11

# Operator Overloading; String and Array Objects

***The whole difference between construction and creation is exactly this:***
***that a thing constructed can only be loved after it is constructed; but a thing created is loved before it exists.***

**— Gilbert Keith Chesterton**

***The die is cast.***

**— Julius Caesar**

***Our doctor would never really operate unless it was necessary. He was just that way. If he didn't need the money, he wouldn't lay a hand on you.***

**— Herb Shriner**

# OBJECTIVES

In this chapter you will learn:

- What operator overloading is and how it makes programs more readable and programming more convenient.
- To redefine (overload) operators to work with objects of user-defined classes.
- The differences between overloading unary and binary operators.
- To convert objects from one class to another class.
- When to, and when not to, overload operators.
- To create `PhoneNumber`, `Array`, `String` and `Date` classes that demonstrate operator overloading.
- To use overloaded operators and other member functions of standard library class `string`.
- To use keyword `explicit` to prevent the compiler from using single-argument constructors to perform implicit conversions.

Outline



# 11.1 Introduction

- **Use operators with objects (operator overloading)**
  - **Clearer than function calls for certain classes**
  - **Operator sensitive to context**
- **Examples**
  - **<<**
    - **Stream insertion, bitwise left-shift**
  - **+**
    - **Performs arithmetic on multiple items (integers, floats, etc.)**

# 11.2 Fundamentals of Operator Overloading

- **Types for operator overloading**
  - **Built in (`int`, `char`) or user-defined (classes)**
  - **Can use existing operators with user-defined types**
    - **Cannot create new operators**
- **Overloading operators**
  - **Create a function for the class**
  - **Name of operator function**
    - **Keyword `operator` followed by symbol**
      - **Example**
        - **`operator+` for the addition operator +**

# 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.)

- **Using operators on a class object**
  - **It must be overloaded for that class**
    - **Exceptions: (can also be overloaded by the programmer)**
      - **Assignment operator (=)**
        - **Memberwise assignment between objects**
      - **Address operator (&)**
        - **Returns address of object**
      - **Comma operator (,)**
        - **Evaluates expression to its left then the expression to its right**
        - **Returns the value of the expression to its right**
- **Overloading provides concise notation**
  - `object2 = object1.add( object2 );`

    **vs.**

    `object2 = object2 + object1;`

# 11.3 Restrictions on Operator Overloading

- Cannot change
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order of operators
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - e.g., & is unary, can only act on one operand
  - How operators act on built-in data types (i.e., cannot change integer addition)
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading + and = does not overload +=
- Operator ?: cannot be overloaded

# Common Programming Error 11.1

**Attempting to overload a nonoverloadable operator is a syntax error.**

| 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.**

# Common Programming Error 11.2

**Attempting to change the "arity" of an operator via operator overloading is a compilation error.**

# Common Programming Error 11.3

**Attempting to create new operators via operator overloading is a syntax error.**

# 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 programmers from changing how operators work on fundamental types.**

# Common Programming Error 11.4

**Attempting to modify how an operator works with objects of fundamental types is a compilation error.**

# Common Programming Error 11.5

**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 Members

- **Operator functions**
  - **As member functions**
    - **Leftmost object must be of same class as operator function**
    - **Use `this` keyword to implicitly get left operand argument**
    - **Operators `()`, `[]`, `->` or any assignment operator must be overloaded as a class member function**
    - **Called when**
      - **Left operand of binary operator is of this class**
      - **Single operand of unary operator is of this class**
  - **As global functions**
    - **Need parameters for both operands**
    - **Can have object of different class than operator**
    - **Can be a `friend` to access `private` or `protected` data**

# 11.4 Operator Functions as Class Members vs. Global Members (Cont.)

- **Overloaded << operator**
  - **Left operand of type `ostream &`**
    - **Such as `cout` object in `cout << classObject`**
  - **Similarly, overloaded >> has left operand of `istream &`**
  - **Thus, both must be global functions**

# Performance Tip 11.1

**It is 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.**

# 11.4 Operator Functions as Class Members vs. Global Members (Cont.)

- **Commutative operators**
  - **May want + to be commutative**
    - **So both "`a + b`" and "`b + a`" work**
  - **Suppose we have two different classes**
    - **Overloaded operator can only be member function when its class is on left**
      - `HugeIntClass + long int`
        - **Can be member function**
    - **When other way, need a global overloaded function**
      - `long int + HugeIntClass`

# 11.5 Overloading Stream Insertion and Stream Extraction Operators

- **<< and >> operators**
  - **Already overloaded to process each built-in type**
  - **Can also process a user-defined class**
    - **Overload using global, `friend` functions**
- **Example program**
  - **Class `PhoneNumber`**
    - **Holds a telephone number**
  - **Print out formatted number automatically**

    `(123) 456-7890`

## Outline

**PhoneNumber.h**

(1 of 1)

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15    friend ostream &operator<<( ostream &, const PhoneNumber & );
16    friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18    string areaCode; // 3-digit area code
19    string exchange; // 3-digit exchange
20    string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

Notice function prototypes for overloaded operators **>>** and **<<** (must be global, **`friend`** functions)

Outline

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

Allows **`cout << phone;`** to be interpreted as: **`operator<<(cout, phone);`**

(1 of 2)

Display formatted phone number

## Outline

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

**`ignore`** skips specified number of characters from input (1 by default)

Input each portion of phone number separately

(2 of 2)

Outline

fig11_05.cpp

(1 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  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "PhoneNumber.h"
10
11 int main()
12 {
13    PhoneNumber phone; // create object phone
14
15    cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17    // cin >> phone invokes operator>> by implicitly issuing
18    // the global function call operator>>( cin, phone )
19    cin >> phone;
20
21    cout << "The phone number entered was: ";
22
23    // cout << phone invokes operator<< by implicitly issuing
24    // the global function call operator<<( cout, phone )
25    cout << phone << endl;
26    return 0;
27 } // end main
```

Testing overloaded **>>** and **<<** operators to input and output a **PhoneNumber** object

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

Outline

fig11_05.cpp

(2 of 2)

# 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—creating "dangling references" to nonexisting objects.**

# Software Engineering Observation 11.3

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

# 11.6 Overloading Unary Operators

- **Overloading unary operators**
  - **Can overload as non-`static` member function with no arguments**
  - **Can overload as global function with one argument**
    - **Argument must be class object or reference to class object**
  - **Remember, `static` functions only access `static` data**

# 11.6 Overloading Unary Operators (Cont.)

- **Upcoming example (Section 11.10)**
  - Overload `!` to test for empty string
  - If non-`static` member function, needs no arguments
    - ```
      class String
      {
      public:
         bool operator!() const;
         ...
      };
      ```
    - `!s` becomes `s.operator!()`
  - If global function, needs one argument
    - `bool operator!( const String & )`
    - `s!` becomes `operator!(s)`


      ```

# 11.7 Overloading Binary Operators

- **Overloading binary operators**
  - **Non-`static` member function, one argument**
  - **Global function, two arguments**
    - **One argument must be class object or reference**

# 11.7 Overloading Binary Operators (Cont.)

- **Upcoming example: Overloading +=**
  - **If non-`static` member function, needs one argument**
    - ```
      class String
      {
      public:
         const String & operator+=( const String & );
         …
      };
      ```
    - `y += z` **becomes** `y.operator+=( z )`
  - **If global function, needs two arguments**
    - `const String &operator+=( String &, const String & );`
    - `y += z` **becomes** `operator+=( y, z )`


      ```

# 11.8 Case Study: `Array` Class

- **Pointer-based arrays in C++**
  - **No range checking**
  - **Cannot be compared meaningfully with ==**
  - **No array assignment (array names are `const` pointers)**
  - **If array passed to a function, size must be passed as a separate argument**
- **Example: Implement an `Array` class with**
  - **Range checking**
  - **Array assignment**
  - **Arrays that know their own size**
  - **Outputting/inputting entire arrays with << and >>**
  - **Array comparisons with == and `!=`**

# 11.8 Case Study: `Array` Class (Cont.)

- **Copy constructor**
  - **Used whenever copy of object is needed:**
    - **Passing by value (return value or parameter)**
    - **Initializing an object with a copy of another of same type**
      - `Array newArray( oldArray );` **or**
        `Array newArray = oldArray` **(both are identical)**
        - `newArray` **is a copy of** `oldArray`
  - **Prototype for class** `Array`
    - `Array( const Array & );`
    - **Must take reference**
      - **Otherwise, the argument will be passed by value…**
      - **Which tries to make copy by calling copy constructor…**
        - **Infinite loop**

Outline

Array.h

(1 of 2)

```
1  // Fig. 11.6: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
```

Most operators overloaded as member functions (except **<<** and **>>**, which must be global functions)

Prototype for copy constructor

**!=** operator simply returns opposite of **==** operator – only need to define the **==** operator

## Outline

```
28
29    // subscript operator for non-const objects returns modifiable lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    int operator[]( int ) const;
34 private:
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based ar
37 }; // end class Array
38
39 #endif
```

Array.h

Operators for accessing specific elements of **Array** object

Outline

Array.cpp

(1 of 6)

```
1  // Fig 11.7: Array.cpp
2  // Member-function definitions for class Array
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // exit function prototype
13 using std::exit;
14
15 #include "Array.h" // Array class definition
16
17 // default constructor for class Array (default size 10)
18 Array::Array( int arraySize )
19 {
20    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
21    ptr = new int[ size ]; // create space for pointer-based array
22
23    for ( int i = 0; i < size; i++ )
24       ptr[ i ] = 0; // set pointer-based array element
25 } // end Array default constructor
```

Outline

Array.cpp

(2 of 6)

```
26
27 // copy constructor for class Array;
28 // must receive a reference to prevent infinite recursion
29 Array::Array( const Array &arrayToCopy )
30    : size( arrayToCopy.size )
31 {
32    ptr = new int[ size ]; // create space for pointer-based array
33
34    for ( int i = 0; i < size; i++ )
35       ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
36 } // end Array copy constructor
37
38 // destructor for class Array
39 Array::~Array()
40 {
41    delete [] ptr; // release pointer-based array space
42 } // end destructor
43
44 // return number of elements of Array
45 int Array::getSize() const
46 {
47    return size; // number of elements in Array
48 } // end function getSize
```

We must declare a new integer array so the objects do not point to the same memory

## Outline

Array.cpp

(3 of 6)

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

Want to avoid self assignment

This would be dangerous if **this** is the same **Array** as **right**

Outline

Array.cpp

(4 of 6)

```
71
72 // determine if two Arrays are equal and
73 // return true, otherwise return false
74 bool Array::operator==( const Array &right ) const
75 {
76    if ( size != right.size )
77       return false; // arrays of different number of elements
78
79    for ( int i = 0; i < size; i++ )
80       if ( ptr[ i ] != right.ptr[ i ] )
81          return false; // Array contents are not equal
82
83    return true; // Arrays are equal
84 } // end function operator==
85
86 // overloaded subscript operator for non-const Arrays;
87 // reference return creates a modifiable lvalue
88 int &Array::operator[]( int subscript )
89 {
90    // check for subscript out-of-range error
91    if ( subscript < 0 || subscript >= size )
92    {
93       cerr << "\nError: Subscript " << subscript
94          << " out of range" << endl;
95       exit( 1 ); // terminate program; subscript out of range
96    } // end if
97
98    return ptr[ subscript ]; // reference return
99 } // end function operator[]
```

integers1[ 5 ] calls integers1.operator[ ]( 5 )

Outline

Array.cpp

(5 of 6)

```
100
101// overloaded subscript operator for const Arrays
102// const reference return creates an rvalue
103int Array::operator[]( int subscript ) const
104{
105   // check for subscript out-of-range error
106   if ( subscript < 0 || subscript >= size )
107   {
108      cerr << "\nError: Subscript " << subscript
109         << " out of range" << endl;
110      exit( 1 ); // terminate program; subscript out of range
111   } // end if
112
113   return ptr[ subscript ]; // returns copy of this element
114} // end function operator[]
115
116// overloaded input operator for class Array;
117// inputs values for entire Array
118istream &operator>>( istream &input, Array &a )
119{
120   for ( int i = 0; i < a.size; i++ )
121      input >> a.ptr[ i ];
122
123   return input; // enables cin >> x >> y;
124} // end function
```

## Outline

**Array.cpp**

(6 of 6)

```
125
126// overloaded output operator for class Array
127ostream &operator<<( ostream &output, const Array &a )
128{
129   int i;
130
131   // output private ptr-based array
132   for ( i = 0; i < a.size; i++ )
133   {
134      output << setw( 12 ) << a.ptr[ i ];
135
136      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137         output << endl;
138   } // end for
139
140   if ( i % 4 != 0 ) // end last line of output
141      output << endl;
142
143   return output; // enables cout << x << y;
144} // end function operator<<
```

Outline

fig11_08.cpp

(1 of 5)

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

Retrieve number of elements in **Array**

Use overloaded **>>** operator to input

Outline

fig11_08.cpp

```
28
29    cout << "\nAfter input, the Arrays contain:\n"
30       << "integers1:\n" << integers1
31       << "integers2:\n" << integers2;
32
33    // use overloaded inequality (!=) operator
34    cout << "\nEvaluating: integers1 != integers2" << endl;
35
36    if ( integers1 != integers2 )
37       cout << "integers1 and integers2 are not equal" << endl;
38
39    // create Array integers3 using integers1 as an
40    // initializer; print size and contents
41    Array integers3( integers1 ); // invokes copy constructor
42
43    cout << "\nSize of Array integers3 is "
44       << integers3.getSize()
45       << "\nArray after initialization:\n" << integers3;
46
47    // use overloaded assignment (=) operator
48    cout << "\nAssigning integers2 to integers1:" << endl;
49    integers1 = integers2; // note target Array is smaller
50
51    cout << "integers1:\n" << integers1
52       << "integers2:\n" << integers2;
53
54    // use overloaded equality (==) operator
55    cout << "\nEvaluating: integers1 == integers2" << endl;
```

Use overloaded << operator to output

Use overloaded != operator to test for inequality

Use copy constructor

Use overloaded = operator to assign

```
56
57    if ( integers1 == integers2 )
58       cout << "integers1 and integers2 are equal" << endl;
59
60    // use overloaded subscript operator to create rvalue
61    cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63    // use overloaded subscript operator to create lvalue
64    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65    integers1[ 5 ] = 1000;
66    cout << "integers1:\n" << integers1;
67
68    // attempt to use out-of-range subscript
69    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70    integers1[ 15 ] = 1000; // ERROR: out of range
71    return 0;
72 } // end main
```

Use overloaded == operator to test for equality

Use overloaded [] operator to access individual integers, with range -checking

fig11_08.cpp

(3 of 5)

```
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
```

## Outline

fig11_08.cpp

(4 of 5)

```
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

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

## Outline

fig11_08.cpp

(5 of 5)

# Software Engineering Observation 11.4

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

# Common Programming Error 11.6

**Note that 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!**

# Common Programming Error 11.7

**If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object's `ptr` would be undefined, a situation called a dangling pointer—this would likely result in a serious run-time error (such as early program termination) when the pointer was used.**

# 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 11.8

**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.**

# Software Engineering Observation 11.6

**It is possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a `private` member of the class.**

# Software Engineering Observation 11.7

**It is possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class `private`.**

# 11.9 Converting between Types

- **Casting**
  - **Traditionally, cast integers to floats, etc.**
  - **May need to convert between user-defined types**
- **Cast operator (conversion operator)**
  - **Convert from**
    - **One class to another**
    - **A Class to a built-in type (`int`, `char`, etc.)**
  - **Must be non-`static` member function**
  - **Do not specify return type**
    - **Implicitly returns type to which you are converting**

# 11.9 Converting between Types (Cont.)

- **Cast operator (conversion operator) (Cont.)**
  - **Example**
    - **Prototype**

```
A::operator char *() const;
```

      - **Casts class A to a temporary `char *`**
      - `static_cast< char * >( s )` **calls** `s.operator char *()`
    - **Also**
      - `A::operator int() const;`
      - `A::operator OtherClass() const;`

# 11.9 Converting between Types (Cont.)

- **Casting can prevent need for overloading**
  - **Suppose class `String` can be cast to `char *`**
  - `cout << s; // s is a String`
    - **Compiler implicitly converts `s` to `char *` for output**
    - **Do not have to overload <<**

# 11.10 Case Study: `String` Class

- **Build class `String`**
  - **String creation, manipulation**
  - **Similar to class `string` in standard library (Chapter 18)**
- **Conversion constructor**
  - **Any single-argument constructor**
  - **Turns objects of other types into class objects**
    - **Example**
      - `String s1( "happy" );`
        - **Creates a `String` from a `char *`**
- **Overloading function call operator**
  - **Powerful (functions can take arbitrarily long and complex parameter lists)**

Outline

String.h

(1 of 3)

```
1  // Fig. 11.9: String.h
2  // String class definition.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class String
11 {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14 public:
15    String( const char * = "" ); // conversion/default constructor
16    String( const String & ); // copy constructor
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment operator
20    const String &operator+=( const String & ); // concatenation operator
21
22    bool operator!() const; // is String empty?
23    bool operator==( const String & ) const; // test s1 == s
24    bool operator<( const String & ) const; // test s1 < s2
25
```

Conversion constructor to make a **String** from a **char *** 

**s1 += s2** will be interpreted as **s1.operator+=(s2)**

Can also concatenate a **String** and a **char *** because the compiler will cast the **char *** argument to a **String**

## Outline

`String.h`

(2 of 3)

```
26    // test s1 != s2
27    bool operator!=( const String &right ) const
28    {
29       return !( *this == right );
30    } // end function operator!=
31
32    // test s1 > s2
33    bool operator>( const String &right ) const
34    {
35       return right < *this;
36    } // end function operator>
37
38    // test s1 <= s2
39    bool operator<=( const String &right ) const
40    {
41       return !( right < *this );
42    } // end function operator <=
43
44    // test s1 >= s2
45    bool operator>=( const String &right ) const
46    {
47       return !( *this < right );
48    } // end function operator>=
```

Overload equality and relational operators

```
49
50    char &operator[]( int ); // subscript operator (modifiable lvalue)
51    char operator[]( int ) const; // subscript operator (rvalue)
52    String operator()( int, int = 0 ) const; // return a substring
53    int getLength() const; // return string length
54 private:
55    int length; // string length (not counting null terminator)
56    char *sPtr; // pointer to start of pointer-based string
57
58    void setString( const char * ); // utility function
59 }; // end class String
60
61 #endif
```

Two overloaded subscript operators, for **`const`** and non-**`const`** objects

Overload the function call operator **`()`** to return a substring

**`String.h`**

Outline

String.cpp

(1 of 7)

```
1  // Fig. 11.10: String.cpp
2  // Member-function definitions for class String.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <cstring> // strcpy and strcat prototypes
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // exit prototype
17 using std::exit;
18
19 #include "String.h" // String class definition
20
21 // conversion (and default) constructor converts char * to String
22 String::String( const char *s )
23    : length( ( s != 0 ) ? strlen( s ) : 0 )
24 {
25    cout << "Conversion (and default) constructor: " << s << endl;
26    setString( s ); // call utility function
27 } // end String conversion constructor
28
```

Outline

String.cpp

(2 of 7)

```
29 // copy constructor
30 String::String( const String &copy )
31    : length( copy.length )
32 {
33    cout << "Copy constructor: " << copy.sPtr << endl;
34    setString( copy.sPtr ); // call utility function
35 } // end String copy constructor
36
37 // Destructor
38 String::~String()
39 {
40    cout << "Destructor: " << sPtr << endl;
41    delete [] sPtr; // release pointer-based string memory
42 } // end ~String destructor
43
44 // overloaded = operator; avoids self assignment
45 const String &String::operator=( const String &right )
46 {
47    cout << "operator= called" << endl;
48
49    if ( &right != this ) // avoid self assignment
50    {
51       delete [] sPtr; // prevents memory leak
52       length = right.length; // new String length
53       setString( right.sPtr ); // call utility function
54    } // end if
55    else
56       cout << "Attempted assignment of a String to itself" << endl;
57
```

## Outline

String.cpp

(3 of 7)

```
58    return *this; // enables cascaded assignments
59 } // end function operator=
60
61 // concatenate right operand to this object and store in this object
62 const String &String::operator+=( const String &right )
63 {
64    size_t newLength = length + right.length; // new length
65    char *tempPtr = new char[ newLength + 1 ]; // create memory
66
67    strcpy( tempPtr, sPtr ); // copy sPtr
68    strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
69
70    delete [] sPtr; // reclaim old space
71    sPtr = tempPtr; // assign new array to sPtr
72    length = newLength; // assign new length to length
73    return *this; // enables cascaded calls
74 } // end function operator+=
75
76 // is this String empty?
77 bool String::operator!() const
78 {
79    return length == 0;
80 } // end function operator!
81
82 // Is this String equal to right String?
83 bool String::operator==( const String &right ) const
84 {
85    return strcmp( sPtr, right.sPtr ) == 0;
86 } // end function operator==
87
```

Outline

String.cpp

(4 of 7)

```
88 // Is this String less than right String?
89 bool String::operator<( const String &right ) const
90 {
91    return strcmp( sPtr, right.sPtr ) < 0;
92 } // end function operator<
93
94 // return reference to character in String as a modifiable lvalue
95 char &String::operator[]( int subscript )
96 {
97    // test for subscript out of range
98    if ( subscript < 0 || subscript >= length )
99    {
100      cerr << "Error: Subscript " << subscript
101         << " out of range" << endl;
102      exit( 1 ); // terminate program
103   } // end if
104
105   return sPtr[ subscript ]; // non-const return; modifiable lvalue
106} // end function operator[]
107
108// return reference to character in String as rvalue
109char String::operator[]( int subscript ) const
110{
111   // test for subscript out of range
112   if ( subscript < 0 || subscript >= length )
113   {
114      cerr << "Error: Subscript " << subscript
115           << " out of range" << endl;
116      exit( 1 ); // terminate program
117   } // end if
```

## Outline

`String.cpp`

(5 of 7)

```
118
119   return sPtr[ subscript ]; // returns copy of this element
120} // end function operator[]
121
122// return a substring beginning at index and of length subLength
123String String::operator()( int index, int subLength ) const
124{
125   // if index is out of range or substring length < 0,
126   // return an empty String object
127   if ( index < 0 || index >= length || subLength < 0 )
128      return ""; // converted to a String object automatically
129
130   // determine length of substring
131   int len;
132
133   if ( ( subLength == 0 ) || ( index + subLength > length ) )
134      len = length - index;
135   else
136      len = subLength;
137
138   // allocate temporary array for substring and
139   // terminating null character
140   char *tempPtr = new char[ len + 1 ];
141
142   // copy substring into char array and terminate string
143   strncpy( tempPtr, &sPtr[ index ], len );
144   tempPtr[ len ] = '\0';
```

Outline

String.cpp

(6 of 7)

```
145
146   // create temporary String object containing the substring
147   String tempString( tempPtr );
148   delete [] tempPtr; // delete temporary array
149   return tempString; // return copy of the temporary String
150} // end function operator()
151
152// return string length
153int String::getLength() const
154{
155   return length;
156} // end function getLength
157
158// utility function called by constructors and operator=
159void String::setString( const char *string2 )
160{
161   sPtr = new char[ length + 1 ]; // allocate memory
162
163   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
164      strcpy( sPtr, string2 ); // copy literal to object
165   else // if string2 is a null pointer, make this an empty string
166      sPtr[ 0 ] = '\0'; // empty string
167} // end function setString
168
169// overloaded output operator
170ostream &operator<<( ostream &output, const String &s )
171{
172   output << s.sPtr;
173   return output; // enables cascading
174} // end function operator<<
```

```
175
176// overloaded input operator
177istream &operator>>( istream &input, String &s )
178{
179   char temp[ 100 ]; // buffer to store input
180   input >> setw( 100 ) >> temp;
181   s = temp; // use String class assignment operator
182   return input; // enables cascading
183} // end function operator>>
```

Outline

String.cpp

(7 of 7)

Outline

fig11_11.cpp

(1 of 5)

```
1  // Fig. 11.11: fig11_11.cpp
2  // String class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha;
7
8  #include "String.h"
9
10 int main()
11 {
12    String s1( "happy" );
13    String s2( " birthday" );
14    String s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18       << "\"; s3 is \"" << s3 << '\"'
19       << boolalpha << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 )
21       << "\ns2 != s1 yields " << ( s2 != s1 )
22       << "\ns2 >  s1 yields " << ( s2 > s1 )
23       << "\ns2 <  s1 yields " << ( s2 < s1 )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 );
26
27
28    // test overloaded String empty (!) operator
29    cout << "\n\nTesting !s3:" << endl;
30
```

Use overloaded stream insertion operator for **String**s

Use overloaded equality and relational operators for **String**s

```
31    if ( !s3 )
32    {
33       cout << "s3 is empty; assigning s1 to s3;" << endl;
34       s3 = s1; // test overloaded assignment
35       cout << "s3 is \"" << s3 << "\"";
36    } // end if
37
38    // test overloaded String concatenation operator
39    cout << "\n\ns1 += s2 yields s1 = ";
40    s1 += s2; // test overloaded concatenation
41    cout << s1;
42
43    // test conversion constructor
44    cout << "\n\ns1 += \" to you\" yields" << endl;
45    s1 += " to you"; // test conversion constructor
46    cout << "s1 = " << s1 << "\n\n";
47
48    // test overloaded function call operator () for substring
49    cout << "The substring of s1 starting at\n"
50       << "location 0 for 14 characters, s1(0, 14), is:\n"
51       << s1( 0, 14 ) << "\n\n";
52
53    // test substring "to-end-of-String" option
54    cout << "The substring of s1 starting at\n"
55       << "location 15, s1(15), is: "
56       << s1( 15 ) << "\n\n";
57
58    // test copy constructor
59    String *s4Ptr = new String( s1 );
60    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
```

Use overloaded negation operator for **String**s

Use overloaded assignment operator for **String**s

Use overloaded addition assignment operator for **String**s

**char *** string is converted to a **String** before using the overloaded addition assignment operator

Use overloaded function call operator for **String**s

ine

g11_11.cpp

(2 of 5)

Outline

fig11_11.cpp

(3 of 5)

```
61
62    // test assignment (=) operator with self-assignment
63    cout << "assigning *s4Ptr to *s4Ptr" << endl;
64    *s4Ptr = *s4Ptr; // test overloaded assignment
65    cout << "*s4Ptr = " << *s4Ptr << endl;
66
67    // test destructor
68    delete s4Ptr;
69
70    // test using subscript operator to create a modifiable lvalue
71    s1[ 0 ] = 'H';
72    s1[ 6 ] = 'B';
73    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74       << s1 << "\n\n";
75
76    // test subscript out of range
77    cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78    s1[ 30 ] = 'd'; // ERROR: subscript out of range
79    return 0;
80 } // end main
```

Use overloaded subscript operator for **`String`**s

Attempt to access a subscript outside of the valid range

## Outline

**fig11_11.cpp**

(4 of 5)

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
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:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday
```

The constructor and destructor are called for the temporary **`String`**

*(continued at top of next slide... )*

*(... continued from bottom of previous slide)*

```
Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself


*s4Ptr = happy birthday to you
Destructor: happy birthday to you

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

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

Outline

fig11_11.cpp

(5 of 5)

# 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 match an overloaded operator's needs by performing a series of implicit, user-defined conversions.**

# Performance Tip 11.2

**Overloading the += concatenation operator with an additional version that takes a single argument of type `const char *` executes more efficiently than having only a version that takes a `String` argument. Without the `const char *` version of the += operator, a `const char *` argument would first be converted to a `String` object with class `String`'s conversion constructor, then the += operator that receives a `String` argument would be called to perform the concatenation.**

# Software Engineering Observation 11.9

**Using implicit conversions with overloaded operators, rather than overloading operators for many different operand types, often requires less code, which makes a class easier to modify, maintain and debug.**

## Software Engineering Observation 11.10

**By implementing member functions using previously defined member functions, the programmer reuses code to reduce the amount of code that must be written and maintained.**

# Error-Prevention Tip 11.2

**Returning a `non-const` char reference from an overloaded subscript operator in a `String` class is dangerous. For example, the client could use this reference to insert a null (`'\0'`) anywhere in the string.**

# 11.11 Overloading ++ and --

- **Increment/decrement operators can be overloaded**
  - **Suppose we want to add 1 to a `Date` object, `d1`**
  - **Prototype (member function)**
    - `Date &operator++();`
    - `++d1` **becomes** `d1.operator++()`
  - **Prototype (global function)**
    - `Date &operator++( Date & );`
    - `++d1` **becomes** `operator++( d1 )`

# 11.11 Overloading ++ and -- (Cont.)

- **To distinguish prefix and postfix increment**
  - **Postfix increment has a dummy parameter**
    - **An `int` with value `0`**
  - **Prototype (member function)**
    - `Date operator++( int );`
    - **`d1++` becomes** `d1.operator++( 0 )`
  - **Prototype (global function)**
    - `Date operator++( Date &, int );`
    - **`d1++` becomes** `operator++( d1, 0 )`

# 11.11 Overloading ++ and −− (Cont.)

- **Return values**
  - **Prefix increment**
    - **Returns by reference (`Date &`)**
    - ***lvalue* (can be assigned)**
  - **Postfix increment**
    - **Returns by value**
      - **Returns temporary object with old value**
    - ***rvalue* (cannot be on left side of assignment)**
- **All this applies to decrement operators as well**

# Performance Tip 11.3

**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).**

# 11.12 Case Study: A `Date` Class

- **Example `Date` class**
  - **Overloaded increment operator**
    - **Change day, month and year**
  - **Overloaded += operator**
  - **Function to test for leap years**
  - **Function to determine if day is last of month**

Outline

Date.h

(1 of 1)

```
1  // Fig. 11.12: Date.h
2  // Date class definition.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
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    bool leapYear( int ) const; // 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;
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
```

Note the difference between prefix and postfix increment

Outline

Date.cpp

(1 of 4)

```
1  // Fig. 11.13: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h"
5
6  // initialize static member at file scope; one classwide copy
7  const int Date::days[] =
8     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10 // Date constructor
11 Date::Date( int m, int d, int y )
12 {
13    setDate( m, d, y );
14 } // end Date constructor
15
16 // set month, day and year
17 void Date::setDate( int mm, int dd, int yy )
18 {
19    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22    // test for a leap year
23    if ( month == 2 && leapYear( year ) )
24       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25    else
26       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // end function setDate
```

## Outline

**Date.cpp**

(2 of 4)

```
28
29 // overloaded prefix increment operator
30 Date &Date::operator++()
31 {
32    helpIncrement(); // increment date
33    return *this; // reference return to create an lvalue
34 } // end function operator++
35
36 // overloaded postfix increment operator; note that the
37 // dummy integer parameter does not have a parameter name
38 Date Date::operator++( int )
39 {
40    Date temp = *this; // hold current state of object
41    helpIncrement();
42
43    // return unincremented, saved, temporary object
44    return temp; // value return; not a reference return
45 } // end function operator++
46
47 // add specified number of days to date
48 const Date &Date::operator+=( int additionalDays )
49 {
50    for ( int i = 0; i < additionalDays; i++ )
51       helpIncrement();
52
53    return *this; // enables cascading
54 } // end function operator+=
55
```

Postfix increment updates object and returns a copy of the original

Do not return a reference to `temp`, because it is a local variable that will be destroyed

Outline

Date.cpp

(3 of 4)

```
56 // if the year is a leap year, return true; otherwise, return false
57 bool Date::leapYear( int testYear ) const
58 {
59    if ( testYear % 400 == 0 ||
60       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
61       return true; // a leap year
62    else
63       return false; // not a leap year
64 } // end function leapYear
65
66 // determine whether the day is the last day of the month
67 bool Date::endOfMonth( int testDay ) const
68 {
69    if ( month == 2 && leapYear( year ) )
70       return testDay == 29; // last day of Feb. in leap year
71    else
72       return testDay == days[ month ];
73 } // end function endOfMonth
74
```

Outline

Date.cpp

(4 of 4)

```
75 // function to help increment the date
76 void Date::helpIncrement()
77 {
78    // day is not end of month
79    if ( !endOfMonth( day ) )
80       day++; // increment day
81    else
82       if ( month < 12 ) // day is end of month and month < 12
83       {
84          month++; // increment month
85          day = 1; // first day of new month
86       } // end if
87       else // last day of year
88       {
89          year++; // increment year
90          month = 1; // first month of new year
91          day = 1; // first day of new month
92       } // end else
93 } // end function helpIncrement
94
95 // overloaded output operator
96 ostream &operator<<( ostream &output, const Date &d )
97 {
98    static char *monthName[ 13 ] = { "", "January", "February",
99       "March", "April", "May", "June", "July", "August",
100      "September", "October", "November", "December" };
101   output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
102   return output; // enables cascading
103} // end function operator<<
```

Outline

fig11_14.cpp

(1 of 2)

```
1  // Fig. 11.14: fig11_14.cpp
2  // Date class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // Date class definition
8
9  int main()
10 {
11    Date d1; // defaults to January 1, 1900
12    Date d2( 12, 27, 1992 ); // December 27, 1992
13    Date d3( 0, 99, 8045 ); // invalid date
14
15    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16    cout << "\n\nd2 += 7 is " << ( d2 += 7 );
17
18    d3.setDate( 2, 28, 1992 );
19    cout << "\n\n  d3 is " << d3;
20    cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22    Date d4( 7, 13, 2002 );
23
24    cout << "\n\nTesting the prefix increment operator:\n"
25       << "  d4 is " << d4 << endl;
26    cout << "++d4 is " << ++d4 << endl;
27    cout << "  d4 is " << d4;
28
```

Demonstrate prefix increment

```
29    cout << "\n\nTesting the postfix increment operator:\n"
30       << "  d4 is " << d4 << endl;
31    cout << "d4++ is " << d4++ << endl;
32    cout << "  d4 is " << d4 << endl;
33    return 0;
34 } // end main
```

Demonstrate postfix increment

```
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
```

Outline

fig11_14.cpp

(2 of 2)

# 11.13 Standard Library Class `string`

- **Class built into C++**
  - **Available for anyone to use**
  - **Class `string`**
    - **Similar to our `String` class**
- **Redo our `String` example using `string`**

# 11.13 Standard Library Class `string` (Cont.)

- Class `string`
  - Header `<string>`, namespace `std`
  - Can initialize `string s1( "hi" );`
  - Overloaded << (as in `cout << s1`)
  - Overloaded relational operators
    - ==, !=, >=, >, <=, <
  - Assignment operator =
  - Concatenation (overloaded +=)

# 11.13 Standard Library Class `string` (Cont.)

- Class `string` (Cont.)
  - Substring member function `substr`
    - `s1.substr( 0, 14 );`
      - Starts at location 0, gets 14 characters
    - `s1.substr( 15 );`
      - Substring beginning at location 15, to the end
  - Overloaded `[]`
    - Access one character
    - No range checking (if subscript invalid)
  - Member function `at`
    - Accesses one character
      - `Example`
        - `s1.at( 10 );`
    - Has bounds checking, throws an exception if subscript is invalid
      - Will end program (learn more in Chapter 16)

Outline

fig11_15.cpp

```
1  // Fig. 11.15: fig11_15.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string s1( "happy" );
13    string s2( " birthday" );
14    string s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18       << "\"; s3 is \"" << s3 << '\"'
19       << "\n\nThe results of comparing s2 and s1:"
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       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
26
27    // test string member-function empty
28    cout << "\n\nTesting s3.empty():" << endl;
```

Passing strings to the `string` constructor

Create empty **`string`**

```
29
30    if ( s3.empty() )
31    {
32       cout << "s3 is empty; assigning s1 to s3;" << endl;
33       s3 = s1; // assign s1 to s3
34       cout << "s3 is \"" << s3 << "\"";
35    } // end if
36
37    // test overloaded string concatenation operator
38    cout << "\n\ns1 += s2 yields s1 = ";
39    s1 += s2; // test overloaded concatenation
40    cout << s1;
41
42    // test overloaded string concatenation operator with C-style string
43    cout << "\n\ns1 += \" to you\" yields" << endl;
44    s1 += " to you";
45    cout << "s1 = " << s1 << "\n\n";
46
47    // test string member function substr
48    cout << "The substring of s1 starting at location 0 for\n"
49       << "14 characters, s1.substr(0, 14), is:\n"
50       << s1.substr( 0, 14 ) << "\n\n";
51
52    // test substr "to-end-of-string" option
53    cout << "The substring of s1 starting at\n"
54       << "location 15, s1.substr(15), is:\n"
55       << s1.substr( 15 ) << endl;
```

Member function **empty** tests if the **string** is empty

Member function **substr** obtains a substring from the **string**

**fig11_15.cpp**

(2 of 4)

Outline

fig11_15.cpp

(3 of 4)

```
56
57    // test copy constructor
58    string *s4Ptr = new string( s1 );
59    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61    // test assignment (=) operator with self-assignment
62    cout << "assigning *s4Ptr to *s4Ptr" << endl;
63    *s4Ptr = *s4Ptr;
64    cout << "*s4Ptr = " << *s4Ptr << endl;
65
66    // test destructor
67    delete s4Ptr;
68
69    // test using subscript operator to create lvalue
70    s1[ 0 ] = 'H';
71    s1[ 6 ] = 'B';
72    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73       << s1 << "\n\n";
74
75    // test subscript out of range with string member function "at"
76    cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77    s1.at( 30 ) = 'd'; // ERROR: subscript out of range
78    return 0;
79 } // end main
```

Accessing specific character in **`string`**

Member function **`at`** provides range checking

```
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

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

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = 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:

abnormal program termination
```

Outline

fig11_15.cpp

(4 of 4)

# 11.14 `explicit` Constructors

- **Implicit conversions**
  - **Performed by compiler using single-argument constructors**
  - **Sometimes, implicit conversions are undesirable or error-prone**
    - **Keyword `explicit`**
      - **Suppresses implicit conversions via conversion constructors**

# Common Programming Error 11.9

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

Outline

fig11_16.cpp

(1 of 2)

```
1  // Fig. 11.16: Fig11_16.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    return 0;
17 }  // end main
```

Would logically want this to generate an error

```
18
19 // print Array contents
20 void outputArray( const Array &arrayToOutput )
21 {
22    cout << "The Array received has " << arrayToOutput.getSize()
23       << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray
```

```
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
```

Outline

fig11_16.cpp

(2 of 2)

Outline

Array.h

(1 of 1)

```
1  // Fig. 11.17: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    explicit Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
```

Use **`explicit`** keyword to avoid implicit conversions when inappropriate

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

Outline

Array.h

(2 of 2)

# Common Programming Error 11.10

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

# Common Programming Error 11.11

**Using the `explicit` keyword on data members or member functions other than a single-argument constructor is a compilation error.**

Outline

Fig11_18.cpp

```
1  // Fig. 11.18: Fig11_18.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    outputArray( Array( 3 ) ); // explicit single-argument constructor call
17    return 0;
18 } // end main
```

Using keyword **explicit** on the conversion constructor disallows this line to erroneously call the conversion constructor

An explicit call to the conversion constructor is still allowed

```
19
20 // print array contents
21 void outputArray( const Array &arrayToOutput )
22 {
23    cout << "The Array received has " << arrayToOutput.getSize()
24       << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // end outputArray
```

```
c:\cpphtp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : 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'
```

Outline

Fig11_18.cpp

(2 of 2)

# 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.**