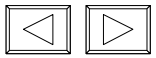
Chapter 8 - Operator Overloading

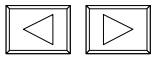
Outline

8.1 Introduction
8.2 Fundamentals of Operator Overloading
8.3 Restrictions on Operator Overloading
8.4 Operator Functions as Class Members vs. as friend Functions
8.5 Overloading Stream-Insertion and Stream-Extraction Operators
8.6 Overloading Unary Operators
8.7 Overloading Binary Operators
8.8 Case Study: An Array Class
8.9 Converting between Types
8.10 Case Study: A `String` Class
8.11 Overloading `++` and `--`
8.12 Case Study: A `Date` Class



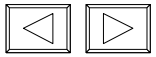
8.1 Introduction

- Operator overloading
  - Enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand
  - Examples of already overloaded operators
    - Operator **<<** is both the stream-insertion operator and the bitwise left-shift operator
    - **+** and **-**, perform arithmetic on multiple types
  - Compiler generates the appropriate code based on the manner in which the operator is used



8.2 Fundamentals of Operator Overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**`+`**)
- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator **`(=)`** or the address operator **`(&)`**
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object



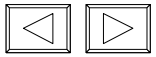
8.3 Restrictions on Operator Overloading

- C++ operators that can be overloaded

Operators that can be overloaded							

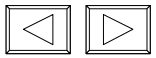
- C++ Operators that cannot be overloaded

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



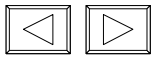
8.3 Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators `&`, `*`, `+` and `-` each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error



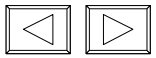
8.4 Operator Functions as Class Members vs. as friend Functions

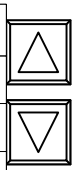
- Member vs non-member
  - Operator functions can be member or non-member functions
  - When overloading `( )`, `[ ]`, `->` or any of the assignment operators, must use a member function
- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
    - If left operand of a different type, operator function must be a non-member function
- Operator functions as non-member functions
  - Must be **`friend`**s if needs to access private or protected members
  - Enable the operator to be commutative



8.5 Overloading Stream-Insertion and Stream-Extraction Operators

- Overloaded **<<** and **>>** operators
  - Overloaded to perform input/output for user-defined types
  - Left operand of types **`ostream &`** and **`istream &`**
  - Must be a non-member function because left operand is not an object of the class
  - Must be a **`friend`** function to access private data members





Outline

1. Class definition

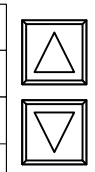
1.1 Function definitions

```
1  // Fig. 8.3: fig08_03.cpp
2  // Overloading the stream-insertion and
3  // stream-extraction operators.
4  #include <iostream>
5
6  using std::cout;
7  using std::cin;
8  using std::endl;
9  using std::ostream;
10 using std::istream;
11
12 #include <iomanip>
13
14 using std::setw;
15
16 class PhoneNumber {
17    friend ostream &operator<<( ostream&, const PhoneNumber & );
18    friend istream &operator>>( istream&, PhoneNumber & );
19
20 private:
21    char areaCode[ 4 ];  // 3-digit area code and null
22    char exchange[ 4 ];  // 3-digit exchange and null
23    char line[ 5 ];      // 4-digit line and null
24 };
25
26 // Overloaded stream-insertion operator (cannot be
27 // a member function if we would like to invoke it with
28 // cout << somePhoneNumber;).
29 ostream &operator<<( ostream &output, const PhoneNumber &num )
30 {
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be **friend** functions.

```
31     output << "(" << num.areaCode << ") "
32            << num.exchange << "-" << num.line;
33     return output;      // enables cout << a << b << c;
34  }
35
36  istream &operator>>( istream &input, PhoneNumber &num )
37  {
38     input.ignore();                      // skip (
39     input >> setw( 4 ) >> num.areaCode; // input area code
40     input.ignore( 2 );                   // skip ) and space
41     input >> setw( 4 ) >> num.exchange; // input exchange
42     input.ignore();                      // skip dash
43     input >> setw( 5 ) >> num.line;      // input line
44     return input;       // enables cin >> a >> b >> c
45  }
46
47  int main()
48  {
49     PhoneNumber phone; // create object phone
50
51     cout << "Enter phone number in the form (123) 456-7890:\n";
52
53     // cin >> phone invokes operator>> function by
54     // issuing the call operator>>( cin, phone ).
55     cin >> phone;
56
57     // cout << phone invokes operator<< function by
58     // issuing the call operator<<( cout, phone ).
59     cout << "The phone number entered was: " << phone << endl;
60     return 0;
61  }
```



Outline

1.1 Function definition

1.2 Initialize variables

2. Get input

bject

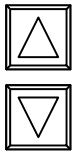
The function call

`cin >> phone;`

interpreted as

`operator>>(cin, phone);`

`input` is an alias for **`cin`**, and **`num`** is an alias for **`phone`**.



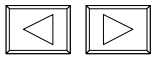
Program Output

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

8.6 Overloading Unary Operators

- Overloading unary operators
  - Can be overloaded with no arguments or one argument
  - Should usually be implemented as member functions
    - Avoid **`friend`** functions and classes because they violate the encapsulation of a class
  - Example declaration as a member function:

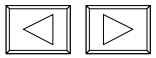
```
class String {
public:
   bool operator!() const;
   ...
};
```

8.6 Overloading Unary Operators

– Example declaration as a non-member function

```
class String {
   friend bool operator!( const String & )
   ...
}
```

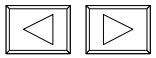



8.7 Overloading Binary Operators

- Overloaded Binary operators
  - Non-static member function, one argument
  - Example:

```
class String {
public:
   const String &operator+=(
                  const String & );
   ...
};
```

  - `y += z` is equivalent to `y.operator+=( z )`

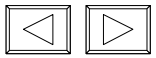


8.7 Overloading Binary Operators

- Non-member function, two arguments
- Example:

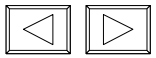
```
class String {
   friend const String &operator+=(
                  String &, const String & );
   ...
};
```

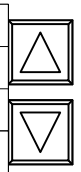
- `y += z` is equivalent to `operator+=( y, z )`



8.8 Case Study: An Array class

- Implement an **`Array`** class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **<<** and **>>**
  - Array comparisons with **==** and **!=**





Outline

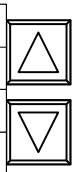
1. Class definition

1.1 Function prototypes

```
1  // Fig. 8.4: array1.h
2  // Simple class Array (for integers)
3  #ifndef ARRAY1_H
4  #define ARRAY1_H
5
6  #include <iostream>
7
8  using std::ostream;
9  using std::istream;
10
11 class Array {
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 constru
17    ~Array();                             // destructor
18    int getSize() const;                  // return size
19    const Array &operator=( const Array & ); // assign arrays
20    bool operator==( const Array & ) const; // compare equal
21
22    // Determine if two arrays are not equal and
23    // return true, otherwise return false (uses operator==).
24    bool operator!=( const Array &right ) const
25       { return ! ( *this == right ); }
26
27    int &operator[]( int );               // subscript operator
28    const int &operator[]( int ) const;  // subscript operator
29    static int getArrayCount();           // Return count of
30                                          // arrays instantiated.
31 private:
32    int size; // size of the array
33    int *ptr; // pointer to first element of array
34    static int arrayCount;  // # of Arrays instantiated
```

Notice all the overloaded operators used to implement the class.

```
35 };
36
37 #endif
38 // Fig 8.4: array1.cpp
39 // Member function definitions for class Array
40 #include <iostream>
41
42 using std::cout;
43 using std::cin;
44 using std::endl;
45
46 #include <iomanip>
47
48 using std::setw;
49
50 #include <cstdlib>
51 #include <cassert>
52 #include "array1.h"
53
54 // Initialize static data member at file scope
55 int Array::arrayCount = 0;   // no objects yet
56
57 // Default constructor for class Array (default size 10)
58 Array::Array( int arraySize )
59 {
60    size = ( arraySize > 0 ? arraySize : 10 );
61    ptr = new int[ size ]; // create space for array
62    assert( ptr != 0 );    // terminate if memory not allocated
63    ++arrayCount;          // count one more object
64
65    for ( int i = 0; i < size; i++ )
66       ptr[ i ] = 0;          // initialize array
```

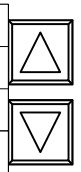


Outline

1. Load header

1.1 Function definitions

1.2 `Array` constructor



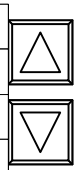
Outline

1.3 `Array` destructor

1.4 `operator=` (assignment)

```
67 }
68
69 // Copy constructor for class Array
70 // must receive a reference to prevent infinite recursion
71 Array::Array( const Array &init ) : size( init.size )
72 {
73    ptr = new int[ size ]; // create space for array
74    assert( ptr != 0 );    // terminate if memory not allocated
75    ++arrayCount;          // count one more object
76
77    for ( int i = 0; i < size; i++ )
78       ptr[ i ] = init.ptr[ i ];  // copy init into object
79 }
80
81 // Destructor for class Array
82 Array::~Array()
83 {
84    delete [] ptr;             // reclaim space for array
85    --arrayCount;              // one fewer object
86 }
87
88 // Get the size of the array
89 int Array::getSize() const { return size; }
90
91 // Overloaded assignment operator
92 // const return avoids: ( a1 = a2 ) = a3
93 const Array &Array::operator=( const Array &right )
94 {
95    if ( &right != this ) {  // check for self-assignment
96
97       // for arrays of different sizes, deallocate original
98       // left side array, then allocate new left side array.
99       if ( size != right.size ) {
100         delete [] ptr;         // reclaim space
```

```
101         size = right.size;     // resize this object
102         ptr = new int[ size ]; // create space for array copy
103         assert( ptr != 0 );    // terminate if not allocated
104      }
105
106      for ( int i = 0; i < size; i++ )
107         ptr[ i ] = right.ptr[ i ];  // copy array into object
108   }
109
110   return *this;   // enables x = y = z;
111}
112
113// Determine if two arrays are equal and
114// return true, otherwise return false.
115bool Array::operator==( const Array &right ) const
116{
117   if ( size != right.size )
118      return false;    // arrays of different sizes
119
120   for ( int i = 0; i < size; i++ )
121      if ( ptr[ i ] != right.ptr[ i ] )
122         return false; // arrays are not equal
123
124   return true;        // arrays are equal
125}
126
127// Overloaded subscript operator for non-const Arrays
128// reference return creates an lvalue
129int &Array::operator[]( int subscript )
130{
131   // check for subscript out of range error
132   assert( 0 <= subscript && subscript < size );
```

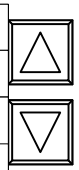


Outline

1.5 `operator==` (equality)

1.6 `operator[]` (subscript for non-`const` arrays)

```
133
134   return ptr[ subscript ]; // reference return
135}
136
137// Overloaded subscript operator for const Arrays
138// const reference return creates an rvalue
139const int &Array::operator[]( int subscript ) const
140{
141   // check for subscript out of range error
142   assert( 0 <= subscript && subscript < size );
143
144   return ptr[ subscript ]; // const reference return
145}
146
147// Return the number of Array objects instantiated
148// static functions cannot be const
149int Array::getArrayCount() { return arrayCount; }
150
151// Overloaded input operator for class Array;
152// inputs values for entire array.
153istream &operator>>( istream &input, Array &a )
154{
155   for ( int i = 0; i < a.size; i++ )
156      input >> a.ptr[ i ];
157
158   return input;   // enables cin >> x >> y;
159}
160
161// Overloaded output operator for class Array
162ostream &operator<<( ostream &output, const Array &a )
163{
```



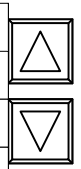
Outline

1.6 `operator[]` (subscript for `const` arrays)

1.7 `getArrayCount`

1.8 `operator>>` (input array)

1.9 `operator<<` (output array)



Outline

1. Load header

```
164   int i;
165
166   for ( i = 0; i < a.size; i++ ) {
167      output << setw( 12 ) << a.ptr[ i ];
168
169      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
170         output << endl;
171   }
172
173   if ( i % 4 != 0 )
174      output << endl;
175
176   return output;   // enables cout << x << y;
177}
178// Fig. 8.4: fig08_04.cpp
179// Driver for simple class Array
180#include <iostream>
181
182using std::cout;
183using std::cin;
184using std::endl;
185
186#include "array1.h"
187
188int main()
189{
190   // no objects yet
191   cout << "# of arrays instantiated = "
192        << Array::getArrayCount() << '\n';
193
```

```
# of arrays instantiated = 0
```

Outline

1.1 Initialize objects

```
// create two arrays and print Array count
Array integers1( 7 ), integers2;
cout << "# of arrays instantiated = "
     << Array::getArrayCount() << "\n\n";

// print integers1 size and contents
cout << "Size of array integers1 is "
     << integers1.getSize()
     << "\nArray after initialization:\n"
     << integers1 << '\n';

// print integers2 size and contents
cout << "Size of array integers2 is "
     << integers2.getSize()
     << "\nArray after initialization:\n"
     << integers2 << '\n';

// input and print integers1 and integers2
cout << "Input 17 integers:\n";
cin >> integers1 >> integers2;
cout << "After input, the arrays contain:\n"
     << "integers1:\n" << integers1
     << "integers2:\n" << integers2 << '\n';

// use overloaded inequality (!=) operator
cout << "Evaluating: integers1 != integers2\n";
if ( integers1 != integers2 )
   cout << "They are not equal\n";

// create array integers3 using integers1 as an
// initializer; print size and contents
Array integers3( integers1 );
```

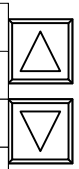
```
# of arrays instantiated = 2
```

```
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
```

```
Input 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
They are not equal
```



Outline

2. Function calls

```
227   cout << "\nSize of array integers3 is "
228        << integers3.getSize()
229        << "\nArray after initialization:\n"
230        << integers3 << '\n';
231
232   // use overloaded assignment (=
233   cout << "Assigning integers2 to
234   integers1 = integers2;
235   cout << "integers1:\n" << integ
236        << "integers2:\n" << integers2 << '\n';
237
238   // use overloaded equality (==) ope
239   cout << "Evaluating: integers1 == i
240   if ( integers1 == integers2 )
241      cout << "They are equal\n\n";
242
243   // use overloaded subscript operato
244   cout << "integers1[5] is " << integ
245
246   // use overloaded subscript operato
247   cout << "Assigning 1000 to integers
248   integers1[ 5 ] = 1000;
249   cout << "integers1:\n" << integers1
250
251   // attempt to use out of range su
252   cout << "Attempt to assign 1000 t
253   integers1[ 15 ] = 1000;  // ERROR
254
255   return 0;
256}
```

```
Size of array integers3 is 7

Array after initialization:
          1          2          3          4
          5          6          7
```

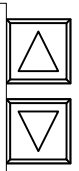
```
Assigning integers2 to integers1:

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

They are equal
```

```
integers1[5] is 13
```

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

Assertion failed: 0 <= subscript && subscript <
size, file Array1.cpp, line 95 abnormal program
termination
```

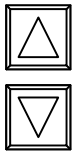
```
As

integers1:
          8          9         10         11
         12       1000         14         15
         16         17
```



Outline

Program Output

```
# of arrays instantiated = 0
# of arrays instantiated = 2

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

Input 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
They are not equal

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



Outline

Program Output

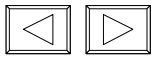
```
Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16       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


Evaluating: integers1 == integers2
They 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]
Assertion failed: 0 <= subscript && subscript < size, file Array1.cpp,
line 95 abnormal program termination
```

8.9 Converting between Types

- Cast operator
  - Forces conversions among built-in types
  - Specifies conversions between user defined and built-in types
  - Conversion operator must be a non-**`static`** member function
  - Cannot be a **`friend`** function
  - Do not specify return type
    - Return type is the type to which the object is being converted
  - For user-defined class **`A`**

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

    - Declares an overloaded cast operator function for creating a **`char *`** out of an **`A`** object



8.9 Converting between Types

```
A::operator int() const;
```

- Declares an overloaded cast operator function for converting an object of **A** into an integer

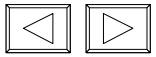
```
A::operator otherClass() const;
```

- Declares an overloaded cast operator function for converting an object of **A** into an object of **otherClass**

- Compiler and casting
  - Casting can prevent the need for overloading
  - If an object s of user-defined class String appears in a program where an ordinary **char *** is expected, such as

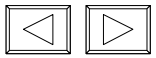
```
cout << s;
```

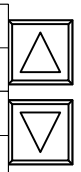
The compiler calls the overloaded cast operator function **operator char *** to convert the object into a **char *** and uses the resulting **char *** in the expression



8.10 Case Study: A `String` Class

- Build a class to handle strings
  - Class **`string`** in standard library (more Chapter 19)
- Conversion constructor
  - Single-argument constructors that turn objects of other types into class objects



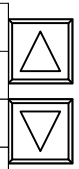


Outline

1. Class definition

1.1 Member functions, some definitions

```
 1 // Fig. 8.5: string1.h
 2 // Definition of a String class
 3 #ifndef STRING1_H
 4 #define STRING1_H
 5
 6 #include <iostream>
 7
 8 using std::ostream;
 9 using std::istream;
10
11 class String {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14
15 public:
16    String( const char * = "" ); // conversion/default ctor
17    String( const String & );    // copy constructor
18    ~String();                   // destructor
19    const String &operator=( const String & );  // assignment
20    const String &operator+=( const String & ); // concatenation
21    bool operator!() const;                     // is String empty?
22    bool operator==( const String & ) const; // test s1 == s2
23    bool operator<( const String & ) const;  // test s1 < s2
24
25    // test s1 != s2
26    bool operator!=( const String & right ) const
27       { return !( *this == right ); }
28
29    // test s1 > s2
30    bool operator>( const String &right ) const
31       { return right < *this; }
32
33    // test s1 <= s2
```



Outline

1.2 Member variables

```
34    bool operator<=( const String &right ) const
35       { return !( right < *this ); }
36
37    // test s1 >= s2
38    bool operator>=( const String &right ) const
39       { return !( *this < right ); }
40
41    char &operator[]( int );              // subscript operator
42    const char &operator[]( int ) const; // subscript operator
43    String operator()( int, int );        // return a substring
44    int getLength() const;                // return string length
45
46 private:
47    int length;                      // string length
48    char *sPtr;                      // pointer to start of string
49
50    void setString( const char * );  // utility function
51 };
52
53 #endif
54 // Fig. 8.5: string1.cpp
55 // Member function definitions for class String
56 #include <iostream>
57
58 using std::cout;
59 using std::endl;
60
61 #include <iomanip>
62
63 using std::setw;
64
```

```
65 #include <cstring>
66 #include <cassert>
67 #include "string1.h"
68
69 // Conversion constructor: Convert char * to String
70 String::String( const char *s ) : length( strlen( s ) )
71 {
72    cout << "Conversion constructor: " << s << '\n';
73    setString( s );       // call utility function
74 }
75
76 // Copy constructor
77 String::String( const String &copy ) : length( copy.length )
78 {
79    cout << "Copy constructor: " << copy.sPtr << '\n';
80    setString( copy.sPtr ); // call utility function
81 }
82
83 // Destructor
84 String::~String()
85 {
86    cout << "Destructor: " << sPtr << '\n';
87    delete [] sPtr;          // reclaim string
88 }
89
90 // Overloaded = operator; avoids self assignment
91 const String &String::operator=( const String &right )
92 {
93    cout << "operator= called\n";
94
95    if ( &right != this ) {          // avoid self assignment
```

Conversion constructor: **`char *`** to **`String`**.

Constructors and destructors will print when called.

Outline

1. Load header

1.1 Function definitions

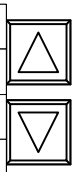
1.2 Conversion constructor

1.3 Copy constructor

1.4 Destructor

1.5 `operator=`

ent)



Outline

1.6 operator+= (concatenation)

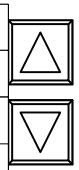
1.7 operator! (string empty?)

1.8 operator== (equality)

```
96       delete [] sPtr;               // prevents memory leak
97       length = right.length;        // new String length
98       setString( right.sPtr );      // call utility function
99    }
100   else
101      cout << "Attempted assignment of a String to itself\n";
102
103   return *this;   // enables cascaded assignments
104}
105
106// Concatenate right operand to this object and
107// store in this object.
108const String &String::operator+=( const String &right )
109{
110   char *tempPtr = sPtr;         // hold to be able to delete
111   length += right.length;       // new String length
112   sPtr = new char[ length + 1 ]; // create space
113   assert( sPtr != 0 );   // terminate if memory not allocated
114   strcpy( sPtr, tempPtr );      // left part of new String
115   strcat( sPtr, right.sPtr );  // right part of new String
116   delete [] tempPtr;            // reclaim old space
117   return *this;                 // enables cascaded calls
118}
119
120// Is this String empty?
121bool String::operator!() const { return length == 0; }
122
123// Is this String equal to right String?
124bool String::operator==( const String &right ) const
125   { return strcmp( sPtr, right.sPtr ) == 0; }
126
127// Is this String less than right String?
```

```
128 bool String::operator<( const String &right ) const
129    { return strcmp( sPtr, right.sPtr ) < 0; }
130
131 // Return a reference to a character in a String as an lvalue.
132 char &String::operator[]( int subscript )
133 {
134    // First test for subscript out of range
135    assert( subscript >= 0 && subscript < length );
136
137    return sPtr[ subscript ];  // creates lvalue
138 }
139
140 // Return a reference to a character in a String as an rvalue.
141 const char &String::operator[]( int subscript ) const
142 {
143    // First test for subscript out of range
144    assert( subscript >= 0 && subscript < length );
145
146    return sPtr[ subscript ];  // crea
147 }
148
149 // Return a substring beginning at index and
150 // of length subLength
151 String String::operator()( int index, int subLength )
152 {
153    // ensure index is in range and substring length >= 0
154    assert( index >= 0 && index < length && subLength >= 0 );
155
156    // determine length of substring
157    int len;
158
```

Notice the overloaded function call operator.



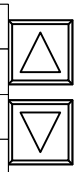
Outline

1.9 `operator<` (less than)

1.10 `operator[]` (subscript)

1.11 `operator[]` (`const` subscript)

1.12 `operator()` (return substring)



Outline

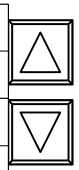
1.13 getLength

1.14 setString

```
159   if ( ( subLength == 0 ) || ( index + subLength > length ) )
160      len = length - index;
161   else
162      len = subLength;
163
164   // allocate temporary array for substring and
165   // terminating null character
166   char *tempPtr = new char[ len + 1 ];
167   assert( tempPtr != 0 ); // ensure space allocated
168
169   // copy substring into char array and terminate string
170   strncpy( tempPtr, &sPtr[ index ], len );
171   tempPtr[ len ] = '\0';
172
173   // Create temporary String object containing the substring
174   String tempString( tempPtr );
175   delete [] tempPtr;  // delete the temporary array
176
177   return tempString;  // return copy of the temporary String
178}
179
180// Return string length
181int String::getLength() const { return length; }
182
183// Utility function to be called by constructors and
184// assignment operator.
185void String::setString( const char *string2 )
186{
187   sPtr = new char[ length + 1 ]; // allocate storage
188   assert( sPtr != 0 );  // terminate if memory not allocated
189   strcpy( sPtr, string2 );       // copy literal to object
190}
```

```
191
192// Overloaded output operator
193ostream &operator<<( ostream &output, const String &s )
194{
195   output << s.sPtr;
196   return output;   // enables cascading
197}
198
199// Overloaded input operator
200istream &operator>>( istream &input, String &s )
201{
202   char temp[ 100 ];   // buffer to store input
203
204   input >> setw( 100 ) >> temp;
205   s = temp;         // use String class assignment operator
206   return input;     // enables cascading
207}
208// Fig. 8.5: fig08_05.cpp
209// Driver for class String
210#include <iostream>
211
212using std::cout;
213using std::endl;
214
215#include "string1.h"
216
217int main()
218{
219   String s1( "happy" ), s2( " birthday" ), s3;
220
```

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
```



Outline

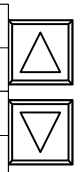
1.15 `operator<<` (output `String`)

1.16 `operator>>` (input `String`)

----

1. Load header

1.1 Initialize objects



Outline

2. Function calls

```
221   // test overloaded equality and relational operators
222   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
223        << "\"; s3 is \"" << s3 << '\"'
224        << "\nThe results of comparing s2 and s1:"
225        << "\ns2 == s1 yields "
226        << ( s2 == s1 ? "true" : "false" )
227        << "\ns2 != s1 yields "
228        << ( s2 != s1 ? "true" : "false" )
229        << "\ns2 >  s1 yields "
230        << ( s2 > s1 ? "true" : "false" )
231        << "\ns2 <  s1 yields "
232        << ( s2 < s1 ? "true" : "false" )
233        << "\ns2 >= s1 yields "
234        << ( s2 >= s1 ? "true" : "false" )
235        << "\ns2 <= s1 yields "
236        << ( s2 <= s1 ? "true" : "false" );
237
238   // test overloaded String empty (!) operator
239   cout << "\n\nTesting !s3:\n";
240   if ( !s3 ) {
241      cout << "s3 is empty; assigning s1 to s3;\n";
242      s3 = s1;                // test overloaded assignm
243      cout << "s3 is \"" << s3 << "\"";
244   }
245
246   // test overloaded String concatenation operator
247   cout << "\n\ns1 += s2 yields s1 = ";
248   s1 += s2;                  // test overloaded
249   cout << s1;
250
251   // test conversion constructor
252   cout << "\n\ns1 += \" to you\" yields\n";
253   s1 += " to you";           // test conversion 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 constructor:  to you
Destructor:  to you
```

Outline

2. Function calls

```
254   cout << "s1 = " << s1 << "\n\n";
255
256   // test overloaded function call operator () for substring
257   cout << "The substring of s1 starting at\n"
258        << "location 0 for 14 characters, s1(0, 14), is:\n"
259        << s1( 0, 14 ) << "\n\n";
260
261   // test substring "to-end-of-String" opti
262   cout << "The substring of s1 starting at\
263        << "location 15, s1(15, 0), is: "
264        << s1( 15, 0 ) << "\n\n";  // 0 is "
265
266   // test copy constructor
267   String *s4Ptr = new String( s1 );
268   cout << "*s4Ptr = " << *s4Ptr << "\n\n";
269
270   // test assignment (=) operator with sel
271   cout << "assigning *s4Ptr to *s4Ptr\n";
272   *s4Ptr = *s4Ptr;          // test overlo
273   cout << "*s4Ptr = " << *s4Ptr << '\n';
274
275   // test destructor
276   delete s4Ptr;
277
278   // test using subscript operator to create lvalue
279   s1[ 0 ] = 'H';
280   s1[ 6 ] = 'B';
281   cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
282        << s1 << "\n\n";
283
```

```
s1 = happy birthday to you
```

```
Conversion 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:
```

```
Destructor: happy birthday
```

```
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
```

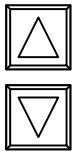
Outline

```
284   // test subscript out of range
285   cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
286   s1[ 30 ] = 'd';      // ERROR: subscript out of range
287
288   return 0;
289}
```

```
Attempt to assign 'd' to s1[30] yields:


Assertion failed: subscript >= 0 && subscript <
length, file string1.cpp, line 82


Abnormal program termination
```

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion 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 constructor:  to you

Destructor:  to you

s1 = happy birthday to you
```



Outline

Program Output

```
Conversion 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

Destructor: happy birthday
Conversion constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15, 0), 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:

Assertion failed: subscript >= 0 && subscript < length, file
string1.cpp, line 82

Abnormal program termination
```

8.11 Overloading ++ and --

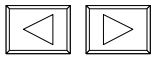
- Pre/post incrementing/decrementing operators
  - Allowed to be overloaded
  - Distinguishing between pre and post operators
    - prefix versions are overloaded the same as other prefix unary operators

```
d1.operator++();      // for ++d1
```

    - convention adopted that when compiler sees postincrementing expression, it will generate the member-function call

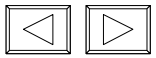
```
d1.operator++( 0 );   // for d1++
```

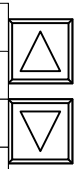
    - `0` is a dummy value to make the argument list of `operator++` distinguishable from the argument list for `++operator`



8.12 Case Study: A `Date` Class

- The following example creates a Date class with
  - An overloaded increment operator to change the day, month and year
  - An overloaded **+=** operator
  - A function to test for leap years
  - A function to determine if a day is last day of a month





Outline

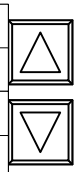
1. Class definition

1.1 Member functions

1.2 Member variables

```
 1 // Fig. 8.6: date1.h
 2 // Definition of class Date
 3 #ifndef DATE1_H
 4 #define DATE1_H
 5 #include <iostream>
 6
 7 using std::ostream;
 8
 9 class Date {
10    friend ostream &operator<<( ostream &, const Date & );
11
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // constructor
14    void setDate( int, int, int ); // set the date
15    Date &operator++();            // preincrement operator
16    Date operator++( int );        // postincrement operator
17    const Date &operator+=( int ); // add days, modify object
18    bool leapYear( int ) const;    // is this a leap year?
19    bool endOfMonth( int ) const;  // is this end of month?
20
21 private:
22    int month;
23    int day;
24    int year;
25
26    static const int days[];       // array of days per month
27    void helpIncrement();          // utility function
28 };
29
30 #endif
```

```
31 // Fig. 8.6: date1.cpp
32 // Member function definitions for Date class
33 #include <iostream>
34 #include "date1.h"
35
36 // Initialize static member at file scope;
37 // one class-wide copy.
38 const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
39                            31, 31, 30, 31, 30, 31 };
40
41 // Date constructor
42 Date::Date( int m, int d, int y ) { setDate( m, d, y ); }
43
44 // Set the date
45 void Date::setDate( int mm, int dd, int yy )
46 {
47    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
48    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
49
50    // test for a leap year
51    if ( month == 2 && leapYear( year ) )
52       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
53    else
54       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
55 }
56
57 // Preincrement operator overloaded as a member function.
58 Date &Date::operator++()
59 {
60    helpIncrement();
61    return *this;  // reference return to create an lvalue
62 }
63
```



Outline

1. Load header

1.1 Define `days[]`

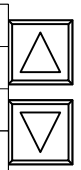
1.2 Function definitions

1.3 Constructor

1.4 `operator++` (preincrement)

```
64 // Postincrement operator overloaded as a member function.
65 // Note that the dummy integer parameter does not have a
66 // parameter name.
67 Date Date::operator++( int )
68 {
69    Date temp = *this;
70    helpIncrement();
71
72    // return non-incremented, saved, temporary object
73    return temp;   // value return; not a reference return
74 }
75
76 // Add a specific number of days to a date
77 const Date &Date::operator+=( int additionalDays )
78 {
79    for ( int i = 0; i < additionalDays; i++ )
80       helpIncrement();
81
82    return *this;    // enables cascading
83 }
84
85 // If the year is a leap year, return true;
86 // otherwise, return false
87 bool Date::leapYear( int y ) const
88 {
89    if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
90       return true;   // a leap year
91    else
92       return false;  // not a leap year
93 }
94
95 // Determine if the day is the end of the month
96 bool Date::endOfMonth( int d ) const
97 {
```

postincrement operator has a dummy **int** value.



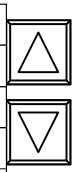
Outline

1.5 `operator++(int)` (postincrement)

1.6 `operator+=`

1.7 `leapYear`

1.8 `endOfMonth`



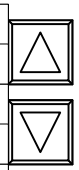
Outline

1.9 `helpIncrement`

1.10 `operator<<` (output `Date`)

```
98    if ( month == 2 && leapYear( year ) )
99       return d == 29; // last day of Feb. in leap year
100   else
101      return d == days[ month ];
102}
103
104// Function to help increment the date
105void Date::helpIncrement()
106{
107   if ( endOfMonth( day ) && month == 12 ) {  // end year
108      day = 1;
109      month = 1;
110      ++year;
111   }
112   else if ( endOfMonth( day ) ) {              // end month
113      day = 1;
114      ++month;
115   }
116   else        // not end of month or year; increment day
117      ++day;
118}
119
120// Overloaded output operator
121ostream &operator<<( ostream &output, const Date &d )
122{
123   static char *monthName[ 13 ] = { "", "January",
124      "February", "March", "April", "May", "June",
125      "July", "August", "September", "October",
126      "November", "December" };
127
128   output << monthName[ d.month ] << ' '
129          << d.day << ", " << d.year;
130
131   return output;   // enables cascading
132}
```

```
133// Fig. 8.6: fig08_06.cpp
134// Driver for class Date
135#include <iostream>
136
137using std::cout;
138using std::endl;
139
140#include "date1.h"
141
142int main()
143{
144   Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
145   cout << "d1 is " << d1
146        << "\nd2 is " << d2
147        << "\nd3 is " << d3 << "\n\n";
148
149   cout << "d2 += 7 is " << ( d2 += 7 ) << "\n\n";
150
151   d3.setDate( 2, 28, 1992 );
152   cout << "  d3 is " << d3;
153   cout << "\n++d3 is " << ++d3 << "\n\n";
154
155   Date d4( 3, 18, 1969 );
156
157   cout << "Testing the preincrement operator:\n"
158        << "  d4 is " << d4 << '\n';
159   cout << "++d4 is " << ++d4 << '\n';
160   cout << "  d4 is " << d4 << "\n\n";
161
162   cout << "Testing the postincrement operator:\n"
163        << "  d4 is " << d4 << '\n';
164   cout << "d4++ is " << d4++ << '\n';
165   cout << "  d4 is " << d4 << endl;
166
167   return 0;
168}
```



Outline

1. Load header

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

objects

2. Function calls

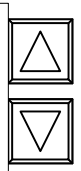
3. Print results

```
d2 += 7 is January 3, 1993
```

```
 d3 is February 28, 1992
++d3 is February 29, 1992
```

```
Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969
```

```
Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969
```



Outline

Program Output

```
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

Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```