Chapter 11
Operator Overloading

Yu-Shuen Wang, CS, NCTU

11.1 Introduction

```
int main()    {
      cout << "5 times 2 is “
            << (5 << 1) << endl
            << "20 divided by 4 is “
            << (20 >> 2) << endl;
}
```

- operator overloading.
- One example of an overloaded operator built into C++ is <<, which is used both as the stream insertion operator and as the bitwise left-shift operator..
- C++ overloads the addition operator (+) and the subtraction operator (-). These operators perform differently, depending on their context in integer, floating-point and pointer arithmetic.

11.2 Fundamentals of Operator Overloading

- C++ enables you to overload most operators—the compiler generates the appropriate code based on the context.
- You can use operators with user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

Software Engineering Observation 11.1

Operator overloading contributes to C++'s extensibility—one of the language's most appealing attributes.

Good Programming Practice 11.1

Use operator overloading when it makes a program clearer than accomplishing the same operations with function calls.

Good Programming Practice 11.2

Overloaded operators should mimic the functionality of their built-in counterparts—for example, the + operator should be overloaded to perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

11.2 Fundamentals of Operator Overloading (cont.)

- An operator is overloaded by writing a non-`static` member function definition or global function definition as you normally would.
  - For example, the function name `operator+` would be used to overload the addition operator (+).
- When operators are overloaded as member functions, they must be non-`static`, because they must be called on an object of the class and operate on that object.

11.2 Fundamentals of Operator Overloading (cont.)

- The assignment operator (=) may be used with every class to perform memberwise assignment of the class's data members.
  - Dangerous for classes with pointer members
- The address (&) and comma (,) operators may also be used with objects of any class without overloading.
  - The address operator returns a pointer to the object.
  - The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

```
i = (a, b);        // i = b
i = a, b;          // i = a
```

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.

```
class vector2
{
 public:
  float x, y;            // Members

  // Constructors
  vector2() {};
  vector2(float inX, float inY);

  // Operators
  vector2 & operator + (const vector2 &v);
  vector2 & operator - (const vector2 &v);
  vector2 & operator * (float f);
  vector2 & operator / (float f);

  // Methods
  void set(float xIn, float yIn);
  vector2 & normalize();
};
```

11.3 Restrictions on Operator Overloading (cont.)

- The precedence of an operator cannot be changed by overloading.
- The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.
- It isn't possible to change the "arity" of an operator (i.e., the number of operands an operator takes):
- Operators &, *, + and - all have both unary and binary versions; these unary and binary versions can each be overloaded.

11.3 Restrictions on Operator Overloading (cont.)

- It isn't possible to create new operators; only existing operators can be overloaded.
- You could overload an existing operator to perform exponentiation.

11.3 Restrictions on Operator Overloading (cont.)

- The meaning of how an operator works on fundamental types cannot be changed by operator overloading.
  - You cannot, for example, change the meaning of how + adds two integers.
- 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.

Software Engineering Observation 11.2

At least one argument of an operator function must be an object or reference of a user-defined type. This prevents you from changing how operators work on fundamental types.

11.3 Restrictions on Operator Overloading (cont.)

- Overloading an assignment operator and an addition operator to allow statements like
  - `object2 = object2 + object1;`
- does not imply that the += operator is also overloaded to allow statements such as
  - `object2 += object1;`
- Such behavior can be achieved only by explicitly overloading operator += for that class.

Common Programming Error 11.4

Assuming that overloading an operator such as + overloads related operators such as += or that overloading == overloads a related operator like != can lead to errors. Operators can be overloaded only explicitly; there is no implicit overloading.

11.4 Operator Functions as Class Members vs. Global Functions

- Operator functions can be member functions or global functions.
  - Global functions are often made `friend`s for performance reasons.
- Member functions use the `this` pointer implicitly to obtain one of their class object.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be declared as a class member.
- For the other operators, the operator overloading functions can be class members or standalone functions.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function.
- A global operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- Operator member functions of a specific class are called only when the left operand of a binary operator is specifically an object of that class, or when the single operand of a unary operator is an object of that class.
- The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`.

11.4 Operator Functions as Class Members vs. Global Functions (cont.)

- To use the operator in this manner where the right operand is a user-defined class, it must be overloaded as a global function.
- Similarly, the overloaded stream extraction operator (>>) is used it must be a global function.
- Each of these overloaded operator functions may require access to the `private` data members of the class object, so these overloaded operator functions can be made `friend` functions of the class.

Performance Tip 11.1

It's possible to overload an operator as a global, non-`friend` *function, but such a function requiring access to a class's* `private` *or* `protected` *data would need to use* set *or* get *functions provided in that class's* `public` *interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.*

11.5 Overloading Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<.
- The C++ class libraries overload these operators to process each fundamental type, including pointers and C-style `char *` strings.
- You can also overload these operators to perform input and output for your own types.

```
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
```

global friend functions !!

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.)

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

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

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- Function `operator>>` returns `istream` reference `input` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The stream insertion operator function takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference.
- When the compiler sees the expression
  - `cout << phone`

  it generates the global function call
  - `operator<<( cout, phone );`

Error-Prevention Tip 11.1

Returning a reference from an overloaded `<<` *or* `>>` *operator function is typically successful because* `cout`, `cin` *and most stream objects are global, or at least long-lived. Returning a reference to an automatic variable or other temporary object is dangerous—this can create "dangling references" to nonexisting objects.*

11.5 Overloading Stream Insertion and Stream Extraction Operators (cont.)

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as global, `friend` functions
  - global functions because the object of class `PhoneNumber` is the operator's right operand.

Software Engineering Observation 11.3

New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s extensibility.

11.6 Overloading Unary Operators

- A unary operator for a class can be overloaded as a non-`static` member function with no arguments or as a global function with one argument that must be an object (or a reference to an object).
- A unary operator such as `!` may be overloaded as a global function with one parameter in two different ways—either with a parameter that is an object, or with a parameter that is a reference to an object.

11.7 Overloading Binary Operators

- A binary operator can be overloaded as a non-`static` member function with one parameter or as a global function with two parameters (one of those parameters must be either a class object or a reference to a class object).
- As a global function, binary operator must take two arguments—one of which must be an object (or a reference to an object) of the class.

11.8 Dynamic Memory Management

- C++ enables you to control the allocation and deallocation of memory in a program for objects and for arrays of any built-in or user-defined type.
  - Known as dynamic memory management; performed with `new` and `delete`.
- You can use the `new` operator to dynamically allocate (i.e., reserve) the exact amount of memory.
- Once memory is allocated, you can access it via the pointer that operator `new` returns.
- You can return memory to the free store by using the `delete` operator to deallocate it.

11.8 Dynamic Memory Management (cont.)

- The new operator allocates storage of the proper size for an object of the specified type, calls the constructor to initialize the object and returns a pointer to the type specified.
- If new is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception."
  - Chapter 16, Exception Handling, discusses how to deal with new failures.

11.8 Dynamic Memory Management (cont.)

- To destroy a dynamically allocated object and free the space for the object, use the `delete` operator as follows:
  - `delete ptr;`
- This statement first calls the destructor for the object to which `ptr` points, then deallocates the memory associated with the object.

Common Programming Error 11.5

*Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "**memory leak**."*

11.8 Dynamic Memory Management (cont.)

- You can provide an initializer for a newly created fundamental-type variable, as in
  - `double *ptr = new double( 3.14159 );`
- You can also use the `new` operator to allocate arrays dynamically.
- For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:
  - `int *gradesArray = new int[ 10 ];`

11.8 Dynamic Memory Management (cont.)

- A dynamically allocated array's size can be specified using any non-negative integral expression that can be evaluated at execution time.
- Also, when allocating an array of objects dynamically, you cannot pass arguments to each object's constructor—each object is initialized by its default constructor.

11.8 Dynamic Memory Management (cont.)

- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`
- If the pointer points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory.
- Using `delete` on a null pointer (i.e., a pointer with the value 0) has no effect.

Common Programming Error 11.6

Using `delete` *instead of* `delete []` *for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator* `delete []`*. Similarly, always delete memory allocated as an individual element with operator* `delete`*—the result of deleting a single object with operator* `delete []` *is undefined.*

11.9 Case Study: `Array` Class (cont.)

- C++ provides the means to implement more robust array capabilities via classes and operator overloading.
- C++ Standard Library class template `vector` provides many of these capabilities as well.

11.9 Case Study: `Array` Class (cont.)

- In the following example, we create a powerful `Array` class:
  - Performs range checking.
  - Allows one array object to be assigned to another with the assignment operator.
  - Objects know their own size.
  - Input or output entire arrays with the stream extraction and stream insertion operators, respectively.
  - Can compare `Array`s with the equality operators `==` and `!=`.

```
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 lvalue
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.)

```
Size of Array integers1 is 7
Array after initialization:
          0          0          0          0
          0          0          0

Size of Array integers2 is 10
Array after initialization:
          0          0          0          0
          0          0          0          0
          0          0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
          1          2          3          4
          5          6          7
integers2:
          8          9         10         11
         12         13         14         15
         16         17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Fig. 11.8 | `Array` class test program. (Part 4 of 6.)

```
Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17
```

Fig. 11.8 | `Array` class test program. (Part 5 of 6.)

```
Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

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

11.9 Case Study: `Array` Class (cont.)

- The `Array` copy constructor copies the elements of one `Array` into another.
- The copy constructor can also be invoked by
  - `Array integers3 = integers1;`
- The equal sign in the preceding statement is *not* the assignment operator.
- When an equal sign appears in the declaration of an object, it invokes a constructor for that object.
- This form can be used to pass only a single argument to a constructor.

```
Array( const Array & ); // copy constructor
```

```
const Array &operator=( const Array & ); // assignment operator
```

11.9 Case Study: `Array` Class (cont.)

- The array subscript operator `[]` is not restricted for use only with arrays; it also can be used, for example, to select elements from other kinds of container classes, such as linked lists, strings and dictionaries.
- Also, when `operator[]` functions are defined, subscripts no longer have to be integers—characters, strings, floats or even objects of user-defined classes also could be used.

11.9 Case Study: `Array` Class (cont.)

- When the compiler sees an expression like `cout << arrayObject` and `cin >> arrayObject`, it invokes global function call
  - `operator<<( cout, arrayObject )`
  - `operator>>( cin, arrayObject )`
- These stream insertion and stream extraction operator functions cannot be members of class `Array`, because the `Array` object is always mentioned on the right side of the stream insertion operator and the stream extraction operator.

Software Engineering Observation 11.4

The argument to a copy constructor should be a `const` *reference to allow a* `const` *object to be copied.*

```
Array( const Array & ); // copy constructor
```

Common Programming Error 11.7

A copy constructor must *receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!*

Error-Prevention Tip 11.2

If after deleting dynamically allocated memory, the pointer will continue to exist in memory, set the pointer's value to 0 to indicate that the pointer no longer points to memory in the free store. By setting the pointer to 0, the program loses access to that free-store space, which could be reallocated for a different purpose. If you do not set the pointer to 0, your code could inadvertently access the re-allocated memory, causing subtle, nonrepeatable logic errors.

```
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor
```

11.9 Case Study: Array Class (cont.)

- When the compiler sees the expression `integers1 = integers2`, the compiler invokes member function `operator=` with the call
  - `integers1.operator=( integers2 )`
- The member function returns the current as a constant reference; this enables cascaded `Array` assignments such as `x = y = z`, but prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array`.

```
const Array &operator=( const Array & ); // assignment operator
```

Software Engineering Observation 11.5

A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

Common Programming Error 11.9

Not providing an overloaded assignment operator and a copy constructor for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

11.9 Case Study: `Array` Class (cont.)

- When the compiler sees the expression `integers1 == integers2`, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Writing `operator!=` in this manner enables you to reuse `operator==`, which reduces the amount of code that must be written in the class.

11.9 Case Study: `Array` Class (cont.)

- When the compiler sees the expression `integers1[5]`, it invokes the appropriate overloaded `operator[]` member function by generating the call
  - `integers1.operator[]( 5 )`

11.10 Converting between Types

- It's often necessary to convert data from one type to another type.
- The compiler knows how to perform certain conversions among fundamental types.
- You can use cast operators to force conversions among fundamental types.
- The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so you must specify how to do this.

11.10 Converting between Types (cont.)

- A conversion operator (also called a cast operator) can be used to convert an object of one class into an object of another class or into an object of a fundamental type.

- A conversion operator must be a non-`static` member function.

11.10 Converting between Types (cont.)

- The function prototype declares an overloaded cast operator function for converting an object of user-defined type `A` into a temporary `char *` object.
  - `A::operator char *() const;`

- The operator function is declared `const` because it does not modify the original object.

11.10 Converting between Types (cont.)

- An overloaded cast operator function does not specify a return type—the return type is the type to which the object is being converted.

- If `s` is a class object, when the compiler sees the expression `static_cast< char * >( s )`, the compiler generates the call
  - `s.operator char *()`

11.11 Building a `String` Class

- The `const char *` conversion constructor means supplying an overloaded assignment operator for assigning character strings to `String` objects is not needed.
- When the compiler encounters the statement
  - `myString = "hello";`

  it invokes the conversion constructor to create a temporary `String` object containing the character string `"hello"`; then class `String`'s overloaded assignment operator is invoked to assign the temporary `String` object to `String` object `myString`.

11.11 Building a `String` Class (cont.)

- The `String` conversion constructor could be invoked in a declaration such as
  - `String s1( "happy" );`
- It can also be invoked when you pass a C string to a function that expects a `String` argument or when you return a C string from a function with a `String` return type.

Software Engineering Observation 11.8

When a conversion constructor is used to perform an implicit conversion, C++ can apply only one implicit constructor call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator. The compiler will not satisfy an overloaded operator's needs by performing a series of implicit, user-defined conversions.

11.11 Building a `String` Class (cont.)

- When the compiler sees the expression `!string1`, it generates the function call
  - `string1.operator!()`

- This function returns `true` if the `String`'s length is equal to zero, and `false` otherwise.

11.12 Overloading ++ and --

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.
- The prefix versions are overloaded exactly as any other prefix unary operator would be.

11.12 Overloading ++ and -- (cont.)

- When the compiler sees the pre-incrementing expression ++d1, the compiler generates the member-function call
  - `d1.operator++()`

- The prototype for this operator function would be
  - `Date &operator++();`

11.12 Overloading ++ and -- (cont.)

- If the prefix increment operator is implemented as a global function, when the compiler sees the expression ++d1, the compiler generates the function call
  - `operator++( d1 )`

- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`

11.12 Overloading ++ and -- (cont.)

- When the compiler sees the post-incrementing expression `d1++`, it generates the member-function call
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables distinguishability.

11.12 Overloading ++ and -- (cont.)

- If the postfix increment is implemented as a global function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`

- The postfix increment operator returns `Date` objects by value, whereas the prefix increment operator returns `Date` objects by reference.

Performance Tip 11.2

The extra object that is created by the postfix increment (or decrement) operator can result in a significant performance problem—especially when the operator is used in a loop. For this reason, you should use the postfix increment (or decrement) operator only when the logic of the program requires postincrementing (or postdecrementing).

```
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.)

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

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

11.14 Standard Library Class `string`

- Figure 11.12 demonstrates many of class `string`'s overloaded operators, it's conversion constructor for C strings and several other useful member functions, including `empty`, `substr` and `at`.

- Function `empty` determines whether a `string` is empty, function `substr` returns a `string` that represents a portion of an existing `string` and function `at` returns the character at a specific index in a `string` (after checking that the index is in range).

```
1  // Fig. 11.12: fig11_12.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string s1( "happy" );
10    string s2( " birthday" );
11    string s3;
12
13    // test overloaded equality and relational operators
14    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
15       << "\"; s3 is \"" << s3 << '\"'
16       << "\n\nThe results of comparing s2 and s1:"
17       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
18       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
19       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
20       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
21       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
22       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
23
```

Fig. 11.12 | Standard Library class `string`. (Part 1 of 5.)

```
24      // test string member-function empty
25      cout << "\n\nTesting s3.empty():" << endl;
26
27      if ( s3.empty() )
28      {
29         cout << "s3 is empty; assigning s1 to s3;" << endl;
30         s3 = s1; // assign s1 to s3
31         cout << "s3 is \"" << s3 << "\"";
32      } // end if
33
34      // test overloaded string concatenation operator
35      cout << "\n\ns1 += s2 yields s1 = ";
36      s1 += s2; // test overloaded concatenation
37      cout << s1;
38
39      // test overloaded string concatenation operator with C-style string
40      cout << "\n\ns1 += \" to you\" yields" << endl;
41      s1 += " to you";
42      cout << "s1 = " << s1 << "\n\n";
43
44      // test string member function substr
45      cout << "The substring of s1 starting at location 0 for\n"
46         << "14 characters, s1.substr(0, 14), is:\n"
47         << s1.substr( 0, 14 ) << "\n\n";
```

Fig. 11.12 | Standard Library class `string`. (Part 2 of 5.)

```
48
49      // test substr "to-end-of-string" option
50      cout << "The substring of s1 starting at\n"
51         << "location 15, s1.substr(15), is:\n"
52         << s1.substr( 15 ) << endl;
53
54      // test copy constructor
55      string s4( s1 );
56      cout << "\ns4 = " << s4 << "\n\n";
57
58      // test overloaded assignment (=) operator with self-assignment
59      cout << "assigning s4 to s4" << endl;
60      s4 = s4;
61      cout << "s4 = " << s4 << endl;
62
63      // test using overloaded subscript operator to create lvalue
64      s1[ 0 ] = 'H';
65      s1[ 6 ] = 'B';
66      cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
67         << s1 << "\n\n";
68
69      // test subscript out of range with string member function "at"
70      cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
71      s1.at( 30 ) = 'd'; // ERROR: subscript out of range
72   } // end main
```

Fig. 11.12 | Standard Library class `string`. (Part 3 of 5.)

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday
```

Fig. 11.12 | Standard Library class `string`. (Part 4 of 5.)

```
The substring of s1 starting at
location 15, s1.substr(15), is:
to you

s4 = happy birthday to you

assigning s4 to s4
s4 = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Fig. 11.12 | Standard Library class `string`. (Part 5 of 5.)

11.14 Standard Library Class `string` (cont.)

- Class `string`'s overloaded equality and relational operators perform lexicographical comparisons using the numerical values of the characters in each `string`.
- Class `string` provides member function `empty` to determine whether a `string` is empty.
- Class `string`'s overloaded += operator performs string concatenation.

11.14 Standard Library Class `string` (cont.)

- Class `string`'s member function `substr` returns a portion of a string as a `string` object.
  - When the second argument is not specified, `substr` returns the remainder of the `string` on which it's called.

- Class `string`'s overloaded `[]` operator can create *lvalues that enable new characters to replace existing characters in a `string`.*
  - *Class `string`'s overloaded `[]` operator does not perform any bounds checking.*

11.14 Standard Library Class `string` (cont.)

- Class `string` provides bounds checking in its member function `at`, which "throws an exception" if its argument is an invalid subscript.
  - By default, this causes a C++ program to terminate and display a system-specific error message.
  - If the subscript is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue* (i.e., a `const` reference), depending on the context in which the call appears.

11.15 `explicit` Constructors

- Any single-argument constructor can be used by the compiler to perform an implicit conversion—the type received by the constructor is converted to an object of the class in which the constructor is defined.
- The conversion is automatic and you need not use a cast operator.
- In some situations, implicit conversions are undesirable or error-prone.

Common Programming Error 11.10

Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in ambiguous expressions that generate compilation errors or result in execution-time logic errors.

```
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.)

```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

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

11.16 `explicit` Constructors (cont.)

- C++ provides the keyword `explicit` to suppress implicit conversions via conversion constructors when such conversions should not be allowed.

- A constructor that is declared `explicit` cannot be used in an 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. 11.15: Fig11_15.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    outputArray( Array( 3 ) ); // explicit single-argument constructor call
15 }  // end main
16
17 // print array contents
18 void outputArray( const Array &arrayToOutput )
19 {
20    cout << "The Array received has " << arrayToOutput.getSize()
21       << " elements. The contents are:\n" << arrayToOutput << endl;
22 } // end outputArray
```

Fig. 11.15 | Demonstrating an `explicit` constructor. (Part 1 of 2.)

```
c:\cpphtp7_examples\ch11\fig11_14_15\fig11_15.cpp(13) : error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
        Reason: cannot convert from 'int' to 'const Array'
        Constructor for class 'Array' is declared 'explicit'
```

Fig. 11.15 | Demonstrating an `explicit` constructor. (Part 2 of 2.)

Common Programming Error 11.11

Attempting to invoke an `explicit` *constructor for an implicit conversion is a compilation error.*

```
outputArray( 3 ); // convert 3 to an Array and output Array's contents
```

Error-Prevention Tip 11.3

Use the `explicit` *keyword on single-argument constructors that should not be used by the compiler to perform implicit conversions.*

```
outputArray( 3 );
```

11.17 Proxy Classes

- Recall that two of the fundamental principles of good software engineering are separating interface from implementation and hiding implementation details.
- We achieve these goals by defining a class in a header file and implementing its member functions in a separate implementation file.
- Header files do contain a portion of a class's implementation and hints about others.
  - A class's `private` members are listed in the class. It potentially exposes proprietary information to clients of the class.

11.17 Proxy Classes (cont.)

- A proxy class allows you to hide even the `private` data of a class from clients of the class.

- Providing clients of your class with a proxy class that knows only the `public` interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details.

```
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
```

Fig. 11.16 | `Implementation` class definition. (Part 1 of 2.)

```
25   private:
26      int value; // data that we would like to hide from the client
27   }; // end class Implementation
```

Fig. 11.16 | `Implementation` class definition. (Part 2 of 2.)

```
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 17
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.

11.17 Proxy Classes (cont.)

- When a class definition uses only a pointer or reference to an object of another class, the class header file for that other class is not required to be included with `#include`.
- This is because the compiler doesn't need to reserve space for an object of the class.
- The compiler does need to reserve space for the pointer or reference.
- The sizes of pointers and references are characteristics of the hardware platform on which the compiler runs, so the compiler already knows those sizes.

```
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.)

11.17 Proxy Classes (cont.)

- Notice that only the header file for `Interface` is included in the client code (line 4)—there is no mention of the existence of a separate class called `Implementation`.

- Thus, the client never sees the `private` data of class `Implementation`, nor can the client code become dependent on the `Implementation` code.

Software Engineering Observation 11.9

A proxy class insulates client code from implementation changes.

```
1  // Fig. 11.19: fig11_19.cpp
2  // Hiding a class's private data with a proxy class.
3  #include <iostream>
4  #include "Interface.h" // Interface class definition
5  using namespace std;
6
7  int main()
8  {
9     Interface i( 5 ); // create Interface object
10
11    cout << "Interface contains: " << i.getValue()
12       << " before setValue" << endl;
13
14    i.setValue( 10 );
15
16    cout << "Interface contains: " << i.getValue()
17       << " after setValue" << endl;
18 } // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

Fig. 11.19 | Implementing a proxy class.