Computer Programming
Chapter 11: Operator Overloading

Hung-Yu Wei
Department of Electrical Engineering
National Taiwan University

[Concept] Overloading Operator

- C++ has defined many useful operators
  - We want to use those operators on the user-defined objects
- Operand
- Operator could be applied for different types (or different classes) --
  - Unary operators
    - ++ ! ~
  - Binary operators
    - + - * / || &&
- Similar Idea
  - Overloading function
    - int MaxFunction (int, int)
    - double MaxFunction (double, double)

Why Operator Overloading?

- C++ is extensible
  - User defined class and objects
  - You want those operators work with your new objects
  - For example
    - Well-known operator +

      int2 = int2 + int1;
    - User defined objects

      Object 2 = Object 2 + Object 1;
  - Notice
    - Defining "Object 2 = Object 2 + Object 1;" does *NOT* imply defining "Object2 += Object1"
      - Need to define separately and clearly for "+="

Operators

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

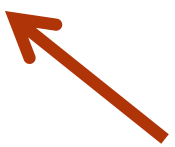
Fig. 11.1 | Operators that can be overloaded.

Operators that cannot be overloaded			
.	.*	::	?:

Fig. 11.2 | Operators that cannot be overloaded.

```
1  // vectors: overloading operators example
2  #include <iostream>
3  using namespace std;
4
5  class CVector {
6    public:
7      int x,y;
8      CVector () {};  //A better solution
9                      //CVector () { x=0; y=0; };
10     CVector (int,int);
11     CVector operator + (CVector);
12 };
13
14 CVector::CVector (int a, int b) {
15   x = a;
16   y = b;
17 }
```

```
19 CVector CVector::operator+ (CVector param) {
20   CVector temp;
21   temp.x = x + param.x;
22   temp.y = y + param.y;
23   return (temp);
24 }
25
26 int main () {
27   CVector a (5,2);
28   CVector b (3,1);
29   CVector c;
30   c = a + b;
31   cout << c.x << "," << c.y;
32
33   system("PAUSE");
34   return 0;
35 }
```



Operator+ is overloaded

Example: Overloading stream insertion and stream extraction operators (11.5)

- C++ standard library defined stream insertion and stream extraction (e.g. cout and cin)

- Stream insertion operator <<
  - ostream &

- Stream extraction operator >>
  - istream &

- operator

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8  using namespace std;
9
10 class PhoneNumber
11 {
12    friend ostream &operator<<( ostream &, const PhoneNumber & );
13    friend istream &operator>>( istream &, PhoneNumber & );
14 private:
15    string areaCode; // 3-digit area code
16    string exchange; // 3-digit exchange
17    string line; // 4-digit line
18 }; // end class PhoneNumber
19
20 #endif
```

Fig. 11.3 | `PhoneNumber` class with overloaded stream insertion and stream extraction operators as `friend` functions.

```
1  // Fig. 11.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  // overloaded stream insertion operator; cannot be
9  // a member function if we would like to invoke it with
10 // cout << somePhoneNumber;
11 ostream &operator<<( ostream &output, const PhoneNumber &number )
12 {
13    output << "(" << number.areaCode << ") "
14       << number.exchange << "-" << number.line;
15    return output; // enables cout << a << b << c;
16 } // end function operator<<
17
```

Fig. 11.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 1 of 2.)

```
18  // overloaded stream extraction operator; cannot be
19  // a member function if we would like to invoke it with
20  // cin >> somePhoneNumber;
21  istream &operator>>( istream &input, PhoneNumber &number )
22  {
23     input.ignore(); // skip (
24     input >> setw( 3 ) >> number.areaCode; // input area code
25     input.ignore( 2 ); // skip ) and space
26     input >> setw( 3 ) >> number.exchange; // input exchange
27     input.ignore(); // skip dash (-)
28     input >> setw( 4 ) >> number.line; // input line
29     return input; // enables cin >> a >> b >> c;
30  } // end function operator>>
```

Fig. 11.4 | Overloaded stream insertion and stream extraction operators for class PhoneNumber. (Part 2 of 2.)

```
1  // Fig. 11.5: fig11_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  int main()
9  {
10    PhoneNumber phone; // create object phone
11
12    cout << "Enter phone number in the form (123) 456-7890:" << endl;
13
14    // cin >> phone invokes operator>> by implicitly issuing
15    // the global function call operator>>( cin, phone )
16    cin >> phone;
17
18    cout << "The phone number entered was: ";
19
20    // cout << phone invokes operator<< by implicitly issuing
21    // the global function call operator<<( cout, phone )
22    cout << phone << endl;
23 } // end main
```


Fig. 11.5 | Overloaded stream insertion and stream extraction operators. (Part 1 of 2.)

11.8 Dynamic Memory Management

- Dynamic allocate and deallocate memory
  - C++ provide strong memory management tools

- Avoid "memory leak"
  - Allocate memory without releasing it
    - You "new" memory blocks but forget "delete" them

- new

```
Time *timePtr = new time;
```

- delete

```
delete timePtr;
```

Example: new, delete

- Initializing

```
double *ptr = new double (3.1415926);
Time *timePtr = new Time (12,45,0);

delete timePtr;
```

- Array []

```
int *gradesArray = new int[10];
delete [] gradesArray;
```

11.9 Example: a new Array class

- We have learned the built-in array in C++ in chapter 6
- We could improve it with a new Array class

[Concept] L-value v.s. R-Value

- x =3;
- 3 =x;

- L-Value = R-Value;
  - Lvalue: space to store new data
  - Rvalue: data to be stored

- Need to define operators for lvalue and rvalue

```
int &Array::operator[] (int subscript)      //lvalue
int Array::operator[] (int subscript)       //rvalue
```

```
1  // Fig. 11.6: Array.h
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Array
10 {
11    friend ostream &operator<<( ostream &, const Array & );
12    friend istream &operator>>( istream &, Array & );
13 public:
14    Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    int getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
```

Fig. 11.6 | Array class definition with overloaded operators. (Part 1 of 2.)

```
22       // inequality operator; returns opposite of == operator
23       bool operator!=( const Array &right ) const
24       {
25          return ! ( *this == right ); // invokes Array::operator==
26       } // end function operator!=
27
28       // subscript operator for non-const objects returns modifiable lva
29       int &operator[]( int );
30
31       // subscript operator for const objects returns rvalue
32       int operator[]( int ) const;
33    private:
34       int size; // pointer-based array size
35       int *ptr; // pointer to first element of pointer-based array
36    }; // end class Array
37
38    #endif
```

Fig. 11.6 | `Array` class definition with overloaded operators. (Part 2 of 2.)

```
1  // Fig 11.7: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // exit function prototype
6  #include "Array.h" // Array class definition
7  using namespace std;
8
9  // default constructor for class Array (default size 10)
10 Array::Array( int arraySize )
11 {
12    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
13    ptr = new int[ size ]; // create space for pointer-based array
14
15    for ( int i = 0; i < size; i++ )
16       ptr[ i ] = 0; // set pointer-based array element
17 } // end Array default constructor
18
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 1 of 7.)

```
19  // copy constructor for class Array;
20  // must receive a reference to prevent infinite recursion
21  Array::Array( const Array &arrayToCopy )
22     : size( arrayToCopy.size )
23  {
24     ptr = new int[ size ]; // create space for pointer-based array
25
26     for ( int i = 0; i < size; i++ )
27        ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
28  } // end Array copy constructor
29
30  // destructor for class Array
31  Array::~Array()
32  {
33     delete [] ptr; // release pointer-based array space
34  } // end destructor
35
36  // return number of elements of Array
37  int Array::getSize() const
38  {
39     return size; // number of elements in Array
40  } // end function getSize
41
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 2 of 7.)

```
42  // overloaded assignment operator;
43  // const return avoids: ( a1 = a2 ) = a3
44  const Array &Array::operator=( const Array &right )
45  {
46     if ( &right != this ) // avoid self-assignment
47     {
48        // for Arrays of different sizes, deallocate original
49        // left-side array, then allocate new left-side array
50        if ( size != right.size )
51        {
52           delete [] ptr; // release space
53           size = right.size; // resize this object
54           ptr = new int[ size ]; // create space for array copy
55        } // end inner if
56
57        for ( int i = 0; i < size; i++ )
58           ptr[ i ] = right.ptr[ i ]; // copy array into object
59     } // end outer if
60
61     return *this; // enables x = y = z, for example
62  } // end function operator=
63
```

Fig. 11.7 | Array class member- and friend-function definitions. (Part 3 of 7.)

```
64   // determine if two Arrays are equal and
65   // return true, otherwise return false
66   bool Array::operator==( const Array &right ) const
67   {
68      if ( size != right.size )
69         return false; // arrays of different number of elements
70
71      for ( int i = 0; i < size; i++ )
72         if ( ptr[ i ] != right.ptr[ i ] )
73            return false; // Array contents are not equal
74
75      return true; // Arrays are equal
76   } // end function operator==
77
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 4 of 7.)

```
78  // overloaded subscript operator for non-const Arrays;
79  // reference return creates a modifiable lvalue
80  int &Array::operator[]( int subscript )
81  {
82     // check for subscript out-of-range error
83     if ( subscript < 0 || subscript >= size )
84     {
85        cerr << "\nError: Subscript " << subscript
86           << " out of range" << endl;
87        exit( 1 ); // terminate program; subscript out of range
88     } // end if
89
90     return ptr[ subscript ]; // reference return
91  } // end function operator[]
92
93  // overloaded subscript operator for const Arrays
94  // const reference return creates an rvalue
95  int Array::operator[]( int subscript ) const
96  {
97     // check for subscript out-of-range error
98     if ( subscript < 0 || subscript >= size )
99     {
100       cerr << "\nError: Subscript " << subscript
101          << " out of range" << endl;
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 5 of 7.)

```
102         exit( 1 ); // terminate program; subscript out of range
103      } // end if
104
105      return ptr[ subscript ]; // returns copy of this element
106   } // end function operator[]
107
108   // overloaded input operator for class Array;
109   // inputs values for entire Array
110   istream &operator>>( istream &input, Array &a )
111   {
112      for ( int i = 0; i < a.size; i++ )
113         input >> a.ptr[ i ];
114
115      return input; // enables cin >> x >> y;
116   } // end function
117
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 6 of 7.)

```
118 // overloaded output operator for class Array
119 ostream &operator<<( ostream &output, const Array &a )
120 {
121    int i;
122
123    // output private ptr-based array
124    for ( i = 0; i < a.size; i++ )
125    {
126       output << setw( 12 ) << a.ptr[ i ];
127
128       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
129          output << endl;
130    } // end for
131
132    if ( i % 4 != 0 ) // end last line of output
133       output << endl;
134
135    return output; // enables cout << x << y;
136 } // end function operator<<
```

Fig. 11.7 | `Array` class member- and `friend`-function definitions. (Part 7 of 7.)

```
1  // Fig. 11.8: fig11_08.cpp
2  // Array class test program.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  int main()
8  {
9     Array integers1( 7 ); // seven-element Array
10    Array integers2; // 10-element Array by default
11
12    // print integers1 size and contents
13    cout << "Size of Array integers1 is "
14       << integers1.getSize()
15       << "\nArray after initialization:\n" << integers1;
16
17    // print integers2 size and contents
18    cout << "\nSize of Array integers2 is "
19       << integers2.getSize()
20       << "\nArray after initialization:\n" << integers2;
21
22    // input and print integers1 and integers2
23    cout << "\nEnter 17 integers:" << endl;
24    cin >> integers1 >> integers2;
```

Fig. 11.8 | Array class test program. (Part 1 of 6.)

```
25
26      cout << "\nAfter input, the Arrays contain:\n"
27         << "integers1:\n" << integers1
28         << "integers2:\n" << integers2;
29
30      // use overloaded inequality (!=) operator
31      cout << "\nEvaluating: integers1 != integers2" << endl;
32
33      if ( integers1 != integers2 )
34         cout << "integers1 and integers2 are not equal" << endl;
35
36      // create Array integers3 using integers1 as an
37      // initializer; print size and contents
38      Array integers3( integers1 ); // invokes copy constructor
39
40      cout << "\nSize of Array integers3 is "
41         << integers3.getSize()
42         << "\nArray after initialization:\n" << integers3;
43
44      // use overloaded assignment (=) operator
45      cout << "\nAssigning integers2 to integers1:" << endl;
46      integers1 = integers2; // note target Array is smaller
47
```

Fig. 11.8 | Array class test program. (Part 2 of 6.)

```
48       cout << "integers1:\n" << integers1
49          << "integers2:\n" << integers2;
50
51       // use overloaded equality (==) operator
52       cout << "\nEvaluating: integers1 == integers2" << endl;
53
54       if ( integers1 == integers2 )
55          cout << "integers1 and integers2 are equal" << endl;
56
57       // use overloaded subscript operator to create rvalue
58       cout << "\nintegers1[5] is " << integers1[ 5 ];
59
60       // use overloaded subscript operator to create lvalue
61       cout << "\n\nAssigning 1000 to integers1[5]" << endl;
62       integers1[ 5 ] = 1000;
63       cout << "integers1:\n" << integers1;
64
65       // attempt to use out-of-range subscript
66       cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
67       integers1[ 15 ] = 1000; // ERROR: out of range
68    } // end main
```

Fig. 11.8 | `Array` class test program. (Part 3 of 6.)

11.13: example Date class with ++

- Prefix increment

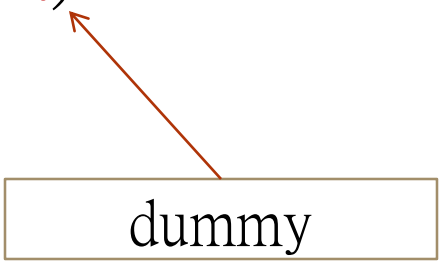
  Date & operator ++ ()

  ++day

- Postfix increment

  Date operator ++ (int)

  day++



```
1  // Fig. 11.9: Date.h
2  // Date class definition with overloaded increment operators.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    const Date &operator+=( int ); // add days, modify object
18    static bool leapYear( int ); // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    int month;
22    int day;
23    int year;
```


Fig. 11.9 | `Date` class definition with overloaded increment operators. (Part 1 of 2.)

```
24
25      static const int days[]; // array of days per month
26      void helpIncrement(); // utility function for incrementing date
27   }; // end class Date
28
29   #endif
```

Fig. 11.9 | `Date` class definition with overloaded increment operators. (Part 2 of 2.)

```
1  // Fig. 11.10: Date.cpp
2  // Date class member- and friend-function definitions.
3  #include <iostream>
4  #include <string>
5  #include "Date.h"
6  using namespace std;
7
8  // initialize static member; one classwide copy
9  const int Date::days[] =
10    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
11
12 // Date constructor
13 Date::Date( int m, int d, int y )
14 {
15    setDate( m, d, y );
16 } // end Date constructor
17
18 // set month, day and year
19 void Date::setDate( int mm, int dd, int yy )
20 {
21    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
22    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
23
```


Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 1 of 5.)

```
24       // test for a leap year
25       if ( month == 2 && leapYear( year ) )
26          day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27       else
28          day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29    } // end function setDate
30
31    // overloaded prefix increment operator
32    Date &Date::operator++()
33    {
34       helpIncrement(); // increment date
35       return *this; // reference return to create an lvalue
36    } // end function operator++
37
38    // overloaded postfix increment operator; note that the
39    // dummy integer parameter does not have a parameter name
40    Date Date::operator++( int )
41    {
42       Date temp = *this; // hold current state of object
43       helpIncrement();
44
45       // return unincremented, saved, temporary object
46       return temp; // value return; not a reference return
47    } // end function operator++
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 2 of 5.)

```
48
49  // add specified number of days to date
50  const Date &Date::operator+=( int additionalDays )
51  {
52     for ( int i = 0; i < additionalDays; i++ )
53        helpIncrement();
54
55     return *this; // enables cascading
56  } // end function operator+=
57
58  // if the year is a leap year, return true; otherwise, return false
59  bool Date::leapYear( int testYear )
60  {
61     if ( testYear % 400 == 0 ||
62        ( testYear % 100 != 0 && testYear % 4 == 0 ) )
63        return true; // a leap year
64     else
65        return false; // not a leap year
66  } // end function leapYear
67
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 3 of 5.)

```
68  // determine whether the day is the last day of the month
69  bool Date::endOfMonth( int testDay ) const
70  {
71     if ( month == 2 && leapYear( year ) )
72        return testDay == 29; // last day of Feb. in leap year
73     else
74        return testDay == days[ month ];
75  } // end function endOfMonth
76
77  // function to help increment the date
78  void Date::helpIncrement()
79  {
80     // day is not end of month
81     if ( !endOfMonth( day ) )
82        day++; // increment day
83     else
84        if ( month < 12 ) // day is end of month and month < 12
85        {
86           month++; // increment month
87           day = 1; // first day of new month
88        } // end if
89        else // last day of year
90        {
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 4 of 5.)

```
91             year++; // increment year
92             month = 1; // first month of new year
93             day = 1; // first day of new month
94          } // end else
95    } // end function helpIncrement
96
97    // overloaded output operator
98    ostream &operator<<( ostream &output, const Date &d )
99    {
100      static string monthName[ 13 ] = { "", "January", "February",
101         "March", "April", "May", "June", "July", "August",
102         "September", "October", "November", "December" };
103      output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
104      return output; // enables cascading
105   } // end function operator<<
```

Fig. 11.10 | `Date` class member- and `friend`-function definitions. (Part 5 of 5.)

```
1  // Fig. 11.11: fig11_11.cpp
2  // Date class test program.
3  #include <iostream>
4  #include "Date.h" // Date class definition
5  using namespace std;
6
7  int main()
8  {
9     Date d1; // defaults to January 1, 1900
10    Date d2( 12, 27, 1992 ); // December 27, 1992
11    Date d3( 0, 99, 8045 ); // invalid date
12
13    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
14    cout << "\n\nd2 += 7 is " << ( d2 += 7 );
15
16    d3.setDate( 2, 28, 1992 );
17    cout << "\n\n  d3 is " << d3;
18    cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
19
20    Date d4( 7, 13, 2002 );
21
22    cout << "\n\nTesting the prefix increment operator:\n"
23       << "  d4 is " << d4 << endl;
```


Fig. 11.11 | Date class test program. (Part 1 of 3.)

```
24       cout << "++d4 is " << ++d4 << endl;
25       cout << "  d4 is " << d4;
26
27       cout << "\n\nTesting the postfix increment operator:\n"
28          << "  d4 is " << d4 << endl;
29       cout << "d4++ is " << d4++ << endl;
30       cout << "  d4 is " << d4 << endl;
31    } // end main
```

Fig. 11.11 | `Date` class test program. (Part 2 of 3.)

11.14 explicit Constructors

- Implicit conversion may lead to unintended (undesirable) results

- Syntax

  explicit Constructor()

- Avoid implicit conversion

```
1  // Fig. 11.14: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Array
10 {
11    friend ostream &operator<<( ostream &, const Array & );
12    friend istream &operator>>( istream &, Array & );
13 public:
14    explicit Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    int getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
```

Fig. 11.14 | `Array` class definition with `explicit` constructor. (Part 1 of 2.)

```
22       // inequality operator; returns opposite of == operator
23       bool operator!=( const Array &right ) const
24       {
25          return ! ( *this == right ); // invokes Array::operator==
26       } // end function operator!=
27
28       // subscript operator for non-const objects returns lvalue
29       int &operator[]( int );
30
31       // subscript operator for const objects returns rvalue
32       const int &operator[]( int ) const;
33    private:
34       int size; // pointer-based array size
35       int *ptr; // pointer to first element of pointer-based array
36    }; // end class Array
37
38    #endif
```

Fig. 11.14 | `Array` class definition with `explicit` constructor. (Part 2 of 2.)

```
1  // Fig. 11.13: Fig11_13.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void outputArray( const Array & ); // prototype
8
9  int main()
10 {
11    Array integers1( 7 ); // 7-element array
12    outputArray( integers1 ); // output Array integers1
13    outputArray( 3 ); // convert 3 to an Array and output Array's contents
14 } // end main
15
16 // print Array contents
17 void outputArray( const Array &arrayToOutput )
18 {
19    cout << "The Array received has " << arrayToOutput.getSize()
20       << " elements. The contents are:\n" << arrayToOutput << endl;
21 } // end outputArray
```

Fig. 11.13 | Single-argument constructors and implicit conversions. (Part 1 of 2.)

11.15 Proxy Classes

- [review] OOP Advantages
  - separating interface from implementation
  - hiding implementation details
- Can we improve it?
  - class definition (*.h) still reveals private data member
    - We could hide this with "proxy class"
- Syntax: forward class declaration
  - No need to #include "ImplementClass.h"

```
class ClassNameToUseProxyClass;
```

```
1  // Fig. 11.16: Implementation.h
2  // Implementation class definition.
3
4  class Implementation
5  {
6  public:
7     // constructor
8     Implementation( int v )
9        : value( v ) // initialize value with v
10    {
11       // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17       value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23       return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

```
1  // Fig. 11.17: Interface.h
2  // Proxy class Interface definition.
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 1
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

Fig. 11.17 | Proxy class `Interface` definition.

```
1  // Fig. 11.18: Interface.cpp
2  // Implementation of class Interface--client receives this file only
3  // as precompiled object code, keeping the implementation hidden.
4  #include "Interface.h" // Interface class definition
5  #include "Implementation.h" // Implementation class definition
6
7  // constructor
8  Interface::Interface( int v )
9     : ptr ( new Implementation( v ) ) // initialize ptr to point to
10 {                                     // a new Implementation object
11    // empty body
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17    ptr->setValue( v );
18 } // end function setValue
19
20 // call Implementation's getValue function
21 int Interface::getValue() const
22 {
23    return ptr->getValue();
24 } // end function getValue
```


Fig. 11.18 | Interface class member-function definitions. (Part 1 of 2.)

```
25
26   // destructor
27   Interface::~Interface()
28   {
29      delete ptr;
30   } // end ~Interface destructor
```

Fig. 11.18 | `Interface` class member-function definitions. (Part 2 of 2.)

Summary: Chapter 11

- operator overloading
  - +
  - =
  - ==, !=
  - ++x, x++
  - []
- new, delete
- proxy class