WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Cpt S 122 – Data Structures

Operator Overloading and Class `string`

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Overloaded Operators of Standard Library class string
- Fundamentals of Operator Overloading
- Overloading *Binary* Operators
- Overloading *Unary* Operators
- Dynamic Memory Management
- Case Study: PhoneNumber, Array classes
  - Implementation of operator overloading
- Converting between types
- explicit Constructors

Introduction

- Enable C++'s operators to work with objects
  - a process called operator overloading.
- One example of an overloaded operator built into C++ is <<
  - used both as the stream insertion operator and
  - as the bitwise left-shift operator.
- C++ overloads the addition operator (+) and the subtraction operator (−) to perform differently
  - depending on their context in integer, floating-point and pointer arithmetic with data of fundamental types.
- You can overload most operators to be used with class objects
  - the compiler generates the appropriate code based on the types of the operands.

Using the Overloaded Operators of Standard Library Class `string`

- Demonstrate many of class `string`'s overloaded operators
  - <, >, ==, != , =, <=, >=, etc
- Several other useful member functions
  - `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`
  - Function `at` returns the character at a specific index in a `string`
    - checked that the index is in range.

```
1  // Fig. 11.1: fig11_01.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. 11.1 | Standard Library `string` class test program. (Part 1 of 6.)

```
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 a char * string
40      cout << "\n\ns1 += \" to you\" yields" << endl;
41      s1 += " to you";
42      cout << "s1 = " << s1 << "\n\n";
43
```

Fig. 11.1 | Standard Library `string` class test program. (Part 2 of 6.)

```
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";
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";
```

Fig. 11.1 | Standard Library `string` class test program. (Part 3 of 6.)

```
68
69      // test subscript out of range with string member function "at"
70      try
71      {
72         cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
73         s1.at( 30 ) = 'd'; // ERROR: subscript out of range
74      } // end try
75      catch ( out_of_range &ex )
76      {
77         cout << "An exception occurred: " << ex.what() << endl;
78      } // end catch
79   } // end main
```

Fig. 11.1 | Standard Library `string` class test program. (Part 4 of 6.)

```
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
```

Fig. 11.1 | Standard Library `string` class test program. (Part 5 of 6.)

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

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:
An exception occurred: invalid string position
```

Fig. 11.1 | Standard Library `string` class test program. (Part 6 of 6.)

Using the Overloaded Operators of Standard Library Class `string`

- 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.
  - Returns `true` if the `string` is empty; otherwise, it returns `false`.
- Demonstrates class `string`'s overloaded `+=` operator for *string concatenation*.
  - Demonstrates that a string literal can be appended to a `string` object by using operator `+=`

Using the Overloaded Operators of Standard Library Class `string`

- Class `string` provides member function `substr` to return a portion of a string as a `string` object.
  - The call to `substr` obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument).
  - The call to `substr` obtains a substring starting from position 15 of `s1`.
  - 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 `s1`.
  - Class `string`'s overloaded `[]` operator does not perform any bounds checking.

Using the Overloaded Operators of Standard Library Class `string`

- Class `string` *does* provide bounds checking in its member function `at`
  - 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.

Fundamentals of Operator Overloading

- Operators provide a concise notation for manipulating string objects.
- We 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.
  - they can be used with objects as long as they have meaning appropriate to those objects.

Fundamentals of Operator Overloading

- Operator overloading is not automatic
  - you must write operator overloading functions to perform the desired operations.
- An operator is overloaded by writing a member function definition or non-member function definition
  - function name starts with the keyword `operator` followed by the symbol for the operator being overloaded.
  - For example, the function name `operator+` would be used to overload the addition operator (+) for use with objects of a particular class.

Fundamentals of Operator Overloading

- To use an operator on class objects, that operator must be overloaded—with three exceptions.
  - The assignment operator (=) may be used with *every* class to perform *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).
    - *Memberwise assignment is dangerous for classes with pointer members*, so we'll explicitly overload the assignment operator for such classes.
  - The address operator returns a pointer to the object; this operator also can be overloaded.
  - The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

Fundamentals of Operator Overloading (Cont.)

- Most of C++'s operators can be overloaded.
- Operators that cannot be overloaded.

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

Fig. 11.2 | Operators that cannot be overloaded.

Fundamentals of Operator Overloading (cont.)

- You cannot change the "arity" of an operator (that is, the number of operands an operator takes)
  - overloaded unary operators remain unary operators.
  - overloaded binary operators remain binary operators.
  - operators &, *, + and – all have both unary and binary versions.
  - these unary and binary versions can be separately overloaded.

Fundamentals of Operator Overloading (Cont.)

- You cannot create new operators
  - only existing operators can be overloaded.
- The meaning of how an operator works on values of fundamental types *cannot* be changed by operator overloading.
  - For example, you cannot make the + operator subtract two ints.
  - Operator overloading works only
    - with objects of user-defined types
    - with a mixture of an object of a user-defined type
    - an object of a fundamental type.

Overloading Binary Operators

- A binary operator can be overloaded as a member function with one parameter
- As a non-member function, binary operator < must take two arguments
  - one of which must be an object or a reference to an object of the class.

Overloading the Binary Stream Insertion and Stream Extraction Operators

- You can input and output fundamental type data using
  - the stream extraction operator >>
  - the stream insertion operator <<.
- The C++ class libraries overload these binary operators
  - each fundamental type, including pointers and `char *` strings.
- You can also overload these operators to perform input and output for your own types.
- Next we overload these operators to input and output `PhoneNumber` objects
  - in the format "`(000) 000-0000`."

```
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 non-member 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 non-member 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.)

Overloading the Binary Stream Extraction Operators (cont.)

- The stream extraction operator function `operator>>`
  - takes `istream` reference `input` and
  - `PhoneNumber` reference `number` as arguments and
  - returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(800) 555-1212`
- When the compiler sees the expression
  - `cin >> phone`
- The compiler generates the *non-member function call*
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `input` becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`.

Overloading Unary Operators

- A unary operator for a class can be overloaded as a (non-`static`) member function with no arguments
  - as a non-member function with one argument that must be an object (or a reference to an object) of the class.
- A unary operator such as `!` may be overloaded as a non-member function with one parameter in two different ways
  - either with a parameter that's an object
    - this requires a copy of the object, so the side effects of the function are *not* applied to the original object, or
  - with a parameter that is a reference to an object
    - no copy of the original object is made, so all side effects of this function are applied to the original object.

Overloading the Unary Prefix and Postfix ++ and -- Operators

- 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.
  - 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.

Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- Suppose that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the member-function call
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`

- If the prefix increment operator is implemented as a non-member function, then, 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 & );`

Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- Overloading the postfix increment operator presents a challenge,
  - the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.
- The *convention* that has been adopted in C++ is that, when the compiler sees the postincrementing 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 the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

Overloading the Unary Prefix and Postfix ++ and -- Operators (cont.)

- If the postfix increment is implemented as a non-member 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 );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as non-member functions.
- The postfix increment operator returns `Date` objects *by value*, whereas the prefix increment operator returns `Date` objects *by reference*
  - the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.

Dynamic Memory Management

- Determine the size of an array *dynamically* at execution time and then create the array.
- 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 required to hold an object or array at execution time.
- The object or array is created in the free store (also called the heap)
  - a region of memory assigned to each program for storing dynamically allocated objects.
- Once memory is allocated in the free store, 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.

Dynamic Memory Management

- The `new` operator allocates storage of the proper size for an object of type `Time`,
  - *calls the default constructor* to initialize the object
  - returns a pointer to the type specified to the right of the `new` operator (i.e., a `Time *`).
- If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by "*throwing an exception*."

Dynamic Memory Management (cont.)

- To destroy a dynamically allocated 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, returning the memory to the free store.

Dynamic Memory Management (cont.)

- You can provide an initializer for a newly created fundamental type variable, as in
  - `double *ptr = new double( 3.14159 );`
- The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object.

Dynamic Memory Management (cont.)

- 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 ];`
- A dynamically allocated array's size can be specified using *any* non-negative integral expression.
- 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.

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.

Case Study: Array Class

- Pointer-based arrays have many problems, including:
  - A program can easily "walk off" either end of an array, because *C++ does not check whether subscripts fall outside the range of an array.*
  - Arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges are not allowed.
  - An entire array cannot be input or output at once.
  - Two arrays cannot be meaningfully compared with equality or relational operators.
  - When an array is passed to a general-purpose function designed to handle arrays of any size, the array's size must be passed as an additional argument.
  - One array cannot be assigned to another with the assignment operator.

Case Study: Array Class (cont.)

- With C++, you can implement more robust array capabilities via classes and operator overloading.
- You can develop an array class that is preferable to "raw" arrays.
- In this 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 `!=`.
- C++ Standard Library class template `vector` provides many of these capabilities as well.

```
1  // Fig. 11.9: fig11_09.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
```

Fig. 11.9 | Array class test program. (Part 1 of 7.)

```
22      // input and print integers1 and integers2
23      cout << "\nEnter 17 integers:" << endl;
24      cin >> integers1 >> integers2;
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
```

Fig. 11.9 | Array class test program. (Part 2 of 7.)

```
44      // use overloaded assignment (=) operator
45      cout << "\nAssigning integers2 to integers1:" << endl;
46      integers1 = integers2; // note target Array is smaller
47
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
```

Fig. 11.9 | Array class test program. (Part 3 of 7.)

```
65      // attempt to use out-of-range subscript
66      try
67      {
68         cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
69         integers1[ 15 ] = 1000; // ERROR: subscript out of range
70      } // end try
71      catch ( out_of_range &ex )
72      {
73         cout << "An exception occurred: " << ex.what() << endl;
74      } // end catch
75   } // end main
```

Fig. 11.9 | Array class test program. (Part 4 of 7.)

```
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
```

Fig. 11.9 | Array class test program. (Part 5 of 7.)

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

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
```

Fig. 11.9 | Array class test program. (Part 6 of 7.)

```
integers1[5] is 13

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

Attempt to assign 1000 to integers1[15]
An exception occurred: Subscript out of range
```

Fig. 11.9 | Array class test program. (Part 7 of 7.)

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 writing as follows:
  - `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.

Case Study: Array Class (cont.)

- The array subscript operator `[]` is not restricted for use only with arrays;
  - it also can be used 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.
  - STL `map` class allows noninteger subscripts.

```
1  // Fig. 11.10: 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. 11.10 | Array class definition with overloaded operators. (Part 1 of 2.)

Case Study: Array Class (cont.)

- When the compiler sees an expression like `cout << arrayObject`, it invokes non-member function `operator<<` with the call
  - `operator<<( cout, arrayObject )`
- When the compiler sees an expression like `cin >> arrayObject`, it invokes non-member function `operator>>` with the call
  - `operator>>( cin, arrayObject )`

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

```cpp
// Fig 11.11: Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstdlib> // exit function prototype
#include "Array.h" // Array class definition
using namespace std;

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   if ( arraySize > 0 )
      size = arraySize;
   else
      throw invalid_argument( "Array size must be greater than 0" );

   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 1 of 8.)

Case Study: Array Class (cont.)

- Declares the *default constructor* for the class and specifies a default size of 10 elements.
- The default constructor validates and assigns the argument to data member `size`,
  - uses `new` to obtain the memory for the internal pointer-based representation of this array
  - assigns the pointer returned by `new` to data member `ptr`.
- Then the constructor uses a `for` statement to set all the elements of the array to zero.

Copy Constructor for class Array

```
23
24  // copy constructor for class Array;
25  // must receive a reference to prevent infinite recursion
26  Array::Array( const Array &arrayToCopy )
27     : size( arrayToCopy.size )
28  {
29     ptr = new int[ size ]; // create space for pointer-based array
30
31     for ( int i = 0; i < size; ++i )
32        ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
33  } // end Array copy constructor
34
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 2 of 8.)

Case Study: Array Class (cont.)

- Declares a *copy constructor* that initializes an `Array` by making a copy of an existing `Array` object.
- *Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory.*
- This is exactly the problem that would occur with default memberwise copying,
  - if the compiler is allowed to define a default copy constructor for this class.
- Copy constructors are *invoked* whenever a copy of an object is needed,
  - such as in passing an object by value to a function,
  - returning an object by value from a function or
  - initializing an object with a copy of another object of the same class.

Case Study: Array Class (cont.)

- The copy constructor for `Array` uses a member initializer to copy the `size` of the initializer `Array` into data member `size`,
  - uses `new` to obtain the memory for the internal pointer-based representation of this `Array`
  - assigns the pointer returned by `new` to data member `ptr`.
- Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object.
- An object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

Note on Copy Constructor Behavior

- A copy constructor must receive its argument by reference, not by value.
- Otherwise the copy constructor call results in infinite recursion
  - Receiving an object by value requires a 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!

Destructor for class Array

```
35  // destructor for class Array
36  Array::~Array()
37  {
38     delete [] ptr; // release pointer-based array space
39  } // end destructor
40
41  // return number of elements of Array
42  int Array::getSize() const
43  {
44     return size; // number of elements in Array
45  } // end function getSize
46
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 3 of 8.)

Destructor for class Array (cont.)

- Declare the class's destructor.
- The destructor is invoked when an object of class `Array` goes out of scope.
- The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

Equality Operator for class Array

```
69  // determine if two Arrays are equal and
70  // return true, otherwise return false
71  bool Array::operator==( const Array &right ) const
72  {
73     if ( size != right.size )
74        return false; // arrays of different number of elements
75
76     for ( int i = 0; i < size; ++i )
77        if ( ptr[ i ] != right.ptr[ i ] )
78           return false; // Array contents are not equal
79
80     return true; // Arrays are equal
81  } // end function operator==
82
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 5 of 8.)

Equality Operator for class Array (cont.)

- Declare the overloaded equality operator (==) for the class.
- When the compiler sees the expression `integers1 == integers2`, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Member function `operator==` immediately returns `false` if the `size` members of the arrays are not equal.
- Otherwise, `operator==` compares each pair of elements.
  - If they're all equal, the function returns `true`.
  - The first pair of elements to differ causes the function to return `false` immediately.

Subscript Operator

```
83  // overloaded subscript operator for non-const Arrays;
84  // reference return creates a modifiable lvalue
85  int &Array::operator[]( int subscript )
86  {
87     // check for subscript out-of-range error
88     if ( subscript < 0 || subscript >= size )
89        throw out_of_range( "Subscript out of range" );
90
91     return ptr[ subscript ]; // reference return
92  } // end function operator[]
93
94  // overloaded subscript operator for const Arrays
95  // const reference return creates an rvalue
96  int Array::operator[]( int subscript ) const
97  {
98     // check for subscript out-of-range error
99     if ( subscript < 0 || subscript >= size )
100       throw out_of_range( "Subscript out of range" );
101
102    return ptr[ subscript ]; // returns copy of this element
103 } // end function operator[]
104
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 6 of 8.)

Stream Extraction Operator

```
105 // overloaded input operator for class Array;
106 // inputs values for entire Array
107 istream &operator>>( istream &input, Array &a )
108 {
109    for ( int i = 0; i < a.size; ++i )
110       input >> a.ptr[ i ];
111
112    return input; // enables cin >> x >> y;
113 } // end function
114
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 7 of 8.)

Stream Insertion Operator

```
115 // overloaded output operator for class Array
116 ostream &operator<<( ostream &output, const Array &a )
117 {
118    int i;
119
120    // output private ptr-based array
121    for ( i = 0; i < a.size; ++i )
122    {
123       output << setw( 12 ) << a.ptr[ i ];
124
125       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
126          output << endl;
127    } // end for
128
129    if ( i % 4 != 0 ) // end last line of output
130       output << endl;
131
132    return output; // enables cout << x << y;
133 } // end function operator<<
```

Fig. 11.11 | `Array` class member- and `friend`-function definitions. (Part 8 of 8.)

Overloaded Assignment Operator

```
47  // overloaded assignment operator;
48  // const return avoids: ( a1 = a2 ) = a3
49  const Array &Array::operator=( const Array &right )
50  {
51     if ( &right != this ) // avoid self-assignment
52     {
53        // for Arrays of different sizes, deallocate original
54        // left-side array, then allocate new left-side array
55        if ( size != right.size )
56        {
57           delete [] ptr; // release space
58           size = right.size; // resize this object
59           ptr = new int[ size ]; // create space for array copy
60        } // end inner if
61
62        for ( int i = 0; i < size; ++i )
63           ptr[ i ] = right.ptr[ i ]; // copy array into object
64     } // end outer if
65
66     return *this; // enables x = y = z, for example
67  } // end function operator=
68
```

```
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. 11.11 | Array class member- and friend-function def
(Part 4 of 8.)

Overloaded Assignment Operator (cont.)

- Overloaded assignment operator function for the Array class.
- When the compiler sees the expression `integers1 = integers2`, the compiler invokes member function `operator=` with the call
  - `integers1.operator=( integers2 )`
- Member function `operator=`'s implementation tests for self-assignment in which an `Array` object is being assigned to itself.
  - if `this` is equal to the `right` operand's address, a self-assignment is being attempted, so the assignment is skipped.

Overloaded Assignment Operator (cont.)

- `operator=` determines whether the sizes of the two arrays are identical
  - the original array of integers in the left-side `Array` object is not reallocated.
- Otherwise, `operator=` uses `delete`
  - to release the memory,
  - copies the `size` of the source array to the `size` of the target array,
  - uses `new` to allocate memory for the target array and
  - places the pointer returned by `new` into the array's `ptr` member.
- Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this`) as a constant reference;
  - this enables cascaded `Array` assignments such as `x = y = z`,
  - prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array&` reference that is returned by `(x = y)`.

The Big Three

- A copy constructor, a destructor, and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.
- Not providing a copy constructor, and an overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

Overloaded Inequality Operator (cont.)

- Overloaded inequality operator (`!=`).
- Member function `operator!=` uses the overloaded `operator==` function to determine whether one `Array` is equal to another, then returns the opposite of that result.
- Writing `operator!=` in this manner enables you to reuse `operator==`, which *reduces the amount of code that must be written in the class*.
- Full function definition for `operator!=` allows the compiler to inline the definition.

Cast Operator: Converting between Types

- Sometimes all the operations "stay within a type."
  - For example, adding an `int` to an `int` produces an `int`.
- It's often necessary, however, to convert data of one type to data of 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, or
  - between user-defined types and fundamental types, so you must specify how to do this.

Converting between Types (cont.)

- Such conversions can be performed with conversion constructors
  - single-argument constructors that turn objects of other types (including fundamental types) into objects of a particular class.
- 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.
- The function prototype
  - `A::operator char *() const;`
  - declares an overloaded cast operator function for converting an object of user-defined type `A` into a temporary `char *` object.
  - The operator function is declared `const` because it does not modify the original object.

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 *()`
- Example:
  - `A::operator int() const;` Convert an object of user defined type A into an integer
  - `A::operator OtherClass() const;` Convert an object of user defined type A into an object of user defined type *Otherclass*
- Nice features of cast operators and conversion constructors
  - the compiler can call these functions implicitly to create temporary objects.
  - cout << s; (object s of user-defined string class to char *, stream insertion operator doest not have to be overloaded)

explicit Constructors

- Any single-argument constructor can be used by the compiler to perform an implicit conversion.
  - The constructor's argument 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.*
  - For example, our `Array` class defines a constructor that takes a single `int` argument.
  - The intent of this constructor is to create an `Array` object containing the number of elements specified by the `int` argument.
  - However, this constructor can be misused by the compiler to perform an *implicit* conversion.

```
1  // Fig. 11.12: Fig11_12.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. 11.12 | 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. 11.12 | Single-argument constructors and implicit conversions. (Part 2 of 2.)

explicit Constructors (cont.)

- The program uses the `Array` class to demonstrate an improper implicit conversion.
  - Calls function `outputArray` with the `int` value `3` as an argument.
- This program does not contain a function called `outputArray` that takes an `int` argument.
  - The compiler determines whether class `Array` provides a conversion constructor that can convert an `int` into an `Array`.
  - The compiler assumes the `Array` constructor that receives a single `int` is a conversion constructor and
    - uses it to convert the argument `3` into a temporary `Array` object that contains three elements.
  - Then, the compiler passes the temporary `Array` object to function `outputArray` to output the `Array`'s contents.

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.
  - `explicit Array( int = 10) //default constructor`
  - No modifications are required to the source-code file containing class `Array`'s member-function definitions.

With an explicit Constructor (cont.)

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

```
c:\cpphtp8_examples\ch11\fig11_13\fig11_13.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. 11.13 | Demonstrating an `explicit` constructor. (Part 2 of 2.)

explicit Constructors (cont.)

- Demonstrate how the explicit constructor can be used to create a temporary `Array` of `3` elements and pass it to function `outputArray`.
- When this program is compiled, the compiler produces an error message indicating that the integer value passed to `outputArray` cannot be converted to a `const Array &`.