Lecture 5

Operator Overloading

Outline

- Overloading built-in C++ operators
- Add Operator +
- Assignment Operator =
- Subscript Operator [ ]
- Function Call Operator ( )
- Unary Operators ++ and --
- Output Operator <<
- Input Operator >>

Operator Overloading

- Built-in C++ operators can be overloaded such as +, > so that they invoke different functions, depending on their operands.

- Example: The + operator in a+b expression will call one function if a and b are integers, but it will call a different function if a and b are objects of a class.

- Overloading doesn't actually add any extra capabilities to C++.
- Everything you can do with an overloaded operator, you can also do with a function.
- Operator overloading is only another way of calling a function.

Overloadable built-in C++ operators

Operator group	Operators
Arithmetic	+ - * / % ++ --
Assignment	= += -= *= /= %=
Comparision	< > == != <= >=
Bitwise	<< >> <<= >>= & ^ | ~ &= ^= |=
Input/output	<< >>
Logical	&& || !
Subscript	[ ]
Function call or expression	( )
Comma	,
Pointer	* ->

Limitations

- Only the existing C++ operators can be overloaded.
  Example: Operator ** can not be overloaded for exponentiation.

- If a built-in operator is binary, then all overloads of it remain binary.
  It is also true for unary operators.

- Operator precedence (priority) and syntax cannot be changed through overloading.
  Example: Operator * has always higher precedence than operator +.

- If an expression contains only built-in data types, overloading can not be applied.
  Example: Operator + can not be overloaded for integers,
  so that x = 3 + 5 works differently.
  At least one operand must be of a user defined type (class).

NON-MEMBER function for operator+ overloading

- **An overloaded operator is a function.**
- To define such a function, the keyword ***operator*** is written followed by the symbol of an operator.
- Example: For addition operation, name of function should be **operator+.**
- Non-member operator+ function takes two arguments, and returns an object.

```
class  ComplexT  {   // Class for complex numbers
    double  re, im;   // Real and imaginary parts
};
```

NON MEMBER

```
ComplexT   operator +   (ComplexT   v1 ,   ComplexT   v2) {
    ComplexT  result;     // local result variable
    result.re   =  v1.re  +  v2.re;
    result.im  =  v1.im  +  v2.im;
    return  result;
}
```

```
int main() {
    ComplexT  c1, c2, c3, c4;      // Complex number objects
    c3 = c1 + c2;                  // The function operator+ is called
    c4 =  operator+  (c1, c2);   // Alternative calling method
}
```

MEMBER function for operator+ overloading

- We can define a member function of ComplexT class, as an overloeded + operator.
- The function will take one argument (another ComplexT object).

```
class ComplexT
{
   double  re, im;
  public:
   // Member function prototype
   ComplexT  operator+  (ComplexT  &);
};
```

```
int main()
{
    ComplexT   z1(1,1),  z2(2,2) , z3;
    z3 = z1 + z2;

    z3 = z1.operator+ (z2);
   // Same as above
}
```

MEMBER

```
// Codes of the overloaded operator+ function ComplexT
ComplexT   ComplexT  ::  operator+  (ComplexT   &  z ) {
   double   re_new,   im_new;
   re_new   =   re  +  z.re;
   im_new   =  im  +  z.im;
   return   ComplexT  (re_new ,  im_new);
}
```

Compiler-provided assignment operator = for array member

- The compiler automatically creates an assignment operator.
- It performs member-by-member assignment by default, which means invoking the **compiler-provided copy constructor.**
- Mostly, there is no need to overload the assignment operator.

```
class String {
    int size;
    char contents [20];  //array member
 public:
     void print();
     String (const char *);
};
```

```
void  String :: print() {
    cout << contents << "  " << size << endl;
}
```

Main program

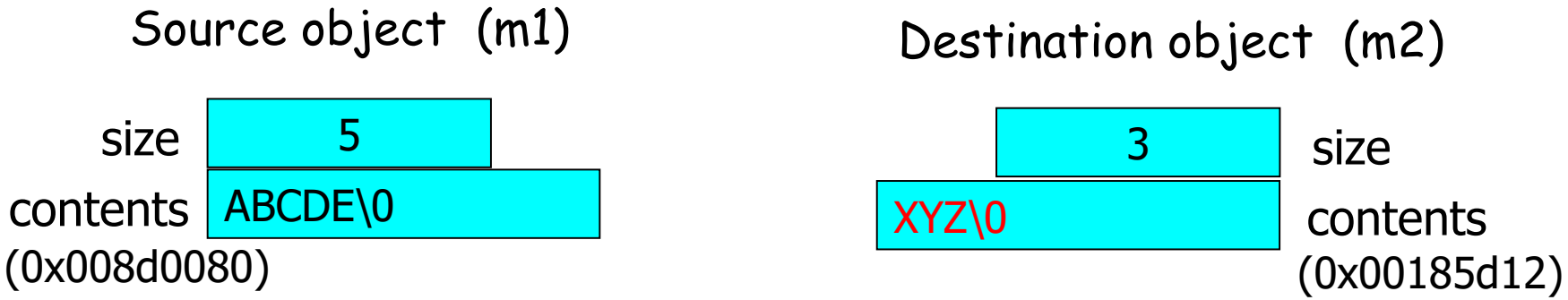
```
// Constructor
String :: String (const char * in_data)
{
    size = strlen(in_data);
    strcpy (contents, in_data);
}
```

```
int main()  {
  String  m1 ("ABCDE");    // Constructor  is called
  String  m2 ("XYZ");      // Constructor  is called
  m2 = m1;  // Compiler-provided assignment operator is called
  m1.print();
  m2.print();
}
```

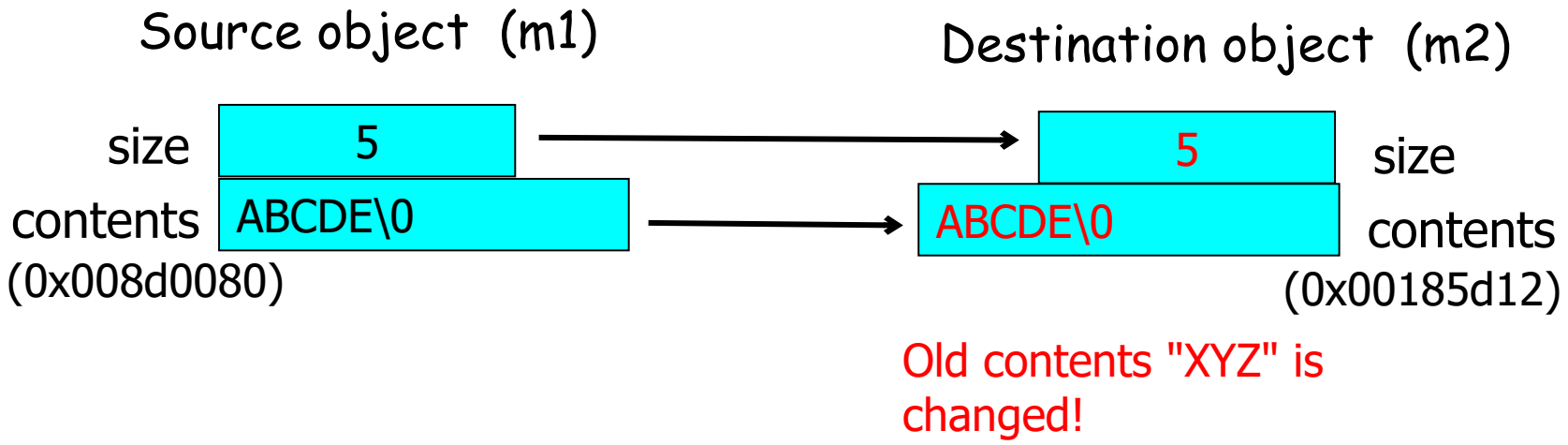
Screen output

```
ABCDE   5
ABCDE   5
```

Compiler-provided assignment operator = for array member



After the compiler-provided assignment operator **(m2 = m1;)** is called :



Compiler-provided assignment operator = for pointer member

In following example, **contents** is a **pointer**, therefore the assignment statement **m2 = m1;** causes copying of **only the pointers**.

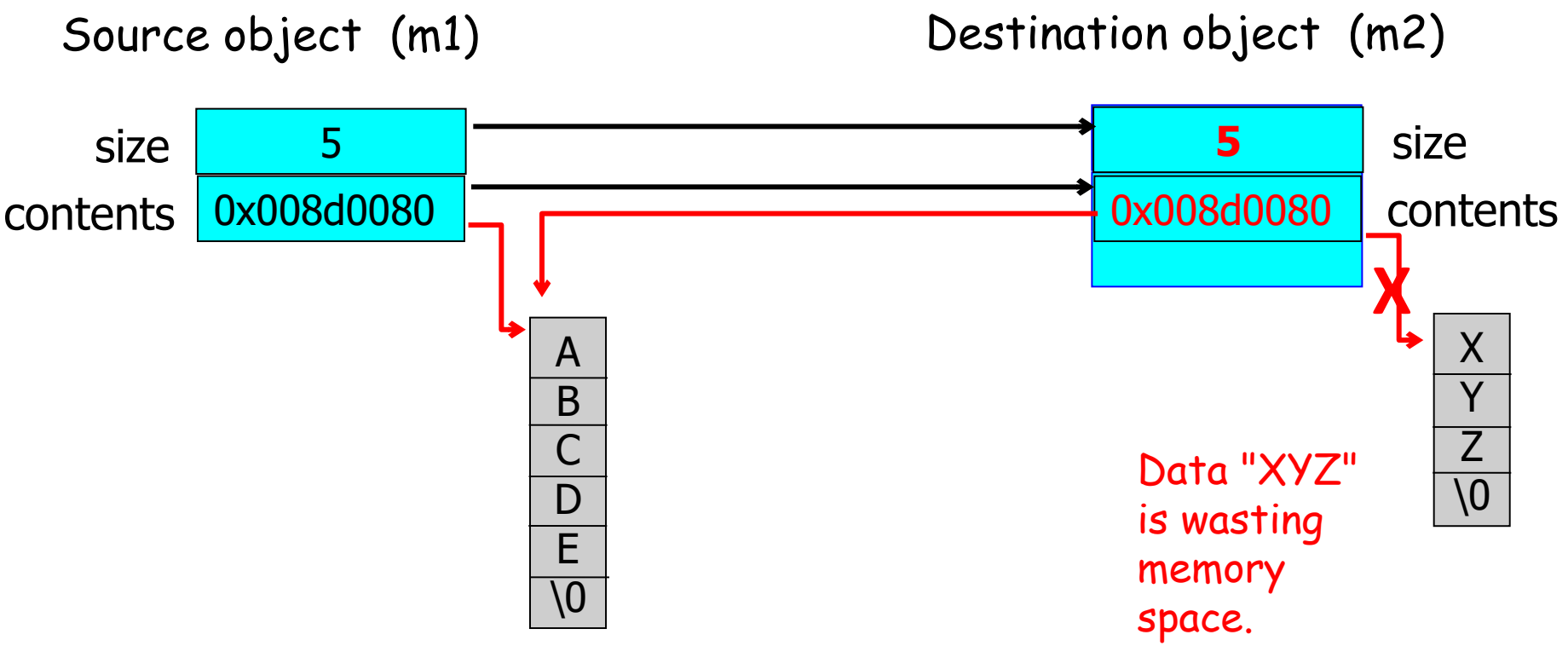
```
class String {
    int size;
    char *contents; // pointer
 public:
     void print();
     String (const char *);
};
```

```
// Constructor
String :: String (const char *in_data)
{
    size = strlen(in_data);
    contents = new char[size +1];
    strcpy (contents, in_data);
}
```

```
int main() {
  String  m1 ("ABCDE");    // Constructor  is called
  String  m2 ("XYZ");       // Constructor  is called
  m2 = m1;          // Compiler-provided assignment operator is called
  m1.print();
  m2.print();
}
```

Compiler-provided assignment operator = for pointer member

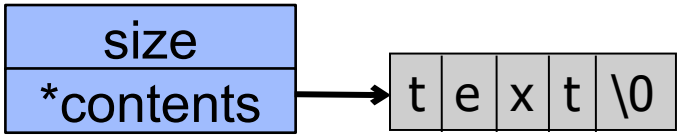
- Disadvantage of compiler-provided assignment operator:
  It may cause memory wasting, if a class contains pointer members.
- Therefore, programmer should write his own overloaded assignment operator.



Overloaded assignment operator =

- When a class contains a pointer member variable, an overloaded assignment operator should be used, instead of compiler-provided assignment operator.

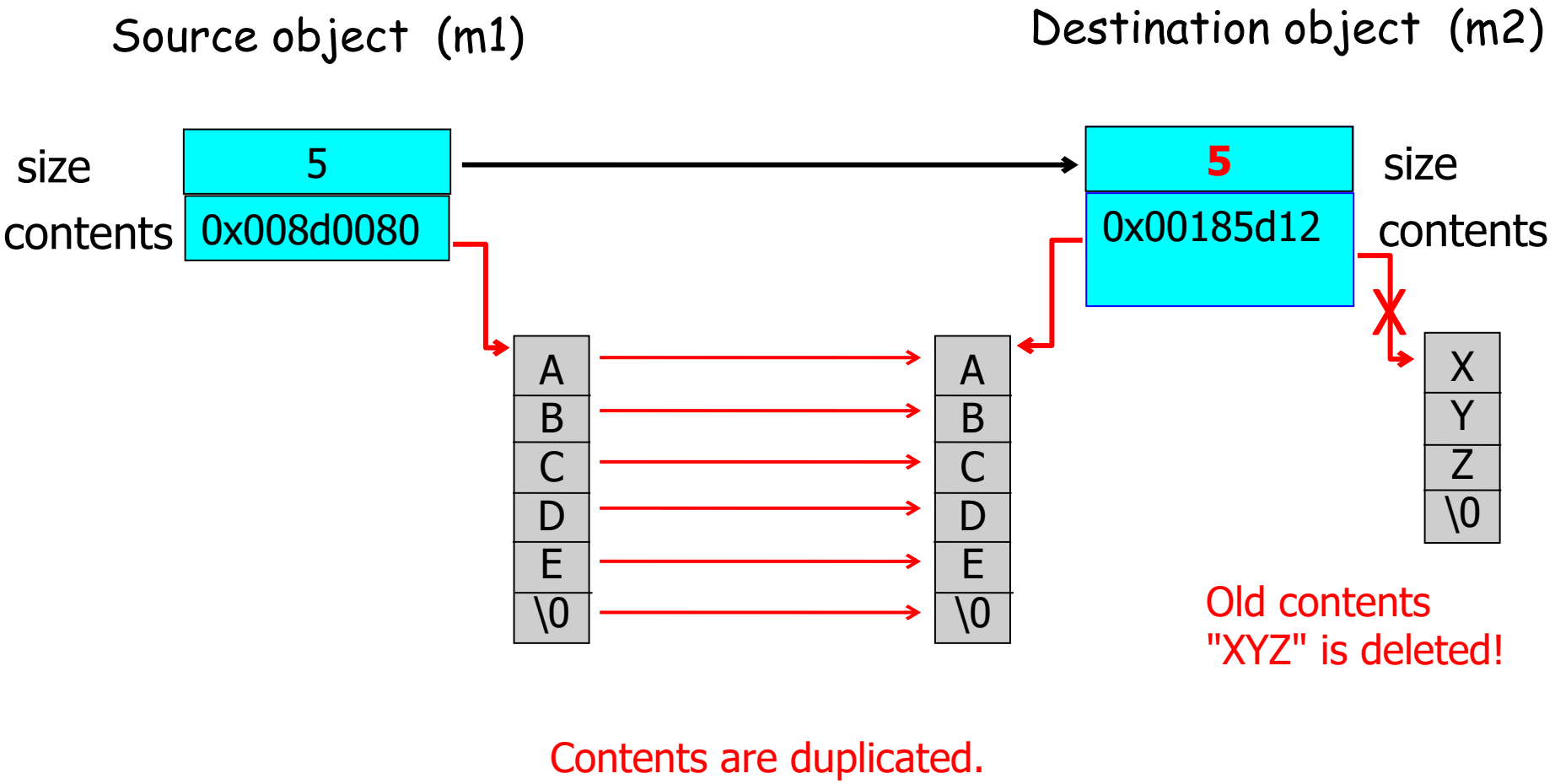
```
class String {
    int size;
    char *contents;  // pointer
 public:
    void  operator= (const  String  &);
    // Overloaded assignment operator

    void print();
    String (const char *); //Constructor
};
```



```
// Overloaded assignment operator (=)
void   String ::  operator=  (const   String   & in_object)
{
// If sizes of source and destination are different,
//  the old contents is deleted.
   if (size  !=  in_object.size) {
       size = in_object.size;
       delete [] contents;                    // Delete old pointer
       contents = new  char [size+1];   // Allocate new pointer
       // Memory allocation for the new contents
   }
   strcpy (contents,  in_object.contents);
}
```

```
int main() {
  String  m1("ABCDE");     // Constructor  is called
  String  m2("XYZ");       // Constructor  is called
  m2 = m1;  // Overloaded assignment operator is called
  m1.print();
  m2.print();
}
```

Overloaded assignment operator =



Overloaded subscript operator [ ] for String class

- If s is an object of String class, the expression **s [i]** is interpreted as:
  s.operator [ ] (i);
- The operator will be used to access the i^{th} character of the string.
- For invalid index values, overloaded [] operator returns the first or the last element.

```
// Overloading the subscript operator
char &  String  ::  operator []  (int  i)    {
    if (i < 0)  i=0;                  // i can not be negative
    if (i >= size)  i=size-1;    // i can not be greater than size
    return   contents[i];          // return i th character
}
```

```
int main() {
    String  s ("ABCD");
    cout << s [2]  << endl;        //Displays C
    cout << s [-3] << endl;        //Displays A
    cout << s [7]  << endl;        //Displays D
}
```

Screen output

```
C
A
D
```

Overloaded function call operator ( )

The function call operator is unique in that it allows any number of arguments.

```
class  C
{
   returntype   operator ( )  (parameter types) ;
};
```

Example usages:

```
C   c;
c () ;          // same as   c .operator ( )  () ;
c (x) ;         // same as    c .operator ( )  (x) ;
c (x , y) ;     // same as   c .operator ( )  (x , y) ;
```

Example : Operator ( )

- The function call operator () is overloaded to print complex numbers on screen.
- Function does not take any function arguments.

```
// The function is called without any arguments.
// It prints the complex number.

void  ComplexT  ::  operator( )  ( )
{
    cout << re << " , " << im <<  endl ;
}
```

```
int main
{
  ComplexT   z (8, 6);
  z ( );   // Displays  z  object member datas
}
```

Screen output

```
8     6
```

Overloaded preincrement operator ++

- Unary operators operate on a single operand.
- Examples: increment (++) , decrement (--) operators.
- Unary operators take no arguments, they operate on the object for which they were called.
- In preincrement, ++ operator appears on the left side of the object, as in ++obj.

Example: We define ++ operator for class ComplexT to increment only the real part of the complex number by 0.1 .

```
void  ComplexT :: operator++ () {
    re = re + 0.1;
}
```

```
int main()
{
    ComplexT  z (1.2,  0.5);
    ++z;   // Preincrement
     //z.operator++();  // same
     z.print();
}
```

Screen output

```
1.3     0.5
```

Overloaded postincrement operator ++

- The declaration, **operator++ (int)** with a single int parameter overloads the postincrement operator.
- The **int** parameter (dummy) will not be used in the function.

```
ComplexT  ComplexT :: operator++ (int)  {  // postincrement operator
    ComplexT   temp;  // local temporary object
    temp = *this;      // old object (original whole object) copied to temp
    re= re + 0.1;      // increment the real part
    return temp;       // return old whole object
}
```

```
int main() {
    ComplexT   z1 (1.2,   0.5),    z2;
    z2 = z1++;
    // Assignment operator is called first (z2 = z1).
    // Then, ++ operator is called for z1.
    z1.print();    // prints the incremented value
    z2.print();    // prints the old value
}
```

Screen output

```
1.3    0.5
1.2    0.5
```

Returning *this in preincrement operator ++

- To be able to assign the preincremented value to a new object, the operator function must return **a reference** to the object.

```
const  ComplexT &     ComplexT  ::  operator++  ()
{
    re = re + 0.1;
    return *this;  // Returns the whole object
}
```

```
int main() {
    ComplexT   z1 (1.2  ,  0.5) ,   z2;
    z2  =  ++z1 ;
    // ++ operator is called first (z1 is modified).
    // Then the incremented value is assigned to z2 .

    z1.print();
    z2.print();   // Same output as z1
}
```

Screen output

```
1.3     0.5
1.3     0.5
```

Non-member function for operator<< overloading

- The << output operator can be overloaded as a non-member function.
- Output operator is a binary operator, it takes exactly two parameters : An ostream object (cout) reference, and a class object.

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

class Tarih {
public:
    int gun, ay, yil;
    Tarih (int gun, int ay, int yil)  //Constructor
          : gun(gun), ay(ay), yil(yil) {}
};

void  operator<<  ( ostream &  cihaz,  Tarih  tar)
{
   cihaz <<  "GUN : "  <<  tar.gun  << endl;
   cihaz <<  "AY  : "  <<  tar.ay   << endl;
   cihaz <<  "YIL : "  <<  tar.yil  << endl;
}

int main() {
    Tarih   trh (1, 4, 2023);  //Calling the constructor
    cout << trh;   //Calling the overloaded  operator<<
    //  operator<< (cout,  trh);  //Alternative calling method
}
```

Screen output

```
GUN : 1
AY  : 4
YIL : 2023
```

Chaining method (cascading) for operator<<

- Overloaded operator<< function can return a new reference to a stream.
- Then, the returned stream can be passed along to the next call of operator<< in the chain.

```
ostream &    operator<<  ( ostream &  cihaz,  Tarih  tar)
{
   cihaz <<  "GUN : "  <<  tar.gun  << endl;
   cihaz <<  "AY  : "  <<  tar.ay   << endl;
   cihaz <<  "YIL : "  <<  tar.yil  << endl;
   return  cihaz;
}


int main() {
    Tarih   trh1 (1, 4, 2023) ,   trh2 (15, 4, 2023); //Two objects defined

    cout << trh1 << "------\n" << trh2 ; //Chaining in operator calling

    //Alternative calling method (chaining)
    // operator<< ( operator<< ( operator<< (cout,trh1) , "------\n") , trh2 );
}
```

Screen output

```
GUN : 1
AY  : 4
YIL : 2023
------
GUN : 15
AY  : 4
YIL : 2023
```

Non-member function for operator>> overloading

- The >> input operator can be overloaded as a non-member function.
- Input operator is a binary operator, it takes exactly two parameters : An istream object (cin) reference, and a class object reference.

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

class Tarih {
public:
    int gun, ay, yil;
    Tarih() {}  //Empty default constructor
};

void   operator>>  ( istream &  cihaz,  Tarih & tar)
{
   cihaz  >>  tar.gun  >>  tar.ay  >>  tar.yil;
}

int main() {
    Tarih   yeni;  //Calling the default constructor
    cout << "Bir tarih giriniz (gun ay yil) : ";
    cin  >> yeni;  //Calling the overloaded  operator>>
    //  operator>>  (cin , yeni);   //Alternative method
    cout << yeni;  //Calling overloaded  operator<<
}
```

Screen output

```
Bir tarih giriniz
(gun ay yil) :  2  4  2023

GUN : 2
AY  : 4
YIL : 2023
```