Operator Overloading

Feza BUZLUCA
Istanbul Technical University
Computer Engineering Department
http://faculty.itu.edu.tr/buzluca
http://www.buzluca.info

Overview

- Introduction
- Fundamentals of Operator Overloading
- Restrictions on Operator Overloading
- Overloading Binary Operators
- Overloading Unary Operators
- Overloading ++ and -- Operators

Operator Overloading

- It is possible to overload built-in C++ operators (such as +, >=, and ++) so that they invoke different functions, depending on their operands
- That is, the ‘+’ in ‘a + b’ will call one function if a and b are integers, but will call a different function if a and b are objects of a class you have created

Operator Overloading: Another Way to Make a Function Call

- Overloading does not actually add any capabilities to C++
- Everything you can do with an overloaded operator you can also do with a function
- However, overloaded operators make your programs easier to write, read, understand, and maintain
- Operator overloading is only another way of calling a function
- Looking at it this way, you have no reason to overload an operator except if it will make the code involving your class easier to write and especially easier to read
- Remember, code is read much more than it is written

Restrictions

- You cannot overload operators that do not already exist in C++
  - You cannot make up a ** operator for (say) exponentiation
- You can overload only the built-in operators
  - Even a few of these, such as the dot operator (.), the scope resolution operator (::), the conditional operator (?:), and sizeof, cannot be overloaded

Restrictions

C++ operators can be divided roughly into binary and unary

- Binary operators take two arguments
  - Examples: a + b, a - b, a / b, ...
- Unary operators take only one argument
  - Examples: -a, ++a, a--

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

Restrictions

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

Restrictions

- The meaning of how an operator works on values of fundamental types cannot be changed by operator overloading
  - Example: you can never overload operator '+' for integers so that a = 1 + 7; behaves differently
- Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type

Overloading the + operator for ComplexT objects

```
// class to define complex numbers
class ComplexT {
   double re, im;
  public:
          :                                          // member functions
   ComplexT operator+(const ComplexT&) const;      // prototype of operator+ function
};

// body of the function for operator +
ComplexT ComplexT::operator+(const ComplexT& z) const
{
   double reNew, imNew;
   reNew = re + z.re;
   imNew = im + z.im;
   return ComplexT(reNew, imNew);          // constructor ComplexT(double,double) is needed
}

int main()
{
   ComplexT z1(1,1), z2(2,2) , z3;
     :                              // Other operations
   z3 = z1 + z2;                    // like z3 = z1.operator+(z2);
```

See Example e51.cpp

Overloading the Assignment Operator (=)

- Since most people expect to be able to assign an object to another object of the same type, the compiler will automatically create an assignment operator type::operator=(const type &) if you do not write one
  - This operator performs memberwise assignment of the class's data members
  - Each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left)
  - If this operation is sufficient, you do not need to overload the assignment operator

Example of Not Needing to Overload the Assignment Operator (=)

- For example, overloading of assignment operator for complex numbers is not necessary

```
void ComplexT::operator=(const ComplexT& z) // unnecessary
{
    re = z.re;        // Memberwise assignment
    im = z.im;
}
```

See Example e52.cpp

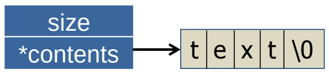
- You do not need to write such an assignment operator function because the operator provided by the compiler does the same thing

Overloading the Assignment Operator (=)

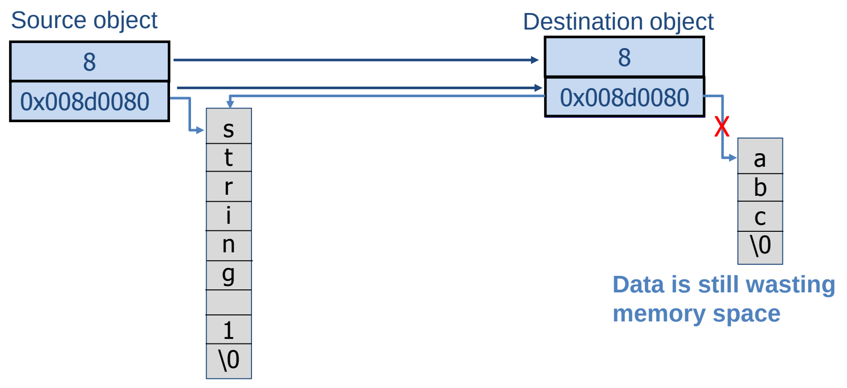
- With classes of any sophistication (especially if they contain pointers!), you have to explicitly create an operator=
- Example: String class

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

void String::operator=( const String &inObject )
{
   if ( size != inObject.size ) {          // if sizes of source and dest.
      size = inObject.size;                // objects are different
      delete [] contents;                  // delete old contents
      contents = new char[size + 1];       // allocate memory for new contents
   }
   strcpy(contents, inObject.contents); // not memberwise assignment
}
```

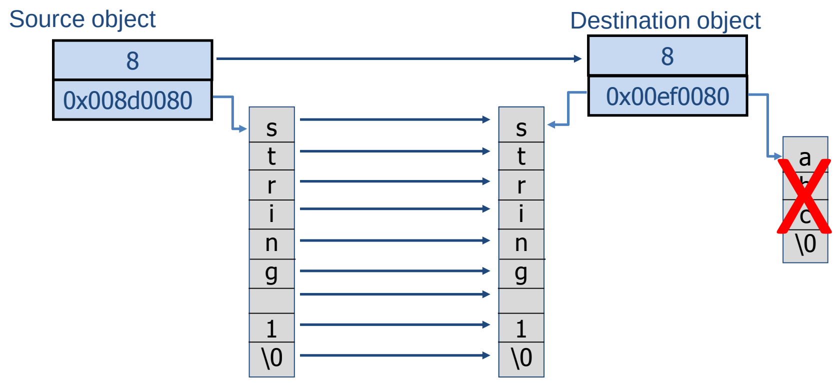


Operator Provided By the Compiler



Data is still wasting memory space

Operator Coded by the Programmer



Return Value of Assignment Operator

- When there is a void return value, as shown in the previous example, you cannot chain the assignment operator (as in a = b = c)
- To fix this, the assignment operator must return a reference to the object that called the operator function (its address)

```
// assignment operator (can be chained as in  a = b = c )
const String& String::operator=( const String &inObject )
{
    if ( size != inObject.size ){                // if sizes of source and destination
       size = inObject.size;                      // objects are different
       delete [] contents;                        // delete old contents
       contents = new char[size + 1];             // allocate memory for new contents
   }
   strcpy( contents, inObject.contents );
   return *this;                                  // return a reference to the object
}
```

See Example e53.cpp

Assignment Operator vs. Copy Constructor

- Notice how similar the overloaded assignment operator is to the copy constructor, which often performs exactly the same operation on the data
- The difference is:
  - Copy constructor actually creates a new object before copying data from another object into it
  - Assignment operator copies data into an already existing object

Unusual Operators

- Same rules apply to all operators
- So, we do not need to discuss each operator
- However, we will examine some interesting operators
  - Subscript operator
  - Function call operator
  - Unary operators

Overloading the Subscript Operator [ ]

- It is usually declared in two different ways:

```
class C {
   returntype & operator[] (paramtype);                    // for the left side of an
                                                           // assignment
         or

   const returntype & operator[] (paramtype) const;  // for the right side
};
```

- The first declaration can be used when the overloaded subscript operator modifies the object
- The second declaration is used with a const object
  - In this case, the overloaded subscript operator can access but not modify the object
- If c is an object of class C, the expression

```
c[i];
```

is interpreted as

```
c.operator[](i);
```

Overloading the Subscript Operator: String Class Example

- Use the operator to access the ith character of the string
  - If i is less than zero, access the first character
  - If i is greater than the size of the string, access the last character

```
char & String::operator[](int i)                    // subscript operator
{
  if ( i < 0 )
    return contents[0];                             // return first character
  if ( i >= size )
    return contents[size - 1];                      // return last character
  return contents[i];                               // return ith character
}
int main()
{
  String s1( "String 1" );
  s1[1] = 'p';                                      // modify an element
  s1.print();
  cout << " 5th character of the string s1 is: " << s1[5] << endl;
  return 0;
}
```

See Example e54.cpp

Overloading the Function Call Operator ()

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

```
class C {
   returntype operator()( paramtypes );
};
```

- If c is an object of class C, the expression

```
c( i, j, k );
```

is interpreted as

```
c.operator()( i, j, k );
```

Overloading the Function Call Operator (): Complex Number Example

- Example: Overload function call operator to print complex numbers on screen
  - In this example, the function call operator does not take any arguments

```
// function call operator without any arguments
void ComplexT::operator( )( ) const
{
    cout << re << " , " << im <<  endl ; // print a complex number
}
```

See Example e55.cpp

Overloading the Function Call Operator (): String Example

- Example: Overload function call operator to copy part of a string into given memory location
  - Two arguments: address of dest. memory and number of characters to copy

```
// function call operator with two arguments
void String::operator( )( char * dest, int num) const
{
    if ( num > size ) num = size;      // if num is greater than size of the string
    for ( int k = 0; k < num; k++ )  dest[k] = contents[k];
}
// ----- Main function -----
int main( )
{
    String s1( "Example Program" );
    char *c = new char[8];              // destination memory
    s1(c, 7);                           // first 7 letters of string1 are copied into c
    c[7] = '\0';                        // end of string (null) character
    cout << c;
    delete [] c;
    return 0;
}
```

Is this statement understandable?

See Example e56.cpp

Unary Operators

- Unary operators operate on a single operand
- Some examples of unary operators are:
  - increment (++) and decrement (--) operators
  - unary minus (as in -5)
  - logical not (!) operator
- Unary operators take no arguments (they operate on the object for which they were called)
- Normally, unary operators appear on the left side of objects
  - as in !obj, -obj, and ++obj

Overloading Unary Operators: ++ in ComplexT Example

- Example: Define ++ operator for class ComplexT to increment 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;                                 // z.operator++()
   z.print();
   return 0;
}
```

Overloading Unary Operators: ++ in ComplexT Example

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

```
// ++ operator
// increment the real part of a complex number by 0.1
const ComplexT & ComplexT::operator++()
{
   re = re + 0.1;
   return *this;
}

int main()
{
   ComplexT z1( 1.2, 0.5 ), z2;
   z2 = ++z1;          // ++ operator called, incremented value assigned to z2
   z2.print();
   return 0;
}
```

See Example e57.cpp

Unary Prefix and Postfix ++ and – Operators

- Recall that ++ and -- operators come in prefix and postfix forms
- When used in assignment statements, different forms have different meanings:

```
z2 = ++z1;     // preincrement
z2 = z1++;     // postincrement
```

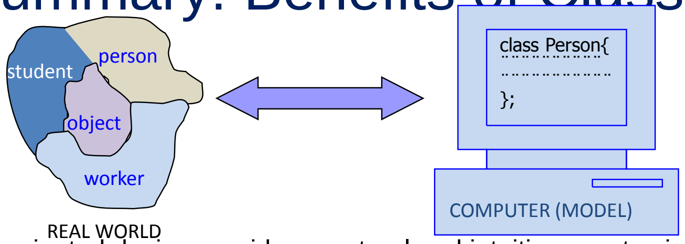
- The declaration operator++( ) with no parameters overloads the preincrement operator
- The declaration operator++(int) with a single int parameter overloads the postincrement operator
- Here, the int parameter serves to distinguish the postincrement form from the preincrement form
  – This parameter is not used

Unary Prefix and Postfix ++ and – Operators: ++ in ComplexT Example

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

See Example e58.cpp

Summary: Benefits of Classes



- Object-oriented design provides a natural and intuitive way to view the software design process (namely, modeling real-world objects)
  - There is a one-to-one relationship between objects in the real world and objects in the program
- Encapsulation: Programs are easy to read and understand. The data and functions of an object are intimately tied together.
- Information hiding: Objects may know how to communicate with other objects, but normally they are not allowed to know how other objects are implemented. This property prevents data corruption. It is easy to find errors.
- Objects are active data structures. They can receive messages and perform some actions according to these messages.