Lecture 5

Operator Overloading

Topics

- 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 that can be done with an overloaded operator, also can be done with a function.
- Operator overloading is only another way of calling a function.

Built-in C++ operators

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

Operator Limitations

- Only the existing C++ operators can be overloaded.
  Example: The notation ** can not be used as an overloaded operator for exponention, because C++ does not have a ** operator.

- If a built-in operator is binary, then all overloads of it remain binary.
- For unary operators, all overloads remain unary.

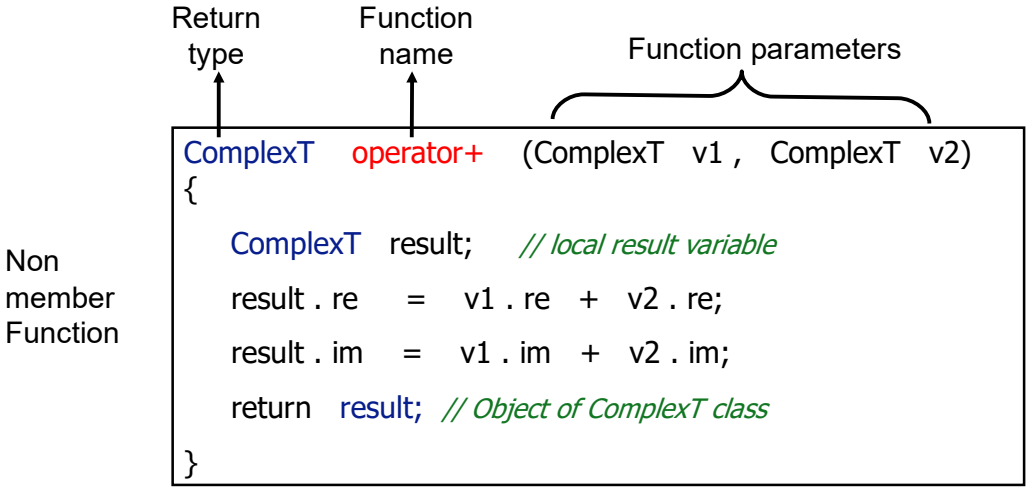
- Operator precedence (priority) and syntax cannot be changed through overloading.
  Example: Operator * has always higher precedence than operator +.

- 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 of the operands must be of a user defined type (class).

Overloading of operator+

- **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 below takes two arguments (parameters), and returns an object.

Non-member function for operator+ overloading



Non member Function

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

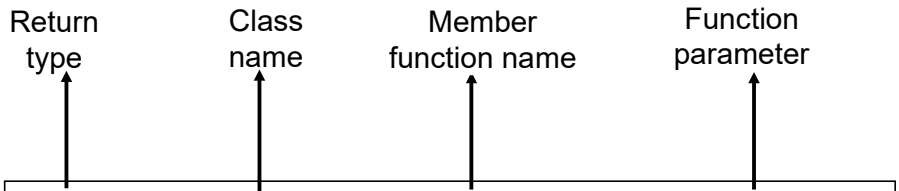


```
int main()
{
    ComplexT  c1, c2, c3, c4;      // Complex number objects
    c3  =  c1  +  c2;              // The function operator+ is called
    c4  =  operator+  (c1, c2);    // Alternative calling method
}
```

Member function for operator+ overloading

- We can define a member function of ComplexT class, as an overloaded + operator.
- The function will take one argument (another ComplexT object).
- Function adds the argument object's data and its data.

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

Member Function

```
ComplexT   ComplexT  ::  operator+  (ComplexT  &  z )
{
   double   re_result,   im_result;   //Local variables

   re_result   =   re  +  z.re;

   im_result   =   im  +  z.im;

   return   ComplexT  (re_result ,  im_result);  //Constructor
}
```

```
int main()
{
    // Complex number objects
    ComplexT   z1 (10, 10),   z2 (20, 20) ,  z3;

   // Calling the overloaded + operator
    z3  =  z1  +  z2;

   // Alternative calling methods
    z3  =  z1 . operator+ (z2);
    z3  =  z2 . operator+ (z1);
}
```


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 *);  //Constructor
};
```

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

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



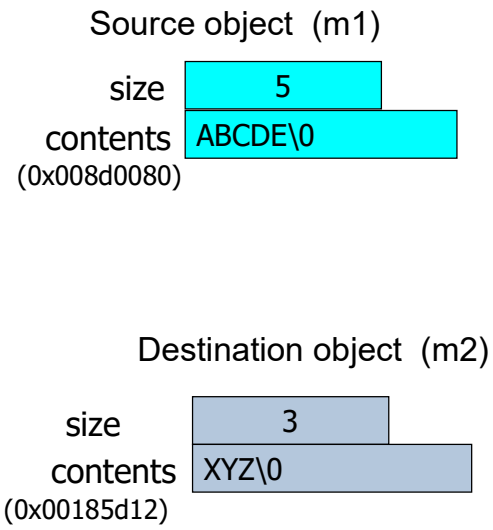
```
int main()
{
  String  m1 ("ABCDE");    // Constructor  is called
  String  m2 ("XYZ");      // Constructor  is called
  m2  =  m1;  // Compiler-provided assignment operator is called.
              // Content of m2 variable is changed.
  m1 . print();
  m2 . print();
}
```

Screen output

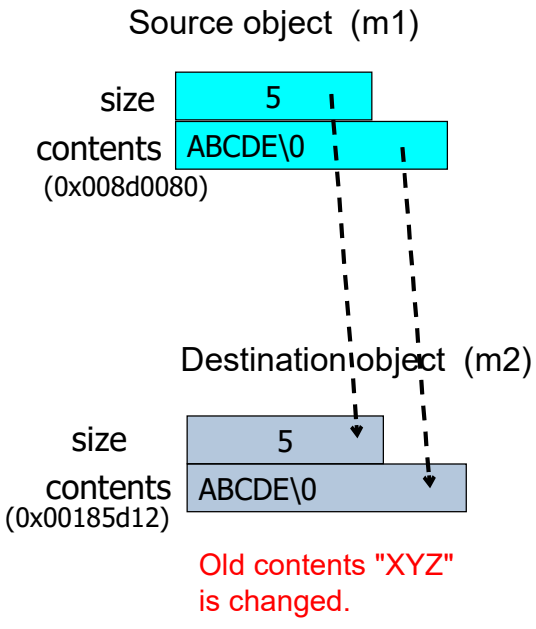
```
ABCDE   5
ABCDE   5
```

Compiler-provided assignment operator = for array member

Initial values in m1 and m2 variables:



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



Compiler-provided assignment operator = for pointer member

In example below, **contents** variable is a **pointer.**
Therefore the assignment statement **m2 = m1**;
causes copying of **only the pointers,** not the data.

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

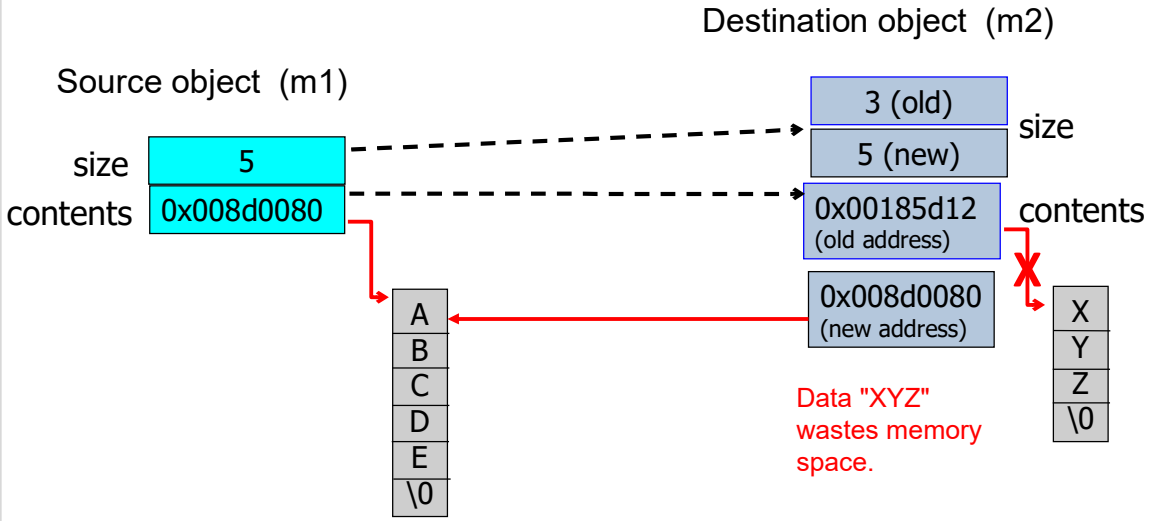


```
// Constructor
String :: String (const char *in_data)
{
    size  =  strlen (in_data);
    contents = new char[size +1];
    strcpy (contents, in_data);
}
```

```
int main()
{
  String  m1 ("ABCDE");    // Constructor  is called
  String  m2 ("XYZ");      // Constructor  is called
  m2  =  m1;          // Compiler-provided assignment operator is called.
                      // Now m2 variable points to contents of m1.
  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 an overloaded assignment operator.

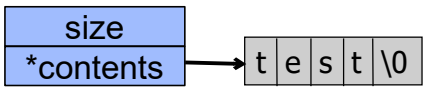


Overloaded assignment operator =

- 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
};
```

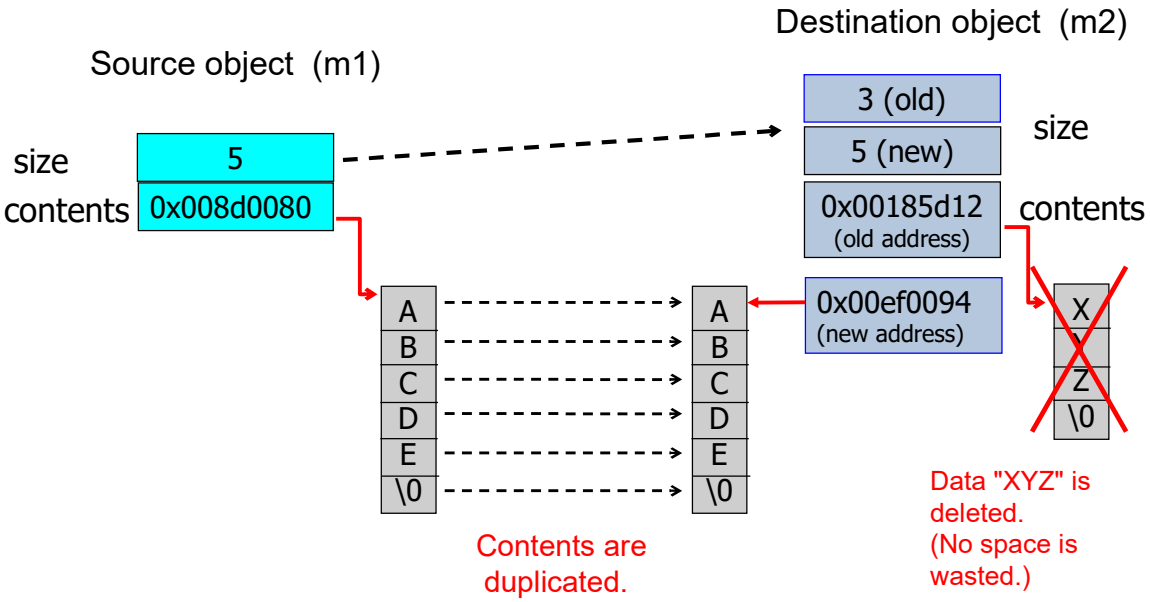


```
// Overloaded assignment operator (=)
void  String :: operator=  (const  String  & input_object)
{
// If sizes of source and destination are different,
//  the old contents is deleted.
  if (size  !=  input_object.size) {
     size  =  input_object.size;
     delete []  contents;                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



```
int main()
{
  String  m1 ("ABCDE");    // Constructor  is called
  String  m2 ("XYZ");       // Constructor  is called
  m2  =  m1;        // Overloaded assignment operator is called.
                    // Contents of m1 variable is copied to m2.
  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 character at index i 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 (Assign i to the first)

   if (i >= size)  i = size-1;    // i can not be greater than size (Assign i to the last)

   return   contents [i];         // Return the character at index location i

}
```

```
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

Overloaded function call operator ( )

- The function call operator () allows any number of arguments.

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

Example usages:

```
C   c;          //define variable 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 () below 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's real and imaginary parts.

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

```
int main
{
  ComplexT   z (10, 20);
  z ( );   // Displays  z  object member datas
}
```

Screen output

```
10    20
```


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 ++z.

Example:
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 (5,  8);
    ++z;                    // Preincrement
     // z.operator++();     // Same result
     z . print();
}
```

Screen output

```
5.1     8
```


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 (5,   8);
   ComplexT   z2;
   z2   =   z1++;
   // Assignment operator is called first (z2 = z1).
   // Then, ++ operator is called for z1.

   z1 . print();    // prints the new incremented value
   z2 . print();    // prints the old original value
}
```

Screen output

```
5.1      8
5        8
```


Returning *this content 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 (5 , 8) ,   z2;
   z2   =   ++z1 ;
   // ++ operator is called first (z1 is modified first).
   // Then the incremented new value is assigned to z2 .

   z1 . print();   // Prints incremented new value
   z2 . print();   // Same output as z1
}
```

Screen output

```
5.1     8
5.1     8
```

Non-member function for operator<< overloading

- The << output operator is overloaded only 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;   //Member data
    Tarih (int d, int m, int y)  //Constructor
          : gun(d), ay(m), yil(y) {}
};

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

int main() {
    Tarih   trh (16, 2, 2024);  //Calling the constructor
    cout << trh;        //Calling the overloaded  operator<<
    // operator<< (cout,  trh);  //Alternative calling method
}
```

Screen output

```
GUN : 16
AY  : 2
YIL : 2024
```


Chaining method (cascading) for operator<<

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

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


int main()
{
    //Define two objects
    Tarih   trh1 (20, 3, 2024);
    Tarih   trh2 (25, 9, 2024);

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

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

}
```

Screen output

```
GUN : 20
AY  : 3
YIL : 2024
------
GUN : 25
AY  : 9
YIL : 2024
```

Non-member function for operator>> overloading

- The >> input operator is overloaded only 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 &  klavye,  Tarih & tar)
{
   klavye  >>  tar.gun  >>  tar.ay  >>  tar.yil;
}

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

Screen output

```
Bir tarih giriniz
(gun ay yil) :  28  10  2024

GUN : 28
AY  : 10
YIL : 2024
```