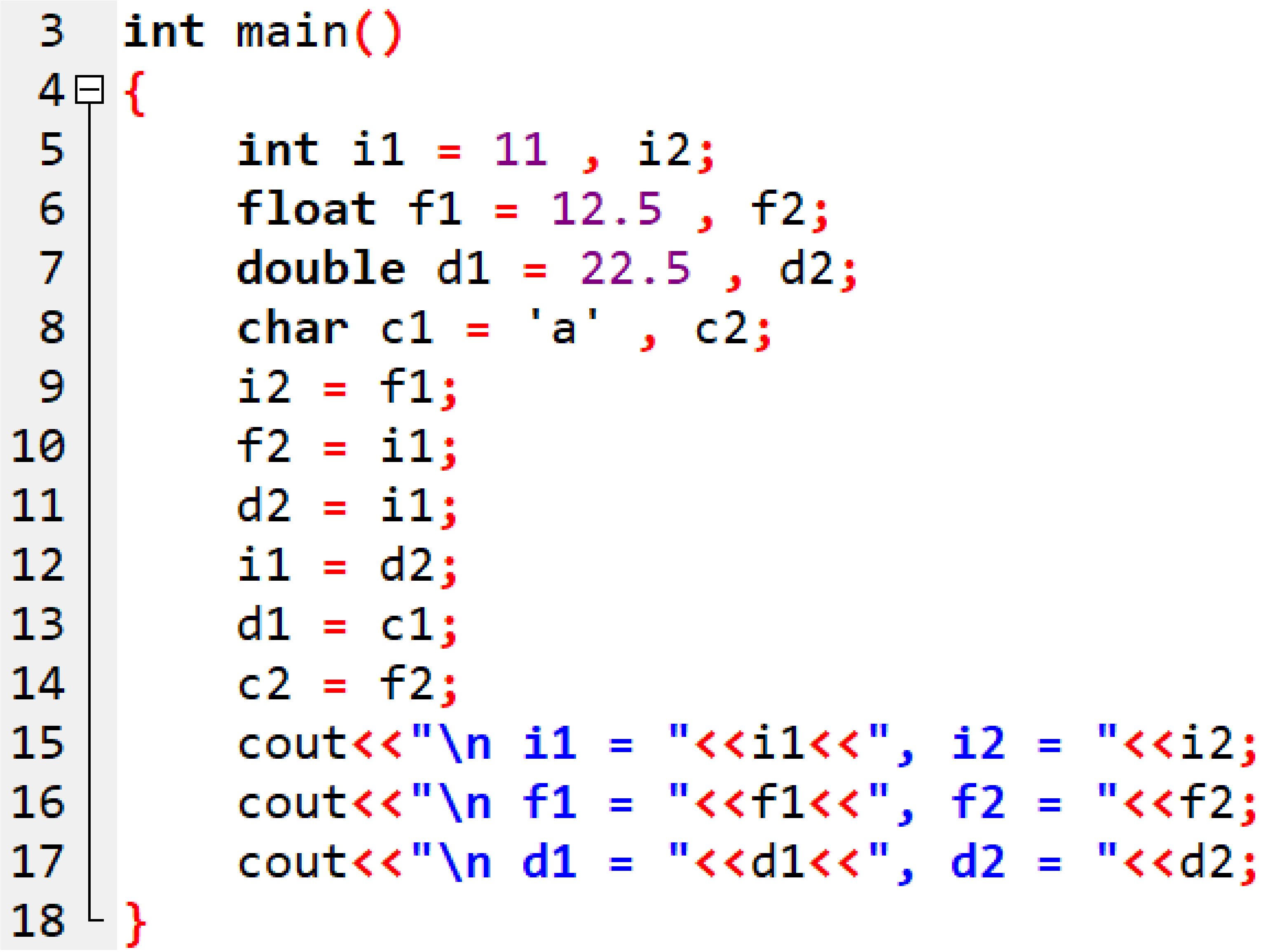


```
dist3 = dist1 + dist2;
```

This statement causes this object to be added to this object with this function.

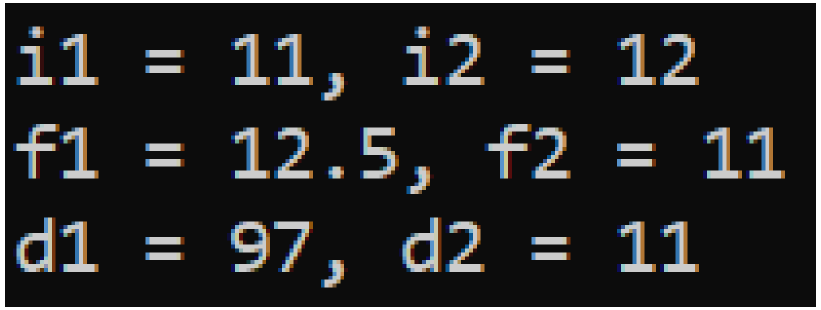
```
Distance Operator + (Distance d2)
  int f = feet + d2.feet;
  float i = inches + d2.inches;
  if (i >= 12.0)
      {i -= 12.0; f++;}
  return Distance (f,i);
```

DATA CONVERSION



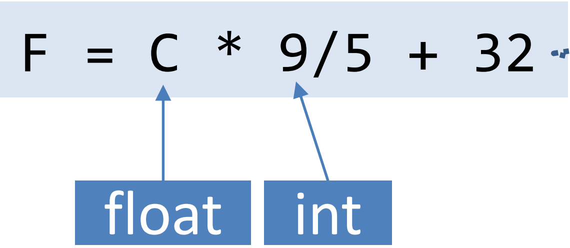
```
int main()
{
    int i1 = 11 , i2;
    float f1 = 12.5 , f2;
    double d1 = 22.5 , d2;
    char c1 = 'a' , c2;
    i2 = f1;
    f2 = i1;
    d2 = i1;
    i1 = d2;
    d1 = c1;
    c2 = f2;
    cout<<"\n i1 = "<<i1<<", i2 = "<<i2;
    cout<<"\n f1 = "<<f1<<", f2 = "<<f2;
    cout<<"\n d1 = "<<d1<<", d2 = "<<d2;
}
```

OUTPUT:



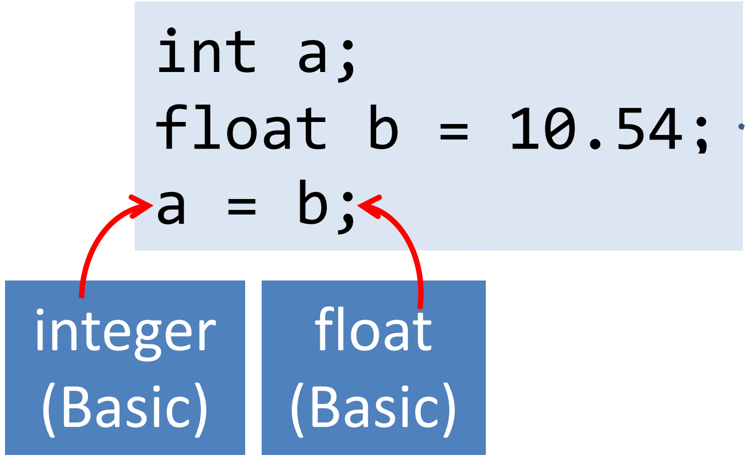
```
i1 = 11, i2 = 12
f1 = 12.5, f2 = 11
d1 = 97, d2 = 11
```

Type Conversion



```
F = C * 9/5 + 32
```

If different data types are mixed in expression, C++ applies automatic type conversion as per certain rules.



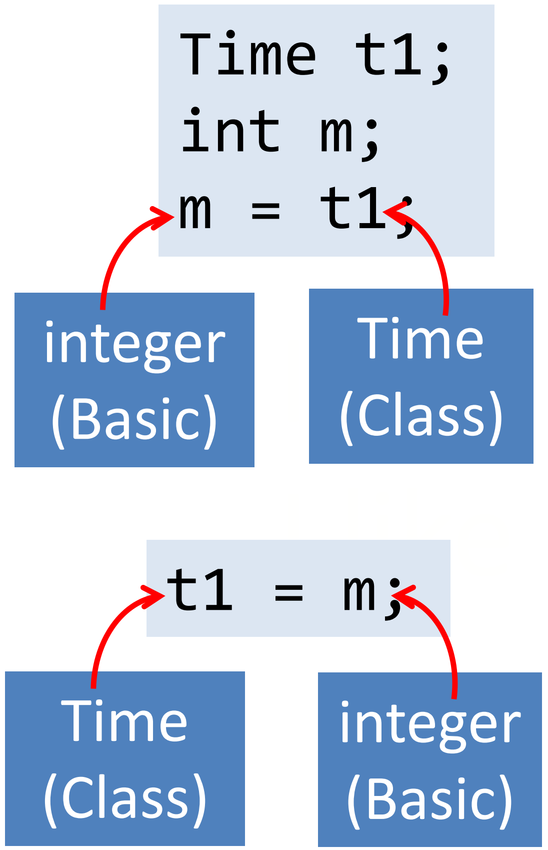
```
int a;
float b = 10.54;
a = b;
```

a = 10;

- float is converted to integer automatically by complier.
- basic to basic type conversion.

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

Type Conversion



```
Time t1;
int m;
m = t1;
```

```
t1 = m;
```

- class type will not be converted to basic type OR basic type will not be converted class type automatically.

Type Conversion

- C++ provides mechanism to perform automatic type conversion if all variable are of **basic type**.

- Three types of situation arise in user defined data type conversion.

1. Basic type to Class type (Using Constructors)
2. Class type to Basic type (Using Casting Operator Function)
3. Class type to Class type (Using Constructors & Casting Operator Functions)

DATA CONVERSION

- The general format of a type conversion function is:

Basic Type → *User_defined type* : one-arg constructor

User_defined type → *Basic Type* : Operator function

↓

```
operator   type ( )
{
        // Conversion steps
        return value;
}
```

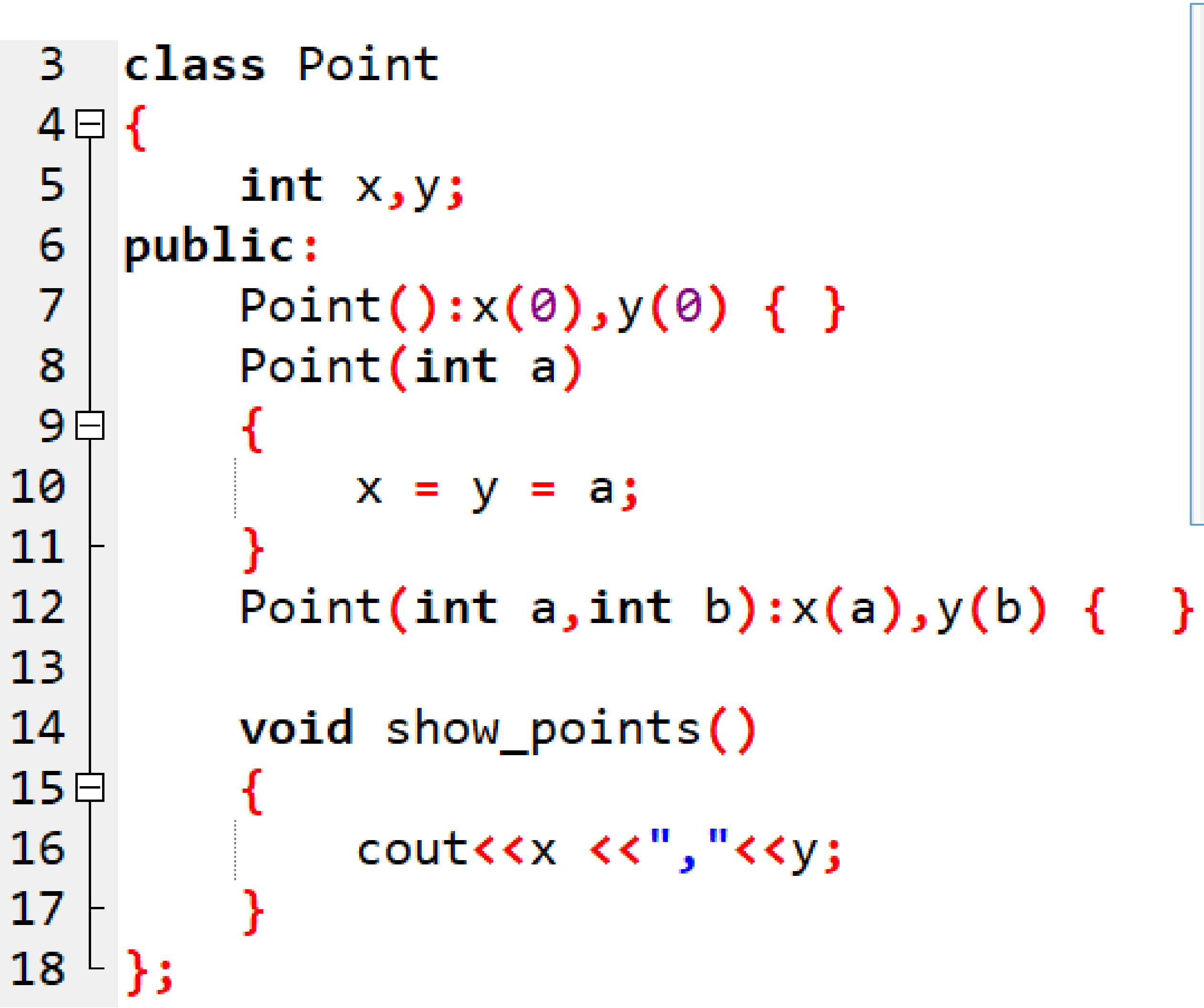
(1) Basic to class type conversion

- Basic to class type can be achieved **using constructor**.

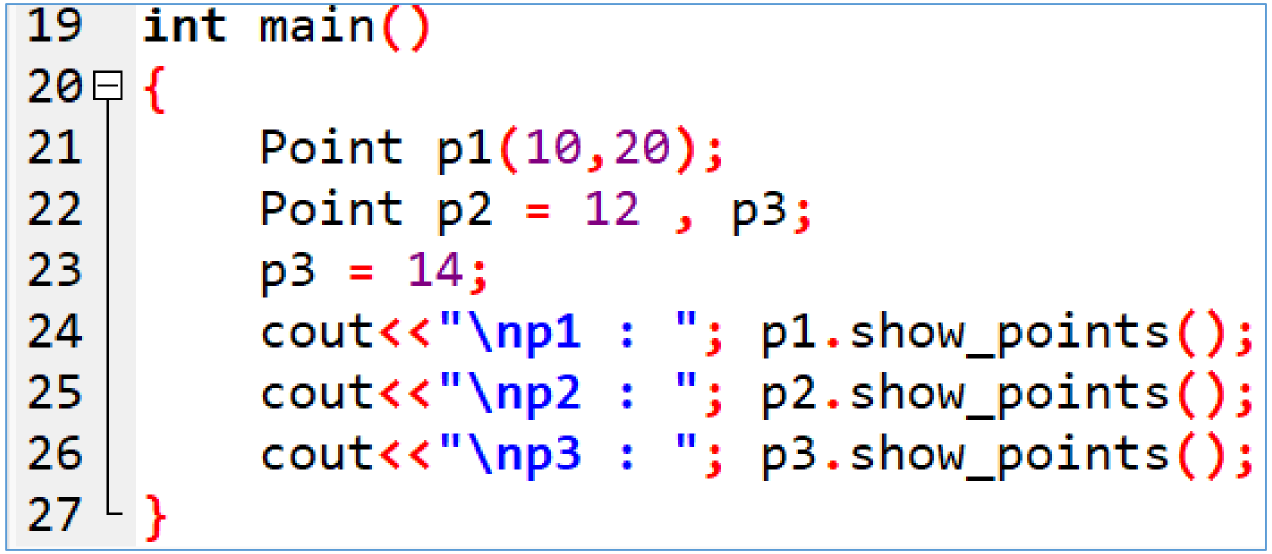
```
class sample
{
  int a;
  public:
  sample(){}
  sample(int x){
    a=x;
  }
  void disp(){
    cout<<"The value of a="<<a;
  }
};
```

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

Example-: Basic → User_defined conversion

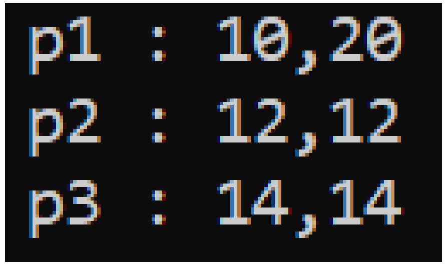


```
class Point
{
    int x,y;
public:
    Point():x(0),y(0) { }
    Point(int a)
    {
        x = y = a;
    }
    Point(int a,int b):x(a),y(b) {  }

    void show_points()
    {
        cout<<x <<","<<y;
    }
};
```



```
int main()
{
    Point p1(10,20);
    Point p2 = 12 , p3;
    p3 = 14;
    cout<<"\np1 : "; p1.show_points();
    cout<<"\np2 : "; p2.show_points();
    cout<<"\np3 : "; p3.show_points();
}
```

OUTPUT:



```
p1 : 10,20
p2 : 12,12
p3 : 14,14
```

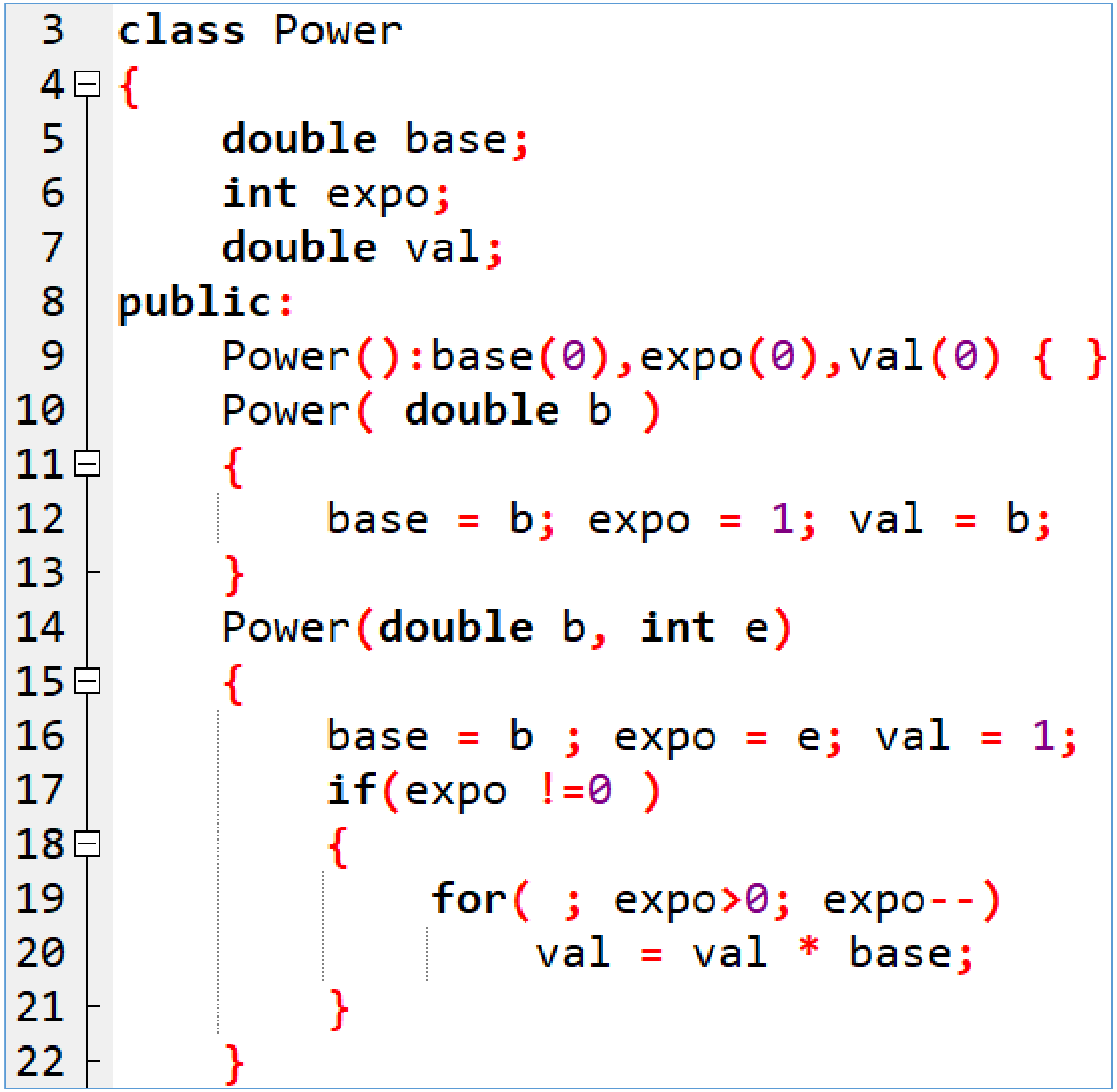
(2) Class to basic type conversion

- The Class type to Basic type conversion is done **using casting operator function**.
- The casting operator function should satisfy the following conditions.
    1. It must be a class member.
    2. It must not mention a return type.
    3. It must not have any arguments.

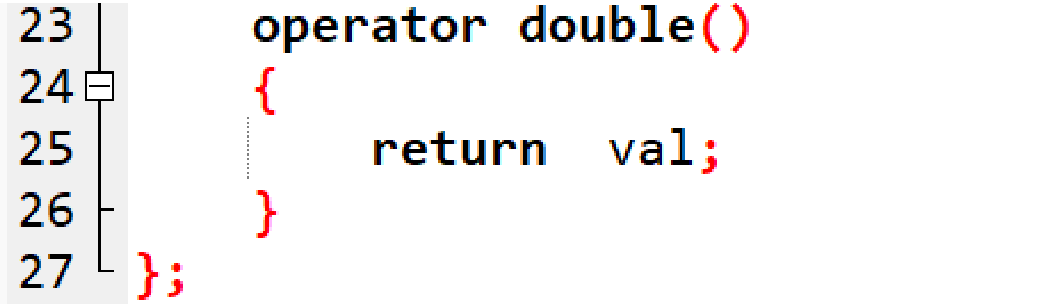
Syntax:

```
operator destinationtype()
{
    ....
    return
}
```

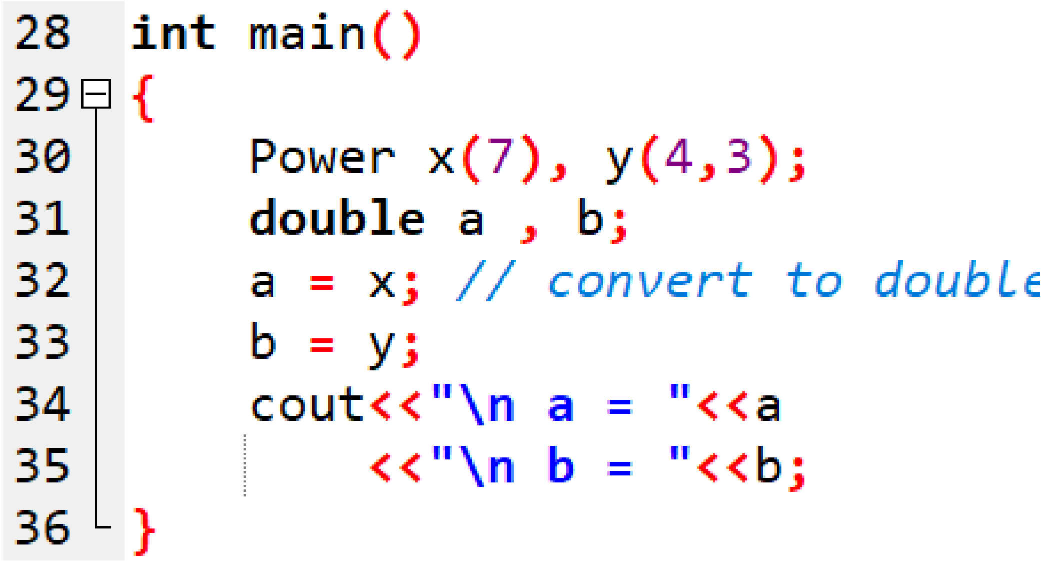
Example: UserDefined → Basic type



```cpp
class Power
{
    double base;
    int expo;
    double val;
public:
    Power():base(0),expo(0),val(0) { }
    Power( double b )
    {
        base = b; expo = 1; val = b;
    }
    Power(double b, int e)
    {
        base = b ; expo = e; val = 1;
        if(expo !=0 )
        {
            for( ; expo>0; expo--)
                val = val * base;
        }
    }
```

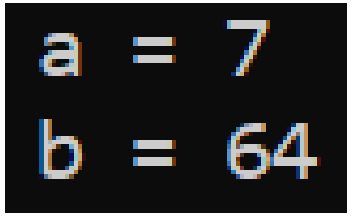


```cpp
    operator double()
    {
        return  val;
    }
};
```



```cpp
int main()
{
    Power x(7), y(4,3);
    double a , b;
    a = x; // convert to double
    b = y;
    cout<<"\n a = "<<a
        <<"\n b = "<<b;
}
```

OUTPUT:



```
a = 7
b = 64
```

Questions:

1. To convert from a user-defined class to a basic type, you would most likely use:

   a. built-in conversion routine.
   b. one-argument constructor.
   c. a conversion operator function that’s a member of the class.

2. To convert from a basic type to a user-defined class, you would most likely use:

   a. built-in conversion routine.
   b. one-argument constructor.
   c. a conversion operator function that’s a member of the class.

Questions:

1. To convert from a user-defined class to a basic type, you would most likely use:

   a. built-in conversion routine.
   b. one-argument constructor.
   c. **a conversion operator function that’s a member of the class.**

2. To convert from a basic type to a user-defined class, you would most likely use:

   a. built-in conversion routine.
   b. **one-argument constructor.**
   c. a conversion operator function that’s a member of the class.

Conversions Between Objects of Different Classes:

```
class Class_A
{
    // members
};
class Class_B
{
    // members
};
```

```
Class_A  obj_A;
Class_B  obj_B;
   . . .
obj_A = obj_B;
```

Destination Object

Source Object

(3) Class type to Class type

- It can be achieved by two ways

  1. **Using constructor**
  2. **Using conversion (casting) operator function**

- Two types of situation arise in user defined data type conversion.

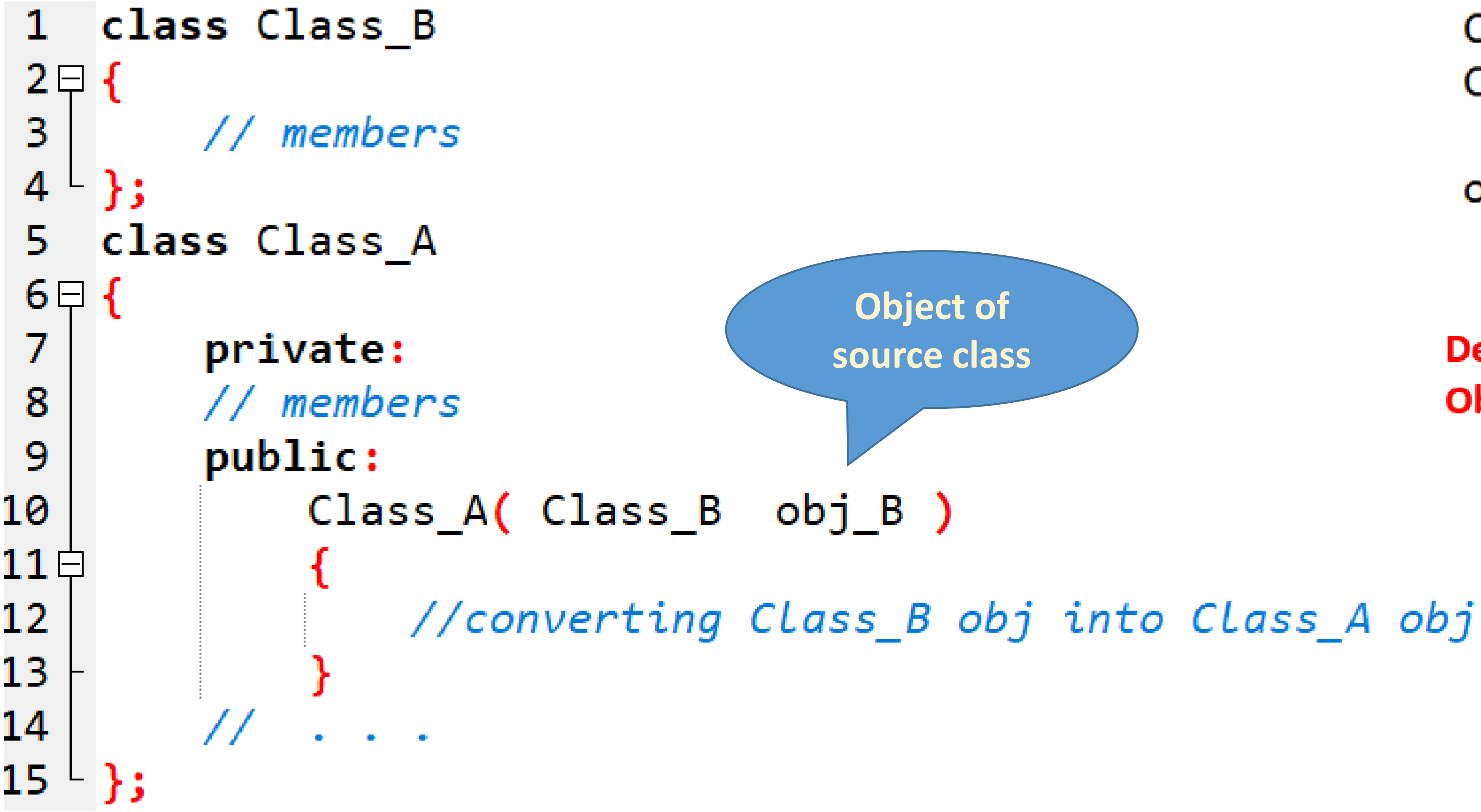
  1. **Routine in Destination Class** ***(Using Constructors)***
  2. **Routine in Source Class** ***(Using Conversion/Casting Operator Functions)***

- ***How do do we know?***
  - ***LHS = RHS***

  (Destination) = (Source)

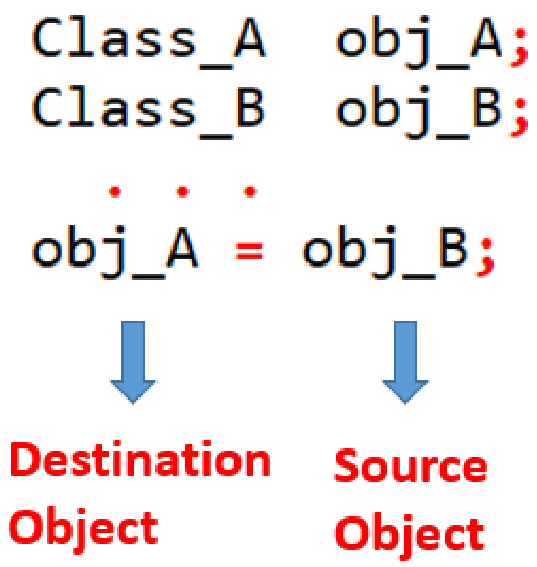
- ***Objective of the routines is the same***
  - ***To convert Fahrenheit to Centigrade***
  - ***To convert Kilometers to Miles/yards***
  - ***To convert Time24 to Time12***

Solution-1 →Conversion routine in destination class: constructor function

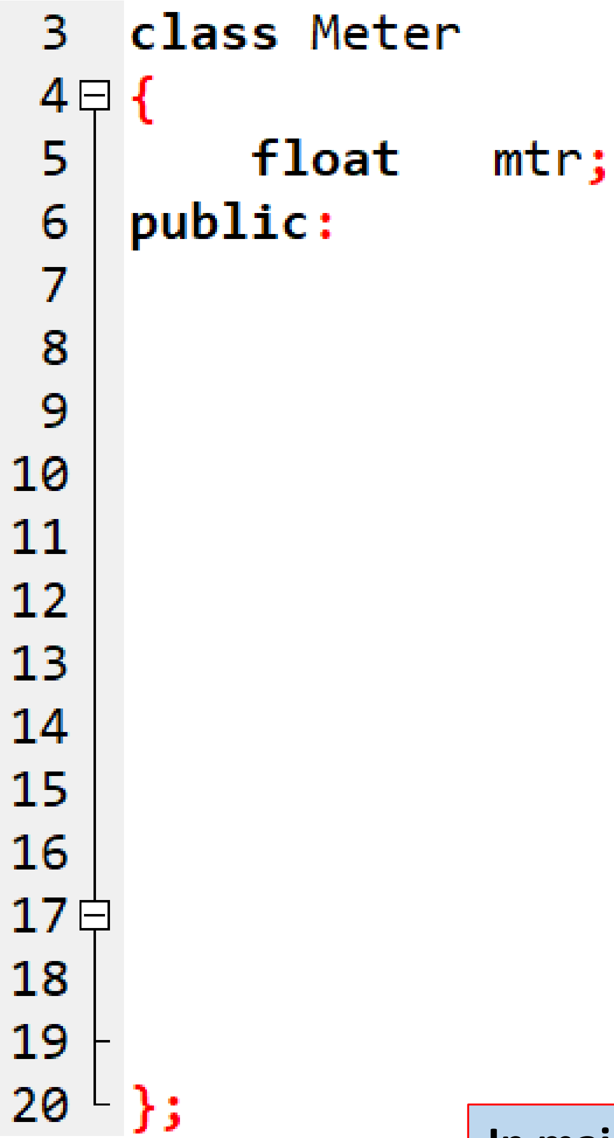


```
class Class_B
{
    // members
};
class Class_A
{
    private:
    // members
    public:
        Class_A( Class_B  obj_B )
        {
            //converting Class_B obj into Class_A obj
        }
    //  . . .
};
```

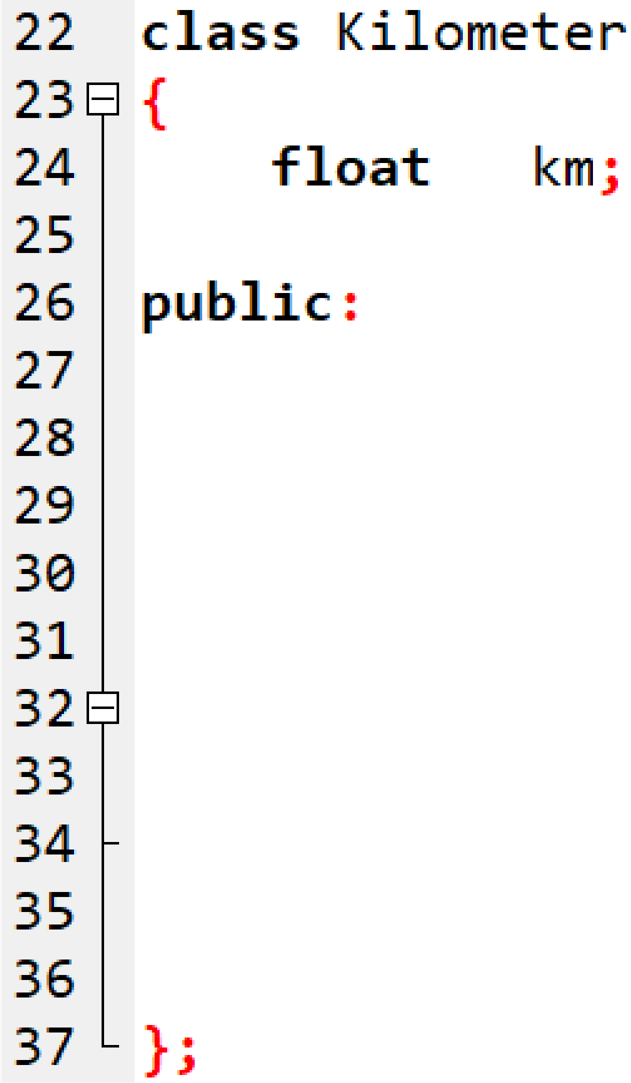
Object of source class



```
Class_A  obj_A;
Class_B  obj_B;
  . . .
obj_A = obj_B;
```



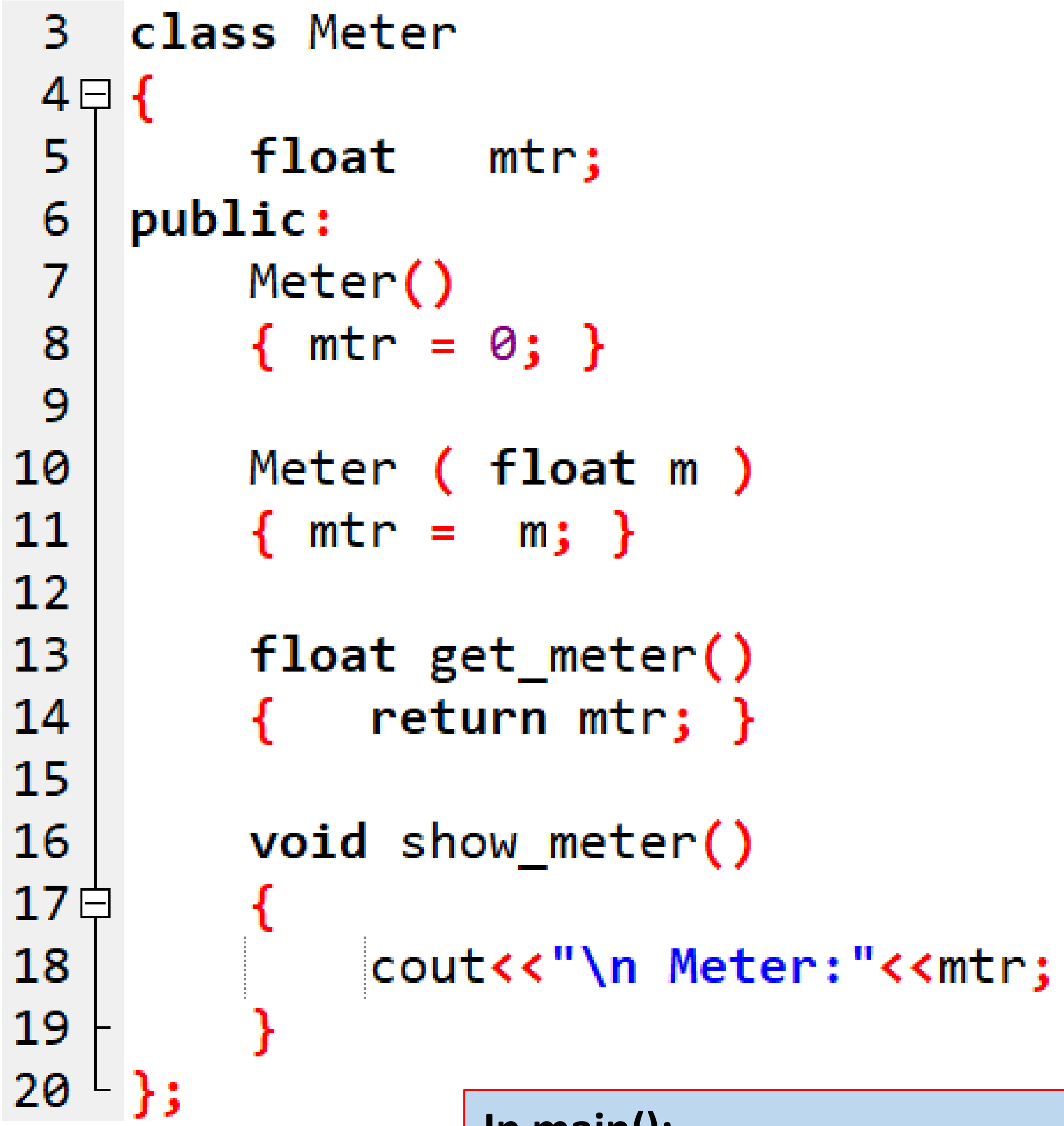
```
class Meter
{
      float    mtr;
public:













};
```



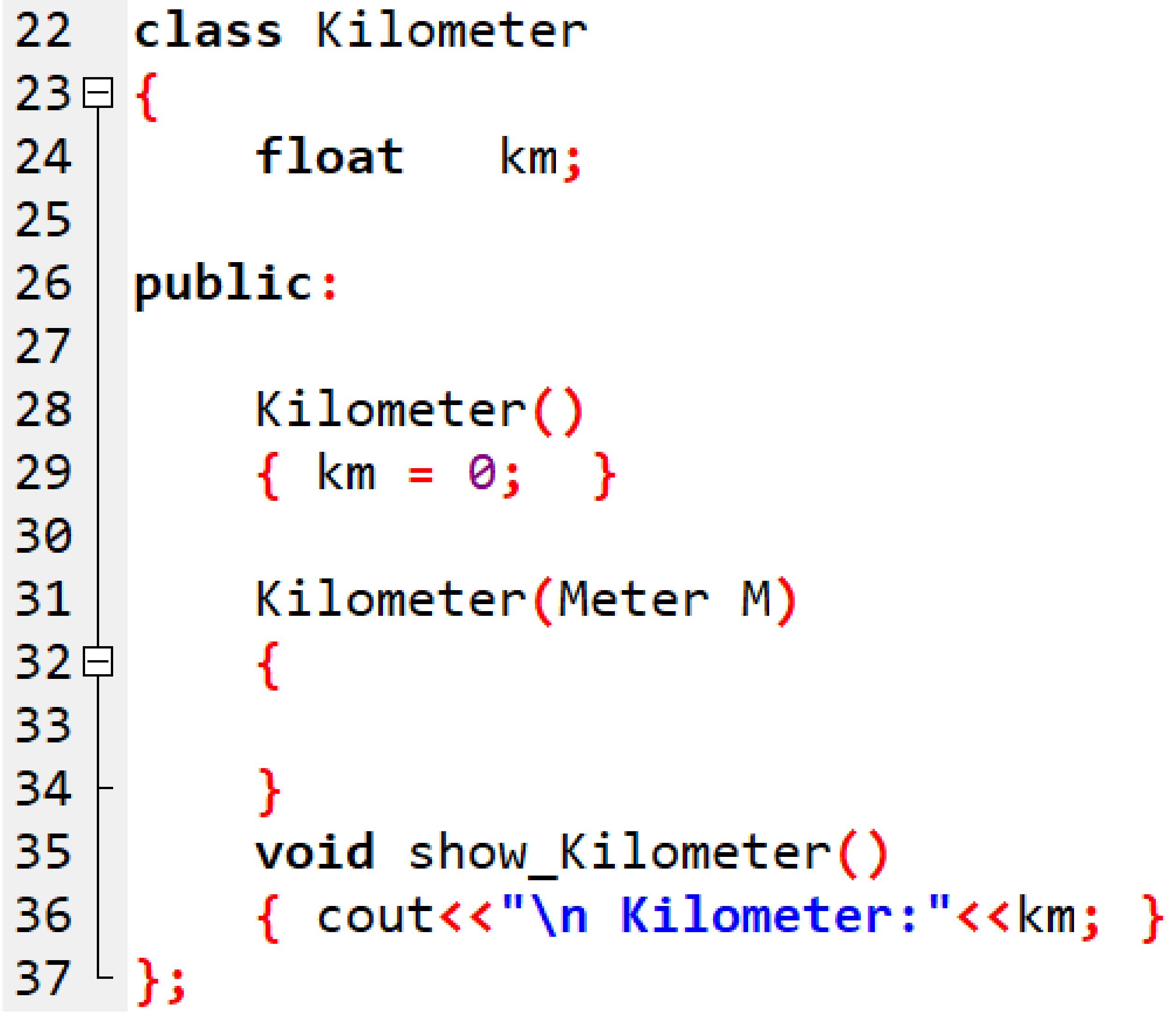
```
class Kilometer
{
      float    km;

public:










};
```

<u>In main():</u>

```
METER      M(1200);
KILOMETER   K  =  M;
```



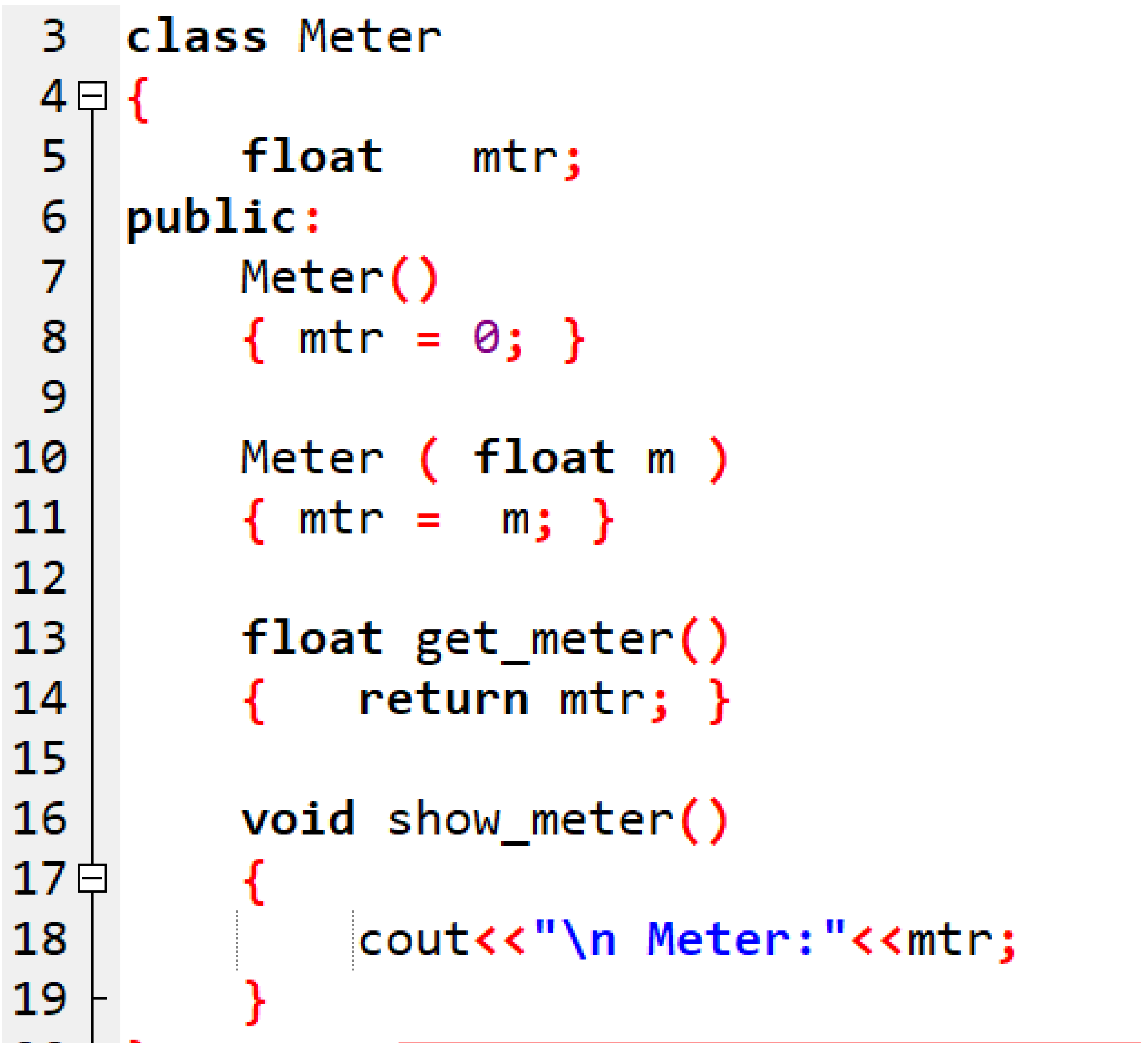
```
class Meter
{
     float   mtr;
public:
     Meter()
     { mtr = 0; }

     Meter ( float m )
     { mtr =  m; }

     float get_meter()
     {   return mtr; }

     void show_meter()
     {
          cout<<"\n Meter:"<<mtr;
     }
};
```



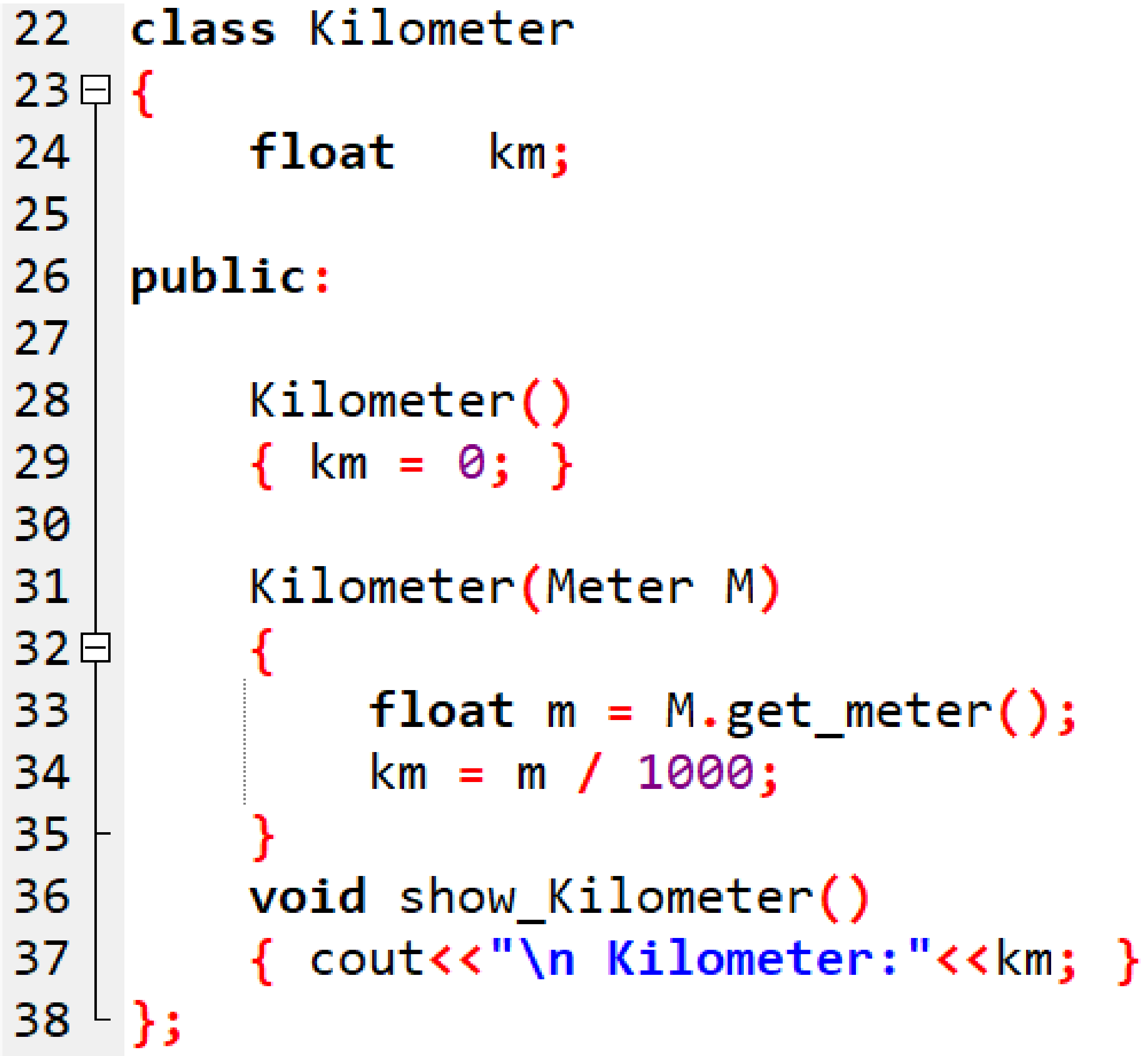
```
class Kilometer
{
     float   km;

public:

     Kilometer()
     { km = 0;   }

     Kilometer(Meter M)
     {

     }
     void show_Kilometer()
     { cout<<"\n Kilometer:"<<km; }
};
```

In main():

```
METER     M(1200);
KILOMETER   K  =  M;
```



```
class Meter
{
      float   mtr;
public:
      Meter()
      { mtr = 0; }

      Meter ( float m )
      { mtr =  m; }

      float get_meter()
      {   return mtr; }

      void show_meter()
      {
            cout<<"\n Meter:"<<mtr;
      }
};
```

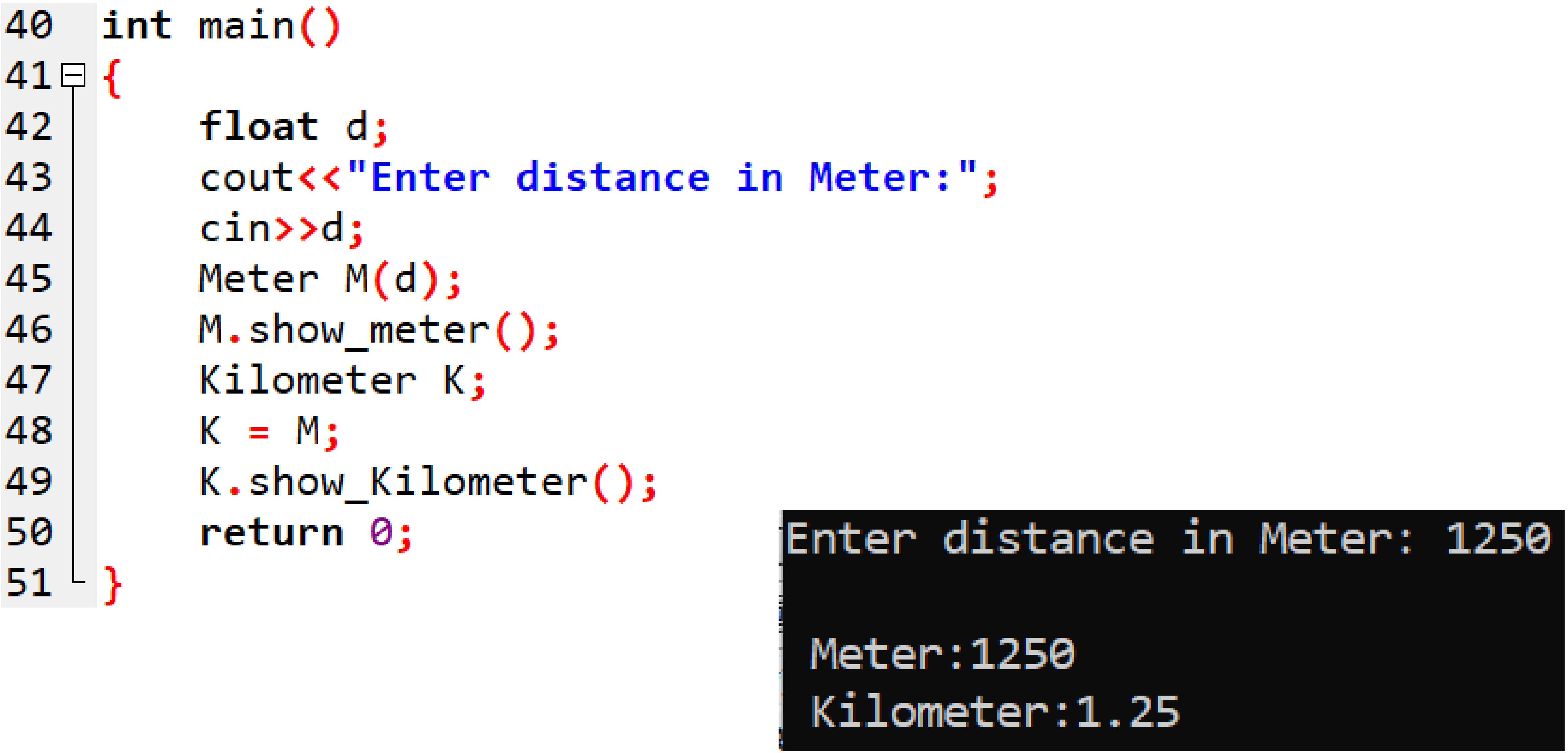


```
class Kilometer
{
      float   km;

public:

      Kilometer()
      { km = 0; }

      Kilometer(Meter M)
      {
            float m = M.get_meter();
            km = m / 1000;
      }
      void show_Kilometer()
      { cout<<"\n Kilometer:"<<km; }
};
```

In main():

METER M(1200);

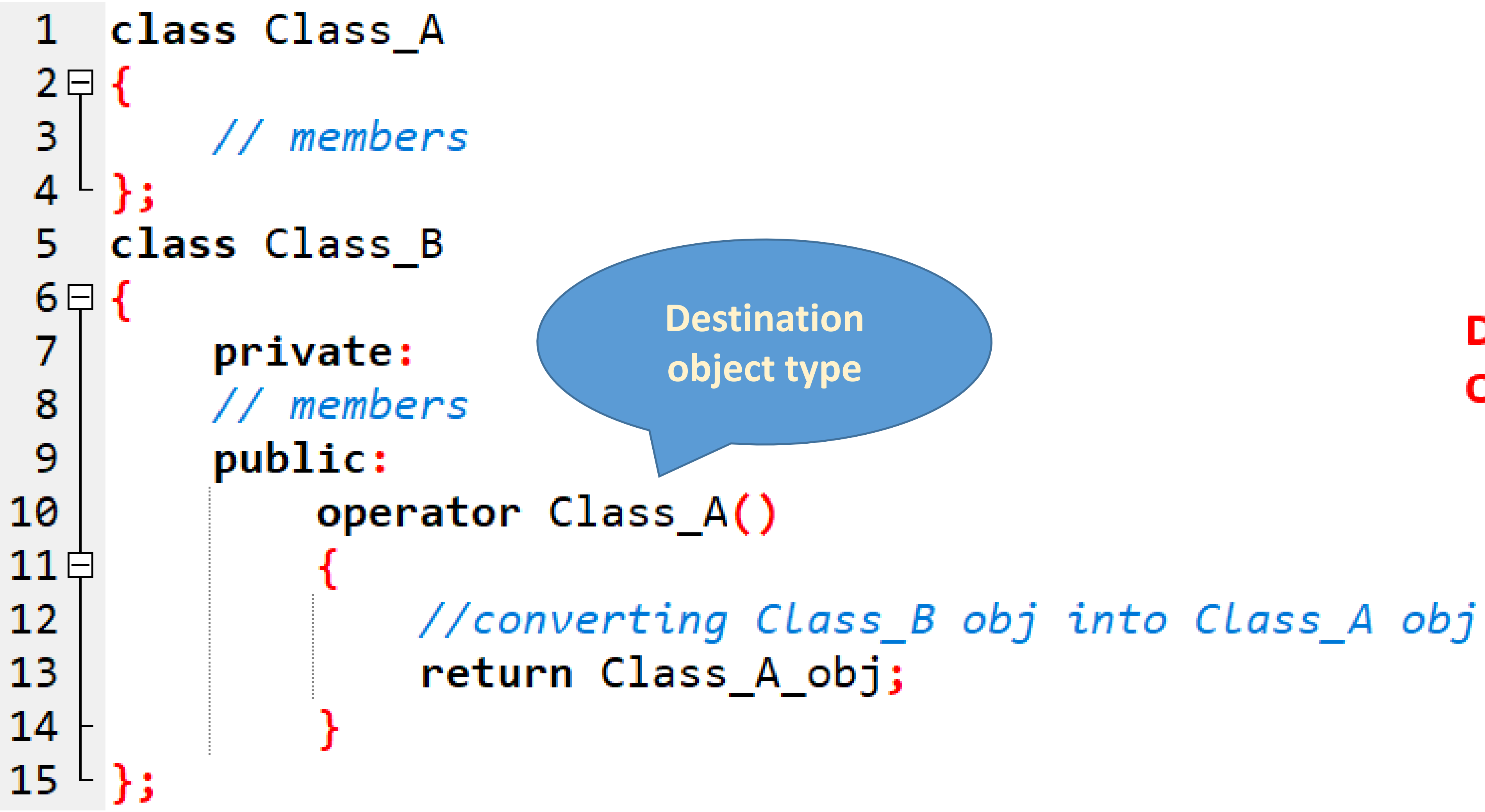
KILOMETER K = M;



```
int main()
{
    float d;
    cout<<"Enter distance in Meter:";
    cin>>d;
    Meter M(d);
    M.show_meter();
    Kilometer K;
    K = M;
    K.show_Kilometer();
    return 0;
}
```

```
Enter distance in Meter: 1250

 Meter:1250
 Kilometer:1.25
```

Solution2→ Conversion routine in source class: operator function



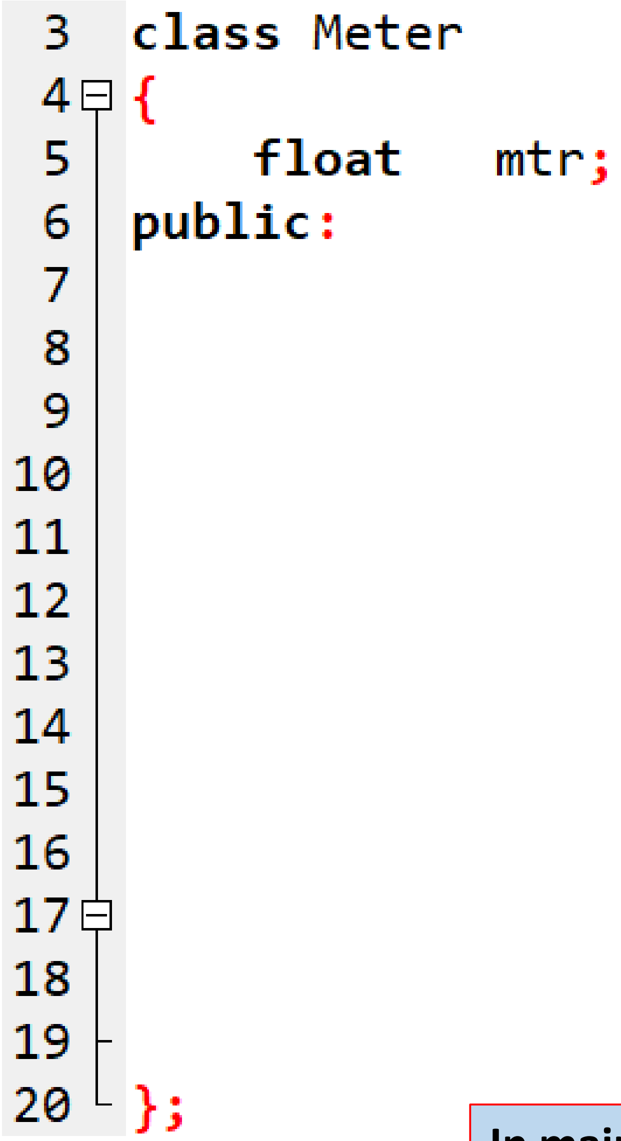
```
class Class_A
{
    // members
};
class Class_B
{
    private:
    // members
    public:
        operator Class_A()
        {
            //converting Class_B obj into Class_A obj
            return Class_A_obj;
        }
};
```

Destination object type

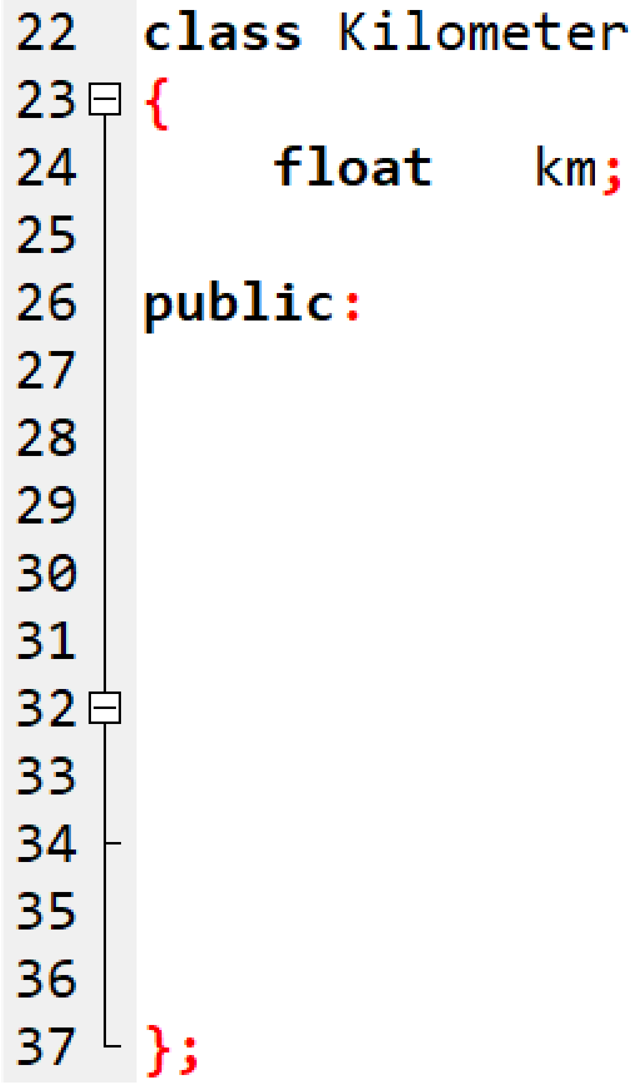
```
Class_A  obj_A;
Class_B  obj_B;

  .  .  .

obj_A = obj_B;
```

Destination Object

Source Object



```
class Meter
{
      float   mtr;
public:














};
```

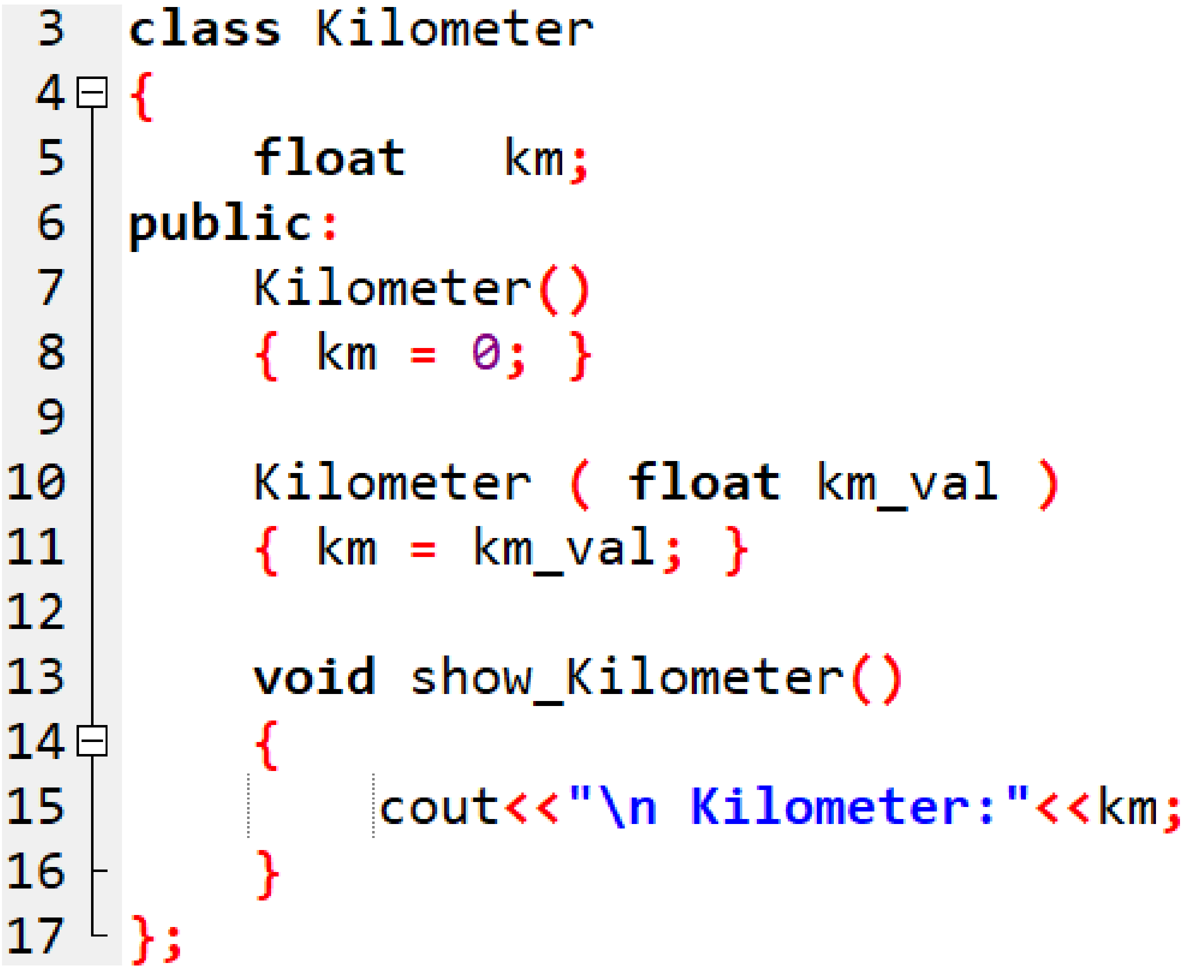


```
class Kilometer
{
      float   km;

public:











};
```

In main():

METER M(1200);

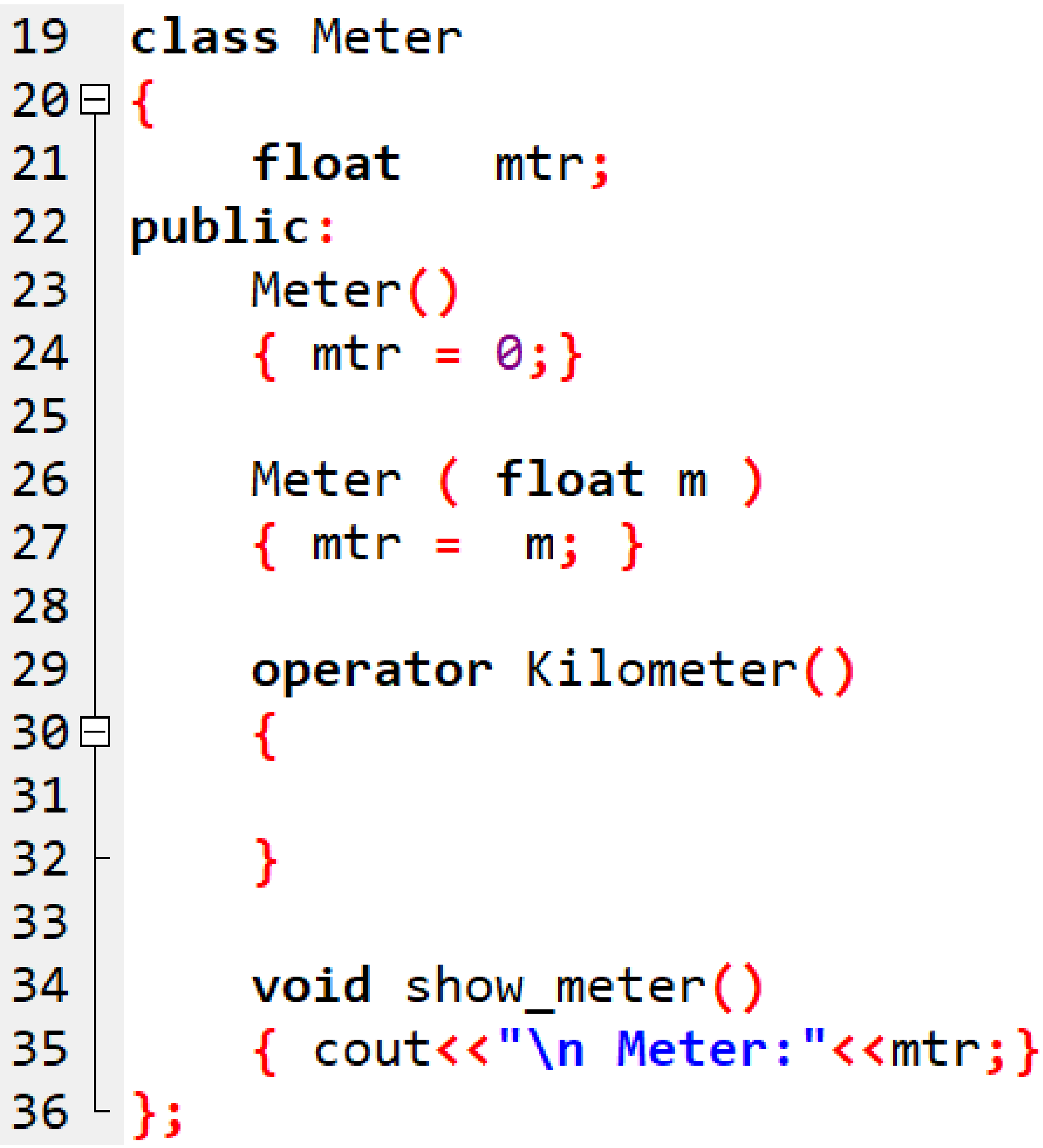
KILOMETER K = M;



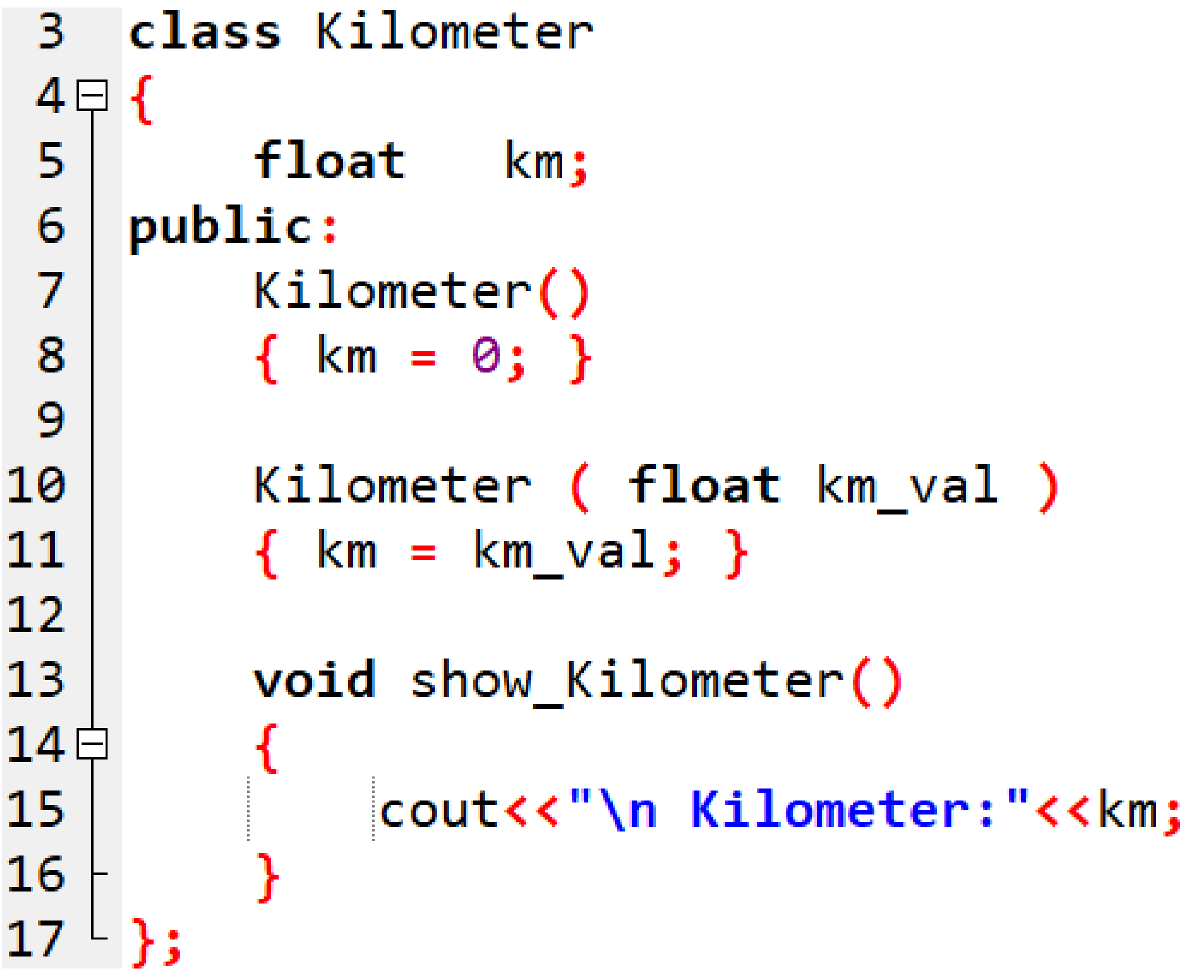
```
class Kilometer
{
    float   km;
public:
    Kilometer()
    { km = 0; }

    Kilometer ( float km_val )
    { km = km_val; }

    void show_Kilometer()
    {
        cout<<"\n Kilometer:"<<km;
    }
};
```

In main():

```
Meter     M(1200);
Kilometer   K;
K  =  M;
```



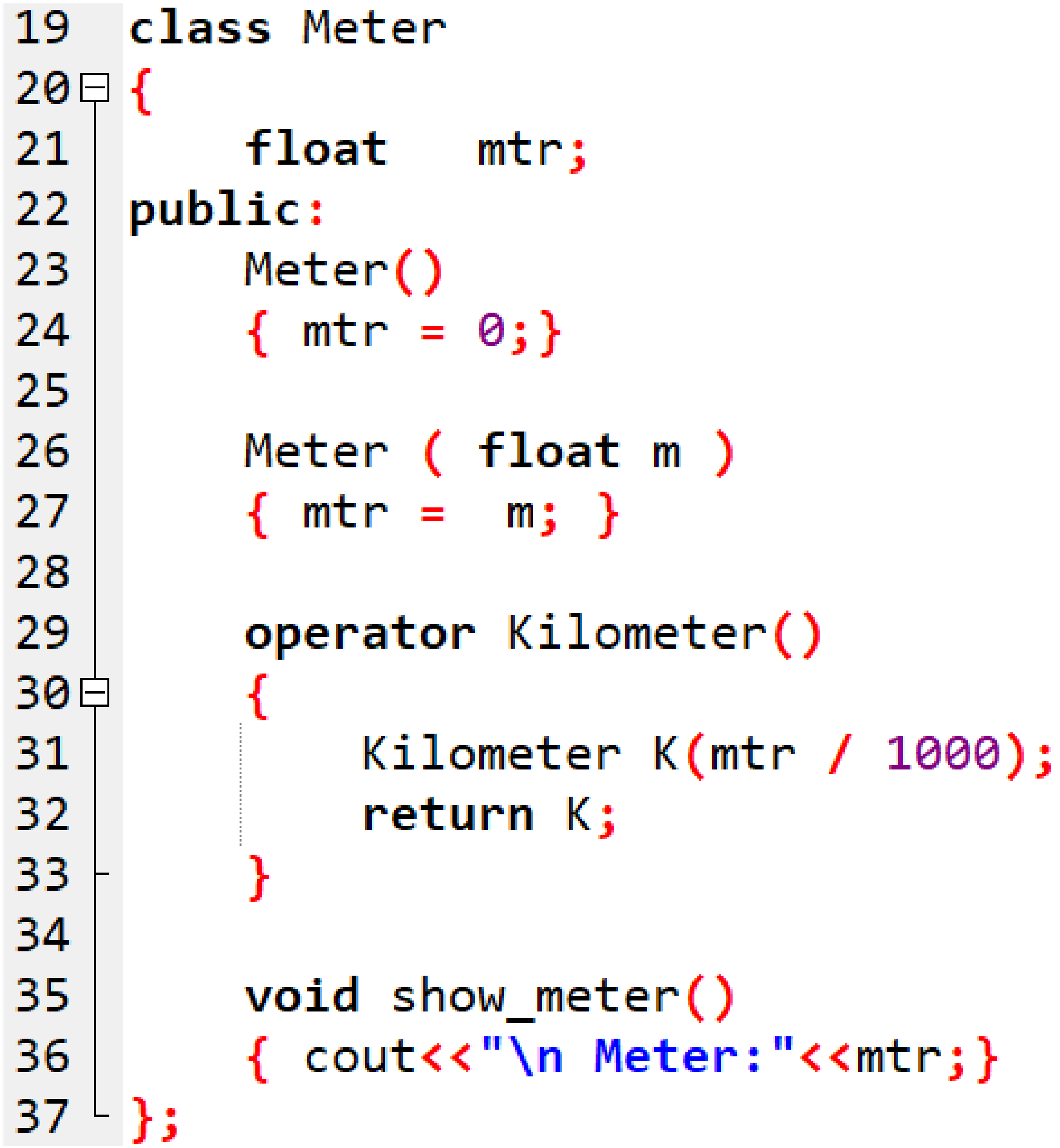
```
class Meter
{
    float   mtr;
public:
    Meter()
    { mtr = 0;}

    Meter ( float m )
    { mtr =  m; }

    operator Kilometer()
    {

    }

    void show_meter()
    { cout<<"\n Meter:"<<mtr;}
};
```



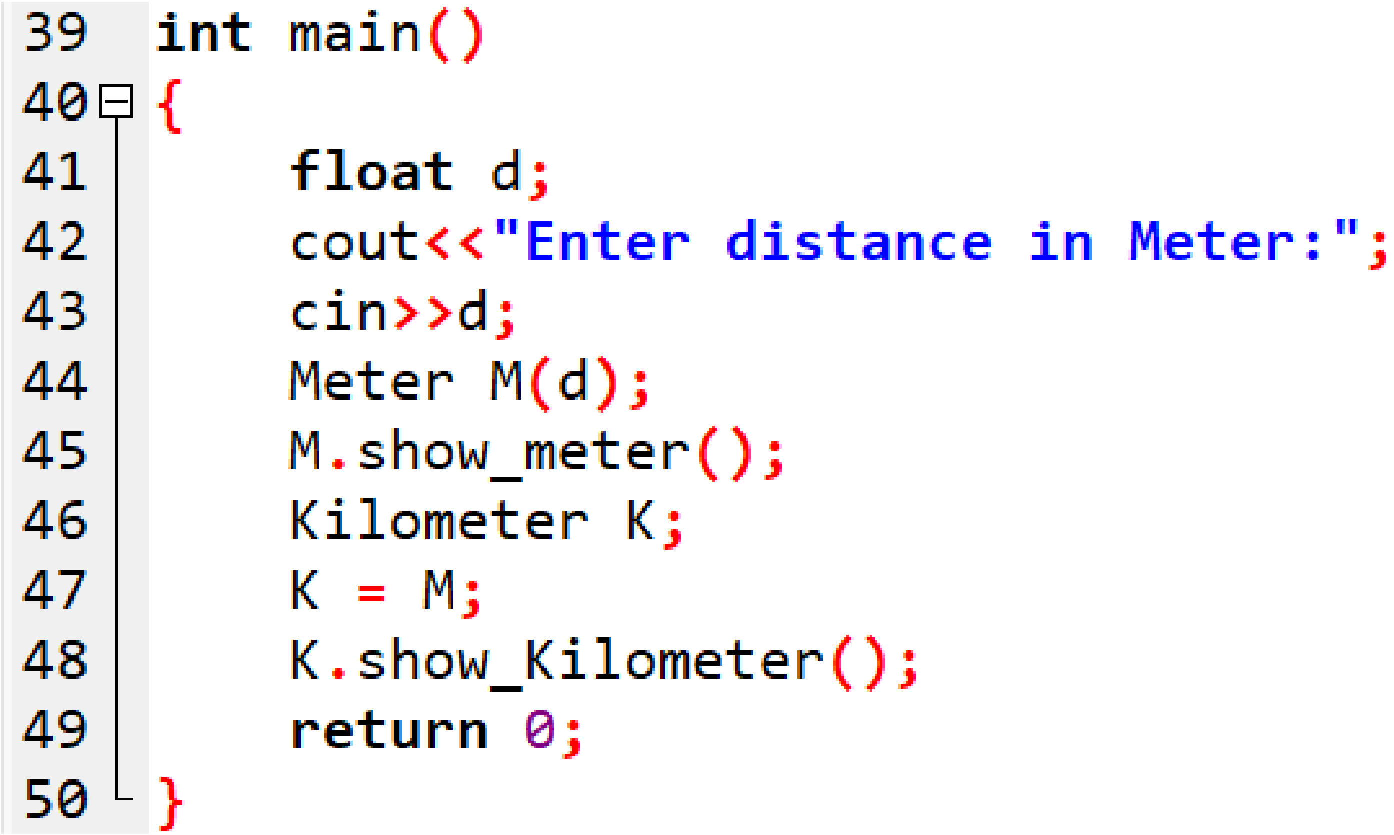
```
class Kilometer
{
    float   km;
public:
    Kilometer()
    { km = 0; }

    Kilometer ( float km_val )
    { km = km_val; }

    void show_Kilometer()
    {
        cout<<"\n Kilometer:"<<km;
    }
};
```

In main():

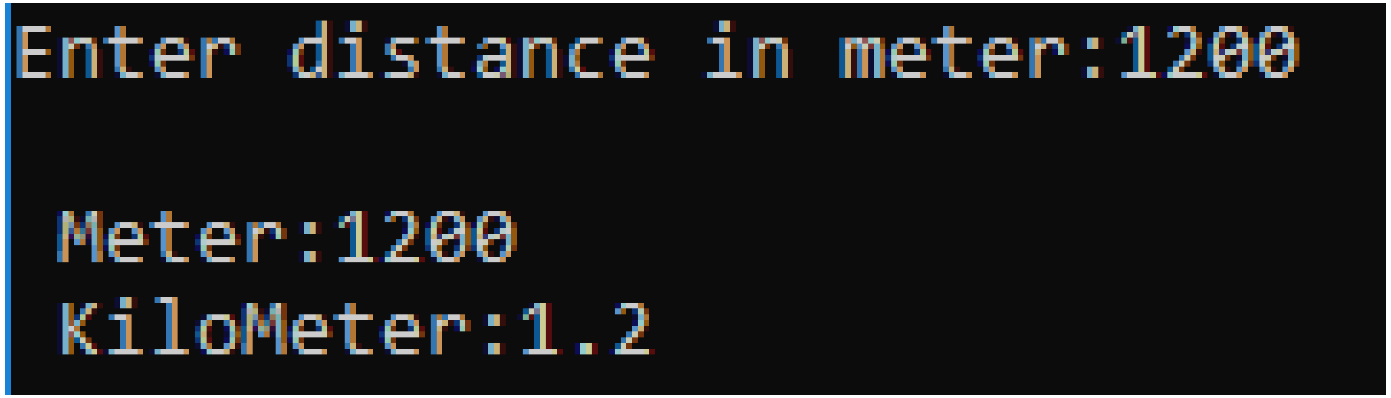
```
Meter     M(1200);
Kilometer   K;
K  =  M;
```



```
class Meter
{
    float   mtr;
public:
    Meter()
    { mtr = 0;}

    Meter ( float m )
    { mtr =  m; }

    operator Kilometer()
    {
        Kilometer K(mtr / 1000);
        return K;
    }

    void show_meter()
    { cout<<"\n Meter:"<<mtr;}
};
```

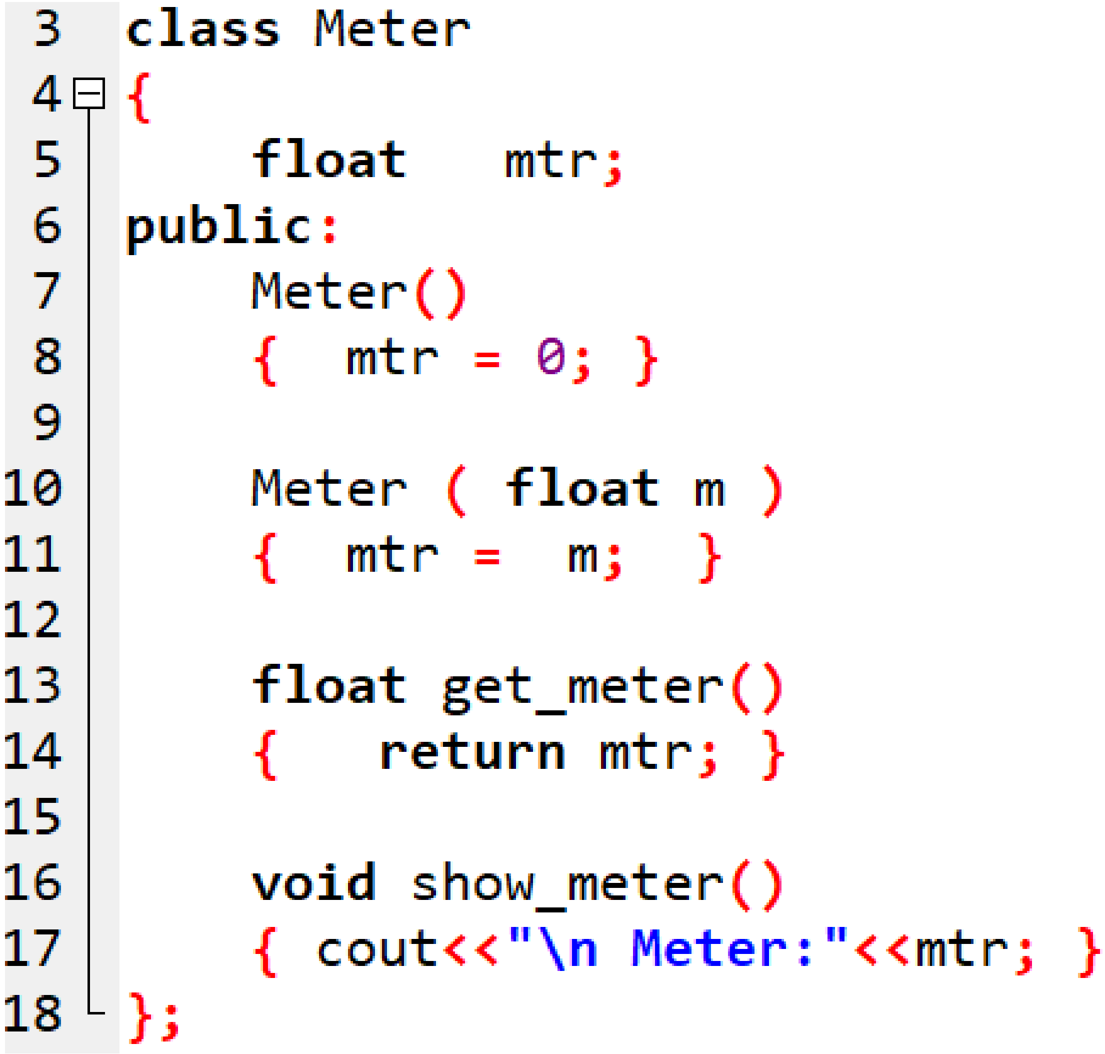


```cpp
int main()
{
	float d;
	cout<<"Enter distance in Meter:";
	cin>>d;
	Meter M(d);
	M.show_meter();
	Kilometer K;
	K = M;
	K.show_Kilometer();
	return 0;
}
```

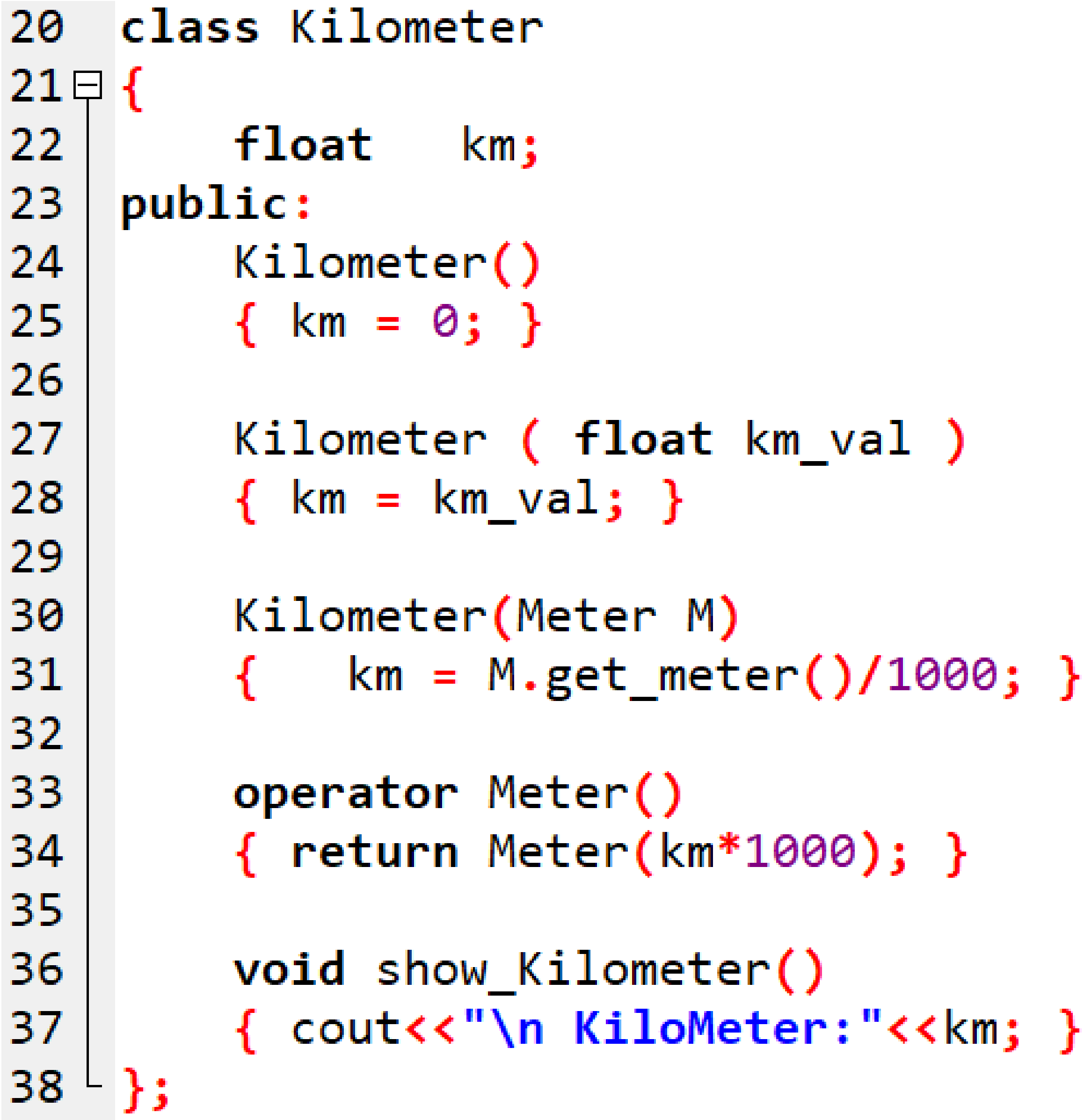


```
Enter distance in meter:1200

 Meter:1200
 KiloMeter:1.2
```

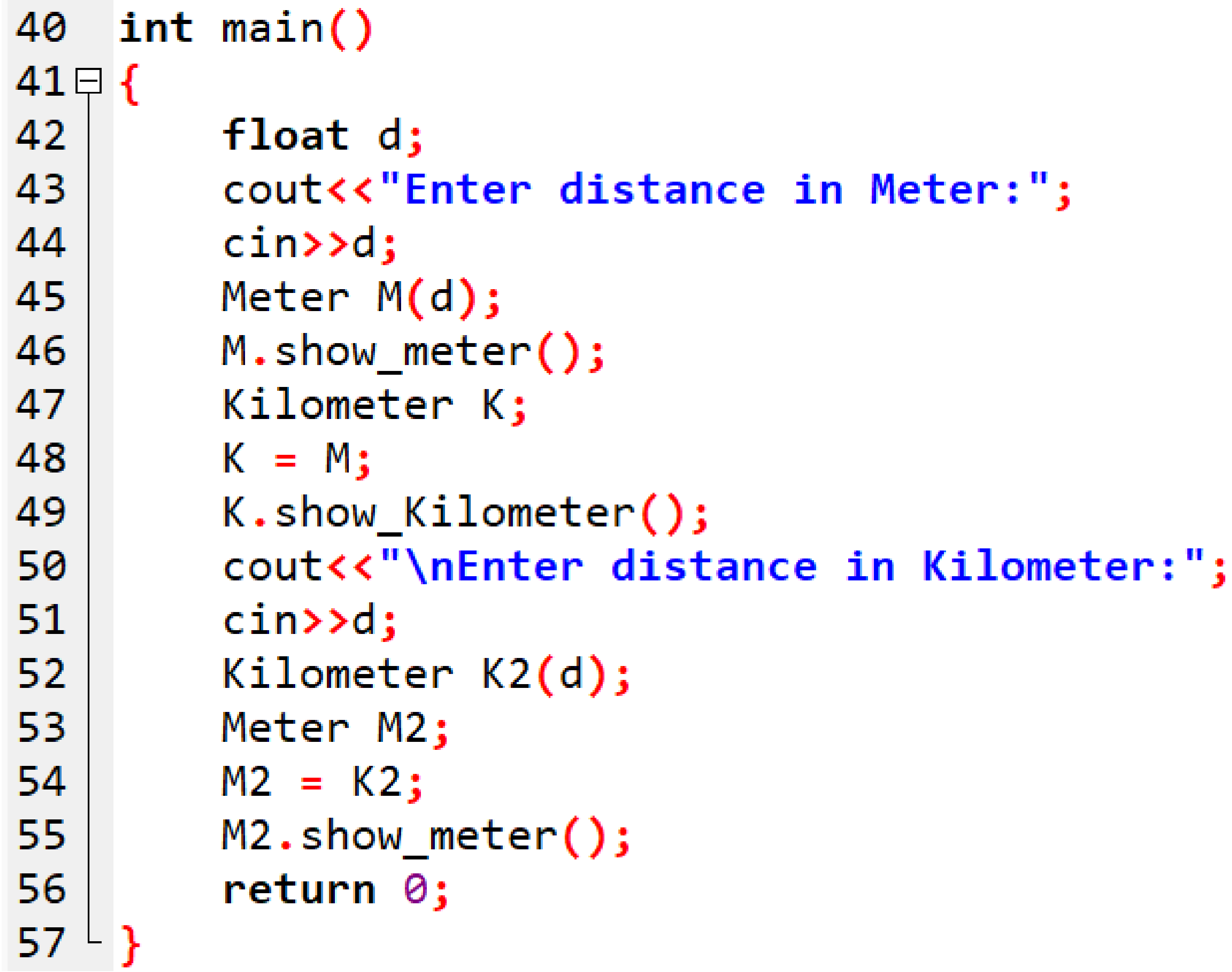
Complete conversion



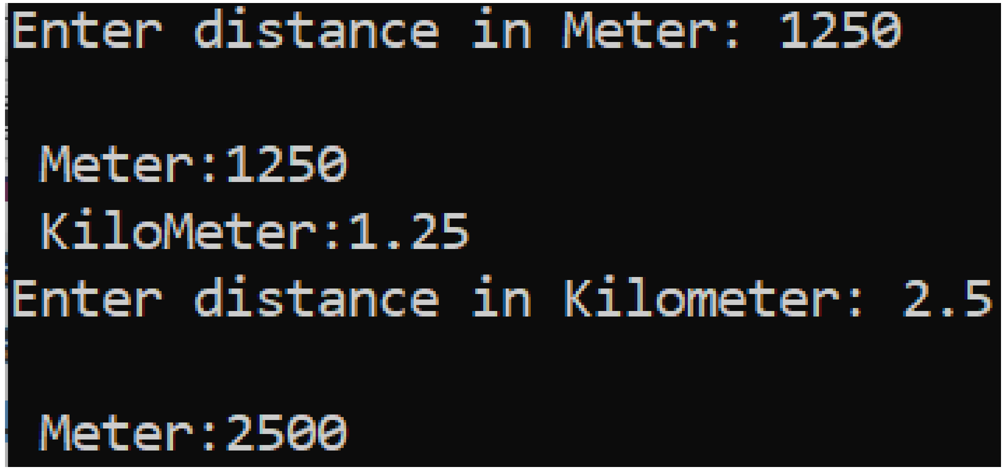
```
class Meter
{
    float   mtr;
public:
    Meter()
    {  mtr = 0; }

    Meter ( float m )
    {  mtr =  m;  }

    float get_meter()
    {   return mtr; }

    void show_meter()
    { cout<<"\n Meter:"<<mtr; }
};
```



```
class Kilometer
{
    float   km;
public:
    Kilometer()
    { km = 0; }

    Kilometer ( float km_val )
    { km = km_val; }

    Kilometer(Meter M)
    {   km = M.get_meter()/1000; }

    operator Meter()
    { return Meter(km*1000); }

    void show_Kilometer()
    { cout<<"\n KiloMeter:"<<km; }
};
```



```cpp
int main()
{
    float d;
    cout<<"Enter distance in Meter:";
    cin>>d;
    Meter M(d);
    M.show_meter();
    Kilometer K;
    K = M;
    K.show_Kilometer();
    cout<<"\nEnter distance in Kilometer:";
    cin>>d;
    Kilometer K2(d);
    Meter M2;
    M2 = K2;
    M2.show_meter();
    return 0;
}
```



```
Enter distance in Meter: 1250

 Meter:1250
 KiloMeter:1.25
Enter distance in Kilometer: 2.5

 Meter:2500
```

- *Write a Program to perform currency conversion*: **Rupee** → **Dollar**

Thank You